Université de Montréal

Une architecture pour l'évaluation qualitative de l'impact de la programmation orientée aspect

par
Jean-Yves Guyomarc'h

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Mai, 2006

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Une architecture pour l'évaluation qualitative de l'impact de la programmation orientée aspect

présenté par :

Jean-Yves Guyomarc'h

a été évalué par un jury composé des personnes suivantes :

Julie Vachon
président-rapporteur

Yann-Gaël Guéhéneuc
directeur de recherche

Philippe Langlais
membre du jury

Mémoire accepté le

RÉSUMÉ

Les limitations de la programmation orientée objet ont poussé les chercheurs en génie logiciel à préconiser une réelle « séparation des préoccupations » qui permette, entre autres, une meilleure modularité et réutilisabilité du code source. Ce souci a donné naissance à la programmation orientée aspect, qui propose d'encapsuler dans des aspects les préoccupations incompatibles avec la logique métier des objets. Les aspects sont capables d'injecter ensuite le code nécessaire à ces préoccupations en utilisant leurs propres mécanismes (points de coupure, greffons et introductions).

La naissance d'un nouveau paradigme implique son étude en termes de qualité. Dans le cas de la programmation orientée aspect, il faut aussi tenir compte de son impact sur les programmes orientés objet. Nous proposons de quantifier cet impact par l'utilisation des métriques de classe. Le calcul de ces métriques est modifié pour refléter les répercussions des différents mécanismes de la programmation orientée aspect. Ceci nous permet d'implanter une architecture pouvant produire des jeux de métriques avant et après introduction des aspects.

Grâce à ces résultats, nous sommes en mesure de mener une expérience en utilisant la visualisation. Cette expérience prouve que des ingénieurs en génie logiciel peuvent évaluer l'impact de la programmation orientée aspect en termes de qualité et que la restructuration d'un programme orienté objet par les aspects peut améliorer sa qualité.

Mots clés : génie logiciel, qualité, séparation des préoccupations, aspect, métrique, visualisation, Java, AspectJ.

ABSTRACT

The limitations of object-oriented programming forced software engineers to seek a real "separation of concerns", which would allow a better modularity and re-utilisability of the source code. This was made possible by the creation of the aspect-oriented programming. It aims to encapsulate the concerns, which are incompatible with the objects business logic, in an abstraction called aspect. Then the aspects use their own mechanisms (pointcuts, advices and inter-type introductions) to introduce the code needed by these concerns.

The creation of a new paradigm implies its study in terms of quality. For aspect-oriented programming, it is also necessary to consider its impact on object-oriented programs. We propose to quantify this impact using class metrics. The computation of these metrics is modified to reflect the repercussions of the various mechanisms of the aspect-oriented programming. This allows the implementation of an architecture to produce sets of metrics, before and after the introduction of aspects.

With these results, we undertook an experiment using visualization. The conclusions of this experiment proves that software engineers can evaluate aspect-oriented programming's impact in terms of quality and that reengineering object-oriented programs through the use of aspects can improve their quality.

Keywords: software engineering, quality, separation of concerns, aspect, metrics, visualization, Java, AspectJ.

TABLE DES MATIÈRES

RÉSUMÉ . . . . . **iii**

ABSTRACT . . . . . **iv**

TABLE DES MATIÈRES . . . . . **v**

LISTE DES TABLEAUX . . . . . **viii**

LISTE DES FIGURES . . . . . **ix**

LISTE DES CODES SOURCE . . . . . **x**

LISTE DES APPENDICES . . . . . **xi**

LISTE DES SIGLES . . . . . **xiii**

NOTATION . . . . . **xiv**

DÉDICACE . . . . . **xv**

REMERCIEMENTS . . . . . **xvi**

INTRODUCTION **1**

CHAPITRE 1 : ÉTAT DE L'ART **5**

1.1 La programmation orientée aspect 5

1.1.1 Les limites de la programmation orientée objet 5

1.1.2 Une solution : la programmation orientée aspect 6

1.1.3 Exemple du FigureElement 7

1.1.4 Implémentation : AspectJ 10

1.1.5 Relations entre la POA et la POO 14

1.2 Qualité 15
1.2.1 Qualité et évaluation 15
1.2.2 Métriques 16
1.2.3 Programmation orientée aspect et qualité 20
1.3 Notre approche 22

CHAPITRE 2 : ÉTUDE DES MÉTRIQUES 24
2.1 Introduction 24
2.1.1 Protocole d'étude 24
2.1.2 Notation et diagrammes 25
2.2 Étude des métriques 25
2.2.1 Weighted Methods per Class 26
2.2.2 Depth of Inheritance Tree 28
2.2.3 Number Of Children 30
2.2.4 Coupling Between Object 31
2.2.5 Response For a Class 33
2.2.6 Lack of Cohesion in Methods 35
2.2.7 Number of Attributes Declared 37
2.2.8 Number of Methods Declared 38
2.2.9 Number of Methods Overridden 39
2.2.10 Specialisation IndeX 40
2.3 Conclusion sur l'étude des métriques 41

CHAPITRE 3 : IMPLÉMENTATION ET ARCHITECTURE . . . 43
3.1 Introduction 43
3.2 Modélisation 44
3.2.1 Le métamodèle PADL 44
3.2.2 Extension du métamodèle PADL 45
3.2.3 Processus de traitement 48

3.3 Extraction des métriques 49
3.3.1 Le module d'extraction de métriques POM 49
3.3.2 Extension de POM 50
3.3.3 Problèmes rencontrés 54
3.3.4 Optimisations 54
3.3.5 Processus de traitement 55
3.4 Mise en page et présentation des résultats 57
3.4.1 Processus de traitement 58
3.4.2 Critiques 58
3.5 Conclusion 59

CHAPITRE 4 : TESTS ET VALIDATION 61
4.1 Objectifs 61
4.2 Tests et validation 61
4.2.1 Tests unitaires 62
4.2.2 Validation 63
4.3 Évaluation 65
4.3.1 Apprentissage automatique 65
4.3.2 Visualisation 66
4.3.3 Expérience 68
4.3.4 Conclusion sur l'évaluation 71

CONCLUSION 72

BIBLIOGRAPHIE 75

LISTE DES TABLEAUX

1.1 Les différents types de greffons 12
1.2 Les métriques de Chidamber et Kemerer 19

2.1 Incidence des mécanismes de la POA sur les métriques étudiées . . 42

4.1 Corpus utilisé pour la validation 64
4.2 Résultats de l'évaluation 70
4.3 Les métriques utilisées par les évaluateurs 70

I.1 Point de coupure – Appel de méthodes et de constructeurs 83
I.2 Point de coupure – Exécution de méthodes et de constructeurs . . . 84
I.3 Point de coupure – Accès aux variables 84
I.4 Point de coupure – Gestionnaire d'exceptions 85
I.5 Point de coupure – Initialisation de classe 85
I.6 Point de coupure – Basés sur la structure lexicale 85
I.7 Point de coupure – Flot de contrôle 85
I.8 Point de coupure – Contexte et arguments 86
I.9 Point de coupure – Structure conditionnelle 86

LISTE DES FIGURES

1.1 Encapsulation des préoccupations transversales 7
1.2 Exemple `FigureElement` 7
1.3 Exemple `FigureElement` : utilisation du patron `Observer` 9
1.4 Exemple `FigureElement` : approche orientée aspect 9

2.1 Ajout de l'interface `DisplayableFigure` à la classe `Figure` 28
2.2 Modification de la hiérarchie de la classe `Curve` 29
2.3 DIT : hiérarchie originale 30
2.4 DIT : hiérarchie modifiée par la POA 30

3.1 Sous-ensemble simplifié du métamodèle PADL 45
3.2 Ajout de l'entité *Aspect* dans le métamodèle PADL 47
3.3 Schéma du processus de modélisation du code source AspectJ . . . 49
3.4 Architecture simplifiée de la bibliothèque POM 50
3.5 Schéma du processus d'extraction des métriques 57
3.6 Schéma du processus de présentation 59
3.7 Schéma du processus global : modélisation, extraction et présentation 60

4.1 Méthode de comparaison POO/PAO 61
4.2 La vue du cœur du système, avant introduction des aspects 67
4.3 La vue du système après introduction des aspects 68

III.1 Diagramme de classe pour l'application TELECOM 93
III.2 Le mécanisme d'introduction avec AspectUML 94

LISTE DES CODES SOURCE

1.1 Déclaration d'un point de coupure 11
1.2 Un greffon de type *after* 12
1.3 L'aspect DisplayAspect 13
1.4 La classe DisplayManager 13
2.1 La classe `Figure` 33
2.2 L'aspect `LoggingAspect` 33
3.1 Implémentation originale de la métrique NAD dans POM 51
3.2 L'aspect NADAspectMetric 52
3.3 L'aspect NODAspectMetric 56
4.1 Assertions définies pour les tests unitaires 63
V.1 Schema XSD de POMXML 96

LISTE DES APPENDICES

Annexe I : **AspectJ – points de coupure** **83**

Annexe II : **Théorie des ensembles** **87**

Annexe III : **Profil UML – AspectUML** **92**

Annexe IV : **Compagnon de lecture** **95**

Annexe V : **POMXML – schema XSD** **96**

Annexe VI : **Expérience – questionnaire** **99**

LISTE DES SIGLES

CBO	*Coupling Between Object*
DIT	*Depth of Inheritance Tree*
GQM	*Goal-Question-Metrics*
LCOM	*Lack of Cohesion in Methods*
LOC	*Lines of codes*
NAD	*Number of Attributes Declared*
NMD	*Number of Methods Declared*
NMO	*Number of Methods Overriden*
NOC	*Number Of Children*
NOD	*Number Of Descendents*
NOP	*Number Of Parents*
OCL	*Object Constraint Language*
PADL	*Pattern and Abstract-Level Description Language*
POA	Programmation Orientée Aspect
POM	*Primitives, Operators, Metrics*
POO	Programmation Orientée Objet
RFC	*Response For a Class*
SIX	*Specialisation IndeX*
UML	*Unified Modeling Language*
VERSO	Visualisation pour l'Évaluation et la Ré-ingénierie des Systèmes à Objet
WMC	*Weighted Methods per Class*
XML	*eXtended Markup Language*

NOTATION

$\oslash$	ensemble vide
$\in$	appartenance
$\cup$	réunion
$\cap$	intersection
$\subset$	inclusion
$\sum_{i=1}^{n}$	somme de i à n
$\lvert E \rvert$	cardinalité de l'ensemble E

À Mathis.

REMERCIEMENTS

J'aimerais tout d'abord remercier mon directeur de recherche, Yann-Gaël Guéhéneuc, qui m'a soutenu tout long de cette maîtrise. Je remercie aussi tous les membres du laboratoire GEODES et tout particulièrement Houari Sahraoui, qui m'a donné le goût d'étudier le génie logiciel, ainsi que Guillaume, Karim et Salima pour s'être portés volontaires pour mes expérimentations.

Merci à mes parents, sans qui tout cela n'aurait pas été possible, et à Marie-Hélène pour avoir été à mes côtés au quotidien. Je remercie Catherine et Siegfried pour leur patience et leur bonne humeur.

Enfin merci à toutes ces personnes, du Québec et d'outre-Atlantique, qui m'ont fait confiance et ont cru en moi.

INTRODUCTION

Les langages orientés objet, dits de troisième génération, sont devenus depuis les trente dernières années les standards en matière de programmation et de production de logiciels. En voulant modéliser la réalité le plus fidèlement possible, grâce aux concepts de classe et d'objet, ils ont amené de nouveaux défis pour le domaine du génie logiciel. Les ingénieurs en logiciel ont dû définir de nouvelles méthodes et de nouveaux outils dans le but d'évaluer la qualité des logiciels pour aider aussi bien les gestionnaires que les développeurs. Les travaux de pionniers, tels que Fenton [1996] ou encore Chidamber et Kemerer [1994], ont révolutionné le domaine en établissant des techniques et des métriques plus adéquates à l'évaluation des programmes orientés objet.

Ces langages de troisième génération ont cependant fait l'objet de sérieuses critiques. Leur inhabilité à implémenter correctement les préoccupations transversales, telles la journalisation ou la sécurité, entraîne, entre autres, une mauvaise modularité des programmes orientés objet et les rend difficilement réutilisables. C'est dans le but de palier à ces problèmes et de réaliser une réelle « séparation des préoccupations » [Parnas, 1972] qu'est né le nouveau paradigme de la programmation orientée aspect (POA) [Kiczales *et al.*, 2001b].

La programmation orientée aspect a été créée pour capturer les préoccupation transversales à un programme orienté objet. Elle propose d'introduire le concept d'*aspect* dans ces systèmes. Un aspect possède les mêmes mécanismes qu'une classe, ce qui lui permet d'encapsuler une préoccupation aussi bien qu'une classe. Afin d'intervenir sur les objets du programme et d'y injecter le comportement nécessaire à la préoccupation qu'il implémente, un aspect dispose de ses propres mécanismes. Il définit, grâce à des *points de coupure*, des endroits précis dans l'exécution d'un programme. Il peut ensuite prendre le contrôle à ces endroits et y injecter le code correspondant à l'action à réaliser via le mécanisme de *greffon*. Il lui est aussi

possible de modifier la structure d'une classe ou l'arbre de hiérarchie au moyen d'*introductions*.

La naissance d'un nouveau paradigme nécessite de nouvelles méthodes et de nouveaux outils pour en évaluer la qualité. Ces outils et ces méthodes permettent, d'une part, d'assurer son intégration dans le milieu industriel et, d'autre part, donnent les moyens nécessaires aux chercheurs en génie logiciel de l'étudier et de l'appréhender. La relation liant programmation orientée aspect et programmation orientée objet nous pousse à aborder l'étude de la qualité différemment. L'impact de la POA sur les programmes orientés objet peut avoir des répercussions sur leur qualité. Les outils d'évaluation de la qualité que nous possédons aujourd'hui ne sont pas capables de nous renseigner sur ces répercussions. Nous avons quantifié cet impact pour pouvoir l'évaluer en termes de qualité. Dans le prolongement des travaux de Sant'Anna *et al.* [2005] et de Zakaria et Hosni [2003], nous avons étudié et formalisé l'impact de la POA sur dix métriques de classes : celles de Chidamber et Kemerer [1994] et certaines de Lorenz et Kidd [1994]. Une fois cet impact capturé et les calculs des métriques formalisés, nous avons implanté une solution logicielle dans le but d'appliquer concrètement nos travaux à l'évaluation de l'impact de la POA.

Notre application, *PADL Aspect Metric*, se décompose en trois phases : *modélisation*, *extraction*, et *présentation*. Pour modéliser un programme orienté aspect, nous avons étendu le métamodèle PADL (*Pattern and Abstract-Level Description Language* [Guéhéneuc, 2003]) permettant d'analyser des programmes Java et y avons ajouté le support du langage AspectJ [AspectJ, 2002], l'implémentation de référence de la POA pour Java. À partir du modèle étendu, il a été possible d'implémenter les calculs des métriques de classes pour calculer l'impact de la POA. Le module POM (*Primitives, Operators, Metrics* [Zaidi, 2004]) permet d'extraire ces mêmes métriques sur un modèle PADL classique. Nous avons donc utilisé la POA pour injecter sur ce module les modifications nécessaires au déploiement de

nos calculs. L'utilisation de la POA dans ce cas permet une réutilisation optimale du module POM. Enfin, la phase de présentation nous permet de transformer les résultats extraits dans un format XML compréhensible par plusieurs médias (navigateur internet ou outil de visualisation, par exemple VERSO (Visualisation pour l'Évaluation et la Ré-ingénierie des Systèmes à Objet [Langelier *et al.*, 2005])).

Il est donc possible, grâce à notre application *PADL Aspect Metric*, d'analyser complètement un programme orienté aspect. Nous obtenons un premier jeu de métriques avant introduction des aspects en analysant seulement la partie objet du programme et un second jeu de métriques reflétant l'impact de la POA en analysant le programme en entier. C'est en comparant les deux jeux de métriques qu'il est possible de juger de la qualité du programme orienté aspect.

À cause de la relative jeunesse du paradigme aspect, il a été très difficile de réunir un corpus de test conséquent. Ce manque d'exemples nous a empêché d'utiliser les techniques classiques du génie logiciel expérimental, tel que l'apprentissage automatique de règles, pour créer un modèle de qualité pour la POA. Nous avons donc établi notre expérience en utilisant la visualisation des programmes et de leurs métriques en trois dimensions [Langelier *et al.*, 2005]. Les résultats obtenus nous ont permis de conclure que l'impact de la POA sur un programme orienté objet, une fois quantifié à l'aide de métriques de classe modifiées, pouvait être évalué en termes de qualité et comparé, lorsque possible, à son équivalent orienté objet.

Thèse

La modification d'un système orienté objet, par l'apport de la programmation orientée aspect, peut-elle être quantifiée puis évaluée en termes de qualité ?

Plan

Dans le chapitre 1 nous présentons les connaissances et les travaux antérieurs sur l'analyse de la qualité des programmes aspect. Nous y introduisons la programmation orientée aspect, sa relation avec le domaine de la qualité et, plus précisément, l'étude des métriques. Enfin, nous introduisons notre propre approche. Le chapitre 2 détaille notre étude de l'impact de la programmation orientée aspect sur les métriques de classe orientées objet. Ces travaux sont les fondements théoriques nécessaires pour répondre à notre question de recherche. Le chapitre 3 fait état des solutions logicielles, ainsi que de leur architecture, que nous avons implémentées afin de capturer de façon concrète l'impact de la programmation orientée aspect sur les programmes orientés objet. Le chapitre 4 met en pratique et évalue concrètement l'application présentée dans le chapitre 3 et d'évaluer la qualité des programmes orientés aspect. Enfin, nous conclurons ce mémoire et proposerons diverses directions de recherche.

CHAPITRE 1

ÉTAT DE L'ART

Dans ce chapitre, nous présentons les connaissances et les travaux antérieurs qui nous ont permis de réaliser notre travail. Nous y présentons la programmation orientée aspect, sa relation avec le domaine de la qualité et, plus précisément, l'étude des métriques. Enfin, nous introduisons notre propre approche.

1.1 La programmation orientée aspect

1.1.1 Les limites de la programmation orientée objet

Le génie logiciel nous dit que chaque tâche logicielle spécifique devrait être la seule responsabilité d'une classe ou d'un petit nombre de classes éventuellement rassemblées au sein d'un module (groupement logique).

En programmation orientée objet, beaucoup de lignes de code sont consacrées à la synchronisation des accès aux ressources, à l'optimisation de l'utilisation des ressources, à la sauvegarde de certaines données dans des bases de données (*i.e.*, la persistance), à la journalisation des états du programme, à la vérification des paramètres entrants, au traitement des exceptions, etc. Ces activités sont des préoccupations transversales au système (*crosscuting concerns*) détaillées dans les travaux de Lopes [1997] et de Mendhekar [1997].

Ces lignes de code ont tendance à être présentes dans la plupart des méthodes des classes d'un programme. Cela crée du code redondant qui ne peut être encapsulé dans un même module (*tangled code*, classification de Hannenberg [2002]). Alors les classes sont, selon Kiczales *et al.* [2001b], peu réutilisables telles quelles par d'autres programmes (autrement dit, les classes sont couplées avec le programme). Cela implique aussi que le changement de politique sur un de ces aspects de l'application

(par exemple si on décide de ne plus journaliser certaines informations) oblige la modification de nombreuses lignes de code éparpillées dans le code source, ce que l'on pourrait qualifier de un problème de localité textuelle : tout code concernant le même aspect devrait être regroupé dans un même endroit. Cela se rapproche du concept de *modular continuity* tel que présenté par Meyer [1997, p. 39] :

> *A method satisfies Modular continuity if, in the software architectures that it yields. a small change in a problem specification trigger a change of just one module, or a small number of modules.*

Ces limites de la programmation orientée objet ont fait l'objet d'études dans le domaine du génie logiciel et ont donné lieu à la naissance de concepts tels que « la séparation des préoccupations » (*separation of concerns*, [Parnas, 1972]). Aujourd'hui, une solution concrète à ces problèmes semble possible.

1.1.2 Une solution : la programmation orientée aspect

Introduite en 1997 par Kiczales [1997], la programmation orientée aspect a pour but de palier aux limites de la programmation orientée objet. Ce nouveau paradigme vient augmenter la programmation orientée objet et, pour cette raison, n'est pas destiné à être indépendant.

Le principal apport de la programmation orientée aspect est le concept d'*aspect*. Cette abstraction permet de capturer les différentes préoccupations transversales d'un système, telles la journalisation et la persistance. La figure 1.1 [Smacchia et Vaucouleur, 2003] illustre cette capacité. La partie gauche de cette figure représente le code des méthodes d'un programme orienté objet. Différents tons de gris sont associés aux lignes de code selon la préoccupation qu'elles implémentent (logique métier, synchronisation, sécurité et persistance). La partie droite montre le même programme restructuré grâce à la POA. On peut voir que le code des méthodes a été nettoyé pour n'y laisser que la logique métier, et les autres préoccupations encapsulées dans des *aspects*.

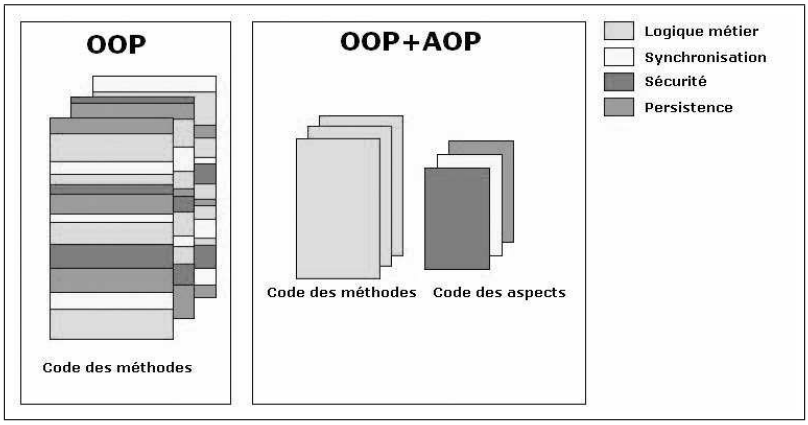

FIG. 1.1: Encapsulation des préoccupations transversales

Grâce à des mécanismes propres à la programmation orientée aspect, un aspect peut intervenir en un point quelconque de l'exécution d'un programme orienté objet. L'aspect peut alors y injecter le comportement nécessaire à la réalisation de la préoccupation transversale qu'il modélise.

1.1.3 Exemple du FigureElement

Afin d'illustrer le fonctionnement de la programmation orientée aspect, nous présentons ci-dessous un exemple tiré d'une entrevue donnée par Kiczales [2003]. Cet exemple sera utilisé tout au long de ce mémoire.

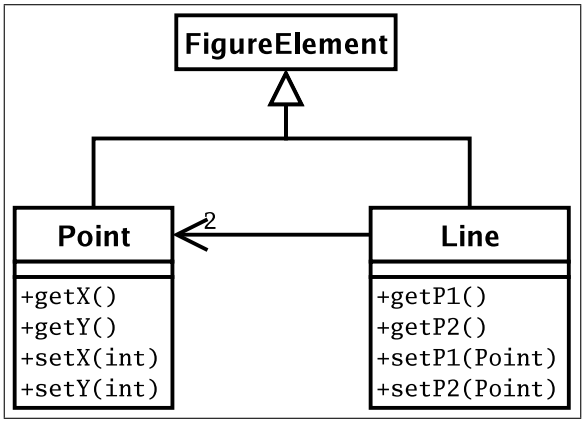

FIG. 1.2: Exemple `FigureElement`

Dans cet exemple, décrit par le diagramme UML (*Unified Modeling Language*)de la figure 1.2, nous retrouvons deux classes, `Point` et `Line`, qui partagent une même interface `FigureElement`. De plus, le diagramme nous indique que la classe `Line` est composée de deux objets de type `Point`. Les méthodes de ces deux classes, préfixées par `set`, modifient l'état de l'objet.

