Université de Montréal
Faculté des Arts et Sciences

Département d'Informatique et de Recherche Opérationnelle

Laboratoire de Génie Logiciel

Présentation du projet IFT3051 sur

Comparaison SOUL et Ptidej

Session d'hiver 2007

Remis au professeur

M. Yann-Gaël Guéhéneuc

Travail réalisé par

Jad Karam

Table des matières

1 – Introduction

Le projet Ptidej est un outil de rétro-conception. Il permet, entre autre, de détecter des patrons de mauvaise conception et de proposer des solutions. Cet outils agit sur des programmes quelconques écrits en Java ou en C++ afin d'améliorer leurs structures en proposant des corrections pour des classes qui présentent des erreurs de conception.
Le but de ce travail consiste à comparer deux environnements de travail afin de souligner leurs différences. Tout d'abord, il m'est demandé de me familiariser avec les environnements Ptidej et SOUL. Par la suite, je dois relever les différences quant à la présence et l'utilisation des anti patrons de conception. M'inspirant de SOUL (Smalltalk Open Unification Language), un outil similaire à Ptidej, j'ai proposé un ensemble d'algorithmes qui pourrait être rajouté dans le package SAD (Software Architectural Defect) de Ptidej pour la détection de défauts. Cet ensemble est formé d'algorithmes présents dans SOUL que j'ai transcrits. Un travail précédent de recherche en maîtrise sur SOUL m'a généreusement aidé afin de créer la liste proposée plus bas. Cette liste d'algorithmes de détection couvre plusieurs d'anti-patrons. Nous avons élaboré ces algorithmes pour s'attaquer à la détection de défauts de conception rencontrés notamment dans les grandes classes, les longues méthodes, les longues listes de paramètres, les blocs de donnée, les interfaces inappropriées, le Feature Envy, le Middle Man, les classes paresseuses et les méthodes abstraites non implémentées.

2 – Description du Projet

2.1 Ptidej

a) Description

La suite d'outils Ptidej est composée de plusieurs modules auxquels nous fournissons un ensemble de classes pour y appliquer différents algoritmes de détection de défauts. Ptidej construit des diagrammes de classes par exemple par rétro-conception de la structure du modèle fourni. Ensuite une détection d'anti-patrons de conception pourra être enclenchée. Les anti-patrons de conception se retrouvent dans des ensembles de classes qui présentent une mauvaise structure.

b) Algorithme dans Ptidej

Les algorithmes qui nous intéressent sont ceux qui forment le module SAD. Ce module est spécialisé dans la détection d'anti-patrons de conception. On retrouve notamment dans ce module des algorithmes qui s'occupent de retrouver des anti-patrons bien définis. Comme le « Blob », le « Swiss Army Knife », le « Spaghetti Code » ou le « Functionnal Decomposition ». Chacun de ces algorithmes vérifie une sorte de mauvaise conception que l'on formalise à l'aide de contraintes qui lui sont spécifiques. Par exemple, l'anti-patron « Blob » sert à retrouver dans un ensemble des classes celles qui sont des classes de contrôle, celles qui sont longues, celles qui sont des classes de données, et celles qui représentent des faibles cohésions.

2.2 SOUL

a) Description

Le méta-langage logique Smalltalk Open Unification Language, dont l'acronyme est SOUL, est la suite d'outil de laquelle je me suis inspiré. Cet environnement est basé sur la programmation logique, secondé par des classes écrites en Smalltalk. Similaire à la syntaxe de Prolog, SOUL est un langage déclaratif de méta-programmation qui sert à écrire et vérifier des prédicats logiques sur du code Smalltalk. Par exemple, avec SOUL, il est possible de détecter des déficiences de conception dans des modèles et des ensembles de classes. Ainsi les programmes écrits sur SOUL détectent des anti-patrons de conception dans des programmes orientés-objet écrits en Java, Smalltalk ou en Object-C par exemple. En installant VisualWorks sur ma machine et en important toutes les bibliothèques nécessaires pour installer SOUL, j'ai eu accès à tous ses algorithmes.

b) Algorithmes dans SOUL

Les bibliothèques téléchargées contiennent des classes écrites en Smalltalk. Pour accéder aux composantes de celles-ci, nous devons définir des prédicats en Prolog. Ces instructions unifieront des classes, méthodes ou cariables à des variables que l'on déclare dans ces prédicats. Ainsi en unifiant les variables aux résultats générés par les algorithmes en Smalltalk, nous obtenons toutes les valeurs possibles. En utilisants des prédicats les programmeurs SOUL détectent les défaillances dans les programmes fournis. Donc en combinant des prédicats spécifiques à chacun des anti-patrons, et en se basant sur Smalltalk, les programmeurs trouvent les erreurs de conception

3 - Comparaison des suites Ptidej et SOUL

3.1 Tableau de comparaison des suites

Le tableau suivant est une comparaison entre les classes qui forment la suite d'outils Ptidej et les algorithmes présents dans la suite SOUL. La colonne de gauche énumère tous les anti-patrons développés dans SOUL. Ce tableau nous indique si ces anti-patrons sont implémentés dans Ptidej. Dans le cas contraire, nous indiquons s'il sera possible de les implémenter ou s'il nous manque des ressources pour cette fin.

Anti-Patrons présents dans SOUL	**Algorithmes implémentés dans Ptidej**		
	Présents	**Possible**	**Manque**
Grandes Classes		**(1)**	
Nombreuses instances		√	
Nombreuses méthodes	√		
Longues Méthodes		**(2)**	
Nombreuses instructions		√	
Nombreuses variables temporaires		√	
Longues Liste de paramètres (Spaghetti Code)			
Nombreux paramètres	√		
Redondance de paramètres		√	

Code Dupliqué

Code Dupliqué			
Nombres magiques			√
Code dupliqué dans méthodes			√

Données

Classes de données	√		
Blocs de donnés (data clumps) Ensemble de paramètres sans logique apparente		√	

Interfaces

Interface ou héritage refusé Interface Refusée Héritage Refusé			**(3)** √ √
Interface inappropriée		√ **(4)**	
Méthode abstraite non implémentées		√ **(5)**	

Responsabilité

Méthodes accédant à des attributs (ou donnés) d'autres classes (Feature Envy)		√ **(6)**	
Chaines de messages (références)			√
Classe déléguée (Middle Man)		√	

Code inutile

Classes paresseuses		√ **(7)**	
Paramètres non utilisés			√
Instance non utilisées			√

Logique Conditionnelle

Switch sur un paramètre			√
Switch sur condition nulle			√

Tableau 3.0 : Comparaison entre les anti-patrons SOUL à ceux de Ptidej.

3.2 Commentaires Supplémentaires

(1) Dans SAD une méthode vérifie s'il existe un grand nombre d'attributs, donc implicitement un certain nombre de variables d'instances.

(2) La classe qui s'occupe de détecter les longues méthodes ne se sert que des métriques afin de détecter les méthodes qui sont longues. Mais dans le cas de SOUL, ils vérifient si le nombre d'instructions (statements) dans une méthode est élevé, et ou s'il y a de nombreuses variables temporaires. Nous devons songer à utiliser une grammaire pour délimiter une instruction.

(3) Dans SAD, la décomposition fonctionnelle détecte la présence ou l'absence d'interfaces ou d'héritage. Cependant, dans un premier cas SOUL détecte si la classe en question respecte un héritage cohérent, et dans un deuxième cas si toutes les méthodes de l'interface sont implémentées sans que, dans les deux cas, il n'y ait de refus d'implémentation de la part de la sous-classe. Cette fonctionnalité est propre à Smalltalk. Par ailleurs, dans le paquet codesmell.detection.SpaghettiCode une autre méthode vérifie l'héritage et calcule sa profondeur.

(4) Ptidej vérifie si une classe impose des implémentations multiples d'interfaces. Tandis que SOUL vérifie si une classe implémente correctement une interface donnée ou bien si elle implémente des méthodes provenant de plusieurs interfaces. Il me semble que la réunion de ces recherches sera un bon algorithme de détection.

