UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À TROIS-RIVIÈRES

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉE

PAR
JEAN-FRANÇOIS GÉLINAS

MESURE DE LA COHÉSION DANS LES SYSTÈMES ORIENTÉS ASPECT

JUILLET 2005

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

L'auteur de ce mémoire ou de cette thèse a autorisé l'Université du Québec à Trois-Rivières à diffuser, à des fins non lucratives, une copie de son mémoire ou de sa thèse.

Cette diffusion n'entraîne pas une renonciation de la part de l'auteur à ses droits de propriété intellectuelle, incluant le droit d'auteur, sur ce mémoire ou cette thèse. Notamment, la reproduction ou la publication de la totalité ou d'une partie importante de ce mémoire ou de cette thèse requiert son autorisation.

MESURE DE LA COHÉSION DANS LES SYSTÈMES ORIENTÉS ASPECT

Jean-François Gélinas

SOMMAIRE

Le développement orienté aspect constitue un nouveau paradigme de programmation du génie logiciel. Cette approche permet de contourner certains problèmes de conception au sein des applications en améliorant la séparation des préoccupations transverses en unités modulaires appelées *aspects*. AspectJ, comme implémentation du paradigme aspect, constitue une extension intéressante de Java. En effet, les langages de programmation orientés objet imposent certaines contraintes qui rendent l'expression des préoccupations transverses difficiles. Le code relatif à ces préoccupations est propagé et dupliqué au sein de plusieurs classes dans un système orienté objet. Plusieurs métriques ont été proposées dans la littérature permettant d'évaluer les attributs de qualité des systèmes orientés objet. Toutefois, aucune de ces métriques ne prend en considération les nouvelles abstractions et les nouvelles dimensions introduites par le paradigme aspect. Ainsi, de nouvelles mesures devront être développées pour évaluer les applications aspects. La cohésion est considérée comme étant un des plus importants attributs de qualité d'un logiciel. La cohésion fait référence au degré de liaison entre les membres d'un composant logiciel. Nous proposons dans ce travail une nouvelle approche permettant d'évaluer la cohésion des aspects à partir de l'analyse de dépendances. Nous introduisons plusieurs critères de cohésion permettant de capturer les dépendances entre les membres des aspects. À partir de ces critères, nous proposons une métrique de cohésion pour les applications aspects et nous la comparons, à l'aide de différentes études de cas, aux autres approches existantes.

MEASURING COHESION IN ASPECT-ORIENTED SYSTEMS

Jean-François Gélinas

ABSTRACT

Aspect-Oriented Software Development is a promising new software engineering paradigm that gains in notoriety. It allows solving a number of software structuring problems and promotes, in particular, improved separation of crosscutting concerns into single units called *aspects*. AspectJ, as an aspect-oriented programming language, represents an interesting extension of Java. In fact, existing object-oriented programming languages suffer from a serious limitation in modularizing adequately crosscutting concerns. Many concerns crosscut several classes in an object-oriented system. Several metrics have been proposed in order to assess object-oriented software quality attributes. However, these metrics do not cover the new abstractions and complexity dimensions introduced by the aspect paradigm. As a consequence, new metrics must be developed to assess aspect-oriented systems quality attributes. Cohesion is considered as one of the most important software quality attributes. Cohesion refers to the degree of relatedness between members of a software component. We propose, in the present work, a new approach for aspect cohesion measurement based on dependence analysis. We introduce several cohesion criteria taking into account aspects' features and capturing various dependencies between their members. We also propose a new aspect cohesion metric and compare it, using several case studies, to few existing aspect cohesion metrics.

REMERCIEMENTS

Sans la contribution de plusieurs personnes, ce projet n'aurait pas été possible. Je tiens à les remercier pour leur soutien et leur aide.

Premièrement, j'aimerais particulièrement remercier mes co-directeurs Mourad Badri et Linda Badri, professeurs au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, qui ont rendu possible, grâce à leurs suggestions et leur expertise, la réalisation de ce travail. Merci d'avoir redéfini les concepts de disponibilité et de passion.

Je voudrais aussi remercier ma famille et mes amis pour leur soutien. Sans vos encouragements, rien de ceci n'aurait été possible. Un merci tout particulier à mes parents René et Diane.

J'aimerais aussi remercier tous mes collègues du laboratoire de génie logiciel ainsi que les membres du département de mathématiques et informatique pour leurs conseils et leurs suggestions.

Finalement, ce travail a été rendu possible grâce au soutient financier de divers organismes comme le CRSNG, NATEC et la fondation de l'UQTR.

TABLE DES MATIÈRES

Page

SOMMAIRE i
ABSTRACT ii
REMERCIEMENTS iii
TABLE DES MATIÈRES iv
LISTE DES TABLEAUX v
LISTE DE FIGURES vi
LISTE DES ABRÉVIATIONS ET DES SIGLES vii
CHAPITRE 1 LA PROGRAMMATION ASPECT 1
CHAPITRE 2 ÉTAT DE L'ART 8
CHAPITRE 3 LES MÉCANISMES ET LES DÉPENDANCES ASPECTS 13
CHAPITRE 4 LES CRITÈRES DE COHÉSION 21
CHAPITRE 5 MESURE DE COHÉSION POUR LA PROGRAMMATION ASPECT 24
CHAPITRE 6 IMPLÉMENTATION DES MÉTRIQUES DE COHÉSION ASPECT 28
CHAPITRE 7 LES ÉTUDES DE CAS 32
CHAPITRE 8 ÉTUDE EXPÉRIMENTALE 46
CHAPITRE 9 LIMITES DE L'APPROCHE ET TRAVAUX FUTURS 52
CHAPITRE 10 CONCLUSION 57
BIBLIOGRAPHIE 59

LISTE DES TABLEAUX

Page

Tableau I Exemples de pointcuts avec leurs significations. 16
Tableau II Mesures de cohésion aspect pour le système Atrack. 47

LISTE DE FIGURES

Page

Figure 1 Code épars et dispersé selon le patron Observateur. 2
Figure 2 Patron observateur implémenté avec AspectJ (aspect abstrait)....5
Figure 3 Patron observateur implémenté avec AspectJ (aspect concret) 6
Figure 4 Syntaxe AspectJ....14
Figure 5 Dépendances entre les membres d'un aspect. 20
Figure 6 Relation UAm (Modules partageant des attributs). 22
Figure 7 Relation UMm (Modules appelant le même module). 23
Figure 8 Interactions entre les membres d'un aspect. 25
Figure 9 Exemple de graphe GD. 25
Figure 10 Représentation de NCM. 26
Figure 11 Représentation UML de l'implémentation des métriques. 29
Figure 12 Code source utilisé par Zhao et Xu....32
Figure 13 Interactions entre les membres de l'aspect. 33
Figure 14 Connexions entre les modules de l'aspect. 33
Figure 15 Code source étendu pour l'aspect PointShadowProtocol. 35
Figure 16 Interactions entre les membres de l'aspect....36
Figure 17 Connexions entre les membres de l'aspect. 36
Figure 18 Représentation de NCM. 38
Figure 19 Interactions entre les membres du patron Observateur....40
Figure 20 Interactions entre les membres du patron Médiateur....42
Figure 21 Interactions entre les membres du patron Commande....44
Figure 22 Code source de l'aspect LogManager. 49
Figure 23 Code source de l'aspect Authentication....50
Figure 24 Code source de l'aspect Observing....51
Figure 25 Code source de l'aspect AjeeLogManager. 51

LISTE DES ABRÉVIATIONS ET DES SIGLES

ACoh	Aspect Cohesion (Mesure de cohésion pour les systèmes aspect)
AOP	Aspect-Oriented Programming (Programmation Orientée aspect)
AOSD	Aspect-Oriented Software Development (Développement de Logiciel Orienté Aspect)
LCOM	Mesure de cohésion pour les systèmes objets (Lack of Cohesion in Methods)
LCOO	Mesure de cohésion pour les systèmes aspects (Lack of Cohesion in Operations)
NCM	Nombre de membres connectés (*Number of Connected Member*)
$\Gamma(A)$	Mesure de cohésion pour les systèmes aspects

CHAPITRE 1

LA PROGRAMMATION ASPECT

La programmation orientée aspect constitue un nouveau paradigme de développement du génie logiciel. D'abord considérée comme un effort de recherche, cette méthode de développement gagne peu à peu en notoriété [Sab04]. Le nombre croissant d'implémentations du paradigme [Ask05, Asp05, Ass05] démontre l'intérêt de la communauté pour cette nouvelle approche. Le développement orienté aspect a pour objectif de contourner certaines limites des modèles actuels de programmation en fournissant un nouveau modèle de conception mieux adapté à la transposition des problématiques du domaine.

L'idée principale de la programmation aspect vient du fait que les paradigmes de programmation actuels rendent difficile l'expression de certaines relations présentes dans un système. Lopez définit que : « plusieurs problématiques relatives au développement des systèmes sont reliées de telle manière que la relation "uses" ne permet pas de les capturer adéquatement» [Lop97]. La programmation aspect se présente comme une technique de factorisation efficace pouvant être appliquée durant les divers cycles de la durée de vie d'un logiciel: spécification des pré-requis, conception, implémentation, phases de test. Grâce à l'introduction de nouveaux mécanismes et de nouvelles abstractions, il est possible de franchir les limites imposées par les paradigmes actuels. L'AOP (*Aspect Oriented Programming*) permet de modéliser efficacement certaines problématiques, au même titre que la programmation objet permet d'exprimer adéquatement des notions comme l'encapsulation et l'héritage [Ast05].

L'implémentation du paradigme aspect peut être symétrique ou asymétrique, dépendamment du fait que l'aspect constitue la composante de base ou non. L'implémentation asymétrique du paradigme aspect s'appuiera alors sur un paradigme sous-jacent : procédural, objet ou autre. Dans ce cas, il y aura donc une co-existence entre deux paradigmes. AspectJ [Asp05], comme implémentation du

paradigme aspect, constitue une extension intéressante de Java [Jav05]. Dans ce cas, le paradigme aspect représente une extension du paradigme objet visant à améliorer la qualité des applications. Bien que la technologie objet ait permis de régler de nombreux problèmes, elle atteint ses limites lors de l'expression des relations entre les composantes. Dans plusieurs situations, les techniques objet ne sont pas adéquates pour capturer toutes les problématiques découlant des décisions de conception [Kic97]. Idéalement, dans une application, chaque tâche devrait être la seule responsabilité d'une classe ou d'une collaboration étroite entre un petit nombre de classes rassemblées au sein d'un groupe logique. Malheureusement, avec la programmation objet, certaines de ces tâches sont dispersées dans plusieurs classes. On parle alors de préoccupations transverses. Ces préoccupations transverses transcendent la hiérarchie des systèmes tels que mentionnés par Gregor Kiczales dans [Kic04]. Dans un système orienté objet, le code relatif à ces préoccupations sera dispersé et dupliqué au sein de plusieurs classes.

```
public class Point {

  int x,y;
  ArrayList observers = new ArrayList();

  public void setX(int x) {
    this.x = x;
    this.informObservers();
  }

  public void setY(int y) {
    this.y = y;
    this.informObservers();
  }

  public void setXY(int x, int y) {
    this.x = x;
    this.y = y;
    this.informObservers();
  }

  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
         it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

```
public class GuiElement ... {

  Color color;
  ArrayList observers = new ArrayList();

  public void setColor(Color color) {
    this.color = color;
    this.informObservers();
  }

  .
  .
  .
  .
  .
  .
  .
  .
  .
  .

  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
         it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

Figure 1 Code épars et dispersé selon le patron Observateur.

La figure 1 présente une implémentation possible du patron de conception Observateur [Gam95]. Les deux classes implémentent le code nécessaire pour représenter adéquatement le patron de conception selon le paradigme objet. Toutefois, les deux classes *Point* et *GuiElement* contiennent du code épars, dupliqué et entremêlé. La préoccupation relative à la mise à jour des observateurs, lors d'une modification du sujet, est diluée au sein des deux classes *Point* et *GuiElement*. En effet, et à titre d'exemple, le fait d'invoquer *this.informObservers()* à trois reprises dans la classe Point et à un endroit dans la classe *GuiElement* est problématique. De plus, les méthodes relatives à la gestion des mécanismes du patron (*attachObservers, detachObservers, informObservers*) sont dupliquées au sein des deux classes. Finalement, la présence d'une référence *observer* dans les deux classes est aussi problématique. Conséquemment, ces classes seront difficiles à maintenir, à comprendre et à réutiliser. La communication, la synchronisation, le traçage d'exceptions sont tous des exemples d'applications susceptibles d'être affectées positivement par la technologie aspect.

Dans notre exemple, présenté à la figure 1, un programmeur pourrait contourner ce problème en utilisant les mécanismes d'héritage pour effectuer une factorisation des mécanismes du patron observateur au sein d'une super classe. Les classes *Point* et *GuiElement* spécialiseraient cette classe parent afin d'obtenir le résultat escompté. Toutefois, il y a plusieurs raisons déconseillant cette utilisation de l'héritage. L'une d'elle étant que les relations d'héritage visent à favoriser la réutilisation de code. Les relations d'héritage devraient exprimer des relations de sous-typage entre les composantes. Ce n'est pas le cas de cet exemple. L'utilisation de l'héritage dans ce cas particulier n'est donc pas recommandée. La programmation aspect permet d'offrir une alternative intéressante à cette problématique. En effet, cette nouvelle technique de développement permet d'améliorer la séparation des préoccupations transverses et de représenter le code épars et entremêlé dans une entité modulaire appelée : aspect. Le développement aspect permet de pallier à ces lacunes en modularisant adéquatement ces préoccupations. La modularité est une propriété d'un système qui a été décomposé en un ensemble de parties cohésives et faiblement

couplées. Elle représente l'une des propriétés les plus importantes et souhaitables pour les systèmes logiciels. Kiczales et d'autres chercheurs ont publié plusieurs papiers décrivant et illustrant des exemples de ce que constitue une préoccupation transverse. Pour en savoir davantage à ce propos, nous invitons le lecteur à se référer à ces diverses publications : [Kic97], [Lop97], [Men97] et [Aks98].