Nous pouvons imaginer que ces différentes classes font partie d'un logiciel de dessin vectoriel, tel que l'application JHotDraw[1]. Dans un tel cas, la création d'un point ou d'une ligne dans l'éditeur aura pour effet d'instancier les classes `Point` ou `Line` et d'associer ces nouveaux objets à une représentation graphique à l'écran. Cette association doit être faite par un composant du logiciel que nous appelons `DisplayManager`. Nous pouvons alors présenter le scénario suivant : un appel à l'une des méthodes préfixées par `set` sur un objet de type `FigureElement` est déclenché par l'un des acteurs du programme. Il faut alors que le `DisplayManager` en soit averti afin de pouvoir répercuter les modifications au niveau de la représentation graphique. Pour implémenter ce scénario, il est possible d'adopter soit une approche orientée objet, soit une approche orientée aspect.

Approche orientée objet. Une solution simple est de faire une mise à jour du `DisplayManager` directement depuis chacune des méthodes préfixées par `set`. Cette solution entraîne bien entendu des problèmes en termes de réutilisation ou de maintenance.

Ce genre de problème a été étudié par la communauté du génie logiciel et a donné naissance aux patrons de conception [Gamma, 1995]. Dans le cas présent, nous pourrions utiliser le patron *Observer* afin d'informer le `DisplayManager` de tous changements intervenant au niveau des objets qu'il gère. La figure 1.3 illustre cette solution. L'utilisation de tels patrons améliore grandement la conception d'une application sans toutefois régler tous les problèmes cités précédemment.

[1] `http://www.jhotdraw.org`

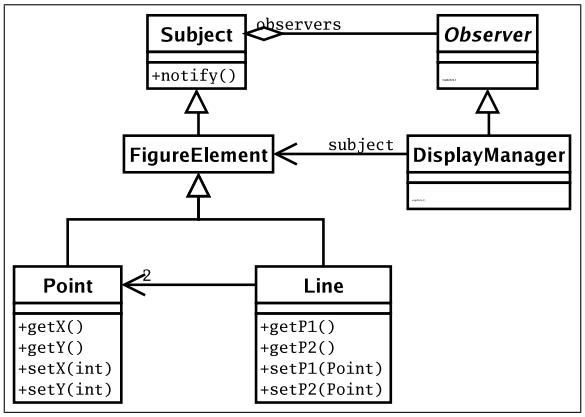

FIG. 1.3: Exemple `FigureElement` : utilisation du patron `Observer`

Approche orientée aspect. Une approche orientée aspect offre une solution qui ne modifie pas les classes déjà existantes.

Un aspect `DisplayAspect` est créé. Celui-ci déclare un point de jonction qui représente l'appel à n'importe quelle méthode préfixée par `set` et appartenant soit à un objet de type `Point` soit à un objet de type `Line`. Ce point de jonction est associé à un greffon (*advice*) qui a pour fonction d'informer le `DisplayManager` du changement. La figure 1.4 représente de façon conceptuelle l'introduction d'un aspect dans notre système.

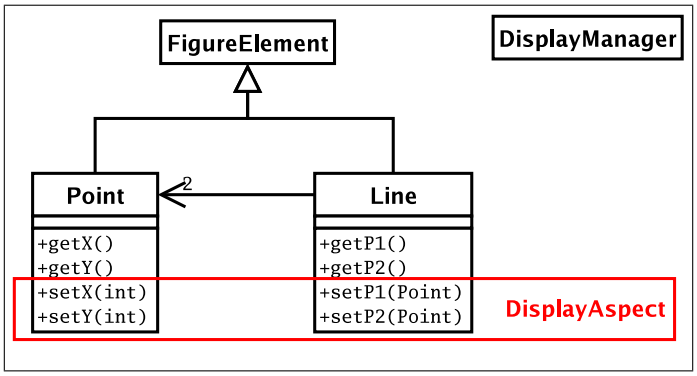

FIG. 1.4: Exemple `FigureElement` : approche orientée aspect

1.1.4 Implémentation : AspectJ

1.1.4.1 Introduction

Le paradigme s'applique aux langages orientés objet de façon générale. On en trouve des implémentations spécifiques à des langages tels que C++ ou encore C♯.

AspectJ est l'une des implémentations du paradigme orienté aspect. Développé au laboratoire de Xerox PARC par Kiczales et son équipe [2001a], c'est une implémentation de la programmation orientée aspect appliquée au langage Java.

Nous présentons ci-dessous les principaux mécanismes et abstractions qui composent AspectJ [Laddad, 2002; Gradecki et Lesiecki, 2003].

1.1.4.2 Point de jonction (*Join Point*)

Le point de jonction, ou *join point*, est le concept qui sous-tend le paradigme aspect. Il permet de définir un point précis dans l'exécution d'un programme afin d'y injecter une action. AspectJ permet de spécifier plusieurs types de points de jonction [Gradecki et Lesiecki, 2003] :

- appel et exécution de méthodes ;
- appel et exécution de constructeurs ;
- accès en écriture ou en lecture d'un attribut ;
- exécution d'un gestionnaire d'exception ;
- initialisation d'une classe ou d'un objet.

AspectJ permet de déclarer ces points de jonction via le mécanisme de point de coupure.

1.1.4.3 Point de coupure (*Pointcut*)

Le point de coupure, déclaré grâce au mot clé `pointcut`, est le mécanisme qui permet de déclarer les points de jonction. L'extrait de code 1.1 présente la déclaration d'un point de coupure.

```
1 pointcut pointMoved () : call(public void Point.set*(int)) ;
```

Code 1.1: Déclaration d'un point de coupure

Un point de coupure, appelé `pointMoved`, est déclaré. Il représente un appel à une méthode préfixée par `set` de la classe `Point`. De plus, cette méthode doit être de type `public` et retourner la valeur `void`. Il est précisé que le point de coupure fait référence à un appel de méthode grâce au prédicat `call`. L'annexe I (p. 83) présente les différents types de points de coupure implémentés par AspectJ.

Les points de coupure offrent donc un mécanisme puissant et souple qui permet de cibler n'importe quel point dans l'exécution d'un programme orienté objet. Il est aussi possible de récupérer le contexte, c'est-à-dire l'objet appelant ou appelé, grâce aux prédicats `target` ou `this`. L'utilisation des opérateurs logiques permet de combiner plusieurs points de jonction ou points de coupure à l'intérieur d'un même point de coupure. Afin d'associer une action (code) à un point de coupure, la paradigme aspect introduit la notion de greffon.

1.1.4.4 Greffon (*Advice*)

Un greffon ou *advice* permet d'encapsuler le code à exécuter et de l'associer à un point de coupure précédemment déclaré. Ce mécanisme peut être de trois types, résumés dans le tableau 1.1. Les deux premiers types de greffon, *before* et *after*, permettent de spécifier que l'action à prendre (le code à injecter) doit être exécutée respectivement avant ou après le point de coupure associé au greffon. Le troisième type de greffon, appelé *around*, a un comportement plus particulier. En effet, l'action (le code) qu'il déclare remplace le code associé au point de coupure. Si le point de coupure est un appel ou une exécution de méthode qui fournit une valeur de retour, le code du greffon remplacera cette méthode et devra calculer et retourner une valeur adéquate. Afin de donner la possibilité au développeur d'exécuter quand même le code remplacé, AspectJ offre le prédicat `proceed()` qui

```
1 after() : pointMoved () {}
2     //{Aviser le DisplayManager du changement}
3 }
```

Code 1.2: Un greffon de type *after*

ne peut être utilisé qu'à l'intérieur d'un greffon de type *around*.

after	le greffon est exécuté après le point de coupure associé
before	le greffon est exécuté avant le point de coupure associé
around	le greffon est exécuté *à la place* du point de coupure associé. L'instruction `proceed()` permet au greffon d'exécuter les instructions remplacées et d'en récupérer les résultat.

Tab. 1.1: Les différents types de greffons

L'extrait de code 1.2 présente un greffon associé au point de coupure `pointMoved`. Ce greffon permet de déclarer qu'après l'appel à une méthode de la classe `Point` préfixée par `set`, il faut exécuter certaines instructions ayant pour but d'aviser le composant en charge de l'affichage qu'un objet de type `Point` a été déplacé.

1.1.4.5 Mécanisme d'introduction (*Inter-Type declaration*)

Le mécanisme d'introduction, ou *Inter-Type declaration*, du paradigme aspect autorise le développeur à déclarer des membres (méthodes ou attributs) dans des classes déjà existantes ou à modifier la hiérarchie des classes et des interfaces. Les extraits de code 1.3 et 1.4 présentent un exemple d'introduction. Dans cet exemple, nous décidons que les points et les lignes ont maintenant besoin d'être colorés. Pour ce faire, l'aspect `DisplayAspect` introduit un attribut appelé `color` de type `java.awt.Color` à la classe `FigureElement`. Cette introduction est concrètement réalisée par la ligne numéro 5 de l'extrait de code 1.3. Des accesseurs en lecture et en écriture pour le nouvel attribut `color` de la classe `Point` sont aussi ajoutés. Il est alors possible de déclarer un point de coupure `newFE()` (extrait de code 1.3, ligne 17) réagissant à la création d'un objet de type `FigureElement` et de lui associer un greffon qui gère la couleur donnée par le `DisplayManager`.

```
1import java.awt.Color;
2
3public aspect DisplayAspect {
4
5    private Color FigureElement.color =
6                                Color.BLACK;
7
8    public void FigureElement.setColor
9                                (Color color){
10       this.color = color;
11   }
12
13   public Color FigureElement.getColor(){
14       return this.color;
15   }
16
17   pointcut newFE(FigureElement fe) :
18       this(fe) &&
19       execution((Point || Line).new(..));
20
21   after(FigureElement fe): newFE(fe) {
22       this.init(fe);
23   }
24
25   private void init(FigureElement fe){
26       fe.setColor(DisplayManager
27                   .getDisplayManager()
28                   .getSystemColor
29                           (fe.getType()));
30       DisplayManager.getDisplayManager()
31                    .init(fe);
32   }
33}
```

Code 1.3: L'aspect DisplayAspect

```
1public class DisplayManager {
2
3    public void init(FigureElement fe){
4        //init code
5    }
6
7    public Color getSystemColor(String type){
8        if(type.equals("Point"))
9            return Color.BLUE;
10       if(type.equals("Line"))
11           return Color.RED;
12
13       return Color.BLACK;
14   }
15}
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
```

Code 1.4: La classe DisplayManager

1.1.4.6 Aspect

Un *aspect* est l'abstraction de plus haut niveau du langage AspectJ. Cette structure permet d'encapsuler le code nécessaire à la modélisation d'une préoccupation transversale tout comme une *classe* en Java permet d'encapsuler le code nécessaire à la modélisation d'un objet.

Un *aspect* est structurellement très proche d'une *classe* Java même si sa sémantique en est différente. En plus de contenir les points de coupure et les greffons dont il a besoin, un *aspect* peut déclarer des méthodes et des variables, étendre d'autres *aspects* ou *classes* et implémenter des `interfaces`.

La différence majeure entre un *aspect* et une *classe* Java réside dans le fait qu'un aspect ne peut pas être instancié à l'aide de l'instruction `new`. En effet, l'instanciation des aspects dans un système se fait via une phase de « tissage » (*weaving*).

1.1.4.7 Tissage (*Weaving*)

Le tissage (*weaving*) est l'étape concrète qui permet d'injecter le comportement décrit par des aspects dans un programme orienté objet. Le tissage peut se faire de façon statique à de la compilation comme le fait AspectJ grâce à son compilateur *ajc*. Il est aussi possible de tisser les aspects de manière dynamique. C'est la méthode utilisée par certaines implémentations telles que HyperJ[2] ou encore AspectWerkz[3].

1.1.4.8 Une implémentation de l'exemple : FigureElement

L'implémentation orientée aspect de l'exemple développé dans les paragraphes précédents, présentée partiellement par les extraits de code 1.3 et 1.4 peut être obtenu via note site de référence (*cf.* annexe IV).

1.1.5 Relations entre la programmation orientée aspect et la programmation orientée objet

il faut ici mettre l'accent sur la relation particulière qui lie la programmation orientée aspect a la programmation orientée objet.

Dépendance. Le paradigme aspect ne sera considéré que dans sa forme initiale, c'est-à-dire comme un ajout à la programmation orientée objet. Dans le cas présent, une dépendance existentielle est observable entre la programmation orientée aspect et la programmation orientée objet.

Java et AspectJ. Nous nous concentrerons sur l'étude du langage AspectJ. Ce choix a été guidé par quatre considérations principales :

[2] `http://www.alphaworks.ibm.com/tech/hyperj`
[3] `http://aspectwerkz.codehaus.org/`

1. Cette implémentation du paradigme aspect a été développée par l'équipe du laboratoire qui est à l'origine de la définition de ce même paradigme ;
2. Cette implémentation a fait l'objet de nombreuses études et offre plusieurs outils sur lesquels s'appuyer. De plus, elle est supportée par des communautés de développeurs telles qu'Eclipse[4] ;
3. AspectJ s'appuie sur le langage Java. Nous considérons le langage Java comme une implémentation accessible du paradigme objet sans méta-niveaux et avec héritage simple.
4. Cela nous permettra aussi de réutiliser nos outils (PADL [Guéhéneuc, 2003], POM [Guéhéneuc *et al.*, 2004]) afin de proposer une implantation de nos résultats que nous détaillerons dans le chapitre 3.

Malgré ce choix, nos résultats peuvent être appliqués à d'autres implémentations de la POA pour d'autres langages orientés objet. En effet l'étude de la qualité de POA telle que nous la présentons ne dépend pas de son implémentation.

1.2 Qualité

1.2.1 Qualité et évaluation

De nombreux travaux ont été menés afin de mieux comprendre et de définir la qualité d'un logiciel. Par exemple, l'approche GQM (Goal-Question-Metrics) de Fenton [1996] permet de définir des métriques pour l'évaluation systématique de la qualité d'un logiciel. Le standard ISO 9126 [ISO9126, 1992], quant à lui, définit un modèle ayant pour but l'évaluation de la qualité. Ce modèle est divisé en six caractéristiques principales : fonctionnalité, fiabilité, utilisabilité, efficacité, maintenabilité, portabilité. Chacune d'entre elles se décompose à son tour en sous-caractéristiques [Khosravi, 2005, p. 170].

[4] `http://www.aspectj.org`

Le modèle proposé par le standard ISO 9126 peut être utilisé tel quel pour évaluer la qualité d'un logiciel ou encore être raffiné afin de répondre à des besoins précis. Ces travaux ont été précédés par les modèles proposés par McCall [1977] et par Boehm [1978].

L'approche de Fenton est la plus dynamique. Elle consiste, tout d'abord, à déterminer des objectifs généraux à atteindre pour un projet donné. Il faut ensuite leur associer des équations dont les réponses permettent de vérifier si l'objectif est atteint. Enfin, chaque question doit être analysée puis liée à une série de métriques. Ces métriques permettent de répondre aux questions.

Cette approche est dite *top-down* : elle cherche à définir les facteurs de qualité pour ensuite les mesurer. Nous nous intéresserons plutôt aux méthodes basées sur les métriques qui cherchent à comprendre la façon dont elles affectent les différentes caractéristiques de la qualité d'un logiciel. En effet, ce type d'approche correspond mieux à nos besoins : pour capturer l'impact de la POA, il faut comprendre comment ses mécanismes affectent les programmes orientés objet. Chidamber et Kemerer [1994] emploient ce type de méthode.

1.2.2 Métriques

1.2.2.1 Définition

Une métrique est formellement définie comme une association entre le monde empirique et le monde numérique. C'est un nombre ou un symbole associé à une entité qui permet de caractériser ses attributs.

Selon Fenton [1996], nous cherchons à formaliser et à mieux comprendre le monde qui nous entoure grâce à des séries de métriques. Ces métriques nous permettent de valider nos intuitions par rapport à ce monde. Les mesures que nous obtenons doivent donc représenter fidèlement les entités observées et préserver les relations qui les lient les unes aux autres.

Pour pouvoir prendre ces mesures, nous utilisons un modèle qui est une abstraction de la réalité observée. La précision de ce modèle et sa fidélité fixent les limites de nos mesures. Le calcul de métriques sur le modèle obtenu peut se faire de manières directe ou indirecte. Il est possible, dans un premier temps, d'associer directement les entités du modèle et leurs attributs à des mesures. Dans un second temps, nous pouvons faire intervenir des mesures indirectes qui caractériseront certains attributs en fonction d'autres attributs ou de mesures directes. Ce ne sont pas les deux seules catégories de métriques. En effet, il faut distinguer les mesures objectives, telles que la longueur ou la grandeur, et les mesures subjectives qui dépendent du jugement et du point de vue de l'observateur.

Fenton classifie les métriques en deux grands ensembles, fonctions du produit étudié : les métriques qui sont en relation avec la taille du produit et celles qui traitent de sa structure. Il divise les métriques reliées à la taille du produit en quatre catégories : longueur, fonctionnalité, complexité, réutilisabilité. Les métriques traitant de la structure interne du produit sont quant à elles décomposées en trois classes distinctes :

1. *Flot de contrôle* : la métrique se base sur la séquence d'exécution des instructions.
2. *Flot de données* : la métrique se base sur la gestion et la création des données.
3. *Structure de données* : la métrique se base sur l'organisation des données.

La première métrique utilisée avec succès dans le domaine du génie logiciel à l'époque de la programmation procédurale a été le nombre de lignes de code (LOC). Elle permet de caractériser un programme et est mise en relation avec sa complexité ou encore sa fiabilité. Ceci permettait aussi bien au développeur qu'au gestionnaire de prendre des décisions en fonction des variations de cette métrique. Mais l'introduction d'un nouveau paradigme, la programmation orientée objet, a forcé les chercheurs à repenser leurs systèmes de métriques.

1.2.2.2 Les métriques orientées objet

L'avènement des langages orientés objet a rendu la métrique LOC quasi-obsolète. En effet, ces langages introduisent un modèle qui se veut une abstraction fidèle de la réalité, constitué d'entités (classes et objets) qui ont des attributs (champs et méthodes) ainsi que des relations entre elles (agrégation, composition, etc.). Le code source lui même est réparti entre les différentes classes du modèle et leurs méthodes. C'est cet éclatement du code source qui rend l'utilisation de la métrique LOC plus compliqué et moins précis.

Les deux principales caractéristiques des langages orientés objet étudiées ont été le couplage et la cohésion.

Couplage. Le couplage peut être défini comme « le degré d'interaction entre deux modules » [Schach, 2002]. Il permet entre autres de caractériser la réutilisabilité et la réponse à la modification d'un module, notamment dans le cas d'opérations de maintenance.

Cohésion. Selon Fenton [1996], « la cohésion d'un module est le degré de similitude qu'ont les actions de ses différents composants ». La mesure de la cohésion d'un module est un bon indicateur pour évaluer la robustesse, la fiabilité ou encore la compréhensibilité d'un programme.

La suite de métrique C&K. En plus du couplage et du manque de cohésion, Chidamber et Kemerer [1994] introduisent quatre autres métriques visant à caractériser un programme orienté objet de la façon la plus complète possible. Ces métriques sont les plus célèbres et les plus étudiées dans le domaine de la qualité en génie logiciel. Le tableau 1.2 présente ces métriques, qui seront étudiées plus en détail au chapitre 2 car elles forment le noyau de notre architecture pour l'évaluation quantitative de l'impact de la POA.

Acronymes	Noms	Descriptions
WMC	*Weighted Methods per Class*	Mesure la complexité d'une classe en se basant sur la complexité de ses méthodes.
DIT	*Depth of Inheritance Tree*	Valeur du plus long chemin depuis le noeud (la classe étudiée) jusqu'à la racine de l'arbre d'héritage.
NOC	*Number Of Children*	Nombre de sous-classes héritant directement de la classe étudiée.
CBO	*Coupling between object classes*	Nombre de classes couplées à la classe étudiée.
RFC	*Response For a Class*	Le nombre de méthodes locales de la classe étudiée plus le nombre d'appels à des méthodes faits par les méthodes locales.
LCOM	*Lack of Cohesion in Methods*	Mesure le manque de cohésion d'une classe en se basant sur les les intersections nulles de l'ensemble des méthodes de la classe étudiée.

TAB. 1.2: Les métriques de Chidamber et Kemerer

Tous ces travaux auxquels s'ajoute la série de métrique de Lorenz et Kidd [1994], ont permis au génie logiciel de se munir des outils nécessaires à l'évaluation de la qualité des logiciels. Aujourd'hui, des outils très performants, comme la suite Rational[5] d'IBM, permettent de tester et d'évaluer les logiciels à tous les stades de leur cycle de vie. Mais l'introduction d'un nouveau paradigme comme la programmation orientée aspect boulverse ces méthodes d'évaluation.

En effet, un nouveau paradigme doit être étudié en termes de qualité afin d'être mieux compris et appréhendé. La POA n'échappe pas à cet axiome. L'étude des rapports entre la qualité (telle que comprise par le génie logiciel) et la programmation aspect en est encore à ses débuts [Guyomarc'h et Guéhéneuc, 2005]. De par la

[5] `http://www.ibm.com/software/rational`

relation particulière qui lie la programmation orientée aspect à la programmation orientée objet, deux approches différentes s'offrent à nous : d'une part, la création de métriques pour la programmation orientée aspect et, d'autre part, l'étude des effets de la POA sur les systèmes orientés objet.

1.2.3 Programmation orientée aspect et qualité

1.2.3.1 Définition de métriques pour la POA

Mesurer les flots de données. Zhao [2002a] est le premier a proposer une méthode servant à quantifier les flots de données dans un programme orienté aspect. Il se base sur modèle de dépendance à trois niveaux :

- Le *Module-Level Dependence Graph*, qui permet de représenter les méthodes, les greffons ainsi que les introductions d'un aspect ;
- Le *Aspect-Level Dependence Graph*, qui permet de représenter un aspect et les différentes interactions de ses éléments ;
- Le *System-Level Dependence Graph*, qui permet de représenter un programme orienté aspect complet.

Zhao extrait de ce modèle une première série de 15 métriques lui servant à caractériser les flots de données dans un système orienté aspect. Il affirme que cette série de métriques permet d'évaluer la complexité d'un programme orienté aspect.

Mesurer le couplage et la cohésion. Dans un effort pour poursuivre ses précédents travaux, Zhao [2004] propose un cadre d'application permettant de mesurer le couplage au niveau des aspects. Il exprime cette métrique comme le degré d'indépendance entre les classes et les aspects cohabitant dans un même système.

Accompagné de Xu, Zhao [2004] donne une définition de la cohésion pour les aspects. Il reprend la définition donnée par Chidamber et Kemerer [1994] en y incluant le mécanisme de greffon propre à la POA. Cette approche a été critiquée et raffinée par Gélinas *et al.* [2005].

