وزارة التعليم العالي و البحث العلمي

Université Badji Mokhtar –Annaba
Badji Mokhtar –Annaba- University

جامعة باجي مختار- عنابة

Année**: 2014 - 2015**

Faculté des sciences de l'ingéniorat
Département d'informatique

T H È S E

Pour obtenir le diplôme de
Docteur 3éme cycle

Extraction d'Aspects à partir des Modèles

Filière : Informatique
Spécialité : Ingénierie des Logiciels Complexes

Préparée par

Fairouz DAHI

Jury :

Président: Mme Hassina SERIDI Professeur Université Badji Mokhtar - Annaba
Directeur de thèse : **Mme Nora BOUNOUR-ZEGHIDA MCA Université Badji Mokhtar - Annaba**
Examinateurs : Mr Djamel MESLATI Professeur Université Badji Mokhtar - Annaba
Mr Zine-Eddine BOURAS MCA Ecole Préparatoire aux Sciences et Techniques - Annaba
Mme Habiba BELLEILI MCA Université Badji Mokhtar - Annaba

وزارة التعليم العالي و البحث العلمي

Université Badji Mokhtar –Annaba
Badji Mokhtar –Annaba- University

جامعة باجي مختار- عنابة

Année**: 2014 - 2015**

Faculté des sciences de l'ingéniorat
Département d'informatique

T H È S E

Pour obtenir le diplôme de
Docteur 3^{éme} cycle

Extraction d'Aspects à partir des Modèles

Filière : Informatique
Spécialité : Ingénierie des Logiciels Complexes

Préparée par

Fairouz DAHI

Jury :

Président: Mme Hassina SERIDI Professeur Université Badji Mokhtar - Annaba
Directeur de thèse : **Mme Nora BOUNOUR-ZEGHIDA MCA Université Badji Mokhtar - Annaba**
Examinateurs : Mr Djamel MESLATI Professeur Université Badji Mokhtar - Annaba
Mr Zine-Eddine BOURAS MCA Ecole Préparatoire aux Sciences et Techniques - Annaba
Mme Habiba BELLEILI MCA Université Badji Mokhtar - Annaba

DÉDICACES

A mes parents,

A mes sœurs,

A la mémoire de mes grands parents.

REMERCIEMENTS

Je tiens tout d'abord à remercier Dieu de m'avoir donné la force, la volonté et la patience pour accomplir ce travail de thèse, Dieu merci.

Je remercie très chaleureusement mon encadreur Dr. Bounour, avec qui je prends toujours autant de plaisir à travailler. Vos conseils toujours avisés et votre détermination, m'ont permis d'avancer à pas sûrs dans l'accomplissement de ce travail.

Je souhaite adresser ma gratitude au directeur du laboratoire LISCO (Laboratoire d'Ingénierie des Systèmes Complexes), Pr. Meslati, pour m'avoir accueilli au sein de son laboratoire de recherche. La confiance que m'avez accordée et les moyens que vous avez mis à ma disposition m'ont permis de mener à bien ce projet de recherche. Je vous en suis très reconnaissante.

Je remercie également les membres de jury, pour avoir accepté d'examiner cette thèse, et aussi pour avoir pris le temps d'évaluer mon travail consciencieusement.

Je tiens à exprimer mes profonds remerciements à mes parents et mes sœurs, ainsi que toute ma famille, pour leurs soutiens et encouragements.

Enfin, je remercie toute personne m'ayant encouragé durant la réalisation de ce travail.

RÉSUMÉS

Résumé

L'existence de préoccupations transversales dispersées, ou enchevêtrées, dans le code source rend complexe la compréhension et l'évolution du système logiciel. Afin d'améliorer la modularité des logiciels et bénéficier des avantages du paradigme orienté aspect, plusieurs approches visent l'identification des préoccupations transversales. En absence de la détection précoce de ces préoccupations transversales, celles ci tendent à être inaperçues et fortement liées, ce qui ne permet pas aux développeurs d'identifier une implémentation optimale.

Nous proposons dans cette thèse une nouvelle approche d'identification des préoccupations transversales au niveau des modèles conceptuels. Ces derniers sont matérialisés par les diagrammes de séquence. Le diagramme de séquence porte une information pertinente qui consiste en des interactions entre les objets du système logiciel, ainsi que l'ordre chronologique de ses tâches.

Mots clés

Aspect, Diagramme de séquence, Fouille d'aspects, Invocation de méthode, Maintenance du logiciel, Préoccupations transversales, Rétro-ingénierie.

Abstract

The existence of crosscutting concerns scattered or tangled in the source code complicates the software comprehension and evolution. To improve the modularity of software systems and to take advantage of the benefits of the aspect-oriented paradigm, several approaches aim to identify the crosscutting concerns to allow their modeling by aspects. In the absence of early detection of these crosscutting concerns, they tend to be overlooked and become closely linked, not allowing developers to identify an optimal implementation.

We propose in this thesis a new approach for crosscutting concerns identification from design model. This latter is materialized by sequence diagrams. Sequence diagram carries pertinent information which consists in interactions between objects of the software system, and the chronological order of its tasks.

Keywords

Aspect mining, Crosscutting concerns, Early aspect, Method invocation, Reverse engineering, Sequence diagram, Software maintenance.

ملخص

وجود الاهتمامات المتداخلة، مبعثرة أو متشابكة، يعقد فهم وتطوير البرامج. من أجل تحسين نمطية البرامج و الاستفادة من فوائد النمذجة بالجوانب ، هناك عدة طرق تهدف إلى تحديد الاهتمامات المتداخلة في جوانب. في غياب الكشف المبكر عن هذه الاهتمامات المتداخلة، تميل هذه الأخيرة أن تكون خفية و تصبح جد مترابطة، مما لا يسمح لمطوري البرامج من القيام بتنفيد مثالي.

نقترح في هذه الأطروحة نهج جديد لتحديد الاهتمامات المتداخلة على مستوى مفاهيمي. يتمثل هذا الأخير في الرسومات التخطيطية للتسلسل. الرسم التخطيطي للتسلسل يحمل معلومات مهمة، تتمثل في التداخلات بين عناصر البرامج، فضلا عن ترتيب التسلسل الزمني للمهام الموكلة إليه.

الكلمات الرئيسية

استخراج الجوانب، استدعاء الطريقة، الاهتمامات المتداخلة، الجوانب، الرسم التخطيطي للتسلسل، الهندسة العكسية، صيانة البرمجيات.

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLES

LISTE DES ALGORITHMES

CHAPITRE 1

INTRODUCTION

« If I had an hour to solve a problem I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions.»

(Albert Einstein 1974)

1.1 Contexte de recherche

La programmation orientée objet est le paradigme de choix pour plusieurs projets logiciels. Elle est particulièrement efficace dans l'expression des fonctionnalités dites verticales, fonctionnalités exprimant les préoccupations métiers du système. Cependant, elle s'avère limitée dans le cas de l'existence des fonctionnalités qui sont enchevêtrées ou dispersées dans différents endroits d'une application, et qui ne bénéficient pas d'une encapsulation adéquate, tant au niveau des modèles de conception que des langages de programmation. Ces fonctionnalités sont dites transversales. Le fait d'être enchevêtrées ou dispersées rend complexe la compréhension du code source, la maintenance, l'évolution et la réutilisation du système logiciel.

Pour remédier aux insuffisances du paradigme orienté objet, plusieurs approches de la séparation avancée des préoccupations ont vu le jour, telle que la programmation orientée aspect. Cette dernière vise à améliorer l'évolution et la lisibilité des systèmes logiciels, en encapsulant chacune des préoccupations, indépendamment les unes des autres et de tout objet métier, ce qui simplifie le travail du programmeur et permet la réutilisation du code existant.

Afin de bénéficier des avantages du paradigme orienté aspect, nous avons besoin d'identifier d'abord les préoccupations transversales, ensuite les transformer en aspects.

Le processus d'identification des préoccupations transversales vise la découverte des préoccupations entrecoupant les classes d'un système logiciel.

Trois applications ont motivé l'émergence du domaine d'identification d'aspects: la maintenance des systèmes logiciels orientés objet déjà implémentés (*legacy systems*), la migration des systèmes logiciels orientés objet vers le paradigme aspect, et le développement orienté aspect.

La maintenance :

La loi d'entropie du logiciel, formulée par ***Belady*** et ***Lehman*** [Bel 76], nous apprend qu'avec le temps, la plupart des systèmes logiciels ont tendance à se dégrader progressivement dans la qualité, sauf s'ils sont maintenus. Toutes les études en génie logiciel ont montré que la maintenance et l'évolution sont les phases les plus longues et les plus coûteuses dans le cycle de vie du logiciel [Mas 03, Hil 07]. Cela est dû au besoin de satisfaire les nouvelles exigences des utilisateurs (telles que celles commerciales et technologiques), ou de modifier celles existantes, afin d'éviter la dégradation de l'efficacité du logiciel. Pour ces raisons, la maintenance logicielle a fait l'objet de beaucoup d'attention ces dernières années. Son utilisation permet d'apporter des modifications aux systèmes logiciels déjà implémentés. Mais auparavant, il convient de comprendre ces systèmes. La dispersion et l'enchevêtrement de certaines préoccupations rendent la compréhension difficile. Le programmeur passe plus de 50% de son temps à essayer de comprendre le programme [Won 95]. A cet effet, nous avons besoin d'identifier et d'extraire au préalable ces préoccupations transversales.