Le paradigme aspect propose une alternative élégante et efficace pour solutionner cette problématique. La figure 2 présente une implémentation possible du patron observateur à l'aide d'aspectJ [Gam95]. L'objectif de cette introduction n'est pas de présenter les détails relatifs aux mécanismes aspect ; nous présenterons chacun de ces éléments dans les chapitres subséquents. Plutôt que d'être dispersés et dupliqués au sein de plusieurs classes, les mécanismes du patron sont regroupés dans une entité modulaire : ObserverProtocol. Cette entité est représentée comme un aspect abstrait et contient tous les mécanismes relatifs à la relation *(addObservers, removeObservers, updateObservers)*. Cet aspect pourra être spécialisé (étendu) pour chaque instance de la relation Observateur; un aspect concret viendra spécifier, à l'aide des spécifications d'interfaces, les classes jouant le rôle de sujet et d'observateur. La figure 3 présente un exemple d'aspect concret représentant la relation « sujet-observateurs » entre la classe *Point* et la classe *GuiElement*. Dans la figure 3, on peut remarquer que le seul rôle de l'aspect concret est de spécifier d'une part, que la classe *Point* implémente l'interface *Subject*, que la classe *GuiElement* implémente l'interface Observer, d'autre part, de fournir une implémentation (les actions à effectuer) lors de la mise à jour des observateurs. Le code épars est donc extrait des classes et il est regroupé sous forme d'un patron de conception réutilisable : ObserverProtocol. Les classes, qui constituent les composantes de base de la relation, ne savent pas (au sens où aucun mécanisme n'est défini) qu'elles participent à cette relation puisque c'est l'aspect qui assure la coordination sujet-observateurs. Ceci devrait avoir pour effet de faciliter la réutilisation, la maintenance et la compréhension des composantes impliquées dans cette relation.

```
 1 public abstract aspect ObserverProtocol {
 2
 3     protected interface Subject  { }
 4
 5     protected interface Observer { }
 6
 7     private WeakHashMap perSubjectObservers;
 8
 9     protected List getObservers(Subject s) {
10         if (perSubjectObservers == null) {
11             perSubjectObservers = new WeakHashMap();
12         }
13         List observers = (List)perSubjectObservers.get(s);
14         if ( observers == null ) {
15             observers = new LinkedList();
16             perSubjectObservers.put(s, observers);
17         }
18         return observers;
19     }
20
21     public void    addObserver(Subject s, Observer o) {
22         getObservers(s).add(o);
23     }
24
25     public void removeObserver(Subject s, Observer o) {
26         getObservers(s).remove(o);
27     }
28
29     protected abstract pointcut subjectChange(Subject s);
30
31     after(Subject s): subjectChange(s) {
32         Iterator iter = getObservers(s).iterator();
33         while ( iter.hasNext() ) {
34             updateObserver(s, ((Observer)iter.next()));
35         }
36     }
37
38     protected abstract void updateObserver(Subject s, Observer o);
39 }
```

Figure 2 Patron observateur implémenté avec AspectJ (aspect abstrait).

```
1  public aspect PointGuiElementObserver extends ObserverProtocol{
2
3    declare parents: Point  implements Subject;
4
5    declare parents: GuiElement implements Observer;
6
7    protected pointcut subjectChange(Subject s):
8        call(void Point.setColor(Color)) && target(s);
9
10 // ...
11
12   protected void updateObserver(Subject s, Observer o) {
13       o.display("Les elements graphiques ont ete mise a jour");
14   }
15 }
16
```

Figure 3 Patron observateur implémenté avec AspectJ (aspect concret).

Au fil des années, l'industrie du développement logiciel est devenue extrêmement sensible aux notions de qualité. De nombreuses études ont été publiées afin de prédire et de quantifier la qualité des applications développées selon le paradigme objet [Bas96, Bie95, Chi94, Hen95, Hit95]. Même s'il s'agit d'une nouvelle technique de programmation, le paradigme aspect ne nous permet pas d'ignorer les principes établis du génie logiciel. Ces principes visent constamment à améliorer, entre autres, la maintenance, la réutilisation et la complexité des composantes au sein des applications. Toutefois, les principes permettant d'évaluer la qualité d'une application objet ne peuvent pas être transposés tels quels au paradigme aspect. Plusieurs métriques ont été développées afin de quantifier les attributs de qualité (couplage, cohésion, complexité, etc) d'une application orientée objet [Bad95, Bad04, Bie95, Chi94, Hen95, Hit95]. Les métriques de qualité sont devenues un outil essentiel permettant aux développeurs et aux gestionnaires d'améliorer le processus de développement logiciel [Pre01]. Toutefois, les métriques orientées objet actuelles ne capturent pas toutes les interactions entre les mécanismes de l'aspect [San03, Zac03, Zha03, Zha04]. Le développement orienté aspect introduit de nouvelles abstractions et de nouvelles dimensions au principe de développement logiciel. De nouvelles mesures tenant compte de ces dimensions devraient être développées pour mesurer les attributs de qualité d'une application aspect. C'est dans ce contexte et dans un objectif d'assurance qualité que s'insère le présent travail de recherche.

L'objectif principal consiste à développer une nouvelle métrique relative à la cohésion des systèmes orientés aspect.

CHAPITRE 2

ÉTAT DE L'ART

Le processus de développement de logiciel est sans contredit un processus complexe; le produit final résulte d'un enchaînement de processus d'analyse, de conception, de développement et de test. À chaque étape, il est important de suivre une méthodologie bien définie pour s'assurer de la qualité du produit final. Longtemps, le processus de gestion a été inefficace; le processus de développement de logiciel n'était pas supporté par des mesures fiables et bien définies permettant de guider et d'évaluer son développement. Selon le principe que l'on ne peut contrôler ce qu'on ne peut mesurer, de nombreuses propositions de métriques pour les systèmes orientés objets (entre autres) ont vu le jour. Ces propositions ont pour objectif de quantifier les attributs de qualité des applications afin d'améliorer certains aspects: la planification, les coûts de production, la réutilisabilité, la complexité, etc. L'objectif étant de produire un logiciel efficace et de qualité au moindre coût.

Ces diverses mesures, regroupées sous l'appellation de métriques de qualité ainsi que modèles de prévision, sont utilisées depuis un certain temps [Wol74, Per81]. Cette démarche avait pour objectif de mieux comprendre la crise du logiciel qui sévissait dans les années 70 et d'y apporter des solutions. Toutefois, dans le contexte du développement actuel, les métriques sont rarement utilisées de manière méthodique et structurée dans les processus de développement. Certaines études indiquent clairement les bénéfices rattachées à leur utilisation; à court terme (sur un projet actuel) et à long terme (amélioration de la productivité sur les projets à venir) [Gra87]. Ces mesures ont pour objectif de quantifier les attributs internes des applications. Plusieurs suites de métriques ont été proposées [Bad95, Bad04, Bie95, Chi94, Hen96, Hit95] permettant de quantifier diverses propriétés (comme le couplage, la cohésion, la complexité, etc.) d'une application. Certaines de ces métriques ont été validées expérimentalement [Bas96, Bri98, Hen95, Li93]. D'autres études émettaient certaines réserves quant à la validité des propriétés mesurées

[Cha00, Hit95]. Chose certaine, le débat sur les indicateurs de la qualité des applications est loin d'être clos.

Yourdon et Constantine ont été les premiers à introduire le concept de cohésion dans les applications. Ils ont définit la cohésion comme étant l'extension des relations fonctionnelles entre les éléments d'un module [You79]. En effet, la cohésion est un objectif à considérer tout au long du processus de conception [Lar04]. Une forte cohésion au sein d'un composant logiciel est une propriété désirable. Il est reconnu que la mesure de cohésion d'une composante reflète sa qualité structurelle. Ainsi, une forte cohésion au sein d'une composante est une propriété souhaitable. Des composants présentant une forte cohésion auront tendance à être plus réutilisables et plus faciles à maintenir [Bie95, Bri98, Cha00, Li93]. La cohésion fait référence au degré de liaison entre les membres d'un composant. Grady Booch mentionne qu'un composant possédant des membres travaillant conjointement à l'implémentation d'un comportement bien défini présentera un haut niveau de cohésion fonctionnelle [Boo93]. Le degré de cohésion d'un composant sera élevé lorsque celui-ci implémente une fonction logique singulière. Il sera difficile de diviser (en diverses parties) un tel composant.

Plusieurs métriques ont été proposées dans la littérature afin de mesurer la cohésion des classes dans les systèmes objets [Bad95, Bad04, Bie95, Chi94, Hit95, Li93]. La majorité des métriques de cohésion OO étaient au départ basées sur l'usage ou le partage des variables d'instance [Kab00]. Ces métriques représentent la cohésion en termes de connexions entre les membres d'une même classe. Certaines se limitent au dénombrement du nombre de méthodes utilisées par les méthodes ou le nombre de variables d'instance partagées entre les méthodes. Récemment, d'autres approches ont vu le jour, certaines tenant compte des relations étroites entre les dépendances de méthodes. La plupart des ces métriques ont été expérimentées et discutées dans la littérature [Bas96, Bri98, Hen95, Li93].

La cohésion peut être utilisée pour identifier les composants mal conçus. Un composant présentant une faible cohésion peut être formé de membres disparates et

non reliés. Plusieurs responsabilités non reliées sont généralement affectées à de tels composants. Les éléments de conception présentant une faible cohésion devraient être considérés pour une éventuelle restructuration. La programmation aspect permet de séparer les préoccupations transverses et d'adresser certains de ces problèmes. On peut donc s'attendre à une augmentation du niveau de cohésion au sein des classes puisque le code épars et transverse est implémenté d'une façon modulaire dans un aspect. Par ailleurs, nous croyons que l'assignation des responsabilités d'un aspect devrait suivre les principes éprouvés du génie logiciel. Un aspect, au même titre qu'une classe, doit exprimer un rôle (ou partie d'un rôle), une préoccupation transverse, d'une manière cohésive. Au même titre qu'une classe à laquelle on aurait attribué plusieurs rôles disparates, un aspect implémentant plusieurs rôles disparates présentera une faible cohésion. Un tel aspect sera probablement difficile à comprendre, à tester, à réutiliser et à maintenir.

Récemment, de nouvelles études ont été publiées dans un premier effort visant à quantifier la qualité d'un aspect [Duf03, Gel04, Gel05, San03, Zac03, Zha03, Zha04]. Zhao et Xu ont proposé, à notre connaissance, la première approche permettant d'évaluer la cohésion d'un aspect [Zha04]. Leur approche est basée sur un modèle défini par un ensemble de graphes de dépendance extraits à partir d'une application aspect. Selon leur approche, la cohésion est définie comme étant le degré de liaison entre les attributs et les modules de l'aspect. Zhao et Xu présentent deux façons de mesurer la cohésion d'un aspect basée sur les dépendances inter-attributs (γ_a), inter-modules (γ_m) ou module-attribut (γ_{ma}). Une première approche suggère que chaque mesure ($\gamma_a, \gamma_m, \gamma_{ma}$) représente un champ caractérisant un attribut de cohésion. Ainsi, la cohésion pour un aspect donné A est définie par un triplet $\Gamma(A) = (\gamma_a, \gamma_m, \gamma_{ma})$. Une autre approche suggère que chaque facette peut être intégrée comme un tout à l'aide de paramètres β pour représenter la mesure de cohésion. La cohésion d'un aspect est exprimée comme étant :

$$\Gamma(A) = \begin{cases} 0 & n = 0 \\ \beta^* \gamma_m & k = 0 \text{ et } n \neq 0 \\ x & \text{Autres} \end{cases}$$

Où $x = \beta_1 * \gamma_a + \beta_2 * \gamma_m + \beta_3 * \gamma_{ma}$, k étant le nombre d'attributs et n le nombre de modules dans l'aspect A.

Selon les auteurs, le choix de la méthode d'évaluation (selon les triplets ou selon un tout) et/ou la pondération des paramètres β1, β2, et β3 est arbitraire. Aucune indication n'est fournie quant au choix de l'approche. De plus, certaines définitions de relation et leur considération dans la mesure de la cohésion sont difficiles à capturer. En général, cette approche suggère une approche complexe pour la mesure de la cohésion aspect qui peut être problématique dans un contexte de développement réel. La génération des graphes de dépendances est un processus lourd et la pondération des paramètres peut être problématique puisqu'un jugement arbitraire est introduit. L'utilisation d'une telle métrique dans un contexte de développement d'applications d'envergure semble donc problématique.

Sant' Anna et al. ont proposé dans [San03] une extension de la métrique LCOM (*Lack of Cohesion in Methods*) développée par Chidamber et Kemerer pour les systèmes orientés objet [Chi94]. La métrique proposée LCOO (*Lack of Cohesion in OPeration*) mesure la quantité de paires de méthode/advice qui n'accèdent pas aux mêmes variables d'instance. Cette métrique mesure le manque de cohésion au sein d'un composant. Selon cette approche, si un composant C_i possèdent n opérations (méthodes et advice) $O_i, \ldots, O_n$ alors $\{I_j\}$ est défini comme l'ensemble des variables d'instance utilisées par l'opération O_j. Définissons $|P|$ comme étant le nombre d'intersections nulles entre les ensembles de variables d'instance. Définissons $|Q|$ comme étant le nombre d'intersections non-nulles entre les ensembles de variables

d'instance. Le manque de cohésion, selon cette approche, est alors définit comme étant :

$LCOO = |P| - |Q|$, if $|P| > |Q|$,

$LCOO = 0$ autrement.

Une forte valeur de LCOO, selon Sant' Anna et al., indique une disparité dans les fonctionnalités fournies par l'aspect. La métrique LCOM, sur laquelle la métrique LCOO est basée, a été largement expérimentée et discutée dans la littérature. Elle souffre de plusieurs problèmes discutés, entre autres, par B. Henderson-Sellers dans [Hen95]. Le fait que cette métrique ne soit pas exprimée en fonction d'un nombre maximal de connexions engendre certaines difficultés dans l'interprétation des résultats. Nous croyons que la métrique LCOO souffre des mêmes problèmes.

Pour ces raisons, nous croyons que les métriques de cohésion actuellement proposées ne permettent pas de capturer adéquatement les nouvelles dimensions apportées par les aspects. Plusieurs abstractions et plusieurs dimensions propres à l'aspect sont laissées pour compte. Nous présentons, dans le prochain chapitre, les dépendances possibles entre les membres d'un aspect. Ces dépendances constituent la base de notre approche. A partir de ces dépendances, une mesure de cohésion pour les aspects est proposée.

CHAPITRE 3

LES MÉCANISMES ET LES DÉPENDANCES ASPECTS

L'aspect est une extension du concept de classe en Java ; il hérite donc de certaines de ses propriétés. Les concepts de base de notre approche seront illustrés à l'aide d'AspectJ [Asp05]. AspectJ constitue une extension asymétrique du paradigme aspect appliquée à Java [Jav05]. La programmation aspect n'est pas limitée à AspectJ, comme la programmation objet n'est pas limitée à Java ; il s'agit d'une implémentation possible du paradigme. AspectJ utilise une approche asymétrique, c'est à dire que les aspects ne constituent pas la composante de base des systèmes. L'aspect se situe à un niveau distinct des classes; chaque aspect étant par défaut un singleton qui n'est pas instancié à partir de l'opérateur *new*.

L'approche que nous proposons est facile à implémenter. Elle se base sur une analyse statique du code. Les concepts que nous proposons sont applicables à la majorité des implémentations actuelles du paradigme aspect. AspectJ introduit, comme les autres environnements Aspect, plusieurs nouvelles abstractions et mécanismes : l'aspect, les point de jointure (joinpoint), les point de coupure (pointcuts), les introductions et l'advice. Ces éléments, que nous appelons les membres de l'aspect, permettent à un aspect d'exprimer une préoccupation qui transverse plusieurs classes de base. La syntaxe pour décrire un aspect est similaire à la syntaxe Java. La figure 4 présente, sous forme syntaxique, un exemple des nouveaux mécanismes introduits par AspectJ.

```
1  aspect <aspect_name> {
2
3  // Les membres suivants ont la même structure qu'une
4  //déclaration normale d'une classe en Java
5
6  <variable declarations>...
7  <method declarations>...
8
9  // Les membres suivants sont spécifiques aux aspects
10
11 Pointcut <designator>
12
13 advice <designator> {
14 [static] before {
15
16 // Code à exécuter avant le début de la méthode
17
18 }
19 [static] after {
20
21 // Code à exécuter après le début de la méthode
22
23 }
24
25 introduce <class>.<variable declaration>; ...
26 introduce <designator> {
27
28 // corps de méthode
29 |
30 }
31 } // fin <aspect_name>
32
```

Figure 4 Syntaxe AspectJ.

Nous allons présenter dans ce qui suit un survol des différents mécanismes fournis par AspectJ. L'aspect constitue la construction de base permettant d'exprimer une préoccupation. L'aspect peut définir, comme une classe, des attributs et des méthodes et, sous certaines restrictions, des relations de généralisation ou de spécialisation. Un aspect définit des points de coupure et des advices pour former une unité de recoupement. Les membres de l'aspect définissent les règles d'intégration. Un aspect est similaire à une classe Java; il contient des champs et des méthodes et peut étendre d'autres classes ou d'autres aspects. Un aspect se déclare comme une classe Java en remplaçant le mot clé class par aspect.