Les travaux de chercheurs tels que Zhao sont, bien entendu, nécessaires à la compréhension d'un paradigme aussi jeune que l'orienté aspect. Ils permettent de baliser le champ d'investigation et jettent les bases requises à une bonne évaluation des programmes orientés aspect. Mais la nature même de la POA nous pousse à approcher l'étude de ce nouveau paradigme sous un autre angle. En effet, sa relation avec le paradigme objet (*cf.* section 1.1.5) nous porte à croire que l'étude de l'impact de la POA sur la POO est tout aussi importante que l'étude du paradigme en lui-même.

1.2.3.2 Effet de la POA sur les systèmes orientés objet

Étudier le comportement dynamique. Le groupe de recherche SABLE de l'Université McGill [2004] propose un cadre pour étudier le comportement dynamique d'AspectJ et son impact sur Java. Pour y parvenir, il propose une série de métriques raffinée et deux outils : une version modifiée du compilateur d'AspectJ et une version modifiée de leur outil *J. Grâce au premier, ils annotent le *bytecode* Java après la phase de tissage, tandis que le second leur permet d'extraire des métriques à partir du *bytecode* précédemment annoté. Leur but est d'offrir une méthode permettant d'étudier l'impact des programmes AspectJ sur le *bytecode* Java, après la phase de tissage. Ainsi ils peuvent analyser l'impact de la POA sur les systèmes orientés objet à l'exécution.

Étudier l'impact de la POA sur les métriques de classe. Zakaria et Hosny [2003], pour leur part, étudient l'impact de la POA sur les métriques de Chidamber et Kemerer [1994]. Ils se placent dans le contexte de la refactorisation de cetaines composantes d'un programme purement objet par le biais de la POA et, sur une base théorique, nous font part de leurs intuitions quant à la modification des métriques étudiées. Ils concluent que tous les mécanismes introduits par la POA pourraient avoir un impact sur les métriques C&K.

Sant'Anna *et al.* [2003] proposent un cadre d'application permettant d'évaluer la réutilisabilité et la maintenabilité des systèmes orientés aspect. Leur travail traite principalement de la qualité des programmes orientés aspect. Mais afin de réaliser leur projet, ils ont étudié l'impact de la POA sur certaines métriques de classe. En raffinant certaines métriques telles que celles de Chidamber et Kemerer [1994] ou encore celles de Lopes [1997], ils proposent une série de métriques adaptées aux systèmes orientés aspect et applicables à leur modèle de qualité. Ils ne considèrent seulement que deux des abstractions du paradigme aspect, à savoir *l'aspect* et *le greffon*, mais leur travail reste le plus substantiel à ce jour en ce qui concerne l'étude de l'impact de la POA sur la POO.

1.3 Notre approche

Comme nous l'avons mis en valeur dans la section 1.1.5, la POA, d'après ses concepteurs, ne se suffit pas à elle même. Elle dépend de la POO pour exister, car elle a pour fonction de modifier, de façon transparente, les programmes orientés objet. Dans les faits, un aspect peut injecter du code, introduire de nouvelles variables ou de nouvelles méthodes, et modifier la hiérarchie des classes et des interfaces du programme. Cette particularité a été largement critiquée par Highley *et al.* [1999], qui mettent de l'avant les dangers des mécanismes d'introduction propre à la POA ainsi que son accès privilégié aux classes et aux objets d'un programme orienté objet.

Dans le cas d'AspectJ, cette modification prend concrètement effet une fois la phase de tissage effectuée. C'est le « tisseur » qui décide de la politique à adopter pour répercuter, par exemple, la modification de la hiérarchie d'une classe C par l'aspect A dans le bytecode. Il existe déjà sur le marché plusieurs tisseurs [Allan *et al.*, 2005; Bonér *et al.*, 2005; Lopes et Ngo, 2004] qui implémentent leur stratégie propre. Mais pour un ingénieur en génie logiciel ou un concepteur de logiciel, la

modification de la hiérarchie d'une classe à une sémantique particulière et un impact bien précis sur la qualité du système. C'est pourquoi nous estimons nécessaire de nous affranchir de la phase de tissage en étudiant l'impact de la POA sur les systèmes orientés objet en demeurant au niveau de la conception.

Cet impact peut être capturé puis interprété par l'utilisation des métriques de classe développées par Chidamber et Kemerer [1994] ou Lorenz et Kidd [1994]. Pour ce faire, nous avons choisi de poursuivre les travaux amorcés par Sant'Anna *et al.* [2003] et Zakaria et Hosny [2003], en étudiant la relation de la POA avec la POO à travers les métriques de classe pour offrir une évaluation quantitative et qualitative de l'impact de la POA sur la POO.

CHAPITRE 2

ÉTUDE DES MÉTRIQUES

Dans la première partie de ce mémoire nous avons présenté les différents axes de recherche en matière de qualité et de POA et avons choisi d'étudier l'impact de la POA sur les métriques de classe. Nous présentons donc, dans ce chapitre, nos résultats quant à l'étude de cet impact. Ces travaux sont les fondements théoriques nécessaires à la réalisation de notre projet.

2.1 Introduction

De par notre étude approfondie de la POA (*cf.* section 1.1), nous avons été en mesure d'isoler les mécanismes propres au paradigme aspect ayant une incidence sur le calcul des métriques de classe. Nous pouvons désormais analyser en détail l'impact de la POA sur les métrique de classe choisies et dégager une nouvelle méthode de calcul pour chacune d'entre elles.

Ce travail théorique nous permet d'implémenter une solution logicielle concrète et de vérifier sa pertinence en termes d'évaluation de la qualité des systèmes orientés aspect. Ces deux étapes de notre projet sont détaillées respectivement dans les chapitres 3 et 4 de ce mémoire.

2.1.1 Protocole d'étude

Afin d'étudier chaque métrique de la façon la plus complète possible et de capturer l'impact de la programmation orientée aspect sur ces métriques, nous suivrons le protocole suivant :

1. **Définition.** D'abord, nous donnons une définition de la métrique orientée objet étudiée. Le cas échéant nous choisissons la définition la plus consensuelle

parmi différentes sources.

2. **Méthode de calcul.** À partir de la définition de la métrique étudiée, nous établissons, de façon formelle, une méthode de calcul en utilisant la théorie des ensembles [Dupin et Valein, 1993, p. 53].
3. **Incidence de la POA.** Nous mettons alors en relief les mécanismes de la POA qui peuvent avoir un impact sur la métrique étudiée.
4. **Nouvelle méthode de calcul.** Nous modifions la méthode de calcul précédemment établie afin d'y refléter l'incidence de la POA.
5. **Exemple.** Nous illustrons l'impact de la POA sur la métrique étudiée grâce à un exemple.

2.1.2 Notation et diagrammes

Notation formelle. Pour les besoins de nos démonstrations, nous utilisons, comme l'ont fait avant nous Chidamber et Kemerer [1994], la théorie des ensembles [Dupin et Valein, 1993, p. 53]. Une description détaillée des ensembles et propriétés utilisés est fournie à l'annexe II (p. 87).

Exemples et diagrammes UML. Afin de présenter les différents exemples étudiés sous forme de diagrammes UML, nous utiliserons le profil UML *AspectUML* développé par Vachon et Mostefaoui [2005, p. 38]. L'annexe III (p. 92) présente une descripiton détaillée du profil *AspectUML*.

2.2 Étude des métriques

Nous présentons l'étude des six métriques de classes développées par Chidamber et Kemerer [1994]. Ces métriques sont très utilisées aussi bien dans le domaine de la recherche que dans celui de l'industrie. De plus, elles ont fait l'objet, en tant que métriques de classe, de nombreuses études scientifiques.

Nous y ajoutons quatre autres métriques. Ces métriques ont été choisies pour plusieurs raisons : leur niveau de complexité, leur couverture des différents mécanismes du paradigme aspect, et enfin parce qu'elles sont implémentées par l'outil POM [Guéhéneuc *et al.*, 2004] que nous utiliserons dans le chapitre 3. Ces métriques sont le fruit des travaux de Lorenz et Kidd [1994].

Notre ensemble d'étude est donc constitué de dix métriques : WMC, DIT, NOC, CBO, RFC, LCOM, NAD, NMD, NMO et SIX. Leur analyse est détaillée dans la section suivante.

2.2.1 Weighted Methods per Class

Définition. *Weighted Methods per Class* (WMC) permet de donner une indication sur la complexité d'une classe à partir de la complexité de ses méthodes [Chidamber et Kemerer, 1994, p. 482].

Méthode de calcul. Soit c une classe, $m_1...m_n$ les méthodes[1]déclarées par cette classe et $mc_1...mc_n$ les valeurs numériques correspondante à la complexité des méthodes alors la valeur de WMC pour la classe c peut être calculée grâce à l'équation 2.1.

$$WMC(c) = \sum_{i=1}^{n} mc_i \tag{2.1}$$

Comme précisé par Chidamber et Kemerer [1994], si la complexité des méthodes d'une classe est considérée comme unitaire, alors le calcul de WMC pour une classe c correspond à la cardinalité de l'ensemble MD_c, avec MD_c l'ensemble des méthodes déclarées par c (cf. équation 2.2).

$$WMC(c) = |MD_c| \tag{2.2}$$

[1]Lorsque le contraire n'est pas précisé, le concept de *méthode* utilisé dans les démonstrations englobe aussi bien les méthodes de la classe que ses constructeurs.

Incidence de la POA. Dans le cas où la complexité des méthodes est considérée comme unitaire, la POA peut modifier la valeur de WMC en introduisant dans une classe donnée de nouvelles méthodes. Si au contraire, la complexité n'est pas considérée comme unitaire, alors la POA peut, en plus d'introduire de nouvelles méthodes, modifier la complexité des méthodes d'une classe via le mécanisme de greffon.

Nouvelle méthode de calcul. Soit une classe c, $itm_1...itm_n$ les méthodes introduites par différents aspects au niveau de la classe c, $itmc_1...itmc_n$ la complexité associée à ces méthodes introduites, $m_1...m_n$ les méthodes déclarées par cette classe, et $gmc_1...gmc_n$ les valeurs numériques correspondantes à la complexité des méthodes en considérant l'apport possible des greffons, alors la valeur de WMC pour la classe c peut être calculée grâce à l'équation 2.3.

$$WMC(c) = \sum_{i=1}^{n} gmcc_i + \sum_{i=1}^{n} itmc_i \tag{2.3}$$

Si la complexité est considérée comme unitaire, la nouvelle méthode de calcul de WMC se réduit à 2.4, avec MD_c l'ensemble des méthodes déclarées par c, et $ITMD_c$ l'ensemble des méthodes introduites sur c.

$$WMC(c) = |MD_c| + |ITMD_c| \tag{2.4}$$

Exemple. Le diagramme *AspectUML* sur la figure 2.1 présente la modification de la classe `Figure` par l'aspect `DisplayAspect`. Afin de permettre à la classe `DisplayManager` d'afficher des objets de type `Figure`, l'aspect `DisplayAspect` déclare que la classe `Figure` implémente l'interface `DisplayableFigure`. Deux méthodes, `getColor()` et `setColor()`, ainsi qu'un champ `color` sont aussi introduits.

Ces modifications apportées par l'aspect `DisplayAspect` augmentent la valeur de la métrique WMC sur la classe `Figure`. En effet, avant introduction

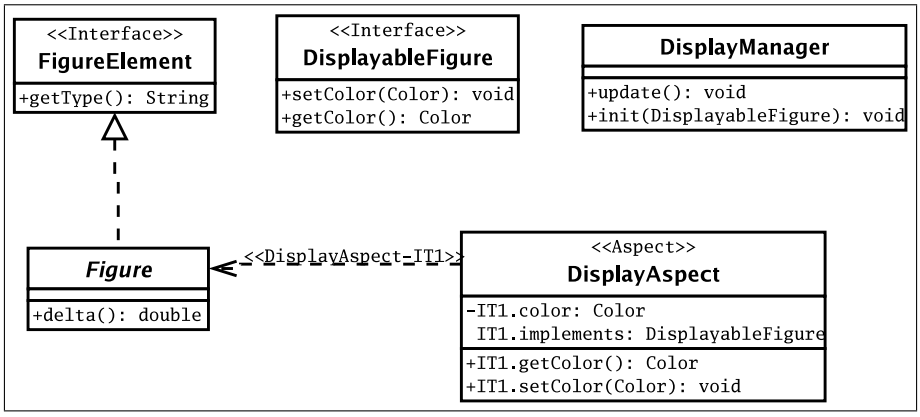

FIG. 2.1: Ajout de l'interface `DisplayableFigure` à la classe `Figure`

$WMC(Figure) = 1$, après introduction des deux nouvelles méthodes,
$WMC_{POA}(Figure) = 3$.

2.2.2 Depth of Inheritance Tree

Définition. *Depth of Inheritance Tree* (DIT) correspond à la valeur du plus long chemin depuis le noeud (la classe étudiée) jusqu'à la racine de l'arbre d'héritage [Chidamber et Kemerer, 1994, p. 483].

Méthode de calcul. La valeur de DIT pour une classe c se calcule de façon récursive. Soit $PARENTS_c$ l'ensemble des classes qui sont parentes immédiates de la classe c et $root$ la classe racine de la hiérarchie alors la valeur de DIT peut se calculer à l'aide de l'équation 2.5.

$$\begin{aligned} DIT(root) &= 0 \\ DIT(c) &= 1 + max(\{DIT(super) | super \in PARENTS_c\}) sinon \end{aligned} \tag{2.5}$$

Incidence de la POA. La POA peut avoir une incidence sur le calcul de la métrique DIT dans la mesure où il est possible de modifier la hiérarchie d'une

classe en utilisant les mécanisme d'introduction.

Nouvelle méthode de calcul. La méthode de calcul ne change pas. En effet, la modification se fait au niveau de l'ensemble des parents immédiats de la classe. Soit c une classe donnée, $ITDP$ l'ensemble des classes dont la hiérarchie a été modifiée, alors la valeur de DIT pourra être différente si et seulement si c ou l'un de ses ancêtres est inclus dans $ITDP$.

Exemple. La figure 2.3 présente une hiérarchie de classes : les classes `Line` et `Curve` héritent de la classe `Figure` qui est la racine de la hiérarchie. Nous avons donc $DIT(Figure) = 0$, $DIT(Line) = DIT(Curve) = 1$. Cette hiérarchie peut être facilement modifiée par la POA. L'exemple 2.6, ainsi que la représentation `AspectUML` correspondante (`cf.` figure 2.2), présente une .utilisation du mécanisme d'introduction permettant de changer l'héritage de la classe `Curve` de `Figure` à `Line` par l'aspect `DisplayAspect`. La hiérarchie correspondante est modélisée dans la figure 2.4. En conséquence la valeur de la métrique DIT sur la classe `Curve` est modifiée, $DIT(Curve) = 2$.

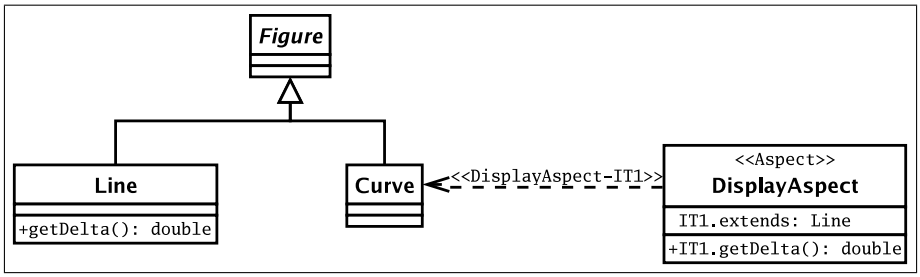

FIG. 2.2: Modification de la hiérarchie de la classe `Curve`

declare parents : `Curve` **extends** `Line` ; (2.6)

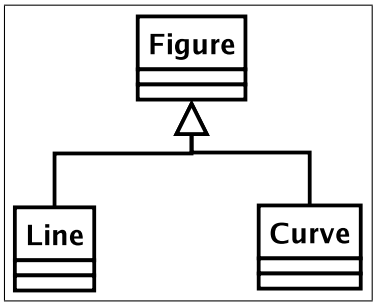

FIG. 2.3: DIT : hiérarchie originale

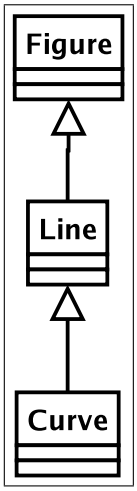

FIG. 2.4: DIT : hiérarchie modifiée par la POA

2.2.3 Number Of Children

Définition. *Number Of Children* (NOC) permet de calculer le nombre de sous-classes immédiates qui héritent de la classe étudiée [Chidamber et Kemerer, 1994, p. 485].

Méthode de calcul. Pour une classe c donnée, avec SCI_c l'ensemble des sous-classes immédiates de c, la valeur de $NOC(c)$ est donnée par la cardinalité de l'ensemble SCI_c tel qu'illustré dans l'équation 2.7.

$$NOC(c) = |SCI_c| \tag{2.7}$$

Incidence de la POA. La POA peut modifier la métrique NOC grâce au mécanisme d'introduction. Une classe X, orpheline ou bénéficiant de l'héritage, peut être déclarée comme sous-classe de c. Le NOC de la classe c est alors augmenté de 1.

Nouvelle méthode de calcul. Soit c une classe donnée, et $ITDP_c$ l'ensemble des classes du programme qui se sont vues attribuer la classe c comme parent via le mécanisme d'introduction. L'équation 2.8 définit la nouvelle méthode de calcul.

$$NOC(c) = |SCI_c \cup ITDP_c| \tag{2.8}$$

Exemple. Dans l'exemple représenté sur la figure 2.2, la hiérarchie de la classe `Curve` est modifiée. Cette classe est déclarée comme sous-classe non plus de `Figure` mais de `Line`. En plus d'avoir un impact sur la valeur de DIT de la classe `Curve`, cette modification entraîne des répercutions sur la valeur de NOC des classes `Figure` et `Line` en périphérie. Comme on peut le voir sur les figures 2.3 et 2.4, le NOC de la classe `Figure` sera diminué de 1 et celui de la classe `Line` sera augmenté de 1.

2.2.4 Coupling Between Object

Définition. *Coupling Between Object* (CBO) permet de mesurer le couplage d'une classe c par rapport aux autres classes du programme [Chidamber et Kemerer, 1994, p. 486].

Méthode de calcul. Soit c une classe, c est considérée comme couplée à une classe x à partir du moment où elle utilise au moins une des méthodes de la classe x. Soit A l'ensemble des classes du programme, O_c l'ensemble des classes $o_1, o_2 ... o_n$ tel que $O_c = A \setminus \{c\}$, MD_{o_i} l'ensemble des méthodes déclarées par la classe o_i, et MA_c l'ensemble des méthodes appelées par la classe c. Il est alors possible de calculer CBO grâce à l'équation 2.9.

$$CBO(c) = |\{o_i \in O | MD_{o_i} \cap MA_c \neq \oslash\}| \tag{2.9}$$

Incidence de la POA. La POA peut intervenir de deux manières sur le comportement de la métrique CBO. Il lui est possible d'introduire de nouvelles méthodes ou d'utiliser des points de coupure afin d'ajouter des greffons aux méthodes déjà déclarées. Dans les deux cas, la valeur de CBO peut croître en fonction des classes utilisées par les mécanismes introduits. Dans un cas précis, la valeur de CBO pour une classe donnée peut être réduite. En effet, l'utilisation du greffon de type around peut remplacer complètement le corps d'une méthode. Si AspectJ propose un prédicat `proceed()`, qui permet au greffon de demander l'exécution de la méthode masquée, son utilisation n'est pas obligatoire. Il est donc possible que le « masquage » de cette méthode réduise la valeur de CBO de la classe modifiée.

Nouvelle méthode de calcul. Soit $ITMA_c$ l'ensemble des appels de méthode faits par l'entremise de méthodes introduites sur la classe c, et GMA_c l'ensemble des appels de méthode faits par l'entremise de greffons associés grâce à des points de coupure à la classe c, alors l'équation précédente 2.9 se voit modifiée en une nouvelle équation 2.10.

$$\begin{aligned} CBO(c) = |\{o_i \in O | MD_{o_i} \cap MA_c \neq \oslash\}| + \\ |\{o_i \in O | MD_{o_i} \cap ITMA_c \neq \oslash\}| + \\ |\{o_i \in O | MD_{o_i} \cap GMA_c \neq \oslash\}| \end{aligned} \quad (2.10)$$

Exemple. Les extraits de code 2.1 et 2.2 présentent respectivement la classe `Figure` et un aspect `LoggingAspect` qui modélise la journalisation. Pour les besoins de l'exemple, nous considérons qu'aucun autre aspect n'intervient sur la classe `Figure`. On remarque que la classe `Figure`, avant introduction des aspects, a un couplage nul. L'aspect `LoggingAspect` ajoute la journalisation des objets de type

```
1 public abstract class Figure
2       implements FigureElement {
3
4   public void translate(int a, int b){
5     //translation
6   }
7
8   public void rotate(double rad){
9     //rotate
10  }
11
12  public void scale(double scale){
13    //scale
14  }
15
16  public String getType(){
17    return "unknown type";
18  }
19
20
21}
22
23
24
25
26
27
28
29
30
31
32
33
34
35
```

Code 2.1: La classe `Figure`

```
1 public aspect LoggingAspect {
2
3   private Logger Figure.logger = new Logger();
4
5   pointcut logTranslate(Figure f) :
6     execution(public void Figure.translate(..))
7     && this(f);
8
9   pointcut logRotate(Figure f) :
10    execution(public void Figure.rotate(..))
11    && this(f);
12
13  pointcut logScale(Figure f) :
14    execution(public void Figure.scale(..))
15    && this(f);
16
17  public void Figure.logOn(final PrintStream out){
18    this.logger.setPrintStream(out);
19  }
20
21  after(Figure f) : logTranslate(f) {
22    f.logger.log("Translation on a "
23                 + f.getType());
24  }
25
26  after(Figure f) : logRotate(f) {
27    f.logger.log("Rotation on a "
28                 + f.getType());
29  }
30
31  after(Figure f) : logScale(f) {
32    f.logger.log("Scale on a "
33                 + f.getType());
34  }
35}
```

Code 2.2: L'aspect `LoggingAspect`

`Figure`. Il introduit donc une nouvelle variable de type `Logger`, un nouvelle méthode permettant de fixer le vecteur de sortie et trois greffons associés aux méthodes `translate()`, `rotate()` et `scale()` de la classe `Figure`. L'ajout de la variable `logger` et son utilisation dans la méthode introduite ainsi que dans les trois greffons augmentent le couplage de la classe `Figure`, $CBO_{POA}(Figure) = 1$.

2.2.5 Response For a Class

Définition. *Response For a Class* (RFC) correspond à la taille du *Response Set* (RS) de la classe étudiée. Le RS d'une classe est défini comme le nombre de méthodes locales de la classe plus le nombre total d'appels à des méthodes faits par les méthodes locales [Chidamber et Kemerer, 1994, p. 487].

Méthode de calcul. Soit c une classe, MD_c l'ensemble des méthodes de c déclarées localement, et $MCALL_i$ l'ensemble des appels de méthode fait par une

méthode m_i tel que $m_i \in MD_c$, alors nous pouvons calculer la valeur de RFC de la classe c grâce à l'équation 2.11

$$RFC(c) = |MD_c| + \sum_{i=1}^{|MD_c|} |MCALL_i| \tag{2.11}$$

Incidence de la POA. La métrique RFC peut être modifiée par deux mécanismes de la POA. Il est possible d'ajouter directement de nouvelles méthodes, ou encore d'associer des greffons grâce à des points de coupure. Il n'est pas nécessaire de savoir exactement où le greffon est introduit dans la classe pour calculer la modification. En effet, il suffit à ce dernier d'être associé à la classe pour ajouter la taille de son RS au RFC de la classe.

