

Génie Logiciel: Labo Final

Schmidt Sébastien - M18

Table des matières

1	Technologies utilisées	3
1.1	GitHub	3
1.2	Maven	3
1.2.1	pom.xml	3
1.3	JUnit	4
1.4	Jacoco	4
1.5	Jenkins	4
2	Diagramme de classes	4
3	Diagramme pour les différents packages	7
4	Diagramme de séquence	7
5	Choix de conception	8
6	Techniques de tests	10
7	Cas concrets d'utilisation	10

1 Technologies utilisées

Pour ce projet final de génie logiciel, j'ai utilisé quelques technologies que je vais me charger d'expliquer dans cette section. Certaines étaient demandées et d'autres non, j'expliquerai mes choix quant à l'utilisation de ces technologies.

1.1 GitHub

C'est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions Git. Il permet la gestion de version décentralisée. Cela fonctionne par commit, chaque personne pouvant travailler sur ce projet possèdera une version locale. Les changements apportés à cette version seront alors «committés» sur le serveur où chacun devra «pull» les données régulièrement pour avoir les dernières versions. Il est également possible de revenir en arrière vers un précédent commit lorsque l'application était stable. C'est principalement pour cette raison que j'ai utilisé un répertoire Git public.

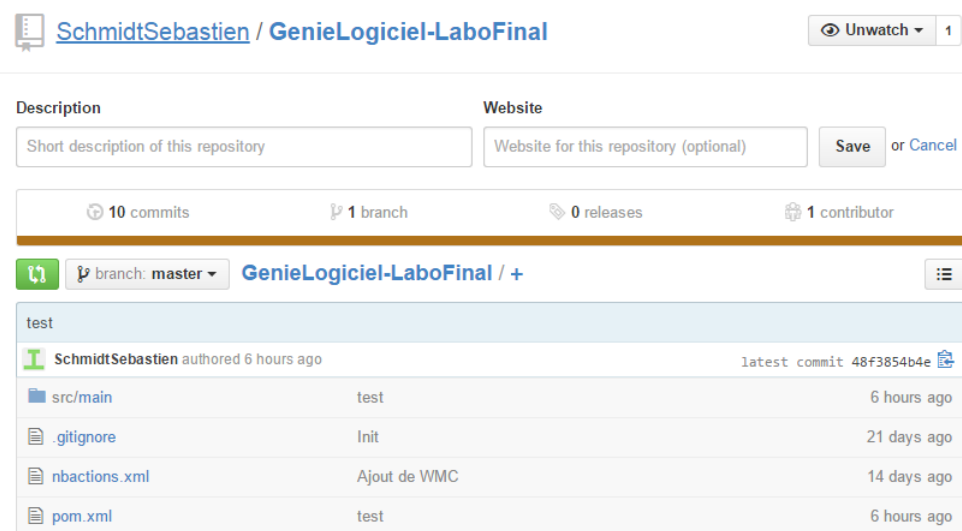


FIGURE 1 – Le repository de l'application finale de génie logiciel sur GitHub

1.2 Maven

Maven est un outil prévu pour la gestion et l'automatisation de production des projets logiciels Java. L'objectif recherché est comparable au système Make sous Unix : produire un logiciel à partir de ses sources, en optimisant les tâches réalisées à cette fin et en garantissant le bon ordre de fabrication. De cette façon quand on met en projet sur un Git, tous les fichiers propres à la configuration de l'IDE ne sont pas pris en compte. Il est semblable à l'outil Ant, mais fournit des moyens de configuration plus simples, eux aussi basés sur le format XML. Maven utilise un fichier de configuration POM (Project Object Model) permettant de décrire un projet logiciel et ses dépendances avec des modules externes (comme javaparser par exemple) et l'ordre à suivre pour sa production.

1.2.1 pom.xml

TODO

1.3 JUnit

Pour effectuer le testing de l'application, j'ai utilisé le framework JUnit dans le but de réaliser des tests unitaires en programmation Java.

1.4 Jacoco

Pour réaliser la couverture des tests, j'ai utilisé un outil permettant la couverture de code comme Jacoco qui est plus adapté aux tests d'intégration que Cobertura. Il est intégré au projet sous forme de dépendance via Maven. Toutes ses informations sont donc placées dans le pom.xml. Le plugin JaCoCo va générer la configuration de l'agent pour lancer les tests.