La migration :

Afin de bénéficier de la décomposition offerte par le paradigme orienté aspect, il est nécessaire de migrer les systèmes logiciels orientés objet existants (*legacy systems*) vers une solution orientée aspect. C'est ce qu'on appelle la migration orientée aspect des systèmes orientés objet [Cec 07]. Cette migration suppose l'extraction des préoccupations transversales, afin de pouvoir ensuite les refactoriser dans des aspects [Deu 03].

Le développement orienté aspect :

Les intervenants peuvent décrire leur système en termes d'exigences, dont chacune peut concerner une ou plusieurs préoccupations. Celles-ci s'enchevêtrent et se dispersent sur les exigences, impliquant l'émergence de préoccupations transversales. Afin de bénéficier d'un développement orienté aspect, il est nécessaire d'extraire et de gérer ces préoccupations transversales durant les phases du cycle de développement du logiciel [Ban 06, Mor 11] (avant la phase de l'implémentation), au niveau exigences, architecture ou conception. Cela permet d'obtenir une implémentation orientée aspect.

1.2 Problématique

Il existe plusieurs approches automatiques ou semi-automatiques permettant d'identifier les aspects, à partir de l'implémentation (dans le but de la maintenance des systèmes implémentés, ou la migration vers l'orienté aspect), ou avant l'implémentation (dans le but du développement orienté aspects). Cependant, l'identification des aspects au niveau de l'implémentation (code source) a l'avantage d'utiliser la version finale du système logiciel. Or, cette identification est difficile, car les aspects sont fortement intégrés avec le code métier, et le code source est lié à la technologie d'implémentation.

D'autre part, identifier les aspects durant les phases du cycle de développement du logiciel (avant l'implémentation) a l'avantage de gérer tôt la transversalité, et par conséquent, de bénéficier d'un développement orienté aspect. Or, cette identification comporte un risque, selon lequel le système logiciel peut ne pas être dans sa version finale (des aspects peuvent disparaitre et d'autres peuvent émerger).

En outre, la plupart des approches d'identification d'aspects souffrent de nombreuses limites, tel que le manque d'une définition précise d'aspect, ce qui ne permet pas à toutes les préoccupations transversales identifiées d'être refactorisées dans des aspects.

1.3 Motivations

Beaucoup d'approches d'identification d'aspects ont été proposées. Par ailleurs, la précision reste un objectif qui n'a jamais été atteint.

Il s'avère opportun de proposer une nouvelle approche d'identification d'aspects, en essayant de garder les points forts et dépasser les points faibles cités ci-dessus, d'une part, et d'unifier le processus d'identification d'aspects pour qu'il soit applicable à la fois avant et après l'implémentation, d'autre part.

A cet effet, une approche qui exploite les relations d'interactions internes des éléments du système sera plus pertinente, car ces relations portent une information utile, indépendante de la syntaxe du langage de programmation et de la technologie d'implémentation. Ces relations sont représentées par les diagrammes d'interactions d'objets sur le plan conceptuel, et les relations d'invocations de méthodes au niveau implémentation.

L'utilisation d'un niveau (sur lequel nous désirons identifier les aspects) plus abstrait que celui du code source des systèmes logiciels (*legacy systems*), permet de rendre l'approche d'identification d'aspects plus générale et indépendante de la technologie d'implémentation. En outre, l'identification d'aspects visant le développement orienté aspect, s'avère plus appropriée si elle est effectuée à un niveau plus proche de la version finale qui représente l'ensemble des fonctionnalités du système.

1.4 Objectifs

Notre travail consiste à proposer une nouvelle approche d'identification d'aspects, qui détecte à la fois les deux symptômes de la transversalité (enchevêtrement et dispersion), et qui sera applicable dans n'importe quel contexte d'identification d'aspects (avant ou après la phase d'implémentation).

Notre approche concernant l'identification d'aspects, est basée sur les diagrammes de séquence, en exploitant les transmissions de messages et leur ordre chronologique. Nous voulons également faciliter la refactorisation des aspects identifiés, en les spécifiant en détails.

Afin d'avoir une évaluation de l'approche proposée, et de détecter les erreurs possibles dans le processus d'identification d'aspects (faux aspects candidats), nous proposons trois nouvelles métriques. Celles ci vont aider l'utilisateur à filtrer les aspects candidats.

Nous avons développé un outil qui implémente notre approche. Cet outil a été appliqué sur deux études de cas : système de gestion de bibliothèque (*Library Management System)* et système de péage des autoroutes portugaises (*Portuguese Highways Toll System*).

1.5 Contenu de la thèse

Outre l'introduction et la conclusion, cette thèse s'articule autour de cinq chapitres.

Chapitre 2 :

Nous présentons la notion de préoccupation et ses caractéristiques, ainsi que la séparation des préoccupations d'une manière générale, tout en mettant l'accent sur les concepts et les apports de la programmation orientée aspect. Le chapitre se termine par définir les métriques orientées préoccupations les plus utilisées.

Chapitre 3 :

L'émergence du paradigme orienté aspect a donné naissance au domaine d'aspect mining, qui vise l'identification des préoccupations transversales (aspects). Dans ce chapitre, nous présentons une étude comparative des travaux les plus connus, ayant traité l'identification d'aspects au niveau du code source orienté objet des systèmes logiciels, tout en cernant leurs points forts et faibles.

Chapitre 4 :

Les préoccupations transversales se manifestent durant les phases du cycle de développement du logiciel, en raison de leur large influence dans la décomposition de logiciel. Le manque de soutien de cette identification, dans les phases antérieures, est un obstacle pour l'application des approches de la modélisation des aspects. Dans ce chapitre, nous présentons une étude comparative des approches les plus connues

d'identification d'aspects durant les phases du cycle de développement du logiciel, ainsi que leurs points forts et faibles.

Chapitre 5.

Ce chapitre présente notre contribution principale dans le domaine d'identification d'aspects. Nous présentons le principe de notre approche basée sur le modèle d'invocations de méthodes, utilisant les diagrammes de séquence, et visant l'identification d'aspects au niveau conceptuel. En outre, et afin de montrer l'applicabilité de notre approche à un niveau d'abstraction plus bas et dans un contexte différent, nous présentons une autre variante de cette approche permettant l'identification d'aspects, par une analyse statique, au niveau implémentation.

Chapitre 6.

La validation de notre approche nous amène à développer un outil permettant son implémentation. Dans ce chapitre, nous présentons les résultats obtenus de l'application de notre approche sur deux études de cas, avec l'utilisation des métriques. Nous vérifions ensuite si ces résultats sont satisfaisants, et respectent les contraintes imposées par les systèmes des études de cas traitées.

La synthèse de nos résultats et/ou les implications et réflexions générales issues des six chapitres, sont exposées dans la conclusion générale de cette thèse.

CHAPITRE 2

CONCEPT DE PRÉOCCUPATION ET PROGRAMMATION ORIENTÉE ASPECT

« L'efficacité d'un projet informatique augmente, si toutes les préoccupations de nature différentes sont bien modularisées, et si un programmeur qui désire faire une modification ne doit parler qu'à ses voisins directs pour la faire, tout en étant sûr de ne pas introduire de bugs.»

(Demeter 1987) [Dem 87]

2.1 Introduction

L'existence des préoccupations enchevêtrées ou dispersées, à travers les classes fonctionnelles du code source orienté objet, et qui ne peuvent pas avoir une encapsulation propre dans le paradigme objet, rend difficile la compréhension du code source objet, sa maintenance, son évolution et sa réutilisation. La programmation orientée aspect a émergé pour supporter la modularité améliorée des systèmes logiciels, en séparant mieux les préoccupations.

Dans ce chapitre, nous commençons par définir les notions de préoccupation et de séparation des préoccupations, pour préciser ensuite le principe et l'intérêt de la programmation orientée aspect. Enfin, nous terminerons par présenter quelques implémentations de cette programmation, et les métriques orientées aspects les plus utilisées.

2.2 Notion de préoccupation

Sutton et ***Rouvellou*** définissent les préoccupations (*concerns*) comme l'ensemble des questions d'intérêt dans un système logiciel [Sut 02]. Une préoccupation peut être directement liée au système ou son environnement [Bus 08]. Elle est définie dans le processus d'ingénierie [Fil 05] comme étant un concept générique décrivant une entité homogène composant le logiciel. Un programme écrit avec la technologie objet [Mad 86] contient deux parties principales.

- Une partie fonctionnelle qui implémente les services pour lesquels l'objet a été écrit; c'est la préoccupation principale de l'objet (préoccupation fonctionnelle). Dans la figure (Fig) 2.1, la couleur verte présente les préoccupations fonctionnelles (logique métier).
- Une autre partie qui adapte l'objet à un environnement et/ou à une application particulière. Cette partie regroupe des préoccupations secondaires, parfois complexes et dispersées dans tout le code source de l'application (préoccupation non fonctionnelle) [Hür 95]. Dans la figure 2.1, les couleurs bleu, jaune et rouge présentent des préoccupations non fonctionnelles (Logging, Persistance et Sécurité).