(5) On pourra, à l'aide du polymorphisme, détecter les corps des fonctions et voir lesquelles sont implémentées et lesquelles ne le sont pas avec les mêmes outils de recherche présents dans les classes de SAD.

(6) Le Feature Envy n'est pas implémenté dans Ptidej. Il s'agit de vérifier si certaines méthodes accèdent à des attributs ou donnés d'autres classes. Ainsi, ces méthodes devraient être déplacées dans les classes qui sont appelées le plus souvent.

(7) Les classes paresseuses sont celles qui ont été démunies de leur fonctionnalités ou bien celles que l'on a réduites en taille et en fonction après une restructuration du code (refactoring).

4 – Proposition d'algorithmes de détection

En nous basant sur le tableau du chapitre 3, énonçant les différents anti-patrons, nous allons décrire ces algorithmes de détection. Ceux-ci sont prévus afin de détecter les erreurs dans la conception de programmes qui nous sont fournis. Pour chacun des algorithmes possiblement implémentables dans la suite Ptidej, nous allons présenter son pseudo-code pour décrire les instructions afin que cette détection soit réalisable.

Pour parcourir le code à analyser fournis en entrée, nous nous sommes basés directement sur la grammaire de Java 1.5. Nous savons que ce langage a une syntaxe qui lui est bien spécifique et qui est décrite dans cette grammaire. En utilisant le Java Tree Builder (JTB) j'ai pu construire un arbre de syntaxe abstraite qui décomposera le code en nœuds. Chaque nœud représente une composante spécifique du code. En générant un compilateur de cette grammaire utilisant le JavaCC (Java Compiler Compiler), nous créons ainsi un analyseur syntaxique de cette grammaire et vérifions le code que l'on nous fournis en entrée. Une fois cet analyseur syntaxique généré, nous pouvons aussi implanter un visiteur pour personnaliser la recherche de nœuds, ainsi que les actions à réaliser.

Tout d'abord, pour créer l'arbre de syntaxe abstraite (AST) nous prenons une grammaire et lançons la commande:

% jtb javaXX.jj

À ce niveua nous obtenons une grammaire structurée en arbre. Cette grammaire est par la suite compilée à l'aide de javaCC. On lance la commande:

%javacc jtb.out.jj

Cette commande nous permettra de créer les visiteurs ainsi que tous arbres de syntaxe pour naviguer dans les classes que l'on voudra analyser. Dans ce chapitre nous allons montrer différentes techniques pour opérer sur ces classes.

Un exemple de traitement de classes que l'on reçoit en entrée est joint en annexe. Il couvre en détails la création de visiteurs, ainsi que l'initialisation d'analyseurs syntaxiques et les façons de l'utiliser. Dans cet exemple nous nous sommes attaqués au décompte du nombre de variables d'instances, le nombre d'instructions ainsi que la détection de paramètres redondants. La méthode « main » est donc détaillée et explique comment visiter les composantes d'un code.

4.1 Grandes Classes

a) Nombreuses instances

Nous cherchons à nous assurer d'avoir un nombre raisonnable de variables d'instances. Ainsi, nous devons évaluer la métrique qui calcule le nombre de variables d'instances. Ces variables se trouvent dans toute classe, leurs parents et leurs enfants. Nous devons alors compter le nombre de variables d'instance dans toute la hiérarchie et analyser ce nombre. Si nous retrouvons une abondance de variables d'instance, nous devrons signaler une déficience au niveau de la conception. Je propose deux façons d'aborder ce problème. Premièrement en utilisant la métrique NAD, nous obtenons le nombre d'attributs de la classe courante puis nous évaluons le nombre des variables des classes dans sa hiérarchie. Une deuxième proposition sera de comparer si les variables d'instances de la classe courante ressemblent à celles de la classe parente et par récurrence visiter tous les parents. L'algorithme qui s'occupe de la détection est le suivant :

Première Proposition :

```
Algorithme Nombreuses Instances :
Entrée : Un ensemble non-vide de classes.
Sortie : Le nombre de variables d'instances.
Pour chaque classe
{
      Méthode NombreInstances (Classe classe, Int NbrInstance)
      {
            CurrentInstVar ←  Évaluer NAD
            NbrInstance ← CurrentInstVar + NbrInstance.
            Si (Classe Super existe) alors
                  NombreInstances (classe super, NbrInstance)
            Finsi
            Return NbrInstance /*nombre de variables d'instances*/
      }
 }
```

Une fois que l'on récupère le nombre d'instances, nous devons calculer le rapport du nombre de variables d'instance. Ce rapport sera donc assujetti au nombre

d'instances que l'on récupère divisé par le nombre de classes dans l'ensemble que l'on analyse. À ce niveau, un rapport critique devra entrer en vigueur pour faire une synthèse des résultats.

Deuxième proposition :

```
Algorithme Nombreuses Instances :
Entrée : Un ensemble non-vide de classes
Sortie : Le nombre de similitudes: variables d'instances
Pour chaque classe
{
      Méthode InstancesSimilaires (Classe classe, Liste Instances)
      {
            HashMap Instances = new HashMap(Key clé,
                                            Value occurrence)
            /*
            Clé est une instance où chaque instance est formée
            d'un modificateur (visibilité), un type et un nom.
            Occurrence est une valeur qui sera incrémentée à
            chaque fois que l'on rencontre une clé similaire
            */
            Instances ← ajouter toute instance
                                  non-statiques de la classe

            Si (Classe Super existe) alors
                  InstancesSimilaires (super,  Instances)
            FinSi

            /*
            En itérant sur la HashMap, on récupère la liste des
            instances qui similaires.
            */
            Tant qu'il reste des éléments dans Instances
                  Si (occurrences > X) // X valeur seuil de
            redondance
                        Similitudes ← Clé
                  FinSi
            Fin TantQue
            Retourner similitudes.
      }

}
```

Un autre moyen de parcourir le code à la recherche de nœuds qui représentent des instances de variables est en créant un visiteur spécifique. Comme mentionné au tout début de ce chapitre, nous construisons un arbre de syntaxe abstraite qui sera

secondé par son analyseur syntaxique en code Java. Cet analyseur sera dépourvu de sa méthode principale « Main ». Cette méthode sera crée dans un fichier à part, et devra instancier un visiteur qui lui reconnaitra les variables d'instance.

```
package visitor;
import syntaxtree.*;

/*
 * Cette classe sert a parcourir le code recu en
 * entree.  Ainsi on specialise ce visiteur a ne
 * parcourir que des noeuds specifiques.  Nous
 * rappelons que le (JTB) nous a servi a l'aide
 * d'une grammaire de construire les noeuds que
 * l'on visite et qui contiennent l'information
 * contenue dans la classe a analyser.  Alors
 * la specialisation se limite aux noeuds de
 * type VariableDeclarator.  A chaque fois que
 * l'on rencontre un noeud de ce type on incremente
 * le compteur du nombre de variables d'instances
 * presentes dans la classe a analyser.
 *
 * Auteur  : Jad Karam
 * Date    : Hiver 2007
 * Version : 1.0
 *
 */
public class VarInstVisiteur extends DepthFirstVisitor
{
    int instVar;
    public VarInstVisiteur(int varInst)
    {this.instVar = varInst;}

      /* On visite le noeud VariableDeclarator et increment
       le compteur a chaque fois qu'on le rencontre */
    public void visit(VariableDeclarator n)
    {
        n.f0.accept(this);
        n.f1.accept(this);
      instVar = instVar+1;

    }
    public int getInstVar()
    {
        return instVar;
    }
}
```

4.2 Longues méthodes

a) Nombreuses instructions

En un premier temps, nous implémentons la caractéristique des « Nombreuses instructions »:

Dans cette partie nous allons identifier le nombre d'instructions.