Le concept de la programmation aspect repose sur le modèle dynamique des points de jointure. Les points de jointure désignent des points précis dans le flux de contrôle d'un programme et constituent le modèle autour duquel le paradigme aspect

s'articule. Il s'agit de la seule structure pouvant être définie à l'extérieur des aspects; c'est-à-dire que les points de jointure peuvent être définis au sein des classes. AspectJ permet de définir des points de jointure sur les éléments suivants :

- un appel de méthode ou de constructeur
- l'exécution d'une méthode ou d'un constructeur
- l'accès en lecture ou écriture d'un champ
- l'exécution d'un bloc "catch" qui traite une exception Java
- l'initialisation d'un objet ou d'une classe (c'est à dire des membres statiques d'une classe).

À partir des points de jointure, l'aspect peut identifier, à l'aide des points de coupure, des endroits d'intérêt dans l'exécution d'un programme. Les points de coupure correspondent à la définition syntaxique d'un sous-ensemble de points de jointure. Éventuellement, certaines valeurs dans le contexte d'exécution seront exposées afin de permettre à l'aspect de représenter une préoccupation transverse. Voici, dans une forme simple, une définition de point de coupure :

pointcut <nom du pointcut> : call (<signature de méthode>)

Ce point de coupure fait référence à tous les points de jointure qui correspondent à l'appel (call) d'une méthode dont la signature correspond à <signature de méthode>. Ainsi, tous les appels correspondant à <signature de méthode> seront interceptés. Les point de coupure peuvent être composés de plusieurs points de jointure. Dans la suite du programme, on pourra y référer en utilisant le nom <nom du pointcut> : pour effectuer une composition de plusieurs points de coupure, pour y référer à l'aide d'un advice ou possiblement les deux. Considérons le point de coupure suivant :

pointcut appelPaiement(Information info): call(void ProcessusTransaction.effectuerPaiement(Information)) && args(info);

Le nom du point de coupure est *appelPaiement*. Le contexte *info* est associé au point de coupure et pourra être utilisé par le ou les advices correspondants, s'il y a lieu. Dans ce cas particulier, le point de jointure est de type appel de méthode *(call)* et la signature est *void ProcessusTransaction.*effectuerPaiement (Information). *Void* indique simplement que la méthode capturée doit posséder un type de retour correspondant à *void ;* c'est à dire que la signature de la méthode ne possède pas de valeur de retour. *ProcessusTransaction.effectuerPaiement* spécifie le nom de la méthode qui doit être capturée dans la classe *ProcessusPaiement.* Il spécifie par la suite *(Information),* que la méthode capturée prend en argument un objet de type *Information.* Finalement, *args(info)* signifie que le point de coupure expose l'argument info afin que les advices puissent effectuer des opérations sur ce contexte. Quelques exemples de points de coupure particuliers sont présentés dans le tableau I suivant:

Tableau I Exemples de pointcuts avec leurs significations.

call(void MaClasse.maMethode(..))	Appel de maMethode de MaClasse prenant un nombre quelconque d'arguments, avec comme type de retour void et n'importe quel droit d'accès (public, private, etc...)
call(MaClasse.maMethode(..))*	Appel de maMethode() de MaClasse prenant un nombre quelconque d'arguments, avec n'importe quel type de retour.
call(MaClasse.new(String))	Appel du constructeur de MaClasse avec un argument de type String
execution (void MaClasse.maMethode())	Exécution de maMethode() de MaClasse retournant void.
set(int MaClasse.x)	Ecriture dans le champ x de type entier (int) de la classe MaClasse
get(String Toto.nom)	Lecture du champ nom de type String de la classe Toto

Handler(IOException)	Exécution d'un bloc "catch" (en java) qui traite une exception de type IOException

Nous pouvons compléter tous ces exemples avec des contraintes supplémentaires sur le contexte d'exécution de ces points de jointure. Par exemple, la définition :

pointcut appelMethode(Foo entree): call(SuperFoo.methode())

peut être complétée par : *&& target(entree);* .

Dans ce cas, l'appel *methode()* doit être effectué sur un objet du même type que *entree*, c'est à dire de type *Foo*. Ce genre de contrainte impliquerait donc que *Foo* fasse partie d'une relation de spécialisation; soit une sous-classe de *SuperFoo*. La même contrainte peut être appliquée sur les points de coupure définissant l'exécution d'une méthode. Dans ce cas, le mot réservé *this* serait utilisé à la place de target. Finalement, l'appel *(call)* et l'exécution *(execution)* d'une méthode sont deux concepts distincts. En effet, dans une application utilisant des mécanismes de synchronisation, une méthode peut être appelée sans être immédiatement exécutée. Le code suivant permet de capturer l'exécution d'une méthode.

pointcut appelMethode(Foo entree): execution (SuperFoo.methode()) &&
this(entree);

Après avoir défini un modèle de point de coupure, il est possible d'augmenter ou de modifier le comportement des classes à l'aide du concept d'advice. Un advice est une abstraction semblable à une méthode; il définit du code à exécuter lorsqu'un point de jointure est atteint. Les points de coupure sont utilisés dans la définition d'un advice (sa signature). Il y a trois types d'advice : les *"before advices"*, les *"around advices"* et les *"after advices"*. Comme leur nom l'indique, les *"before advices"* s'exécutent avant que les points de jointure ne soient exécutés; les *"around advices"* permettent d'exécuter du code avant et après les points de jointure (ou

carrément remplacer l'exécution d'une méthode); et les *"after advices"* s'exécutent uniquement après l'exécution des points de jointure. L'avantage des advices vient du fait qu'ils peuvent avoir accès à certaines valeurs du contexte exposées par le point de coupure.

Prenons un exemple avec l'advice suivant :

```
before(Information entree): appelTransaction (entree) {
log.enregistrer("Tentative transaction:" + entree);
}
```

Ce *"before advice"* concerne tous les points de jointure définis au sein du point de coupure *appelTransaction.* Avant d'appeler les méthodes désignées par *appelTransaction,* on fait appel à *log.enregistrer(...)* qui va enregistrer l'argument *entree* de ces méthodes.

De la même manière, on aurait pu définir un *"after advice"* qui aurait enregistré les informations après les appels de méthode. Il aurait suffit pour cela de remplacer le mot réservé *before* par *after* dans la définition ci-dessus. Finalement, dans l'exemple présenté ci-dessous, l'advice de type *around* capture tous les appels de *Connection.close()* et affiche le temps de l'horloge système avant et après ceux-ci :

```
void around(Connection conn) :
call(Connection.close()){
System.out.println(System.currentTimeMillis());
proceed();
System.out.println(System.currentTimeMillis());
}
```

On remarque que si on n'avait pas fait appel à *proceed()* dans l'advice, l'appel de méthode *Connection.close* n'aurait pas été effectué. Ce mécanisme permet donc aussi de supprimer l'exécution de certains blocs de code.

De plus, les mécanismes d'introduction définissent comment AspectJ modifie la structure statique d'un programme, c'est-à-dire la relation entre les composants et ses membres. Les points de coupure et l'advice affectent dynamiquement le flux du programme, alors que l'introduction affecte la hiérarchie des classes. Comme nous pouvons le remarquer, le paradigme aspect introduit un modèle relativement complexe permettant de capturer des points d'intérêt. Plusieurs nouveaux types de relation entre les membres sont introduits par le paradigme aspect. Pour plus d'information sur AspectJ, le lecteur peut se référer à [Asp05].

De la même façon que pour les membres d'une classe, une forte liaison doit exister entre les membres d'un aspect. Comme nous l'avons présenté, un aspect est défini par ses attributs, ses méthodes, ses introductions, ses points de coupure et l'advice. Pour exprimer un rôle, plusieurs interactions (directes et indirectes) entre les membres de l'aspect sont nécessaires. Afin de développer adéquatement ce rôle, les membres de l'aspect se doivent d'interagir entre eux. Chacune de ses constructions apporte une dimension supplémentaire permettant de modéliser le concept à représenter (une préoccupation). La figure 5 présente schématiquement les diverses relations entre les membres d'un aspect. L'utilisation de point de coupure est intimement liée au concept d'advice. Le fait qu'on puisse définir des point de coupure anonymes semble renforcer cette position. Les dépendances de point de coupure seront indirectement prises en compte par les advice. Nous croyons que les dépendances basées sur l'exposition de contexte (fournies par les point de coupure, par exemple) sont d'un autre ordre et sont plus rattachées au concept de couplage.

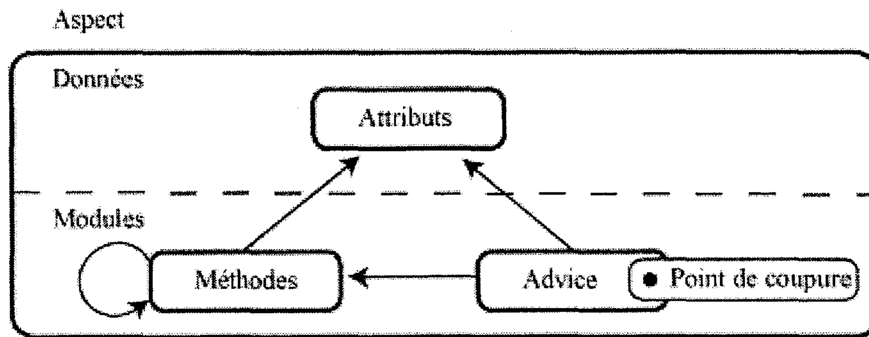

Figure 5 Dépendances entre les membres d'un aspect.

Le chapitre suivant présente les interactions entre les membres de l'aspect. Au cours de ce chapitre, nous discutons de l'apport de chacun de ses membres et de la nature conceptuelle de l'aspect au sein des applications.

CHAPITRE 4

LES CRITÈRES DE COHÉSION

Un aspect peut être représenté par ses données (attributs) et ses modules (méthodes et advice). Dans ce chapitre, nous définissons les connexions pouvant exister entre les membres d'un aspect. À ce titre, nous présentons les diverses dépendances, directes et indirectes, pouvant exister entre les attributs et les modules. Les interactions entre les membres de l'aspect sont situés à deux niveaux distincts : Modules-Données et Modules-Modules. À partir de ces relations, une mesure de cohésion pour les aspects sera définie [Gel04].

Considérons un aspect $Aspect_i$. Définissons $A = \{ A_1, A_2 \ldots A_a\}$ comme étant l'ensemble de ses attributs, $M = \{ M_1, M_2 \ldots M_m\}$ comme étant l'ensemble de ses modules. Les critères de connexion Modules-Données définis dans ce qui suit, sont basés sur l'utilisation des attributs. Ils permettent de capturer les dépendances entre les modules. Les critères de connexion Modules-Modules sont basés sur les invocations de méthode. Ils permettent, en particulier, de capturer les dépendances entre les modules qui ne partagent pas d'attributs communs, mais qui appellent une ou plusieurs méthodes qui n'accèdent à aucun attribut en commun de l'aspect.

Critère de connexion Modules-Données

Définition 1

Définissons UA_{Mi} comme l'ensemble des attributs utilisés directement ou indirectement par un module M_i. Un attribut est utilisé directement par un module M_i si l'attribut apparaît dans son corps. Un attribut est utilisé indirectement par un module M_i s'il est utilisé directement par un autre module appelé directement ou indirectement par M_i. Il y a m ensembles $UA_{M1}, UA_{M2}, \ldots UA_{Mm}$. Deux modules Mi et Mj d'un aspect sont directement reliés par la *relation* UA_M si $UA_{Mi} \cap UA_{Mj} \neq$

$\varnothing$. Ceci implique qu'il y a au moins un attribut partagé (directement ou indirectement) par les deux modules.

Dans l'exemple illustré par la figure 6, nous pouvons remarquer que les modules M_1 et M_2 sont directement reliés par la relation UA_M. Ils partagent, en effet, le même attribut A_1. D'un autre côté, les modules M_1, M_3 et M_4 sont aussi reliés par la relation UA_M. Le module M_1 accède directement à l'attribut A_2, tout comme le module M_3, alors que le module M_4 utilise le module M_3 qui accède à l'attribut A_2. En effet, selon la figure 6, nous avons $UA_{M1} = \{A_1, A_2\}$, $UA_{M2} = \{A_1\}$, $UA_{M3} = \{A_2\}$ et $UA_{M4} = \{A_2\}$. Par la suite nous avons, $UA_{M1} \cap UA_{M2} = \{A_1\}$, $UA_{M1} \cap UA_{M3} = \{A_2\}$ et $UA_{M1} \cap UA_{M4} = \{A_2\}$.

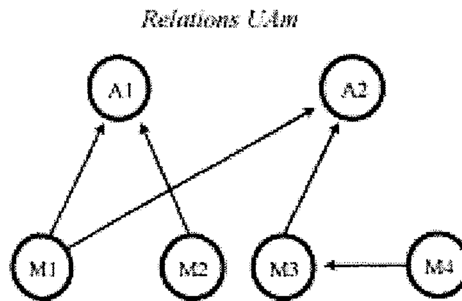

Figure 6 Relation UAm (Modules partageant des attributs).

Critère de connexion Modules-Modules

Définition 2

Définissons UM_{Mi} comme l'ensemble des modules utilisés directement ou indirectement par un module M_i. Un module M_j est utilisé directement par un module M_i , si le module M_j apparaît dans le corps du module M_i. Un module M_j est utilisé indirectement par un module M_i s'il est utilisé directement par un autre module appelé directement ou indirectement par M_i. Il y a m ensembles UM_{M1}, UM_{M2}, ... UM_{Mm}. Deux modules Mi et Mj d'un aspect sont directement reliés par la *relation* UM_M si $UM_{Mi} \cap UM_{Mj} \neq \varnothing$. Ce qui veut dire qu'il y a au moins un module

conjointement partagé (directement ou indirectement) par les deux modules. Nous considérons aussi que M_i et M_j sont directment reliés si $M_j \in UM_{Mi}$ ou $M_i \in UM_{Mj}$.

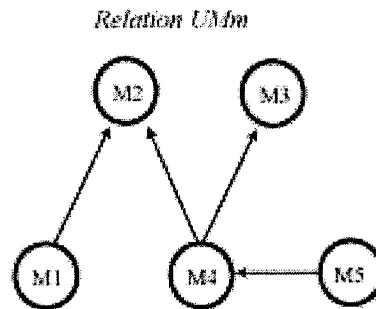

Figure 7 Relation UMm (Modules appelant le même module).

Dans l'exemple illustré par la figure 7, nous pouvons noter, par exemple, que les modules M_1 et M_4 sont directement reliés par la relation UM_M. Ils utilisent le même module M_2. D'un autre côté, les modules M_1 et M_5 sont aussi reliés par la relation UM_M. Le module M_1 utilise directement le module M_2 comme c'est le cas pour le module M_4, qui est utilisé à son tour par le module M_5. En effet, selon la figure 7, nous avons: $UM_{M1} = \{M_2\}$, $UM_{M4} = \{M_2, M_3\}$, $UM_{M5} = \{M_2, M_3, M_4\}$. Ainsi, $UM_{M1} \cap UM_{M4} = \{M_2\}$, $UM_{M1} \cap UM_{M5} = \{M_2\}$ et $UM_{M4} \cap UM_{M5} = \{ M_2, M_3\}$.

Nous définissons, dans le chapitre suivant, une métrique de cohésion pour les aspects basée sur les critères de connexion présentés précédemment.

