UNIVERSITÉ DU QUÉBEC

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

COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN MATHÉMATIQUES ET INFORMATIQUE APPLIQUÉES

PAR
FRANCIS TESSIER

ASSURANCE QUALITÉ ET DÉVELOPPEMENT DE LOGICIELS ORIENTÉS ASPECT : DÉTECTION DE CONFLITS ENTRE LES ASPECTS BASÉE SUR LES MODÈLES

JUIN 2005

Université du Québec à Trois-Rivières

Service de la bibliothèque

Avertissement

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

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

ASSURANCE QUALITÉ ET DÉVELOPPEMENT DE LOGICIELS ORIENTÉS ASPECTS : DÉTECTION DE CONFLITS ENTRE LES ASPECTS BASÉE SUR LES MODÈLES

Sommaire

La POA (Programmation Orientée Aspect) est un nouveau paradigme du génie logiciel qui suscite de plus en plus l'intérêt des chercheurs. Il s'agit d'une nouvelle approche de conception qui porte essentiellement sur la résolution de la dispersion des préoccupations et des spécifications transverses. Ces dernières sont représentées de façon modulaire dans des unités appelées *aspects*. Les nombreux mécanismes dont disposent les aspects leur permettent de se greffer à une multitude de classes et d'emplacements dans le programme. Cette souplesse peut engendrer de multiples conflits insoupçonnés. Ces conflits peuvent avoir des conséquences importantes en termes de qualité. Actuellement, il n'existe aucun outil permettant de détecter les conflits engendrés par l'insertion des aspects dans le code. Notre travail a porté sur le développement d'une méthode permettant la détection de conflits générés par les aspects.

Nous avons tenté de créer une approche simple et intuitive supportant les diverses implémentations et conceptualisations des aspects. Pour arriver à cet objectif, nous avons dû extraire l'essence des aspects pour obtenir les concepts de base. Pour supporter notre approche, nous avons dû émettre des hypothèses et proposer une technique de modélisation, car aucune de celles existantes n'a atteint un certain niveau de maturité. Tout cela nous a permis de définir une approche qui permet de couvrir toutes les phases de développement d'un logiciel, en particulier les premières. L'implémentation préliminaire (prototype) de notre détecteur de conflits nous indique que l'idée a du potentiel et permet d'assurer de façon précoce la qualité du développement. L'originalité de ce travail porte à la fois sur l'approche utilisée pour la détection de conflits entre les aspects ainsi que son application sur des modèles. Par contre, il est à noter qu'elle ne permet pas de résoudre totalement tous les conflits potentiels. Il faut une intervention humaine pour certains types de conflits. Le sujet est assez important et constitue un enjeu crucial pour la maturité du paradigme aspect.

**QUALITY ASSURANCE AND DEVELOPMENT
OF ASPECT ORIENTED SOFTWARE DEVELOPMENT:
A MODEL BASED DETECTION OF CONFLICTS BETWEEN ASPECTS.**

Abstract

AOP (Aspect Oriented Programming) is a new software engineering paradigm that gains in fame. It represents a new design approach that relates primarily to the resolution of concerns dispersion and crosscutting specifications. The crosscutting concerns are represented in a modular way in single units called aspects. The new constructs and mechanisms allow aspects to be weaved in several classes of a program. This flexibility can generate many unsuspected conflicts. These conflicts can have important consequences in terms of quality. Currently, there is no tool making it possible to detect the conflicts generated by the insertion of aspects in the code. Our work concerned the development of a method allowing detection of conflicts generated by the aspects.

We tried to create a simple and intuitive approach supporting the various implementations and conceptualizations of the aspects. To achieve this objective, we had to extract the information from the aspects to obtain the basic concepts. To support our approach, we had to put assumptions and to propose a technique of modeling, because none of those existing reached a certain level of maturity. All that enabled us to define an approach which makes it possible to cover all the phases of software development, in particular the first phases. The preliminary implementation (prototype) of our detector of conflicts indicates that the idea has potential and makes it possible to ensure, in an early way, software quality. The originality of this work concerns the approach used for the detection of conflicts between aspects as well as its application on models. Otherwise, it should be noted that it does not make it possible to solve all the potential conflicts completely. A human intervention is needed for certain types of conflicts. The subject is rather important and constitutes crucial task for the maturity of the aspect paradigm.

Remerciements

Je tiens à remercier les personnes suivantes :

Mes directeurs de recherche Mourad Badri et Linda Badri, professeurs au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières, pour leur grande patience, leur support constant, leurs conseils judicieux ainsi que pour leur dévouement de tous les instants. Grâce à leurs conseils éclairés et leurs révisions fréquentes de mes articles, j'ai pu soumettre des travaux de qualité. Ce fut un plaisir de travailler et d'apprendre avec eux. Merci pour tout !

M. François Lévesque, responsable du service des technopédagogies et des ressources informationnelles de l'École nationale de police du Québec. Grâce à lui, j'ai pu obtenir un stage très formateur et obtenir un emploi pendant deux ans. Cet emploi m'a permis de mettre en pratique mes connaissances et apprendre ce qu'est la dynamique du travail en équipe. Je le remercie chaleureusement pour son support indéfectible et la très grande latitude d'horaire qu'il m'a offert. Merci encore !

Mes professeurs du département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Merci !

Finalement, je remercie tous ceux qui m'ont aidé à réaliser mon rêve et ce mémoire. Je tiens à souligner l'appui constant de ma famille ainsi que celui de mes amis et collègues de travail et d'université. Merci !

Tables des matières

Liste des tableaux

Page

Liste des figures

Page

Liste des abréviations et des sigles

Acronyme	Signification
AJDE	AspectJ Development Environment
AJDT	AspectJ Development Tools
AOP	Aspect Oriented Programming
AOSD	Aspect Oriented Software Development
API	Application Programming Interface
AspectJ	Projet Aspect d'Eclipse
GUI	Graphical User Interface
JAC	Java Aspect Components
LGPL	Lesser General Public License
OMG	Object Management Group
PARC	Palo Alto Research Center (Xerox)
POA	Programmation Orientée Aspect
RAD	Rapid Application Development
SDK	Software Development Kit
UML	Unified Modeling Language
UMLAF	UML Aspect Factory

Sigle	Signification
$\subseteq$	Inclusion
$\forall$	Tout
$\wedge$	Et
*	N'importe lequel
$\cap$	Intersection
$\neg$	Non
A_i	Aspect
Arg_i	Argument
C_i	Classe
J_i	Point de jointure
O_i	Précédence
P_i	Point de coupure
R_{ac}	Relation aspect – classe
$S(R_{ac})$	Sémantique de la relation aspect – classe
T_{ac}	Table de connexion aspect – classe
TJ_i	Type de point de jointure
TO_i	Type d'opération (accès ou modification)
TP_i	Type de point de coupure

L'utilisation de l'anglais a été réduite au minimum. Cependant, plusieurs termes sont difficiles à traduire ou n'ont pas de traduction satisfaisante. Alors, le lecteur retrouvera parfois quelques termes en anglais.

Introduction

Actuellement, on observe de plus en plus l'émergence du paradigme de développement à base de composants. Depuis plusieurs années déjà, la complexité des défis et enjeux auxquels est confronté le développement de logiciels sont tels qu'il faut constamment améliorer les outils, les processus, ainsi que les méthodes de résolution de problèmes. Dans ce contexte, le développement à base de composants est très prometteur. La notion de composant est, cependant, très large et peut désigner beaucoup de choses. Nous considérons la définition suivante : « Un composant logiciel est une unité de composition dotée d'interfaces spécifiées. Un composant logiciel peut être déployé indépendamment et sujet à une composition par une tierce entité. » [UMV 02a]. L'objectif des composants logiciels est d'apporter un nouveau degré de séparation des préoccupations. Plusieurs nouvelles approches émergentes telles que la programmation orientée aspect [IBM 01], [LAD 02] et la programmation orientée sujet [IBM 01]. Tout comme l'avènement du paradigme objet a permis d'isoler, de compartimenter et de faciliter la modularisation des problèmes, les composants constituent, à un certain degré, un prolongement et une extension naturelle de cette façon de concevoir. En découpant, toujours plus finement, le problème en divers petits éléments, nous simplifions et nous facilitons sa résolution. Le principe de diviser pour régner est l'adage et le précepte de base de la programmation à base de composants.

La tendance du développement basé sur les composants logiciels est assez large et englobe la programmation orientée aspect. Ce nouveau paradigme est voué, à notre avis, à être probablement la prochaine grande révolution dans l'art de la programmation depuis l'apparition de l'objet comme mentionné dans plusieurs publications telles que [LAD 02] et [ORR 03]. La force des aspects réside dans leur comportement naturel à étendre les aptitudes des objets. Ils offrent de nouvelles méthodes élégantes pour repousser les limites éprouvées par le développement orienté objet [LAD 02]. La programmation orientée aspect ne remplace pas la programmation orientée objet, il s'agit plutôt d'un complément [EID 01]. Dans la Figure 1 [UMV 02a], nous pouvons constater le niveau d'abstraction versus le niveau de réutilisation, les aspects se situent au niveau des composants.

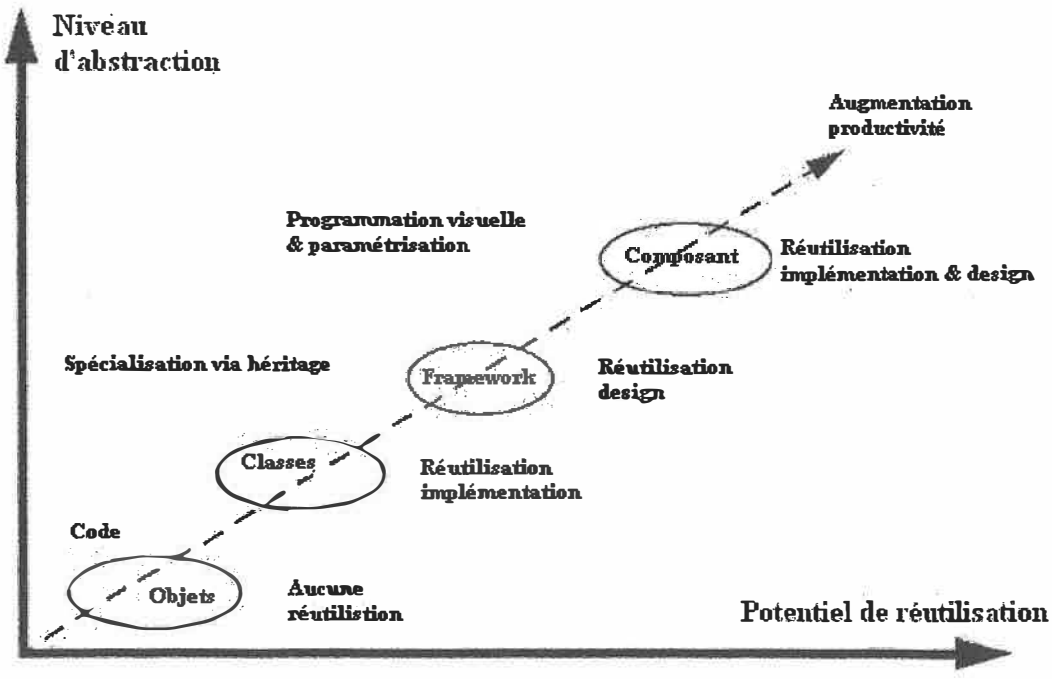

Figure 1 : Niveau d'abstraction VS la réutilisation

Chapitre 1

Introduction aux Aspects

1.1 Historique

Les premières idées sur le concept des aspects existent depuis longtemps déjà. Pour la première apparition du concept, tel qu'il est à l'heure actuelle, il faut remonter à 1997. Il s'agit essentiellement d'une idée issue de la communauté des chercheurs. Gregor Kiczales est souvent cité comme étant un des créateurs du concept et le fondateur d'AspectJ. Il dirigeait l'équipe du PARC de Xerox qui a développé la POA (Programmation Orientée Aspect) et AspectJ [KIC 03]. Actuellement, AspectJ est le plus abouti et le plus cité des environnements de développement orienté aspect [LAD 03]. Il représente une extension orientée aspect du langage Java. Le principal intérêt des aspects est d'offrir une nouvelle approche dans la résolution des problèmes. Cette approche permet de résoudre les problèmes d'une manière plus élégante qu'avec les approches classiques. La principale force des aspects réside dans leur capacité à étendre le comportement des objets. Les exigences non fonctionnelles (et parfois fonctionnelles) des programmes se retrouvent souvent dans diverses classes. Les aspects permettent de les regrouper dans des unités modulaires appelées aspects. Les aspects permettent surtout de mieux prendre en charge les exigences non fonctionnelles telles que, par exemple, la synchronisation, la performance et la fiabilité. Les objets, dans ce type de cas, éprouvent

beaucoup de difficulté. Ces exigences peuvent, en fait, être reliées à plusieurs objets. La gestion de telles fonctionnalités, à l'aide de la programmation orientée objet, devient difficile et engendre souvent une augmentation du couplage entre les constituants d'un programme. Les aspects sont particulièrement bien adaptés pour répondre à ce type d'exigences [BERT 03], [CON 03]. Les aspects permettent d'extraire ces exigences des classes de base, pour les concentrer dans des modules indépendants. Ceci améliore la modularité des différents éléments et permet d'améliorer la cohésion des classes avec tous les avantages qui peuvent en découler.

Parmi les principaux avantages, nous pouvons citer :

- Augmentation de la productivité : L'utilisation des aspects évite d'avoir à tout réécrire (redondance) et concevoir de nouveau certaines parties d'un programme.
- Diminution de la complexité du projet : En séparant le projet en diverses sections (modularité), la complexité est réduite.
- Amélioration de la qualité logicielle à différents points de vue (maintenance, test, réutilisation).

La Figure 2, extraite de [BAL 02], présente clairement le positionnement et l'intégration des aspects dans du code objet. Grâce au mécanisme d'intégration, les aspects sont plus souples et peuvent répondre plus efficacement aux problèmes engendrés par la dispersion des préoccupations, d'une part, et à la modularisation et la séparation des

préoccupations [IBM 01], d'autre part. De plus, l'intégration peut se faire aussi bien de façon statique que dynamique. L'intégration dynamique se fait lors de l'exécution du programme. L'intégration statique se fait lors de la compilation et il faut absolument recompiler le programme pour que les modifications effectuées aux aspects puissent être effectives. Nous pouvons citer, comme exemples, JAC [OWC 05] pour les implémentations dynamiques des aspects, et AspectJ [XAT 03] pour les implémentations statiques. Les trameurs (ou *Weavers*) sont le cœur de la programmation par aspect. C'est via ces outils que les aspects peuvent se greffer aux classes et au contrôle du programme. Le comportement des aspects est fortement relié à la façon dont les trameurs exécutent la liaison des aspects. Chaque implémentation des aspects utilise des trameurs différents.

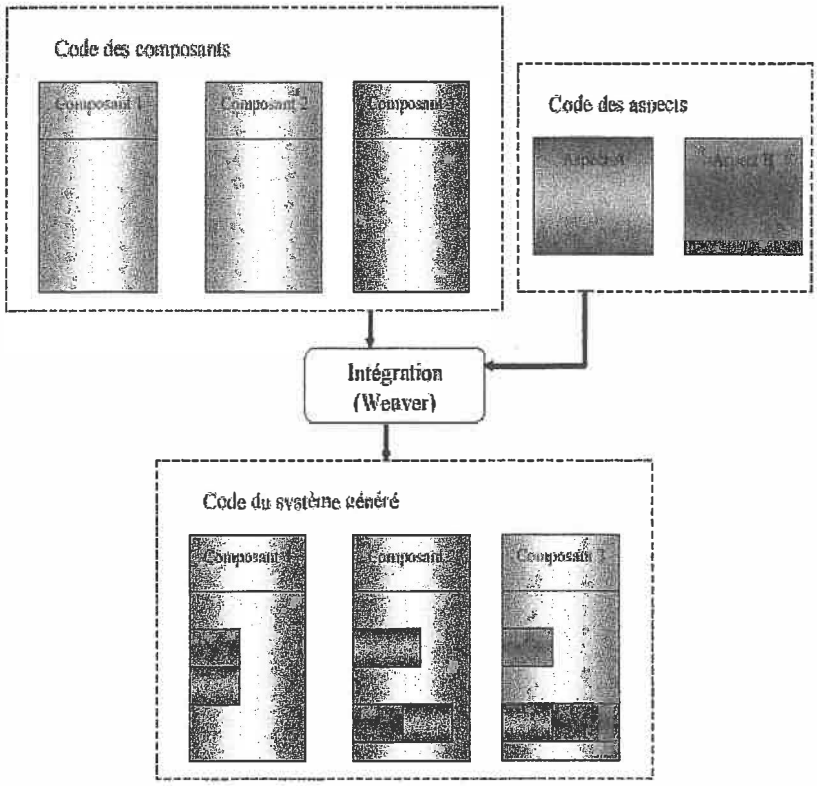

Figure 2 : Intégration du code des aspects

Le mécanisme de liaison entre les aspects et les objets permet aux aspects de connaître le contexte des objets au moment où l'aspect doit s'exécuter. On parle de programmation adaptative. Cette approche permet de programmer des aspects robustes et de prolonger la durée de vie des programmes en augmentant la flexibilité du code selon [OWC 05]. Le développement en programmation orientée aspect peut se résumer en 3 étapes très simples comme mentionnées dans [XAT 03] :

1. **La décomposition des éléments ou séparation des préoccupations :** Il s'agit donc d'identifier tous les composants et les aspects qui peuvent être isolés. On

sépare, tel que c'est illustré par la Figure 3, toutes les préoccupations, qu'elles soient fonctionnelles ou non.