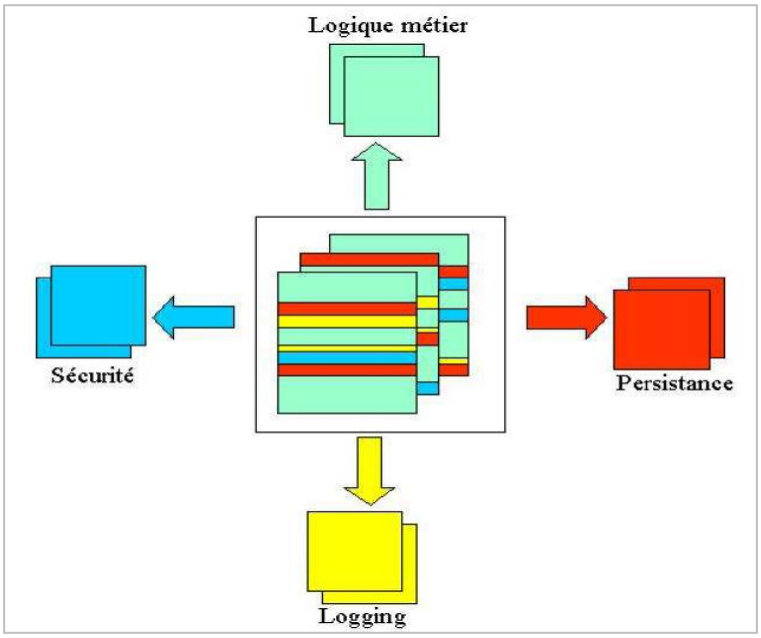


Fig. 2.1. Un système vu comme un ensemble de préoccupations [Poa 13]

Si on prend l'exemple du paiement par carte de crédit [Poa 11], la préoccupation fonctionnelle est : « Effectuer des paiements » (débiter un montant d'un compte défini). Parmi les préoccupations non fonctionnelles, on trouve: « Traçage », « Intégrité de la transaction », « Identification/Authentification », « Sécurité » et « Performances ».

2.3 Caractéristiques d'une préoccupation

Une préoccupation transversale est caractérisée par deux propriétés, qui sont l'enchevêtrement et la dispersion [Ber 05a].

2.3.1 Enchevêtrement (*Tangling*)

La programmation orientée objet (POO) sépare les données et les traitements associés dans des classes. Ces dernières peuvent être corrélées. C'est typiquement le cas des contraintes d'intégrité référentielle. Par exemple, un objet client ne doit pas être supprimé tant qu'une commande pour ce client n'est pas prise en compte, sinon on risque de perdre les coordonnées du client [Paw 04].

La classe client n'est pas le meilleur lieu pour l'implémentation de la contrainte d'intégrité référentielle empêchant la suppression d'un client n'ayant pas toutes ses factures traitées, car la vérification d'une commande non traitée n'appartient pas à la logique de gestion d'un client. Si on modifie la classe client pour tenir compte des contraintes d'intégrité introduites par les autres classes de l'application, elle sera dépendante et, par conséquent, non réutilisable d'une manière indépendante. Même la classe commande n'est pas davantage adaptée à l'implémentation de cette fonctionnalité : si on cherche à supprimer un client, il n'y a aucune raison pour que la classe commande permette de le faire. Finalement, ni la classe client ni la classe commande ne sont adaptées à l'implémentation de la contrainte d'intégrité référentielle. Cette contrainte est enchevêtrée dans ces deux classes.

Alors que le découpage des données en classes a pour but de rendre les classes indépendantes entre elles, on constate que des fonctionnalités enchevêtrées, telles que les contraintes d'intégrité référentielle, viennent se superposer à ce découpage et brisent

l'indépendance des classes. La POO ne fournit aucune solution pour proprement prendre en compte ces fonctionnalités enchevêtrées [Paw 04].

L'enchevêtrement se produit quand deux ou plusieurs fonctionnalités sont contenues dans une même autre fonctionnalité (au niveau implémentation, deux ou plusieurs préoccupations différentes seront mises en œuvre dans le même corps du code d'un module, ce qui le rend plus difficile à comprendre).

2.3.2 Dispersion (*Scattering*)

En POO, le mécanisme principal d'interaction entre objets est l'invocation de méthodes. Un objet qui souhaite effectuer un traitement invoque une méthode d'un autre objet. Un objet peut aussi invoquer une de ses propres méthodes. Dans tous les cas, il existe un rôle d'invocateur et un rôle d'invoqué [Paw 04]. Dans l'invocation d'une méthode, il suffit que les paramètres transmis par l'invocateur soient conformes à la signature de cette méthode, pour que l'invoqué prenne en charge la demande.

L'implémentation d'une méthode en POO est localisée dans une classe. La modification d'une méthode est une opération simple : il suffit de modifier le seul fichier qui contient la classe dans laquelle est définie la méthode. Lorsque la modification porte sur le code de la méthode, elle devient transparente pour tous les objets qui invoquent cette méthode. La modification de la signature de la méthode nécessite de modifier toutes les classes qui invoquent cette méthode. Ces modifications deviennent plus coûteuses si la méthode est d'usage courant.

Par conséquent, en POO, l'implémentation d'une méthode est localisée dans une classe, tandis que son invocation, ou utilisation, est dispersée. Ce phénomène de dispersion du code est un frein à la maintenance et l'évolution des applications orientées objets. Toute modification dans la manière d'utiliser un service entraîne des modifications nombreuses, coûteuses et sujettes à erreur [Paw 04].

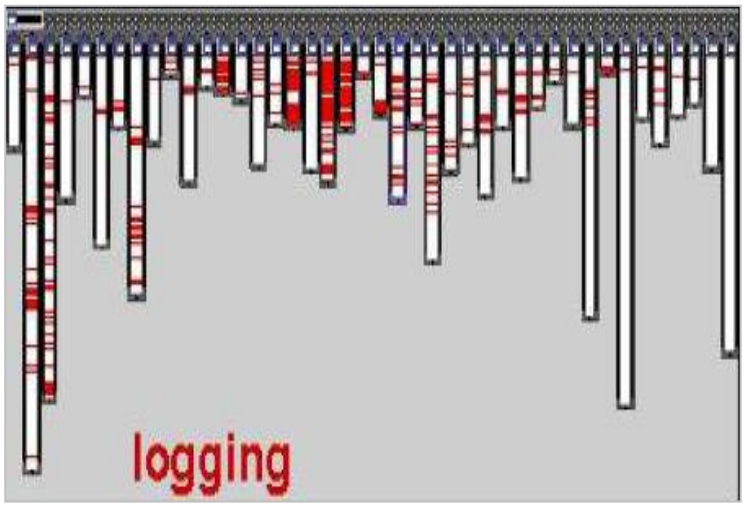


Fig. 2.2. Dispersion de la fonctionnalité Logging dans le projet Jakarta Tomcat [Kic 01b]

La dispersion se produit quand une même fonctionnalité est dispersée à travers le système (au niveau implémentation, un code similaire sera distribué à travers de nombreux modules du programme). Un des exemples les plus connus de cette situation est le projet *Tomcat* [Tom 12] (figure 2.2), où la fonctionnalité *Logging* (les lignes rouges dans la figure 2.2 représentent les lignes du code lié à la fonctionnalité *Logging*) est dispersée à travers presque tous les modules du système (rectangles verticaux blancs).

2.4 Séparation des préoccupations

La séparation des préoccupations, introduite par ***Parnas*** [Pam 72] et ***Dijkstra*** [Dij 76], est considérée comme l'une des approches les plus prometteuses du génie logiciel. La séparation (initiale) des préoccupations permet de décomposer un système en unités fonctionnelles qui encapsulent son comportement. Ces unités fonctionnelles sont traduites initialement dans des procédures et fonctions (en utilisant la programmation structurée [Dah 72]), ensuite dans des objets et classes (avec l'introduction du paradigme orienté objet [Mad 86]).

Cependant, il y a certaines caractéristiques (citées ci-dessus) qui ne peuvent pas être encapsulées dans ces unités, car transversales à travers le système. Les exemples typiques de ces caractéristiques sont les exigences non fonctionnelles, tels que la synchronisation, la coordination, la sécurité et le logging.

De nombreux chercheurs ont reconnu les problèmes imposés par l'existence de l'enchevêtrement et la dispersion, et ont commencé à y faire face. Durant ces dernières années, plusieurs technologies ont apparu pour résoudre ces problèmes. Ces technologies sont connues comme les techniques de « la séparation avancée des préoccupations ».

La séparation avancée des préoccupations vise à améliorer l'évolution et la lisibilité des logiciels, en encapsulant chacune des préoccupations indépendamment les unes des autres. Elle permet la simplification du travail du programmeur et la réutilisation de code existant [Hür 95]. En séparant les préoccupations dans le paradigme orienté objet, il faudrait que l'objet contienne uniquement sa partie fonctionnelle, et que son adaptation soit effectuée à l'extérieur de l'objet, ce qui permet, par conséquent, d'une part, de réduire le travail et l'expertise demandés au programmeur de l'objet, d'autre part, de mieux réutiliser la partie fonctionnelle des objets dans d'autres environnements et d'autres applications. Parmi les principales approches actuelles de la séparation avancée des préoccupations, nous pouvons en retenir quelques unes.

La composition de filtres (*Composition Filters, CF*) [Ber 01] ajoute à l'objet une interface contenant des filtres. Ces derniers interceptent et manipulent les messages, en modifiant leurs portés (objets cibles) et les comportements prévus (sélecteurs du message). A travers ce contrôle des messages, et en utilisant une interface bien construite, la composition de filtres fournit une solution convenable pour résoudre les problèmes posés par le paradigme orienté objet, en considérant les objets de base comme des fonctionnalités, tandis que les interfaces ajoutées sont les préoccupations [Mes 07]. Exemples d'implémentation : *Sina/St* [Koo 95] et *ComposeJ* [Wic 99].