CHAPITRE 5

MESURE DE COHÉSION POUR LA PROGRAMMATION ASPECT

Grâce aux critères de connexion définis au chapitre précédant, nous définissons la cohésion d'un aspect par le degré de liaison entre ses modules. Notre approche est inspirée par quelques travaux permettant d'évaluer la cohésion d'une classe [Bad95, Bad04, Bie95]. Nous croyons que notre approche constitue une adaptation intéressante de certaines métriques reconnues et testées pour la programmation objet. Les aspects doivent représenter une préoccupation transverse de manière cohésive; tous les membres d'un aspect devraient interagir entre eux afin d'implanter une propriété logique du système. Certains membres de l'aspect sont indirectement considérés. Par exemple, les points de coupure sont indirectement considérés dans la signature de l'advice. Nous définissons notre mesure de cohésion pour la programmation aspect dans ce qui suit [Gel05].

Définissons $NM(Aspect_i)$ comme étant le nombre total de paires de modules dans un aspect. NM est le nombre maximal de connexions entre les modules de l'aspect. Ainsi, pour un aspect possédant N modules :

$$NM(Aspect_i) = N * (N - 1) / 2, N > 1.$$

Définition 3 : Deux modules M_i et M_j peuvent être reliés de diverses façons : soit en partageant des attributs (définition 1), soit en partageant des modules (définition 2) ou les deux.

Considérons le graphe non-dirigé G_D où chaque nœud représente un module de l'aspect. Il y a un arc entre deux modules M_i et M_j s'ils sont reliés. Définissons $NC(Aspect_i)$ comme étant le nombre de connexions entre les modules. $NC(Aspect_i)$ est obtenu en calculant le nombre d'arcs dans le graphe G_D. Une nouvelle métrique pour la cohésion d'un aspect est définie à partir des relations entre ces modules. La

métrique de cohésion proposée, ACoh, représente le nombre relatif de modules connectés au sein de l'aspect : $ACoh(Aspect_i) = NC(Aspect_i) / NM(Aspect_i) \in [0,1]$.

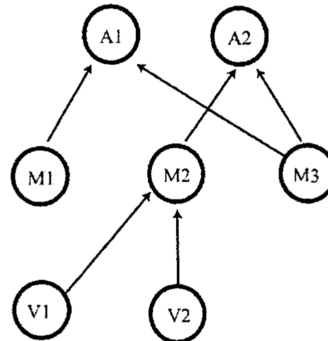

Figure 8 Interactions entre les membres d'un aspect.

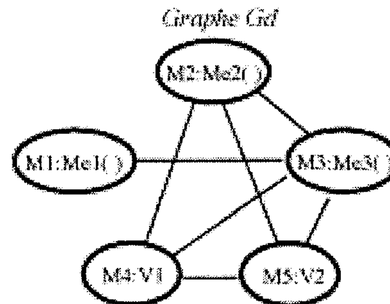

Figure 9 Exemple de graphe GD.

Afin d'illustrer notre métrique, considérons un exemple. La figure 8 présente les interactions entre les membres d'un aspect. L'aspect est défini par deux attributs A_1 et A_2 et cinq modules (trois méthodes Me_1, Me_2 et Me_3 et deux advices V_1 et V_2). La figure 9 présente le graphe G_D correspondant (modules reliés) découlant des critères de connexion. Sachant que nous avons cinq modules (trois méthodes et deux advices), le nombre de paires de modules potentielles est : $NM = N * (N-1) / 2 = 5 * 4 / 2 = 10$. Selon la figure 9, le nombre d'arcs dans le graphe G_D est de 7. La cohésion pour l'aspect considéré est donc : $ACoh = NDC / NM = 7 / 10 = 0.70$.

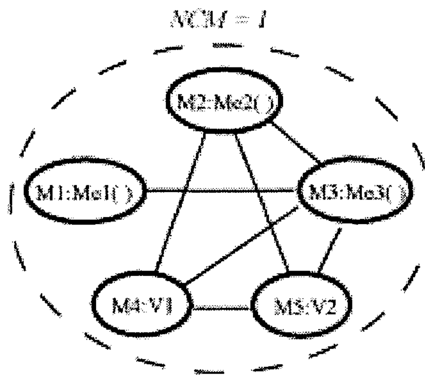

Figure 10 Représentation de NCM.

De plus, le graphe G_D peut-être utilisé pour générer le nombre de groupes de membres connectés NCM (*Number of Connected Members*) au sein de l'aspect. Une représentation de NCM est fournie à la figure 10. Le nombre de groupes (connexes) de membres connectés (NCM) peut aussi être utilisé comme un indicateur de manque de cohésion au sein de l'aspect. Une valeur de NCM égale à 1 indique qu'il existe un groupe unique de membres connectés au sein d'un aspect, alors que toute autre valeur (>1) de NCM peut être utilisée pour indiquer que la modélisation de la préoccupation au sein d'un aspect pourrait être plus efficace si elle était séparée en deux ou plusieurs aspects. Ceci permet de révéler une disparité au sein des fonctionnalités offertes par l'aspect. La valeur de NCM, au sein d'un aspect possédant des membres travaillant conjointement pour représenter un comportement bien défini, sera égale à 1.

La valeur de la métrique ACoh exprime le degré de liaison entre les membres d'un aspect. Une valeur faible de ACoh indique que les membres de l'aspect sont faiblement reliés entre eux, malgré le fait qu'ils peuvent constituer un groupe singulier représentant un comportement bien défini. Toutefois, la métrique ACoh peut aussi indiquer (de manière implicite) l'existence de plusieurs (deux ou plus) groupes de membres connectés. En effet, ces différents groupes peuvent refléter, dans certains cas, la disparité dans l'affectation des rôles à un aspect. Sans la métrique NCM, on doit parcourir manuellement le code pour identifier les problèmes de conception. Une faible valeur de ACoh peut être interprétée de diverses façons et

révéler diverses situations : (1) les membres de l'aspect constituent un seul groupe de membres connectés mais faiblement reliés (2) les rôles assignés à l'aspect sont disparates (non reliés) (3) potentiellement, les deux situations. En pratique, nous pourrions avoir deux aspects avec des valeurs de cohésion semblables (assumons par exemple une valeur de 0,5) : dans le premier cas, les membres sont faiblement reliés mais constituent un seul groupe de membres connectés, et dans le second cas les rôles assignés à l'aspect sont disparates (non reliés) et reflétés dans l'implémentation. Sans la métrique NCM, il est impossible d'identifier cette problématique sans accéder directement au code. La métrique NCM reflète explicitement cette problématique. Utilisées conjointement, les deux métriques reflètent plusieurs problèmes de structure au niveau de la conception des aspects. Les études de cas du chapitre 7 illustreront cette dimension. La métrique ACoh indique le degré de cohésion des membres de l'aspect. NCM, prise avec ACoh, permet de mieux interpréter les résultats.

De plus, Briand et al. suggèrent dans [Bri98] qu'une mesure de cohésion doit posséder les propriétés suivantes : non négative et standardisée, avoir un minimum et un maximum, être monotone et que la cohésion ne doit pas augmenter lorsque deux composants sont combinés. La métrique ACoh fournit une mesure positive et continue entre un minimum de 0 et un maximum de 1. La dernière propriété sera illustrée dans le chapitre suivant.

CHAPITRE 6

IMPLÉMENTATION DES MÉTRIQUES DE COHÉSION ASPECT

L'intérêt d'une mesure des attributs de qualité repose sur sa capacité à assister les programmeurs dans le processus de développement. Pour être utile, cette mesure doit être facile à interpréter et à générer. Une proposition de métrique peut être à priori intéressante mais difficile, voir impossible, à utiliser dans un contexte réel de développement. L'implémentation d'une métrique de qualité doit permettre de s'adapter à des projets d'envergure afin de fournir une mesure dans un délai de calcul raisonnable.

L'objectif du travail est d'automatiser la proposition de la métrique ACoh afin d'analyser des systèmes aspect d'envergure. Ultimement, nous voulons mettre notre proposition en relief avec les autres propositions (LCOO, Zhao), afin de démontrer que notre approche reflète mieux la cohésion au sein des membres d'un aspect. Pour ce faire, nous avons utilisé une approche permettant d'interpréter du code afin de capturer les dépendances entre les membres d'un aspect. À partir de ce code, nous pouvons extraire les dépendances entre les membres de l'aspect afin de mesurer le degré de liaison entre eux. À partir de ces dépendances, nous calculons la cohésion pour un aspect.

Nous avons utilisé une extension de JavaCC pour implémenter notre métrique de cohésion au sein des systèmes aspects. JavaCC (*Java Compiler Compiler*) [Jcc05], disponible gratuitement, est un générateur d'analyseurs lexicaux et syntaxiques pour les applications Java. JavaCC permet de lire les descriptions d'un langage et de générer du code, en java, permettant d'analyser une application. JavaCC est particulièrement utile pour instrumenter un code déjà existant afin de capturer certaines propriétés d'un système développé en Java. Le paradigme aspect introduit de nouvelles abstractions et de nouvelles constructions au sein des systèmes. Nous avons donc étendu la grammaire disponible pour la grammaire java 1.5 [Jav05] afin

de supporter ces nouvelles dimensions. Notre grammaire permet d'analyser des projets respectant la version du compilateur 1.2.x d'AspectJ.

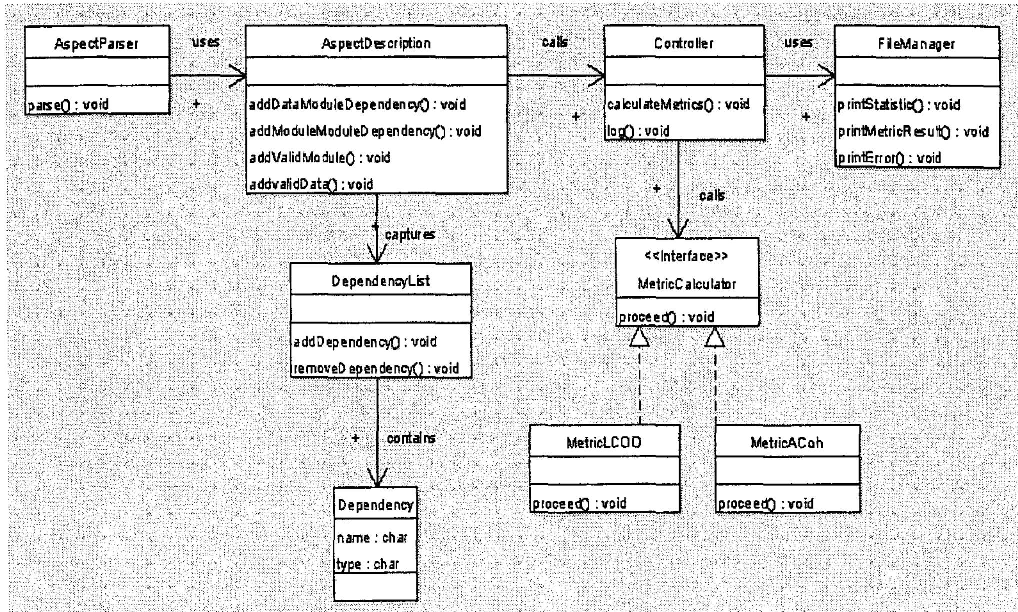

Figure 11 Représentation UML de l'implémentation des métriques.

La figure 11 représente, sous une forme simplifiée, notre modèle de conception permettant d'implémenter les métriques de cohésion aspect. Certains attributs et certaines opérations ont été omis afin de faciliter la compréhension du modèle. Les classes et les interfaces permettant d'implémenter les métriques de cohésion au sein d'un système aspect sont : *AspectDescription, DependencyList, Dependency, Controller, FileManager, MetricCalculator, MetricLCOO* et *MetricACoh.* Il s'agit des principales classes au sein de l'application. La figure 11 ne représente pas les classes jouant un rôle secondaire (helper classes).

La classe *AspectParser* est générée à partir de l'extension de la grammaire de JavaCC. JavaCC est un générateur de programme qui génère les classes suivantes :

Fichiers indépendants générés :

- *SimpleCharStream.java* - représente le flux de caractères en entrée.
- *Token.java* - représente une unité (token) en entrée
- *TokenMgrError.java* – représente une erreur qui a été levée par l'application
- *ParseException.java* – représente une exception indiquant que l'entrée n'est pas conforme à la grammaire du parser.

Fichiers personnalisés générés.

- *AspectParser.java* – la classe parser
- *AspectParserTokenManager.java* – le gestionnaire d'unité (token manager class).
- *AspectParserConstants.java* – l'interface permettant d'associer une interface entre le token et les noms symboliques.

Ces classes permettent de supporter la classe *AspectParser* pour analyser le code d'un système aspect. La classe *AspectParser* est représentée au sein de notre diagramme de conception.

Le répertoire de l'application (racine des sources) à analyser est fournit sous forme de chaînes de caractères à la classe *AspectParser*. À partir de cette racine, l'arborescence du projet aspect sera entièrement parcourue afin d'analyser chacun des aspects faisant partie du projet. Pour chacun de ces aspects, la classe *AspectParser* créera une instance de la classe *AspectDescription* permettant de représenter les dépendances entre les membres d'un aspect. Cette classe contient toutes les dépendances Données-Modules et Modules-Modules entre les membres d'un aspect. Cette classe contient aussi l'ensemble des attributs valides et des modules valides d'un aspect. Ces ensembles de membres valides (attributs ou modules) permettent de différencier, entre autres, une interaction entre des membres valides et une interaction entre des membres non valides. Par exemple, un appel système de type *System.out.println()* pourrait utiliser une méthode de l'aspect pour effectuer un affichage. La liste de modules valides permettra d'éliminer ce type de

dépendances entre les modules. Ainsi, il n'y aura pas de dépendances entre le module *println()* et la méthode de l'aspect.

Les dépendances valides seront représentées à l'aide de la classe *DependencyList.* Chacun des objets présents dans ces listes sera de type *Dependance.* Ces dépendances peuvent être de deux types : Données-Modules et Modules-Modules. Elles représentent une connexion entre deux membres de l'aspect selon les définitions que nous avons établies. La mesure de cohésion au sein des aspects sera calculée à partir de ces connexions entre les membres de l'aspect.

La classe *Controller* est utilisée pour découpler la relation entre l'impression, le calcul des métriques et l'analyse du code. Il s'agit d'un pont permettant de faire varier l'implémentation des diverses classes en minimisant l'impact des modifications. La classe *FileManager* est responsable des sorties; elle gère notamment : l'écriture des statistiques, l'écriture des résultats des métriques ainsi que l'écriture des erreurs. L'interface *MetricCalculator* définit quant à elle l'ensemble des opérations communes aux mécanismes de calcul des métriques. L'implémentation des métriques de cohésion sera faite à partir de cette interface. Dans notre étude, les classes MetricACoh et MetricLCOO permettent d'implémenter, respectivement, notre proposition (ACoh) ainsi que celle de Sant'Anna et Al (LCOO).

Il y a donc autant d'instances d'*AspectDescription* que d'aspects présents dans le système. La cohésion de chacun des aspects présents dans l'arborescence du projet sera calculée et présentée sous forme de fichier texte. La mesure de cohésion de chacun des aspects est disponible dans un répertoire du projet. Le chapitre suivant présente diverses études de cas qui permettront de situer les différentes approches y compris celle que nous proposons sur des exemples concrets.

CHAPITRE 7

LES ÉTUDES DE CAS

Nous allons présenter, dans ce qui suit, trois études de cas. Chacune de ces études illustrera notre approche sur des exemples concrets. Les exemples ont été choisis afin de représenter des situations diverses permettant de comparer notre métrique avec les autres propositions.