D'une autre part, nous allons travailler pour délimiter une instruction, dans le sens que si un code est écrit sur une même ligne, la métrique ne soit pas erronée. JavaCC détecte les instructions et nous permet de compter leur nombre.

```
Algorithme nombreuses instructions :
Entrée : Un ensemble non-vide de classes.
Sortie : NbrInstr : le nombre d'instructions.
Pour chaque classe
{
      Pour chaque méthode
      {
            //Métrique utilisée NOS : Number Of Statements
            Évaluer « NOS »
      }
}
```

La métrique NOS sert a compter toutes les instructions, buts confondus.

Dans une condition, nous pouvons avoir plusieurs instructions. Celles-ci peuvent être des affectations, des comparaisons, des lectures de fichiers et ainsi de suite. Donc, une condition peut ne pas être une simple instruction. Des appels récursifs peuvent avoir lieux et leur traitement sera décrit plus bas.

Cette métrique est bien particulière car qu'elle fait appel directement à la grammaire de Java. En parcourant le code reçu en entrée, nous pouvons construire un arbre de syntaxe abstraite qui représente la structure de la classe et, plus précisément le nombre d'instructions dans le code.

À ce niveau, nous devons utiliser le générateur d'arbres de syntaxe abstraite (Java Tree Builder) ainsi que le compilateur Java Compiler Compiler (JavaCC). Pour pouvoir naviguer dans l'arbre construit, nous allons créer un visiteur spécialisé pour recueillir les informations sur chacun des nœuds de l'arbre et traiter celles qui nous intéresse. Ce visiteur est :

Algorithme de recherche de nombre d'instructions :

Entrée: Ensemble non-vide de classes

Sortie : nbrStat le nombre de variables temporaires

```
/* Cette classe sert d'un visiteur.  Ce visiteur
 * a pour fonction de compter le nombre d'instructions
 * dans une classe donnee.  A chaque noeud statements par
 * lequel passe notre analyseur syntaxique en parcourant le
 * code, nous incrementons un compteur de nombre d'instructions.
 * A la fin de l'analyse de la classe nous retournons le nombre
 * d'instructions parcourues dans le code.
 *
 * Auteur  : Jad Karam
 * Date    : Hiver 2007
 * Version : 1.0
 */
package visitor;
import syntaxtree.*;


public class StatementVisitor extends DepthFirstVisitor
{
    int nbrStat;
```

```
/*
  Nous n'avons pas besoin de preciser le chemin complet pour
  l'analyseur a cause que l'on ne filtre pas les donnees.
  Nous ajoutons ce visiteur tout simplement afin que, une fois
  que l'on rencontre une variable, on incremente le compteur du
      nombre d'instructions dans le code
 */

public void visit(Statement n)
    {
      n.f0.accept(this);
      nbrStat = nbrStat + 1;
    }
    /* On retourne le nombre d'instructions */
    public int getNbrStat()
    {
      return nbrStat;
    }
}
```

b) Recherche d'une abondance de variables temporaires

En un deuxième temps, nous allons vérifier si la classe en question possède un nombre élevé de variables temporaires. Pour ceci, nous allons parcourir toutes les méthodes des classes afin de compter leur nombre d'attributs; les variables locales que l'on pourra qualifier de variables temporaires. Les variables des méthodes sont temporaires car à l'extérieur du bloc de méthode, la variable n'existe plus.

En Ptidej, nous comptons toutes les variables déclarées dans une classe et non seulement celles qui sont temporaires et locales à une méthode.

En Soul, une méthode est représentée par un arbre logique de structure de donné d'analyse. Cet arbre, pour une méthode, est formé de cinq champs : le nom de la classe, le nom de la méthode, la liste des arguments, la liste des variables temporaires, et une liste des conditions.

Syntaxe :

method([Classe],[Méthode],[<Liste des arguments>],[<Liste variables temporaires>],[<Liste des instructions (statements)>]).

La recherche des variables temporaires est un appel qui parcoure ces 5 champs en se concentrant sur le champ des variables temporaires.

Cet algorithme est similaire à

Algorithme de recherche de **variables temporaires :**

Entrée: Ensemble non-vide de classes

Sortie : *NVT;* nombre de variables temporaires

```
Pour chaque classe
{
      Pour chaque méthode
      {
            public void visit(LocalVariableDecleration n)
                        NVT ← NVT + 1;

      }
}
```

4.3 Longue Liste de Paramètres

a) Paramètre redondant

Cette partie vérifie si un paramètre quelconque est redondant quant aux appels de fonctions. En parcourant toutes les méthodes des classes, nous récupérons les fonctions. En parcourant les arguments de ces méthodes appelées, nous créons une liste qui contiendra tous les arguments utilisés. Ainsi, nous pouvons détecter les arguments qui sont redondants; des paramètres qui se retrouvent à plusieurs reprises dans les arguments d'une méthode. Dans le cas d'une redondance exagérée nous pouvons créer une méthode qui retournera la valeur de l'argument en question, en évitant de l'envoyer en paramètre à chaque appel.

Algorithme de recherche **de paramètres redondants: (appel + déclaration)**

Entrée: Un ensemble non-vide de classes

Sortie: *NPR;* liste de paramètres redondants

```
/*
 * Cette classe est un visiteur.  Son but est de nous
 * montrer le chemin des donnees afin que l'on puisse
 * nous rendre aux informations recherchees.  Cette
 * decrit le parcours que l'on suit afin, en visitant
 * le bon cheminement des noeuds de l'arbre on pourra
 * atteindre un noeud en particulier.  La fonction
 * "accept" sert a accepter un noeud et de preciser
 * que le chemin des donnees passe bel et bien par
 * ce noeud.
 *
 * Auteur  : Jad Karam
 * Date    : Hiver 2007
 * Version : 1.0
 */
```

```
package visitor;
import syntaxtree.*;
import java.util.*;
```

```
public class ParamRedondantVisit extends DepthFirstVisitor
```

```
{
    HashMap paramHM = new HashMap();
    String param="";
    ArrayList listeParam = new ArrayList();


    /* Noeud racine, on accepte ses composantes. */
    public void visit(CompilationUnit n)
    {n.f2.accept(this);n.f3.accept(this);}

     /* On n'a pas besoin du nom du "package", on saute ce
       noeud */
    public void visit(PackageDeclaration n) {}

      /* On n'a pas besoin du nom des "import", on saute ce
       noeud */
    public void visit(ImportDeclaration n) {}

            /*   on   accepte   le   noeud   qui   appellera
"ClassDeclaration()"    */
    public void visit(TypeDeclaration n)
    {n.f0.accept(this);}

    /* Appelee par TypeDeclaration, sert a visiter le noeud
      "UnmodifiedClassDeclaration()" */
    public void visit(ClassDeclaration n)
    {n.f1.accept(this);}

    /* on se dirige vers "ClassBody()" */
    public void visit(UnmodifiedClassDeclaration n)
    {n.f4.accept(this);}

    /* Prochain noeud  "ClassBodyDeclaration()" */
    public void visit(ClassBody n)
    {n.f1.accept(this);}

    /* Sert a parcourir une classe imbriquees dans celle que
      l'on analyse. Ici on analyse tout le contenu de cette
      classe imbriquee  en faisant le meme chemin qui nous a
      servi tout au long.*/
      public void visit(NestedClassDeclaration n)
      {n.f0.accept(this);}

       /*  Ce  noeud  a  plusieurs  enfants  qui  meneront  aux
      composantes du corps de classes */
      public void visit(ClassBodyDeclaration n)
      {n.f0.accept(this);}

      /* Sert a reperer les declarations des methodes */
      public void visit(MethodDeclarationLookahead n)
      {n.f1.accept(this);}

     /* on saute la declaration des interfaces, champs et
      variables */
      public void visit(InterfaceDeclaration n){}

      public void visit(FieldDeclaration n){}
```

```
      public void visit(VariableDeclarator n){}

    /* un parametre a un nom, et on l'extrait de ce noeud */
    public void visit(VariableDeclaratorId n)
    {n.f0.accept(this);}

    /* skip */
    public void visit(VariableInitializer n){}

    /* skip */
    public void visit(ArrayInitializer n){}

    /* On parcourt les signatures des methodes */
    public void visit(MethodDeclaration n)
    {n.f2.accept(this);}

       /* on parcourt la declaration des methodes pour
connaitre
       ses parametres */
    public void visit(MethodDeclarator n)
    {n.f1.accept(this);}

    /* ce noeud nous achemine vers "FormalParameter()" */
    public void visit(FormalParameters n)
    {n.f1.accept(this);n.f2.accept(this);}

    /* ce noeud a 2 derivees, le nom du parametre, son type,
     et on prend le "modifier" "final" si declare */
    public void visit(FormalParameter n)
    {n.f0.accept(this);n.f1.accept(this);n.f2.accept(this);}


    /* Le constructeur possede des parametres et il faudra
      les analyser */
    public void visit(ConstructorDeclaration n)
    {n.f2.accept(this);}

    /* Cette fonction sert a recuperer les nom et types de
      parametres. */
    public void visit(NodeToken n)
    {
        /* On construit un string forme du nom et du type du
            parametre */
            if(!(n.toString().equals(",")||
                              n.toString().equals(")")))
            {
                  param += n.toString()+ " " ;
            }

        /* Au moment ou le nom, le type et le modifier sont
                  dans  le  string  on  l'ajoute  dans  la
HasMap.*/
      if(n.toString().equals(",")||
n.toString().equals(")"))
      {
            int occurrence = 0;
            Object verif;
            if(param.equals(""))
```

```
                    return;

           /* Ici notre HashMap est formee de cles : les details
                du parametre et de valeurs: ses occurrences */
          else{
              if((verif = paramHM.get(param)) != null)
                occurrence = ((Integer)verif).intValue();
              paramHM.put(param,new Integer(occurrence+1));
              param = "";
          }
        }
        }

        public ArrayList trouveRedondance()
        {

        /* On cherche dans la HashMap les parametres qui
           se retrouvent a plusieurs reprises dans le code
           alors on verifie si ce parametre a une valeur de
           plus de 1, alors il se trouve au moins deux fois
           tout au long de la classe.  On pourra toujours
           modifier la valeur seuil.  */
          Set ensemble = paramHM.keySet();
          Iterator it = ensemble.iterator();
          while (it.hasNext()) {
                String parametre = (String)(it.next());
                if(((Integer)paramHM.get(parametre)) > 1)
                      listeParam.add(parametre);

          }
          return listeParam;
        }

    }
```