GenieLogiciel-LaboFinal

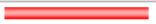

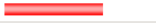
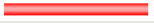

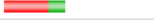






Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
GUI		0 %		0 %	20 20	164 164	17 17	3 3
Visitor		0 %		0 %	45 45	143 143	24 24	7 7
GodClass.Calculator		58 %		28 %	21 43	36 87	13 34	2 5
default		0 %		0 %	12 12	24 24	7 7	2 2
GodClass		0 %	n/a	n/a	3 3	14 14	3 3	1 1
Utils		80 %		75 %	3 17	4 33	0 9	0 2
GUI.States		0 %	n/a	n/a	6 6	11 11	6 6	3 3
Total	1.641 of 1.957	16 %	74 of 91	19 %	110 146	396 476	70 100	18 23

FIGURE 2 – La couverture du code à un état peu avancé du projet

1.5 Jenkins

Outil d'intégration continue qui est en réalité un fork de l'outil Hudson. Il s'interface avec des systèmes de gestion de versions tels que CVS, Git et Subversion, et exécute des projets basés sur Apache Ant et Apache Maven. Ce serveur permet donc de configurer un projet en fonction des dépendances dont on a besoin. Pour ce projet, j'ai configuré le serveur pour aller récupérer les données d'un projet Maven sur un repository Git tout en effectuant la couverture du code avec Jacoco. Tous ces outils sont par conséquent configuré dans les paramètres du serveur. Un build s'effectue une fois par jour.

2 Diagramme de classes

Sur la figure 3 on peut voir que j'ai en fait un objet «Calculator» qui va encapsuler les différentes classes qui permettront de calculer les métriques. Elle possède par conséquent quatre classes :

GeneralCalculator : Cette classe aura la tâche de stocker les données générales à l'application comme le nombre de lignes qu'une classe possède, son nom ou encore son nombre de méthodes.

WeightedMethodCountCalculator : Cette classe va compter le nombre de branchement d'une classe et on obtiendra le métrique calculé par la méthode de McCabe. Moins il y a de branchements, moins la classe sera complexe, mieux ce sera.

ClassCohesionCalculator : Cette classe mesure la cohésion d'une classe en fonction de ses variables membres. Plus celles-ci sont utilisées dans différentes méthodes, plus la classe sera cohérente.

AccessToForeignDataCalculator : Cette classe mesure le nombre de classes externe qui seront utilisées via ses attributs ou ses accesseurs. De nouveau moins il y en aura, mieux ce sera.