La séparation multidimensionnelle des préoccupations *(Multi-DimensionalSeparation of Concerns, MDSOC)* [Tar 99] est basée sur le principe selon lequel, tous les artefacts du développement de logiciel sont multiples, constitués de préoccupations qui se chevauchent. Ainsi, le développement du logiciel s'améliore si les systèmes logiciels peuvent être décomposés et composés selon les différentes combinaisons et organisations possibles des préoccupations [Sut 03]. Les auteurs utilisent le terme « séparation multidimensionnelle des préoccupations » pour désigner une séparation, modularisation et intégration souples et progressives des artefacts du logiciel basés sur un certain nombre

de préoccupations. Ces mécanismes permettent une séparation nette des préoccupations multiples, se chevauchant et interagissant potentiellement en même temps, en supportant la remodularisation (à la demande), afin d'encapsuler de nouvelles préoccupations à tout moment [Bri 08]. Exemple d'implémentation : *Hyper/J* [Oss 01].

La programmation orientée aspect *(Aspect Oriented Programming, AOP)* [Kic 97] est l'une des approches de la séparation (avancée) de préoccupations les plus étudiées. Dans la section suivante, nous présenterons avec plus de détails cette approche.

2.5 Programmation orientée aspect

2.5.1 Principe

La programmation orientée aspect (POA) [Kic 97] est devenue une des approches les plus connues, et le terme « aspect » est devenu un standard pour dénoter les préoccupations transversales. En fait, la plupart des approches de la séparation avancée des préoccupations sont considérées comme des techniques spécifiques pour la programmation orientée aspect [Con 10].

La programmation orientée aspect est devenue aujourd'hui, un moyen essentiel pour gérer la complexité et l'évolution des systèmes logiciels, en séparant les préoccupations. Les concepts de la POA ont été formulés par ***Gregor Kiczales*** et son équipe, travaillant pour le *Xerox* (entreprise américaine). Comme tous les paradigmes de programmation, la POA repose sur des notions existantes, et un certain nombre d'idées latentes, qui ont déjà été exploitées dans le passé, mais de manière plus ponctuelle.

Les principes de la POA sont résumés par la loi de ***Demeter*** 1987 [Dem 87] :

« L'efficacité d'un projet informatique augmente, si toutes les préoccupations de nature différentes sont bien modularisées, et si un programmeur qui désire faire une modification ne doit parler qu'à ses voisins directs pour la faire, tout en étant sûr de ne pas introduire de bugs.»

Cette loi signifie que l'agrégation homogène d'informations disparates, centrée sur la similarité des préoccupations, est de nature à augmenter l'efficacité du projet

informatique. Il en résulte, nécessairement, une plus grande facilité et fiabilité à modifier un programme, par simple recours aux spécialistes du module à modifier.

Gregor Kiczales [Kic 97] a défini le rôle de la POA comme un moyen d'implémenter les préoccupations transversales : « *Aspect oriented programming has been proposed as a mechanism that enables the modular implementation of crosscutting concerns* ».

Afin d'isoler les préoccupations d'une application et éviter la dispersion et l'enchevêtrement, la solution employée se base sur l'inversion des dépendances [Paw 04]. Ce principe introduit une nouvelle manière de programmer. Il faut, en écrivant un programme, interdire au code métier d'invoquer directement le code technique (transversal). L'application ne dépend plus du service technique, mais c'est l'aspect qui dépend de l'application [Paw 04] (figure 2.3). Par conséquent, les objets métiers sont développés comme en POO (dans des classes), mais indépendamment des préoccupations transversales. Ces dernières sont également décrites et développées de manière complètement indépendantes l'une de l'autre et de tout objet métier, et encapsulées séparément dans des unités appelées aspects.

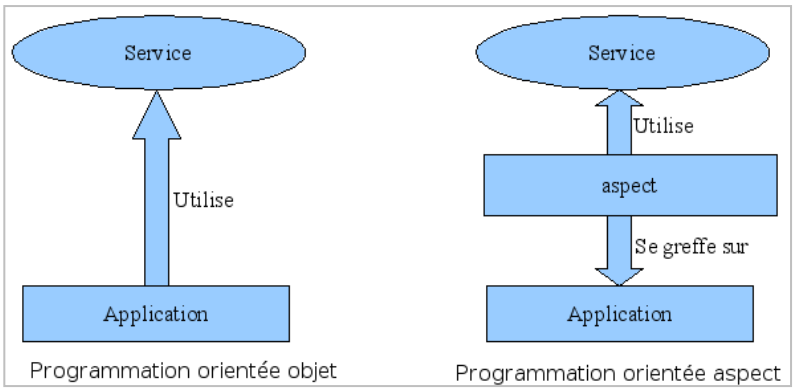


Fig. 2.3. L'inversion de dépendances dans la POA par rapport à la POO [Poa 12]

La POA n'est donc pas un concurrent de la technique source (POO), mais plutôt un cadre logique et cohérent, dans lequel cette technique peut s'intégrer, afin de résoudre des problèmes concrets [Tie 05]. La POA introduit plusieurs concepts, qui seront présentés dans la section suivante.

2.5.2 Concepts de base

2.5.2.1 Aspect

Plusieurs chercheurs ont défini la notion d'aspect, parmi lesquels on peut citer notamment :

Pawlak [Paw 04]:

« Aspect : entité logicielle qui capture une fonctionnalité transversale à une application.»

Filman [Fil 05]:

« An aspect is a modular unit designed to implement a concern. An aspect definition may contain some code and the instruction on where, when and how to invoke it. »

Gradecki [Gra 03]:

« A structure analogous to a Java class that encapsulates joinpoints, pointcuts, advice and inter-type declaration. »

Un aspect est une entité modulaire qui encapsule une préoccupation transversale. Une application en POA est composée d'un ensemble de classes et d'aspects. Pour obtenir une application opérationnelle, intégrant les fonctionnalités des classes et celles des aspects, une opération de tissage est nécessaire.

Le tissage peut être effectué à la compilation ou à l'exécution. Dans le premier cas, le tisseur est un programme, qui, avant l'exécution, prend en entrée un ensemble de classes et un ensemble d'aspects, et fournit en sortie une application augmentée d'aspects.

Le tissage à l'exécution permet d'exécuter, à la fois, l'application et les aspects qui lui ont été ajoutés. Les aspects ont une existence propre lors de l'exécution [Paw 04].

2.5.2.2 Point de jonction

Les points de jonction sont des points du programme autour desquels, un ou plusieurs aspects peuvent être ajoutés. Ils peuvent être de différents types, faisant référence à des évènements du programme. Les plus courants sont les appels de méthodes [POA 15], car ils constituent les éléments principaux de l'exécution d'un programme.

Les types de point de jonction les plus utilisés en POA sont [Paw 04]:

- Méthodes : les événements liés aux méthodes peuvent être des emplacements, autour desquels des aspects peuvent être greffés. Ces évènements sont l'appel, le début et la fin d'une méthode.
- Constructeurs : Les constructeurs sont des méthodes d'initialisation des données et des ressources utilisées par l'objet. Semblables aux méthodes, les appels, les débuts et les fins de constructeurs font partie des points de jonctions.
- Exceptions : la levée et la récupération d'une exception constituent des points de jonction. Elles sont utilisées, par exemple, pour un aspect de sûreté de fonctionnement. Dans ce cas, tous les points qui correspondent à la levée d'une exception peuvent être regroupés, afin de mettre en œuvre une politique globale de gestion de cette exception.
- Attributs : les langages orientés aspect considèrent les opérations de lecture et d'écriture sur les attributs comme des types de points de jonction.

2.5.2.3 Point de coupure

Si un point de jonction correspond à un point particulier du programme, un point de coupure (coupe) (*pointcut*) désigne quant à lui un ensemble de points de jonction.

Il existe plusieurs types de points de coupures, telles que les coupes d'appels de méthodes, les coupes d'exécution de méthodes et les coupes de modification de données désignant des instructions d'écriture sur un ensemble d'attributs.

La différence entre les coupes d'exécution et d'appel de méthodes est le contexte dans lequel s'inscrit le programme. Dans le cas de coupe d'appel de méthodes, le contexte sera celui qui a appelé la méthode, et dans le cas de coupe d'exécution, le contexte sera celui de la méthode appelée [Paw 04].

2.5.2.4 Consigne (*code advice*)

Les codes advice, ou encore les consignes, sont des blocs de code qui exécuteront un aspect. Les codes advice qui composent un aspect vont caractériser le comportement de celui-ci. Il faut que chaque code advice d'un aspect soit associé à une coupe, pour être exécuté, et il ne sera exécuté que si un évènement défini par un point de coupure est

intercepté. De plus, il est possible de spécifier à quel moment ces codes advice doivent être exécutés, avec l'un des types suivants [Paw 04]:

- Le type *before* : un code advice de type *before* exécute son bloc de code avant chacun de ses points de jonction.
- Le type *after* : un code advice de ce type exécute son bloc de code après chacun de ses points de jonction.
- Le type *around* : ici, le code advice exécute son bloc de code avant et après chacun de ses points de jonction. Le mot-clé `proceed` permet de délimiter les parties avant et après. Ce mot-clé correspond à l'exécution du point de jonction : la partie avant est d'abord exécutée, puis vient le point de jonction via l'utilisation de `proceed`, et la partie après.

2.5.2.5 Mécanisme d'introduction

Le mécanisme d'introduction est un mécanisme d'extension, permettant d'étendre une classe, en lui ajoutant de nouveaux éléments, mais ne permet pas de redéfinir une méthode [Paw 04]. Il permet donc d'étendre la structure d'une application, et non pas le comportement de cette dernière.