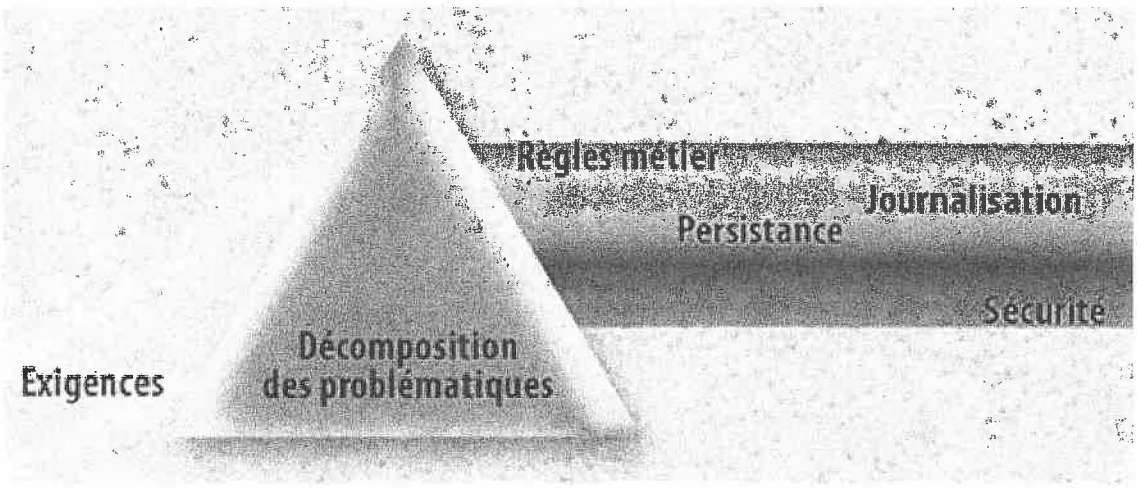

Figure 3 : Décomposition des problématiques

2. **L'implémentation de chaque préoccupation :** Chaque problématique sera codée séparément sous la forme d'un aspect. Dans un aspect, le programmeur définit aussi les règles d'intégration et de liaison avec les composants concernés.

3. **L'intégration du système :** Tout langage de programmation orientée aspect offre un mécanisme d'intégration appelé un trameur ou *weaver*. Le trameur, à l'image d'un métier à tisser, va donc composer le système final sur la base des règles et des modules qui lui ont été fournis tel qu'illustré par la Figure 4.

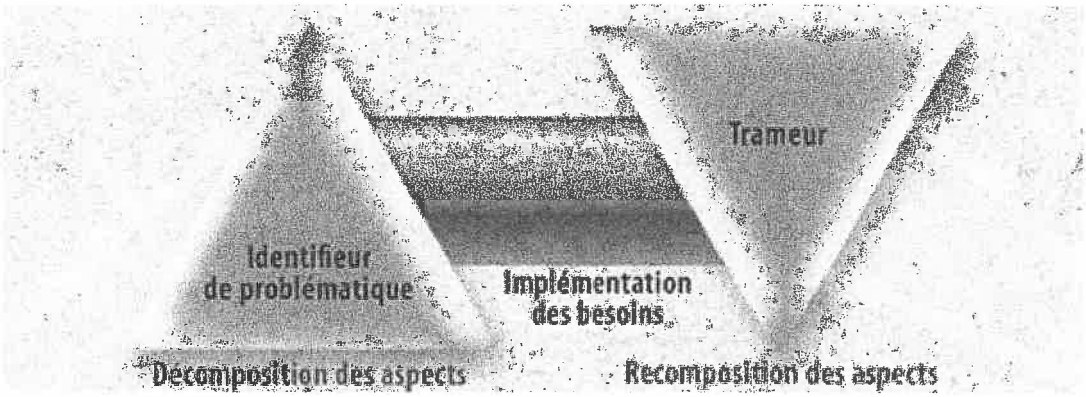

Figure 4 : Décomposition et recomposition des aspects

Il y a 3 éléments importants dans un programme orienté aspect [LAD 02], [XAT 03] :

1. **Point de jointure (*Joinpoint*) :** Il s'agit de balises dans le code du programme pour l'exécution de l'aspect. Un appel de méthode, un appel de constructeur, un accès en lecture ou écriture constituent des exemples de balises valides.

2. **Point de coupure (*Pointcut*) :** C'est une construction qui permet de désigner un ensemble de points de jointure et de capturer le contexte spécifique de chacun de ces points.

3. **Advice**[1] **(*advice*) :** Il s'agit du code à exécuter selon les différentes conditions contenues dans les points de coupure. Il existe 3 types d'advices:

- *Before (avant) :* s'exécute avant que les points de jointure ne soient exécutés.

[1] Aucune traduction satisfaisante de ce terme n'existe pour le moment, nous utiliserons donc le terme anglais dans la suite du texte.

- Around (autour) : s'exécute autour ou avec le point de jointure et peut avoir le contrôle sur son exécution.
- After (après) : s'exécute après le point de jointure.

1.2 Développement actuel

Actuellement, nous pouvons trouver plusieurs implémentations des aspects dans différents langages de programmation orientée objet. Nous pouvons citer, entre autres, AspectJ, JAC et Hyper/J pour JAVA. Il existe également des implémentations pour .Net (Aspect #, Eos, AspectDNG et LOOM.NET), le C++ (AspectC++, FeatureC++, XWeaver) ainsi que pour d'autres langages de programmation (AspectS for Smalltalk) [AOS 05]. Par ailleurs, on peut également noter plusieurs travaux et recherches portant sur les avantages de l'utilisation des aspects pour résoudre des problèmes concrets. Nous pouvons citer par exemple l'utilisation des aspects pour améliorer la sécurité des applications Web [REI 03] ou tout simplement la création d'outils évolués de débogage (Bugdel). Présentement, l'outil le plus utilisé et le plus répandu est AspectJ, qui représente une extension du langage Java. Cet outil peut se greffer à plusieurs environnements de programmation (JBuilder, Sun One Studio, Eclipse). Nous avons décidé de porter nos travaux sur cette implémentation. Cependant, notre approche est assez générique pour s'appliquer, avec de légères adaptations, à d'autres implémentations.

1.3 Outils de développement

Cette section est un résumé des outils de programmation et des environnements que nous avons évalués au début de ce projet de recherche. Nous avons essentiellement porté nos efforts sur l'implémentation la plus répandue et la plus citée dans la documentation et les travaux de recherche, celle de AspectJ. À cause de la nature expérimentale d'AspectJ (il est encore en plein développement), nous avons rencontré plusieurs problèmes de programmation liés surtout au manque de maturité des plates-formes. Nous introduisons sommairement, dans ce qui suit, deux des environnements supportant AspectJ et JAC.

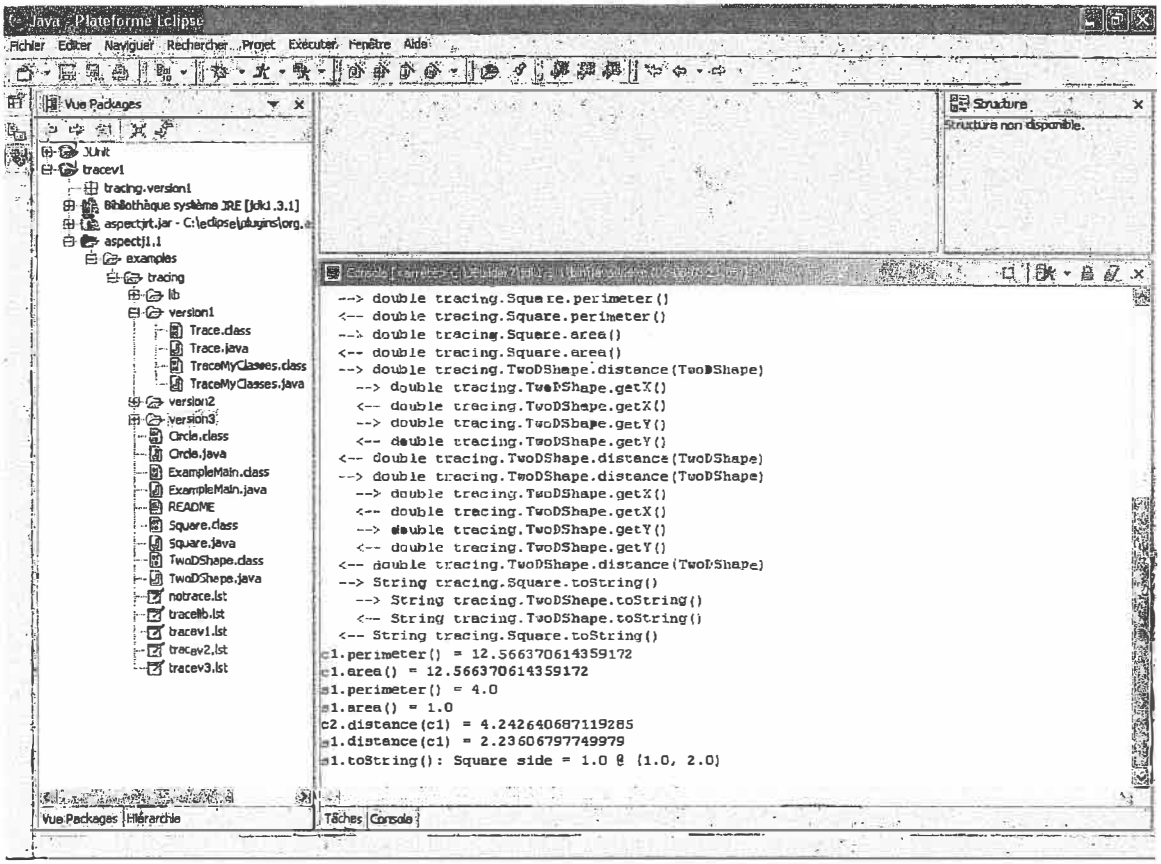

Figure 5 : Exécution d'un petit programme sous Eclipse

Eclipse 3.0 : Cette nouvelle version d'Eclipse est beaucoup plus conviviale et puissante que les précédentes. Plusieurs problèmes rencontrés avec les versions précédentes ont été corrigés dans cette nouvelle version. De plus, le nombre de modules disponibles pour cette nouvelle version est impressionnant. Eclipse est en phase de devenir un environnement de développement complet bien qu'il reste un projet open-source.

JAC (Java Aspect Components) **:** Il s'agit d'une implémentation différente de la programmation aspect. JAC est un outil RAD (Rapid Application Development) et un

serveur d'applications. JAC se présente comme une solution rapide de développement et répond très bien à ce besoin, car il est très convivial, dispose d'un environnement de modélisation UMLAF (UML Aspect Factory) et certains aspects sont déjà disponibles via des APIs (Application Programming Interface). Il s'agit d'un bon environnement de développement et l'utilisation du tutoriel intégré est très efficace.

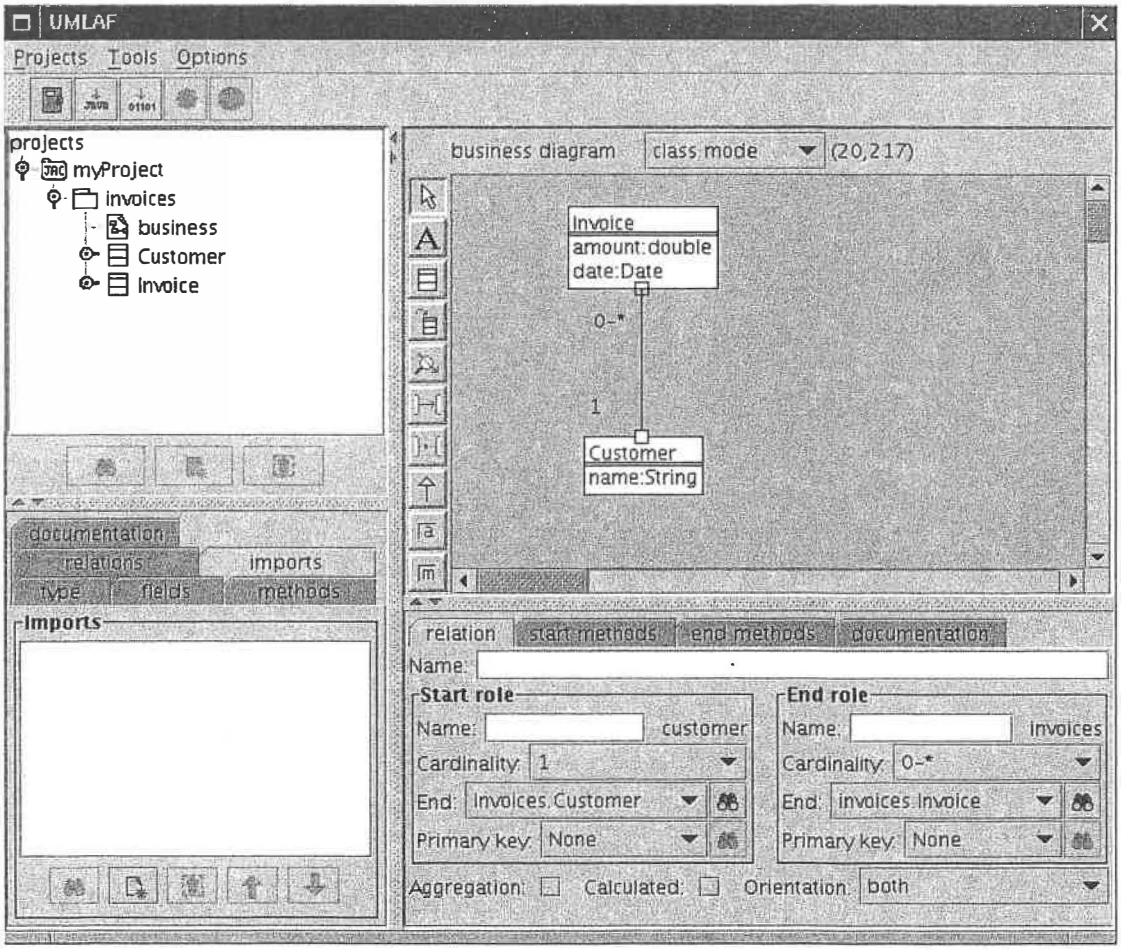

Figure 6 : Fenêtre de l'environnement JAC

Il existe cependant quelques différences notables entre JAC et AspectJ. Le tableau suivant résume ces différences [UMV 02b].

Tableau I : Comparaison entre AspectJ et JAC

AspectJ	JAC
Extension du langage JAVA	Framework POA, serveur d'applications
Nouvelle grammaire pour les aspects	Aspects écrits en Java pur
Utilise le code source. Chaque modification nécessite une nouvelle compilation	Un bytecode permet l'ajout, la suppression et la modification dynamique des aspects
Ne gère pas la distribution	Distribue automatiquement les aspects sur des serveurs distants
Ne permet que le développement d'aspects	Permet le développement d'aspects et/ou leur configuration dynamique
Atelier UML supportant les aspects	Un serveur + un atelier UML + une IHM de configuration, intégré à JBuilder, Forte et Emacs.
Pas d'aspects prédéveloppés	Bibliothèque d'aspects prédéveloppés configurables
Open Source Mozilla Public Licence	Disponible en licence LGPL

De plus, JAC permet via des APIs de disposer immédiatement et rapidement de plusieurs aspects. Nous donnons dans le Tableau II, la liste des aspects disponibles avec les librairies actuelles de JAC [OWC 05].

Tableau II : Liste des aspects de JAC

Liste des aspects disponibles		
Accès distant	Authentification	Binding
Broadcasting	Cache	Cohérence
Confirmation	Débogage	Déploiement
GUI (graphical user interface)	Intégrité	Load balancing
Persistance	RTTI (run time type information)	Session
Synchronisation	Trace	Transaction
Utilisateur		

1.4 Définition d'un aspect

Bien que le concept d'aspect a été bien décrit, il subsiste néanmoins beaucoup de questions sur ce qu'est la nature d'un aspect. Ce débat, tout comme celui sur les composants, repose sur un concept général qui prend différentes significations selon le contexte. Un aspect peut gérer des préoccupations fonctionnelles (session, transaction, trace) ainsi que des préoccupations non fonctionnelles (sécurité, performance). L'optimisation est une préoccupation non fonctionnelle, mais comment exprime-t-on l'optimisation via un aspect? De plus, on peut noter qu'il existe de nombreux outils et que ceux-ci diffèrent grandement en termes d'implémentation, certains sont des extensions alors que d'autres sont intégrés au langage. Plusieurs de ces implémentations n'ont même pas atteint le stade de la version 1.0. Par exemple, AspectC++ (version 0.9.3) et JAC (version 0.12.1) sont encore en plein développement. Le côté encore jeune

et très évolutif des aspects ne facilite pas leur compréhension et la création d'outils universels. Cela est dû au fait qu'il s'agit d'un concept nouveau et en manque de maturité.

Mais qu'est-ce qu'un aspect pour nous? Un aspect est un module perpendiculaire ou transverse qui peut s'intégrer via un trameur quelconque au code déjà existant pour y ajouter ou modifier des comportements. La programmation orientée aspect se définie par trois concepts fondamentaux que l'on retrouve dans toutes les implémentations. Un module indépendant appelé l'aspect, des mécanismes de liaison que nous appelons des points de coupure et finalement, des points d'ancrage dans le code ou communément nommé point de jointure. En partant de ces trois principes fondamentaux, nous pouvons créer un outil général pour la détection de conflits et que nous pourrions adapter en fonction des mécanismes de liaison utilisés.

Pour le présent travail, nous nous sommes concentrés sur AspectJ exclusivement, mais nous espérons étendre nos travaux à d'autres langages dans le futur.

Chapitre 2

POA: Évaluation et Perspectives de recherche

2.1 État de l'art

La présente section donne les résultats du survol de l'état de l'art que nous avons effectué dans le domaine du développement orienté aspect au début de cette recherche. L'objectif principal de ce travail de synthèse était essentiellement de recenser et d'évaluer ce qui a été réalisé dans ce domaine (en lien avec notre problématique), d'une part, et de dégager des ouvertures intéressantes en termes de recherche pour répondre aux nombreux problèmes qui restent posés dans le développement orienté aspect, d'autre part. Le domaine est en pleine effervescence. La problématique du développement orienté aspect peut être divisée en plusieurs thèmes distincts (concepts de base, méthodologie de développement, modélisation, comportement des aspects, interactions entre les aspects, relations entre aspects et classes, test et traçabilité, qualité et métriques, etc.). Pour chacun des thèmes retenus et que nous avons privilégiés, une évaluation de plusieurs travaux de recherche a été réalisée. Il est évident que la liste des travaux étudiés dont certains sont cités dans cette section est loin d'être exhaustive. Elle permet, néanmoins, de dresser un portrait global de la couverture des thèmes abordés. En effet, cette évaluation nous a permis de dégager les avancées intéressantes dans ce domaine relativement à la problématique du thème, les principales approches adoptées, et surtout

les principales faiblesses actuelles. Ces lacunes constituent, à notre avis, des pistes de recherche intéressantes.