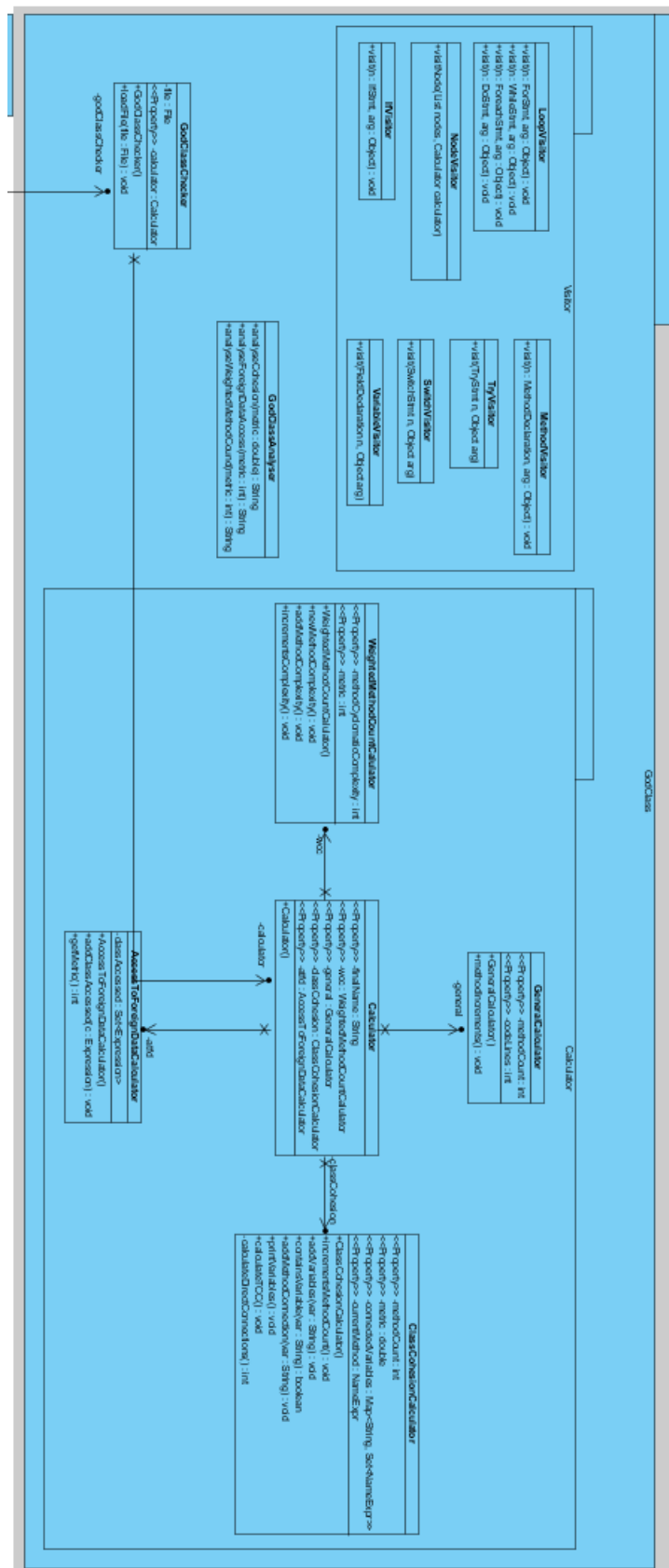


FIGURE 3 – Première partie du diagramme de classe

Toutes ces classes auront donc les métriques calculés. C'est la classe «GodClassChecker» qui possèdera une instance de la classe «Calculator». Comme le java parser utilise des visitors pour parcourir l'ensemble de l'arbre AST construit, je passerai cette classe en paramètre et les différents métriques seront calculés au fur et à mesure que la classe sera parsée. Quand le parsing sera terminé, il me suffira ensuite d'utiliser la classe «GodClassAnalyser» qui comme son nom l'indique va analyser les différents résultats obtenus. Elle possède trois méthodes : une pour chaque métrique à analyser. Chacune d'entre elle prendra le métrique correspondant et sortira une petite analyse afin de savoir si la classe a bien été conçue.

Comme dit précédemment, le javaparser utilise les visiteurs pour parcourir l'ensemble de la classe. Chacun des visiteurs que j'ai créé va hériter d'une classe VoidVisitorAdapter. Celle-ci contiendra une série de méthodes adaptées pour parcourir chaque type de lignes. Les visiteurs utilisés :

MethodVisitor : Il va visiter l'ensemble de chaque méthode et va pouvoir ensuite appeler le visiteur correspondant au type de «statement» rencontré.

NodeVisitor : C'est lui qui, appelé par le «MethodVisitor» va pouvoir redirigé le parser vers le bon visiteur

LoopVisitor : Celui-ci, comme son nom le laisse penser, permet de parcourir les différents types de boucle. Il possède trois méthodes : une pour les boucle while, une boucle pour les boucles for et une pour les boucle do.

TryVisitor : Ce visiteur va pouvoir parcourir les blocs Try catch.

SwitchVisitor : Va permette de parcourir les switches et ses différentes case.

VariableVisitor : On arrive à la fin de l'arbre, il ne reste plus que les variables à analyser, c'est lui qui va s'en occuper.