En effet, le mécanisme d'introduction ne s'appuie pas sur la notion de coupe, mais va opérer sur des emplacements bien définis dans le programme. Contrairement au code advice qui étend le comportement d'une application si et seulement si certains points de jonction sont exécutés, le mécanisme d'introduction est sans condition. L'exécution est réalisée dans tous les cas [Paw 04].

La POA est supportée par plusieurs implémentations, parmi lesquelles nous pouvons citer : *AspectJ* [Kic 01a], *AspectC++* [Gal 01] et *JAC* (Java Aspect Components) [Jac 14].

2.5.3 Langage orienté aspect : *AspectJ*

La programmation orientée aspect est un domaine où la recherche est très active. *AspectJ* [Kic 01a] a toujours été considéré par ***Gregor Kiczales*** comme le projet permettant

d'illustrer les concepts de la POA. La première version d'AspectJ a été diffusée en 1998. Le langage AspectJ est basé sur Java, avec quelques extensions syntaxiques. Les aspects sont modularisés dans des unités séparées. Le tisseur est un recompilateur qui produit le code source valide de Java. Nous présentons dans ce qui suit quelques notions de base du langage AspectJ [Paw 04].

- Points de coupure : AspectJ permet de décrire des coupes plus précises. A l'intérieur d'un aspect, le mot clé `pointcut` définit un point de coupure. Un aspect peut avoir plusieurs points de coupure. Chaque point de coupure est associé à une expression, qui définit son ensemble de points de jonctions.
- Points de jonction : il est nécessaire que les points de jonction soient exécutés pour que les codes advice le soient également, sinon l'aspect n'a aucun sens. Les types de points de jonction fournis par AspectJ peuvent concerner des méthodes, des attributs, des exceptions, des constructeurs ou des blocs de code `static`. Le tableau (Tab) 2.1 récapitule les types de points de jonction offerts par AspectJ.
- Code advice : un code advice est un bloc de code, qui peut contenir toutes les instructions possibles en Java, tel que l'appel de méthode et l'affectation de variable. Un mot clé est ajouté par AspectJ : `proceed`. Toutes les instructions qui précèdent `proceed` sont exécutées avant le point de jonction, et toutes celles qui suivent `proceed` le sont après.
- Mécanisme d'introduction : le mécanisme d'introduction est désigné en AspectJ sous le terme de déclaration intertype. AspectJ ne fournit pas de mot clé pour désigner une introduction.
- Compilation : AspectJ est un tisseur d'aspects qui intervient au moment de la compilation. Les aspects sont ajoutés aux classes pour produire une application finale, dans laquelle les aspects et les classes sont tissés. L'application finale peut être exécutée comme une application normale de Java.
- Aspects abstraits : les aspects en AspectJ peuvent être considérés comme abstraits (tels que les classes), avec le mot clé `abstract` (`abstract aspect`). Il s'agit de définir un aspect dont certains éléments (méthodes ou points de coupure) ne le

sont pas. La définition concrète de ces éléments est effectuée ultérieurement dans un sous-aspect. Cela permet de factoriser un ensemble de définitions communes à d'autres aspects.

- Héritage d'aspect : comme pour les classes, le mot clé `extends` permet d'hériter d'un aspect, afin d'enrichir, redéfinir ou spécialiser le comportement de cet aspect, sans avoir à le réécrire complètement. Seul l'héritage simple d'aspect est supporté.
- Instanciation d'aspect : un aspect peut avoir plusieurs instances, en utilisant des mots clés : `perthis` et `pertarget` pour désigner que chaque instance est associée à des groupes d'objets de l'application, et `percflow` et `percflowbelow` pour préciser que chaque instance est associée à des flots de contrôle de l'application.
- Ordonnancement d'aspect : lorsque deux ou plusieurs aspects interviennent sur un même point de jonction, AspectJ fournit deux mécanismes pour ordonner ces aspects : soit d'une manière explicite, en laissant le programmeur fournir un ordre, soit d'une manière implicite, en appliquant un certain nombre de règles prédéterminées.
- Aspect privilégié : un aspect `privileged` peut accéder à tous les attributs et les méthodes d'une application, quelle que soit leur lisibilité (privés, protégés, etc).

Tab. 2.1. Tableau récapitulatif des points de jonction possibles et la syntaxe à utiliser pour définir la coupe [Paw 04]

Syntaxe	Description du point de jonction
`call``(methodeExpression)`	Appel d'une méthode, dont le nom vérifie `methodeExpression`.
`execution``(methodeExpression)`	Exécution d'une méthode, dont le nom vérifie `methodeExpression`.
`get``(attributExpression)`	Lecture d'un attribut, dont le nom vérifie `attributExpression`.
`set``(attributExpression)`	Ecriture d'un attribut, dont le nom vérifie `attributExpression`.

handler(exceptionExpression)	Exécution d'un bloc de récupération d'une exception, dont le nom vérifie exceptionExpression.
initialization(constanteExpression)	Exécution d'un constructeur de classe, dont le nom vérifie constanteExpression.
preinitialization(constanteExpression)	Exécution d'un constructeur hérité, dont le nom vérifie constanteExpression.
staticinitialization(classeExpression)	Exécution d'un bloc de code static dans une classe, dont le nom vérifie classeExpression.
adviceexecution()	Exécution d'un code advice.

2.6 Quelques métriques orientées préoccupation

Les métriques (*metrics*) orientées préoccupation ont été largement utilisées dans la littérature. Avec l'émergence du paradigme orienté aspect, certaines de ces métriques restent utiles (éventuellement adaptées) tel que le couplage, d'autres sont devenues incapables de représenter les nouveaux concepts de ce paradigme, tandis que de nouvelles ont vu le jour, telles que les métriques de dispersion et d'enchevêtrement (introduites dans la section 4.2.2). Dans ce qui suit, nous présenterons les métriques les plus utilisées.

2.6.1 CONT et LOCC

La métrique *CONT* (*program element CONTribution*) [Ead 08b] énumère le nombre de lignes du code (exceptés les commentaires et les lignes blanches) dans un élément de programme, qui sont associées avec une préoccupation. La ligne entière est prise en compte, même si seulement une partie est associée à la préoccupation. En effet, une ligne peut être associée à de multiples préoccupations. Les lignes en dehors de la définition de la classe (par exemple: la déclaration de package, les importations) ne sont pas comptées.

Lorsque l'élément est l'ensemble du programme P, la contribution est la somme des contributions de tous les éléments, c'est à dire le nombre total de lignes de code associées avec la préoccupation c. Dans ce cas, cette métrique est appelée *LOCC* (*Lines Of Concern Code*) [Ead 08b], et calculée comme suit : $LOCC(c) = CONT(c, P)$.

2.6.2 Diffusion

La diffusion (*diffusion*) [San 03] est principalement utilisée pour mesurer la dispersion d'une préoccupation, du fait qu'elle compte le nombre des artefacts du logiciel liés à la préoccupation.

La métrique *CDOC* (*Concern Diffusion Over Components*) mesure le nombre de composants qui contribuent à l'implémentation d'une préoccupation. Cette métrique compte aussi le nombre de composants qui accèdent aux composants primaires, par exemple : instanciation, appels de méthodes, déclarations d'attributs.

La métrique *CDO* (*Concern Diffusion over Operations*), similaire à *CDOC*, compte le nombre des opérations qui contribuent à la fonctionnalité d'une préoccupation, ainsi que les méthodes et les codes advice qui accèdent à ces opérations.

Quant à la métrique *CDLOC* (*Concern Diffusion over Lines of Code*), elle compte, pour chaque préoccupation, le nombre de lignes de code qui implémentent des préoccupations différentes.

La valeur élevée de la métrique de diffusion signifie que la préoccupation est plus dispersée. En outre, l'utilisation de la métrique *CDLOC* peut mesurer l'enchevêtrement, du fait qu'elle fournit une indication sur le nombre de préoccupations implémentées dans un module.

2.6.3 LCOO

La métrique *cohesion* [San 03] est définie comme le rapprochement entre les composants internes d'un module. Dans ce sens, une nouvelle métrique est définie pour mesurer la cohésion dans le domaine orienté aspect, qui est la métrique *LCOO* (*Lack of Cohesion in Operations*) [San 03]. *LCOO* est définie comme suit :

Soit C_1 un composant avec n opérations (méthodes et codes advice) $o_1, \dots, o_n$ et $\{I_j\}$ l'ensemble des variables d'instance utilisées par l'opération o_j. Soit $|P|$ le nombre des intersections nulles entre les ensembles des variables d'instance. $|Q|$ est le nombre des intersections non nulles entre les ensembles des variables d'instance. $LCOO = |P| -$

$|Q|$, si $|P| > |Q|$, sinon $LCOO = 0$. Plus cette métrique est élevée, moins la cohésion d'une opération est présente. Une opération moins cohésive signifie une mauvaise modularité de la conception.

2.6.4 Couplage

Le couplage (*coupling*) est une métrique qui indique la force des interactions entre les composants du logiciel [Ste 74]. Parmi les différentes définitions données, on peut rappeler celle de [Cec 04], où le couplage est déterminé en utilisant les appels de méthodes (*Coupling on Method Call CMC*). Il comptabilise le nombre de modules (classes ou aspects) ou interfaces qui déclarent des méthodes appelées par un module donné. La métrique *CMC* considère les dépendances d'un module donné avec d'autres modules, en termes de méthodes.