2.2 Thèmes de recherche

Pour effectuer une évaluation des principaux travaux réalisés dans ce domaine et déterminer la couverture de ses différents axes, une grille comportant huit critères de comparaison a été conçue. Il est à noter que certains critères sont fortement couplés et que la grille définie constitue un premier cadre de comparaison qui est sujet à évolution. Nous présentons, dans ce qui suit, la liste des principaux thèmes retenus. Par la suite, nous regroupons l'information concernant la couverture de ces thèmes dans un tableau. Une analyse détaillée de ces thèmes est présentée dans la section suivante. Les thèmes sélectionnés sont :

Concepts généraux reliés au paradigme aspect: Nous nous sommes intéressés dans cette partie à plusieurs articles abordant la POA d'une manière générale et donnant une vision globale de l'approche. Ces articles sont nombreux, car le sujet est relativement récent et nécessite une vulgarisation [LAD 02]. En fonction de la démocratisation de la technologie aspect, ce thème sera de plus en plus abordé dans le futur. Il nous a permis, néanmoins, de relever l'intérêt important suscité par les aspects et leur utilisation de plus en plus croissante durant les dernières années.

Interaction entre les aspects: Ce point porte essentiellement sur les interactions pouvant exister entre les différents aspects décrits dans un problème et leurs conséquences éventuelles. Dans [HAN 03], les auteurs identifient 4 grandes catégories d'inconsistances ou conflits en lien avec l'utilisation des aspects :

1. **Spécification transverse :** la manière dont les aspects s'intègrent aux classes et l'utilisation des points de jointure.
2. **Aspect - Aspect :** les conflits entre deux aspects concurrentiels dans le contexte de l'exécution du code.
3. **Base – Aspect :** la base fait appel à un aspect invasif et une dépendance circulaire est créée entre l'aspect et la base.
4. **Préoccupation – Préoccupation :** les préoccupations entrent en conflits entre elles par le biais de l'utilisation des aspects.

Ce thème de la problématique est assez complexe et particulièrement difficile à gérer. La souplesse que procure la programmation orientée aspect constitue certes un point fort. Cependant, elle engendre de nouveaux conflits. La détection et la gestion de ces conflits restent un problème crucial et exigent actuellement un effort important de la part du développeur. Les conflits non détectés peuvent avoir des conséquences importantes aussi bien sur la qualité du produit que sur ses coûts de développement et de maintenance. Il est évident qu'une détection précoce de ces conflits, le plus tôt possible dans le processus

de développement, présente des avantages multiples. Il est absolument nécessaire de développer, à ce niveau, des approches efficaces de détection et de correction de ce type de conflits. Par ailleurs, l'intégration de ces approches dans des environnements de développement est indispensable.

Relation entre les aspects et les classes: Ce point porte sur les relations qui peuvent exister entre les aspects et les classes de base. Comme les aspects sont orthogonaux aux classes, il faut bien définir leur collaboration avec les classes et maîtriser les impacts éventuels. Ce point est étroitement relié au problème de l'intégration des aspects dans le code.

Méthodologie de la POA: Ce thème est constitué de tout ce qui se rapporte à la méthodologie de ce paradigme. Cela concerne la bonne séparation des préoccupations ainsi que les techniques pour retracer la présence des aspects fonctionnels et non fonctionnels dans les énoncés de l'analyse. Nous englobons également dans ce point toutes les questions relatives au comportement dynamique des composants.

Modélisation: Ce point regroupe tous les outils et techniques nécessaires pour représenter les aspects et les préoccupations. Il englobe les outils UML ainsi que tous les outils externes permettant de gérer et de modéliser les composants orthogonaux.

Test et traçabilité: Cela concerne les méthodes et outils de test permettant de vérifier, par exemple, la bonne connexion des aspects ainsi que l'impact réel des aspects sur l'exécution du programme. Ce sujet inclut les compilateurs et le trameur qui lie les aspects aux classes. En fait, il s'agit d'une manière générale de valider que l'exécution soit conforme aux spécifications et aux résultats attendus.

Qualité et métriques: Parce que le sujet est très récent et en plein développement, il n'existe pas encore de travaux importants [GEL 04], [GEL 05], [ZHA 02] sur le sujet et encore moins de consensus sur des métriques adaptées aux aspects et à leurs spécificités. Il est possible de s'inspirer des travaux effectués pour les systèmes orientés objet, de les adapter et/ou les étendre au paradigme aspect. Il est évident que le paradigme aspect apporte une dimension intéressante en complément au paradigme objet. Cette dimension apporte aussi son lot de problèmes et de complexité. Il est pertinent de se pencher rapidement sur ces questions et de développer des métriques permettant d'évaluer, par exemple, les dépendances multiples qui sont engendrées dans un programme orienté aspect ainsi que la qualité de la structure de tels programmes.

Comportement des aspects: Ce point, très large, englobe les fonctionnalités spécifiques et ce qui est en lien avec une utilisation concrète des aspects. La performance, la sécurité et l'optimisation sont des éléments souvent mentionnés dans ce thème. Par exemple, un article portant sur l'utilisation de la POA pour obtenir une

application Web sécuritaire entrerait dans ce thème, car il présente le comportement, les avantages et/ou les inconvénients de l'utilisation concrète des aspects dans un cadre applicatif.

Les différents points présentés précédemment constituent une division assez sommaire du domaine. Il est indéniable, cependant, qu'il existe un couplage important entre certaines de ces dimensions. La gestion des interactions entre les aspects, par exemple, dépend des outils de modélisation. Par ailleurs, les relations pouvant exister entre les classes et les aspects sont décrites par la méthodologie de séparation des préoccupations. Chacun de ces thèmes constitue, cependant, un axe distinct et important de recherche. Dans la prochaine section, nous présentons les principales approches abordées dans chacun des thèmes ainsi qu'une brève analyse relevant les forces et les faiblesses.

2.3 État actuel et prévision des axes de recherche

Nous avons tenté de dresser un premier bilan des principaux travaux effectués dans le domaine du développement orienté aspect en effectuant une couverture des thèmes présentés dans la section précédente. Nous avons réalisé une compilation de plusieurs articles selon les sujets couverts. La Figure 7 représente globalement le résultat de ce travail. Il est à noter que cette figure ne se veut pas une représentation exhaustive de l'ensemble des travaux effectués à ce jour dans ce domaine. Elle donne plutôt un portrait

global sur le domaine et renseigne sur l'importance relative de la couverture de ses différents thèmes. Ce graphique illustre, en particulier, les écarts importants qui existent dans la couverture des différents thèmes et mets en évidence les axes qui ont fait l'objet de relativement peu de travaux. Il permet également de dresser, selon l'importance des thèmes et surtout l'importance de leur impact sur l'évolution et la pénétration du paradigme, un premier plan de tendances que pourrait, ou même devrait, suivre la recherche dans ce domaine dans le futur. La problématique, par exemple, de l'interaction entre aspects constitue un sujet crucial, à la base même de l'évolution du paradigme. Sa très faible couverture peut engendrer plusieurs problèmes qui peuvent, en fait, ralentir l'évolution et la pénétration du paradigme aspect.

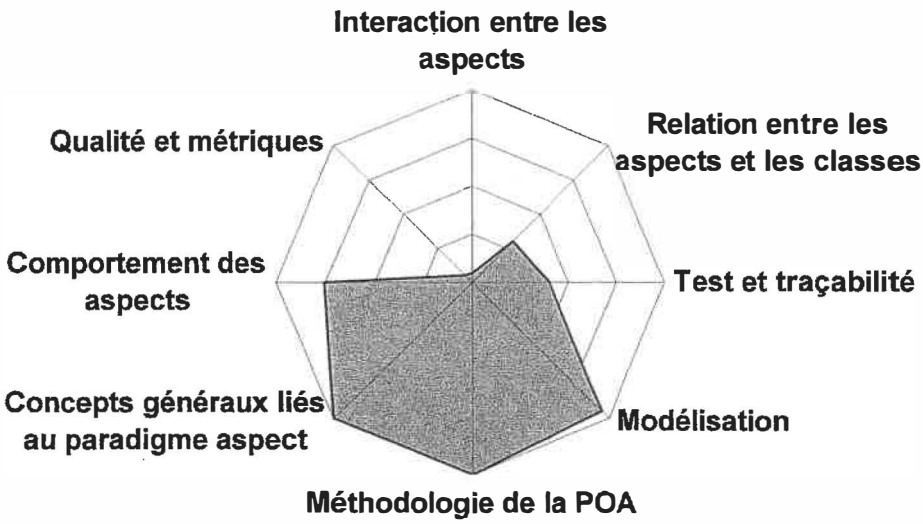

Figure 7 : Couverture de la problématique

Notre démarche se veut plus qualitative que quantitative. Un des points, que ce travail voulait mettre à jour, est la couverture relative des sujets. On voit clairement que beaucoup de travaux ont porté sur la présentation générale de l'approche, l'introduction aux concepts généraux, la méthodologie, la modélisation ainsi que le comportement des aspects. Il est normal de retrouver une telle prédominance de ces sujets. En fait, ils constituent les éléments de base sur lesquels il faut s'attarder en premier pour bâtir un concept solide et cohérent. On retrouve, à l'opposé, les thèmes reliés aux interactions entre les aspects, aux relations entre aspects et classes, au test des programmes orientés aspect ainsi qu'à l'évaluation des attributs de leur qualité. Ces différentes questions constituent des sujets cruciaux et nécessitent des efforts de recherche importants. Ces différentes questions sont reliées à l'évolution du paradigme aspect et à sa maturité. Les sujets reliés aux interactions entre les aspects, d'une part, et les relations entre les aspects et les classes de base, d'autre part, sont fondamentaux dans ce type de développement. Ils représentent actuellement des sujets complexes et difficiles à gérer. Des efforts de recherche importants doivent par conséquent être investis à ce niveau. En effet, ces thèmes représentent de nombreux défis, car les aspects peuvent influencer le flux de contrôle d'un programme. Les aspects peuvent modifier la pile d'exécution et/ou changement de l'état du programme [STO 03a]. Ces changements sont dynamiques et ne sont donc visibles que lors de l'exécution du programme. Ceci ne fait qu'augmenter leur complexité et la difficulté de la mise au point dans ce contexte.

Concepts généraux reliés au paradigme aspect: Il s'agit d'un sujet qui est souvent couvert par des articles généralistes et de présentation des langages de programmation orientée aspect. On y retrouve souvent des informations redondantes, mais néanmoins pertinentes. À cause de la complexité du thème, plusieurs articles ont de la difficulté à vulgariser les concepts reliés au paradigme aspect. Il faut également noter qu'il existe de nombreux outils expérimentaux [REI 03]. Il est parfois difficile de bien comprendre les exemples, lorsqu'on est confronté à des compilateurs capricieux. L'implémentation aspect la plus avancée et la plus documentée est AspectJ [GRA 03], [KIS 02], [LAD 03].

Interaction entre les aspects: Il s'agit d'un point très important et malheureusement faiblement couvert dans l'approche orientée aspect actuellement. De tous les papiers que nous avons étudiés, seulement quelques-uns couvrent partiellement le sujet. Le rapport du workshop AAOS 2003 [HAN 03] identifie clairement les problèmes majeurs auxquels doivent faire face les aspects pour être intégrés au développement traditionnel. L'un des trois éléments clés présentés dans le rapport est l'identification et la résolution des conflits et des inconsistances dans la description des aspects. Dans [XAT 03], on fait mention du problème de boucle infinie. Un aspect permet de faire la trace d'un programme. Le problème est que l'aspect s'appelle lui-même, ce qui provoque une exception de dépassement de pile. Donc, l'ajout d'un aspect ne modifie pas uniquement une classe, mais peut également influencer d'autres aspects.

Relation entre les aspects et les classes: La couverture de ce sujet est relativement moyenne. La programmation orientée aspect ajoute une complexité accrue aux relations pouvant exister entre les constituants d'un programme. La raison principale est due au fait que les aspects peuvent s'insérer un peu partout dans le programme, et sont très dynamiques lors de l'exécution. La souplesse de cette programmation permet de faire (à l'état actuel des choses) un peu n'importe quoi. Cela a des qualités comme des défauts qu'il faut apprendre à gérer. Il faut également anticiper les effets primaires, les effets secondaires et les effets désirés lors de l'intégration des aspects [STO 03a]. Les aspects permettent l'inversion des dépendances [NOR 01]. Le programme, dans ce cas, n'utilise pas le service, c'est le service qui utilise le programme. Donc, on ne peut pas traiter les aspects comme étant des objets. Ce point est très important pour le développement et l'utilisation de la programmation orientée aspect.

Méthodologie de la POA: Comme on peut le constater aisément, beaucoup de travaux ont porté sur le sujet (méthodologie, modèles, langages) [HAN 03]. De plus, nous pouvons également citer l'utilisation d'approches telles que les méthodes formelles [AND 01], [CHE 01], la technique des cas d'utilisation [OWC 05] et d'autres travaux d'analyse pour extraire et décrire les aspects [BERT 03]. La méthodologie, d'une manière générale, la séparation des préoccupations en particulier, sont des points importants et de base pour l'ensemble des autres points. Dans les articles étudiés, plusieurs auteurs ont

présenté des méthodes et techniques diverses pour capturer les préoccupations [CON 03]. La force des méthodes proposées est de bien capturer certaines préoccupations fonctionnelles telles que l'authentification, la gestion des threads, etc. Cependant, il est toujours difficile de faire ressortir tous les éléments, car ces derniers sont orthogonaux et souvent très reliés à d'autres éléments. De plus, la capture des préoccupations non fonctionnelles telles que l'optimisation, la performance et la fiabilité, entre autres, reste très difficile.

Modélisation: Il s'agit principalement de tous les outils et les diverses extensions des ateliers UML pour supporter la modélisation aspect. L'approche la plus répandue est l'extension du langage UML actuellement utilisée pour la modélisation objet [ALD 01]. Cela constitue une force majeure. Il suffit de réutiliser des idées déjà éprouvées pour réaliser rapidement des progrès.

Test et traçabilité: Il s'agit d'un sujet relativement peu couvert. On note quelques outils développés pour faciliter la phase de tests des programmes et le comportement des aspects. Ils sont, cependant, loin de fournir tous les éléments nécessaires pour supporter efficacement le processus de test de ce type de programmes. Il est vraiment nécessaire de développer des techniques efficaces et appropriées tenant compte des spécificités du paradigme aspect. Par ailleurs, le développement d'outils

supportant ces techniques est indispensable à notre avis tenant compte de la complexité supplémentaire engendrée par le paradigme (au niveau contrôle) et surtout ses souplesses. Certains travaux ont tenté d'intégrer des notations formelles permettant de vérifier et valider le tramage (weaving) [AND 01], [STO 03a]. Pour faciliter les tests, la plupart des auteurs ayant abordé le sujet présentent des approches compartimentées. Il s'agit de test intra-advice, intra-aspect, intra-module, inter-modules, etc. [ZHA 01b]. Les auteurs pensent que de cette façon les tests sont simplifiés, mais leur nombre augmente. Ces approches constituent, à notre avis, un premier pas dans un sujet qu'il faut absolument explorer. Il reste, en effet, de nombreux problèmes à surmonter. En particulier, les problèmes reliés à la subtilité des aspects et ceux engendrés par la dimension dynamique inhérente à ce type d'applications. Cette partie de la problématique constitue un axe de recherche important.

Qualité et métriques: Il s'agit là aussi d'un sujet relativement très peu couvert. Très peu de travaux ont porté sur la qualité des programmes orientés aspect ou sur le développement de métriques tenant compte des spécificités de ce paradigme [GEL 04], [GEL 05], [ZHA 02]. Cela représente un important axe de recherche qu'il serait pertinent de considérer. Il est nécessaire de développer des métriques permettant d'évaluer, en particulier, la complexité (à différents niveaux) inhérente à ce type d'applications ainsi que les différents types de dépendances engendrés par ce paradigme. Les métriques peuvent également, à notre avis, contribuer à faciliter la gestion de

certains types de conflits dus aux dépendances. Il serait également pertinent d'évaluer l'impact de l'utilisation des aspects sur la qualité des programmes (cohésion des classes, couplage, taille du code, etc.).

Comportement des aspects: Les articles, qui ne couvraient aucun sujet précis, présentaient souvent les fonctionnalités spécifiques aux composants dans un cadre concret. Certains articles traitent de l'intégration des aspects dans le cadre d'un projet Web [REI 03]. On cite souvent l'exemple démontrant la dispersion du code pour l'authentification (logging) dans le module TOMCAT [OWC 05]. De plus, les auteurs de [REI 03] démontrent qu'une application Web en orienté aspect présente une meilleure encapsulation, localisation du code et sécurité. En ce qui concerne le comportement distinct des aspects, les différentes implémentations de la POA ne fonctionnent pas toutes selon les mêmes principes. Par exemple, JAC permet une modification en temps réel des aspects en travaillant sur le bytecode alors que AspectJ nécessite une re-compilation [OWC 05].

Les aspects offrent une grande souplesse qui peut, malheureusement, conduire à briser l'encapsulation des classes [HAN 03]. On constate que cela peut engendrer de nombreux problèmes. Les métriques, par exemple, peuvent servir comme indicateurs et contribuer dans la gestion de ces problèmes. On constate cette problématique dans la couverture des différents domaines. Ceci est dû également, en partie, au fait que la POA constitue un

paradigme relativement récent. À notre avis, il y a un grand déséquilibre entre ce qu'apporte le paradigme (plusieurs aspects intéressants) et les outils (à différents niveaux) qui doivent l'accompagner pour en assurer la réussite. La grande majorité des travaux et des articles couvrent les domaines de base, c'est-à-dire les concepts généraux, la méthodologie, la modélisation et le comportement des aspects. À l'opposé, on retrouve les problèmes reliés aux interactions entre les aspects, aux relations entre les aspects et les classes, à la qualité, ainsi qu'aux tests qui sont peu couverts et qui doivent composer avec les problèmes engendrés par la souplesse des aspects. Bref, on retrouve d'une part, les thèmes qui démontrent les avantages de la programmation orientée aspect et de l'autre, les thèmes qui doivent en assurer la qualité. Ces derniers domaines présentent des axes intéressants de recherche. Le Tableau III : Grille d'évaluation présente un résumé des notions, des approches, des forces ainsi que des faiblesses pour chacun des thèmes. Il ne s'agit pas d'un résumé exhaustif. Néanmoins, il représente l'état de l'art dans ce domaine tenant compte de travaux récents et d'approches mis de l'avant lors de certaines rencontres scientifiques importantes.