Nouvelle méthode de calcul. Soit $ITMD_c$ l'ensemble des méthodes introduites sur la classe c, $ADVICE_c$ l'ensemble des greffons associés à la classe c, $ITMCALL_j$ l'ensemble des appels de méthode faits par une méthode m_j tel que $m_j \in ITMD_c$, et $AMCALL_k$ l'ensemble des appels de méthode faits par un greffon a_k tel que $a_k \in ADVICE_c$, nous pouvons alors modifier l'équation 2.11 afin d'offrir une nouvelle définition de RFC (*cf.* équation 2.12).

$$\begin{aligned} RFC(c) = |MD_c| + \sum_{i=1}^{|MD_c|} |MCALL_i| + \\ |ITMD_c| + \sum_{j=1}^{|ITMD_c|} |ITMCALL_j| + \\ + \sum_{k=1}^{|ADVICE_c|} |AMCALL_k| \end{aligned} \tag{2.12}$$

Exemple. La classe `Figure` (*cf.* l'extrait de code 2.1), a un RFC de 4. En effet, toutes ses méthodes sont vides ou presque. En utilisant l'équation 2.11, nous

pouvons calculer la valeur de cette métrique, $RFC(Figure) = 4$. Mais l'aspect `LoggingAspect` (*cf.* extrait de code 2.2) modifie beaucoup cette valeur. En effet, l'aspect introduit une nouvelle méthode qui fait appel à la méthode `setPrintStream` de la classe `Logger` ainsi que trois greffons qui font tous appel à la méthode `log` de la classe `Logger`. En utilisant la nouvelle méthode de calcul 2.12 de RFC, on vérifie que la valeur de RFC pour la classe `Figure` a bien augmenté, $RFC_{POA}(Figure) = 9$.

2.2.6 Lack of Cohesion in Methods

Définition. *Lack of Cohesion in Methods* (LCOM) permet de mesurer le manque de cohésion d'une classe [Chidamber et Kemerer, 1994, p. 488].

Méthode de calcul. Afin de mesurer le manque de cohésion d'une classe, on se base sur l'utilisation des variables d'instance de la classe par ses méthodes. Considérons une classe c, et MD_c l'ensemble des méthodes $m_1, m_2...m_n$ déclarées par c. Soit I_j l'ensemble des variables d'instances utilisées par la méthode m_j. Nous avons donc n ensembles de ce type, $I_1, I_2...I_n$.

Soit P_c, l'ensemble des paires de méthodes de la classe c qui ne partagent pas l'utilisation d'une même variable d'instance et Q_c l'ensemble des paires de méthodes de la classe c qui utilisent au moins une même variable d'instance, $P_c = \{(I_i, I_j)|I_i \cap I_j = \oslash\}$ et $Q_c = \{(I_i, I_j)|I_i \cap I_j \neq \oslash\}$, alors la valeur de $LCOM(c)$ peut être calculée via l'équation 2.13.

$$\begin{aligned} LCOM(c) &= |P_c| - |Q_c| \ si \ |P_c| > |Q_c| \\ &= 0 \ sinon \end{aligned} \tag{2.13}$$

Incidence de la POA. L'impact des mécanismes de la programmation orientée aspect sur la métrique LCOM est important. Tout d'abord, l'introduction de nouveaux champs et de nouvelles méthodes aura un impact direct sur la cohésion de la classe. Les méthodes introduites peuvent utiliser les champs originaux de la classe et ainsi en augmenter la cohésion ou, tout au contraire, réduire cette cohésion en n'utilisant pas ces champs. L'introduction de nouveaux champs (qui peuvent être utilisés seulement par les méthodes introduites ou les greffons) peut aussi augmenter la cohésion, mais non la réduire telle qu'elle a été définie par Chidamber et Kemerer. Enfin, les greffons jouent un rôle important dans cet impact. Les greffons de type *before* et *after* peuvent augmenter la cohésion en associant l'utilisation de champs, introduits ou non, à des méthodes de la classe. Les greffons de type *around*, quant à eux, ont un effet encore plus lourd. Leur habilité à « masquer » ou à étendre une méthode leur permet de réduire ou d'augmenter à volonté la cohésion d'une classe.

Nouvelle méthode de calcul. La nouvelle méthode de calcul n'est pas différente de l'équation 2.13. En effet, l'impact de la POA sur cette métrique se fait en amont des calculs. Pour une classe c, il faut aussi bien tenir compte de l'ensemble MD_c de ses méthodes que de l'ensemble $ITMD_c$ des méthodes introduites. Ensuite, le calcul de l'ensemble I_j des variables d'instances (introduites ou non) utilisées par une méthode m_j (introduite ou non) est modifié en fonction des greffons associés à la méthode m_j. Une fois les ensemble de variables d'instances utilisées par les différentes méthodes de la classe c déterminés, il est possible de calculer les ensembles de paires de méthodes P_c et Q_c comme vu précédemment. On applique alors l'équation 2.13 pour obtenir la valeur de LCOM après introduction des aspects.

Exemple. L'absence de variable dans la classe `Figure` (*cf.* extrait de code 2.1) nous permet d'affirmer que cette classe a une cohésion nulle. Mais l'aspect `LoggingAspect` (*cf.* extrait de code 2.2) introduit une variable de type `Logger` sur la classe `Figure`, ainsi qu'une nouvelle méthode et trois greffons. Tous les quatre utilisent la variable introduite. Les trois greffons sont respectivement associés aux méthodes `translate()`, `rotate()`, `scale()`. Au final, nous obtenons quatre méthodes qui utilisent la même variable, et une méthode, `getType()`, qui n'en utilise aucune. Cela fait donc six paires de méthodes qui partagent une variable et quatre paires de méthodes qui n'en partagent aucune. Nous pouvons alors appliquer l'équation 2.13 pour calculer la nouvelle valeur de LCOM, $LCOM_{POA}(Figure) = 2$, ce qui donne une augmentation de deux unités.

2.2.7 Number of Attributes Declared

Définition. *Number of Attributes Declared* (NAD) permet de calculer, pour une classe donnée, le nombre d'attributs déclarés [Lorenz et Kidd, 1994, p. 54].

Méthode de calcul. Soit c une classe et AD_c l'ensemble des attributs déclarés par la classe c. La valeur de la métrique NAD pour la classe c sera donnée par la cardinalité de l'ensemble AD_c. Ce calcul est défini par l'équation 2.14.

$$NAD(c) = |AD_c| \tag{2.14}$$

Incidence de la POA. La POA peut modifier cette métrique, dans la mesure où un aspect est capable, via le mécanisme d'introduction, de déclarer de nouveaux attributs dans une classe donnée.

Nouvelle méthode de calcul. Soit c une classe, AD_c l'ensemble de ses attributs, et $ITAD_c$ l'ensemble des attributs introduits par différents aspects dans la classe c. La valeur de la métrique NAD pour la classe c sera donnée par la cardinalité

de l'ensemble AD_c à laquelle on ajoute la cardinalité de l'ensemble $ITAD_c$ (cf. équation 2.15).

$$NAD(c) = |AD_c| + |ITAD_c| \tag{2.15}$$

Exemple. Dans le diagramme 2.1 on remarque que l'aspect `DisplayAspect` introduit un champ de type `Color` dans la classe `Figure`. Ceci modifie la valeur de NAD sur cette classe. Elle était nulle avant introduction des aspects et est maintenant augmentée d'une unité : $NAD_{POA}(Figure) = 1$.

2.2.8 Number of Methods Declared

Définition. *Number of Methods Declared* (NMD) permet de calculer, pour une classe donnée, le nombre de méthodes déclarées [Lorenz et Kidd, 1994, p. 51].

Méthode de calcul. Soit c une classe, et MD_c l'ensemble des méthodes déclarées par la classe c. La valeur de la métrique NMD pour la classe c sera donnée par la cardinalité de l'ensemble MD_c. Ce calcul est défini par l'équation 2.16.

$$NMD(c) = |MD_c| \tag{2.16}$$

Incidence de la POA. La POA peut modifier cette métrique, dans la mesure où un aspect est capable, via le mécanisme d'introduction, de déclarer de nouvelles méthodes dans une classe donnée.

Nouvelle méthode de calcul. Soit c une classe, MD_c l'ensemble de ses méthodes et $ITMD_c$ l'ensemble des méthodes introduites par différents aspects au niveau de la classe c. La valeur de la métrique NMD pour la classe c sera donnée par la cardinalité de l'ensemble MD_c à laquelle on ajoute la cardinalité de l'ensemble $ITMD_c$ (cf. équation 2.17).

$$NMD(c) = |MD_c| + |ITMD_c| \tag{2.17}$$

Exemple. Le diagramme 2.1 montre comment l'aspect `DisplayAspect` introduit un champ nommé `color` ainsi que des accesseurs. Ceux-ci prennent la forme de deux méthodes : `getColor()` et `setColor()`. Cette introduction modifie la valeur de NMD sur la classe `Figure`. Elle passe de un à trois, $NMD_{POA}(Figure) = 3$.

2.2.9 Number of Methods Overridden

Définition. *Number of Methods Overridden* (NMO) correspond au nombre de méthodes d'une classe donnée qui redéfinissent[1] une méthode héritée [Lorenz et Kidd, 1994, p. 66].

Méthode de calcul. Soit une classe c, MD_c l'ensemble des méthodes déclarées par c, et MH_c l'ensemble des méthodes héritées par c, alors la valeur de NMO est la cardinalité de l'intersection de ces deux ensembles. La comparaison des méthodes se fait seulement au niveau de la signature (*cf.* équation 2.18).

$$NMO(c) = |MD_c \cap MH_c| \tag{2.18}$$

Incidence de la POA. Il est tout d'abord possible d'introduire sur une classe c de nouvelles méthodes qui redéfinissent des méthodes héritées par c. Il est aussi possible de faire des changements dans l'arbre d'héritage de c, en introduisant de nouveaux ancêtres ou en introduisant de nouvelles méthodes dans des ancêtres de c. Il est alors possible que certaines méthodes de c deviennent des redéfinitions de

[1] Claude Delannoy [Delannoy, 2004, p. 195] décrit la *redéfinition* comme suit :

> « Une classe dérivée peut [...] fournir une nouvelle définition d'une méthode d'une classe ascendante. [...] Alors que la surdéfinition permet de cumuler plusieurs méthodes de même nom, la *redéfinition* substitue une méthode à une autre. »

méthodes introduites au niveau des ancêtres de c ou de méthodes déclarées par des ancêtres introduits dans la hiérarchie d'héritage de c.

Nouvelle méthode de calcul. Soit une classe c, MD_c l'ensemble des méthodes déclarées par c, MH_c l'ensemble des méthodes héritées par c, $ITMD_c$ l'ensemble des méthodes introduites sur c, et $ITMH_c$ l'ensemble des méthodes introduites dans l'arbre d'héritage de c, soit par introduction de méthodes sur les ancêtres de c, soit par introduction de nouveaux ancêtres dans l'arbre d'héritage de c. Il est alors possible de calculer NMO grâce à l'équation 2.19

$$NMO(c) = |(MD_c \cup ITMD_c) \cap (MH_c \cup ITMH_c)| \tag{2.19}$$

Exemple. Lors de la modification de la hiérarchie de `Curve`, le développeur s'aperçoit qu'il crée une erreur au niveau de la méthode `getDelta()` (calcul la pente d'une droite) de la classe `Line`. En héritant de la classe `Line` par introduction, `Curve` hérite aussi de la méthode `getDelta()`. Or le calcul de cette méthode n'est plus le même pour une ligne courbe. Le développeur doit donc redéfinir, en utilisant l'introduction, la méthode `getDelta()` pour la classe `Curve`. Ce faisant il augmente la valeur de NMO de cette classe d'une unité, $NMO_{POA}(Curve) = 1$.

2.2.10 Specialisation IndeX

Définition. *Specialisation IndeX* (SIX) mesure le degré de spécialisation d'une classe par rapport à ses ancêtres (classes dont elle hérite). Une spécialisation optimale implique une réutilisation complète du comportement hérité, ainsi qu'une extension à ce comportement (ajout ou redéfinition de méthodes, par exemple) [Lorenz et Kidd, 1994, p. 71].

Méthode de calcul. Pour le calcul de l'index de spécialisation, Lorenz et Kidd [Lorenz et Kidd, 1994, p. 71] proposent l'utilisation d'un ratio mettant en rela-

tion trois métriques : le nombre de méthodes redéfinies, la profondeur dans l'arbre d'héritage et le nombre de méthodes déclarées. L'équation 2.20 donne le calcul correspondant pour une classe donnée c.

$$SIX(c) = \frac{NMO(c) \times DIT(c)}{NMD(c)} \tag{2.20}$$

Incidence de la POA. Nous avons vu précédemment (cf. sections 2.2.2, 2.2.8 et 2.2.9) que les métriques DIT, NMD et NMO pouvaient être modifiées par l'utilisation de la POA. La métrique SIX étant une combinaison de ces trois métriques, elle est donc concernée par l'impact de la POA sur la POO.

Nouvelle méthode de calcul. Il n'est pas nécessaire de modifier la méthode de calcul de la métrique SIX, car cette dernière est simplement une combinaison de métriques dont les méthodes de calcul ont déjà été étudiées et modifiées.

Exemple. La figure 2.2 présente un cas intéressant pour la métrique SIX. En effet, la classe `Curve` en héritant de `Line` voit sont DIT modifié. De plus, l'introduction de la méthode `getDelta()` modifie aussi bien la valeur de NMO que celle de NMD. Ce faisant, la valeur de SIX passe de 1 à 2, simplement à cause des aspects.

2.3 Conclusion sur l'étude des métriques

Dans ce chapitre nous avons fait l'étude de dix métriques de classe et proposé, pour chacune d'elle, une nouvelle méthode de calcul en fonction de l'impact de la POA. Le tableau 2.1 (p. 42) résume l'impact des différents mécanismes sur les métriques étudiées. Ce tableau montre bien que toutes les métriques sont concernées par la POA à différents niveaux. Le mécanisme d'introduction de méthodes est le plus présent, car on le retrouve sur toutes les métriques, sauf sur celles qui pourraient être dites « structurelles » telles que DIT, NOC ou encore NAD. Ces

dernières sont touchées soit par les modification au niveau de l'arbre d'héritage, soit par l'introduction de nouveaux champs.

Il nous faut maintenant implémenter une solution concrète afin d'appliquer nos calculs sur des exemples réels.

	Métriques									
Abstractions	WMC	DIT	NOC	CBO	RFC	LCOM	NAD	NMD	NMO	SIX
greffon - *before*	√			√	√	√				
greffon - *after*	√			√	√	√				
greffon - *around*	√			√	√	√				
introduction - champs						√	√		√	√
introduction - méthodes	√			√	√	√		√	√	√
introduction - héritage		√	√						√	√

TAB. 2.1: Incidence des mécanismes de la POA sur les métriques étudiées

CHAPITRE 3

IMPLÉMENTATION ET ARCHITECTURE

Dans ce chapitre nous présentons l'architecture[1] visant à analyser de façon concrète l'impact de la POA sur les programmes orientés objet. Il a été possible de mettre en place cette architecture grâce au travail théorique décrit dans le chapitre 2.

3.1 Introduction

Après avoir exploré les différents axes de recherche sur la POA dans le chapitre 1 et étudié son impact sur les métriques de classe dans le chapitre 2, nous implémentons maintenant concrètement les formules de calcul des métriques détaillées dans la section 2.2 ainsi que la chaîne de traitement des métriques pour évaluer l'impact de la POA sur la qualité par rapport à la POO. Cette phase peut se résumer en trois principales étapes :

1. **Modélisation.** D'abord, il est nécessaire d'obtenir une modélisation interne du programme à analyser, tant pour la partie orientée objet que pour la partie orientée aspect.
2. **Extraction.** Il faut ensuite implémenter une stratégie d'extraction de métriques sur le modèle créé. Cet implémentation fera intervenir les calculs formalisés dans le chapitre 2.
3. **Présentation.** Une fois le programme analysé et les données extraites, il s'agit de mettre en page et de présenter ces données. Nous avons choisi d'opter pour un format XML.

[1] Les chapitres 3 et 4 traitent tous deux de contenus techniques. Afin d'en alléger la forme et d'en rendre la lecture plus interactive, un site Web, pensé comme un compagnon de lecture, a été créé. L'annexe IV détaille le contenu du site ainsi que les méthodes pour y accéder.

4. **Évaluation de l'impact.** Une fois les résultats transformés, il est possible de les évaluer (*cf.* chapitre 4) en utilisant par exemple le logiciel VERSO (Visualisation pour l'Évaluation et la Ré-ingénierie des Systèmes à Objet [Langelier *et al.*, 2005]). Ce logiciel permet de visualiser de larges programmes et se base sur les valeurs des métriques associées aux objets pour générer sa représentation.

3.2 Modélisation

La donnée brute que nous avons choisi d'analyser est le code source d'un programme orienté aspect. Avant de pouvoir en extraire des métriques, il nous faut modéliser ce programme par le biais d'une structure interne.

Cette étape est primordiale pour la suite de notre travail. En effet, c'est la représentation des programmes sur laquelle les valeurs des métriques seront calculées. Son architecture et sa précision fixent donc les frontières à l'intérieur desquelles notre analyse évoluera.

3.2.1 Le métamodèle PADL

PADL (*Pattern and Abstract-Level Description Language*) est le fruit des travaux de Albin-Amiot [2003] et de Guéhéneuc [2003, p. 221]. Ce métamodèle définit un ensemble minimal de constituants nécessaires à la description de motifs de conception et des programmes orientés objet. Créé dans l'intention de « garantir la traçabilité des motifs de conception entre les phases d'implantation et de rétro-conception des programmes » [Guéhéneuc, 2003, p. 5], PADL offre une très bonne solution quant à la synthèse et à la modélisation du code source Java.

L'architecture de PADL a été conçue de façon à être facilement modifiable. En effet, comme préconisé par certain chercheurs [Astudillo, 1997], l'implémentation de PADL divise le modèle en une hiérarchie de types d'un côté (en utilisant le

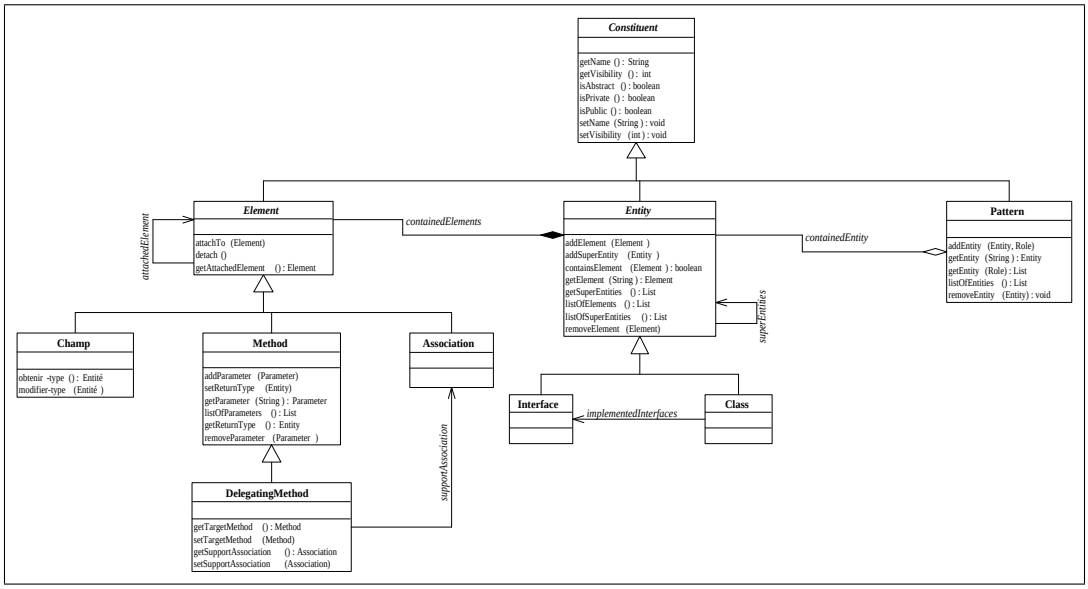

FIG. 3.1: Sous-ensemble simplifié du métamodèle PADL

mécanisme d'interface de Java) et en une hiérarchie de classes de l'autre. La hiérarchie de types est destinée à être utilisée comme l'interface publique du métamodèle, alors que la hiérarchie de classes liée à la première reflète le comportement interne du métamodèle. Elle est destinée aux développeurs du métamodèle et est cachée des utilisateurs. La figure 3.1 présente le coeur de la hiérarchie de classes de PADL.

Ses possibilités quant à la modélisation du code source Java et son architecture soignée nous ont fait choisir PADL comme métamodèle de base. Plusieurs fois étendu afin de supporter des langages tels que le *Abstract Object Language* [Antoniol et Guéhéneuc, 2005] ou encore le C++ [Flores et Robidoux, 2004], nous y avons à notre tour ajouté le support du langage AspectJ.

3.2.2 Extension du métamodèle PADL

Afin d'étendre le métamodèle PADL et d'y intégrer le support du langage AspectJ, nous avons procédé en deux étapes distinctes, l'extension du modèle et l'extraction des données.

Extension du modèle. Nous avons dû étendre le métamodèle PADL afin d'y intégrer le support d'AspectJ. Nous avons respecté la sémantique imposée par la relation entre POA et POO (*cf.* section 1.1.5). Cette relation nous dicte que seule la partie aspect du programme a connaissance de l'existence de la partie objet et non l'inverse. Nous nous sommes appliqués à refléter cette contrainte dans le métamodèle.

L'extension du métamodèle a été grandement facilitée par son architecture particulière : la séparation entre typage et implémentation. L'extension de la hiérarchie de typage nous permet de respecter la contrainte citée plus haut et de séparer ainsi, dans le métamodèle, la partie aspect du modèle orienté objet. D'autre part, l'extension de la hiérarchie de classe nous permet de rendre compte des similarités entre AspectJ et Java. Par exemple, si un *Aspect* et une *Classe* sont deux types tout à fait différents, leur comportement est assez similaire. Il est donc logique que l'implémentation de l'entité *Aspect* hérite du comportement de l'entité *Classe* dans notre métamodèle. La figure 3.2 montre comment nous avons traduit ces contraintes au niveau du métamodèle dans le cas de la relation entre classe et aspect. Le module *PADL Aspect Creator* étend cette manière d'implémenter à l'ensemble des mécanismes du langage AspectJ. Ce module propose une extension de la hiérarchie de typage, ainsi que son pendant au niveau de la hiérarchie de classes. La classe `AjcCompilerWrapper` permet d'utiliser le compilateur *ajc*, alors que la classe `AspectWalker` étend le visiteur fourni par l'*Abstract Structure Model* dans le but de créer concrètement les différentes entités du modèle. Les sources de ce module peuvent être consultées en ligne via CVS (*cf.* annexe IV). Une fois le métamodèle étendu, nous avons du mettre en place le processus d'extraction des données.