4.4 Blocs de Donnés

a) Détection de blocs de paramètres qui ne possèdent aucune logique.

Cet anti-patron décrit une partie de code qui regroupe un ensemble de paramètres. Ce bloc est formé de termes qui généralement se retrouvent dans un même contexte. Le défaut intervient quand l'on retrouve ces paramètres ensemble accompagnées de certains qui, sans logique apparente, accompagnent mal les autres de ces paramètres qui suivent. Ceci peut expliquer que la fonction appelée n'est pas spécialisée et gère des donnés farfelues. En se basant sur JWordNet, un générateur de synonymes, antonymes, hyponymes, hyperonymes et autres nous pouvons produire une liste de noms qui se rapportent à un paramètre. Dans le cas où nous ne trouvons, parmi les termes générés, aucune similitude par rapport au reste de la liste des paramètres, nous signalons un défaut de conception.

Par exemple si nous avons l'appel suivant :

```
void foo(int width, Object numberOfCars,
float area, StringBuffer name…)
```

nous constatons qu'il n'y a pas de logique entre les paramètres de cette fonction. Nous devons ainsi signaler un défaut de conception.

```
Algorithme DataClumpVisitor extends DepthFirstVisitor
Entrée : Un ensemble non vide de classes
Sortie : Un descriptif (label) DataClump
Pour chaque classe

/*
  Ayant déjà implanté l'algorithme qui retourne la liste des
paramètres, il nous suffit de créer une instance de JWordNet
pour pouvoir générer les mots qui     définissent un
paramètre que l'on cherche à regrouper.
*/

FileBackedDictionary(RemoteFileManager.lookup(hostname));
DictionaryDatabase dictionary = new FileBackedDictionary();
IndexWord word;
Synset[] senses;


/*
```

```
Dans ce cas nous allons produire tous les noms avec JWordNet
l'idée est de générer les mots provenant du premier
paramètre et de si l'on trouve, avec ces mots, des
similitudes avec le restant de la liste des paramètres de la
fonction (performance).  Nous pouvons aussi produire tous
les mots de tous les paramètres et puis de voir si l'on
trouve des similitudes parmis tous les ensembles produits.
*/

word = dictionary.lookupIndexWord(POS.NOUN,parametre);
senses = word.getSenses();

/*
  De cette façon nous pouvons comparer les sens obtenus les
  uns les autres, et retourner ceux qui ne représentent pas
  de sens en commun.
*/
```

4.5 Interfaces

a) Interface inappropriée

En programmation, la notion d'interfaces est cruciale quant à la flexibilité et réutilisation de systèmes orientées-objet. L'anti-patron « Interface Inappropriée » est détecté lorsque l'on trouve un ensemble de classes qui possèdent un ensemble de méthodes en commun sans qu'elles ne soient fournies par une interface. Une détection automatique consiste à passer à travers toutes les méthodes d'un ensemble de classes, et trouver les méthodes similaires se trouvant dans plusieurs classse. Il s'agit de trouver les sous-classes de chacune des classe et de générer tous les sous-ensembles de sous-classes que l'on pourra former. Dans le cas où l'on retrouve des classes qui implémentent des méthodes sans qu'elles soient fournies par une superclasse ou bien une interface, alors on signale un défaut de conception. Dans ce cas nous nous intéressons qu'aux interfaces et nous négligerons les méthodes qui sont ré-implémentées (overridden) dans les sous-classes.

Les techniques de lister les noms des méthodes se base sur les algorithmes de visiteurs déjà écrites plus haut.

Algorithme **Interface Inappropriée:**

Entrée : Un ensemble non-vide de Classes

Sortie : **Une** liste d'interfaces ou

```
Pour chaque classe
{
      subClass ← retracer ses sous-classes;
            /*
      Interaction de l'utilisateur : dans cette partie,
l'utilisateur devra choisir un nombre de sous-classes dans
un sous-ensemble.  Ainsi il devra choisir la dimension de
l'interface.  Dans chaque sous ensemble on devra, avec une
certaine heuristique, générer toute sorte d'interface qui
peut être produite avec les méthodes qui s'y trouvent sans
compter celles qui sont (overridden).
      */
      subSet ← générer tous les sous-ensemble possibles de
      subClass
      Pour chaque subSet
      {
            Pour chaque subClass
            {
                  Key = String Nom_Méthode;
                  Valeur = Integer Occurence;
                  Crée une HashMap (Key, Valeur);
                  si méthode n'est pas implémentée dans
superclasse
                        add clé à hashmap (increment valeur
de key)
                  finsi
            }
            //Partie de la creation des interfaces
            Parcourir HashMap; (iterator)
            si (valeur >= X)  /* X étant une limite critique
                                    d'occurence       */
            interface ← ajouter key (nom de méthode);

            /*
                  Alternative 1 : retourner la liste
            d'interfaces pour chaque sous-ensemble crée en
            retournant ses classes bien évidemment.
            */
                  retourner interface.
            Finsi
      }
                  /*
      Par la même occasion, faisons l'intersection des
      interfaces.  On devra choisir la dimension de
      l'interface dans ce cas aussi pour y entrer les
      méthodes similaires.  Nous pouvons utiliser la même
      méthode de hashmap plus haut pour trouver les
      similitudes.  Cette opération sert à trouver les
      ensembles de méthodes similaires à rajouter dans
      l'interface.  Ces méthodes seront placées dans un
      ordre où l'occurrence maximale dominera le classement.
      Cette proposition donnera une solution au problème
      aussi.  Une alternative sera de retourner l'ensemble
```

```
d'interfaces trouvées par cet algorithme et comparer
ces interfaces avec celle qui se trouvent dans le
modèle que l'on traite.
            */

      //Alternative 2 : Proposition de solution
      Pour chaque interface
      {
            Key = String Nom_Méthode;
            Valeur = Integer Occurence;
            Crée une HashMap (Key, Valeur);
      }
      Itérer sur la hashmap
      Si( Valeur > 2)
            Interface ← Key (nom_méthode)
      Retourner Interface
}
```