Le nombre élevé des méthodes de plusieurs modules, indique que la fonctionnalité du module donné ne peut être facilement isolée des autres. Le couplage élevé est associé avec une dépendance élevée des fonctions dans d'autres modules. La métrique *CMC* se base sur le niveau programmation, et elle doit être appliquée aux systèmes modélisés en utilisant les techniques orientées aspect.

2.7 Conclusion

L'accent a été mis, tout au long de ce chapitre, sur la notion de préoccupation, ainsi que les principes du paradigme orienté aspect, ainsi que quelques métriques orientées aspect.

L'émergence du paradigme orienté aspect a poussé les chercheurs vers plus de précision de la notion de transversalité, afin de faciliter la détection des préoccupations transversales et bénéficier de ce paradigme. Dans le prochain chapitre, nous exploiterons les travaux les plus connus qui ont abordé l'identification des préoccupations transversales (aspects) au niveau du code source des systèmes logiciels.

CHAPITRE 3

IDENTIFICATION D'ASPECTS À PARTIR DU CODE SOURCE ORIENTÉ OBJET : ÉTUDE COMPARATIVE DES APPROCHES

« Understanding a software system at source code level requires understanding the different concerns that it addresses, which in turn requires a way to identify these concerns in the source code.»
(Ceccato et al. 2006) [Cec 06]

3.1 Introduction

La rétro-ingénierie, en général, se définit comme le processus d'analyse d'un système logiciel, afin d'identifier ses composants et ses relations internes, et créer des représentations sous d'autres formes, souvent à un plus haut niveau d'abstraction. L'objectif étant d'effectuer la compréhension du système logiciel, il s'agit de retrouver les décisions de conception à partir du code source seulement, ou avec peu de connaissances supplémentaires sur sa production [Bou 07].

Le processus d'identification d'aspects, appelé « *aspect mining* » [Han 01], est un processus spécialisé de la rétro-ingénierie du logiciel [Ape 06, Bou 06, Kel 06]. Il

consiste à découvrir, à partir de l'implémentation d'un système logiciel orienté objet donné, les préoccupations transversales entrecoupant les classes fonctionnelles qui, potentiellement, pourraient devenir des aspects (aspects candidats).

Beaucoup sont les approches ayant visé l'identification d'aspects à partir du code source orienté objet. Dans ce chapitre, nous présenterons la définition de la transversalité au niveau code source, et nous exploiterons ensuite les approches les plus connues d'aspect mining, afin d'extraire leurs points forts et faibles.

3.2 Définitions de la transversalité

Plusieurs définitions ont été proposées à la transversalité au niveau code source. Dans [Ber 05b], une ontologie définit les termes communs de la communauté orientée aspect, et précise notamment les termes « transversalité » ou « préoccupation transversale ».

La transversalité est définie comme la dispersion et/ou l'enchevêtrement des préoccupations, résultant de l'incapacité de la décomposition choisie à modulariser efficacement ces préoccupations. La préoccupation transversale est définie comme une préoccupation qui ne peut pas être représentée dans la décomposition choisie. Par conséquent, les éléments des préoccupations transversales sont dispersés et enchevêtrés dans les éléments d'autres préoccupations [Ber 05b]

Dans ce qui suit, nous présentons quelques définitions de la transversalité, basées sur le code source, et qui seront complétées dans les sections 3.5.

3.2.1 Définition basée sur la projection

La transversalité se produit entre deux modules lorsque leurs projections, dans un domaine résultat, se croisent sans qu'aucune projection ne puisse être sous ensemble de l'autre [Mas 03]. Cette définition a été fournie au niveau programmation.

Masuhara et ***Kiczales*** dans [Mas 03] modélisent les mécanismes orientés aspect comme un tissage combinant deux programmes différents (A et B, l'un d'entre eux étant orienté aspect), et produisant comme résultat un autre programme (X). En se basant sur ce

processus de tissage, la définition de la transversalité fournie est basée sur le terme projection. Cette dernière est définie comme suit : « Pour un module m_A (du programme P_A écrit dans le langage A), on dit que la projection de m_A dans X est l'ensemble des points de jonction identifiés par les éléments A_{ID} dans m_A », tels que A_{ID} sont les moyens (dans le langage A) d'identification des points de jonction dans X (dans l'orienté objet sont les méthodes et les champs des signatures).

Les auteurs dans [Mas 03] utilisent l'exemple d'affichage des figures pour illustrer les mécanismes du code advice et les points de coupure dans AspectJ (figure 3.1). La classe $Point$ et le code de la mise à jour de l'affichage s'entrecoupent dans le domaine résultat X.

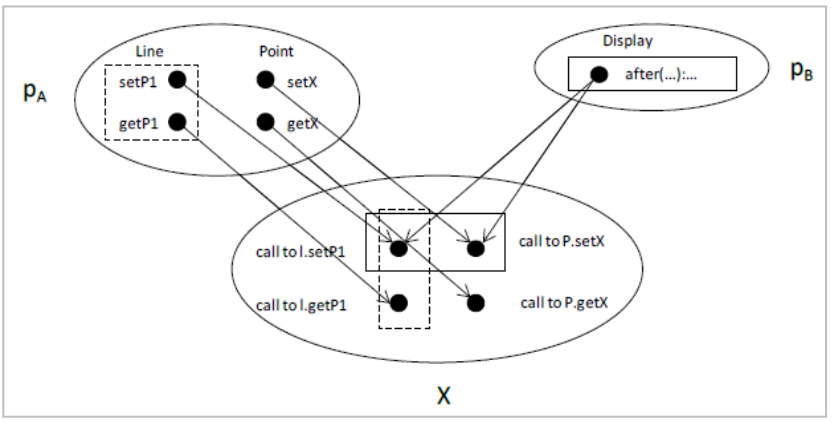


Fig. 3.1. La classe $Point$ et le code advice de la mise à jour de l'affichage s'entrecoupent dans le domaine résultat X [Mas 03]

3.2.2 Définition basée sur la décomposition

La transversalité se produit lorsque les décompositions des éléments du code source, selon différents critères, se croisent. Cela est dû à l'arbitraire dans la hiérarchie de la décomposition [Mez 03]. Cette définition de la transversalité proposée par ***Mezini*** et ***Ostermann*** dans [Mez 03] utilise un framework qui prend de multiples décompositions simultanément, et qui est basé sur les termes : modèle et espace de préoccupations. Ce dernier représente les artefacts qui sont composés pour obtenir le système final.

Les auteurs utilisent un exemple abstrait avec différentes figures, pour représenter l'espace de préoccupations (espace abstrait de préoccupation) (figure 3.2). Un modèle est

défini comme le résultat d'une décomposition du système selon un critère particulier. En suivant l'exemple abstrait, le modèle est assimilé au résultat de la classification de l'ensemble de figures, selon un des trois critères différents : couleur, forme et taille. Comme exemple, la figure 3.3 présente le modèle résultat à partir de la décomposition de l'espace abstrait selon la préoccupation couleur.

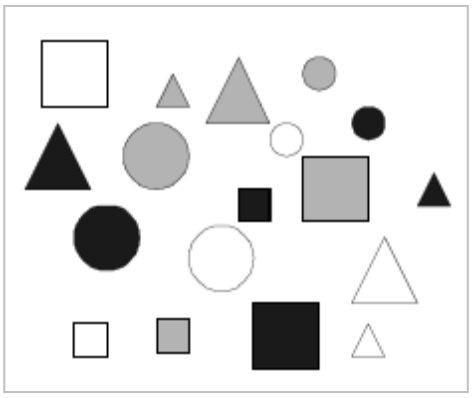


Fig. 3.2. Exemple d'un espace abstrait de préoccupations [Mez 03]

Fig. 3.3. Modèle de la préoccupation « couleur »

Les auteurs affirment que les modèles résultant de la décomposition simultanée du système, selon différents critères, sont généralement transversaux par rapport à l'exécution du système [Mez 03]. Ils définissent la transversalité comme une relation entre deux modèles par rapport à l'espace abstrait de préoccupations, comme suit : « deux modèles s'entrecoupent entre eux (sont transversaux) s'il existe au moins deux ensembles $e1$ et $e2$ de leurs projections respectives, tel que : $e1 \cap e2 \neq \emptyset$, et ni $e1 \subseteq e2$, ni $e2 \subseteq e1$ ». Comme on peut le voir dans la figure 3.4, l'ensemble dans le premier rectangle gauche (soit $e1$) qui représente la couleur blanche, et l'ensemble dans la forme bleue

(soit $e2$) qui représente la taille grande, se coupent. Par conséquent, les modèles de préoccupations couleur (en rectangles) et taille (en formes bleues) sont transversaux.