Tableau III : Grille d'évaluation

Sujet	Notions	Approches	Faiblesses	Forces
Concepts généraux reliés au paradigme aspect	- Présentation - Notions de base - Programmation [GRA 03], [KIS 02], [LAD 03]	- Exemple et code - Présentation des composants - JAC [OWC 05] - AspectC - AspectJ [XAT 03]	- Complexité du sujet - Instabilité outils	- Extension à l'objet - Multiple projet en cours(C, Java, etc.) - Beaucoup d'articles
Interaction entre les aspects	- Impact direct - Impact indirect - Conflit	- Trace - Filtre de composition [BERG 03]	- Relation dépendances inversée - Joinpoint commun	- Facilité de tracer un programme
Relation entre les aspects et les classes	- Extension - Impact - Conflit	- Approche par contrat - Connecteur OnlineAuction - Agent [GAR 01]	- Différentes implémentations	- Documentation programmation - Extension de l'objet
Méthodologie de la POA	- Séparation préoccupations [CON 03] - Compréhensibilité - Enseignement [HAB 01] - Dynamique [POP 03]	- Modèle FRIDA [BERT 03] - Pattern [HAC 03] - Processus algébriques [AND 01] - Logique quasi-booléenne [CHE 01] - Knit [EID 01] - Méthodes formelles [AND 01], [CHE 01]	- Orthogonalité [BERG 03] - Aspects non fonctionnels	- Capture les aspects fonctionnels aisément - Fortement lié à la modélisation
Modélisation	- Outil d'analyse - Représentation des ressources [ALD 01] - Réutilisation	- Use case [JAC 03] - Design rational Graph (DRG) [BAN 01] - Intégration des agents	- Expression des fonctionnalités abstraites	- Réutilisation des principes OO - Adaptation des outils OO
Test et traçabilité	- Impact en temps réel - Stabilité - Cohérence - Exactitude - Trace et suivi [STO 03a]	- Graphe de contrôle du flux - Intra-advice/aspect/module [ZHA 01b] - Inter-modules - Méthode algébrique [AND 01]	- Impact subtil des aspects - Peu d'outils - Contexte du control flow en temps réel	- Compartimentation des tests - Facilité de tracer un programme
Qualité et métriques	- Fonctionnalité - Fiabilité - « Utilisabilité » - Rendement - Maintenabilité - Portabilité	- Graphe de dépendances aspect [ZHA 02] - Graphe de dépendances interprocédural [ZHA 02]	- Débuggeur de qualité - Aucune métrique	- Possibilité de réutilisation des techniques de l'OO
Comportement des aspects	- Sécurité [SLO 03] - Fonctionnalités spécifiques - Performance - Optimisation - Avantage - Comportement	- AJDT [CLE 03] - Orienté Web [REI 03] - Weaving dynamique	- Complexité - Différents outils et méthodes de programmation	- Beaucoup d'exemples - Application dans divers domaines

2.4 Insuffisances actuelles

Ce chapitre a présenté un survol rapide de la problématique de la programmation orientée aspect selon plusieurs points de vues. D'une manière générale, il est indéniable de constater que l'intérêt que porte la communauté à ce paradigme est en croissance constante. À notre avis, il ne serait pas étonnant que les aspects soient intégrés dans tous les langages orientés objet et environnements de programmation professionnels. Les aspects représentent une extension «naturelle» des objets. Cependant, il reste de nombreux problèmes complexes et subtils qu'il faut résoudre avant de voir une pénétration importante des aspects dans le développement à grande échelle. La grande latitude permise par les aspects, qui constitue une de leurs forces, est néanmoins un point qu'il faut bien gérer. Il est aisé d'engendrer des conflits dans la définition des aspects. Ces conflits sont très difficiles, voire impossible pour certains, à détecter par les compilateurs actuels. Ceci constitue une des faiblesses importantes des outils disponibles supportant ce type de développement. Le présent travail de recherche se concentre sur les conflits entre les aspects. Nous croyons qu'il s'agit là d'un point critique, car si on ne trouve pas de solutions efficaces pour gérer les conflits subtils entre les aspects, la POA restera une merveilleuse idée sans véritable implémentation robuste et efficiente.

Chapitre 3

État de l'art sur la détection des conflits

3.1 Motivation

Bien que les aspects apportent indéniablement maintes solutions intéressantes à la représentation des préoccupations transverses, il reste néanmoins qu'ils offrent une grande latitude d'action. Cette latitude se traduit par une grande liberté d'interagir avec toutes les classes d'un programme. Ceci accroît considérablement la complexité de leur modélisation [HAN 03], d'une part, et rajoute une dimension supplémentaire en termes de contrôle, d'autre part. Par ailleurs, l'amélioration de la modularité et de la localité des préoccupations contraste, tel que mentionné dans divers travaux, notamment dans le résumé de la conférence AAOS 2003 (Analysis of Aspect-Oriented Software 2003) [HAN 03], avec la tendance des aspects à détruire la localité de l'exécution du flux de contrôle. En effet, l'intégration des aspects peut entraîner des effets secondaires et affecter le flux interne du programme [STO 03a]. Cela entraîne une difficulté dans la prédiction des interactions entre les différents modules d'un programme et de leur impact. Ces différentes questions montrent clairement que le développement de mécanismes efficaces pour la bonne gestion des interactions entre les aspects et la détection des conflits qui en découlent constitue un enjeu crucial pour la maturité de la POA ainsi que sa large diffusion.

Par ailleurs, une détection précoce de ces conflits permet de réduire les coûts de développement et de faciliter la programmation tout en préservant un haut niveau de qualité. Cependant, la détection des conflits ainsi que leur résolution sont loin d'être des tâches aisées. En effet, il n'est pas très facile de modéliser les aspects et de les intégrer dans un modèle objet. Selon Hanneman [HAN 03], la détection et la résolution des conflits et l'analyse de l'impact des aspects constituent des points clés pour le développement et l'intégration de la POA. C'est dans ce contexte et dans un objectif d'assurance-qualité que nous avons décidé de nous pencher sur ces problèmes. Plus spécifiquement, nous avons décidé de nous concentrer sur les conflits entre les aspects. Selon nous, il s'agit de l'une des problématiques actuelles les plus importantes à résoudre. Le développement et l'intégration des aspects dépendent en grande partie de la gestion des conflits. Nous proposons dans le chapitre suivant une nouvelle technique qui permet de capturer rapidement les conflits tôt dans le processus de développement. Elle se fonde sur une analyse des modèles et présente des éléments de solution originaux. Elle permet également d'en assurer le suivi, lors des étapes ultérieures du développement et la gestion en fonction des détails que procurent ces phases. Nous donnons, dans ce qui suit, une brève présentation des rares travaux publiés dans la littérature sur la détection des conflits entre les aspects.

3.2 Évaluation des approches de détection

Actuellement, il y a très peu d'articles qui couvrent la problématique de la détection des conflits entre les aspects. Les travaux de Douence et al. [DOU 02] constituent une bonne approche. Les auteurs se basent sur l'utilisation d'un langage formel pour la description des aspects. Par analyse statique, ils détectent les interactions entre les aspects et proposent un framework pour résoudre les conflits. Ces travaux sont davantage axés sur une analyse du code des aspects. La méthode proposée par Douence se concentre exclusivement sur une détection sémantique de l'interaction des aspects. Notre approche est relativement plus précoce pour la détection des conflits. Elle est plus orientée vers une analyse des modèles. L'analyse statique du code, dans le cadre de notre approche, est complémentaire à l'analyse des modèles.

Störzer et Krinke [STO 03a], [STO 03b] ont également proposé des solutions intéressantes permettant de détecter les conflits entre les aspects. Dans l'article de Störzer, Krinke & Breu [STO 03a], les auteurs présentent une analyse des traces d'exécution pour détecter les modifications et les divers types d'effets engendrés par les aspects. Les approches dynamiques sont certes intéressantes, mais présentent néanmoins l'inconvénient majeur d'être coûteuses et relativement tardives dans le processus de développement. Les aspects ont des impacts globaux et peuvent influencer grandement un système. La détection précoce des conflits présente plusieurs avantages. Störzer et Krinke [STO 03b] présentent également une analyse de l'interférence dans l'héritage

entre classes. Une interférence est provoquée par l'ajout d'aspects introduisant de nouvelles méthodes. Nous pouvons également citer les travaux de Bergmans [BERG 03] qui s'attardent sur une détection des conflits sémantiques engendrés par des préoccupations transverses. Leur papier met l'accent sur la difficulté de traiter des préoccupations partagées par plusieurs modules. L'approche adoptée passe par l'utilisation des filtres de composition. Les travaux de ces auteurs sont davantage axés sur l'analyse des conséquences de l'intégration des aspects sur le comportement des classes. Nos recherches tentent plutôt de faire une analyse prédictive précoce de l'impact de l'introduction des aspects. Notre démarche est davantage basée sur l'analyse des modèles, complétée ultérieurement par une analyse statique du code.

Par ailleurs, nous pouvons également citer les outils de développement qui ne cessent de se raffiner. Cependant, ils n'apportent qu'une faible assistance aux programmeurs dans la détection des conflits. Les outils de développement AJDT [CLE 03] permettent, par exemple, de visualiser graphiquement les points de jointure. La Figure 8 illustre un exemple d'interface. À droite, nous retrouvons l'explorateur de projet représentant la structure de la classe et l'ensemble des points de jointure ainsi que les types d'advice qui s'appliquent. Dans la fenêtre centrale, nous retrouvons le code source de la classe *HelloWorld2* et dans la marge de gauche les marqueurs qui indiquent quels sont les advices qui vont s'appliquer à cette ligne. Cet outil permet de visualiser rapidement les aspects et leurs imbrications dans les classes. Cependant, aucune vérification ou

détection de conflits n'est implémentée. En fait, c'est un outil statique facilitant la programmation et la gestion manuelle des conflits.

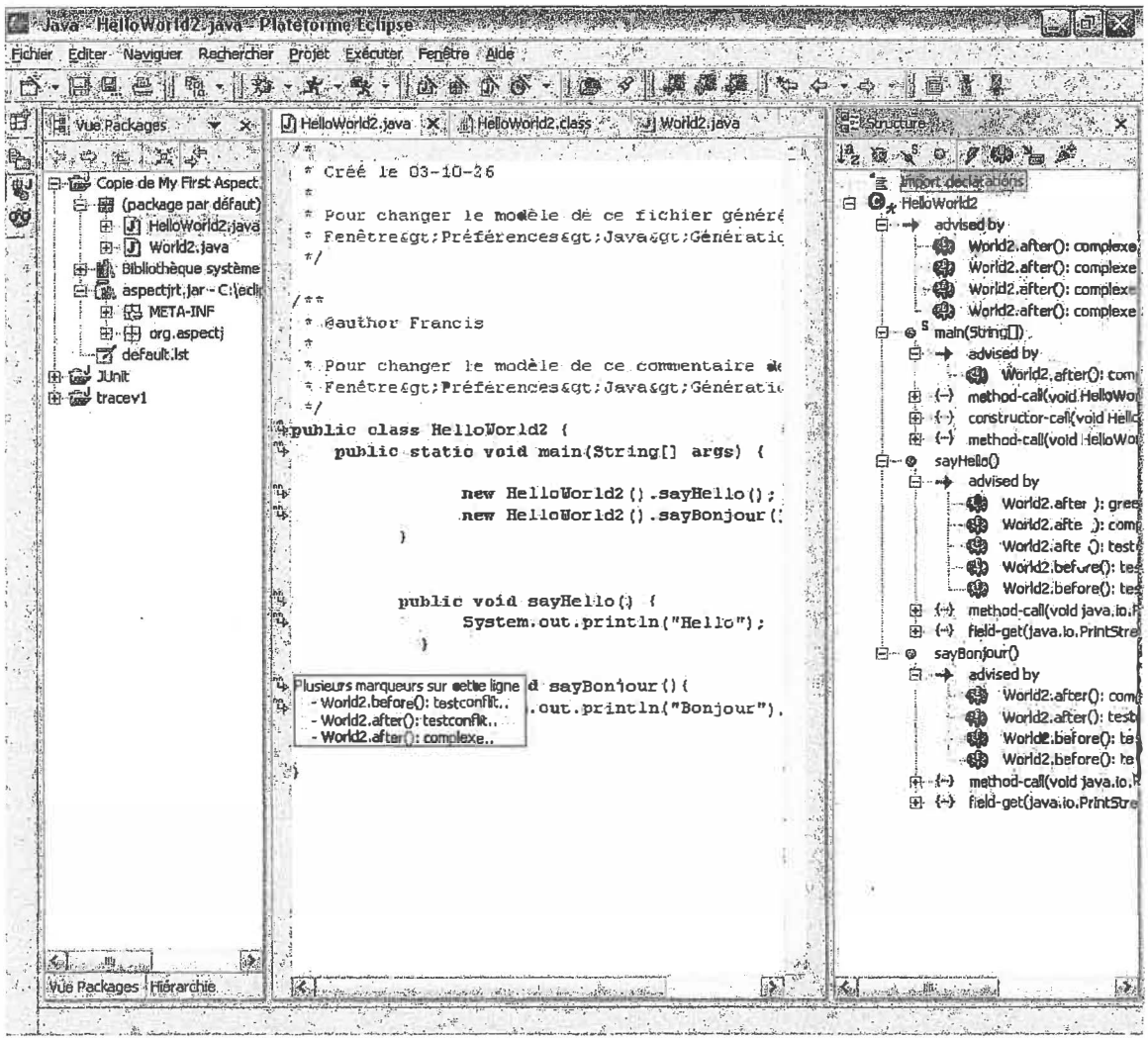

Figure 8: Exemple d'interface avec AJDT incorporé dans Eclipse

Chacune des approches abordées précédemment facilite, à un certain degré, la détection des conflits. Cependant, elles restent encore limitées et nécessitent une intervention

humaine pour détecter et résoudre les conflits. L'intervention du programmeur reste indispensable pour saisir et comprendre la subtilité de l'interaction entre les aspects et prendre une bonne décision, comme mentionnée dans les travaux de Douence [DOU 02] et ceux de Hanneman [HAN 03], quant à la résolution des conflits qui en découlent.

3.3 Connexions Aspects - Classes

Les points de jointure (joinpoint) et les points de coupure (pointcut) constituent des éléments très importants dans la POA. Il s'agit de constructions de base permettant la connexion entre les aspects et les objets. Le degré de séparation des préoccupations dépend grandement des types de point de jointure disponibles [CHI 01]. Nous donnons, dans ce qui suit, un résumé des principaux points de jointure et points de coupure.

Points de jointure : Ils représentent des points spécifiques et sont définis dans l'exécution du programme. Il s'agit de méthodes bien précises. Les points de jointure sont relativement limités et ne sont pas d'une grande finesse. Le Tableau IV décrit tous les points de jointure d'AspectJ [LAD 02]. Actuellement, ils sont trop rudimentaires pour permettre une analyse dynamique de l'interaction entre les aspects et les classes. Cela limite le travail à une analyse statique comme mentionné dans [HAN 03].

Tableau IV : Points de jointure

Points de jointure	Définition
Method call	Quand une méthode est appelée, n'inclut pas les supers appels.
Method execution	Quand le code d'une méthode est exécuté.
Constructor call	Lors de l'appel du constructeur, n'inclut pas «this()» ou les super constructeurs.
Initializer execution	Quand l'initialiseur non statique d'une classe fonctionne.
Constructor execution	Quand le code d'un constructeur actuel est exécuté, après «this()» ou le super constructeur.
Static initializer execution	Quand l'initialiseur statique de la classe fonctionne.
Object pre-initialization	Avant le code d'initialisation d'un objet. Cela inclut le temps entre le départ du premier appel de constructeur et le départ du constructeur du parent. L'exécution de ce point de jointure englobe les points de jointure du code trouvé dans les appels du «this()» et «super()» constructeur.
Object initialization	Quand le code d'initialisation d'un objet fonctionne. Cela inclut le temps entre le retour du constructeur du parent et du retour du premier constructeur appelé. Cela inclut toutes les initialisations dynamiques et les constructeurs créés par l'objet.
Field reference	Quand un champ non final est référencé.
Field assignment	Quand un champ est assigné.
Handler execution	Quand une exception est soulevée.

Points de coupure : Il s'agit de constructions qui permettent de désigner des points de jointure et recueillir le contexte spécifique à ces points. Ceci constitue une des forces des aspects. Le Tableau V fournit une liste des points de coupure disponibles dans AspectJ [LAD 02]. Dans la définition des points de coupure, certains peuvent être

nommés et d'autres *anonymes*. Il est également possible d'utiliser les opérateurs *OU* (||), *ET* (&&) ou *NON* (!) pour définir des points de coupure combinés.

Tableau V : Points de coupure

Points de coupure	Définition
Call to methods and constructors	Capture l'exécution des points après l'évaluation des arguments de la méthode, mais avant l'appel lui-même.
Execution of methods and constructors	Capture l'exécution des méthodes. Par rapport au point de coupure d'appel, les points d'exécution représentent le corps de la méthode ou du constructeur.
Field-access	Capture les lectures et les écritures dans un champ d'une classe. Cela inclut les procédures «get» et «set».
Exception-handler	Capture l'exécution d'un indicateur d'exception.
Class-initialization	Capture l'exécution de l'initialisation d'une classe statique, le code spécifié dans le bloc statique à l'intérieur de la définition de la classe.
Lexical-structure-based	Capture tous les points de jointure à l'intérieur d'une classe ou d'une méthode (within).
Control-flow-based	Capture les points de coupure basés sur les autres flux de contrôle (control flow) des autres points de coupure. Inclusion des appels à la méthode spécifiée : cflow. Exclusion des appels à la méthode spécifiée : cflowbelow.
Self-, target- and arguments-type	Capture les points de jointure basés sur les objets eux-même (self-object), objet cible (target-object) et les arguments de type. Ce sont les seules constructions qui capturent le contexte des points de jointure.
Conditional-test	Capture les points de jointure basés sur des conditions. (if)

Les différents éléments contenus dans les deux précédents tableaux sont importants. Ils constituent la base d'AspectJ et représentent les fondements des aspects. Leur connaissance est indispensable pour toute approche de résolution de conflits.

3.4 Description des types de conflits

Il existe divers types de conflits pouvant se produire entre les différents aspects d'un programme. Les travaux du workshop sur l'analyse des logiciels orientés aspect [HAN 03] ont permis d'identifier et de discuter quatre principales catégories de conflits. Nous présentons sommairement ces différentes catégories dans ce qui suit.

Spécifications transverses : Il s'agit de la façon dont les aspects s'intègrent aux classes. L'utilisation actuelle des points de jointure peut mener à deux sous-types de problèmes.

- Point de jointure accidentelle : Il peut y avoir un chevauchement des points de jointure à cause de l'utilisation des «*». Les «*» sont vagues et permettent de toucher à une multitude de points de jointure. On peut, ainsi, effectuer des points de jointure accidentelle;

- Récursivité accidentelle : Il s'agit du cas où une jonction entraîne un appel récursif. Lorsque le point de jointure est atteint pour une première fois, l'aspect s'exécute. Si le même point de jointure est inclus dans le code de l'aspect, cela

amène l'aspect à s'exécuter une nouvelle fois. Ceci peut conduire à une multitude d'appels récursifs.