Extraction des données. Nous avons choisi d'analyser le code source des programmes orientés aspect. Pour ce faire, nous avons étudié en détail le fonctionnement du langage AspectJ ainsi que celui du plugiciel (*plugin*) AJDT [Clement *et*

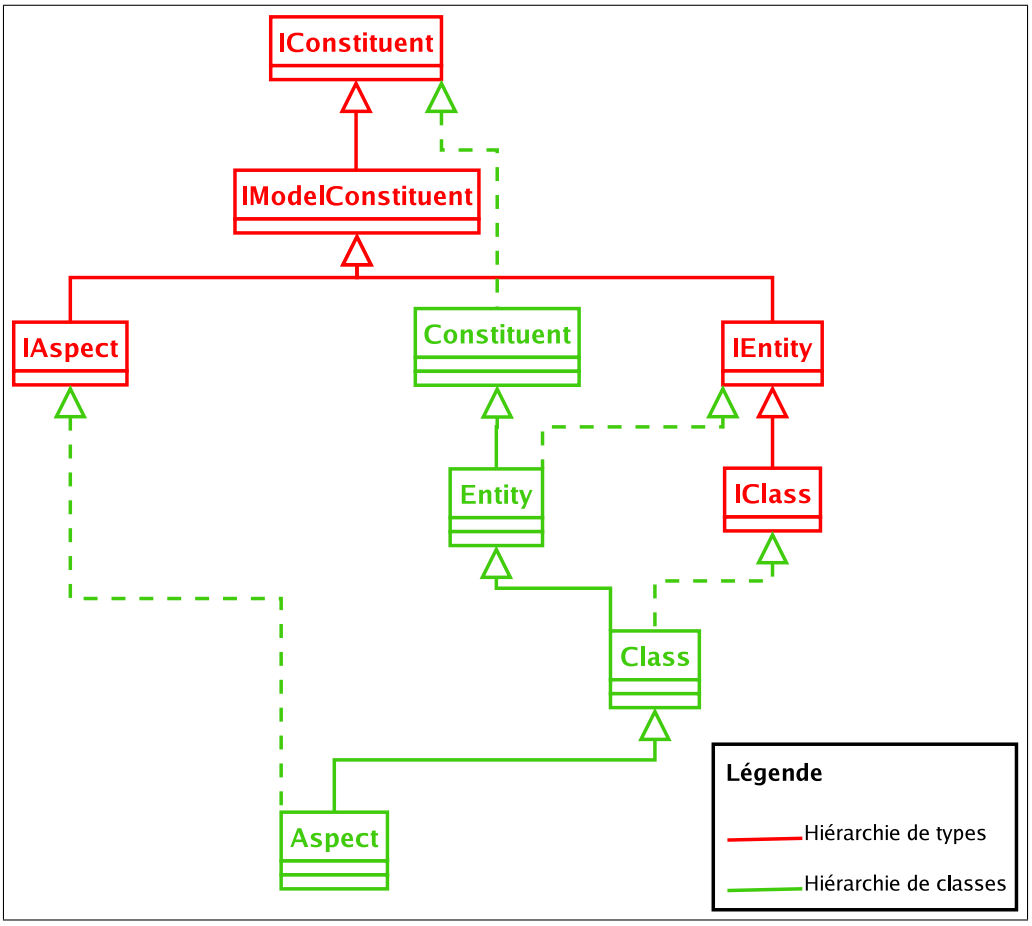

FIG. 3.2: Ajout de l'entité *Aspect* dans le métamodèle PADL

al., 2003] d'Eclipse. L'étude de cet outil nous a fourni une solution adéquate pour le traitement du code source AspectJ.

En effet, le compilateur d'AspectJ, *ajc*, crée une structure de données appelée *Abstract Structure Model* au moment du tissage. Cette structure contient tous les détails d'implémentation, sous forme brute, du programme AspectJ compilé. Elle implémente le pattern *Visiteur* [Gamma, 1995, p. 331] qui, adapté à nos besoins, nous permet de parcourir la structure et d'en extraire l'information. Le processus de traitement d'extraction des données se décline comme suit :

1. Les données en entrée sont constituées d'un fichier texte au format LST (chaque ligne contient le chemin relatif à un fichier source) et d'une hiérarchie de répertoires contenant les fichiers source ;
2. Notre système appelle le compilateur *ajc* et lui fournit les données sources à tisser ;
3. L'*Abstract Structure Model* est obtenu du compilateur *ajc* par notre système ;
4. Les données du programme orienté aspect analysé sont prêtes à être extraites via le pattern *Visiteur* afin de construire notre modèle PADL.

3.2.3 Processus de traitement

La figure 3.3 présente le processus de modélisation. La donnée source est un programme orienté aspect constitué de sources orientées objets et de sources orientées aspects. Tout d'abord, les sources objet sont isolées et transformées en fichier JAR pour les rendre plus facilement manipulables. Ce fichier JAR, que nous identifions comme le cœur du programme, est fourni au module *PADL ClassFile Creator* qui produit un premier modèle PADL correspondant à la partie orientée objet du programme. Nous intervenons ensuite grâce à notre module *PADL Aspect Creator*. Ce dernier prend en entrée les sources (objet et aspect) du programme et le modèle intermédiaire créé précédemment. Notre module analyse le programme et

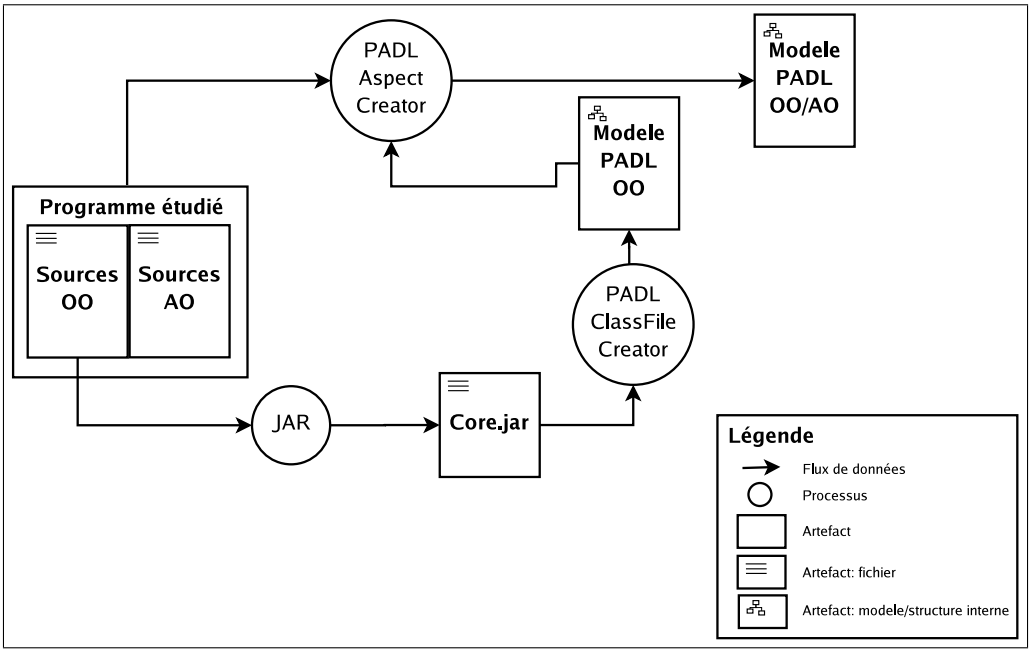

FIG. 3.3: Schéma du processus de modélisation du code source AspectJ

complète le modèle en y ajoutant la partie aspect. Il peut alors terminer la phase de modélisation en produisant le modèle hybride POO/POA.

3.3 Extraction des métriques

Nous possédons maintenant un modèle du programme orienté aspect à analyser. À partir de ce modèle nous pouvons calculer et extraire les métriques détaillées au chapitre 2.

3.3.1 Le module d'extraction de métriques POM

POM (*Primitives, Operators, Metrics*) est une bibliothèque développée par Farouk Zaidi [2004] dans le but d'extraire des métriques en se basant sur un modèle PADL.

Cette bibliothèque utilise un système de primitives et d'opérateurs inspiré de la théorie ensembliste mathématique afin de calculer des métriques de classes telles

que celles de Chidamber et Kemerer [1994] ou celles de Lorenz et Kidd [1994]. La figure 3.4 présente l'architecture simplifiée de la bibliothèque POM.

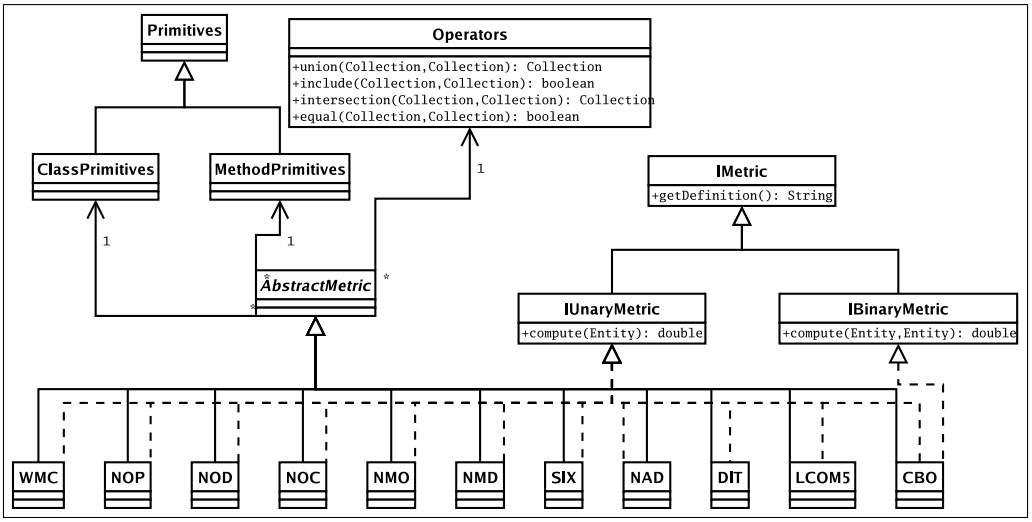

FIG. 3.4: Architecture simplifiée de la bibliothèque POM

Nous avons réutilisé la bibliothèque POM afin de calculer les métriques sur un programme orienté aspect.

3.3.2 Extension de POM

Puisque l'ajout de la composante aspect à un modèle PADL est transparent pour la partie orientée objet du modèle, ce dernier reste un modèle PADL orienté objet valide. Il est donc toujours possible d'appliquer l'outil POM pour extraire des métriques de classe. Les résultats ne tiendront compte que de la partie orientée objet du modèle.

Afin de garder ce comportement, nous avons utilisé la programmation orientée aspect dans le but d'implémenter les résultats de notre étude théorique de l'impact de la POA sur les métriques de classe. De cette manière, nous ne modifions pas la bibliothèque POM de façon directe. Mais, lorsque le modèle PADL représentant le programme analysé contient une partie aspect, notre implémentation est capable de capturer les modifications faites par AspectJ sur le programme étudié. Il peut alors

```
1 public class NAD extends AbstractMetric implements IMetric, IUnaryMetrics {
2         public NAD(final IAbstractLevelModel anAbstractLevelModel) {
3                 super(anAbstractLevelModel);
4         }
5
6         public double compute(IEntity entity) {
7                 return getActorsOfTheMetric(entity).size();
8         }
9
10        public List getActorsOfTheMetric(IEntity entity) {
11                return super.classPrimitives.implantedFields(entity);
12        }
13
14        public String getDefinition() {
15                return "number of attributes declared";
16        }
17 }
```

Code 3.1: Implémentation originale de la métrique NAD dans POM

les répercuter sur les résultats des métriques extraites. L'utilisation de la POA pour étendre l'outil POM nous permet une réutilisation optimale de cette bibliothèque. Nous illustrons notre architecture en détaillant l'implémentation de la métrique *Number of Attributes Declared* (NAD).

Implémentation originale de NAD. L'extrait de code 3.1 présente l'implémentation originale de la métrique NAD dans la bibliothèque POM. Afin de calculer le nombre d'attributs déclarés par une entité `entity` de type `IEntity` (*e.g.* une classe Java), la bibliothèque POM fait appel à la primitive `List declaredFields(IEntity)` qui lui renvoie la liste des attributs déclarés par l'entité selon le modèle PADL étudié.

Impact de la POA sur NAD. L'étude de la métrique NAD, détaillée dans la section 2.2.7, nous a permis de conclure que sa valeur était augmentée par le nombre d'attributs introduits par des aspects.

L'extrait de code 3.2 présente l'implémentation, en langage AspectJ, de l'impact de la POA sur la métrique NAD. L'aspect abstrait `AbstractAspectMetric` est le parent de tous les aspects dédiés au calcul de l'impact de la POA sur les différentes métriques étudiées. Cet aspect déclare un point de coupure `computeUnary` qui associe un point de jointure à l'exécution d'une méthode `public double`

```
1 public abstract aspect AbstractAspectMetric {
2
3         pointcut computeUnary() :
4                 execution(public double IUnaryMetrics+.compute(IEntity));
5
6         pointcut computeBinary() :
7                 execution(public double IBinaryMetrics+.compute(IEntity, IEntity));
8
9         protected void logDiff(final IEntity computedEntity,
10                                final double original,
11                                final double modified){
12                this.getLocalLogger().debug(computedEntity.getName() +
13                                            ": " + original + "/" +
14                                            modified);
15        }
16
17        abstract protected Logger getLocalLogger();
18 }
19
20 public aspect NADAspectMetric extends AbstractAspectMetric {
21
22        private final Logger logger = Logger.getLogger(this.getClass());
23
24        pointcut computeSpecific(NAD metric, IEntity entity) :
25                computeUnary() &&
26                this(metric) && args(entity);
27
28        double around(NAD metric, IEntity entity) : computeSpecific(metric, entity) {
29                double ooResult = proceed(metric, entity);
30
31                final AspectMap map = AspectMap.getAspectMap();
32
33                double aoResult = ooResult;
34
35                if(map.containITFieldRelations(entity)){
36                        aoResult += map.getITFieldRelations(entity).size();
37                }
38
39                if(ooResult != aoResult)
40                        this.logDiff(entity, ooResult, aoResult);
41
42                return aoResult;
43        }
44
45        protected Logger getLocalLogger(){
46                return this.logger;
47        }
48 }
```

Code 3.2: L'aspect NADAspectMetric

`compute(IEntity)` appartenant à une entité appelée `UnaryMetric` ou à ses sous-classes. Par conséquent, l'exécution de la méthode `compute` de la classe `NAD` de POM est capturée par ce point de coupure. L'aspect `NADAspectMetric` hérite de l'aspect `AbstractAspectMetric` et nous permet de spécifier le comportement à adopter lors du calcul de la métrique NAD par l'outil POM.

Le point de coupure `computeSpecific` réutilise le point de coupure `computeUnary` hérité tout en spécifiant, grâce à l'instruction `this`, que le point de jointure ne concerne que les objets de type `NAD`. Il en récupère l'entité `entity` passée en argument de la méthode compute via l'instruction `args`. Ainsi, ce point de coupure associe un point de jointure seulement à l'exécution de la méthode `compute` appartenant à la classe `NAD` de POM.

Afin d'implémenter notre méthode de calcul, l'aspect `NADAspectMetric` déclare un greffon de type `around`. Ce greffon remplace l'exécution de la méthode `compute`. Tout d'abord, l'instruction `proceed` permet au greffon d'exécuter de façon normale le calcul de la métrique NAD sur l'entité `entity`. Cette dernière n'ayant pas conscience de l'existence des aspects dans le modèle, la valeur obtenue est la valeur originale de NAD sur cette entité sans l'impact de la POA. Ensuite, le greffon vérifie si cette entité bénéficie de l'introduction d'attributs par des aspects appartenant au modèle. Dans l'affirmative, elle récupère la liste de ces introductions avec l'instruction à la ligne 36, et en ajoute la taille au résultat original. Ce résultat peut alors être retourné à l'objet qui a ordonné l'exécution de la méthode `compute` de la classe `NAD`.

L'implémentation de la métrique NAD illustre bien notre approche globale : spécifier les points de coupure de l'aspect `AbstractAspectMetric`, associer un greffon de type `around`, puis analyser le modèle afin de calculer l'impact de la POA.

3.3.3 Problèmes rencontrés

L'instrumentation du code des méthodes introduites et des greffons nous a posé certains problèmes. En effet, le modèle fourni par le compilateur *ajc* n'étais pas assez précis pour nous donner accès à ces éléments. Cette limitation inhérente au modèle nous a empêché d'implémenter les métriques RFC, LCOM et CBO qui font toutes intervenir ces éléments dans leur calcul.

Plutôt que de nous contenter de résultats approximatifs et difficilement évaluables, nous avons préféré implémenter l'impact de la POA sur des métriques telles que NOD (nombre de descendants) et NOP (nombre d'ancêtres). Elles ont l'avantage d'être validables et de donner de l'information qui complète bien les métriques déjà implémentées DIT et NOC.

3.3.4 Optimisations

Le génie logiciel expérimental est un champ de recherche qui est appelé à traiter de gros volumes de données. Nous nous sommes appliqués à optimiser notre système de plusieurs manières.

Relation OO/AO dans le modèle. Notre extension du métamodèle PADL nous a posé certains problèmes pendant l'implémentation du calcul des métriques. En effet, ces calculs se font sur des entités appartenant à la partie orientée objet du programme étudié. Or, dans le modèle, ces entités n'ont pas « conscience » de l'existence de la partie aspect car la relation entre OO et OA est unidirectionnelle. Ceci implique, pour chaque calcul, l'analyse du modèle afin de capturer les modifications apportées par la composante aspect sur l'entité évaluée.

Afin d'optimiser nos calculs, nous avons mis en place une classe `AspectMap`, qui permet de faire le lien entre les entités et les aspects du modèle. Elle permet aussi de connaître, pour une entité donnée, les différents points de coupure et introductions qui lui sont associés. Nous avons encore une fois utilisé la POA afin de réaliser

cette optimisation : un aspect `AspectMapper` intervient pendant la création du modèle. Lorsqu'un aspect (ou un de ses éléments) est ajouté au modèle, l'aspect `AspectMapper` récupère l'information et la transmet à la classe `AspectMap`. Comme on peut le voir dans l'extrait de code 3.2, la classe `AspectMap`, qui implémente le pattern `Singleton`, est récupérée à la ligne 31. Il est ensuite possible de prendre connaissance de la relation de l'entité évaluée envers les aspects du modèle grâce à des méthodes telles que celles utilisées aux ligne 35 et 36.

Cache. La POA nous a offert, encore une fois, une méthode simple pour réduire le nombre de calculs et optimiser les temps de calcul. Les mécanismes d'introduction nous permettent de mettre en place une méthode de mise en mémoire cache des résultats.

L'extrait de code 3.3 présente l'implémentation de l'impact de la POA sur la métrique *Number of Descendents* (NOD) dont le calcul est récursif. Il faut calculer la liste des descendants de chaque entité appartenant à la hiérarchie de l'entité étudiée avant de pouvoir calculer la valeur de NOD. Cette liste a une valeur fixe pour chaque entité du modèle, mais elle est recalculée pour certaines entités à chaque fois que la métrique NOD sera appelée. Afin de réduire le nombre de calculs, il est utile de sauvegarder les résultats dans une mémoire cache. La POA nous permet de le faire de façon transparente. Il nous suffit d'introduire un attribut `cacheAspectedDescendent` de type `List` à l'interface `IEntity` (ligne 5). Ensuite cet attribut est utilisé comme une mémoire cache (ligne 24-26 et 32).

Grâce à ces quelques lignes de code, nous avons mis en place un système de mémoire cache qui sauvegarde la valeur calculée à même l'entité concernée.

3.3.5 Processus de traitement

La figure 3.5 présente le processus d'extraction des valeurs des métriques. Le modèle hybride POO/POA est fourni en entrée au module POM. Notre système

```
1 public aspect NODAspectMetric extends AbstractAspectMetric {
2
3         private final Logger logger = Logger.getLogger(this.getClass());
4
5         private List IEntity.cachedAspectedDescendents = null;
6
7         pointcut computeSpecific(NOD metric, IEntity entity) :
8                 computeUnary() &&
9                 this(metric) && args(entity);
10
11        double around(NOD metric, IEntity entity) : computeSpecific(metric, entity) {
12                double ooResult = proceed(metric, entity);
13
14                double aoresult = this.getDescendents(entity, metric).size();
15
16                this.logDiff(entity, ooResult, aoresult);
17
18                return aoresult;
19        }
20
21        private List getDescendents(final IEntity entity, final NOD metric){
22
23                //Test if result cached
24                if(entity.cachedAspectedDescendents != null){
25                        return entity.cachedAspectedDescendents;
26                }
27
28                //Compute the result
29                ...
30
31                //Cache the result
32                entity.cachedAspectedDescendents = aspected_Desc;
33                return entity.cachedAspectedDescendents;
34        }
35
36        protected Logger getLocalLogger(){
37                return this.logger;
38        }
39
40 }
41
```

Code 3.3: L'aspect NODAspectMetric

demande au module POM d'extraire un ensemble précis de métriques. Le module *PADL Aspect Metric*, qui contient notre implémentation de l'impact de la POA sur le métriques de classes, introduit ses aspects afin de veiller à ce que les calculs des différentes métriques soient ajustés en fonction des informations présentes dans le modèle hybride POO/POA. Enfin, la suite de métrique est extraite et sa valeur sauvegardée dans une structure intermédiaire.

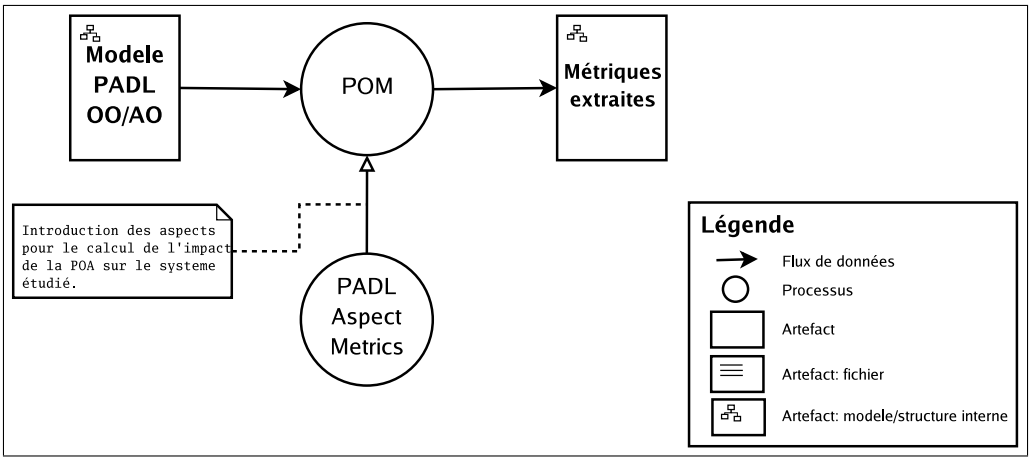

FIG. 3.5: Schéma du processus d'extraction des métriques

3.4 Mise en page et présentation des résultats

Une fois les métriques calculées, il nous faut mettre en forme les résultats obtenus afin de pouvoir procéder à l'évaluation des résultats. Nous avons utilisé un format basé sur XML pour sa versatilité et le logiciel VERSO [Langelier *et al.*, 2005] pour ses possibilités quant à la visualisation tridimensionnelle des systèmes à objets. L'outil de visualisation VERSO permet l'affichage en trois dimensions et l'analyse de programmes en utilisant les métriques objet. Cet outil sera discuté plus en détail dans le chapitre 4.

L'outil VERSO utilise le format XML pour ses données. Un fichier XML représente un programme et, pour chacune de ces entités, les métriques qui ont été calculées.

Afin de nous conformer à ce format, nous avons créé le module PomXML. Ce module a pour but de traiter des résultats bruts obtenus après extraction des métriques afin de les transformer au format XML voulu et de proposer différents modes de présentation (notamment via l'outil VERSO, dans un navigateur ou au format texte).