b) Méthodes abstraites non implémentées

Nous savons qu'en Java il est impossible de ne pas implémenter les méthodes déclarées abstraites dans l'interface. Une erreur à la compilation empêchera le programme de s'exécuter. Dans certaines situations, les programmeurs contournent cette étape en déclarant les signatures des fonctions en gardant le corps de celles-ci vide. Dans ce cas nous devons signaler un défaut de conception. L'algorithme qui sert à la détection de l'anti-patron « méthodes abstraites non implémentées » est le suivant :

```
Algoritme  méthodes abstraites non implémentées :

Entrée : Un ensemble non-vide de classes.
Sortie : Liste de méthodes abstraites non implémentées.
Pour chaque classe
{
      Si interface existe (présence de « implements »)
            Visiter interface
            LstI  ← toutes méthodes dans interface, ou
                        class abstraite
            LstC ← toutes méthodes dans sous-classe
            Liste ← Intersection (LstI, LstC)
            Parcourir Liste et visiter méthodes.
            Pour Chaque méthode de liste
            {
                  Évaluer NOS.
                  Si (NOS = 0) //ou un certain nombre critique
                              //On retourne les méthodes qui ne
                              //sont pas implémentées
                        Retourner nom_méthode
                  Sinon
                        Retourner pass;
                  Finsi
            }
      Finsi
}
```

4.6 Responsabilité

a) Feature Envy

Cet anti-patron est présent lorsqu'une classe contient des méthodes qui puisent les valeurs de leurs variables dans d'autres classes.

Ces classes qui ne se contentent pas de leurs attributs et abusent des autres classes. Se trouvant dans des situations dans lesquelles un certain nombre de méthodes possèdent un nombre important d'appel de classes, ces méthodes en question devraient être délocalisée et placées dans les classes convenables. Notre partie se limite à la détection de ce défaut de conception et l'algorithme qui s'en occupe est le suivant :

```
Algorithme Feature Envy
Entrée : Un ensemble non-vide de classes
Sortie : Un descriptif (label) « Feature Envy » qui
qualifie chaque classe de l'ensemble.
Pour chaque classe
{
      Pour chaque méthode
      {
            #appels ← nombre d'appel
                        de classes ou instances d'objets

             /* X est une valeur seuil que l'utilisateur
            devra préciser dépendemment de la grandeur du
            programme reçu pour l'analyse */

            Si (#appels > X )alors
                  Retourner feature envy
            Sinon
                  Retourner pass.
            Finsi
      }
}
```

---

b) Middle Man (Classe déléguée)

Une classe est déléguée lorsque ses méthodes ne contiennent que des appels de méthodes vers d'autres classes qui ne sont ni la classe courante ni la superclasse. Pour détecter les classes déléguées nous devrons parcourir leurs méthodes. L'analyse du code se limite à des méthodes qui ne contiennent que des appels de classes externes et qui ne retournent rien d'autre que les réponses aux paramètres envoyés. Cette classe déléguée fera l'objet d'une façade dans la structure générale du programme, les données y transitent fréquemment pour se rendre aux classes dans la hiérarchie. Nous devons décider si cette façade est nécessaire pour la gestion des sous classes ou bien si elle est tout simplement inutile ou encombrante.

```
Algorithme Middle Man:
Entrée: Un ensemble non-vide de classes
Sortie: Un descriptif (label) MiddleMan, qui qualifie la classe
Pour chaque classe
{
      /*
      On doit analyser le contenu des méthodes afin de savoir si
      les instructions qui s'y trouvent sont des appels de
      classes; des instatiations.
      */
      Pour chaque méthode
      {
            mot ← Lire méthode mot par mot
            si (instruction = appel) alors
                  NbInst ++;
            finsi
      }
      /*
            Évaluer si le nombre « metrique » est important alors
                              qualifier la classe de MiddleMan.
      */
      Si (NbInst > 5) //déterminer une base critique ici mise a 5
            retourner MiddleMan ;
      Sinon
            Retourner Pass
      Finsi
}
```

4.7 Code Inutile : Classes paresseuse

Ce sont des classes qui sont dépourvues de fonctionnalités à la suite de réarrangement de code. Une division de classes en plusieurs petites classe plus spécialisée peut engendrer ce défaut de conception. Ce défaut est la création de classes paresseuses, certaines qui n'ont aucune fonctionnalité, quitte à ne pas contenir de code consistant.

```
Algorithme classes paresseuse
Entrée : Un ensemble non-vide de classes.
Sortie : Un descriptif (label), qualifiant la classe de
« paresseuse ».
Pour chaque classe
{
      Pour chaque méthode
      {
            // Utiliser les métrique :
            NIV ← Nombre de variables d'instance.
            NM ← Nombre de méthodes
            NOS ← Nombre d'instructions.
            // Fixer un nombre critique pour chacune de ces
métriques.
            Si (NIV < NbrCritique1 && NM < NbrCritique2
                        && NOS < NbrCritique3) alors
                  Retourner Classe Paresseuse
            Sinon
                  Retourner Pass
            Finsi
      }
}
```

---

5 – Conclusion et Discussion

5.1 – En résumé

La maintenance de logiciel étant très coûteuse pousse des programmeurs à créer des suites d'outils permettant de détecter automatiquement des défauts de conception. La détection d'anti-patrons prend de plus en plus d'ampleur afin d'améliorer la qualité des logiciels présents ainsi que de prévenir ceux en construction. Ce travail m'a permis de comprendre quelles sont les erreurs qui peuvent se glisser dans un programme si l'auteur ne s'attarde pas minutieusement à la qualité de son code.

La suite Ptidej est un regroupement de modules qui, parmi d'autres fonctionnalités, détectent automatiquement des anti-patrons de conception. En comparant cet environnement avec SOUL, j'ai proposé une série d'algorithmes qui se chargent de détecter certains anti-patrons. Les algorithmes proposés couvrent uniquement les classes Java et analysent syntaxiquement leur contenu. Cette analyse aide à détecter automatiquement les anti-patrons proposés mais il est toujours nécessaire d'avoir une interaction avec l'utilisateur. Certains paramètres sont requis pour adapter et spécialiser la détection. Donc il est nécessaire que l'utilisateur suive le bon déroulement du processus.

Dans ce projet j'ai couverts les aspects suivants :

- Familiarisation avec Ptidej et SOUL (Smalltalk et prolog).
- Comparaison des environnements Ptidej et SOUL.
- Proposition d'algorithmes de détection d'anti-patrons.

5.2 - Discussion

Les algorithmes établis dans ce projet présentent plusieurs aspects d'implémentation. Certains algorithmes sont itératifs, d'autres récurrents et d'autres se basent sur des arbres de syntaxe abstraite. En construisant un arbre, la récurrence rentre évidemment en jeu. Ayant proposé de construire un arbre de syntaxe abstraite en me basant sur la grammaire de Java 1.5, ceci m'a permis de parcourir le code et de construire des nœuds qui représentent toutes les composantes du code. Étant limité dans le temps, je n'ai pu construire les visiteurs pour tous les algorithmes. Dans deux cas de détection d'anti-patrons, j'ai opté pour cette méthode d'implémentation, mais il ne reste qu'à répéter ce même processus pour les autres cas possibles. J'aurai certainement eu du plaisir à aller jusqu'au bout mais j'aurai eu besoin de deux semaines supplémentaires afin de compléter ces tâches.