Conflits Aspect – Aspect : Ce type de conflit se présente lorsque plusieurs aspects se retrouvent dans le même système. Ils peuvent ainsi entrer en conflit les uns avec les autres.

- Exécution conditionnelle : L'application d'un aspect est conditionnelle à l'exécution d'un autre;
- Exclusion mutuelle : L'exécution d'un aspect implique l'exclusion de l'exécution d'un autre;
- Ordonnancement : Un ordre doit être défini lorsque plusieurs aspects partagent le même point de jointure;
- Ordonnancement dépendant du contexte dynamique : L'ordre des aspects dépend du contexte dynamique dans lequel ils s'exécutent;
- Échanges et conflits aux niveaux de la spécification : Ce type de conflit se produit lorsqu'un aspect introduit un élément pouvant compromettre les spécifications des autres.

Conflits Base – Aspect : Ce type de conflit se présente lorsque la base fait appel à un aspect invasif. Il s'agit du cas dans lequel la base a connaissance de l'aspect et doit

interagir avec ce dernier. La base n'est plus indépendante de l'aspect. Cela conduit à une dépendance circulaire entre la base et l'aspect.

Conflits Préoccupation – Préoccupation : Une préoccupation peut affecter une autre. Par exemple, un aspect traitant de la journalisation (log) peut entrer en conflit avec un autre s'occupant de la synchronisation. Il s'agit, dans ce cas, de conflits concernant les préoccupations.

- Changement de fonctionnalités : L'ajout d'un aspect peut empêcher d'atteindre un point de jointure requis par un autre aspect. Ce point ressemble à l'exclusion mutuelle, mais concerne plus spécifiquement les préoccupations;
- Comportement inconsistant : Cela s'applique lorsqu'un aspect détruit ou manipule l'état d'un autre aspect ou d'une préoccupation de base.
- Anomalies de composition : C'est un problème qui peut survenir avec la substitution de sous-types. Un aspect peut, par exemple, altérer ou substituer un élément qu'il partage avec un autre aspect.

3.5 Description des niveaux de conflits

Pour permettre une meilleure classification des conflits, nous avons introduit deux niveaux différents de conflits [TES 04a]. Le premier niveau est celui des conflits directs. Ce type de conflit concerne principalement les conflits reliés à une relation directe entre

plusieurs aspects. Par exemple, deux aspects partageant le même point de jointure ou un aspect ayant un point de jointure inclut dans le code d'un autre aspect. Ces types de conflits sont aisément identifiables comparativement à l'autre niveau de conflit, lequel est le niveau indirect.

En premier lieu, nous nous sommes attardés sur les conflits de niveau direct entre les aspects, car les relations indirectes sont difficilement détectables. Comme leur détection est ardue, la résolution des conflits l'est aussi. Nous avons, cependant, identifié une piste de recherche pour résoudre le problème des conflits de niveau indirect [TES 04b]. L'idée consiste à utiliser plusieurs diagrammes UML à la fois tels que les diagrammes de séquence, d'activité, de collaboration et d'états-transitions. L'idée consiste à extraire le chemin des données pour pouvoir ainsi déterminer quand, dans le processus, les aspects interviennent et ordonnancer de cette façon leur ordre d'interaction. Cela permettrait de savoir qu'un aspect X a agi par exemple sur certaines données et qu'un aspect Y a manipulé ces données par la suite. En clair, l'usage du diagramme d'états-transitions permet de déterminer l'ordre d'exécution des aspects (séquencement dans l'exécution des événements) alors que le diagramme de séquence permet de connaître le moment de l'exécution. Tout ceci facilite la capture de l'information relative au comportement et aux interactions des aspects.

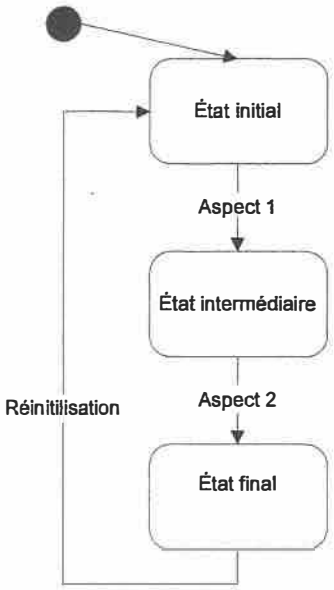

Figure 9 : Diagramme d'état

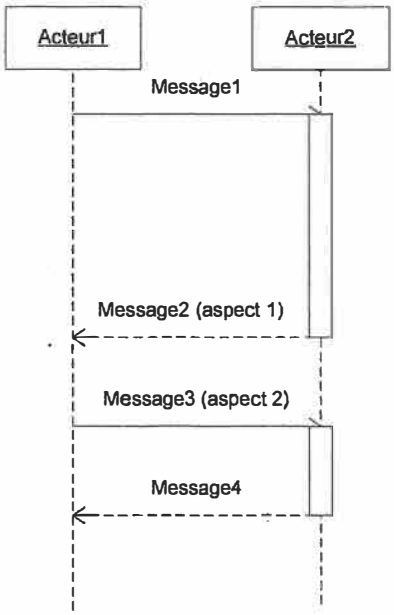

Figure 10 : Diagramme de séquence

Les Figure 9 et Figure 10 représentent un exemple d'interaction entre les deux aspects retenus. Les aspects ne partagent pas de point de jointure commun, mais nous pouvons clairement identifier que le premier aspect peut avoir un impact important sur le second aspect. Dans cet exemple simple, si le premier aspect n'est jamais atteint, le second ne le sera pas non plus. Il faut que le premier aspect puisse agir pour permettre à l'objet d'atteindre un état intermédiaire qui est par la suite manipulé par le second aspect pour finalement atteindre l'état final. Par contre, pour ce type de détection, il faut une représentation complète du système en UML pour déterminer les types de conflits possibles. En utilisant ces diagrammes, nous pouvons utiliser le concept de table de connexion que nous avons introduit dans ce travail (présenté dans ce qui suit). À partir du moment que l'information se retrouve dans nos tables de connexion, le reste de la démarche de détection et de résolution est le même pour les deux niveaux de conflits. Ces tables sont la base de tout notre processus de détection.

Nous évaluons l'impact potentiel par les opérations précédentes et l'état du système. Dans ce cas-ci, nous parlons de détection dynamique des conflits entre aspects, car cette interaction ne peut être visible et décelée que lors de l'exécution de l'objet. En fait, il faut qu'un aspect altère l'état d'un objet nécessaire pour un second aspect et ceci peut se dérouler avec un délai temporel. Dans ce cas, le premier aspect a un impact indirect sur le second. En bref, il s'agit d'identifier les impacts et les exécutions des aspects via leurs

interactions avec les objets du système. On détecte les conflits dans une pile d'exécution d'opérations.

3.6 Réflexion sur les travaux actuels

La détection de conflits entre les aspects est un sujet relativement récent et reste faiblement couverte. Il suscite de plus en plus l'intérêt des chercheurs. Cette section présente une évaluation comparative entre notre technique et les rares approches proposées dans la littérature à ce sujet. Ces dernières sont, pour le moment, assez pointues. D'une manière générale, Douence et al. ont élaboré une représentation formelle de la problématique [DOU 02], Clement et al. un outil de développement [CLE 03], Storzer et al. une approche basée, d'une part, sur l'analyse des traces d'exécutions dans [STO 03a] et, d'autre part, sur l'analyse de l'interférence dans [STO 03b]. Notre démarche est relativement plus précoce. Elle supporte la prédiction des conflits relativement tôt dans le processus de développement. La comparaison que nous présentons dans cette section tente de mettre en évidence les caractéristiques de ces approches, leurs points forts ainsi que leurs points faibles. Une synthèse de la comparaison est donnée dans le Tableau VI. Douence propose une approche algébrique. Elle partage, dans la manière d'aborder les conflits, plusieurs similitudes avec l'approche que nous avons adoptée. La technique proposée par cet auteur est cependant assez complexe comparativement à la solution que nous proposons. Nous nous limitons à une

sémantique plus simple pour représenter le même type de situations. Par ailleurs, la solution que nous proposons est relativement plus précoce et supporte une grande partie du processus de développement. Elle commence par l'analyse des modèles (une des originalités de ce travail) de conception et se poursuit jusqu'au code.

La méthode proposée reste très proche de ce qui se fait déjà dans les environnements de développement orientés objet. Les concepts que nous introduisons ne sont pas totalement inconnus. Ils représentent des adaptations de concepts largement éprouvés. Lors de la modélisation, les aspects sont traités comme des tables de base de données ayant des relations n à n. Les tables de connexion sont identiques à celles que nous retrouverons pour une relation n à n entre 2 tables d'un modèle entité - relation. Nous croyons qu'en utilisant les T_{AC} (table de connexion entre un aspect A et une classe C) pour générer les règles représentant la sémantique des relations, nous offrons une méthode plus simple, plus visuelle et avec des notions plus facile à assimiler pour une intégration dans les outils de développement actuels. Par ailleurs, l'avantage de la modélisation proposée et de la cueillette précoce d'information est de créer un *Pointcut Wizard* apportant une aide au programmeur dans la détection des conflits lors de la création des diagrammes de classes incluant les relations aspects - classes. La détection précoce des conflits offre plusieurs avantages (coût, temps, qualité). De plus, les spécifications au niveau du modèle permettent de créer un squelette de code tout en simplifiant la programmation des connexions des aspects. Les règles engendrées par analyse des modèles sont faciles à

évaluer. L'utilisation de concepts mathématiques simples et puissants tels que les ensembles facilite l'implémentation de la technique. Par ailleurs, le cadre que nous proposons suit les recommandations formulées dans [KIC 03] et [HAN 03].

Tableau VI : Comparaison des techniques

	Douence [DOU 02]	AJDT [CLE 03]	Analyse de trace [STO 03a]	Analyse d'Interférence [STO 03b]	Notre approche [TES 04d]
Analyse	Statique	Non applicable	Dynamique	Statique	Statique et ouverture pour dynamique
Modélisation	Non applicable	Non applicable	Non applicable	Diagramme de classe	Diagramme de classe et UML modifié
Analyse code	Non applicable	Non applicable	Exécution du code	Extends et implements	Pointcut et joinpoint
Méthode formelle	Oui	Non	Non	Oui	Oui
Conflits gérés	- Interaction aspect - aspect - Visibilité	Aucun mais aide pour la programmation	- Superposition - Amputation - Changement de comportement	- Interférence - Héritage	- Conflit aspect - aspect - Aide à la programmation
Simplicité	Méthode lourde et très théorique	Oui	Méthode simple mais lourde à exécuter	Oui	Oui
Technique visuelle	Non	Oui	Non	Oui	Oui
Outils d'aide à la programmation	Non	Oui, liste des pointcuts et les joinpoints illustrés	Non	Non	Oui, tableau d'information
Détection et/ou résolution des conflits	Détection et résolution	Non applicable	Détection	Détection	Détection précoce et résolution Ordonnancement
Évolution	- Extension linguistique - Modélisation du code - Instanciation complète AspectJ	- Pointcut wizard - Pointcut reader - Utilisation des filtres de composition	- Utilisation en conjoncture avec des méthodes statiques	- Filtre de composition	- Intégration de l'analyse des traces - Raffinement des outils et de la méthode - Implémentation

Chapitre 4

Méthode de détection

4.1 Description détaillée

4.1.1 Principes de base

Ce chapitre décrit l'essentiel de notre méthode de détection. Nous avons élaboré notre technique de détection en utilisant une multitude de concepts déjà présents et éprouvés dans le but d'avoir un maximum de réutilisation. Les principes de base que nous avons retenus dans notre démarche sont les suivants : une approche couvrant une large partie du processus de développement, la création d'une modélisation de la connexion entre les aspects et les classes, la création de tables de connexion, l'utilisation d'un processus d'inférence ainsi que la création d'une représentation mathématique. Tous ses éléments créent un cadre général et formel pour la détection des conflits.

4.1.2 Principales étapes

L'approche adoptée tente de couvrir (à long terme) une grande partie du processus de développement orienté aspect. Notre objectif est de supporter la détection des conflits le plus tôt possible et d'en assurer un suivi continu. L'idée est d'offrir un certain niveau de prédiction de l'impact engendré par l'insertion d'un aspect et de renseigner le

programmeur sur le comportement anticipé. Par ailleurs, nous tenons à développer une méthode simple et aussi conviviale que possible pour faciliter son implémentation et éventuellement son intégration dans les environnements de développement actuels. Durant l'élaboration des principes de notre approche, nous avons tenu compte de certaines recommandations formulées par Gregor Kiczales [KIC 03] et dans les discussions de workshop [HAN 03]. Ces recommandations ont essentiellement pour objectif d'établir des lignes directrices à suivre pour faciliter l'intégration et la démocratisation de la POA. Ces conseils sont en particulier liés au fait que la POA ne représente pas en soit une solution à tous les problèmes. Il faut donc établir les ponts avec les technologies actuelles (et éprouvées) et éviter de tout réinventer. Il s'agit essentiellement d'adapter les outils actuels pour supporter les aspects et les incorporer dans les processus orientés objets. La Figure 11 permet d'illustrer l'intégration des principales étapes de notre méthodologie au processus de développement orienté aspect.

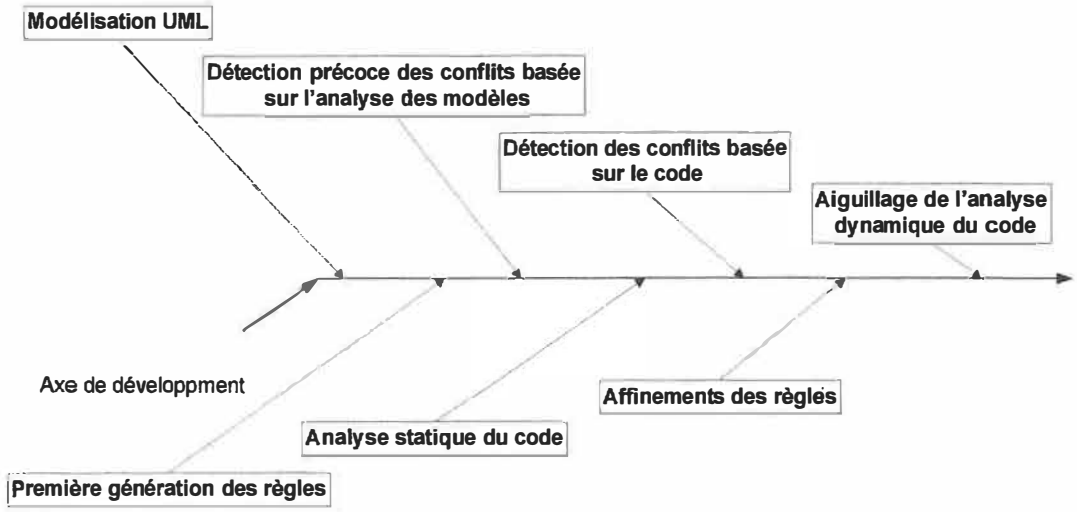

Figure 11 : Approche et processus de développement

Nous proposons, dans le cadre de notre approche, une simple extension du diagramme de classes UML permettant la modélisation des aspects et leurs connexions aux classes [TES 04b]. Cette modélisation, inspirée de plusieurs travaux sur le sujet, nous permet de gérer de façon précoce les conflits potentiels engendrés par les aspects ayant des points de jointure communs. Par ailleurs, la modélisation retenue dans le cadre de notre approche est assez souple et extensible pour pouvoir intégrer des évolutions intéressantes dans ce domaine. Notre objectif principal n'est pas de créer un outil complet de modélisation, mais plutôt de définir les paramètres essentiels et nécessaires permettant de supporter une analyse formelle des modèles en vue d'une détection précoce de conflits. En effet, plusieurs travaux de recherche intéressants se concentrent plus sur la modélisation des aspects et leur intégration aux classes [ALD 03], [FAY 03].

L'extension du diagramme de classes UML nous permet d'introduire le concept de table de connexion aspect - classe. Cette table contient des informations de base reliées à la connexion aspect - classe nécessaires pour permettre une première analyse. Par la suite, l'analyse statique du code permettra d'affiner les informations recueillies initialement dans les modèles, en particulier, concernant la nature des méthodes utilisées (modificateur, visiteur) ainsi que l'ordonnancement de l'exécution des aspects. Nous travaillons actuellement sur le développement d'un framework, composé de plusieurs outils, supportant ces différentes étapes et pouvant facilement s'intégrer aux environnements actuels. Il s'agit essentiellement d'apporter une aide intéressante aux programmeurs en leur permettant de gérer les conflits potentiels. Par ailleurs, ces outils nous permettront ultérieurement de piloter une analyse dynamique dont le but est d'évaluer le comportement des aspects lors de l'exécution et de détecter les conflits qui peuvent échapper à l'analyse statique. Cette analyse sera aiguillée, en fonction des informations recueillies et portera sur des zones sensibles du code. Ces informations peuvent également être utilisées pour piloter le processus de test (génération de séquences de test, vérification, etc.).

4.1.3 Modélisation de la connexion des aspects aux classes

Plusieurs auteurs tels que Aldawud [ALD 03] et Fayad [FAY 03] fondent leurs travaux sur le pouvoir d'extension offert par le langage UML. Les extensions proposées

permettent d'illustrer les aspects ainsi que leurs connexions aux classes. L'OMG (Object Management Group) a défini UML comme un langage de modélisation ouvert et extensible [OMG 03]. L'utilisation des stéréotypes pour la représentation des aspects et de leurs éléments est très répandue. Nous concentrons nos efforts sur le diagramme de classes. Essentiellement, nous partons du principe qu'il y a une inversion des dépendances entre les classes et les aspects comme souligné par Nordberg [NOR 01]. Ce sont les aspects qui utilisent les classes et non l'inverse [OWC 05]. Les aspects peuvent être considérés comme des composants de haut niveau utilisant des objets stables.