Étude de cas 1

Notre première étude de cas sera basée sur l'exemple utilisé par Zhao et Xu [Zha04] pour illustrer leur approche. Le code est présenté à la figure 12.

```
1  public class Point {
2   protected int x, y;
3   public Point(int _x, int _y) {
4    x = _x; y = _y;
5   }
6   public int getX() {return x;}
7   public int getY() {return y;}
8   public void setX(int _x) {x = _x;}
9   public void setY(int _y) {y = _y;}
10  public void printPosition() {
11   System.out.println("Point at (" + x + ", " +   y + ")");
12  }
13  public static void main(String[] args) {
14   Point p = new Point(1, 1);
15   p.setX(2);
16   p.setY(3);
17  }
18 }
19 public class Shadow {
20  public static final int offset = 10;
21  public int x, y;
22  Shadow(int _x, int _y) {
23   x = _x;
24   y = _y;
25  }
26  public void printPosition() {
27   System.out.println("Shadow at (" + x + ", " + y +   ")")
28  }
29 }
```

```
30 public aspect PointShadowProtocol {
31  private int shadowCount = 0;
32  public static int getCount() {
33   return   PointShadowProtocol.aspectOf().shadowCount;
34  }
35  public static void associate(Point p, Shadow s) {
36   p.shadow = s;
37  }
38  public static Shadow getShadow(Point p) {
39   return p.shadow;
40  }
41  private Shadow Point.shadow;
42  pointcut setting(Point p, int x, int y): args(x,  y)&&
43  target(p)&& initialization(Point.new(int, int));
44  pointcut settingX(Point p) : target(p) && call(void setX(int));
45  pointcut settingY(Point p) : target(p) && call(void setY(int));
46  after(Point p, int x, int y) returning() :     setting(p, x, y) {
47   Shadow s = new Shadow(x, y);
48   associate(p, s);
49   shadowCount++;
50  }
51  after(Point p) : settingX(p) {
52   Shadow s = getShadow(p);
53   s.x = p.getX() + Shadow.offset;
54   p.printPosition();
55   s.printPosition();
56 }
57  after(Point p) : settingY(p) {
58   Shadow s = getShadow(p);
59   s.y = p.getY() + Shadow.offset;
60   p.printPosition();
61   s.printPosition();
62  }
63 }
```

Figure 12 Code source utilisé par Zhao et Xu.

Cet exemple permettra de démontrer la faisabilité de notre approche et de la positionner par rapport aux approches de Zhao et Xu et Sant' Anna et al. En examinant le code de cet exemple, on peut noter que plusieurs dépendances entre les membres de l'aspect sont définies. Deux classes sont définies : la classe *Point* et la

classe *Shadow*. La classe *Point* représente le concept simple d'un point possédant une coordonnée en x et y. Quelques méthodes simples sont définies pour supporter la manipulation d'un point. La classe *Shadow* représente, quant à elle, le concept de l'ombrage projeté par un point. L'exemple est simple et la classe *Shadow* encapsule la valeur correspondant au décalage entre le point et son ombre. L'association entre un point et son ombrage (classes *Point* et *Shadow*) est adéquatement capturée par l'aspect *PointShadowProtocol.* À toutes les fois qu'un objet de type Point est créé, l'aspect génère une instance de la classe *Shadow* qui lui sera associée. Quelques modules sont définis pour supporter le déplacement des points et de leurs ombrages correspondants. La figure 13 illustre les membres de l'aspect considéré ainsi que leurs diverses interactions.

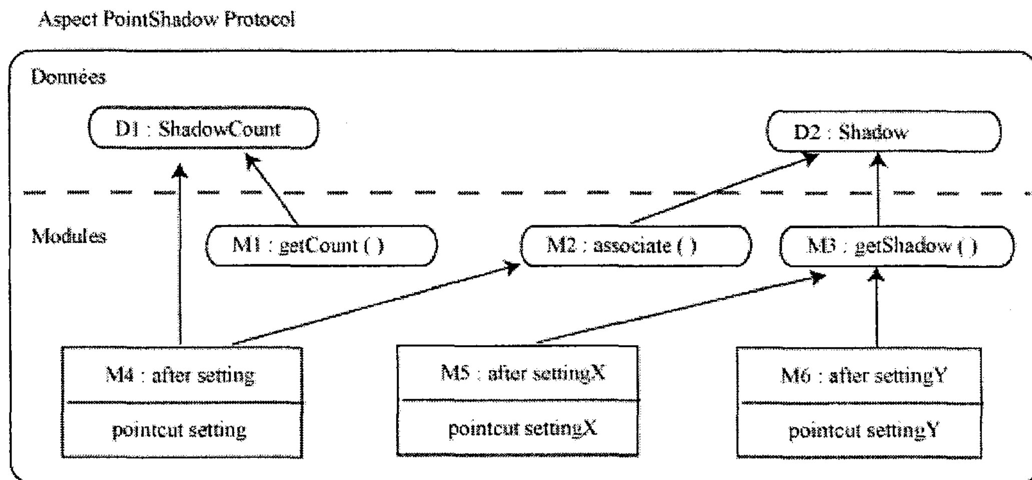

Figure 13 Interactions entre les membres de l'aspect.

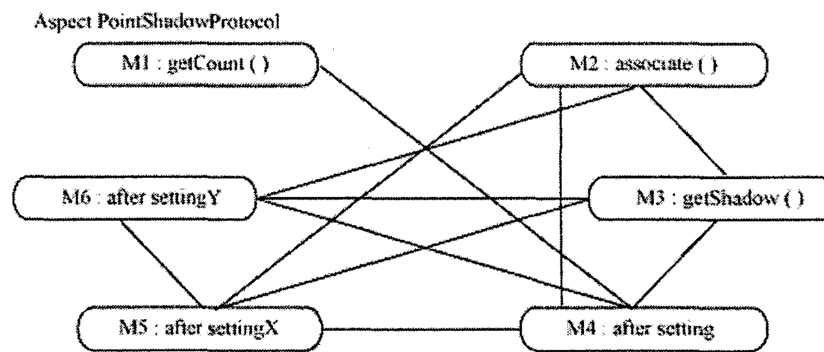

Figure 14 Connexions entre les modules de l'aspect.

Le graphe présenté à la figure 14 représente le nombre de paires de modules connectés capturées par notre approche. Les connexions entre les modules sont représentées par un arc. Par exemple, en utilisant la définition 1, un arc est tracé entre le module *M2: associate()* et *M3: getShadow()* (figure 14). En effet, les modules *M2: associate()* et *M3: getShadow()* accèdent à la donnée *D2: shadow* (figure XX). En utilisant de nouveau la définition 1, nous pouvons relier le Module *M4: after setting* avec le Module *M6:after settingY* (figure 14). L'attribut *D2: shadow* est utilisé indirectement par les deux advices (M4 et M6) étant donné qu'il est respectivement utilisé, de façon directe, par le Module *M2: associate ()* (utilisé par M4) et *M3: getShadow()* (utilisé par M6). Les Modules *M5: after settingX* et *M6: afterSettingY* peuvent être reliés en utilisant la définition 1 ou la définition 2. Afin de simplifier la notation, nous utiliserons *PSP* pour représenter *PointShadowProtocol.* Pour cet exemple, nous donnons, dans ce qui suit, les valeurs de cohésion obtenues selon les trois approches :

ACoh (PSP) = NC (PSP) / NM(PSP) = 11 / 15 = 0.73

LCOO = $|P| - |Q| = 13 - 2 = 11$

$\Gamma(A)$ (Zhao) = 1/18 = 0.056

Comme nous l'avons mentionné, plusieurs dépendances existent entre les membres de l'aspect. La valeur obtenue (0,73) selon notre approche (métrique ACoh), reflète les dépendances entre les membres de l'aspect. Pour cet exemple, la valeur de NCM est égale à 1. La valeur de cohésion suggérée par la métrique de Zhao et Xu, dans leur papier [Zha04], est égale à 1/18 (0,056). Cette mesure indique une cohésion très faible, pour ne pas dire inexistante, au sein des membres de l'aspect. Toutefois, la valeur obtenue à l'aide de cette approche ne reflète pas correctement la structure de l'aspect considéré (*PointShadow Protocol*). Par ailleurs, LCOO fournit une mesure de cohésion égale à 11. Cette mesure est difficile à interpréter puisqu'il n'existe aucune recommandation pour l'interprétation des valeurs fournies par cette métrique.

Étude de cas 2

Gregor Kiczales mentionne dans [Kic04] qu'une préoccupation transverse est un élément de programme (conception, prérequis, test, etc.) que l'on veut considérer, ou isoler, dans une unité. Les préoccupations peuvent être relativement petites ou étendues à plusieurs composantes au sein de l'application. Par exemple, la stratégie utilisée pour représenter la gestion des erreurs peut être composée de plusieurs aspects de tailles diverses. Nous croyons que l'assignation des responsabilités aux aspects devrait suivre les mêmes principes d'assignation des responsabilités aux classes. Un aspect devrait exprimer une préoccupation de manière cohésive. Comme les classes auxquelles on aurait assigné des responsabilités disparates, un aspect implémentant plusieurs préoccupations disparates présentera une faible cohésion. Cet aspect sera (intuitivement) difficile à maintenir, à comprendre, à tester et à réutiliser. Afin d'illustrer cet important problème de conception, prenons un exemple concret.

```
public aspect PointShadowProtocol {

        private int shadowCount = 0;
        public static int getCount() {
                return PointShadowProtocol.aspectOf().shadowCount;
        }
        public static void associate(Point p, Shadow s) {
                p.shadow = s;
        }
        public static Shadow getShadow(Point p) {
                return p.shadow;
        }
        private Shadow Point.shadow;
        pointcut setting(Point p, int x, int y) : args(x, y)
                && target(p)
                && initialization(Point.new (int, int));
        pointcut settingX(Point p) : target(p) && call(void setX(int));
        pointcut settingY(Point p) : target(p) && call(void setY(int));

        after(Point p, int x, int y) returning() : setting(p, x, y) {
                Shadow s = new Shadow(x, y);
                associate(p, s);
                shadowCount++;
        }
        after(Point p) : settingX(p) {
                Shadow s = getShadow(p);
                s.x = p.getX() + Shadow.offset;
                p.printPosition();
                s.printPosition();
        }
        after(Point p) : settingY(p) {
                Shadow s = getShadow(p);
                s.y = p.getY() + Shadow.offset;
                p.printPosition();
                s.printPosition();
        }
        private boolean Point.assertX(int x) {
                return (x <= 100 && x >= 0);
        }
        private boolean Point.assertY(int y) {
                return (y <= 100 && y >= 0);
        }
        pointcut roleA_assertX(Point p, int x) : target(p)
                && args(x)
                && call(void setX(int));
        pointcut roleA_assertY(Point p, int y) : target(p)
                && args(y)
                && call(void setY(int));
        before(Point p, int x) : roleA_assertX(p, x) {
                if (!p.assertX(x)) {
                        System.out.println("Valeur illegale pour x");
                }
        }
        before(Point p, int y) : roleA_assertY(p, y) {
                if (!p.assertY(y)) {
                        System.out.println("Valeur illegale pour y");
                }
        }

}
        private boolean Point.assertX(int x) {
                return (x <= 100 && x >= 0);
        }
        private boolean Point.assertY(int y) {
                return (y <= 100 && y >= 0);
        }

        pointcut roleA_assertX(Point p, int x) : target(p)
                && args(x)
                && call(void setX(int));

        pointcut roleA_assertY(Point p, int y) : target(p)
                && args(y)
                && call(void setY(int));

        before(Point p, int x) : roleA_assertX(p, x) {
                if (!p.assertX(x)) {
                        System.out.println("Valeur illégale pour x");
                }
        }
        before(Point p, int y) : roleA_assertY(p, y) {
                if (!p.assertY(y)) {
                        System.out.println("Valeur illégale pour y");
                }
        }
}
```

Figure 15 Code source étendu pour l'aspect PointShadowProtocol.

Assignons à l'aspect *PointShadowProtocol* présenté dans l'exemple précédant une nouvelle préoccupation; la validation des pré-conditions lors de l'assignation d'une

valeur aux attributs x et y de la classe *Point*. Le code correspondant à cette nouvelle assignation est présenté à la figure 15. Un message sera simplement envoyé lorsque les pré-conditions (bornées par un maximum et un minimum) ne seront pas respectées. Ainsi, l'aspect exprime deux préoccupations : l'association entre un point et son ombre (préoccupation A) et la validation des pré-conditions (préoccupation B). Ces deux rôles sont, en effet, disparates. Aucune connexion directe ou indirecte n'existe entre les deux préoccupations.

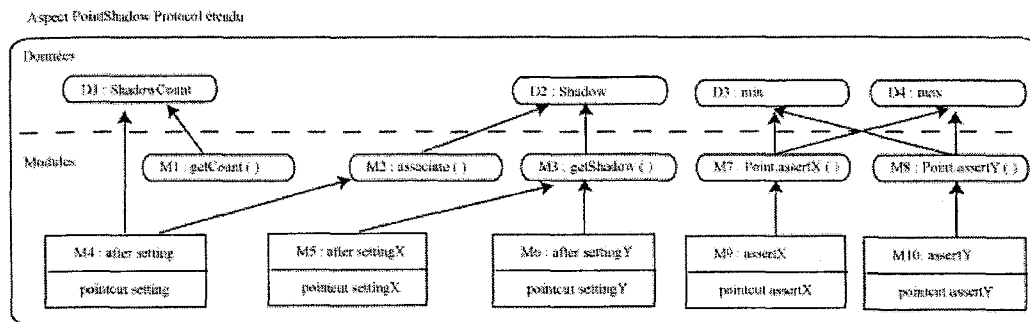

Figure 16 Interactions entre les membres de l'aspect.

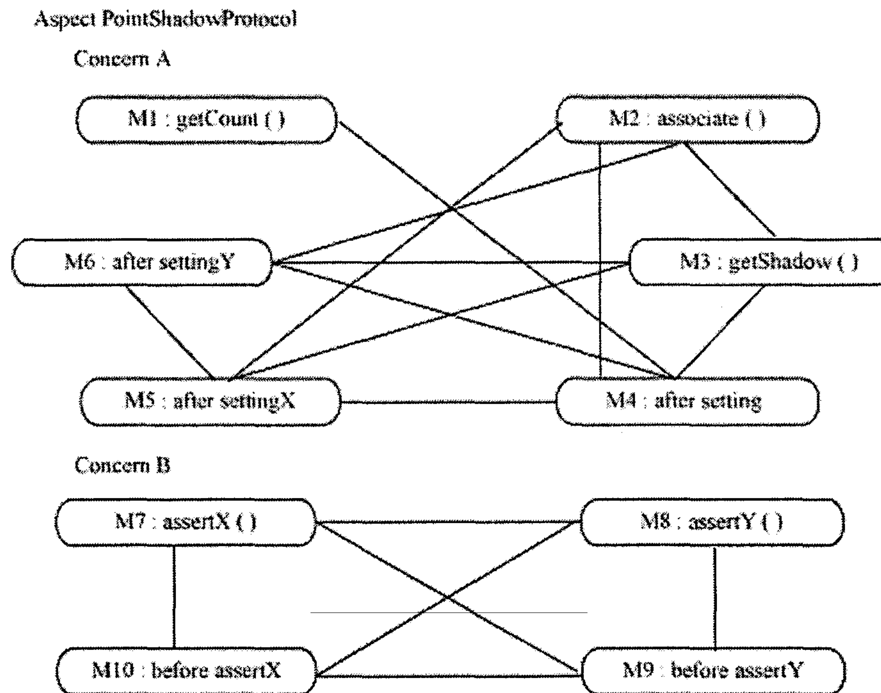

Figure 17 Connexions entre les membres de l'aspect.