Étant un langage de programmation logique, SOUL m'a été un obstacle à cause de son traitement par unification. Les requêtes étant générées en Prolog, ces algorithmes sont spécifiques et structurés en arbres qui retournent toutes les solutions possibles rencontrées. Pour passer en Java, qui est un langage impératif, j'ai du analyser les différences des langages, comprendre la fonction des algorithmes et passer de l'aspect logique à l'aspect orienté objet. Je me suis donc basé sur les définitions, les buts ainsi que le déroulement des algorithmes proposés dans SOUL afin de construire ceux présents dans ce travail.

Dans le tableau 3.0, on remarque qu'il y a des anti-patrons qui ne peuvent pas être implémentés dans Ptidej. Cet empêchement est du à la nature du langage Smalltalk qui, grâce aux méta-classes et aux méta-opérations, nous permet de recueillir des valeurs de variables ainsi que l'état du programme. Ainsi dans les anti-patrons « Nombres Magiques », « Chaines de Messages », « Paramètres non utilisés », « Instances non utilisées », « Switch sur un paramètre », et « Switch sur une condition nulle », ces anti-patrons sont traités facilement dans le cadre du langage Smalltalk. Dans le cas des « Interfaces refusées » et de « L'héritage refusé », nous pouvons définir des opérations dans les méta-classes qui nous retournerons l'état des objets et des classes. Dans le cas

du « Code dupliqué dans les méthodes » ce traitement fait référence non seulement à la ressemblance syntaxique du code, mais à la structure et la complexité du code qui est souvent syntaxiquement différent.

5.2 – Travail Futur

Ce projet est une ouverture à divers travaux futurs. Ces travaux porteront d'une part sur l'implémentation des algorithmes proposés suivant un des modèles présentés dans ce projet. D'une autre, l'utilisation des grammaires de langages différents pour détecter des anti-patrons présents dans des applications autres que Java ou C++. Par ailleurs nous pouvons orienter la recherche d'anti-patrons dans une direction plutôt lexicale et littéraire. Un bon programmeur est celui qui est capable d'écrire du code que les hommes comprennent. Ainsi j'espère un jour que la détection automatique agira sur la vérification des noms donnés aux variables et aux méthodes présents dans le code afin de juger si ces descriptifs sont bien cohérent avec les traitements qui leurs sont attribués.

Bibliographies

Design Patterns: Elements of Reusable Object-Oriented Software (Addison-Wesley Professional Computing Series) Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Ptidej :
- *http://www.yann-gael.gueheneuc.net/Work/Research/Introduction/*

SOUL :
- *Munoz03-MasterThesis.pdf*
- *http://prog.vub.ac.be/SOUL/index.html*

JTB : *http://compilers.cs.ucla.edu/jtb/jtb-2003/*
JavaCC : *https://javacc.dev.java.net/*