La Figure 12 donne un exemple de la modélisation UML retenue intégrant deux aspects et une classe. Cette figure illustre le fait que nous privilégions une relation «aspect vers classes» et non l'opposé. Les aspects agissent sur les classes et non le contraire. Notre modélisation tient donc compte de cette orientation de manipulation de l'information. Les aspects sont connectés aux classes via un lien unidirectionnel. La représentation retenue traduit le courant de pensée des travaux actuels sur la modélisation. Nous utilisons les stéréotypes «aspect» et «advice» pour la représentation des aspects et des advices. Les deux tables du modèle caractérisent les deux relations qui existent dans l'exemple entre les deux aspects et la classe considérée. Nous offrons, en fait, deux niveaux de visualisation. Le premier niveau porte sur la représentation aspect - classe. Ce niveau permet de connaître toutes les connexions que possède un aspect avec les classes du modèle. Le second niveau porte sur la représentation classe - aspect. Ce

niveau permet de connaître toutes les connexions d'une classe avec l'ensemble des aspects du programme. Ainsi, lors de la création des liens, nous pouvons définir un ordonnancement des aspects durant la phase de conception. Pour chaque aspect présent dans le modèle, l'ensemble des informations relatives à toutes les relations qu'il entretient avec les classes du modèle est présenté aux deux niveaux : Individuel (Aspect - Classe) et Global (Aspect - Classes).

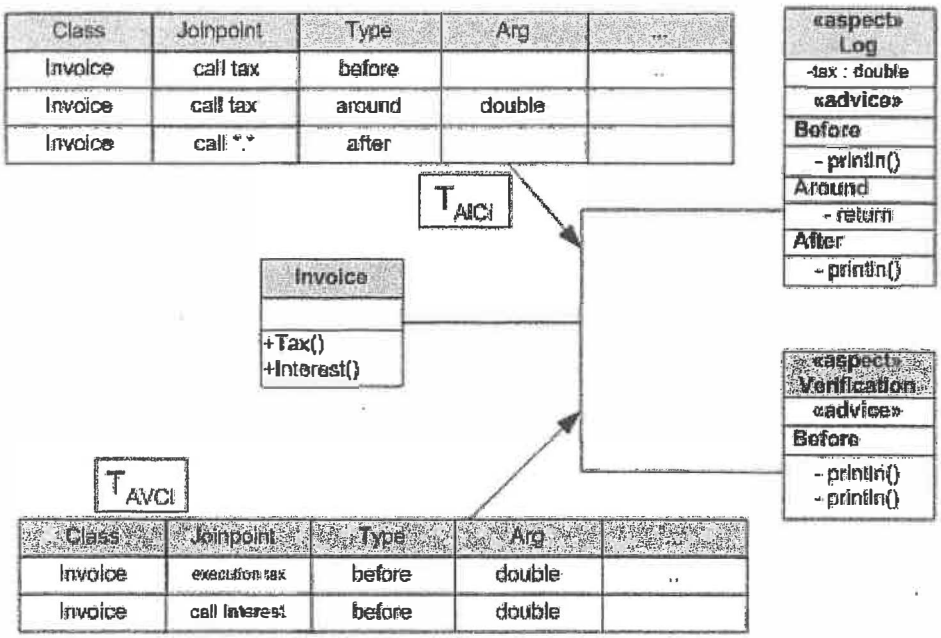

Figure 12 : Relation entre aspects et classes

En plus, pour faciliter la modélisation et pour établir la base de notre technique de détection, nous définissons le type de connexion pendant la création du lien. C'est-à-dire que lors de la modélisation, lorsque nous relions un aspect à une classe, nous devons alors spécifier le lien. Tout comme les objets, les aspects sont compartimentés. Les

divers compartiments d'un aspect sont ses attributs et les stéréotypes «advice» avec leurs méthodes. La modélisation adoptée permet de tenir compte des liens multiples qui peuvent exister entre un aspect et plusieurs classes du modèle. Ainsi, un aspect A_i peut être connecté à plusieurs classes. C_1, ... , C_n simultanément, ce qui correspond à la définition et au rôle d'un aspect. Plusieurs aspects peuvent être connectés à la même classe (une ou plusieurs classes partagées).

Pour connecter un aspect à une classe, nous créons un lien de connexion. Ce lien représente la relation R_{AC} existante entre un aspect A et une classe C. Pendant la création du lien, nous définissons l'ensemble $S(R_{AC})$ qui représente ce que nous appelons la sémantique de la relation R_{AC}. Ces informations représentent les éléments de base de la relation. Ces éléments de base seront décrits dans une section ultérieure. Ceux-ci sont insérés dans plusieurs tables T_{AC}, lesquelles sont appelées table de connexion. Chaque table T_{AC} caractérise la relation entre un aspect A et une classe C. Par exemple, le lien de l'aspect *Verification* avec la classe *Invoice* sur la méthode *Tax* de l'exemple donné par la Figure 12 précédente peut être décrit par le tableau suivant :

Tableau VII : Descriptions des liens de connexion

Aspect	Classe	Joinpoint	Argument	Type de joinpoint	Advice	Pointcut	Type pointcut	Précé-dence	Type opération
Verifica-tion	Invoice	Tax	Double	Method execution	Before	-	Execution of method	-	Accès

Les tables de connexion supportent deux niveaux de visualisation. Le premier niveau est la représentation aspect - classe (T_{A*}) comme dans l'exemple ci-haut. Ce niveau permet de visualiser toutes les connexions d'un aspect avec une classe. Le second niveau est la représentation classe - aspect (T_{*C}). Ce niveau permet une visualisation de tous les liens d'une classe avec les aspects associés. Ainsi, lors de la création des liens, nous pouvons définir un ordonnancement des aspects durant la phase de conception. Pour chaque aspect présent dans le modèle, toute l'information reliée à la connexion peut être représentée sous deux types : Individuel (Aspect - Classe) ou Global (Aspect – Classes (2 et +)). Comme illustré à la Figure 12, la table T_{AlCi} représente la relation R (Aspect *Log* : Classe *Invoice*) et la table T_{AvCi} représente les relations R (Aspect *Verification* : Classe *Invoice*). Le contenu de ces tables représente la base sur laquelle notre approche est fondée. Chaque enregistrement d'une table est associé à un point de coupure. Cet enregistrement synthétise toute l'information nécessaire et pertinente sous forme d'une ligne simple à analyser et à lire. Les colonnes sont les différents éléments (ou sémantique) de la relation. Ces éléments nous renseignent précisément sur le mode de connexion des aspects aux classes et sur le comment de leur exécution (contrôle).

Chaque enregistrement de la table est associé à un point de coupure synthétisant toute l'information sur une ligne. Les colonnes de la table représentent les divers éléments de connexion. Ces éléments nous renseignent, de façon précise, sur le mode de connexion

des aspects aux classes. Par ailleurs, cela permet d'appréhender la complexité des connexions et de percevoir rapidement, pour les cas simples, la concurrence occasionnée par des aspects pointant, par exemple, sur des points de jointure communs. Il faut pouvoir distinguer les interactions souhaitées de celles qui peuvent être accidentelles et donc indésirables. Notre modèle apporte déjà une aide intéressante à la gestion précoce des interactions. Par ailleurs, le contenu des tables nous permet de générer des règles formelles décrivant les relations aspects - classes. L'évaluation des règles nous permet de détecter les conflits éventuels.

L'analyse des modèles rend possible une meilleure compréhension des interactions possibles entre les aspects et les classes et les conflits qui résultent de ces interactions. Les mécanismes adoptés à ce stade sont simples et aident le développeur, tôt dans le processus de développement, à appréhender ses modèles et détecter visuellement certains des conflits. Le reste est supporté, tel que présenté ultérieurement, par une analyse formelle des modèles.

4.1.4 Relations entre aspects et classes : éléments de base

Nous donnons, dans le Tableau VIII, la liste des éléments de base que nous considérons. Ils représentent un ensemble pertinent d'informations caractérisant les connexions entre aspects et classes et nécessaires à la gestion des conflits. La cueillette de l'information contenue dans les tables se fait tout le long du processus de développement, depuis les

modèles jusqu'au code. Cette information est synthétisée essentiellement pour les deux principaux acteurs : le développeur et les algorithmes de détection des conflits. L'information réunie, sous une forme graphique, constitue une synthèse intéressante qui offre déjà (et telle quelle) un support au développeur. À partir de cette synthèse et du résultat de l'analyse, le développeur peut apporter les corrections nécessaires aux modèles.

Tableau VIII : Éléments de base

Nom		Description
Aspect	(A_i)	Aspect impliqué dans la relation.
Classe	(C_i)	Classe incluant le point de jointure.
Joinpoint	(J_i)	Point de jointure impliqué.
Argument	(Arg_i)	Type d'argument pour les méthodes surchargées.
Type joinpoint	(TJ_i)	Type de point de jointure, information supplémentaire.
Advice	-	Les advices *before*, *around*, *after* et variantes.
Pointcut	(P_i)	Nom du pointcut (anonyme ou nommé).
Type pointcut	(TP_i)	Un des types de pointcut, information supplémentaire.
Précédence	(O_i)	Ordre de précédence ou ordre d'exécution des aspects.
Type opération	(TO_i)	Type d'opération : accès ou modification.

4.2 Processus d'inférence

La démarche adoptée dans la détection des conflits suit un processus itératif. Ces différentes étapes sont illustrées dans la Figure 13. Le processus démarre avec les

modèles et se poursuit jusqu'au code. Les modèles sont analysés en vue d'extraire une synthèse de la sémantique des relations entre les aspects et les classes. Cette synthèse est traduite sous forme de règles formelles. L'analyse de la base de règles nous permet de détecter les conflits entre aspects. Ceci permet de revenir sur les modèles pour apporter les corrections nécessaires. Lors de la phase de programmation et grâce aux détails qu'elle procure, l'analyse statique du code permettra d'affiner la base de règles. Cet affinement permet de renforcer le processus d'inférence pour la détection de conflits présents dans le code. Par ailleurs, les différentes informations recueillies dans les règles permettront de mieux piloter une analyse dynamique. Un accent particulier, par exemple, sera mis sur certaines parties du code susceptibles d'être à l'origine de problèmes engendrés par les conflits. De cette façon, l'analyse dynamique serait complémentaire à l'analyse statique et le processus de détection des conflits serait continu.

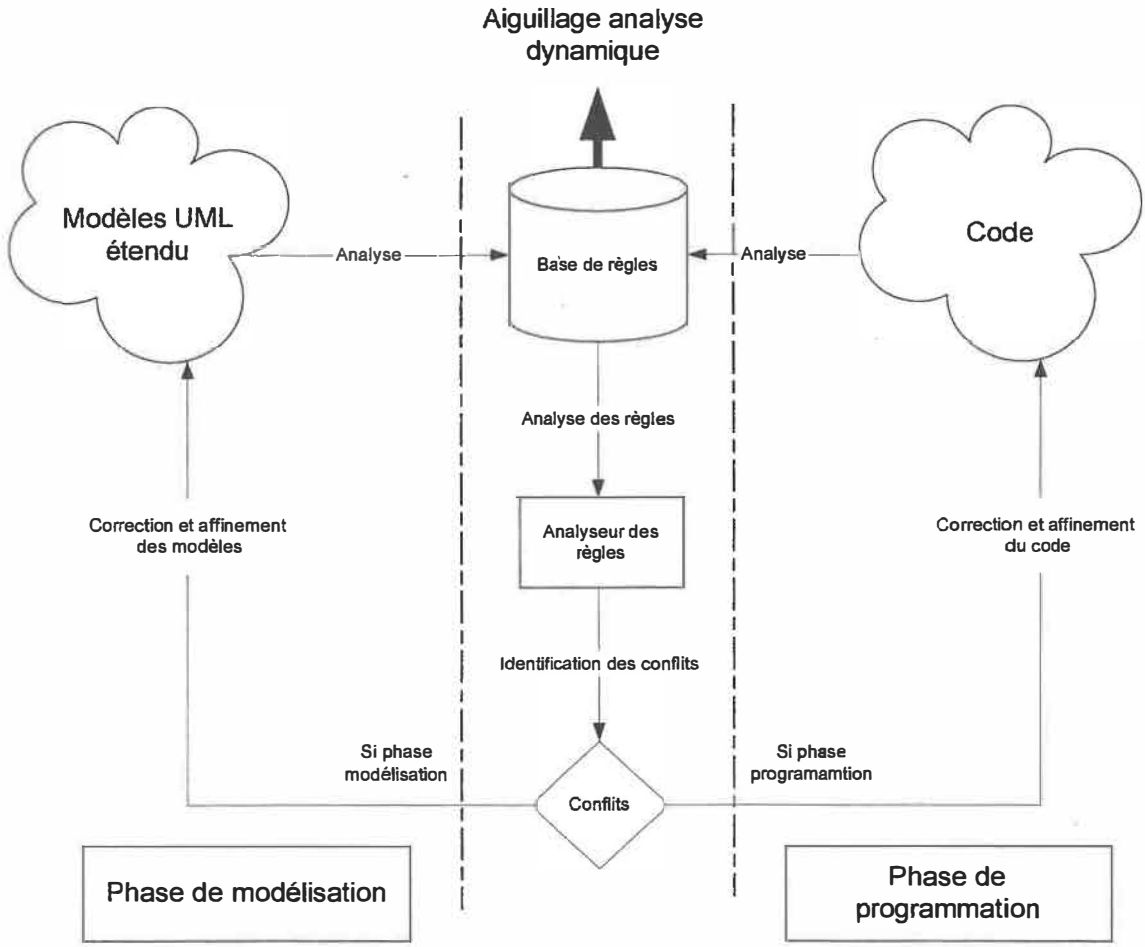

Figure 13 : Processus d'inférence

4.3 Cadre formel

Nous avons adopté, dans le cadre de notre approche, une notation formelle nous permettant de représenter les aspects et leurs connexions aux classes. Cette notation procure un cadre formel et constitue une base solide pour notre approche de détection de conflits. L'idée de base consiste à utiliser la théorie des ensembles. Dans ce contexte, nous pouvons par exemple représenter un point de jointure comme étant un élément d'un ensemble plus grand, c'est-à-dire une classe.

4.3.1 Formalisation à l'aide des ensembles

L'approche adoptée est fondée sur une description formelle des relations qui existent entre les aspects et les classes. La description formelle des aspects est un sujet récent et faiblement couvert comme mentionné dans les travaux de Douence *et al.* [DOU 02]. Il s'agit d'une piste de recherche qui présente un potentiel intéressant [AND 01]. Dans nos travaux actuels, nous nous concentrons principalement sur les conflits de niveau direct. Les aspects utilisent les points de jointure et de coupure pour définir leurs connexions aux classes. Nous traduisons ces relations (sémantique présentée précédemment) par une représentation mathématique compartimentée selon les divers éléments de base. Chacun des éléments pouvant être représenté sous la forme d'un ensemble. Cela peut se représenter formellement sous la forme : $J_k \subseteq C_l$, c'est-à-dire un point de jointure K (J_k) est inclus dans la classe L (C_l). Nous définissons l'ensemble J comme étant l'ensemble

de tous les points de jointure possibles. Plus précisément, le domaine de cet ensemble est constitué d'éléments tels que les appels ou exécutions de méthodes, les constructeurs, les accès en lecture/écriture, les exceptions levées, etc. Il inclut tous les points de jointure disponibles. L'ensemble P (programme) représente l'ensemble des classes disponibles. Le domaine de cet ensemble est défini comme étant la totalité des classes disponibles dans le système sur lesquelles on peut greffer un aspect. Les aspects peuvent être considérés comme des ensembles incluant une ou plusieurs classes. Nous représentons la liaison d'un aspect A_i avec un point de jointure J_k à l'aide de l'opérateur d'intersection $\cap$. Nous obtenons donc : $(A_i \cap J_k) = L_{ik}$. Nous obtenons ainsi un nouvel ensemble contenant des liaisons (L_{ik}) représentant le lien entre un aspect i et un point de jointure k. Considérons un aspect *A* relié à une méthode *m* définie dans une classe *C*. Nous représentons cette liaison par :

Liaison (A, J, C) = ((Aspect (A) $\cap$ Point de jointure (J)) $\subseteq$ Classe (C)).

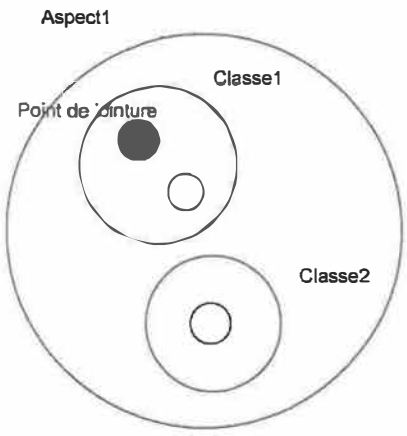

Figure 14 : Illustration de la liaison via des ensembles

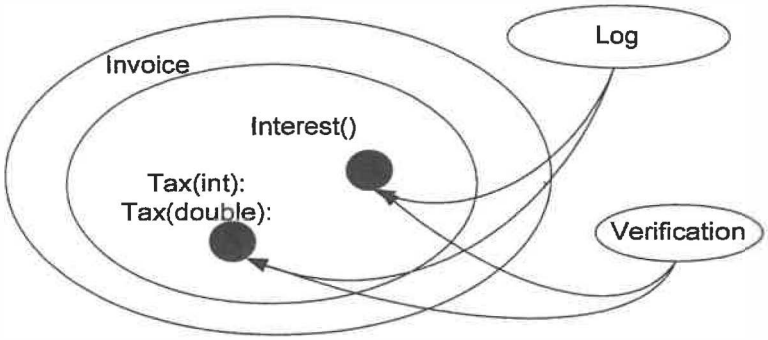

Figure 15 : Liaisons

4.3.2 Notation

Pour la représentation des ensembles de la Figure 15, nous avons un premier ensemble qui est le programme dans lequel est inclus l'ensemble *Invoice*. À l'intérieur de ce sous-ensemble, nous retrouvons deux points de jointures inclus (⊆) dans la classe. Les deux aspects sont en lien avec ces points et l'intersection (∩) de l'aspect avec un point de jointure crée un enregistrement dans la table de la relation aspect - classe de la Figure 12. L'intersection du point de jointure avec l'aspect crée un ensemble qui est inclus dans une classe. Pour un point de coupure *within*, par exemple, cela se traduit par la Liaison (A, ∀J, C)= ((Aspect (A) ∩ Tout Point de jointure (∀J)) ⊆ Classe (C)). Avec le point de coupure *within*, nous capturons tous les points de jointure inclus dans la classe. Alors, nous utilisons la représentation mathématique ∀J pour exprimer que tous les points de jointure sont capturés. La notation formelle adoptée est à la base du processus de construction des règles. Nous prenons en compte tous les éléments de base : aspect, point de jointure et classes. Ce formalisme permet de déterminer les éléments communs

partagés par les aspects d'un programme. Plus l'ensemble des éléments partagés est grand, plus il y a de risques de conflit. Deux enregistrements ne comportant aucun élément commun sont moins susceptibles d'entrer en conflit. Il reste néanmoins la possibilité d'un conflit indirect, qui nécessite une analyse plus approfondie.