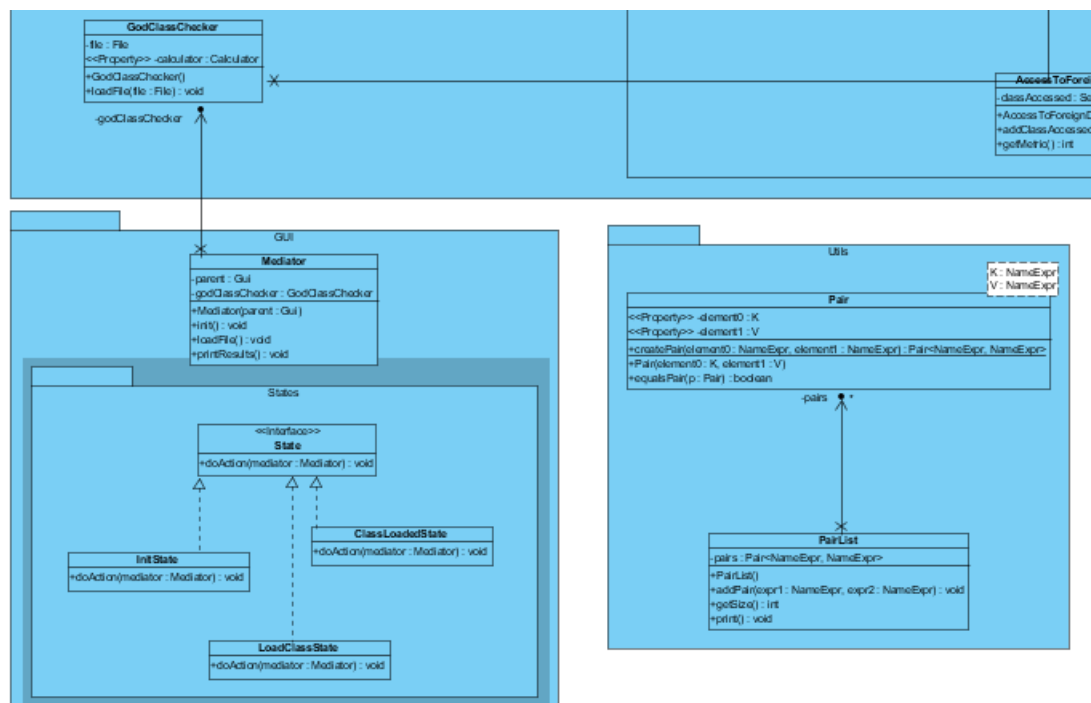


FIGURE 4 – Deuxième partie du diagramme de classe

Sur la seconde partie du diagramme de classe (figure 4, on peut voir l'utilisation d'un médiateur, celui-ci permet en réalité de centraliser les interactions entre les différents composants de l'interface graphique. Il aura en variable membre une instance

d'un «GodClassChecker» et utilisera le «GodClassAnalyser» pour analyser et afficher des messages sur ces métriques calculés.

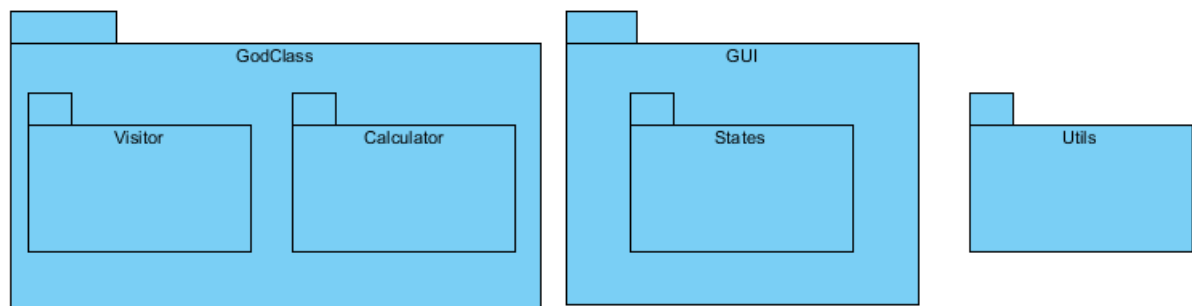
On peut également voir un package «States» qui comprendra trois états. Ceux-ci reflètent l'état dans lequel se trouve l'application :

InitState : Etat initial de l'application, des actions effectuées qu'une seule fois ce feront à ce moment.

LoadClassState : Une nouvelle classe est chargée en mémoire, il faut la charger et la parser.

ClassLoadedState : La classe a été parsée, il faut maintenant afficher les résultats et les calculer.

3 Diagramme pour les différents packages



J'ai divisé mon application en trois packages principaux :

1. Un package «GodClass» qui encapsulera un nouveau package «Visitor» et un autre «Calculator».
2. Un package «GUI» qui englobera un autre package «States»
3. Et enfin un package «Utils» dans lequel je placerai les classes utiles utilisées dans d'autres classes.

4 Diagramme de séquence

Pour ce diagramme de séquence, je vais faire abstraction des classes d'état, des classes représentent les différents métriques (ils seront tous représentés par la classe «Calculator») ainsi que la classe GUI gérant l'interface graphique. On commence avec le médiateur qui appelle une méthode du «GodClassChecker» pour charger la classe en mémoire. On passe ensuite dans un premier visiteur qui est celui de la méthode et elle va appeler plusieurs méthodes de la classe «Calculator». On va ensuite parcourir chaque noeuds de la méthode dans une boucle : il y aura deux cas d'utilisation :

1. Si le noeud est une variable, on va dans ce cas là ajouter des informations concernant ce que le parser est train de lire. Ces informations seront enregistrées via des appels de méthodes de la classe «Calculator».
2. Si le noeud est un statement comme un if, une boucle, un bloc try, ... il faudra incrémenter la complexité cyclomatique du caclulator. Pour chaque noeud de celui-ci on parcourra la boucle.