Annexe

```
/*
* Fichier Main.java, qui sert de lancer l'analyseur syntaxique.
* Dans ce fichier, nous initialisons le parseur afin de
* parcourir les classes donnees en entree.  Cette classe
* couvre deux facons de recevoir des fichiers en entree.
* Le but de cette classes est de retourner les resultats
* des metriques qui se chargent de compter les nombre de
* variables d'instances, le nombre d'instructions et de
* detecter les parametres redondants dans un ensemble de
* classes.
*
* Auteur  : Jad Karam
* Date    : Hiver 2007
* Version : 1.0
*
*/


import syntaxtree.*;
import visitor.*;
import java.util.*;
import java.io.*;

public class Main
{
    JavaParser parser;
    boolean premier = true;
    public void fileScan(File file)
    {

        File unFichier = file;
        if(!(unFichier.exists()))
        {
            System.out.println("le fichier ou repertoire: "
                               + unFichier.getAbsolutePath()
                               +" n'existe pas dans ce
repertoire");
            return;

        }
        /* si c'est un fichier .java on le traite directement*/
        if (unFichier.isFile() && isJava(file))
        {
            traiter(unFichier);
        }
        /* Dans le cas ou l'on tombe sur un repertoire, on parcoure re-
cursivement
           tous ses fichiers et ses repertoires */
        else if(unFichier.isDirectory())
        {
            /* Tableau contenant tous les noms de fichiers
               présent dans le repertoire */
            String [] listeFichiers = unFichier.list();
            List fichiers = new ArrayList();
```

```
            /* Dans cette boucle nous parcourons le contenu du reper-
toire */
            for(int i=0; i<listeFichiers.length; i++)
            {
                File file2 = new File(unFichier,listeFichiers[i]);

                /*si c'est un fichier, on l'ajoute au ArrayList
                  des fichiers à Scanner*/
                if (file2.isFile() && isJava(file2))
                    fichiers.add(file2);

                /* sinon on analyse le contenu de ce sous-repertoire */
                if(file2.isDirectory())
                    fileScan(file2);
            }

            /*pour éviter de decoder une liste de fichier vide    */
            if(fichiers!=null)
            {
                /* On traite la liste de fichiers
                   dans traiter(List fichiers) */
                traiter(fichiers);
            }
        }
    }

    /* On traite une liste de fichiers du repertoire */
    public void traiter(List fichiers)
    {
        /* On itere sur la liste de fichiers et analyse chacun */
        ListIterator iterator = fichiers.listIterator();
        while(iterator.hasNext())
            traiter((File)iterator.next());
    }


    /* Le parseur s'initialise et commence ici. */
    public void traiter(File file)
    {

        try
        {
            /* Pour le premier fichier, on initialise le parseur */
            if(premier)
            {
                parser = new JavaParser(new
java.io.FileInputStream(file));
                premier = false;
                analyser(parser,file);
            }

            /* A chaque nouveau fichier, nous devons reinitialiser le
               parseur afin de ne pas meler les Token et le reste */

            else{
                parser.ReInit(new java.io.FileInputStream(file));
                analyser(parser,file);
```

```
            }

        }
        /* Dans le cas ou un fichier n'est pas dans le repertoire cou-
rant */
        catch (java.io.FileNotFoundException e)
        {
            System.err.println("Fichier " + file.getName() +
                               " n'existe pas ou n'est pas dans le
repertoire.");
            return;
        }

    }

    /* L'analyseur syntaxique debute ici. Dans cette partie nous
       allons essayer de detecter 3 anti-patrons.  Le premier test
       est de detecter un nombre eleve de variables d'instances,
       le deuxieme est de detecter un nombre eleve d'instructions
       et finalement de detecter des parametres redondants (parametres
       qui se retrouvent dans plusieurs methodes appelees)
    */
    public void analyser(JavaParser parser, File file)
    {
        try{
            /*On pointe le pareur au noeud parent (racine)*/
            Node root = parser.CompilationUnit();

            /* Reinitialisation du parseur */
            parser.ReInit(new java.io.FileInputStream(file));

            /*Instance de la classe qui compte le nombre de variables
d'instances */
            ComptInst cInst = new ComptInst(parser,file);

            /* Retourne le nombre de variables */
            int nbrInst = cInst.getResults();
            System.out.println("Le nombre total de variables
d'instances pour le fichier "
                               + file.getName() + " est : " + nbrInst);

            /* Reinitialisation du parseur */
            parser.ReInit(new java.io.FileInputStream(file));
            /* Partie qui s'occupe des parametres redondants */
            VerifRedondance vr = new VerifRedondance(parser);

            ParamRedondantVisit prv = new ParamRedondantVisit();

            /* On retourne la liste de tous les parametres qui se re-
trouvent
               plus d'une fois dans des methodes */
            ArrayList listeRedondance = vr.getResults();

            /* On affiche la liste des parametres redondants */
            if(listeRedondance != null)
            {
```

```
                Iterator it = listeRedondance.iterator();
                System.out.print("Parametres Redondants =  " );
                while(it.hasNext())
                    System.out.print( it.next());
                System.out.println();
            }
            else
                return;
            /* Reinitialisation du parseurs */
            parser.ReInit(new java.io.FileInputStream(file));
            /* Rien ne nous empeche d'appeler directement le visiteur
               et de parcourir le code fourni en entree afin de compter
               le nombre d'instructions */
            StatementVisitor sv = new StatementVisitor();
            int nbrStm = 0;
            /* On accepte le noeud racine (Compilation Unit) */
            root.accept(sv);
            nbrStm = sv.getNbrStat();
            System.out.println("Nombre total d'instructions = " +
nbrStm);
            System.out.println();



        }
        catch(java.io.FileNotFoundException e){
            System.err.println("Fichier " + file.getName() +
                               " n'existe pas ou n'est pas dans le
repertoire.");
            return;
        }
        catch(ParseException e) {
            System.err.println(e.getMessage());
    }
    }



    /* Methode qui retourne vrai si le fichier est un .java */
    public boolean isJava(File file)
    {
        String nom = file.getName();
        String finNom = nom.substring(nom.length()-5);
        return finNom.equals(".java");
    }


    public static void main(String [] args)
    {
        /* Si l'on a un repertoire a analyser les fichiers,
           on appelle la classe Main sans arguments. Dans le
           cas ou on veut limiter l'analyse a quelques fichiers,
           nous pouvons appeler ces fichiers en argument
        */
        Main m = new Main();
        /* si aucun argument n'est entré */
        if (args.length == 0 ){
```

```
            /* on cherche à partir du répertoire courant */
            m.fileScan(new File("."));
        }
        /* Analyser les fichiers et répertoires dont le
           nom apparait en argument */
        if(args.length >0)
            for(int i=0; i<args.length; i++)
                m.fileScan (new File(args[i]));

        else
        {
            System.err.println(" Usage est comme suit :");
            System.err.println("         java Main inputfiles");
            System.err.println("OU");
            System.err.println("         java Main");
            return;
        }
    }


}

/*************************************************************************/

/* Fichier ComptInst.java */
/* Cette classe sert a compter le nombre de variables
*  d'instance qui se retrouvent dans une classe et dans
*  les classes parents aussi si celles-si se trouvent dans
*  le meme repertoire que la classe fille.  Sinon, il suffit
*  de refaire le traitement dans la classe Main a la recherche
*  de la classe parent qui peut se retrouver ailleur.
*  En s'inspirant du nom du package de la classe que l'on analyse,
*  on pourra ainsi reperer la classe parent.
*
*  Auteur  : Jad Karam
*  Date    : Hiver 2007
*  Version : 1.0
*
*/


import syntaxtree.*;
import visitor.*;
import java.util.*;
import java.io.*;

public class ComptInst
{
    JavaParser parser;
    int instVar;
    String fichier;
    File file;

    public ComptInst(JavaParser parser, File file)
    {
        this.parser = parser;
        this.file = file;
    }
```

```
    public int getResults()
    {
        analyse(0,file.getName());
        return instVar;
    }
    /* Analyseur syntaxique qui appelle le visiteur
       VarInstVisiteur avec le nombre de variables
       d'instance precedent pour retourner la somme
       de toutes les variables d'instance de la branche.
    */
    public void analyse(int varInst, String fichier)
    {
        try{
            System.out.println("Analyse du fichier: " +
fichier);
            String classeSuper;
            Node root = parser.CompilationUnit();
            VarInstVisiteur v = new VarInstVisiteur(varInst);
            root.accept(v);
            instVar = v.getInstVar();

            /*
               L'instance h est de la classe Heritage qui retourne
               tout simplement le nom de la classe de laquelle on he-
rite
               les fonctionnalites.
            */

            Heritage h = new Heritage();
            root.accept(h);
            classeSuper = h.getClasseSuper();
            if(!classeSuper.equals("")){
                classeSuper += ".java";
                System.out.println("Classe Super = "  + classeSuper );
                parser.ReInit(new java.io.FileInputStream(classeSuper));
                analyse(instVar,classeSuper);}
            else
            {
                return;
            }

        }
        catch(ParseException e) {
        System.err.println(e.getMessage());
    }
        catch(java.io.FileNotFoundException e){
            System.err.println(e.getMessage());
        }
    }
}




/*************************************************************/
```

```
/* Fichier VerifRedondance.java */

import visitor.*;
import syntaxtree.*;
import java.io.*;
import java.util.*;

/*
* Cette classe est appellee par le main et appelle
*  par la suite le visiteur en recevant le parseur
*  en entree.  Elle sert donc a detecter les
*  parametres qui se retrouvent a plusieurs reprises
*  dans des methodes de la classes a analyser
*
*  Auteur  : Jad Karam
*  Date    : Hiver 2007
*  Version : 1.0
*/


public class VerifRedondance
{