4.3.3 Règles de base

Considérons maintenant deux aspects A_i et A_j qui sont respectivement reliés à deux points de jointure J_k et J_m de la même Classe C_l. Ceci est représenté par les expressions suivantes :

Liaison $(A_i, J_k, C_l) = ((A_i \cap J_k) \subseteq C_l)$ et Liaison $(A_j, J_m, C_l) = ((A_j \cap J_m) \subseteq C_l)$.

À partir de ces deux liaisons, nous pouvons illustrer les différentes règles suivantes :

- $(i \neq j)$ et $(k = m)$: il y a un conflit direct entre A_i et A_j, car ils sont en lien avec le même point de jointure.
- $(i \neq j)$ et $(k \neq m)$: il n'y a pas de conflit direct entre A_i et A_j, ils font partie du même ensemble C_l.
- $(i = j)$ et $(k \neq m)$: un seul aspect avec deux points de jointure distincts.
- $(i = j)$ et $(k = m)$: un seul aspect qui est en liaison avec un point de jointure.

L'outil d'extraction que nous avons développé est encore au stade de prototype expérimental. Il nous permet de manipuler les modèles UML étendus considérés et d'en extraire l'information nécessaire pour l'enregistrer dans les tables de relations aspect - classe. Le contenu de ces tables nous permet de générer les règles formelles décrivant les relations aspect - classe. Ces règles seront éventuellement affinées suite à nos expérimentations. Par ailleurs, une version de l'outil pouvant extraire l'information directement du code est au stade final de son développement. Les seules informations que nous manipulons actuellement sont celles décrites dans le Tableau VIII. Nos efforts, pour le moment, ont plus été axés sur le développement de la technique et du prototype la supportant. Les aspects reliés à la performance de nos algorithmes ainsi qu'à la quantité de règles générées seront abordés dans le futur. Le nombre de règles est, cependant, étroitement lié aux interactions aspects - classes décrites dans les modèles.

4.4 Étude de cas

4.4.1 Cas simple et théorique

Nous considérons, dans ce qui suit, une étude de cas simple. Nous illustrons plusieurs types de conflits (spécification, aspect - aspect et préoccupations). L'exemple considéré contient une classe Facture (*Invoice*) qui prend un montant en entrée, calcule la valeur de la taxe et, si l'achat est à crédit, ajoute les intérêts. La classe Facture définit deux méthodes principales :

- Tax () : méthode surchargée qui ajoute une taxe à un montant
- Interest (double) : majoration du montant si l'acheteur achète à crédit.

Cette classe est accompagnée de deux aspects :

- Vérification (*Verification*) : valide les données en entrée
- Journalisation (*Log*): imprime des messages, enregistre dans un fichier en plus de gérer la taxe applicable.

La Figure 12, illustre ce cas sous forme graphique. À première vue, nous pouvons constater que des conflits possibles peuvent surgir parce que les deux aspects se partagent les mêmes points de jointure. L'advice *Around* peut également occasionner un problème de comportement. Il peut, en effet, capturer le contexte avec la méthode

proceed(). Si cette dernière méthode n'est pas utilisée, l'exécution sera différente. Ce problème sera discuté ultérieurement. La sémantique des relations entre aspects et classes est décrite par plusieurs informations réunies dans les tables rattachées aux relations. L'analyse du contenu de ces tables nous permet de générer les règles formelles décrivant les relations. Leur évaluation, par l'analyseur de règles, nous permet de détecter les conflits potentiels. Il est clair qu'à ce stade du processus, l'efficacité de la démarche dépend étroitement des informations fournies dans les modèles. Dans le cadre de l'exemple considéré, pour des raisons de simplification de la notation, nous noterons les aspects V*erification* et *Log* respectivement par A_v et A_l, la classe *Invoice* par C_i et les points de jointure *Tax* et *Interest* par J_t et J_i. La représentation formelle extraite du modèle est donnée par les expressions suivantes :

- Liaison 1 (A_v, J_t, C_i) = Before ((*Verification* ∩ Execution *Tax*(double)) ⊆ *Invoice*)
- Liaison 2 (A_v, J_i, C_i) = Before ((*Verification* ∩ Call *Interest*(double)) ⊆ *Invoice*)
- Liaison 3 (A_l, J_t, C_i) = Before ((*Log* ∩ Call *Tax*(..)) ⊆ *Invoice*)
- Liaison 4 (A_l, J_t, C_i) = Around ((*Log* ∩ Call *Tax*(double)) ⊆ *Invoice*)
- Liaison 5 (A_l, J_*, C_*) = After ((*Log* ∩ Call *(..)) ⊆ *)

Nous pouvons remarquer que la liaison 1 peut entrer en conflit avec la liaison 3. Elles partagent le même point de jointure, même si elles se connectent différemment à la même classe. Il s'agit d'un conflit de catégorie aspect - aspect lié à l'ordonnancement dépendant du contexte dynamique. En effet, s'il y a un appel de la méthode *Tax* avec une donnée de type autre que *Double*, l'un des aspects ne s'exécutera pas. Cependant, si une donnée de type *Double* est utilisée, les deux aspects s'exécuteront. Nous sommes devant un problème d'ordonnancement. Par ailleurs, la liaison 5 présente un exemple de conflit de spécification transverse. L'utilisation du point de jointure *.* conduit à un chevauchement des points de jointure dans de nombreuses classes et l'utilisation d'une fonction *println* conduit à une récursivité accidentelle.

Ultérieurement, l'analyse du code nous permettra d'extraire des informations complémentaires et d'affiner par conséquent les règles formelles. Les Figure 16 et Figure 17 illustrent deux implémentations de l'advice *Around*. Ces deux implémentations ont un comportement différent lors de l'exécution. Le code donné par la Figure 16 remplace le montant et le montant final est uniquement celui retourné par l'aspect. Cela bloque l'exécution de la liaison 1. Alors que le code de la Figure 17 permet une exécution de la liaison 1. Le résultat retourné dans ce dernier cas correspond à la taxe de la facture jumelée à la taxe ajoutée par l'aspect. Dans le cas de cet exemple, nous pouvons parler de conflits de préoccupations, plus précisément de comportement inconsistant ainsi que d'exécution conditionnelle. Par ailleurs, l'exécution du *proceed* influence l'exclusion

mutuelle entre les aspects. La liaison 1 dépend de la liaison 4, car nous avons une connexion au même point de jointure avec deux advices. La liaison 1 utilise un advice *Before* qui va s'exécuter avant l'*Around* de la liaison 4. Nous pouvons dans ce cas parler de dépendance de la liaison 4 vis-à-vis de la liaison 1. Le code fourni par la Figure 16 engendre dans ce cas une exclusion. Ainsi, la connaissance dans le cadre de l'exemple du comportement du *proceed* nous permet d'évaluer son impact. Une méthode avec *proceed* permettra l'exécution des autres méthodes, alors que son absence occasionne des anomalies d'exécution. Cette connaissance ne peut être obtenue que par analyse du code.

```
pointcut Jointure5(double montant) : call(* Facture.Tax(..)) && args(montant);
        double around(double montant): Jointure5(montant){
                return montant * tax;}
```

Figure 16 : Around sans proceed

```
        double around(double montant): Jointure5(montant){
                montant = proceed(montant);
                return montant * tax);}
```

Figure 17 : Around avec proceed

Les conflits engendrés par les aspects sont souvent implicites et difficiles à saisir, surtout lorsque cela influence le flux de contrôle à l'exécution. Notre démarche est plutôt statique. Elle nous permet, cependant, d'extraire un ensemble suffisant d'informations permettant de détecter un grand nombre de conflits. La technique proposée couvre

plusieurs types de points de jointure offerts par AspectJ tout en étant assez générale pour s'appliquer à différentes implémentations des aspects.

4.4.2 Discussion

Nous avons présenté, dans ce chapitre, une nouvelle technique de détection des conflits entre les aspects. L'originalité de l'approche réside, en particulier, dans l'analyse des modèles UML et la détection précoce des conflits entre aspects. Elle prend en compte les différentes possibilités d'intégration des aspects aux classes. La technique présentée s'applique aux points de jointure et de coupure définis dans AspectJ. Il s'agit d'éléments de base communs à tous les langages (ou environnements) de programmation orientée aspect. Nous croyons que la technique pourrait s'intégrer à tous les environnements actuels. Par ailleurs, elle tente de couvrir une grande partie du processus de développement, depuis les modèles de conception jusqu'au code. La détection précoce de conflits basée sur une analyse sémantique des modèles offre plusieurs avantages. Elle permet, par ailleurs, de piloter les deux analyses statiques et dynamiques du code de l'application.

L'analyse statique (modèles ou code) permet d'extraire un ensemble d'informations portant sur la sémantique des relations qu'entretiennent les aspects avec les classes. Ces informations sont traduites sous forme de règles formelles. L'évaluation de ces règles permet de détecter les conflits éventuels entre les aspects. Le processus est itératif. Il

s'affine au fur et à mesure de l'avancement dans le processus de développement, en fonction des détails apportés par chaque phase. Par ailleurs, ces informations sont regroupées et visualisées sous forme de tables intégrées aux modèles.

Nous prévoyons d'utiliser ces informations pour piloter une analyse dynamique qui serait complémentaire à notre approche. Tous les points de jointure définis dans le Tableau IV constituent des éléments importants. Il est, par ailleurs, intéressant de connaître comment un aspect se lie à une méthode. Ce processus est très complexe et exige une bonne gestion de l'ordonnancement dans l'exécution pour éviter les mauvaises surprises. La technique proposée couvre tous les points de jointure offerts par AspectJ. Par ailleurs, tous les points de coupure (nommées et anonymes) du Tableau V le sont également à l'exception du «control-flow-based». La gestion de ce dernier est assez complexe et sera considérée dans nos travaux futurs.

Chapitre 5

Implémentation

5.1 Présentation

Comme première expérimentation de notre approche, nous avons implémenté quelques règles de bases [TES 04c]. Dans le but d'effectuer une première simulation, nous avons repris l'exemple donné à la Figure 12 et nous nous sommes limités au contenu du Tableau IX, c'est-à-dire l'aspect, le point de jointure, l'argument, l'advice, la précédence et le type d'opération (Méthode d'accès ou de modification). Nous définissons le type d'opération de façon manuelle pour le moment. Cet élément est très important, car il détermine le niveau d'impact des conflits. Deux aspects en conflits avec un advice qui a un accès en lecture seulement (exemple : impression, journalisation, etc.) est moins problématique qu'une méthode altérant le «control flow». L'élément de classe est omis de la table parce que toutes les connexions ont le point de jointure (J_i) inclus dans la même classe. La liaison 6 est un ajout à l'étude de cas pour démontrer l'efficacité des règles à gérer des ajouts ou des modifications.

Tableau IX :Éléments de base pour la simulation

No	A_i	J_i	Arg_i	Advice	O_i	TO_i
1	Verification	Tax	Double	Before	-	Modification
2	Verification	Interest	Double	Before	-	Accès
3	Log	Tax	*	Before	-	Accès
4	Log	Tax	Double	Around	-	Modification
5	Log	*	*	After	-	Accès
6	Log	Tax	Int	Before	-	Accès

Dans ce qui suit, nous décrivons les quelques règles formelles que nous avons implémentées ainsi que les résultats obtenus. Nous simplifions dans ce qui suit la représentation. La représentation mathématique des deux liaisons aspect - classes (R_{ac}) partageant une même classe et n'ayant pas de point de jointure en commun (C_i, $\neg J_i$) est :

- $R_{ac}i= \{C_i, J_i, Arg_i, Advice_i, O_i, TO_i\}$ (première relation)
- $R_{ac}k = \{C_k, J_k, Arg_k, Advice_k, O_k, TO_k\}$ (seconde relation)

L'intersection de ces deux ensembles donne un sous-ensemble où la classe est commune, mais non les points de jointure.

- $\therefore \longrightarrow R_{ac}i \cap R_{ac}k \Longrightarrow \{C_{i=k}\} \wedge J_{i\neq k}$

Les règles implémentées sont :

- 2 R_{ac} partageant classe et aucun point de jointure (C_i, $\neg J_i$): aucun conflit direct, les connexions sont indépendantes;
- 2 R_{ac} partageant classe, point de jointure, argument et advice (C_i, J_i, Arg_i, Advice): Haut risque de conflit parce qu'ils partagent les mêmes conditions d'exécution. Le type d'opération et l'ordre d'exécution sont alors très importants;
- 2 R_{ac} partageant classe, point de jointure, argument mais pas l'advice (C_i, J_i, Arg_i, $\neg$Advice): Risque moyen de conflit, car l'advice n'est pas le même, le premier aspect exécuté influence le second;
- 2 R_{ac} partageant classe, point de jointure et aucun argument (C_i, J_i, $\neg Arg_i$): faible risque de conflit, les aspects ne seront pas exécutés au même moment parce qu'ils ne partagent pas l'argument. Cependant, un risque de conflits persiste, car un aspect peut changer le type d'argument et le second aspect n'atteindra jamais les conditions de son exécution;
- 2 R_{ac} ne partageant aucune classe ($\neg C_i$) ou $R_{aci} \cap R_{ack} = \{\}$: aucun conflit direct, le sous-ensemble est vide.

Comme nous pouvons le constater, à ce niveau, on se rend compte de l'importance de l'ordre d'exécution et du type d'opération effectuée par l'advice. Deux aspects effectuant des opérations de lecture et de journalisation ne seront pas en conflit direct. Par contre, si nous avons un aspect en modification qui s'exécute avant un aspect en lecture, la configuration est différente et présente des risques de conflits.

5.2 Discussion

Nous avons présenté, dans la section 3.4, divers types de conflits que l'on peut retrouver dans un programme orienté aspect. Notre approche met d'abord l'accent sur les conflits reliés à la dynamique d'interaction Aspect - Aspect. Le contenu des tables T_{AC} nous permet de déterminer les points de connexions des aspects. Son analyse nous permet de déterminer la sémantique liée à l'ordonnancement de leur exécution. Lorsque les points de connexion se retrouvent dans une même classe et qu'ils sont adjacents les uns aux autres et s'exécutent selon un ordre précis (exécution conditionnelle, exclusion mutuelle, etc.), cela peut engendrer plusieurs problèmes. Si on considère les appels consécutifs à plusieurs méthodes, chaque appel représente un point de jointure possible. Les aspects peuvent interagir avec chacun des appels. Chacune des connexions est décrite par les éléments du Tableau VIII. Une analyse de ces éléments permet de mettre en évidence ce qui est partagé par les aspects et d'identifier ainsi les conflits potentiels. Notre technique permet d'inclure les « * » dans les relations. Ceci facilite la détection de conflits reliés aux spécifications transverses. Par ailleurs, l'analyse des points de coupure nous permet de déceler certains conflits liés à la récursivité accidentelle. Ce type de conflit se produit lorsqu'un aspect, rattaché à un point de jointure, exécute sa méthode qui est incluse dans le point de coupure d'une autre relation aspect - classe.

Dans l'étude de cas, nous retrouvons l'utilisation du *return* au lieu du *proceed*. Il s'agit d'une exécution conditionnelle pouvant conduire à une inconsistance. L'aspect *Verification* dépend de l'exécution de la fonction *around* de l'aspect *Log*. Nous ne pouvons déterminer avec certitude s'il s'agit d'un comportement inconsistant. Cependant, il peut facilement être identifié et un avertissement peut être émis. Pour le moment, nous ne tenons pas compte des conflits Base – Aspect. Le tableau ci-dessous résume les principaux conflits détectés par notre technique.

Tableau X : Conflits détectés

Catégorie	Type	Phase	Référence
Spécification transverse	Point de jointure accidentelle	Modélisation	Liaison 5
	Récursivité accidentelle	Modélisation	Liaison 5
Aspect - Aspect	Exécution conditionnelle	Modélisation et analyse du code	Liaison 1 et 4
	Exclusion mutuelle	Modélisation et analyse du code	Liaison 1 et 4
	Ordonnancement	Modélisation	Liaison 1 et 3
	Ordonnancement dépendant du contexte dynamique	Modélisation	Liaison 1 et 4
	Échanges et conflits aux niveaux de la spécification	Modélisation et analyse du code	
Base - Aspect	Aspect invasif et dépendance circulaire	N/A	
Préoccupation - Préoccupation	Changement de fonctionnalités	Analyse du code	
	Comportement inconsistant	Modélisation et analyse du code	Liaison 4 avec proceed
	Anomalies de composition	Analyse du code	

Nous donnons dans ce qui suit un exemple de messages donnés par le prototype suite à une analyse :

- Les liaisons 1 et 2 partagent C_i (Invoice)

MESSAGE : Aucun conflit potentiel n'est détecté, mais l'aspect agit en MODIFICATION, il effectue un Exit if argument < 0

- Les liaisons 1 et 3 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Double) et Advice (Before)

AVERTISSEMENT : Haut risque de conflit
ACTION : Définition d'un ordre de précédence requise

- Les liaisons 1 et 4 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Double)

AVERTISSEMENT: Risque moyen de conflit
ACTION : Vérification - Advice impliquant 2 modifications

- Les liaisons 1 et 5 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Double)

AVERTISSEMENT: Risque moyen de conflit
ACTION : Advice en accès et modification

- Les liaisons 1 et 6 partagent C_i (Invoice) et J_i (Tax)

AVERTISSEMENT: Faible risque de conflit

- Les liaisons 2 et 3 partagent C_i (Invoice)

MESSAGE : Aucun conflit potentiel n'est détecté et l'aspect de la liaison 2 effectue un ACCÈS sur un argument > 40

- Les liaisons 2 et 4 partagent C_i (Invoice)

MESSAGE : Aucun conflit potentiel n'est détecté et l'aspect de la liaison 2 effectue un ACCÈS sur un argument > 40

- Les liaisons 2 et 5 partagent C_i (Invoice) et J_i (Interest) et Arg_i (Double)

AVERTISSEMENT: Aucun risque de conflit
ACTION : Aucune - Advice impliquant 2 accès

- Les liaisons 2 et 6 partagent C_i (Invoice)

MESSAGE : Aucun conflit potentiel n'est détecté et l'aspect de la liaison 2 effectue un ACCÈS sur un argument > 40

- Les liaisons 3 et 4 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Double)

AVERTISSEMENT: Risque moyen de conflit
ACTION : Advice en accès et modification

- Les liaisons 3 et 5 partagent C_i (Invoice) et J_i (Tax) et Arg_i (*)

WARNING : Aucun risque de conflit
ACTION : Aucune - Advice impliquant 2 accès

- La liaison 3 et 6 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Int) et Advice (Before)

AVERTISSEMENT: Conflit avec peu d'impact, car 2 accès
ACTION : Définition d'un ordre de précédence requise

- Les liaisons 4 et 5 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Double)

AVERTISSEMENT: Risque moyen de conflit
ACTION : Advice en accès et modification

- Les liaisons 4 et 6 partagent C_i (Invoice) et J_i (Tax)

AVERTISSEMENT: Faible risque de conflit

- Les liaisons 5 et 6 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Int)

AVERTISSEMENT: Aucun risque de conflit
ACTION : Aucune - Advice impliquant 2 accès

- CONFLIT: CONFLIT DÉTECTÉ - DÉPENDANCE CIRCULAIRE *.* pour la liaison 5

Par exemple, si nous définissons un ordre de précédence pour les relations 3 et 6, le prototype renvoie le message suivant :

- Les liaisons 3 et 6 partagent C_i (Invoice) et J_i (Tax) et Arg_i (Int) et Advice (Before)

AVERTISSEMENT : Conflit avec peu d'impact, car méthode en accès
ACTION : Aucune, l'ordre suivant est défini : 3 avant 6 et cela en accès seulement.