Notre objectif dans cette étude de cas est de démontrer que notre approche reflète (à travers les mesures de cohésion) la disparité pouvant exister entre les fonctionnalités fournies par l'aspect (plusieurs rôles affectés à l'aspect). Cette nouvelle configuration devrait inévitablement influencer, à la baisse, notre mesure de cohésion pour cet aspect. Le graphe présenté à la figure 17 illustre les paires de modules connectés au sein de l'aspect. Les modules de l'aspect sont représentés par des nœuds alors que les arcs représentent une relation entre les modules connectés. En examinant attentivement les figures 16 et 17, on remarque que les deux préoccupations ne partagent rien. Elles sont vraiment disparates. Les valeurs de cohésion obtenues pour cet aspect, selon les trois approches, sont :

ACoh (PSP) = NC (PSP) / NM (PSP) = 17 / 45 = 0.38

LCOO = $|P| - |Q| = 42 - 3 = 39$

$\Gamma(A)$ (Zhao) = 7 / 270 = 0.026

En effet, deux préoccupations disparates ont été assignées à un même aspect. Cette situation est reflétée par une baisse significative de la valeur de ACoh. Cette baisse est aussi reflétée par la métrique LCOO maintenant égale à 39 (elle enregistre une hausse). Cependant, comme mentionné par Henderson-Seller dans [Hen95] (pour la métrique LCOM), cette valeur est difficile à interpréter car il n'y a aucune recommandation pour l'interprétation des valeurs fournie par LCOM. En effet, quand la valeur de cohésion entre les membres d'un aspect est nulle, on peut s'attendre à ce que la cardinalité de P (le nombre de paires sans similarité) soit élevée et que la cardinalité de Q (le nombre de paires possédant des similitudes) soit faible. Ceci explique la forte valeur de LCOO dans cet exemple. Un autre problème concernant LCOM (et LCOO indirectement) est que l'on tente de mesurer la cohésion structurelle d'une composante alors qu'un des objectifs majeurs de la programmation orienté objet est l'habilité de créer logiquement (i.e sémantiquement) des modules cohésifs (classes) [Hen95]. LCOO ne capture pas cette caractéristique qui semble importante dans un système aspect. Henderson-Seller suggère aussi dans [Hen96]

qu'une mesure de LCOM pondérée sur une échelle de pourcentage (fraction/pourcentage d'une valeur de cohésion parfaite) est plus adéquate. La métrique ACoh reflète correctement ces propriétés. Finalement, la mesure obtenue avec la métrique de Zhao et Xu (0,026) ne reflète pas la structure de l'aspect considéré (en termes de dépendance entre les membres).

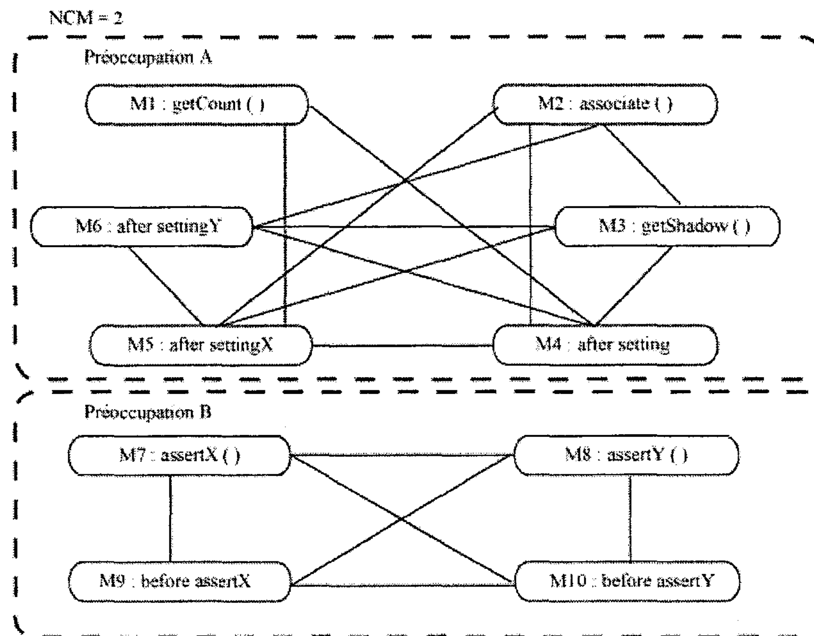

Figure 18 Représentation de NCM.

De plus, sachant que plusieurs responsabilités disparates ont été assignées à un même aspect, nous devrions obtenir plus d'un groupe de membres connexes. En effet, cette assignation est reflétée par la valeur de NCM (égal à 2 dans ce cas). En effet, en examinant le graphe de la figure 17 et la figure 18 représentant les relations entre les attributs et les modules, on peut observer que l'aspect est composé de deux groupes de membres reliés entre eux (un groupe exprimant la préoccupation A, l'autre groupe exprimant la préoccupation B). Ces deux groupes sont représentés à la figure 18 et ils ne partagent rien entre eux. Ce manque de cohésion est adéquatement capturé par la métrique ACoh. Cet aspect devrait être considéré pour une restructuration (séparer l'aspect en deux aspects distincts reflétant les deux préoccupations).

Étude de cas 3

La prochaine étude de cas sera basée sur l'implémentation AspectJ des patrons de conception [Gam95] proposée par Hanneman et Kiczales dans [Han02]. Hanneman et Kiczales mentionnent [Han02] que les améliorations provenant du paradigme aspect apportées aux patrons de conception viennent principalement de la capacité à modulariser l'implémentation des patterns en abstraction réutilisable. En utilisant le paradigme aspect, 74% des patrons de conception sont implémentés de façon plus modulaire alors que 52% sont réutilisables. Ces améliorations proviennent du fait que les dépendances au niveau du code, entre les modules participant à la réalisation d'un patron de conception, sont éliminées. En utilisant la programmation aspect, 12 des 23 patrons de conception sont représentés par un aspect abstrait disponible dans une librairie. La cohésion au sein de tels aspects devrait être élevée.

Ceci constitue une première investigation d'une dimension intéressante apportée par la programmation aspect. En effet, un aspect réutilisable peut être considéré comme un modèle de patrons de conception. Pour cette première analyse, nous avons exclu les définitions de méthodes abstraites, les interfaces ainsi que les considérations d'héritage. Nous présentons dans ce qui suit trois patrons de conception avec les mesures de cohésion correspondantes (métrique ACoh, métrique LCOO et métrique $\Gamma(A)$). Il s'agit des patrons observateur, médiateur et commande.

Patron observateur

Le parton observateur définit des dépendances un à plusieurs entre des composants de manière à ce que toutes les dépendances soient mises à jour lorsqu'un composant change d'état [Gam95]. La figure 19 représente les diverses interactions entre les membres (données et modules) de l'aspect *ObserverProtocol.*

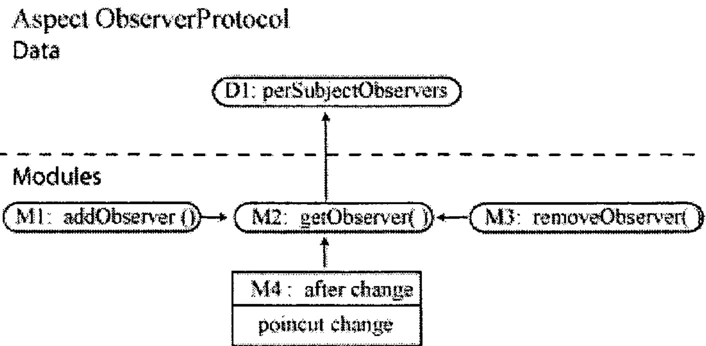

Figure 19 Interactions entre les membres du patron Observateur.

Pour cet exemple, nous obtenons avec les trois approches les valeurs de cohésion suivantes :

ACoh (OP) = NC (OP) / NM (OP) = 6/6 = 1

LCOO = |P| - |Q| = 6 – 0 = 6

Γ(A) (Zhao) = 5 / 12 = 0.41

Comme nous l'avons mentionné plus haut, une forte cohésion devrait être présente au sein des aspects appartenant à la librairie de patrons de conception. En utilisant notre métrique, nous obtenons une mesure élevée de cohésion (ACoh = 1) pour le patron observateur. En examinant les différents détails relatifs aux dépendances qui existent entre les membres de l'aspect, nous pouvons noter que les membres constituent un groupe singulier de membres reliés. Ceci indique que tous les membres du patron observateur travaillent conjointement pour implémenter une préoccupation commune. La valeur de NCM, qui est égale à 1, vient renforcer cette position. Une forte cohésion au sein d'un aspect indique que les différents membres expriment un rôle singulier de manière cohésive. Il sera difficile de séparer un tel aspect. C'est le cas du patron observateur implémenté avec le paradigme aspect.

Nous obtenons une valeur Γ(A) = 5/12 (0.41) en utilisant l'approche de Zhao et Xu. La raison pour laquelle nous obtenons une valeur de cohésion aussi élevée peut être partiellement expliquée par le fait que le patron observateur n'utilise qu'un seul

attribut. Ainsi, la mesure correspondant à la cohésion inter-attribut (γ_a) sera égale à 1. Comme bref rappel, $\Gamma(A)$ est calculé en utilisant $\beta_1 * \gamma_a + \beta_2 * \gamma_m + \beta_3 * \gamma_{ma}$; où chaque γ correspond à une facette de cohésion. Étant donné que nous utilisons une valeur arbitraire de 1/3 pour la pondération de nos paramètres β, la contribution de $\beta_1 * \gamma_a$ devient importante. Une autre pondération aurait changé la valeur de la cohésion, ce qui soulève encore une fois le côté problématique de cette approche.

De plus, la valeur obtenue avec la métrique LCOO est de 6, qui, prise individuellement, est difficile à interpréter pour les mêmes raisons que nous avons mentionnées précédemment. La différence fondamentale entre les résultats fournis par la métrique ACoh et LCOO provient du fait que notre approche tient compte de l'usage indirect des attributs, ce qui n'est pas le cas de la métrique LCOO. Comme nous l'avons mentionné, les membres du patron observateur sont fortement liés. Nous croyons que notre approche capture adéquatement les dépendances entre les membres de l'aspect.

Patron médiateur

Le patron médiateur définit un composant qui encapsule comment un ensemble d'objets interagissent. Le médiateur permet d'appliquer les principes de faible couplage entre les composants en éliminant les références directes entre les composants permettant ainsi de varier les interactions indépendamment [Gam95]. La figure 20 représente les diverses interactions entre les membres (données et modules) de l'aspect *MediatorProtocol*.

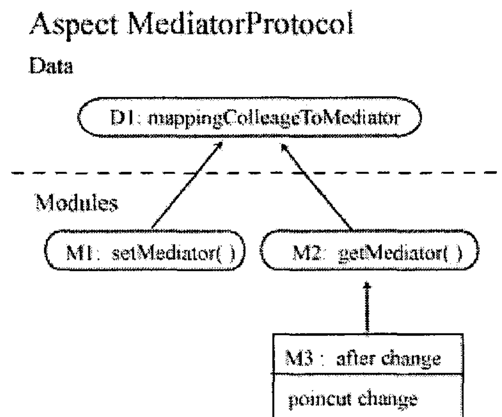

Figure 20 Interactions entre les membres du patron Médiateur.

Pour cet exemple, nous obtenons avec les trois approches les valeurs de cohésion suivantes :

ACoh(MP) = NC(OP) / NM(OP) = 3/3 = 1

LCOO = |P| - |Q| = 2 - 1 = 1

Γ(A) (Zhao) = 6 / 9 = 0.66

Pour ce patron, nous avons obtenu une mesure de cohésion élevée (ACoh = 1) pour le patron médiateur. Ceci est reflété par la structure du patron, au même titre que le patron observateur. Les membres du patron de conception médiateur sont effectivement fortement liés. En examinant les différents détails relatifs aux dépendances existant entre les membres du patron Médiateur, on peut noter que les membres constituent un groupe singulier de membres connectés. Ceci indique que

tous les membres du patron médiateur travaillent ensemble pour implémenter un comportement commun bien délimité. La valeur de NCM dans ce cas est aussi égale à 1.

L'approche de Zhao et Xu nous amène aux mêmes problèmes discutés précédemment pour le patron observateur. De plus, la valeur de LCOO est égale à 1. Le fait que LCOO ne propose pas de valeur maximale semble encore une fois problématique pour interpréter les résultats.

Patron Commande

La patron de conception Commande permet d'encapsuler une requête comme un composant, laissant la possibilité de paramétrer les clients avec diverses requêtes, queue ou trace, et supporter les opérations inverses (undoable) [Gam95]. La figure 21 représente les diverses interactions entre les membres (données et modules) de l'aspect *CommandProtocol.*

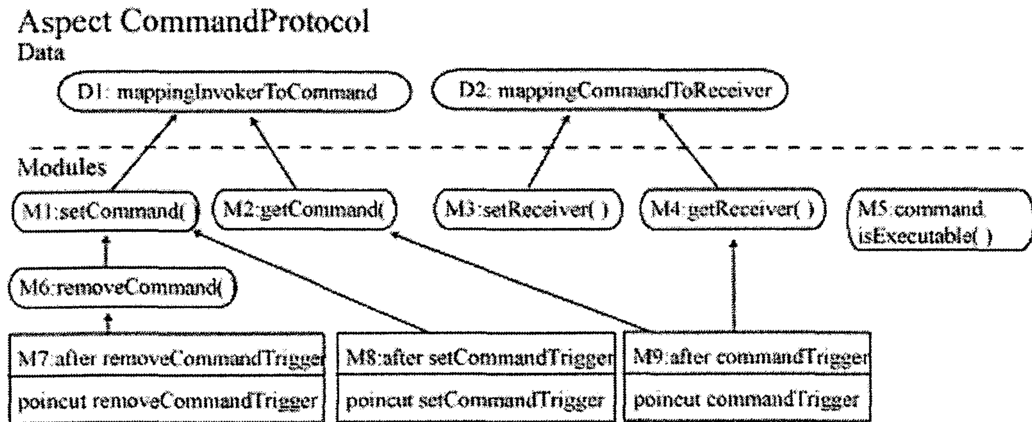

Figure 21 Interactions entre les membres du patron Commande.

Pour cet exemple, nous obtenons avec les trois approches les valeurs de cohésion suivantes :

ACoh (CP) = NC (OP) / NM (OP) = 18/36 = 0,50
LCOO = |P| - |Q| = 34 – 2 = 32
Γ(A) (Zhao) = 7 / 216 = 0.032

Nous obtenons, dans ce cas, une baisse significative de cohésion (ACoh = 0,5 et LCOO = 32). Ces résultats reflètent la structure du patron Commande. Les membres du patron Commande ne sont pas aussi fortement reliés que les membres des patrons de conception Observateur et Médiateur. De plus, le nombre de groupes de membres connectés (valeur NCM) obtenu pour ce cas est de 2. Ceci indique que les membres

du patron de conception Commande constituent deux groupes de membres connectés, par opposition aux deux autres patrons considérés. En effet, le module 5 (*Command.isExecutable( ))* n'est pas directement relié à aucun autre membre aspect. Cette dimension est reflétée par la valeur de NCM.

Comme nous l'avons mentionné précédemment, la valeur de la métrique ACoh est plus faible que les valeurs de cohésion des patrons Observateur et Mediateur. Nous remarquons, grâce à ce cas, que la métrique ACoh discrimine les patrons selon leur structure. De plus, la valeur de cohésion obtenue en utilisant l'approche de Zhao et Xu est égale à 0,032. Nous pouvons noter une différence significative entre le patron Commande (0,032), d'un côté, et les patrons Observateur et Médiateur (respectivement 0,66 et 0,41), d'un autre côté. L'écart est important.