    JavaParser parser;
    ArrayList listeRedondance;

    public VerifRedondance(JavaParser parser)
    {
        this.parser = parser;
    }
    /*
      Methode qui sert a retourner la liste
       des parametres redondants trouves par le
       visiteur ParamRedondantVisit
    */
    public ArrayList getResults()
    {
        analyse();
        return listeRedondance;
    }
    public void analyse()
    {
        try{

            /* On pointe le noeud racine au noeud Compilation Unit */
            Node root = parser.CompilationUnit();
            /* On appelle le visiteur qui analysera syntaxiquement les
               parametres formels de toutes les methodes des classes */
            ParamRedondantVisit prv = new ParamRedondantVisit();
            /* on accepte le noeud racine */
            root.accept(prv);
            /* on recupere la liste de parametres redondants */
            listeRedondance = prv.trouveRedondance();
        }
        catch(ParseException e) {
        System.err.println(e.getMessage());
    }
    }
}
```

```
/*************************************************************************/

/*  Fichier Heritage.java
* Cette classe est un autre visiteur qui sert a
* retourner le nom de la classe de laquelle on herite
* les fonctionnalites.  Cette classe nous aide a remonter
* l'arbre de derivation et d'heritage de classes afin de
* compter le nombre de variables d'instances dans une classe
* et ses parents.  Cependant certain fichiers, desquels on herite,
* sont dans des  repertoires differents et nous devrons suivre le
* classpath * par exemple.  Dans ce cas nous pouvons envoyer ce
* chemin a la class main, et elle parcourras les repertoires a la
* recherche de la classe mere.  Ce parcours se fait comme celui
* qui nous a servi de retrouver les parametres d'une methode.
* Dans ce cas nous devrons tout simplement retourner le nom
* de la classe de laquelle on herite en visitant le noeud qui
* s'occupe des "extends".
*
* Auteur   : Jad Karam
* Date     : Hiver 2007
* Version  : 1.0
*
*/


package visitor;

import syntaxtree.*;
import java.util.*;
import java.io.*;


public class Heritage extends DepthFirstVisitor
{
    String heritage = "";
    int i = 0;

    /* On skip le nom du package */
    public void visit(PackageDeclaration n) {}

    /* On skip le nom des importations */
    public void visit(ImportDeclaration n){}

    /* On parcourt le type afin d'avoir la declaration de la classe */
    public void visit(TypeDeclaration n){n.f0.accept(this);}

    /* On ne s'occupe pas des classes interners */
    public void visit(NestedClassDeclaration n){}

    /* On visite la UnmodifiedClassDeclaration() */
    public void visit(ClassDeclaration n)
    {n.f1.accept(this);}

    /* On visite le noeud [ "extends" Name() ] */
    public void visit(UnmodifiedClassDeclaration n)
    {n.f2.accept(this);}
```

```
    /* Skip */
    public void visit(ClassBody n){}
    public void visit(InterfaceDeclaration n){}
    public void visit(UnmodifiedInterfaceDeclaration n){}

    /* On recupere le nom de la classe super */
    public void visit(NodeToken n)
    {
    if(n.toString().equals(""))
            return;
        else
            heritage = n.toString();
    }

    /* On retourne le nom de la classe super */
    public String getClasseSuper()
    {
    return heritage;
    }

}


/*************************************************************************/

/* Fichier ParamRedondantVisit.java
* Cette classe est un visiteur.  Son but est de nous
* montrer le chemin des donnees afin que l'on puisse
* nous rendre aux informations recherchees.  Cette
* decrit le parcours que l'on suit afin, en visitant
* le bon cheminement des noeuds de l'arbre on pourra
* atteindre un noeud en particulier.  La fonction
* "accept" sert a accepter un noeud et de preciser
* que le chemin des donnees passe bel et bien par
* ce noeud.
*
* Auteur  : Jad Karam
* Date    : Hiver 2007
* Version : 1.0
*/
package visitor;
import syntaxtree.*;
import java.util.*;

public class ParamRedondantVisit extends DepthFirstVisitor
{
    HashMap paramHM = new HashMap();
    String param="";
    ArrayList listeParam = new ArrayList();


    /* Noeud racine, on accepte ses composantes. */
    public void visit(CompilationUnit n)
    {n.f2.accept(this);n.f3.accept(this);}

    /* On n'a pas besoin du nom du "package", on saute ce noeud */
    public void visit(PackageDeclaration n) {}
```

```
     /* On n'a pas besoin du nom des "import", on saute ce noeud */
    public void visit(ImportDeclaration n) {}

    /* on accepte le noeud qui appellera "ClassDeclaration()" */
    public void visit(TypeDeclaration n)
    {n.f0.accept(this);}

    /* Appelee par TypeDeclaration, sert a visiter le noeud "Unmodi-
fiedClassDeclaration()" */
    public void visit(ClassDeclaration n)
    {n.f1.accept(this);}

    /* on se dirige vers "ClassBody()" */
    public void visit(UnmodifiedClassDeclaration n)
    {n.f4.accept(this);}

    /* Prochain noeud  "ClassBodyDeclaration()" */
    public void visit(ClassBody n)
    {n.f1.accept(this);}

    /* Sert a parcourir une classe imbriquees dans celle que l'on ana-
lyse.
     Ici on analyse tout le contenu de cette classe imbriquee en fai-
sant le
     meme chemin qui nous a servi tout au long.*/
    public void visit(NestedClassDeclaration n){n.f0.accept(this);}

    /* Ce noeud a plusieurs enfants qui meneront aux composantes du
corps de
       classes */
    public void visit(ClassBodyDeclaration n)
    {n.f0.accept(this);}

    /* Sert a reperer les declarations des methodes */
    public void visit(MethodDeclarationLookahead n){n.f1.accept(this);}

    /* on saute la declaration des interfaces, champs et variables */
    public void visit(InterfaceDeclaration n){}

    public void visit(FieldDeclaration n){}

    public void visit(VariableDeclarator n){}

    /* un parametre a un nom, et on l'extrait de ce noeud */
    public void visit(VariableDeclaratorId n)
    {n.f0.accept(this);}

    /* skip */
    public void visit(VariableInitializer n){}

    /* skip */
    public void visit(ArrayInitializer n){}

    /* On parcourt les signatures des methodes */
    public void visit(MethodDeclaration n)
    {n.f2.accept(this);}
```

```
/* on parcourt la declaration des methodes pour connaitre
   ses parametres */
public void visit(MethodDeclarator n)
{n.f1.accept(this);}

/* ce noeud nous achemine vers "FormalParameter()" */
public void visit(FormalParameters n)
{n.f1.accept(this);n.f2.accept(this);}

/* ce noeud a 2 derivees, le nom du parametre, son type,
 et on prend le "modifier" "final" si declare */
public void visit(FormalParameter n)
{n.f0.accept(this);n.f1.accept(this);n.f2.accept(this);}


/* Le constructeur possede des parametres et il faudra les
  analyser */
public void visit(ConstructorDeclaration n)
{n.f2.accept(this);}

/*
  Cette fonction sert a recuperer les nom et types de parametres.
  */
public void visit(NodeToken n)
{
 /*
  On construit un string forme du nom et du type du parametre
  */
  if(!(n.toString().equals(",") || n.toString().equals(")")))
  {
    param += n.toString()+ " " ;
  }

    /* Au moment ou le nom, le type et le modifier sont dans
        le string on l'ajoute dans la HasMap.*/
   if(n.toString().equals(",") || n.toString().equals(")"))
   {
        int occurrence = 0;
        Object verif;
        if(param.equals(""))
               return;

    /* Ici notre HashMap est formee de cles : les details du
  parametre et de valeurs: ses occurrences */
     else{
               if((verif = paramHM.get(param)) != null)
               occurrence = ((Integer)verif).intValue();
               paramHM.put(param,new Integer(occurrence+1));
               param = "";
          }
   }
      }

      public ArrayList trouveRedondance()
      {

          /* On cherche dans la HashMap les parametres qui
             se retrouvent a plusieurs reprises dans le code
```

```
                alors on verifie si ce parametre a une valeur de
                plus de 1, alors il se trouve au moins deux fois
                tout au long de la classe.  On pourra toujours
                modifier la valeur seuil.  */
                Set ensemble = paramHM.keySet();
                Iterator it = ensemble.iterator();
                 while (it.hasNext()) {
                       String parametre = (String)(it.next());
                       if(((Integer)paramHM.get(parametre)) > 1)
                             listeParam.add(parametre);

                 }
          return listeParam;
        }

    }


/*********************************************************************/

/* Fichier StatementVisitor.java */
/* Cette classe sert d'un visiteur.  Ce visiteur
* a pour fonction de compter le nombre d'instructions
* dans une classe donnee.  A chaque noeud statements par
* lequel passe notre analyseur syntaxique en parcourant le
* code, nous incrementons un compteur de nombre d'instructions.
* A la fin de l'analyse de la classe nous retournons le nombre
* d'instructions parcourues dans le code.
*
* Auteur  : Jad Karam
* Date    : Hiver 2007
* Version : 1.0
*
*
*/

package visitor;
import syntaxtree.*;


public class StatementVisitor extends DepthFirstVisitor
{
    int nbrStat;

    /*
       Nous n'avons pas besoin de preciser le chemin complet pour
       l'analyseur a cause que l'on ne filtre pas les donnees.
       Nous ajoutons ce visiteur tout simplement afin que, une fois
       que l'on rencontre une variable, on incremente le compteur du
       nombre d'instructions dans le code
    */

public void visit(Statement n)
    {
      n.f0.accept(this);
      nbrStat = nbrStat + 1;
    }
```

```
    /* On retourne le nombre d'instructions */
    public int getNbrStat()
    {
      return nbrStat;
    }

}

/*************************************************************/

package visitor;
import syntaxtree.*;

/* Fichier VarInstVisiteur.java
* Cette classe sert a parcourir le code recu en
* entree.  Ainsi on specialise ce visiteur a ne
* parcourir que des noeuds specifiques.  Nous
* rappelons que le (JTB) nous a servi a l'aide
* d'une grammaire de construire les noeuds que
* l'on visite et qui contiennent l'information
* contenue dans la classe a analyser.  Alors
* la specialisation se limite aux noeuds de
* type VariableDeclarator.  A chaque fois que
* l'on rencontre un noeud de ce type on incremente
* le compteur du nombre de variables d'instances
* presentes dans la classe a analyser.
*
* Auteur  : Jad Karam
* Date    : Hiver 2007
* Version : 1.0
*
*/
public class VarInstVisiteur extends DepthFirstVisitor
{
    int instVar;

    public VarInstVisiteur(int varInst)
    {
        this.instVar = varInst;
    }
    /* On visite le noeud VariableDeclarator et increment
       le compteur a chaque fois qu'on le rencontre */
    public void visit(VariableDeclarator n)
    {
        n.f0.accept(this);
        n.f1.accept(this);
        instVar = instVar+1;

    }

    public int getInstVar()
    {
        return instVar;
    }

}
```