5.3 Bilan de l'implémentation

Cette première implémentation démontre la faisabilité de notre technique de détection. Les résultats obtenus concordent avec ceux attendus lors de l'étude de cas théorique. Cependant, la phase de test a également fait ressortir la complexité de tout ce processus et la difficulté de saisir tous les conflits.

Au-delà de l'originalité de la technique et des avantages cités précédemment relatifs au fait qu'elle permet une détection précoce des conflits, nous pouvons également citer sa simplicité de mise en œuvre et la possibilité de son intégration à un environnement de développement. Il a été relativement facile de coder la première version du prototype, car les règles utilisées étaient peu nombreuses et simples.

Cette première phase d'évaluation préliminaire a fait, cependant, ressortir une insuffisance dans notre approche. Bien qu'il ne s'agisse pas de dire que notre technique est inefficace, il est indéniable de constater que la détection des conflits dans des programmes aspects se révèle une tâche ardue. Il est très complexe dans certains cas de pouvoir établir des constats clairs sur les conflits, car il est difficile de pouvoir discerner les comportements voulus de ceux qui sont indésirables. Au mieux, nous pouvons dans ces cas identifier des endroits sujets à cette problématique sans toutefois pouvoir indiquer clairement qu'il s'agit d'un conflit, exception faite de cas sans équivoque

comme la dépendance circulaire. De plus, à cause de cette difficulté liée essentiellement à la complexité qui découle de certaines interactions, il apparaît, à la lumière de cette première investigation, difficile de pouvoir créer un outil de détection et de résolution de tous les conflits. L'intervention humaine reste importante dans certains cas.

5.4 Travaux futurs

Pour cette première simulation et analyse statique du code, nous avons extrait manuellement l'information des programmes utilisés comme exemples. Il nous reste donc à développer tous les outils d'extraction de l'information à partir du code. Il s'agit également de développer un outil complet de modélisation UML supportant nos stéréotypes et notre idée de *Pointcut Wizard*. Un autre élément important à considérer lors de l'analyse du code porte sur l'identification des méthodes qui agissent seulement en lecture et celles qui agissent en modification. Les exemples utilisés sont simples, mais démontrent néanmoins l'efficacité de notre technique de détection. Il reste cependant que ces études de cas ne présentaient pas beaucoup de règles. Dans le cas d'un gros système, on peut s'attendre à ce que le nombre de règles générées soit important.

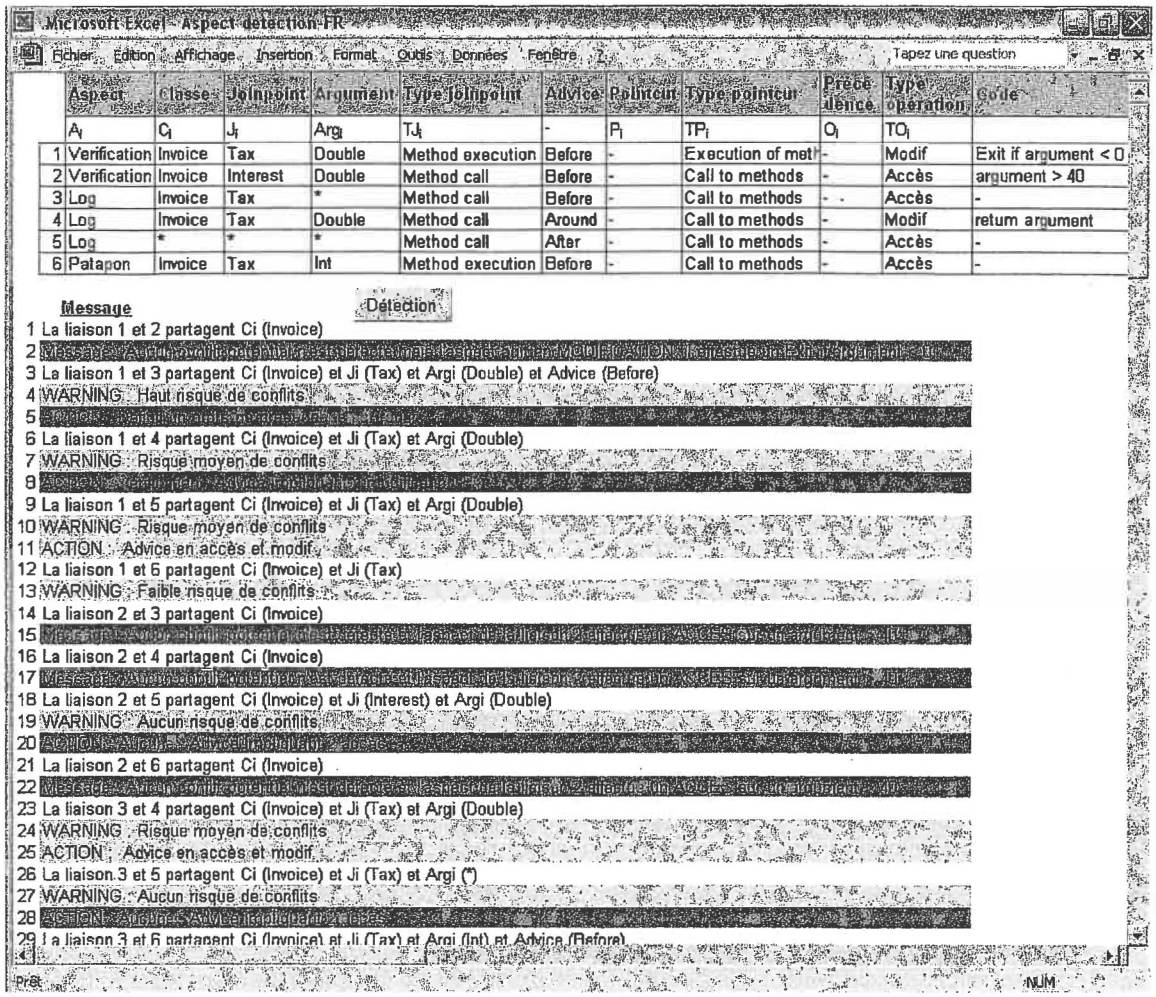

Figure 18: Exemple d'interface

Comme nous pouvons le constater, notre prototype donne d'ores et déjà beaucoup d'information sous la forme de messages. Cependant, nous sommes conscients que le nombre de messages peut être important et poser des problèmes lors de l'interprétation. Nous pensons développer un outil d'aide à l'interprétation qui agirait comme premier filtre de messages et permettrait de mieux aiguiller les développeurs pour l'identification des conflits entre aspects.

Conclusion

Pour conclure, nous pouvons affirmer que le présent travail a apporté des idées nouvelles, supportant une technique de détection, puissante et assez simple, des conflits entre les aspects. Ce travail apporte un plus en termes d'assurance-qualité dans le domaine de la technologie aspect. Cette dernière sera à notre avis la prochaine évolution logique de la programmation orientée objet. Au-delà des caractéristiques et avantages de la technique proposée (déjà discutées dans le mémoire), il s'avère que la création d'un outil performant de détection des conflits nécessite beaucoup de travail et de connaissances théoriques. Par ailleurs, l'une des difficultés rencontrées porte sur le manque de travaux reliés. Dans notre cas, il ne s'agissait pas de réfuter ou d'améliorer un concept déjà existant, mais plutôt de développer quelque chose de nouveau.

Un des points importants de notre approche est son caractère fédératif. L'utilisation des outils de modélisation, l'utilisation du code, l'utilisation d'une représentation mathématique ainsi que la création d'un processus d'inférence que nous espérons rendre assez intelligent à l'avenir sont tous des éléments qui concourent à supporter un outil intéressant pour la détection des conflits entre les aspects. Nous pouvons souligner en particulier l'application précoce de notre technique (analyse des modèles), relativement aux rares approches publiées dans la littérature. La détection de conflits entre les aspects n'est pas une tâche facile. Nous espérons que l'évolution des travaux dans ce domaine

puisse offrir un cadre performant pour la détection des conflits entre les aspects, ce qui est nécessaire pour la maturité de cette nouvelle et prometteuse technologie.

Bibliographie

[ALD 01] Aldawud O., Elrad T. & Bader A., "A UML profile for aspect oriented modeling", OOPSLA 2001, Tampa Bay, États-Unis.

[ALD 03] Aldawud, O., Elrad, T. & Bader A., "A. UML profile for aspect-oriented software development", AOSD 2003, 3rd Aspect-Oriented Modeling Workshop, Boston, États-Unis.

[AND 01] Andrews J. H., "Using process algebra as a foundation for programming by separation of concerns", ICSE 2001, Toronto, Canada.

[AOS 05] Site de la conférence annuelle AOSD (Aspect-Oriented Software Development), "Tools for Developpers", 2005, [En ligne]. http://aosd.net/wiki/index.php?title=Tools_for_Developers (Consulté le 28 juin 2005).

[BAL 02] Baltus J., Institut d'informatique des FUNDP, "La programmation orientée aspect et AspectJ : présentation et application dans un système distribué", 2002, [En ligne]. http://www.info.fundp.ac.be/~ven/CISma/FILES/2002-baltus.pdf (Consulté le 28 juin 2005).

[BAN 01] Baniassad E. L. A., Murphy G. C. & Schwanninger C., "Determining the why of concerns", ICSE 2001, Toronto, Canada.

[BERG 03] Bergmans L. M. J., "Towards Detection of Semantic Conflicts between Crosscutting Concerns", AAOS 2003, Darmstadt, Allemagne.

[BERT 03] Bertagnolli S. d. C. & Lisboa, M. L. B., "The FRIDA Model", AAOS 2003, Darmstadt, Allemage.

[CHE 01] Chechik M. & Easterbrook S., "Reasoning about compositions of concerns", ICSE 2001, Toronto, Canada.

[CHI 01] Chiba S., "What are the best join points?", OOPSLA 2001, Tampa Bay, États-Unis.

[CLE 03] Clement A., Colyer A. & Kersten M., "Aspect-Oriented Programming with AJDT", AAOS 2003, Darmstadt, Allemagne.

[CON 03] Constantinides C. A., "A Case Study on Making the Transition from Functional to Fine-Grained Decomposition", AAOS 2003, Darmstadt, Allemagne.

[DOU 02] Douence R., Fradet P. & Südholt M., "Detection and resolution of aspect interactions", INRIA, technical report, no. RR-4435, Avril 2002.

[EID 01] Eide E., Reid A., Flatt M. & Lepreau J., "Aspect weaving as component knitting : separation concerns with Knit", ICSE 2001, Toronto, Canada.

[FAY 03] Fayad, M.E. et Rangahat A., "Modeling aspects using software stability and UML", UML 2003, 4th Aspect-Oriented Modeling Workshop, San Francisco, États-Unis.

[GAR 01] Garcia A. F. & De Lucena C. J. P., "An aspect-based object-oriented model for multi-agent systems", ICSE 2001, Toronto, Canada.

[GEL 04] Gélinas J.F., Badri L. & Badri M., "Aspect Cohesion Measurement based on Dependence Analysis", Proceedings of the 8th ECOOP (European Conference on Object-Oriented Programming) Workshop on QAOOSE (Quantitative Approaches in Object-Oriented Software Engineering), 2004, Oslo, Norvège.

[GEL 05] Gélinas J.F., Badri L. & Badri M., "Measuring Cohesion in Aspect-Oriented Systems", Proceedings of the IASTED International Conference on Software Engineering (SE'05), 2005, Innsbruck, Autriche.

[GRA 03] Gradecki J. D. & Lesiecki N., "Mastering AspectJ", Wiley, Indianapolis, 456 pages, 2003.

[HAB 01] Habra N., "Separation of concerns in software engineering education", ICSE 2001, Toronto, Canada.

[HAC 03] Hachani O. & Bardou D., "On Aspect-Oriented Technology and Object-Oriented Design Patterns", AAOS 2003, Darmstadt, Allemagne.

[HAN 03] Hanneman J., Chitchyan R. & Rashid A., "Analysis of aspect-oriented software", AAOS 2003, Darmstadt, Allemagne.

[HOA 04] Hoare, C. A. R., Communicating Sequential Processes, "Communicating sequential processes", 2004, [En ligne] http://www.usingcsp.com/cspbook.pdf (Consulté le 28 juin 2005).

[IBM 01] IBM, Multi-dimensional separation of concerns : software engineering using Hyperspaces, 2001, [En ligne]. http://www.research.ibm.com/hyperspace/ (Consulté le 28 juin 2005).

[JAC 03] Jacobson I., Journal of Object Technology, "Use cases and aspects – working seamlessly together", 2003, [En ligne]. http://www.jot.fm/issues/issue_2003_07/column1 (Consulté le 28 juin 2005).

[KIC 97] Kiczales G., Lamping J., Mendhekar A. Maeda C., Lopes C. V., Loingtier J. & Irwin J., "Aspect-oriented programming", ECOOP 97, Jyväskylä, Finlande.

[KIC 03] Kiczales G., The ServerSide.com, "Interview with Gregor Kiczales", 2003, [En ligne]. http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp (Consulté le 28 juin 2005).

[KIS 02] Kiselev I., "Aspect-Oriented programming with AspectJ", Sams, 288 pages, 2002.

[LAD 02] Laddad R., JavaWorld, "I want my AOP!", janvier 2002, [En ligne]. http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html (Consulté le 28 juin 2005).

[LAD 03] Laddad R., "AspectJ in action : practical aspect-oriented programming", Manning, 512 pages, 2003.

[NOR 01] Nordberg III M. E., "Aspect-oriented dependency inversion", OOPSLA 2001, Tampa Bay, États-Unis.

[OWC 05] ObjectWeb Consortium, The JAC (Java Aspect Components) Project, 2005, [En ligne]. http://jac.objectweb.org/documentation.html (Consulté le 28 juin 2005).

[OMG 03] OMG, "UML Summery V1.5", 2003, [En ligne]. http://www.omg.org/docs/formal/03-03-01.pdf (Consulté le 28 juin 2005).

[ORR 03] Orr J., JavaWorld, "Java tools reign supreme : Most Innovative Java Product or Technology: AspectJ 1.0.6, Eclipse.org", juin 2003, [En ligne]. http://www.javaworld.com/javaworld/jw-06-2003/jw-0609-eca_p.html (Consulté le 28 juin 2005).

[POP 03] Popovici A., Alonso G. & Gross T., "Just-in-time aspects: efficient dynamic weaving for Java", AOSD 2003, Boston, États-Unis.

[REI 03] Reina M., Torres J. & Toro M., "Aspect-Oriented Web Development vs. Non Aspect-Oriented Web Development", AAOS 2003, Darmstadt, Allemagne.

[SLO 03] Slowikowski P. & Zielinski K., "Comparison Study of Aspect-Oriented and Container Managed Security", AAOS 2003, Darmstadt, Allemagne.

[STO 03a] Störzer M., Krinke J. & Breu S., "Trace Analysis for Aspect Application", AAOS 2003, Darmstadt, Allemagne.

[STO 03b] Störzer M. & Krinke J., "Interference analysis for AspectJ", AOSD 2003, Boston, États-Unis.

[TES 04a] Tessier F., Badri M. & Badri L., "Détection de Conflits entre les Aspects : De la Modélisation à l'Analyse du Code", Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA), 2004, Paris, France.

[TES 04b] Tessier F., Badri M. & Badri L., "Towards a Formal Detection of Semantic Conflicts Between Aspects : A Model-Based Approach", Proceedings of the 7th International Conference on the Unified Modeling Language (UML), 5th Workshop on Aspect-Oriented Modeling (AOM), 2004, Lisbonne, Portugal.

[TES 04c] Tessier F., Badri M. & Badri L., "A Model-Based Detection of Conflicts Between Crosscutting Concerns: Towards a Formal Approach", Proceedings of the 2th IEEE International Conference SEFM (Software Engineering and Formal Methods), Workshop on Aspect-Oriented Software Development (AOSD), 2004, Beijing, Chine.

[TES 04d] Tessier F., Badri M. & Badri L., "An Early Detection of Semantic Conflicts Between Aspects : A Model Analysis Based Technique", The 8th IASTED International Conference on Software Engineering And Applications (SEA), 2004, Cambridge, États-Unis.

[UMV 02a] Université Marne la Vallée, "Le composant CCM", 2002, [En ligne]. http://www-igm.univ-mlv.fr/~dr/XPOSE2002/CCM/corba.htm (Consulté le 28 juin 2005)

[UMV 02b] Université Marne la Vallée, "JAC", 2002, [En ligne]. http://www-igm.univ-mlv.fr/~dr/XPOSE2002/JAC/Accueil.htm (Consulté le 28 juin 2005)

[XAT 03] Xerox AspectJ Team, Eclipse.org, "AspectJ programming guide", 2003, [En ligne]. http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/doc/progguide/index.html (Consulté le 28 juin 2005).

[ZHA 01a] Zhao J., "Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging", Technical-Report SE-2001-134-17, Information Processing Society of Japan (IPSJ), Octobre 2001.

[ZHA 01b] Zhao J., "Testing criteria for aspect-oriented software", Technical-Report SE-2001-135-6, Information Processing Society of Japan (IPSJ), Novembre 2001.

[ZHA 02] Zhao J., "Towards a Metrics Suite for Aspect-Oriented Software," Technical-Report SE-2002-136-25, Information Processing Society of Japan (IPSJ), 2002.