CHAPITRE 8

ÉTUDE EXPÉRIMENTALE

En pratique, l'utilité des métriques de cohésion provient de leur potentiel à valider des projets d'envergure qui ne peuvent pas être facilement caractérisés manuellement par les développeurs. Jusqu'ici, les aspects utilisés pour mettre en contraste ACoh et les deux autres approches sont relativement petits et peu complexes. Toutefois, étant donnée la situation actuelle du développement aspect, il est relativement difficile de trouver des vraies applications aspects d'envergure.

Pour cette première expérimentation, nous avons développé un outil de mesure de cohésion (en Java) pour les programmes AspectJ afin d'automatiser dans cette version les mesures de LCOO et ACoh. Nous avons aussi mis de côté pour cette première expérimentation les méthodes abstraites et les déclarations d'interfaces. Même si nous croyons que l'approche proposée reflète adéquatement la cohésion aspect comparée aux autres approches, une dimension intéressante est apportée par l'héritage. Les aspects concrets fournissent une implémentation concrète pour les méthodes abstraites ou spécifient les participants concrets avec leurs implémentations d'interfaces respectives.

Système sélectionné

Même s'il est encore en phase de développement, nous avons choisi d'analyser le projet Atrack, un projet open source développé par Bodkin et Mulez [Atr05] permettant d'effectuer un suivi des erreurs dans une application. Ce projet a pour objectif de promouvoir la programmation aspect avec AspectJ. Cette application utilise l'AOP pour fournir un support systématique pour les préoccupations techniques, middleware et business. Le projet présente et démontre une utilisation efficace d'une librairie assurant un support commun pour aspectJ. Cette librairie fournit un support pour la gestion d'exceptions, la sécurité, le logging, le traçage, les transactions, etc.

Résultats

Nous avons mesuré les valeurs de cohésion pour les métriques ACoh et LCOO pour le système sélectionné. La table II présente les résultats obtenus. La métrique LCOO mesure le nombre de paires de méthodes/advice qui n'accèdent pas aux mêmes variables d'instance. Une valeur élevée de LCOO indique une disparité dans les fonctionnalités fournies par l'aspect. ACoh est basé sur les critères de connexion Attributs-Modules et Modules-Modules et détermine le degré de liaison des modules.

Il est voulu qu'une faible valeur de LCOO et une forte valeur de ACoh impliquent une forte liaison entre les membres de l'aspect. Comme mentionné par Henderson-Seller [Hen95], une mesure doit fournir des valeurs qui doivent être interprétées de manière unique en termes de cohésion. LCOO ne fournit pas cette habileté. Étant donné qu'il n'y a pas de recommandations pour l'interprétation des valeurs de LCOO, les résultats de la table II sont difficiles à comparer. Ceci est partiellement dû au fait que la valeur de LCOO ne représente pas un ratio du nombre potentiel de connexions. De plus, la métrique LCOM ne fournit pas l'habilité d'obtenir des valeurs sur une plage de probabilités équivalentes [Hen95]. Puisque LCOO est basée sur cette approche, cette remarque est aussi valable pour LCOO. Par exemple, une valeur de LCOO de -12 ou de -3 sera traitée comme 0. Ceci est problématique et sera démontré par un exemple de code.

Tableau II Mesures de cohésion aspect pour le système Atrack.

Liste des aspects traités	# modules	LCOO	ACoh
AjeeExceptionHandling	1	0	1
AjeeLogManager	0	0	0
AtrackActionDefinition	0	0	0
AtrackLogManager	0	0	0
ATrackTransactionControl	1	0	1
Authentication	2	1	1
ControllerExceptionHandling	2	1	1
ExecutionMonitor	12	66	0,32

ExecutionTracer	20	170	0,47
HttpSessionTracer	3	3	1
Invariants.java	1	0	1
LogManager	15	105	0,74
ModelExceptionHandling	1	0	1
Observing	0	0	0
Standards	0	0	0
VirtualMocks	15	101	0,08

Quelques résultats peuvent être extraits de la table II. Considérons les valeurs de LCOO pour les aspects *LogManager* (LCOO = 105) et *VirtualMocks* (LCOO = 101). Selon ces résultats, il semble que ces deux aspects soient très disparates. La valeur de ACoh pour l'aspect *VirtualMocks* (ACoh = 0,08) semble supporter cette affirmation alors que la valeur ACoh pour l'aspect *LogManager* est assez élevée (ACoh = 0,74). L'aspect *VirtualMocks* définit 15 modules plus ou moins disparates. Le nombre de relations maximum entre les membres de l'aspect est de : $N * (N - 1) / 2 = 105$. Pour cet aspect, seulement 8 modules sont reliés. On obtient une mesure de cohésion faible (ACoh = 0,08). Pour l'aspect *VirtualMocks*, les valeurs de LCOO et ACoh suggèrent une valeur de cohésion faible (LCOO = 101 et ACoh = 0,08). Toutefois, ce n'est pas le cas pour l'aspect *LogManager* (LCOO = 105 et ACoh = 0,74).

La figure 22 présente une partie du code source correspondant à l'aspect *LogManager*. Un attribut, *logManagerLogger*, est défini et accédé par la méthode *getLogger*(). Aucune autre méthode, mis à part *getLogger*( ), n'accède directement à l'attribut *logManagerLogger*. Ceci explique la faible valeur de LCOO puisque $|P|$ est défini comme étant le nombre d'intersections nulles entre les ensembles de variables d'instance. Ce nombre d'intersections vides est donc très élevé. Lorsque les modules n'utilisent pas de variables d'instance, les interactions ne sont pas capturées par LCOO malgré le fait que ces modules soient bel et bien reliés. Un nombre important de modules reliés ne sont pas pris en considération par la valeur de la métrique LCOO. D'un autre côté, la valeur de ACoh = 0,74 peut être expliquée par le fait que

notre approche tient compte des interactions entre les modules. Les méthodes *logTrace*, *logWarn*, *logInfo* accèdent indirectement à l'attribut *logManagerLogger* en appelant directement la méthode *getLogger*(). Une dépendance entre ces modules sera capturée. Ces nombreuses dépendances sont représentées dans la valeur élevée de ACoh = 0,74. Il s'agit d'une dimension importante de la cohésion qui n'est pas capturée par l'approche de Sant'Anna et al.

```
1  public aspect LogManager {
2      private Log Loggable.logManagerLogger =
3              LogFactory.getLog(getClass());
4
5      // this should be protected
6      public Log Loggable.getLogger() {
7          if (logManagerLogger==null) {
8              logManagerLogger = LogFactory.getLog(getClass());
9          }
10         return logManagerLogger;
11     }
12     public void Loggable.logTrace(Object message) {
13         getLogger().trace(message);
14     }
15
16     public void Loggable.logInfo(Object message) {
17         getLogger().info(message);
18     }
19
20     public void Loggable.logWarn(Object message) {
21         getLogger().warn(message);
22     }
23 // ..
24
```

Figure 22 Code source de l'aspect LogManager.

Un autre exemple concerne l'aspect *Authentication.* Cet aspect est relativement simple. Le code correspondant à cet aspect est présenté à la figure 23. Cet aspect définit une variable *SUBJECT_ID*, une méthode *forceRedirect*( ) et un advice de type around. Selon l'approche de Sant' Anna et al, nous obtenons une mesure de cohésion LCOO = |P| – |Q| = 1 – 0 puisque les modules, *forceRedirect*( ) et l'advice, n'accèdent pas à l'attribut *SUBJECT_ID.* Toutefois, plusieurs appels à la méthode force Redirect() sont effectués au sein du corps de l'aspect. Cette dimension est capturée par la métrique ACoh. Nous obtenons une cohésion égale à 1 pour cet aspect puisque tous les modules sont reliés entre eux. La définition des interactions

Modules-Modules permet de capturer adéquatement des relations qui sont laissées pour compte par l'approche LCOO.

```
1  public aspect Authentication {
2      public static String SUBJECT_ID = "subject";
3      private pointcut securedCall() : within(*);
4
5      ActionForward around(final Action action, final HttpServletRequest request,
6      final HttpServletResponse response) throws Exception :
7          AtrackActionDefinition.aTrackActionExecute(action, *, *, request, response) &&
8              securedCall() {
9          HttpSession session = request.getSession();
10         Subject subject = (Subject)session.getAttribute(SUBJECT_ID);
11
12         if (subject == null) {
13             try {
14                 return proceed(action, request, response);
15             } catch (SecurityException e) {
16                 return forceRedirect(request, response);
17             } catch (Exception e) {
18                 if (e.getCause() instanceof SecurityException) {
19                     return forceRedirect(request, response);
20                 } else {
21                     throw e;
22                 }
23             }
24         } else {
25             try {
26                 return (ActionForward)Subject.doAsPrivileged(subject,
27                     new PrivilegedExceptionAction() {
28                         public Object run() throws Exception {
29                             return proceed(action, request, response);
30                         }
31                     }, null);
32             } catch (PrivilegedActionException e) {
33                 throw e.getException();
34             }
35         }
36     }
37
38     private ActionForward forceRedirect(final HttpServletRequest request,
39     final HttpServletResponse response) throws IOException {
40         // redirect to force authentication
41         String loginUrl = "/login.jsp";
42         response.sendRedirect(loginUrl);
43         reurn null;
44     }
45 }
```

Figure 23 Code source de l'aspect Authentication.

De plus, certains aspects (*Observing*, *Standards*, etc) présentés dans la table II ne définissent aucun module. Les valeurs de LCOO et de ACoh pour ces différents aspects sont égales à 0. Un autre exemple de cohésion faible est représenté par l'aspect *Observing*. La déclaration de cet aspect est présentée à la figure 24. Cet aspect ne contient aucune déclaration ; il est vide. Les auteurs définissent que cet aspect permettra d'extraire des observations des diverses entités du domaine. Cet

aspect agira comme un conteneur permettant de supporter les interfaces de *this*. La valeur des métriques de cohésion pour cet aspect est nulle (LCOO = 0 et ACoh = 0). Une cohésion parfaite est suggérée par LCOO. Avec LCOO, un aspect vide ou un aspect parfaitement cohésif fournira la même valeur de cohésion (LCOO = 0). Dans le cas d'un aspect vide, LCOO = 0. Toutefois, si nous prenons, par exemple, un aspect qui définit 3 modules tous parfaitement reliés, la valeur de la métrique devient : $LCOO = |P| - |Q| = 0 - 3 = 0$. Comme nous l'avons mentionné, la probabilité d'atteignabilité des valeurs n'est pas uniforme. On doit consulter le nombre de modules présents dans un aspect pour interpréter la valeur de LCOO. Ceci est problématique. Avec la métrique ACoh, la valeur de cohésion de cet aspect sera égale à 0, puisqu'un aspect vide ne déclare aucun module. Nous croyons que l'utilisation d'un aspect vide, dans ce contexte, est discutable.

```
1 public aspect Observing {
2 }
```

Figure 24 Code source de l'aspect Observing.

Finalement la figure 25 présente le code relatif à l'aspect *AjeeLogManager*. L'aspect *AjeeLogManager* modifie la hiérarchie des membres du package *ajee* à l'aide de : *declare parents*. Chacune des composantes du package implémentera l'interface *Loggable*. Dans nos hypothèses de base, nous n'avons pas tenu compte de la déclaration des interfaces. Ainsi, aucun module n'est défini au sein de cet aspect. Les valeurs de LCOO et de ACoh pour ces différents aspects sont égales à 0.

```
1 aspect AjeeLogManager {
2     declare parents: ajee..* implements Loggable;
3 }
```

Figure 25 Code source de l'aspect AjeeLogManager.

CHAPITRE 9

LIMITES DE L'APPROCHE ET TRAVAUX FUTURS

Bien que nous croyions que notre approche constitue une première proposition réaliste pour la mesure de la cohésion au sein des aspects, certaines hypothèses ont été prises en compte lors du développement de notre approche. Toutefois, ces hypothèses ne sont pas irréalistes; elles ont permis d'abstraire le modèle des dépendances entre les membres de l'aspect afin de faciliter le développement d'une première approche. Ainsi, notre mesure de cohésion au sein des aspects pourrait être affinée en intégrant chacune des hypothèses suivantes.

Tout d'abord, nous pensons qu'une dimension intéressante est apportée par l'héritage. Comme une classe, un aspect peut définir des relations de généralisation et de spécialisation. Un aspect concret peut spécialiser un aspect abstrait ou une classe abstraite; héritant ainsi de certaines propriétés de la super classe ou du super aspect. La représentation des patrons de conception sous forme de librairie par Hanneman et Kiczales [Han02] constitue un exemple éloquent. Ces mécanismes, ou membres hérités de l'aspect parent, auront un impact sur les relations de dépendances et influenceront la mesure de cohésion d'un aspect ACoh. En effet, les aspects concrets fournissent l'implémentation des méthodes abstraites et/ou spécifient les participants concrets avec l'implémentation de leur interface. De nouvelles relations (Modules-Données et Modules-Modules) entre les membres de l'aspect seront définies. Jusqu'ici, ces nouvelles relations définies à partir de l'héritage ne sont pas capturées par les métriques présentées dans ce travail.

De plus, ce travail s'est limité aux relations directes entre les membres de l'aspect. Il serait intéressant d'étudier la cohésion des aspects en tenant compte de leurs relations indirectes. En examinant concrètement du code, nous avons réalisé que les relations indirectes entre les membres de l'aspect contribuent à la cohésion aspect. Les relations indirectes font référence à la fermeture transitive des relations directes. La figure 26 présente un exemple de relations indirectes entre les membres d'un aspect :

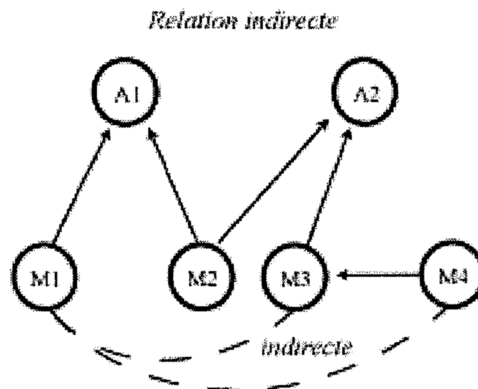

Figure 26 : Relations indirectes entre les membres aspect.

Dans cet exemple, l'aspect est représenté par trois modules : M1, M2 et M3; où le module M1 et le module M2 partagent directement/indirectement une donnée D1 et où le module M2 et le module M3 partagent directement/indirectement une donnée D2. La fermeture transitive de la relation directe suggère que M1 et M3 et M1 et M4 soient indirectement connectés (par les relations directes ou indirectes). Le concept de fermeture transitive a été utilisé par Bieman et al. [Bie95] pour la mesure de LCC (*Loose Class Cohesion*) permettant d'évaluer la cohésion au sein des classes dans un système orienté objet. Les relations indirectes entre les membres de l'aspect sont représentées à la figure 26 par un trait pointillé. Ainsi, la métrique ACoh pourrait être étendue pour tenir compte de ces relations. En tenant compte de ces relations indirectes, la mesure de cohésion serait alors égale ou supérieure (plus cohésive) que la mesure de cohésion qui ne tient compte que des relations directes. En effet, la valeur de cohésion ACoh est obtenue à partir de : $ACoh(Aspect_i) = NC(Aspect_i) / NM(Aspect_i) \in [0,1]$. Le dénominateur $NM(Aspect_i)$ représente le nombre maximum de membres connectés soit : $N * (N - 1) / 2$ relations. Il restera inchangé même si les relations indirectes sont considérées. Le $NC(Aspect_i)$ représente le nombre de membres connectés. En tenant compte des relations indirectes, le nombre de relations $NC(Aspect_i)$ sera donc égal ou supérieur. Ainsi, la cohésion entre les membres de l'aspect sera égale ou supérieur en tenant compte des relations indirectes. Ces deux métriques pourraient être implémentées séparément comme dans l'étude de Bieman [Bie95]. Notre métrique pourrait être séparée en deux

métriques de cohésion : TACoh (*Tight Aspect Cohesion*) et LACoh (*Loose Aspect Cohesion*); représentant respectivement une mesure de cohésion intégrant les définitions de relations directes et indirectes.