1. Nous avons développé un Schema XSD correspondant à la structure XML utilisée par VERSO (*cf.* annexe V p. 96).
2. Une partie du module (codée en Java et intégrée à la bibliothèque POM) a été mise en place de façon à pouvoir transformer directement les résultats obtenus après l'extraction en un fichier XML répondant au Schema XSD développé.
3. Outre la possibilité d'utiliser le logiciel VERSO afin de visualiser nos résultats, une feuille de transformations XSLT a été créée pour pouvoir présenter les résultats dans un butineur Internet.

3.4.1 Processus de traitement

La figure 3.6 présente le processus de présentation. Une fois extraites et sauvegardées dans une structure intermédiaire, les métriques sont fournies au module *PomXML*. Celui-ci produit alors un fichier XML à partir de ces résultats. La validité de ce fichier est vérifiée grâce au processus de validation XML et au Schema XSD que nous avons développé. Il est possible d'afficher ces résultats soit dans un butineur (via notre feuille de transformation XSLT), soit dans l'outil VERSO.

3.4.2 Critiques

L'utilisation du méta-langage XML offre un format puissant et de nombreuses technologies connexes très utiles : validation, transformation, extraction. De plus, XML jouit aujourd'hui d'une très bonne intégration avec Java.

Son usage implique cependant quelques concessions. En effet, XML est tout

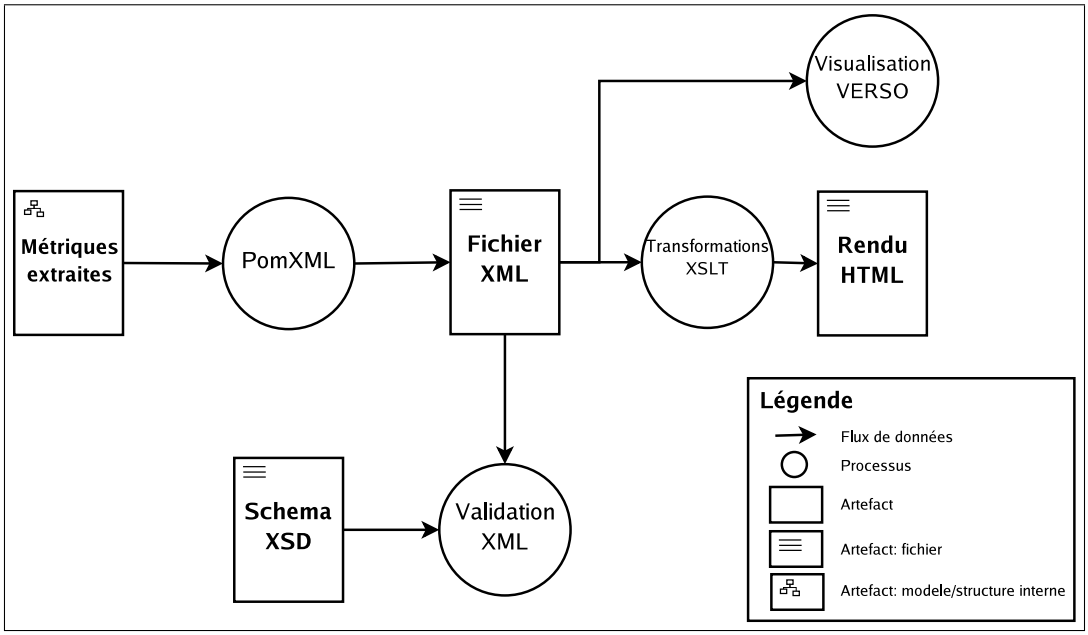

FIG. 3.6: Schéma du processus de présentation

d'abord un format dont la mise en place s'avère compliquée et lourde au niveau de l'implémentation. C'est aussi un format très verbeux qui produit de gros fichiers difficiles à manipuler. Enfin, XML est un format rigide qui supporte mal les modifications (sans l'utilisation d'outils d'automatisation).

Malgré ces quelques problèmes, XML reste la solution optimale pour nos besoins en matière de représentation. Les autres formats possibles (tel que le *comma separated value*) sont souvent tributaires des mêmes problèmes et n'offrent pas autant de versatilité que XML.

3.5 Conclusion

Dans le cadre de notre projet, nous avons réalisé un système permettant l'analyse d'un programme orienté aspect en prenant en charge la modélisation, l'extraction des métriques et la présentation des résultats. Le diagramme 3.7 présente les trois processus unifiés.

L'implémentation de ce projet a été entièrement réalisée avec les langages Java

et AspectJ[2]. Le développement a été fait grâce à l'environnement Eclipse 3.1[3] et à son plugin AJDT[4] (pour le support d'AspectJ). Pour la phase de présentation des résultats, nous avons utilisé les technologies XML et développé nos solutions avec le logiciel oXygen[5]. De plus, tout notre projet a été automatisé grâce à l'outil Ant[6] : depuis la compilation de nos modules, jusqu'à la production des fichiers résultats, en passant par la compilations et l'exécution des tests unitaires, et l'exécution des nos études de cas. Même le site Web présenté à l'annexe IV est généré automatiquement grâce à l'outil Maven[7].

Nous devons maintenant tester le système que nous avons implémenté et vérifier notre hypothèse selon laquelle la quantification de l'impact de la POA sur un système orienté objet nous permettrait d'évaluer le programme analysé en termes de qualité.

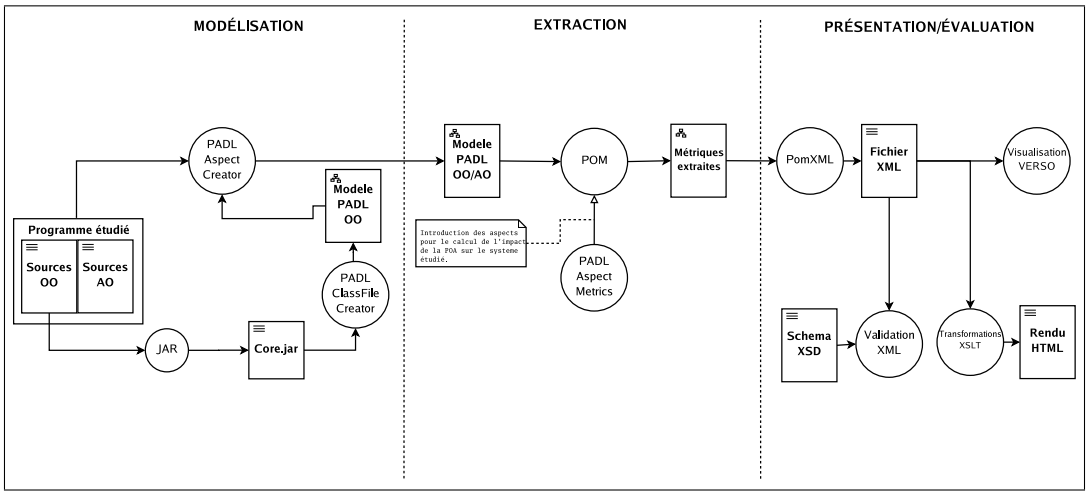

FIG. 3.7: Schéma du processus global : modélisation, extraction et présentation

[2] http://www.eclipse.org/aspectj
[3] http://www.eclipse.org
[4] http://www.eclipse.org/ajdt
[5] http://www.oxygenxml.com
[6] http://ant.apache.org
[7] http://maven.apache.org

CHAPITRE 4

TESTS ET VALIDATION

Dans ce chapitre nous présentons les travaux que nous avons réalisés dans le but de tester les solutions logicielles présentée dans le chapitre 3 et d'évaluer la qualité des programmes orientés aspect grâce à notre architecture.

4.1 Objectifs

Nous avons proposé et implémenté dans le chapitre 3 une architecture permettant de quantifier l'impact de la programmation orientée aspect. Le programme créé, *PADL Aspect Metric*, nous permet de capturer cet impact sur dix métriques de classe connues ([Chidamber et Kemerer, 1994; Lorenz et Kidd, 1994]). Il s'agit maintenant de tester et de valider notre application avant de pouvoir l'utiliser pour évaluer la qualité des programmes orientés aspect[1].

4.2 Tests et validation

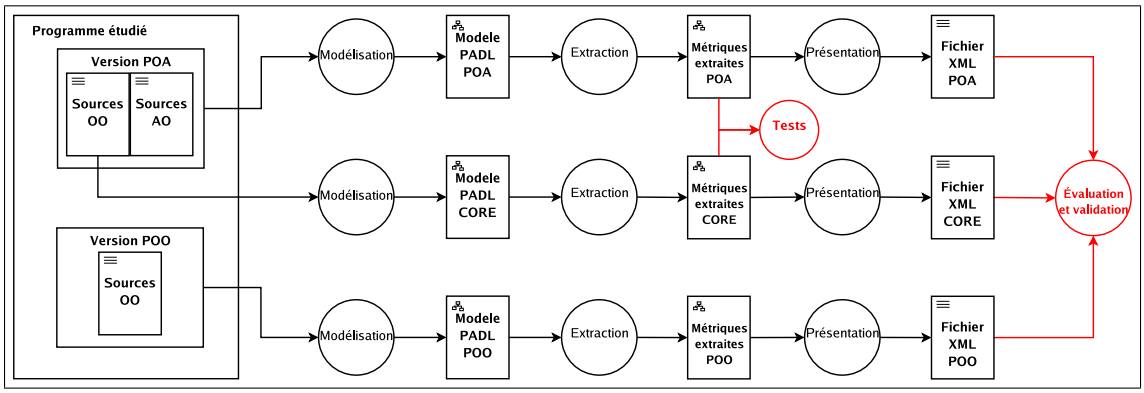

FIG. 4.1: Méthode de comparaison POO/PAO

[1]Les programmes et les exemples utilisés dans ce chapitre peuvent être consultés et téléchargés depuis le site Web détaillé dans l'annexe IV.

Afin de pouvoir tester, valider, ou encore évaluer un programme, nous avons besoin de données à comparer. La figure 4.1 présente notre méthode de comparaison. Nous avons minimalement un programme orienté aspect. Nous isolons la partie orientée objet de ce programme que nous considérons comme son coeur (*CORE*). Ces deux éléments, la partie objet et le programme au complet, sont transformés grâce au trois étapes détaillées dans le chapitre 3 : modélisation, extraction et présentation. Après la phase d'extraction, nous obtenons deux jeux de métriques. Le premier correspond au programme sans les préoccupations implémentées par les aspects et le second correspond au même programme après introduction des aspects. En comparant ces deux jeux de métriques, nous établissons de façon concrète l'impact de la POA sur le programme étudié. Il est dès lors possible de tester notre application.

Dans le meilleur des cas, nous possédons aussi une version orientée objet complète du programme. Dans cette version, les préoccupations précédemment encapsulées par les aspects ont été implémentées avec la POO. Elle nous permet d'obtenir un troisième jeu de métriques qui nous servira à comparer l'implémentation orientée objet des préoccupations par rapport à l'implémentation orientée aspect. Malheureusement, il est très rare d'avoir accès à ce type de programme.

4.2.1 Tests unitaires

Pour tester notre application de façon optimale, nous avons utilisé le cadre d'application JUnit[2]. Implémenté pour Java, il nous a permis de mettre en place 34 tests unitaires qui nous ont suivis tout au long de la phase d'implémentation. Ces tests sont divisés en trois catégories :

- **TestModelsSetup :** regroupe une série de tests visant à vérifier si les deux modèles créés à la phase de modélisation sont complets et cohérents ;

[2] `http://www.junit.org`

```
1 public void assertAspectMetric(final String entitytotest,
2                                 final String metric,
3                                 final double diff){
4         IEntity OOentitytotest = (IEntity)COREcodelevelmodel.getActorFromName(entitytotest);
5         IEntity AOentitytotest = (IEntity)AOcodelevelmodel.getActorFromName(entitytotest);
6
7         assertTrue("The entity to test does not exists in the OO model",OOentitytotest != null);
8         assertTrue("The entity to test does not exists in the AO model",AOentitytotest != null);
9
10        double oovalue = COREmetrics.compute(metric, OOentitytotest);
11
12        double aovalue = AOmetrics.compute(metric, AOentitytotest);
13
14        assertEquals("Aspect" + metric + " on " + entitytotest + " failed.",
15                     diff, (aovalue - oovalue), 0);
16 }
17
18 public void assertMapping(final String mappingType,
19                           final IEntity entity,
20                           final int nbMappingExpected,
21                           final int nbMapping){
22        assertEquals("Wrong " + mappingType + " relation mapping on " + entity.getName(),
23                     nbMappingExpected, nbMapping);
24 }
25
```

Code 4.1: Assertions définies pour les tests unitaires

- **TestAspectMapping :** s'assure que tous les liens entre la partie aspect et la partie objet du programme ont été répercutés dans le modèle ;
- **TestAspectMetric :** contient une série de tests qui comparent les valeurs des métriques, avant et après introduction des aspects, et s'assurent que ces valeurs sont exactes. Pour faciliter l'implémentation de ces tests, nous avons factorisé les tâches à accomplir dans la méthode *assertAspectMetric(String, String, double)* (*cf.* extrait de code 4.1). Cette méthode récupère les valeurs d'une métrique `metric` pour une entité `entitytotest` avant et après introduction des aspects. Elle vérifie ensuite que la différence entre ces deux valeurs correspond au résultat `diff` attendu.

4.2.2 Validation

Pour valider la version finale de notre architecture, nous avons réuni un corpus de 6 programmes orientés aspect. Ils ont été traités par notre application et les mesures obtenues ont été vérifiées manuellement. Le tableau 4.1 présente les caractéristiques principales de ces programmes. Nous avons créé certains de ces programmes et obtenu les autres de l'Internet :

- *Persistance :* est un exemple d'implémentation de la persistance [Miles, 2004] ;
- *Exemple JUnit :* est un exemple que nous avons créé pour les besoins de nos tests unitaires ;
- *FigureElement :* est l'exemple que nous avons créé pour nos démonstrations à partir d'une entrevue donnée par Kiczales [2003] ;
- *CacaoSolver :* est un solveur de contraintes[3] implémenté par Douence et Jussien [2002] ;
- *Robot Simulation :* est une simulation mécanique programmée en Java et refactorisée avec AspectJ ;
- *AJHotDraw :* est une restructuration orientée aspect de l'application JHotDraw[4].

	Partie objet				**Partie aspect**					
	Nb. classes	**Nb. interfaces**	**Nb. champs**	**Nb. méthodes**	**Nb. aspects**	**Nb. points de coupure**	**Nb. greffons**	**Nb. champs introduits**	**Nb. méthodes introduites**	**Nb. modif. hiérarchie**
Persistance	5	1	7	29	3	6	3	2	5	4
Exemple JUnit	7	2	3	6	4	3	3	4	5	3
FigureElement	8	2	9	31	2	6	6	2	5	2
CacaoSolver	10	0	21	52	5	8	8	5	1	0
Robot Simulation POA	47	7	214	538	8	8	1	0	0	16
AJHotDraw	255	49	708	3348	10	3	5	1	20	10
Robot Simulation POO	47	7	214	538	–	–	–	–	–	–
JHotDraw	255	49	709	3371	–	–	–	–	–	–

TAB. 4.1: Corpus utilisé pour la validation

[3] `http://www.emn.fr/x-info/douence/Protos/CacaoSolver.java`
[4] `http://ajhotdraw.sourceforge.net` et `http://jhotdraw.org`

4.3 Évaluation

L'objectif final de notre travail est d'utiliser notre étude de l'impact de la POA sur les programmes orientés objet dans le but d'en évaluer la qualité. Notre première intuition a été de nous servir de l'apprentissage automatique.

4.3.1 Apprentissage automatique

L'apprentissage automatique est aujourd'hui très utilisé dans le domaine du génie logiciel expérimental pour évaluer la qualité en se basant sur des métriques [Khosravi, 2005, p. 181].

Imaginons que nous possédions des programmes bien documentés et dont la qualité a été évaluée par des experts. Nous aurions donc, pour chaque entité des programmes évalués, des informations telles que : l'entité X a une bonne réutilisabilité, l'entité Y a une mauvaise modularité, l'entité Z a une portabilité moyenne, etc. Nous pourrions alors extraire les métriques de ces programmes et les fournir, avec les données des experts, à un moteur d'apprentissage de règles tel que WEKA[5]. Cette application pourrait lier les valeurs des métriques aux évaluations des experts pour donner des règles du type :

```
La réutilisabilité d'une entité E est:
   bonne    si LCOM(E) <= 0.5 et CBO(E) <  3
   moyenne  si LCOM(E) <= 0.5 et CBO(E) >= 3
   mauvaise si LCOM(E) >  0.5 et CBO(E) >= 3
```

Cette technique d'apprentissage donne de très bons résultats mais nécessite pour cela de très gros corpus de données. Dans notre cas, cela signifie que nous avons besoin de nombreux programmes et que chacun d'entre eux doit être documenté et évalué par des experts. Comme nous l'avons précisé dans la section précédente,

[5] `http://www.cs.waikato.ac.nz/ml/weka`

notre corpus de test se réduit malgré tous nos efforts[6] à six programmes de petite ou moyenne taille. La programmation orientée aspect étant un paradigme jeune, il est très difficile d'en trouver des exemples complexes.

Devant ce problème, nous avons opté pour une autre approche qui consiste à utiliser la visualisation et des experts pour évaluer la qualité des programmes.

4.3.2 Visualisation

Le logiciel VERSO (Visualisation pour l'Évaluation et la Ré-ingénierie des Systèmes à Objet [Langelier *et al.*, 2005]) permet de visualiser des programmes orientés objet en trois dimensions. Il prend comme donnée source un fichier XML représentant le programme à visualiser ainsi que les valeurs des métriques de ses entités. C'est le même format que nous avons utilisé pour notre étape de présentation (*cf.* section 3.4). La figure 4.2 donne un exemple du type de visualisation obtenu grâce à l'outil VERSO. Les entités du programme sont regroupées selon leur *package* et disposées en fonction d'un algorithme de placement de type *treemap* [Johnson et Shneiderman, 1991]. Les entités sont représentées par des rectangles qui ont trois attributs : la hauteur, l'orientation et la couleur. Chaque attribut peut être associé à une métrique. Selon la valeur des métriques associées, le rectangle sera plus ou moins grand, aura une orientation différente et sa couleur variera du bleu au rouge.

Cette technique de visualisation est tout à fait adaptée à nos besoin. En effet, nous désirons que l'évaluateur compare un même programme dans différents états : avant implémentation de certaines préoccupations, après implémentation de ces préoccupations avec la POA, et après implémentation de ces préoccupations avec la POO (*cf.* tableau 4.1 p. 64). En visualisant ces différents états grâce au logiciel VERSO, il sera capable de remarquer les différences au niveau des métriques et d'en tirer possiblement des conclusions quant à la qualité du programme. Il aura

[6] Nous avons contacté sans succès plusieurs auteurs dont Celina Gibbs, auteur d'une version orientée aspect de MMTK [Gibbs et Coady, 2004].

simplement à superposer ses différentes représentations ou à les alterner sous un même angle de caméra afin de faire apparaître ces différences.

Les images 4.2 et 4.3 représentent le coeur du programme AJHotDraw dans deux états différents. La première n'inclus pas les aspects alors que la seconde est le résultat de notre extraction des métriques après introduction des aspects. La métrique DIT est associée à l'orientation des rectangles, WMC à leur hauteur, et SIX à leur couleur. On peut noter, par exemple, que la classe *JavaDrawViewer* (indiquée sur les figures 4.2 et 4.3) a subi un changement dû à l'impact de la POA. En effet, lorsqu'on examine les valeurs des métriques de cette classe, on remarque qu'après introduction des aspects les valeurs de DIT et WMC pour cette classe ont augmenté de 1, et que SIX a augmenté de 0.8.

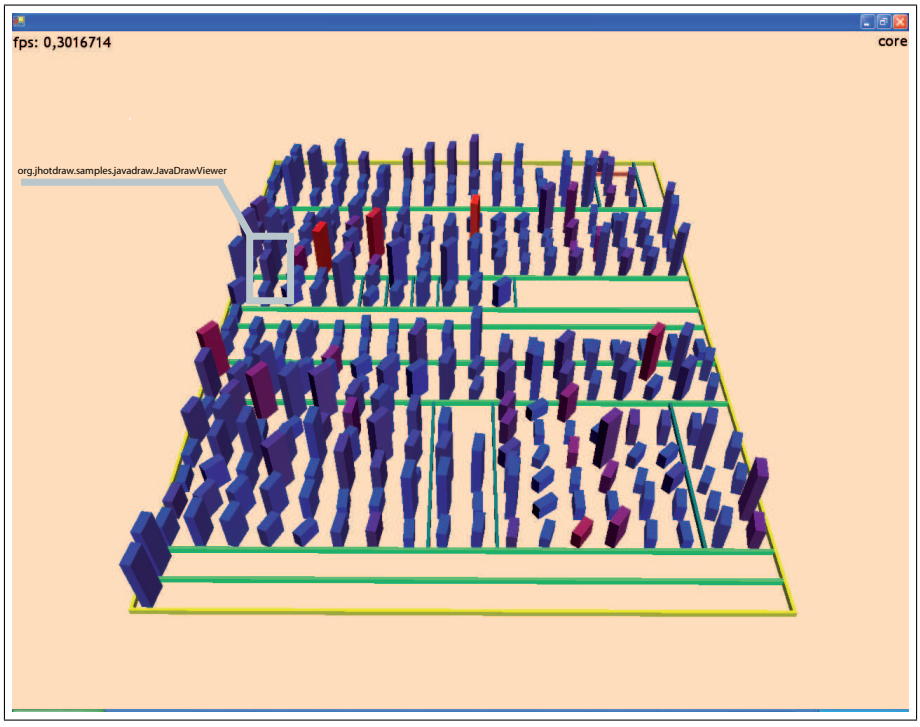

FIG. 4.2: La vue du cœur du système, avant introduction des aspects

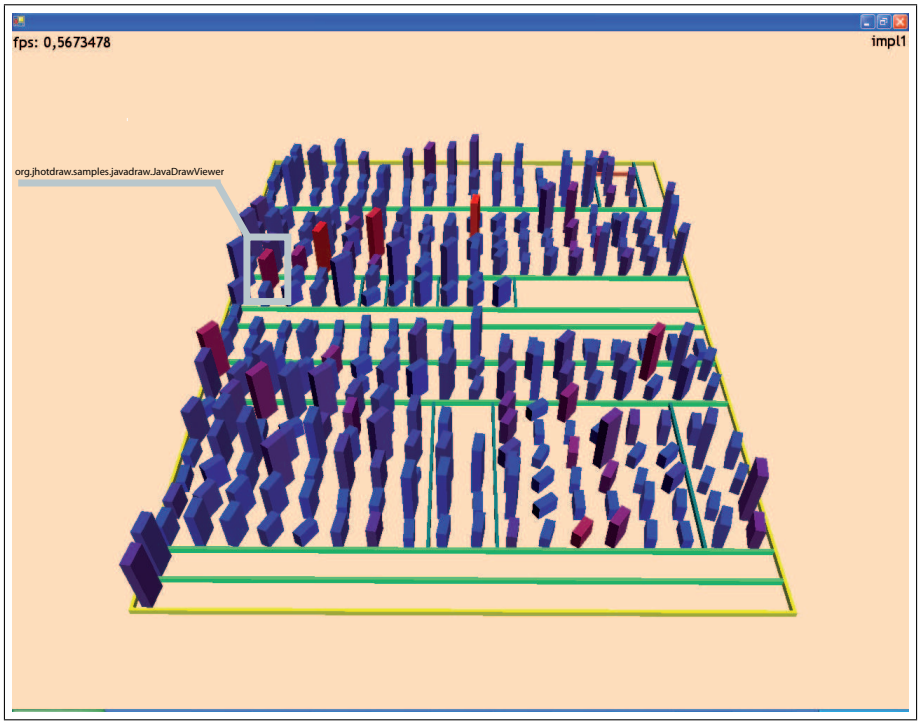

FIG. 4.3: La vue du système après introduction des aspects

4.3.3 Expérience

Nous avons demandé à trois utilisateurs du logiciel VERSO de se soumettre à une expérience afin de savoir si l'impact de la POA sur la programmation orientée objet pouvait être évalué.