Une fois cette boucle terminée, il reste à calculer le WMC de et d'afficher les différents résultats obtenus.

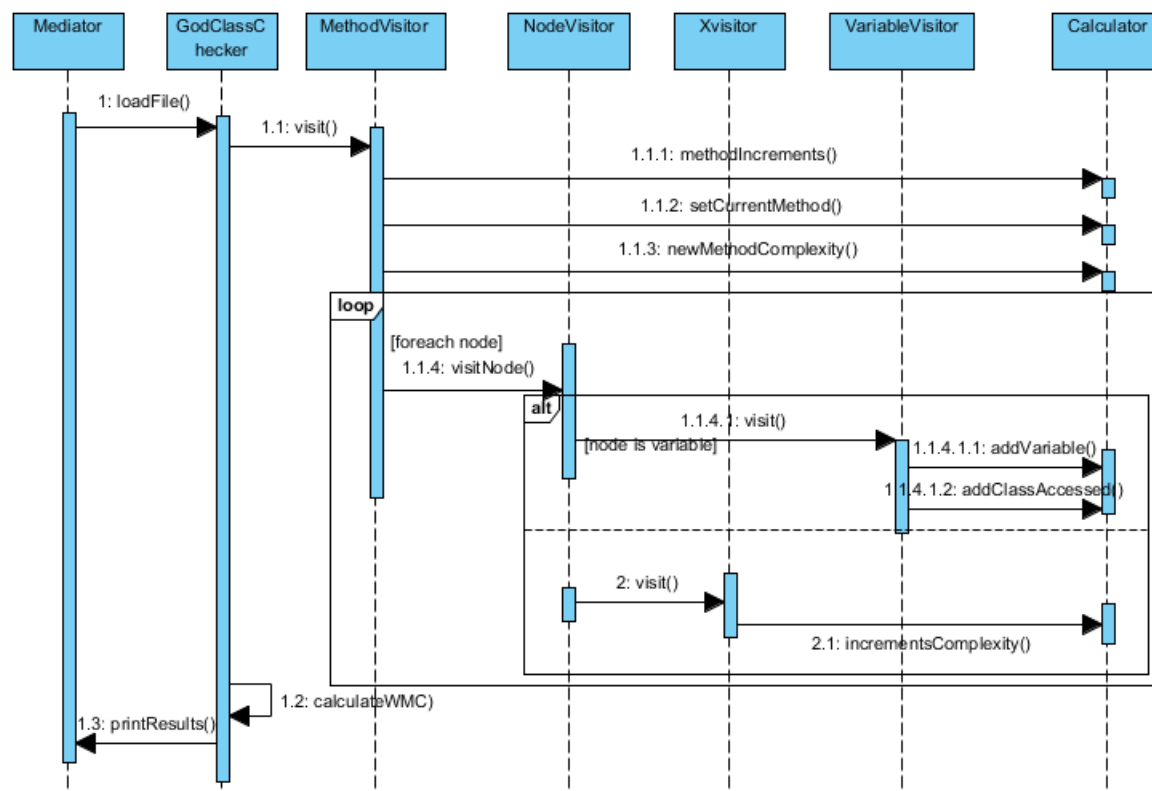


FIGURE 5 – Diagramme de séquence de la principale fonctionnalité de l'application

5 Choix de conception

Au niveau de la conception, j'ai utilisé trois patrons de conception vus au cours : Le state pattern, le mediator pattern et le visitor pattern.

Concernant l'interface graphique de l'application, j'ai utilisé un médiateur afin d'encapsuler toutes les interactions graphiques. Lorsqu'un logiciel est composé de plusieurs classes, les traitements et les données sont répartis entre toutes ces classes. Plus il y a de classes, plus le problème de communication entre celles-ci peut devenir complexe. En effet, plus les classes dépendent des méthodes des autres classes plus l'architecture devient complexe. Cela ayant des impacts sur la lisibilité du code et sa maintenabilité dans le temps. Le modèle de conception Médiateur résout ce problème. Pour ce faire, le Médiateur est la seule classe ayant connaissance des interfaces des autres classes. Lorsqu'une classe désire interagir avec une autre, elle doit passer par le médiateur qui se chargera de transmettre l'information à la ou les classes concernées. Bien qu'il n'y ait pas énormément de classes dans cette application, si par la suite on désire rajouter plus de classes faisant encore plus d'interactions, il n'y aurait pas de problème, elles passeraient toutes par ce médiateur. Par exemple pour charger un fichier java, ce sera le médiateur qui va s'en charger, il ne reste plus qu'à appeler la méthode du médiateur dans l'interface graphique.

```

1  public void loadFile ()
2  {
3      JFileChooser fileChooser = new JFileChooser(".");
4      int ret = fileChooser.showOpenDialog(parent);
5
6      if (ret == JFileChooser.APPROVE_OPTION)
7      {
8          try
9          {
  