LACoh (*Loose Aspect Cohesion*) représente le nombre relatif de services indirectement connectés (correspondant à la fermeture transitive de la relation directe) :

LACoh = NDC(AP) / NS(AP) Є [0,1]

Cette métrique permet de capturer une dimension supplémentaire de cohésion. Reprenons l'étude de cas 1 du chapitre 7. À partir de la figure 14 qui représente les relations entre les membres de l'aspect, nous pouvons extraire un nouveau graphe des connexions entre les modules de l'aspect. La figure 27 présente le nouveau graphe représentant les connexions indirectes entre les membres de l'aspect PointShadowProtocol. Les relations indirectes sont identifiées par un trait pointillé.

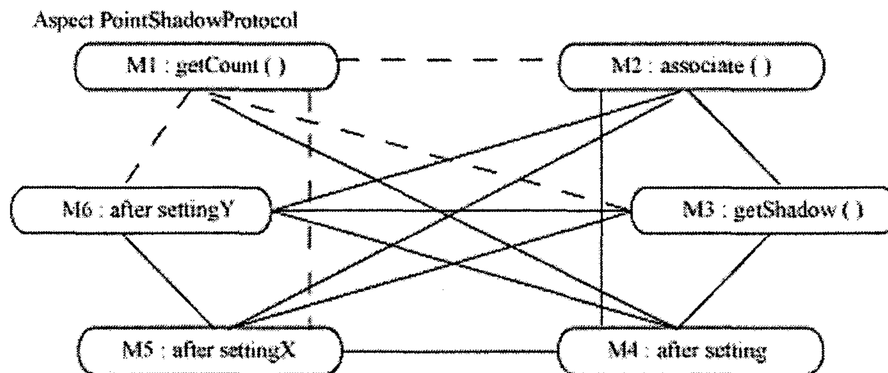

Figure 27 : Connexions indirectes entre les modules de l'aspect.

Les mesures de cohésion obtenues sont :

TACoh (PSP) = NC (PSP) / NM(PSP) = 11 / 15 = 0.73

LACoh (PSP) = NC (PSP) / NM(PSP) = 15 / 15 = 1

En considérant la fermeture transitive de la relation directe, on obtient une valeur de cohésion plus forte. Les deux mesures de cohésion peuvent être utilisées conjointement. La mesure TACoh permet de quantifier la structure des relations directes entre les membres d'un aspect et LACoh apporte un niveau d'information supplémentaire sur la qualité des relations indirectes entre les membres de l'aspect.

En reprenant l'exemple étendu de l'aspect PointShadowProtocol présenté à l'étude du cas 2 et en incluant les relations indirectes entre les membres de l'aspect, on obtient la graphe présenté à la figure 28.

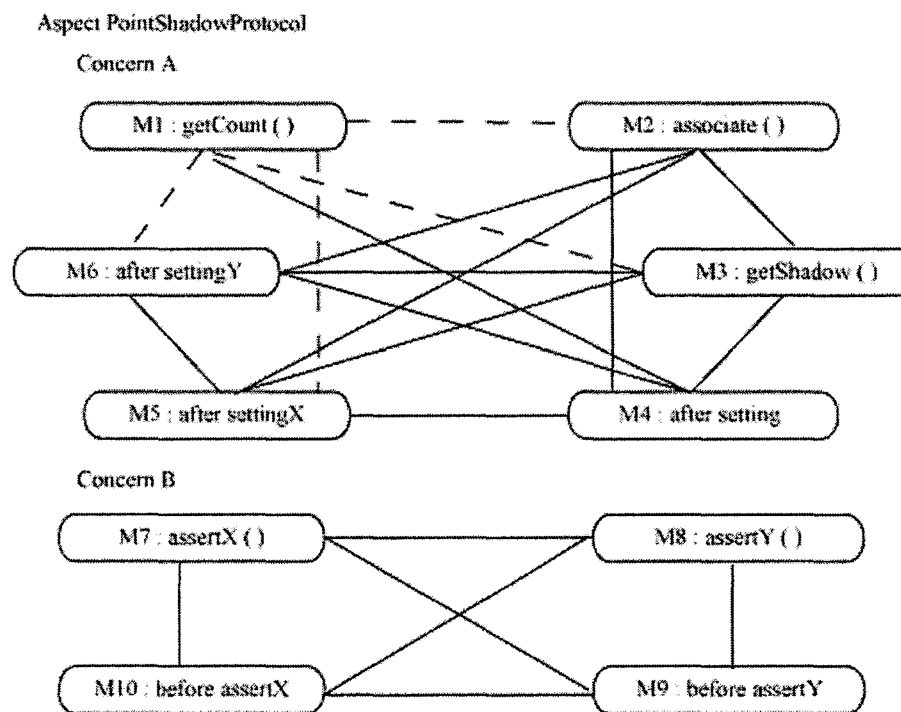

Figure 28 : Connexions indirectes entre les modules de l'aspect.

Les mesures de cohésion obtenues sont :

TACoh (PSP) = NC (PSP) / NM (PSP) = 17 / 45 = 0.38

LACoh (PSP) = NC (PSP) / NM(PSP) = 21 / 45 = 0.47

Comme on peut le remarquer, la mesure LACoh permet de mieux apprécier la qualité des relations entre les membres de cet aspect. La mesure LACoh (0.47) suggère que cet aspect est candidat à une restructuration car aucune relation indirecte n'existe entre les membres représentant les deux préoccupations. Les deux groupes de membres connectés sont isolés. Ceci est reflété par la valeur de NCM égale à 2. Pris conjointement avec TACoh et NCM, LACoh permet d'identifier des problèmes de conception au sein des systèmes aspect en apportant un niveau d'information supplémentaire.

Finalement, certaines considérations de programmation ont été mises de côté dans ce travail. Pour les études de cas et l'étude expérimentale, nous n'avons pas tenu compte des déclarations d'interfaces et des méthodes abstraites. Il s'agit de considérations ayant trait à l'implémentation des métriques (LCOO, Zhao et ACoh). Évidemment, d'autres hypothèses d'implémentation pourraient être retenues et fournir des résultats de mesure de cohésion différents de ceux que nous avons mesurés.

CHAPITRE 10

CONCLUSION

Le développement de logiciel orienté aspect constitue un nouveau paradigme de programmation du génie logiciel. Cette nouvelle approche permet de contourner certains problèmes de conception au sein des applications en améliorant la séparation des préoccupations transverses en unités modulaires appelées aspects. Toutefois, un niveau de complexité est introduit au sein des applications. Plusieurs métriques ont été proposées dans la littérature permettant d'évaluer les attributs de qualité des systèmes orientés objet. Toutefois, aucune de ces métriques ne prend en considération les nouvelles abstractions et les nouvelles dimensions introduites par le paradigme aspect.

Nous avons présenté une nouvelle approche pour la mesure de la cohésion au sein des systèmes orientés aspect. Cette approche est basée sur les dépendances entre les membres de l'aspect. L'approche proposée mesure le degré de liaison entre les modules de l'aspect à partir d'un graphe représentant la connexion entre les membres d'un aspect. La mesure de cohésion ACoh fournit un ratio entre le nombre total de paires de modules dans un aspect et le nombre maximal de connexions entre les modules de l'aspect. Le graphe considéré peut aussi être utilisé pour générer le nombre de groupes de membres connectés (NCM) au sein de l'aspect. Le nombre de groupes de membres connectés (NCM) peut aussi être utilisé comme un indicateur de manque de cohésion au sein de l'aspect. Nous croyons que cette approche représente une première proposition réaliste tenant compte des nouvelles dimensions apportées par le paradigme aspect.

Diverses études de cas ont été présentées afin de comparer notre approche aux autres approches de mesure de cohésion des systèmes aspect. La métrique ACoh est bien adaptée pour refléter les problèmes de conception; notamment lors de l'assignation de rôles disparates à un aspect. Nous croyons que l'assignation des rôles aux aspects doit suivre les mêmes règles que l'assignation des rôles aux classes dans les systèmes

orientés objet. Une assignation disparate est reflétée par la mesure de ACoh et NCM. Ces métriques permettent d'identifier des aspects candidats éventuellement à une factorisation; par exemple en séparant les rôles assignés à un même aspect à plusieurs aspects. Nous avons aussi démontré que notre proposition permet de capturer des interactions ignorées par d'autres approches.

BIBLIOGRAPHIE

[Aks98] Aksit, M. & Tekinerdogan, B. (1998). Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, *ECOOP98 Proceedings*.

[Atr05] Atrack Project Website, [En ligne]. https://atrack.dev.java.net/ (Consulté le 28 juin 2005).

[Ast05] AspectJ Team. *The AspectJ Programming Guide*. [En ligne] http://eclipse.org/aspectj/doc/released/progguide/ (Consulté le 28 juin 2005).

[Asp05] AspectJ Website [En ligne] http://eclipse.org/aspectj/ (Consulté le 28 juin 2005).

[Ass05] AspectSharp [En ligne] http://www.castleproject.org/index.php/AspectSharp (Consulté le 28 juin 2005).

[Ask05] AspectWerkz [En ligne] http://aspectwerkz.codehaus.org/ (Consulté le 28 juin 2005).

[Bad95] Badri, L., Badri, M. & Ferdenache, S. (1995). Towards Quality Control Metrics for Object-Oriented Systems Analysis, *Proceedings of TOOLS (Technology of Object-Oriented Languages and Systems) Europe'95*.

[Bad04] Badri, L. & Badri, M. (2004) A Proposal of a New Class Cohesion Criterion: An Empirical Study, *Journal of Object Technology*, vol. 3, no. 4.

[Bas96] Basili, V.R., Briand, L.C. & Melo, W. (1996). A validation of object-oriented design metrics as quality indicators, *IEEE Transactions on Software Engineering*, 22 (10), pp. 751-761.

[Bie95] Bieman, J.M. & Kang, B.K (1995). Cohesion and reuse in an object-oriented system, *Proceedings of the Symposium on Software Reusability (SSR'95)*.

[Boo93] Booch, G. (1993). *Object-Oriented Analysis and Design With Applications, Second edition*, (2e éd.). Addison Wesley Professional.

[Bri98] Briand, L.C., Daly, J. & Wusr, J. (1998). A unified framework for cohesion measurement in object-oriented systems, *Empirical Software Engineering*, Vol.3, No.1, pp. 67-117.

[Cha00] Chae, H.S., Kwon, Y. R. & Bae, D. H. (2000). A cohesion measure for object-oriented classes, *Software Practice and Experience*, No. 30, pp. 1405-1431.

[Chi94] Chidamber, S.R. & Kemerer, C.F. (1994). A Metrics suite for object Oriented Design, *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, pp. 476-493.

[Duf03] Dufour, B., Goard, C., Hendren, L., Verbrugge, C., De Moor, O. & Sittampalam, G. (2003). Measuring the Dynamic Behavior of AspectJ Programs, *Sable Technical Report,* No.2003-8.

[Gam95] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[Gel04] Gélinas, J.F., Badri, L. & Badri, M. (2004). Aspect cohesion measurement based on dependence analysis, *Proc.of the 8th ECOOP*

Workshop on quantitative approaches in object-oriented software engineering (QAOOSE2004), Oslo, Norway.

[Gel05] Gélinas, J.F., Badri, L. & Badri, M. (2005). Mesuring cohesion in aspect-oriented systems, *Proc.of the IASTED International Conference on Software Engineering (SE2005)*, Innsbruck, Austria.

[Gra87] Grady, R. B. & Caswell, D. R. (1987). *Software Metrics: Establishing a Company-Wide Program*, Engle- wood Cliffs, N. J.: Prentice-Hall.

[Han02] Hannemann, J. & Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. *In Proceedings of the 17th Annual Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 161–173.

[Hen95] Henderson-Sellers, B. (1995). *A book of Object-Oriented Knowledge*, (2e éd.), Sydney : Prentice Hall,.

[Hen96] Henderson-Sellers, B. (1996). *Object-Oriented Metrics Measures of Complexity*, Sydney : Prentice-Hall.

[Hit95] Hitz, M. & Montazeri, B. (1995). Measuring coupling and cohesion in object oriented systems, *Proceedings of the Int. Symposium on Applied Corporate Computing*, pp. 25-27.

[Jav05] Java Website, [En ligne]. http://java.sun.com/ (Consulté le 28 juin 2005).

[Jcc05] JavaCC Website, [En ligne]. https://javacc.dev.java.net/ (Consulté le 28 juin 2005).

[Kab00] Kabaili, H., Keller, R.K., Lustman, F. & Saint-Denis, G. (2000). Class Cohesion Revisited : An Empirical Study on Industrial Systems, *Proceeding of the Workshop on Quantitative Approaches Object-Oriented Software Engineering, QAOOSE'2000.*

[Kic01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. & Griswold, W. G. (2001). An Overview of AspectJ, *European Conference on Object-oriented Programming*, Lecture Notes in Computer Science, p. 327–353, volume 2072, Springer.

[Kic97] Kiczales, G., Lamping J., Mendhekar A. (1997). *Aspect-Oriented Programming*, Xerox PARC, Palo Alto, CA. June.

[Kic04] Kiczales, G. (2004). *It's the crosscutting*, Software Development Magazine, February.

[Lar04] Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*, (2e éd.), Prentice-Hall.

[Li93] Li, W. & Henry, S. (1993). Object oriented metrics that predict maintainability, *Journal of Systems and Software*, Vol. 23, pp. 111-122, February.

[Lop97] Lopes C. V. (1997). *D: A Language Framework for Distributed Programming, Xerox PARC,* Palo Alto, CA. November, 1997.

[Men97] Mendhekar, A. & Kiczales G. (1997). *RG: A Case Study for Aspect-Oriented Programming*, Xerox PARC, Technical report SPL97-009 P9710044, February.

[Per81] Perlis, A., Sayward, F. & Shaw, M. (1981) *Metrics: An Analysis and Evaluation. Cambridge*, Mass.: MIT Press.

[Pre01] Pressman, R. S. (2001). *Software Engineering, A practitioner's approach,* (5e éd.), McGraw Hill.

[Sab04] Sabbah, D. (2004). From Promise to Reality, *Proceedings of the 3rd international conference on Aspect-oriented software development* (AOSD'04), p.1-2.

[San03] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C. & Von Staa, A. (2003). On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. *Proc. XXIII Brazilian Symposium on Software Engineering.*

[Wol74] Wolverton, R. W (1974). The Cost of Developing Large-Scale Software, *IEEE Trans. Computers C-23*, 615-636.

[You79] Yourdon, E. & Constantine, L. (1979). *Structured Design*, Prentice Hall, Englewood Cliffs.

[Zac03] Zacaria, A. & Hosny, H. (2003). Metrics for Aspect-Oriented Software Design. *Proc. Third International Workshop on Aspect-Oriented Modeling, AOSD'03*.

[Zha03] Zhao, J. (2003). Coupling Measurement in Aspect-Oriented Systems, *Technical-Report SE-142-6*, Information Processing Society of Japan (IPSJ).

[Zha04] Zhao, J. & Xu, B. (2004). Measuring Aspect Cohesion, *Proc. International Conference on Fundamental Approaches to Software Engineering* (FASE'2004), LNCS 2984, pp.54-68.