Protocole. Nous fournissons aux évaluateurs trois fichiers XML et un questionnaire (*cf.* annexe VI). Les trois fichiers représentent respectivement le coeur du programme AJHotDraw, la version orientée aspect de l'implémentation de certaines préoccupations transversales et les mêmes préoccupations dans leur version orientée objet. Les deux fichiers correspondant aux implémentations objet et aspect ont été rendus anonymes de façon à ne pas biaiser les évaluateurs.

Nous leur avons demandé, dans un premier temps, d'évaluer chacune des deux implémentations en les comparant avec le coeur du système. Cela consistait à don-

ner une note allant de 1 à 5 (*très mauvais* à *très bon*) pour sept caractéristiques de la qualité : réutilisabilité, fiabilité, maintenabilité, utilisabilité, évolutibilité, robustesse, et modularité [ISO9126, 1992]. Il leur était aussi possible de ne pas se prononcer en cochant la réponse *n/a* (non applicable). Les évaluateurs disposaient, pour cette expérience, des valeurs des dix métriques que nous avons étudiées et implémentées : DIT, NAD, NMD, NMO, NOC, NOP, NOD, SIX, et WMC. Nous leur avons demandé ensuite de comparer sur la même échelle les deux implémentations pour savoir laquelle était de meilleure qualité.

Enfin, ils ont dû préciser quelles métriques ils avaient utilisées pour évaluer chaque caractéristique, ainsi que les métriques dont ils auraient eu besoin pour affiner leur jugement. De plus, nous leur avons demandé si ce type d'évaluation leur semblait intéressant et adéquat.

Résultats. L'expérience a été menée sur le programme AJHotDraw et sa version purement orientée objet JHotDraw. C'est en effet le seul programme assez gros que nous possédons. Le tableau 4.2 présente les résultats obtenus quant à son l'évaluation. On peut remarquer que la verion orientée aspect du programme a été globalement mieux notée que la version orientée objet. Les évaluateurs ont pu traiter cinq caractéristiques sur sept. Ils ont tous les trois conclu qu'il n'était pas possible d'évaluer la fiabilité et la robustesse du programme avec les métriques à disposition. Selon eux, ils auraient besoin de métriques qualifiant le couplage et la cohésion, entre autres.

Le tableau 4.3 présente les métriques utilisées par les évaluateurs pour noter les différentes caractéristiques. On peut remarquer que les métriques DIT, WMC et NOC sont considérées comme les meilleurs indicateurs. Elles sont utilisées pour toutes les caractéristiques évaluables. Les métriques NMO et SIX semblent être de bons indicateurs d'appoint.

Caractéristiques	**Impl. POA**	**Impl. POO**
Réutilisabilité	*bonne*	*moyenne*
Fiabilité	*n/a*	*n/a*
Maintenabilité	*très bonne*	*moyenne*
Utilisabilité	*moyenne*	*moyenne*
Évolutibilité	*bonne*	*moyenne*
Robustesse	*n/a*	*n/a*
Modularité	*bonne*	*moyenne*

TAB. 4.2: Résultats de l'évaluation

	Métriques								
Caractéristiques	WMC	DIT	NOP	NOC	NOD	NAD	NMD	NMO	SIX
Réutilisabilité	√	√		√				√	√
Fiabilité									
Maintenabilité	√	√		√					√
Utilisabilité	√	√		√					
Évolutibilité	√	√		√					√
Robustesse									
Modularité	√	√		√					√

TAB. 4.3: Les métriques utilisées par les évaluateurs

4.3.4 Conclusion sur l'évaluation

Selon les résultats de notre expérience, la version orientée aspect du programme JHotDraw est de meilleure qualité. Ce résultat était prévisible car AJHotDraw est une restructuration de certaines préoccupations de JHotDraw avec la POA. Ces deux versions n'ont pas été implémentées en parallèle.

Ceci n'est cependant pas le résultat le plus important de notre expérience. En effet, nous cherchions à savoir si l'impact de la POA sur les programmes orientés objets pouvait être évalué en termes de qualité. Notre expérience démontre que cela est possible. Les experts ont été capables d'évaluer cinq des sept caractéristiques proposées en faisant une comparaison, basée sur les métriques extraites, du programme avant et après introduction des aspects. Ils ont aussi précisé que les deux caractéristiques restantes seraient évaluables avec l'apport de nouvelles métriques.

Ces résultats sont très encourageants. En obtenant des programmes orientés aspect plus complexes et mieux documentés, il sera possible de préciser ce type d'expérience. Nous pourrons demander aux experts d'évaluer la qualité de chaque entité du programme modifiée par la POA pour ensuite mettre en relation leur expertise avec les différents mécanismes de la POA. Ceci nous permettra de déterminer, pour chacun de ces mécanismes, son impact sur la qualité des programmes orientés objet.

CONCLUSION

Le travail présenté dans ce mémoire a pour but de découvrir si la modification d'un système orienté objet, par l'apport de la POA, peut être quantifiée puis évaluée en termes de qualité. Notre travail nous permet de répondre à cette problématique par l'affirmative. Nous avons en effet réussi à quantifier cet impact grâce aux métriques de classe.

La POA [Kiczales *et al.*, 1997] propose d'introduire le concept d'aspect dans le paradigme objet. Un aspect a pour fonction d'encapsuler de façon modulaire des préoccupations, telles que la journalisation ou la persistance, qui ne font pas partie de la logique métier des objets. Il peut ensuite injecter le code nécessaire à la préoccupation qu'il encapsule. Pour ce faire, il possède des mécanismes qui lui permettent de prendre le contrôle de l'exécution d'un programme orienté objet à tout moment, d'ajouter de nouvelles variables ou méthodes à une classe et même de modifier la structure de leur hiérarchie. C'est ce que nous avons appelé l'impact de la POA.

Pour capturer et quantifier cet impact, nous avons étudié ses répercussions au niveau des métriques de classe. En effet, si la POA peut modifier les programmes orientés objet, elle doit aussi avoir un impact sur les métriques de classe. Nous avons donc formalisé ces répercussions en redéfinissant les calculs des métriques de Chidamber et Kemerer [1994] et certaines de celles de Lorenz et Kidd [1994]. Notre méthode se base sur le formalisme utilisé par Chidamber et Kemerer et a été pensée de façon à être facilement réutilisable. De cette manière, il est possible de redéfinir de nouvelle métriques dans l'avenir.

Pour pouvoir tester nos calculs et essayer d'évaluer cet impact, nous avons implémenté une solution logicielle concrète. Cette architecture nous permet de nous affranchir de la phase de tissage en modélisant et en analysant directement le code source des programmes aspect. Nous avons utilisé et étendu le métamodèle PADL

[Guéhéneuc, 2003] pour pouvoir créer notre modèle. L'extraction des métriques sur le modèle obtenu a pu se faire grâce au module POM [Guéhéneuc *et al.*, 2004], dont nous avons modifié les méthodes de calcul grâce à la POA. Enfin, nous avons développé une étape de présentation des résultats permettant d'obtenir les métriques au format XML. Concrètement, notre application isole le code source objet du programme et extrait un premier jeu de métriques de façon classique. Il est analysé une seconde fois mais en tenant compte de l'introduction des aspects. Ce second jeu de métriques reflète l'impact de la POA.

Le travail précédent nous a permis de vérifier, en menant une expérience, que l'impact de la POA était évaluable en termes de qualité. La relative jeunesse de ce paradigme nous a empêché de réunir un corpus de test conséquent. Ce manque d'exemple a rendu les techniques d'apprentissage de règles pour l'évaluation des logiciels inapplicables. Nous avons donc mis en place une expérience basée sur la visualisation. À l'aide du logiciel VERSO [Langelier *et al.*, 2005], des évaluateurs ont pu comparer visuellement les deux jeux de métriques d'un même programme, avant et après introduction des aspects. Ils ont alors jugé de la qualité de l'impact de la POA avec ces seules données. De la même manière, il est possible de comparer la qualité d'un programme orienté objet par rapport à la refactorisation de certaines de ses préoccupations grâce aux aspects.

Nous avons donc établi et démontré que la modification d'un programme orienté objet par la POA était quantifiable et évaluable en termes de qualité. Il serait intéressant de combiner maintenant notre approche avec celle de Sant'Anna [2005] ou Zhao [2002b] afin d'obtenir une évaluation complète de la qualité des programmes orientés aspect.

Contribution

Nous avons fait une revue de la littérature concernant la qualité et la programmation orientée aspect. Nous avons proposé une étude formelle de impact de la POA sur les métriques de classe. Nous offrons aussi une application concrète de notre travail. Cette architecture couvre tout le processus décrit, de la modélisation des données jusqu'à la présentation des résultats, en passant par l'extraction des métriques et est, de plus, facilement extensible. Enfin, nous avons mis en place une expérience permettant d'évaluer la qualité des programmes orientés aspect et de les comparer avec leur version orientée objet.

Travaux futurs

La précision du modèle utilisé devra être augmentée de façon à pouvoir instrumenter le code des méthodes et celui des greffons. Il sera alors possible de redéfinir de nouvelles métriques de classe en utilisant le même formalisme et les implémenter dans notre application. L'expérience proposée pourra être répétée et affinée dans le but de documenter les programmes évalués. Enfin, il faudra modifier l'application implantée pour supporter la version 1.5 d'AspectJ et les résultats présentés dans ce mémoire pourront être portés vers d'autres plateformes et d'autres implémentations de la POA.

À plus long terme, il sera important de réunir un corpus conséquent de programmes orientés aspect bien documentés. Cela permettra de multiplier les expériences et d'utiliser les techniques d'apprentissage automatique de règles. Il sera aussi intéressant de combiner les approches centrées aspect de l'évaluation de la POA [Sant'Anna *et al.*, 2005; Zhao, 2002a] avec celle présentée dans ce mémoire. Enfin, notre application pourra être intégrée à des environnements tels qu'Eclipse pour servir d'outil d'aide au développement.

BIBLIOGRAPHIE

[Albin-Amiot, 2003]

Hervé Albin-Amiot. *Idiomes et patterns Java : application à la synthèse de code et à la détection*. Thèse de doctorat, École des Mines de Nantes et Université de Nantes, Février 2003.

[Allan *et al.*, 2005]

Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam et Julian Tibble. abc : The aspectbench compiler for aspectJ. *GPCE*, pages 10–16, 2005.

[Almaer, 2003]

Dion Almaer. Interview with Gregor Kiczales : Aspect Oriented Programming (AOP). Compte rendu d'entrevue, 2003.

[Antoniol et Guéhéneuc, 2005]

Giuliano Antoniol et Yann-Gaël Guéhéneuc. Feature identification : A novel approach and a case study. Tibor Gyimóthy et Vaclav Rajlich, éditeurs, *proceedings of the* 21^{st} *International Conference on Software Maintenance*. IEEE Computer Society Press, Septembre 2005.

[AspectJ, 2002]

AspectJ. The AspectJ project at Eclipse.org, Décembre 2002.

[Astudillo, 1997]

Hernán Astudillo. Maximizing object reuse with a biological metaphor. *Theory and Practice of Object Systems*, 3(4) : 235–251. John Wiley & Sons, Inc., 1997.

[Boehm *et al.*, 1978]

B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. Macleod et M. Merritt.

Characteristics of Software Quality. TRW Series of Software Technology. North-Holland, 1978.

[Bonér *et al.*, 2005]

Jonas Bonér, Alexandre Vasseur et Joakim Dahlstedt. JRockit JVM support for AOP, part 2. Rapport technique, BEA dev2dev, Août 2005.

[Chidamber et Kemerer, 1994]

Shyam R. Chidamber et Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6) : 476–493, Juin 1994.

[Clement *et al.*, 2003]

Andy Clement, Adrian Colyer et Mik Kersten. Aspect-oriented programming with AJDT. Jan Hannemann, Ruzanna Chitchyan et Awais Rashid, éditeurs, *Workshop on Analysis of Aspect-Oriented Software (ECOOP 2003)*, Juillet 2003.

[Delannoy, 2004]

Claude Delannoy. *Programmer en Java*. Eyrolles, 3^{eme} édition, 2004. ISBN : 2-212-11501-6.

[Douence et Jussien, 2002]

Rémi Douence et Narendra Jussien. Nonintrusive constraint solver enhancements. Yvonne Coady, Eric Eide, David H. Lorenz, Mira Mezini, Klaus Ostermann et Roman Pichler, éditeurs, *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, Mars 2002.

[Dufour *et al.*, 2004]

Bruno Dufour, Christopher Goard, Laurie J. Hendren, Oege de Moor, Ganesh Sittampalam et Clark Verbrugge. Measuring the dynamic behaviour of aspectJ programs. *OOPSLA*, pages 150–169, 2004.

[Dupin et Valein, 1993]

Jean-Claude Dupin et Jean-Luc Valein. *Initiation au raisonnement mathéma-*

tique : logique et théorie des ensembles. Collection Flash U. A. Colin, 1993. ISBN : 2-200-21376-X.

[Fenton et Pfleeger, 1996]

Norman Fenton et Shari Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2^{eme} édition, 1996. ISBN : 1-85032-275-9.

[Flores et Robidoux, 2004]

Ward Flores et Sébastien Robidoux. C++ Parser for PADL. Rapport technique, Diro, Université de Montréal, Octobre 2004.

[Gamma, 1995]

Erich Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley professional computing series. Addison-Wesley, Janvier 1995. ISBN : 0-201-63361-2.

[Gélinas *et al.*, 2005]

Jean-François Gélinas, Linda Badri et Mourad Badri. Measuring cohesion in aspect-oriented systems. *IASTED Conf. on Software Engineering*, pages 343–348, 2005.

[Gibbs et Coady, 2004]

Celina Gibbs et Yvonne Coady. Garbage collection in Jikes : Could dynamic aspects add value ? Robert Filman, Michael Haupt, Katharina Mehner et Mira Mezini, éditeurs, *DAW : Dynamic Aspects Workshop*, pages 56–63, Mars 2004.

[Gradecki et Lesiecki, 2003]

Joseph D. Gradecki et Nicholas Lesiecki. *Mastering AspectJ : Aspect-Oriented Programming in Java*. John Wiley and Sons, Mars 2003. ISBN : 0471431044.

[Guéhéneuc *et al.*, 2004]

Yann-Gaël Guéhéneuc, Houari Sahraoui et Farouk Zaidi. Fingerprinting design patterns. Eleni Stroulia et Andrea de Lucia, éditeurs, *proceedings of the* 11^{th}

Working Conference on Reverse Engineering, pages 172–181. IEEE Computer Society Press, Novembre 2004.

[Guéhéneuc, 2003]

Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception.* Thèse de doctorat, École des Mines de Nantes et Université de Nantes, Juin 2003.

[Guyomarc'h et Guéhéneuc, 2005]

Jean-Yves Guyomarc'h et Yann-Gaël Guéhéneuc. On the impact of aspect-oriented programming on object-oriented metrics. Fernando Brito e Abreu, Coral Calero, Michele Lanza, Geert Poels et Houari A. Sahraoui, éditeurs, *proceedings of the* 9^{th} *ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 42–47. Springer-Verlag, Juillet 2005.

[Hanenberg, 2002]

S. Hanenberg. A proposal for classifying tangled code. Pascal Costanza, Günter Kniesel, Katharina Mehner, Elke Pulvermüller et Andreas Speck, éditeurs, *Second Workshop on Aspect-Oriented Software Development of the German Information Society*. Institut für Informatik III, Universität Bonn, Février 2002.

[Highley *et al.*, 1999]

T. J. Highley, Michael Lack, Perry Myers et Professor Paul Reynolds. Aspect oriented programming : A critical analysis of a new programming paradigm. Rapport technique, Mars 1999.

[ISO9126, 1992]

ISO9126. *ISO9126 Information Technology – Software Product Evaluation – Quality characteristics and guidelines for their use.* International Organization for Standardization, Geneva, 1992.

[Johnson et Shneiderman, 1991]

B. Johnson et Ben Shneiderman. Tree maps : A space-filling approach to the

visualization of hierarchical information structures. *IEEE Visualization*, pages 284–291, 1991.

[Khosravi, 2005]

Khashayar Khosravi. Qualitative quality model considering program architectures. Thèse de maîtrise, Université de Montréal, Août 2005.

[Kiczales *et al.*, 1997]

Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier et John Irwin. Aspect-oriented programming. Mehmet Akşit et Satoshi Matsuoka, éditeurs, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, volume 1241 de *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, Juin 1997.

[Kiczales *et al.*, 2001a]

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm et W. G. Griswold. An overview of AspectJ. J. L. Knudsen, éditeur, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, Juin 2001. Springer-Verlag.

[Kiczales *et al.*, 2001b]

Gregor Kiczales, Tzilla Elrad, Mehmet Akşit, Karl Lieberherr et Harold Ossher. Discussing aspects of AOP. *Comm. ACM*, 44(10) : 33–38, Octobre 2001.

[Laddad, 2002]

Ramnivas Laddad. I want my AOP ! *JavaWorld magazine*, Janvier 2002.

[Langelier *et al.*, 2005]

Guillaume Langelier, Houari Sahraoui et Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. *proceedings of the* 20^{th} *IEEE/ACM international conference on Automated Software Engineering*, pages 214–223. ACM Press, 2005.

[Lopes et Ngo, 2004]

C. V. Lopes et T. C. Ngo. The aspect markup language and its support of aspect plugins. ISR Technical Report UCI-ISR-04-8, University of California, Irvine, 2004.

[Lopes, 1997]

Cristina Isabel Videira Lopes. *D :–a language framework for distributed programming : a thesis*. Thèse de doctorat, Northeastern University, 1997.

[Lorenz et Kidd, 1994]

Mark Lorenz et Jeff Kidd. *Object-Oriented Software Metrics : A Practical Guide*. Prentice-Hall, 1994. ISBN : 13-179292-X.

[McCall *et al.*, 1977]

James A. McCall, Paul K. Richards et Gene F.Walters. Factors in software quality, volume I : Concepts and definitions of software quality. Rapport technique RADC-TR-77-369, vol. I, Rome Air Development Center, Griffiss AFB, Rome, NY 13441-5700, Juillet 1977.

[Mendhekar *et al.*, 1997]

A. Mendhekar, G. Kiczales et J. Lamping. RG : A case-study for aspect-oriented programming. Rapport technique SPL97-009 P9710044, Palo Alto, CA, USA, Février 1997.

[Meyer, 1997]

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, 2^{eme} édition, 1997. ISBN : 0-13-629155-4.

[Miles, 2004]

Russell Miles. *AspectJ Cookbook*. O'Reilly, 1^{ere} édition, Décembre 2004.

[OMG, 2003]

OMG. UML 2.0 object constraint language (OCL) specification, 2003.

[OMG, 2004]

OMG. *UML 2.0 Superstructure Specification*. Object Management Group, Octobre 2004.

[Parnas, 1972]

David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) : 1053–1058, Décembre 1972.

[Sant'Anna *et al.*, 2003]

Cláudio Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena et Arndt von Staa. On the reuse and maintenance of aspect-oriented software : An assessment framework. *XVII Brazilian Symposium on Software Engineering*, Octobre 2003.

[Sant'Anna *et al.*, 2005]

Cláudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Uirá Kulesza et Carlos Lucena. Assessing Aspect-Oriented Artifacts : Towards a Tool-Supprorted Quantitative Method. Fernando Brito e Abreu, Coral Calero, Michele Lanza, Geert Poels et Houari A. Sahraoui, éditeurs, *proceedings of the* 9^{th} *ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 58–69. Springer-Verlag, Juillet 2005.

[Schach, 2002]

Stephen R. Schach. *Object-oriented and classical software engineering*. McGraw-hill, 5^{eme} édition, 2002.

[Smacchia et Vaucouleur, 2003]

Patrick Smacchia et Sébastien Vaucouleur. Dossier spécial : AOP, Intérêts et Usages. Ressource électronique, Juillet 2003.

[Vachon et Mostefaoui, 2004]

Julie Vachon et Farida Mostefaoui. Achieving supplementary requirements using

aspect-oriented development. *proceedings of the* 7^{th} *International Conference on Enterprise Information Systems (ICEIS 2005)*, pages 584–587, 2004.

[Vachon et Mostefaoui, 2005]

Julie Vachon et Farida Mostefaoui. Modélisation et vérificationformelle de la composition des aspects. Lionel Seinturier, éditeur, *proceedings of the* 2^{nd} *French Workshop on Aspect-Oriented Software Development (JFDLPA 2005)*, Septembre 2005.

[Zaidi, 2004]

Farouk Zaidi. POM, un outil pour le calcul des métriques de qualité, Juillet 2004. Présentation PowerPoint.

[Zakaria et Hosny, 2003]

Aida Atef Zakaria et Hoda Hosny. Metrics for aspect-oriented software design. Omar Aldawud, Mohamed Kandé, Grady Booch, Bill Harrison et Dominik Stein, éditeurs, *Third International Workshop on Aspect Oriented Modeling*, Mars 2003.

[Zhao et Xu, 2004]

Jianjun Zhao et Baowen Xu. Measuring aspect cohesion. *FASE*, pages 54–68, 2004.

[Zhao, 2002a]

J. Zhao. Towards a metrics suite for aspect-oriented software. Rapport technique SE-136-25, Information Processing Society of Japan (IPSJ), Mars 2002.

[Zhao, 2002b]

Jianjun Zhao. Change impact analysis for aspect-oriented software evolution. *Proceedings of the Workshop on Principles of Software Evolution*, pages 108–112. ACM Press, 2002. ISBN : 1-58113-545-9.

[Zhao, 2004]

Jianjun Zhao. Measuring coupling in aspect-oriented systems, Août 2004.

Annexe I

AspectJ – points de coupure

Cette annexe présente les différents points de coupure implémentés par AspectJ. Ces données ont été reproduites à partir des travaux de Laddad [Laddad, 2002].