```



```

10         File file = fileChooser.getSelectedFile();
11         this.godClassChecker = new GodClassChecker();
12         this.godClassChecker.loadFile(file);
13     }
14     catch (ParseException | IOException ex)
15     {
16         Logger.getLogger(Gui.class.getName()).log(Level.SEVERE, null,
17             ex);
18     }
19 }

```

Comme l'application peut avoir plusieurs états, j'ai élaboré une hiérarchie d'états correspondant chacun à un état dans lequel l'application peut se trouver. Par conséquent, l'interface graphique possèdera une variable membre correspondant à un état, il suffira ensuite de créer une nouvelle instance de l'état en question et d'appeler sa méthode `doAction()` qui permettra de remplir le rôle de cet état.

```

1  "
2      private void jButtonOpenActionPerformed(java.awt.event.ActionEvent evt) {
3          //GEN-FIRST:event_jButtonOpenActionPerformed
4          this.state = new LoadClassState();
5          this.state.doAction(mediator);
6
7          this.state = new ClassLoadedState();
8          this.state.doAction(mediator);
9      } //GEN-LAST:event_jButtonOpenActionPerformed

```

La méthode `doAction` de l'état `LoadClass` ne fera qu'appeler la méthode du médiateur permettant de charger la classe.

```

1 public class LoadClassState implements State
2 {
3     @Override
4     public void doAction(Mediator mediator)
5     {
6         mediator.loadFile();
7     }
8 }

```

J'ai ensuite utilisé les visiteurs pour parcourir l'ensemble du fichier. Dans un premier temps la classe «MethodVisitor» est appelée. Celle-ci appellera par après la classe «NodeVisitor» qui permettra d'utiliser correspondant au cas qui sera concerné.

```

1      // Parcours des noeuds enfants
2      List<Node> nodes = n.getBody().getChildrenNodes();
3      NodeVisitor.visitNode(nodes, calculator);

```

Dans la classe «NodeVisitor», on va passer en revue tous les types de «Statement» que l'on peut rencontrer et chacun des cas correspondra à une action différente.

```

1      if (node instanceof ForStmt)
2      {
3          System.out.println("Boucle for détectée");
4          new LoopVisitor().visit((ForStmt)node, calculator);
5      }
6      else if (node instanceof WhileStmt)
7      {
8          System.out.println("Boucle while détectée");
9          new LoopVisitor().visit((WhileStmt)node, calculator);
10     }
11     else if (node instanceof ForeachStmt)
12     {

```

```
13         System.out.println("Boucle foreach détectée");
14         new LoopVisitor().visit((ForeachStmt)node, calculator);
15     }
16     else if (node instanceof DoStmt)
17     {
18         System.out.println("Boucle do détectée");
19         new LoopVisitor().visit((DoStmt)node, calculator);
20     }
```

Par exemple, si le «statement» que l'on examine est une boucle, on va y faire le traitement suivant :

1. Récupère l'objet passé en paramètre permettant de sauvegarder les résultats
2. Si la condition est une condition binaire, on incrémente la complexité de McCabe
3. On la réincrémente une seconde fois car la boucle ajoute une branche dans l'arbre
4. On récupère ensuite l'ensemble de noeuds et on les reparcours grâce au «NodeVisitor», on repart ensuite pour un tour et ainsi de suite jusqu'à la fin de l'application

```
1  @Override
2  public void visit(ForStmt n, Object arg)
3  {
4      Calculator calculator = (Calculator)arg;
5      WeightedMethodCountCalulator wcc = calculator.getWcc();
6
7      // Test de la condition
8      if (n.getCompare() instanceof BinaryExpr)
9          wcc.incrementsComplexity();
10
11     wcc.incrementsComplexity();
12
13     List<Node> nodes = n.getBody().getChildrenNodes();
14     NodeVisitor.visitNode(nodes, calculator);
15 }
```

6 Techniques de tests

7 Cas concrets d'utilisation