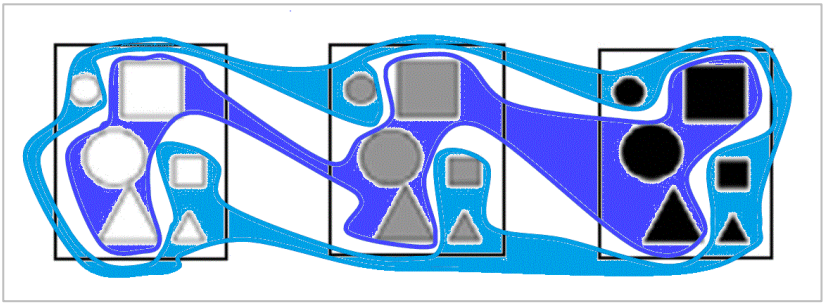


Fig. 3.4. Modèles de préoccupations « couleur » (rectangles) et « taille » (formes bleues) qui sont transversaux

3.3 Problèmes posés par la transversalité

La dispersion et l'enchevêtrement brisent l'indépendance des classes, ce qui pose un certain nombre de problèmes, parmi lesquels nous citons les suivants:

- La programmation du code transversal est difficile et complexe, car tous les problèmes doivent être traités en même temps.
- Le code transversal est difficile à comprendre, à cause du manque d'abstraction.
- Le code transversal est difficile à maintenir et à modifier, car les préoccupations sont fortement couplées, et elles sont difficiles à identifier, vu le caractère implicite de la correspondance entre les exigences et leurs implémentations.
- Le code enchevêtré provoque l'héritage des anomalies dues au couplage fort des préoccupations différentes. Il devient impossible de redéfinir l'implémentation d'une méthode, ou une préoccupation particulière mélangée dans une sous-classe, sans redéfinir toutes les deux.
- La prise en compte de plusieurs exigences au sein d'un même module, empêche le programmeur de se focaliser sur une exigence à la fois, ce qui entraine une diminution de la productivité.

- Lorsqu'un module implémente multiples exigences, d'autres systèmes nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser ce module dans l'état où il est (une difficulté ou impossibilité de réutiliser ce module).

3.4 Navigateurs dédiés

Les premiers outils d'aspect mining qui ont été proposés sont les navigateurs dédiés. Ces derniers sont les navigateurs du code, qui aident l'ingénieur à trouver des préoccupations transversales, afin de maintenir et de faire évoluer le système logiciel. La plupart de ces navigateurs exigent de l'utilisateur l'établissement d'une « graine » sur une préoccupation, afin d'avoir un point de départ pour explorer tout le code. On peut citer quelques exemples de ces navigateurs : *Concern Graphs* [Rob 02], *Aspect Browser* [Yos 99], *Aspect Mining Tool* [Han 01] et *Prism* [Zha 03].

3.4.1 Concern Graphs

Les graphes de préoccupation (*concern graphs*) sont basés sur la localisation d'une représentation abstraite des éléments du programme, contribuant à l'implémentation de la préoccupation [Rob 02]. La structure d'une préoccupation est stockée dans un graphe et, en même temps, les relations entre les éléments de préoccupation, tels que les classes, les méthodes et les champs sont documentées. Cette approche est implémentée dans l'outil FEAT (*Feature Exploration and Analysis Tool*). FEAT consiste en un plug-in Eclipse. Cet outil supporte l'analyse de dépendances entre une préoccupation et le reste du programme, et permet de visualiser le code source dans Java associé à un élément du graphe de préoccupation. Pour identifier une préoccupation, FEAT supporte l'utilisation des requêtes structurelles et des recherches lexicales intégrées. L'outil affiche un graphe de préoccupation comme une collection d'arbres qui respectent certaines conventions, par exemple la racine de chaque arbre est une classe qui contribue à l'implémentation d'une préoccupation. Puisque cet outil fournit une représentation abstraite de différentes parties d'un système contribuant à une préoccupation particulière, une préoccupation dispersée peut être identifiée, en observant cette représentation abstraite. Cependant, l'outil ne

fournit pas explicitement une vue pour traiter les préoccupations transversales, et ne démontre pas comment isoler les préoccupations transversales.

3.4.2 Aspect Browser

Aspect Browser est un outil de localisation et de visualisation des préoccupations transversales dans un système logiciel [Yos 99]. Il est basé sur la supposition que les aspects, définis comme des décisions secondaires de conception, ont une signature (textuelle et lexicale), qui est une expression régulière textuelle, et qui aidera à leur identification. Chaque aspect contient une expression régulière, et le code qui y correspond est marqué avec une couleur (un aspect est défini comme une expression régulière et une couleur).

L'outil analyse le code source, en recherchant les lignes de code redondantes (lignes de code apparaissant plus d'une fois). Ensuite, il analyse les identifiants dans le code qui les sépare en balises. Il présente ensuite différents fichiers du système comme des barres verticales. Les lignes sources du code sont représentées dans ces barres comme des lignes. Lorsqu'une ligne du code est liée à un aspect particulier, l'outil montre une ligne avec la couleur associée à l'aspect. Cette représentation visuelle permet d'avoir une vision globale sur la manière dont les préoccupations sont dispersées, ainsi que le nombre des préoccupations enchevêtrées dans un fichier.

3.4.3 Aspect Mining Tool

Aspect Mining Tool (*AMT*) permet (comme Aspect Browser) de localiser et visualiser les préoccupations transversales dans un système logiciel [Han 01]. Il permet de chercher les aspects candidats en utilisant l'analyse syntaxique (avec des correspondances basées sur les expressions régulières, telle que chaque ligne de code qui correspond à une expression régulière est marquée) et l'analyse basée type. A cet effet, il a été considéré comme un outil multimodal. En utilisant l'analyse basée type, le développeur peut rechercher toutes les occurrences d'un type particulier dans un système. L'utilisation récurrente d'un type indique l'existence possible de préoccupations cachées et non représentées (ne sont pas bien modularisées) dans la décomposition du système actuel.

AMT permet de visualiser l'effet de ces aspects sur les fichiers sources. Chacun de ces derniers est représenté comme une barre verticale, et les lignes du code lié à une préoccupation particulière sont marquées avec des couleurs différentes.

3.4.4 Prism

Prism est un plug-in Eclipse [Zha 03]. L'intervenant (*stakeholder*) agit dans ce cas en ayant connaissance de l'existence des aspects dans le code source. Cet intervenant fournit une description initiale de la structure transversale d'un aspect, en utilisant des expressions lexicales ou des modèles basés types. Les aspects identifiés dans cette phase sont ceux communs dans la plupart des systèmes logiciels, tel que logging, la sécurité et la persistance. Pour les préoccupations transversales spécifiques au domaine d'application, leur identification se base sur la compréhension de ce domaine par l'intervenant.

3.5 Approches d'identification d'aspects

Avec les navigateurs dédiés, l'utilisateur doit avoir une connaissance considérable au sujet de la structure totale du programme analysé. En outre, l'identification d'un aspect nécessite beaucoup de temps dû à l'interaction exigée avec l'utilisateur. Pour ces raisons, les travaux de recherches ont convergé vers l'identification automatique des préoccupations.

Les approches d'identification des préoccupations transversales au niveau implémentation se sont orientées vers l'automatisation du processus d'aspect mining, et visent précisément l'identification des préoccupations entrecoupant les classes fonctionnelles des systèmes logiciels. Elles se concentrent principalement sur l'analyse du code source, ou sur les données obtenues par sa manipulation. Parmi ces approches, nous citons, dans ce qui suit, les plus connues.

3.5.1 Analyse formelle des concepts (FCA)

Notre étude faite sur les techniques d'aspect mining montre que plusieurs travaux se sont intéressés à la classification (partitionnement) des entités du programme. Certains de ces travaux utilisent l'analyse de concepts formels (*Formal Concept Analysis FCA*) pour regrouper les méthodes selon, entre autres, leurs noms ou leurs appels [Dah 11b].

L'analyse de concepts formels [Gan 99] est une méthode de regroupement conceptuelle. Elle a été initialement utilisée dans le domaine de l'intelligence artificielle, pour la représentation et l'acquisition des connaissances [Aré 05]. Le point de départ, ayant inauguré l'émergence de FCA, se situe au niveau du besoin des techniques de hiérarchisation des grands ensembles d'informations (web par exemple). Il en est ainsi car, avec l'explosion des données, à la suite des dernières avancées en matière de stockage, de transmission et d'intégration, les techniques concurrentes de hiérarchisation (tels que le regroupement statistique et les réseaux de neurones artificiels [Mul 91]) se sont révélées moins pertinentes [Tre 14].

FCA constitue une approche théorique de structuration des données, permettant d'identifier des concepts potentiellement intéressants au sein d'un ensemble de données, décrits par une matrice binaire ($objets \times attributs$). Pour ce faire, FCA ramène les données d'une relation binaire, et exploite les propriétés de la correspondance, afin d'en isoler une hiérarchie des concepts, dits formels [Gan 99]. Trois principaux éléments interviennent dans FCA:

- Le contexte formel : représenté par le triplet (O, A, R), où O désigne l'ensemble d'objets, A est l'ensemble des attributs, et R décrit la relation binaire qui relie les objets à leurs attributs. Ce contexte est structuré dans une matrice binaire M, tel que : si un objet i possède la propriété j, $M[i,j] = 1$, sinon $M[i,j] = 0$.
- Le concept : c'est la paire (X, Y), où X est un ensemble d'objets, et Y les attributs de ces objets. Un concept associe un ensemble maximal d'objets, à l'ensemble d'attributs que ces objets partagent [God 95].

- Le treillis des concepts : ou treillis de Galois, est une hiérarchie de concepts. Ces derniers sont ainsi organisés : un attribut, présent dans un concept, est hérité par tous les sous-concepts, et inversement pour les objets [Ngu 04].

FCA a été appliquée aux nombreux problèmes d'ingénierie de logiciels, telles que la restructuration du code en composants plus cohérents, par identification des classes candidates [Deu 99], et la hiérarchisation des classes. Dans le domaine de l'aspect mining, FCA a été utilisée de plusieurs manières :

a. FCA des identifiants :

Selon cette approche, la définition de la transversalité est basée sur la supposition qui dit que des préoccupations intéressantes dans le code source sont reflétées par l'utilisation des conventions de noms des classes et des méthodes du système. C'est pourquoi, l'analyse FCA des noms de classes et de méthodes a été utilisée [Tou 04]).