Pointcut	**Description**
call(public void MyClass.myMethod(String))	Call to myMethod() in MyClass taking a String argument, returning void, and with public access
call(void MyClass.myMethod(..))	Call to myMethod() in MyClass taking any arguments, with void return type, and any access modifiers
call(* MyClass.myMethod(..))	Call to myMethod() in MyClass taking any arguments returning any type
call(* MyClass.myMethod*(..))	Call to any method with name starting in "myMethod" in MyClass
call(* MyClass.myMethod*(String,..))	Call to any method with name starting in "myMethod" in MyClass and the first argument is of String type
call(* *.myMethod(..))	Call to myMethod() in any class in default package
call(MyClass.new())	Call to any MyClass' constructor taking no arguments
call(MyClass.new(..))	Call to any MyClass' constructor with any arguments
call(MyClass+.new(..))	Call to any MyClass or its subclass's constructor. (Subclass indicated by use of '+' wildcard)
call(public * com.mycompany..*.*(..))	All public methods in all classes in any package with com.mycompany the root package

Tab. I.1: Point de coupure – Appel de méthodes et de constructeurs

Pointcut	**Description**
execution(public void MyClass.myMethod(String))	Execution of myMethod() in MyClass taking a String argument, returning void, and with public access
execution(void MyClass.myMethod(..))	Execution of myMethod() in MyClass taking any arguments, with void return type, and any access modifiers
execution(* MyClass.myMethod(..))	Execution of myMethod() in MyClass taking any arguments returning any type
execution(* MyClass.myMethod*(..))	Execution of any method with name starting in "myMethod" in MyClass
execution(* MyClass.myMethod*(String,..))	Execution of any method with name starting in "myMethod" in MyClass and the first argument is of String type
execution(* *.myMethod(..))	Execution of myMethod() in any class in default package
execution(MyClass.new())	Execution of any MyClass' constructor taking no arguments
execution(MyClass.new(..))	Execution of any MyClass' constructor with any arguments
execution(MyClass+.new(..))	Execution of any MyClass or its subclass's constructor. (Subclass indicated by use of '+' wildcard)
execution(public * com.mycompany..*.*(..))	All public methods in all classes in any package with com.mycompany the root package

Tab. I.2: Point de coupure – Exécution de méthodes et de constructeurs

Pointcut	**Description**
get(PrintStream System.out)	Execution of read-access to field out of type PrintStream in System class
set(int MyClass.x)	Execution of write-access to field x of type int in MyClass

Tab. I.3: Point de coupure – Accès aux variables

Pointcut	**Description**
handler(RemoteException)	Execution of catch-block handling RemoteException type
handler(IOException+)	Execution of catch-block handling IOException or its subclasses
handler(CreditCard*)	Execution of catch-block handling exception types with names that start with CreditCard

Tab. I.4: Point de coupure – Gestionnaire d'exceptions

Pointcut	**Description**
Staticinitialization(MyClass)	Execution of static block of MyClass
Staticinitialization(MyClass+)	Execution of static block of MyClass or its subclasses

Tab. I.5: Point de coupure – Initialisation de classe

Pointcut	**Description**
within(MyClass)	Any pointcut inside MyClass's lexical scope
within(MyClass*)	Any pointcut inside lexical scope of classes with a name that starts with "MyClass"
withincode(* MyClass.myMethod(..))	Any pointcut inside lexical scope of any myMethod() of MyClass

Tab. I.6: Point de coupure – Basés sur la structure lexicale

Pointcut	**Description**
cflow(call(* MyClass.myMethod(..))	All the joinpoints in control flow of call to any myMethod() in MyClass including call to the specified method itself
cflowbelow(call(* MyClass.myMethod(..))	All the joinpoints in control flow of call to any myMethod() in MyClass excluding call to the specified method itself

Tab. I.7: Point de coupure – Flot de contrôle

Pointcut	**Description**
this(JComponent+)	All the joinpoints where this is instanceof JComponent
target(MyClass)	All the joinpoints where the object on which the method is called is of type MyClass
args(String,..,int)	All the joinpoints where the first argument is of String type and the last argument is of int type
args(RemoteException)	All the joinpoints where the type of argument or exception handler type is RemoteException

Tab. I.8: Point de coupure – Contexte et arguments

Pointcut	**Description**
if(EventQueue.isDispatchThread())	All the joinpoints where EventQueue.isDispatchThread() evaluates to true

Tab. I.9: Point de coupure – Structure conditionnelle

Annexe II

Théorie des ensembles

Pour les besoins des démonstrations et équations du chapitre 2, nous avons appliqué la théorie des ensembles [Dupin et Valein, 1993, p. 53] aux paradigmes orientés objet et aspect. Cette annexe présente les ensembles et propriétés qui ont été définis afin de mener à bien nos travaux.

II.1 Rappels

II.1.1 Définitions

Ensemble. Groupement ou collection d'objets appelés *éléments*. Les ensembles sont désignés par des lettres majuscules ($E, F, G, ..., \Omega, \Gamma, ...$).

Élément. Objet faisant partie d'un ensemble et possédant certaines *propriétés*. Les éléments d'un ensemble sont désignés par des lettres minuscules (x, y, z, ..., α, β, δ, ...).

Propriété. Une propriété relative à un ensemble E est une assertion qui a un sens pour chaque élément de E. Elle est éventuellement vraie pour certains éléments de E et fausse pour les autres.

Appartenance. L'appartenance d'un élément x à un ensemble E est notée $x \in E$.

Compréhension. Un sous-ensemble (ou *partie*) A de l'ensemble E est défini par compréhension lorsqu'il est précisé par le biais d'une propriété caractéristique à ce sous-ensemble.

Extension. Un sous-ensemble A de l'ensemble E peut être défini par l'énumération de tous les éléments le composant. On le dit alors défini par extension.

Singleton. Partie ne contenant qu'un seul élément.

Paire. Partie constituée seulement de deux éléments.

Partie vide. Partie ne contenant aucun élément. Elle est notée $\oslash$.

Inclusion. Pour A et B deux parties de E, on dit que A est contenu (inclus) dans B si tous les éléments de A appartiennent à B. Cette propriété est notée $A \subset B$. A et B appartenant à E, on a aussi $A \subset B \subset E$.

II.1.2 Inclusion, réunion et appartenance.

Soit A, B et C des parties quelconques d'un même ensemble E. Les équations (II.1) à (II.11) présentent les principales propriétés de l'inclusions, de la réunion et de l'appartenance[1].

[1]Propriétés tirées de [Dupin et Valein, 1993, p. 57].

$$A \cap \overline{A} = \oslash \tag{II.1}$$

$$A \cup \overline{A} = E \tag{II.2}$$

$$\overline{A \cap B} = \overline{A} \cup \overline{B} \tag{II.3}$$

$$\overline{A \cup B} = \overline{A} \cap \overline{B} \tag{II.4}$$

$$A \cap A = A;\ A \cup A = A \quad \text{(idempotence de } \cap \text{ et de } \cup) \tag{II.5}$$

$$A \cap B = B \cap A;\ A \cup B = B \cup A \quad \text{(commutativité de } \cap \text{ et de } \cup) \tag{II.6}$$

$$A \cap (B \cap C) = (A \cap B) \cap C \quad \text{(associativité de } \cap) \tag{II.7}$$

$$A \cup (B \cup C) = (A \cup B) \cup C \quad \text{(associativité de } \cup)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C) \tag{II.8}$$

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C) \quad \text{(distributivité)}$$

$$A \subset B \Leftrightarrow \overline{B} \subset \overline{A} \tag{II.9}$$

$$A \cup B = A \Leftrightarrow B \subset A;\ A \cap B = A \Leftrightarrow A \subset B \tag{II.10}$$

$$(A \subset B \ et\ B \subset C) \Rightarrow A \subset C \tag{II.11}$$

Nos définitions

II.1.3 Ensembles

$\mathbf{MD_c}$ Ensemble des méthodes déclarées par la classe c.

$\mathbf{ITMD_c}$ Ensemble des méthodes introduites par des aspects sur la classe c.

ITDP Ensemble des classes, d'un même système, dont la hiérarchie a été modifiée.

ITDP$_c$ Ensemble des classes du programme qui se sont vues attribuer la classe c comme parent via le mécanisme d'introduction.

PARENTS$_c$ Ensemble des classes qui sont parentes immédiates de la classe c.

SCI$_c$ Ensemble des sous-classes immédiates de c.

A Ensemble des classes d'un même programme.

O$_c$ Ensemble des classe $o_1, o_2 ... o_n$ tel que $O_c = \{A \setminus \{c\}\}$.

ITMA$_c$ Ensemble des appels de méthode faits par l'entremise de méthodes introduites sur la classe c.

GMA$_c$ Ensemble des appels de méthode faits par l'entremise de greffons associés grâce à des points de coupure à la classe c.

MCALL$_i$ Ensemble des appels de méthode faits par une méthode m_i.

ITMCALL$_j$ Ensemble des appels de méthode faits par une méthode introduite m_j.

AMCALL$_k$ Ensemble des appels de méthode faits par un greffon a_k.

ADVICE$_c$ Ensemble des greffons associés à la classe c.

I$_j$ Ensemble des variables d'instances utilisées par la méthode m_j.

P$_c$ Ensemble des paires de méthodes de la classe c qui ne partagent pas l'utilisation d'une même variable d'instance.

$\mathbf{Q_c}$ Ensemble des paires de méthodes de la classe c qui utilisent au moins une même variable d'instance.

$\mathbf{AD_c}$ Ensemble des attributs déclarés par la classe c.

$\mathbf{ITAD_c}$ Ensemble des attributs introduits par différents aspects au niveau de la classe c.

$\mathbf{MH_c}$ Ensemble des méthodes héritées par c.

$\mathbf{ITMH_c}$ Ensemble des méthodes introduites dans l'arbre d'héritage de c, soit par introduction de méthodes sur les ancêtres de c, soit par introduction de nouveaux ancêtres dans l'arbre d'héritage de c.

Annexe III

Profil UML – AspectUML

Afin de présenter les différents exemples de l'impact de la POA sur les métriques de classes (*cf.* chapitre 2) sous forme de diagrammes UML [OMG, 2004], nous utilisons le profil UML *AspectUML* développé par Vachon et Mostefaoui [2004; 2005].

III.1 Le profil AspectUML

AspectUML est un profil UML qui permet de décrire certaines abstractions du paradigme aspect en les intégrant à deux modèles du langage UML : diagramme de cas d'utilisation et diagramme de classes. De plus, le profil *AspectUML* spécifie certaines contraintes (en utilisant un formalisme proche d'OCL [OMG, 2003]) propres au paradigme aspect de façon à enrichir les diagrammes.

Nous nous intéressons au diagramme de classes. Dans un diagramme de classes respectant les spécifications du profil *AspectUML*, un aspect est représenté par une classe portant le stéréotype « Aspect ». Un point de coupure est modélisé par une interface portant le stéréotype « Pointcut ». Un aspect doit implémenter l'interface correspondante au point de coupure qu'il utilise. Afin de lier un « Pointcut » avec les points de jointure correspondants, une relation de dépendance « crosscuts » est utilisée. La figure III.1 présente un exemple de diagramme de classes utilisant la spécification *AspectUML* pour une application appelée TELECOM : deux aspects, `Timing` et `Billing`, ainsi que deux points de coupure, `OpComplete` et `OpDrop`. Chaque point de coupure a une relation de dépendance de type « crosscut-call », spécifiant les points de jointure potentiels. Le profil AspectUML ne décrivant pas le mécanisme d'introduction propre au paradigme aspect, nous avons du l'ajouter pour nos besoins.

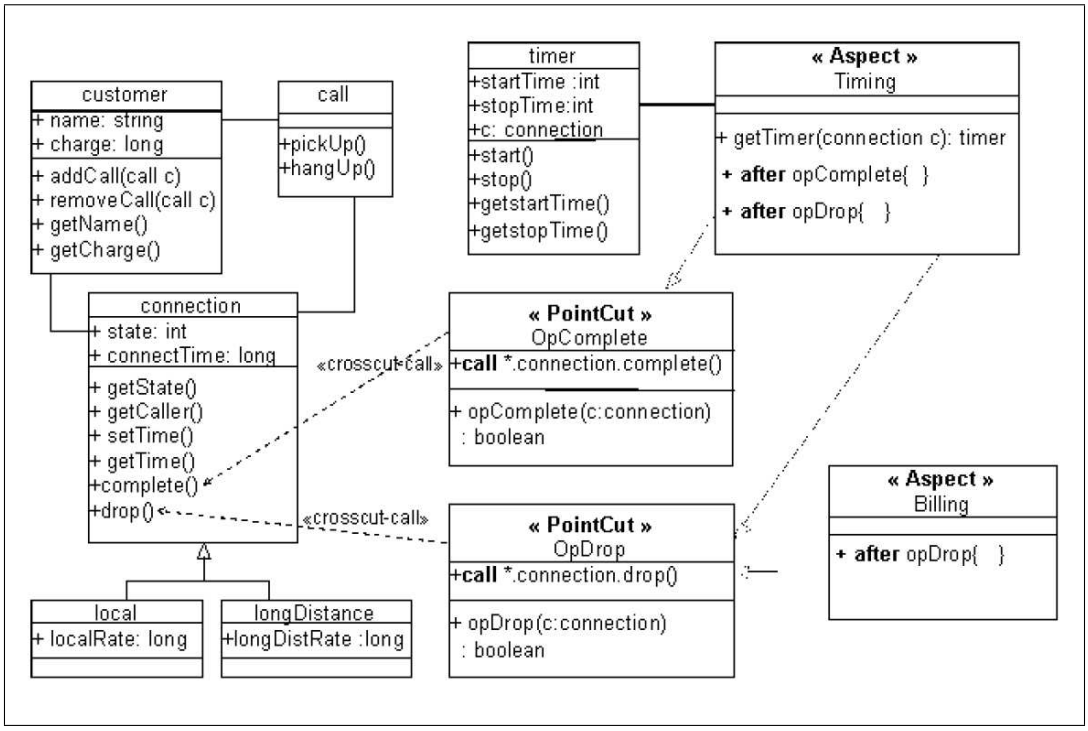

FIG. III.1: Diagramme de classe pour l'application TELECOM

III.2 Extension du profil AspectUML

Nous modélisons le mécanisme d'introduction par une relation de dépendance. Cette relation de dépendance a un identificateur unique de forme suivante :<< $\{nom\_aspect\} - IT\{num\}$ >>. $\{nom\_aspect\}$ est le nom de l'aspect responsable de l'introduction et origine de la relation de dépendance, IT pour *Inter-type* et $\{num\}$ est un numéro permettant de rendre l'identificateur unique.

Les champs et les méthodes introduits seront déclarés de façon classique au niveau de l'aspect responsable de l'introduction, mais leur déclaration sera précédée de la partie $IT\{num\}$ du nom de la relation de dépendance correspondante. De cette manière, il sera facile d'associer le champ ou la méthode introduit sur l'entité cible.

De la même façon, le mécanisme d'introduction de type *declare parents* sera identifié dans l'aspect par le numéro de la relation de dépendance, suivi des mots

clés `extends` ou `implements` et terminé par le nom de l'entité à étendre/implémenter.

La figure III.2 présente une extension de l'exemple III.1. Seuls les éléments modifiés apparaissent. On peut remarquer que l'aspect `Timing` déclare trois introductions sur la classe `Customer` : un champ `dirty` de type `boolean`, une méthode `write(Output)`, et l'implémentation de la classe `Serializable`. De plus il déclare que la classe `Connection` doit elle aussi implémenter l'interface `Serializable`.

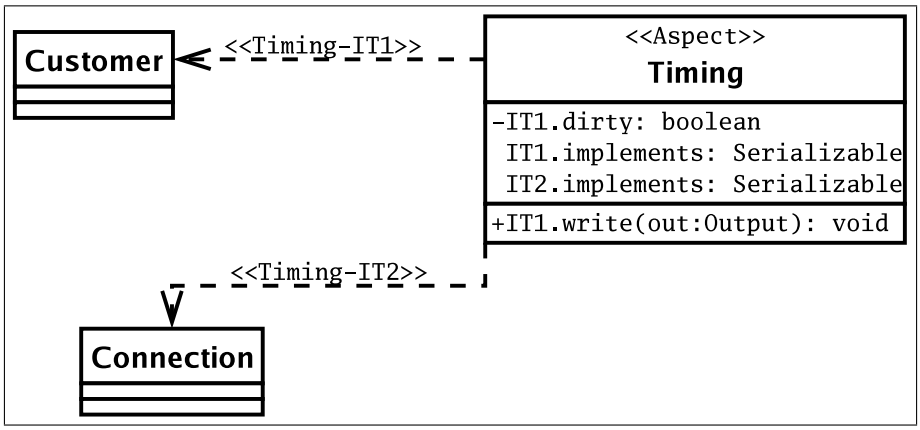

FIG. III.2: Le mécanisme d'introduction avec AspectUML

Ces extensions au profil *AspectUML* ont été proposées aux auteurs et sont présentement à l'étude.

Annexe IV

Compagnon de lecture

Vous trouverez, sur ce site, une version électronique de ce mémoire au format PDF, les résultats de nos expériences, un résumé des tests unitaires, une bibliographie interactive et la méthode pour accéder aux sources du projet et des exemples utilisés.

IV.1 Accès en ligne

Le site est accessible à l'adresse :
`http://www-etud.iro.umontreal.ca/~ptidej/guyomarj/maitrise`

IV.2 Obtenir une copie

Si vous aviez un quelconque problème pour accéder au site en ligne, vous pouvez nous contacter par courrier électronique à l'adresse suivante :

`jy.guyomarch@gmail.com`

Vous pouvez également obtenir une version du site sur disque compact en écrivant à :

```
Yann-Gaël Guéhéneuc pour Jean-Yves Guyomarc'h
DIRO, Université de Montréal
C.P. 6128, succursale Centre-Ville
Montréal, QC, H3C 3J7
Canada
```

Annexe V

POMXML – schema XSD

Cette annexe présente le Schema XSD développé afin d'utiliser le format d'entrée de l'outil VERSO [Langelier *et al.*, 2005].

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <!--W3C Schema by Jean-Yves Guyomarch-->
3 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
4  <!-- Global Complex Types -->
5  <xs:complexType name="AssociationsType">
6   <xs:sequence>
7    <xs:element name="asso" type="assoType" minOccurs="0" maxOccurs="unbounded"/>
8   </xs:sequence>
9  </xs:complexType>
10 <xs:complexType name="ChildrenType">
11
12  <xs:sequence>
13   <xs:element name="Child" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
14  </xs:sequence>
15 </xs:complexType>
16 <xs:complexType name="ImplementedInterfacesType">
17  <xs:sequence>
18   <xs:element name="Interface" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
19  </xs:sequence>
20 </xs:complexType>
21
22 <xs:complexType name="ObjRelStructureType">
23  <xs:sequence>
24   <xs:element name="ObjectProperties" type="ObjectPropertiesType"/>
25   <!-- TODO: RelationProperties -->
26   <xs:element name="RelationProperties" type="xs:string"/>
27  </xs:sequence>
28 </xs:complexType>
29 <xs:complexType name="ObjectType">
30  <xs:sequence>
31
32   <xs:element name="Name" type="xs:string"/>
33   <xs:element name="Type" type="xs:string"/>
34   <xs:element name="ImplementedInterfaces" type="ImplementedInterfacesType"/>
35   <xs:element name="Parents" type="ParentsType"/>
36   <xs:element name="Children" type="ChildrenType"/>
37   <xs:element name="UmlTargets" type="UmlTargetsType"/>
38   <xs:element name="ObjectProperties" type="ObjectValueType"/>
39  </xs:sequence>
40 </xs:complexType>
41
42 <xs:complexType name="ObjectValueType">
43  <xs:sequence>
44   <xs:element name="Prop" type="PropValueType" maxOccurs="unbounded"/>
45  </xs:sequence>
46 </xs:complexType>
```

```
47 <xs:complexType name="ObjectPropertiesType">
48  <xs:sequence>
49   <xs:element name="Prop" type="PropType" maxOccurs="unbounded"/>
50  </xs:sequence>
51
52 </xs:complexType>
53 <xs:complexType name="ObjectsType">
54  <xs:sequence>
55   <xs:element name="Object" type="ObjectType" minOccurs="0" maxOccurs="unbounded"/>
56  </xs:sequence>
57 </xs:complexType>
58 <xs:complexType name="ParentsType">
59  <xs:sequence>
60   <xs:element name="Parent" type="xs:string" maxOccurs="unbounded"/>
61
62  </xs:sequence>
63 </xs:complexType>
64 <xs:complexType name="PropType">
65  <xs:sequence>
66   <xs:element name="PropName" type="xs:string"/>
67   <xs:element name="ProType" minOccurs="0">
68    <xs:simpleType>
69     <xs:restriction base="xs:string">
70      <xs:enumeration value="double"/>
71
72      <xs:enumeration value="int"/>
73      <xs:enumeration value="float"/>
74      <xs:enumeration value="long"/>
75     </xs:restriction>
76    </xs:simpleType>
77   </xs:element>
78   <xs:element name="ProComments" type="xs:string" minOccurs="0"/>
79   <xs:element name="PropMin" type="xs:double" minOccurs="0"/>
80   <xs:element name="PropMax" type="xs:double" minOccurs="0"/>
81
82  </xs:sequence>
83 </xs:complexType>
84 <xs:complexType name="PropValueType">
85  <xs:sequence>
86   <xs:element name="PropName" type="xs:string"/>
87   <xs:element name="ProValue" type="xs:double"/>
88  </xs:sequence>
89 </xs:complexType>
90 <xs:complexType name="UmlTargetsType" mixed="true">
91
92  <xs:choice minOccurs="0" maxOccurs="unbounded">
93   <xs:element name="Target" type="xs:string"/>
94  </xs:choice>
95 </xs:complexType>
96 <xs:complexType name="assoType">
97  <xs:sequence>
98   <xs:element name="metric" type="xs:string"/>
99   <xs:element name="graph" type="xs:string"/>
100  </xs:sequence>
101
102 </xs:complexType>
103
```

```
104 <xs:complexType name="timestampType">
105  <xs:sequence>
106   <xs:element name="year" type="xs:int"/>
107   <xs:element name="month" type="xs:int"/>
108   <xs:element name="day" type="xs:int"/>
109   <xs:element name="hour" type="xs:int"/>
110   <xs:element name="minute" type="xs:int"/>
111   <xs:element name="second" type="xs:int"/>
112
113  </xs:sequence>
114 </xs:complexType>
115 <!-- ROOT -->
116 <xs:element name="DssDocument">
117  <xs:complexType>
118   <xs:sequence>
119    <xs:element name="title" type="xs:string"/>
120    <xs:element name="timestamp" type="timestampType"/>
121    <xs:element name="Associations" type="AssociationsType"/>
122
123    <xs:element name="ObjRelStructure" type="ObjRelStructureType"/>
124    <xs:element name="Objects" type="ObjectsType"/>
125   </xs:sequence>
126  </xs:complexType>
127 </xs:element>
128 </xs:schema>
```

Code V.1: Schema XSD de POMXML

Annexe VI

Expérience – questionnaire

Pouvez-vous évaluer les sept (7) qualités suivantes sur une échelle de 1 à 5 et ce, pour chacune des deux (2) implémentations proposées ? Vous pouvez ajouter des qualités à évaluer, au besoin.

Échelle d'évaluation : **1** = très mauvais, **2** = mauvais, **3** = moyen, **4** = bon, **5** = très bon, **n/a** = non applicable

Qualité	**Implémentation 1**						**Implémentations 2**					
	1	**2**	**3**	**4**	**5**	**n/a**	**1**	**2**	**3**	**4**	**5**	**n/a**
Réutilisabilité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Fiabilité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Maintenabilité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Utilisabilité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Évolutibilité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Robustesse	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Modularité	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

Pouvez-vous comparer les deux (2) implémentations en utilisant la même échelle ?

	1	2	3	4	5
Implémentation 1 :	☐	☐	☐	☐	☐
Implémentation 2 :	☐	☐	☐	☐	☐

Justifier votre réponse.

Lorsque possible, indiquez les métriques que vous avez utilisées afin d'évaluer les différentes qualités. Sinon, indiquez les métriques dont vous auriez eu besoin.

Qualité	**Métriques utilisées**	**Métriques désirées**
Réutilisabilité		
Fiabilité		
Maintenabilité		
Utilisabilité		
Évolutibilité		
Robustesse		
Modularité		

Pensez-vous que cette technique de comparaison visuelle utilisant les métriques est bien adaptée à l'évaluation de la qualité des logiciels ?

Oui ☐ Non ☐

Précisez.

Commentaires.