Dans ce cas, les objets de FCA sont les noms de classes et de méthodes du code source, et les sous chaines générées à partir de ces noms sont les attributs. Les sous chaînes peu signifiantes sont écartées des résultats. Les concepts résultants consistent en des groupes maximaux d'entités du programme, partageant un nombre maximal de sous chaînes. Les aspects candidats sont les classes et les méthodes qui contiennent les sous chaînes récurrentes dans le treillis des concepts (un aspect est défini comme un ensemble de classes et de méthodes qui ont des noms proches, en termes de sous chaines en commun). Cette approche a un support d'outil *DelfSTof* [Men 05].

b. FCA des traces d'exécution :

Une autre approche d'aspect mining applique l'analyse des concepts formels aux traces d'exécution, afin d'identifier les aspects possibles [Ton 04]. L'identification des différents aspects dans le code est supportée par des moyens de l'analyse dynamique du code. Les traces d'exécution sont générées pour les scénarios des cas d'utilisation des principales fonctionnalités de l'application donnée. La relation entre les traces d'exécution et les méthodes appelées est soumise à l'analyse des concepts formels. Les traces sont analysées en utilisant FCA. Les cas d'utilisation sont les

objets de FCA, et les méthodes appelées pendant l'exécution d'un cas d'utilisation sont les attributs. Les concepts qui en résultent, spécifiques à un cas d'utilisation particulier, sont considérés comme des aspects candidats, si les attributs (les méthodes) du concept appartiennent à plus d'une classe (c'est-à-dire dispersés), ou bien les différentes méthodes d'une même classe sont contenues dans plus d'un concept spécifique d'un cas d'utilisation (c'est-à-dire enchevêtrées). En d'autres termes, la dispersion se produit lorsque la trace d'exécution d'un scénario de cas d'utilisation invoque deux méthodes qui appartiennent à au moins deux différentes classes, tandis que l'enchevêtrement se produit quand deux méthodes d'une même classe sont invoquées dans les traces d'exécution des scénarios de deux différents cas d'utilisations. *Dynamo* [Ton 04] est un outil d'aspect mining basé sur l'application de FCA sur les traces d'exécution.

c. FCA des appels :

Une approche alternative a été proposée dans [Dah 11a], qui vise à détecter le code dispersé, en exploitant les relations d'appels. Pour cela, et dans le contexte de FCA (O, A, R), l'ensemble des objets O représente les méthodes appelantes qui existent dans le code source analysé, l'ensemble des attributs A constitue l'ensemble des méthodes appelées du code, et la relation binaire R désigne les appels entre les méthodes appelantes et les méthodes appelées. La matrice binaire M est carrée. Ses lignes et ses colonnes contiennent les noms des méthodes, tel que : si la méthode i appelle la méthode j, $M\ [i,j] = 1$, sinon $M[i,j] = 0$.

Dans un concept (X, Y), X est un ensemble de méthodes appelantes, et Y est l'ensemble des méthodes appelées par ces méthodes appelantes. Le treillis de concepts regroupe les méthodes par appels. Il aide à déterminer les méthodes dont l'appel est dispersé. Les aspects candidats sont tous les attributs (méthodes appelées) de tous les concepts du treillis, qui ont le nombre de leurs objets (nombre des méthodes appelées) supérieur à un certain seuil. Ce dernier doit être supérieur à 1, car un aspect candidat est une méthode appelée plus d'une fois. *ACF-Mining* [Dah 11a] est un outil qui implémente cette approche.

L'utilisation de l'analyse des concepts formels dans le domaine de l'aspect mining est avantageuse. FCA, particulièrement sa branche algorithmique, se distingue par un solide fondement mathématique, outre de bonnes propriétés structurelles du résultat.

Cependant, les approches existantes de l'aspect mining par FCA connaissent des limites. Concernant l'aspect mining par FCA des noms de classes et de, ces noms peuvent ne pas être significatifs et dépendent de la syntaxe du programmeur, tandis que l'aspect mining par FCA des traces d'exécution, celles-ci peuvent être volumineuses. En outre, cette technique nécessite l'analyse des cas d'utilisation, et l'identification de ces derniers n'est parfois pas une tâche précise. FCA des appels de méthodes exploite les appels entre les méthodes, et ces derniers portent une information intéressante sur le code, indépendante de son programmeur. Cependant, cette approche ne prend pas en compte l'enchevêtrement.

3.5.2 Clustering

Le *clustering* est une technique de classification de données fréquemment utilisée dans la fouille de données (*data mining*) [Goe 99]. C'est une méthode de regroupement permettant la découverte des groupes d'objets. Ces objets sont semblables (ou liés) l'un à l'autre lorsqu'ils appartiennent au même groupe, mais différents dans les autres groupes [He 06].

L'objectif général du clustering consiste à séparer un ensemble d'objets en différents groupes (*clusters*), en fonction d'une certaine notion de similarité, de façon que les objets qui sont considérés comme similaires appartiennent au même groupe, alors que ceux qui sont considérés différents sont associés à des groupes distincts [He 06].

Le clustering consiste alors à réduire au minimum la distance entre les objets d'un même groupe, tout en augmentant au maximum la distance entre les groupes. Différentes notions caractérisant le rapprochement de deux objets peuvent être distinguées, telles que la distance et la similarité.

La technique du clustering a été utilisée dans le domaine de l'aspect mining, afin de regrouper les entités du programme (comme il a été fait avec FCA), selon un critère

choisi. ***Cojocar*** et al. dans [Coj 09] considèrent le problème d'identification des préoccupations transversales comme un problème de clustering, et ils cherchent les méthodes similaires d'un système logiciel, afin de les regrouper ensemble et détecter les groupe contenant les préoccupations transversales. Pour découvrir les préoccupations transversales à travers un système, ils ont tout d'abord analysé le code source du système logiciel. Ensuite, ils ont appliqué un algorithme du clustering pour partitionner le système logiciel, dans lequel les éléments appartenant à une préoccupation transversale devraient être groupés ensemble. L'étape finale consiste à analyser les résultats obtenus manuellement. Le clustering a été utilisé de plusieurs manières :

a. Clustering des noms de méthodes :

Shepherd et ***Pollock*** [She 04, She 05] reportent une expérience dans laquelle ils ont utilisé le clustering pour regrouper des méthodes liées. Cette technique commence en plaçant chaque méthode dans un groupe séparé, et récursivement fusionne les groupes pour lesquels la distance entre les noms de méthodes est plus petite qu'un certain seuil. Les auteurs ont implémenté cette technique comme partie d'un IDE orienté aspect nommé *AMAV* (*Aspect Miner And Viewer*) qui permet une adaptation facile de la mesure de la distance utilisée par l'algorithme. Pour une expérience initiale, ils ont utilisé une mesure de distance simple, inversement proportionnelle à la longueur de la sous-chaîne commune aux noms des méthodes. Cet algorithme d'aspect mining est utilisé en combinaison avec l'outil de visualisation de l'IDE qui, non seulement affiche tous les groupes qui ont été trouvés, mais aussi construit une fenêtre qui affiche les méthodes en rapport avec un groupe (le contexte de la classe d'une méthode).

b. Clustering des traces d'exécution :

He et ***Bai*** [He 06] ont appliqué la technique du clustering sur les traces d'exécution. Celles ci sont obtenues comme suit : le système orienté objet est exécuté selon des scénarios spécifiés, et chaque scénario correspond à une séquence d'appels de méthodes. S'il y a un groupe de codes qui a une action similaire (séquence d'appels de méthodes similaire), et qui apparait fréquemment dans les traces d'exécution,

alors une préoccupation transversale peut dans ce cas exister. Les séquences d'appels de méthodes similaires sont le code des préoccupations transversales.

c. Clustering par agrégation :

Un travail dans [Fil 12] propose l'utilisation d'une mesure de distance qui est une fonction d'agrégation des symptômes de dispersion, clonage du code (*code cloning*) (capture les méthodes avec des opérations similaires) et convention de noms, utilisée pour regrouper les méthodes. La distance de dispersion entre deux méthodes est réduite lorsque l'ensemble d'intersection de leurs classes (où elles sont définies), et les méthodes et les classes qui les invoquent, est plus grand. La distance de l'opération de clonage entre deux méthodes est réduite lorsque l'ensemble d'intersection des méthodes, et les classes invoquées par ces méthodes, est plus grand. La distance de nom capture l'utilisation des conventions dans les noms de méthodes, classes, type de retour et de paramètres.

Cependant, les distances définies dans cette approche sont imprécises, car dans la première distance, l'existence d'une méthode invoquée par la classe où la seconde méthode est définie, ou l'existence de deux méthodes invoquées par la même classe, n'indique pas nécessairement que ces deux méthodes appartiennent au même aspect dispersé (leur distance de dispersion est réduite).

Dans la seconde distance, les mêmes méthodes ou classes invoquées par deux méthodes n'indiquent pas nécessairement que ces deux méthodes appartiennent au même aspect de clonage (leur distance de l'opération de clonage est réduite).

Dans la troisième distance, l'auteur se base sur l'analyse syntaxique qui est imprécise, car elle dépend de la syntaxe du programmeur. En outre, l'approche ne détecte pas le symptôme d'enchevêtrement, car avant d'appliquer l'algorithme du clustering, elle élimine les méthodes dont leur fan-in [You 79] est inférieure à 2. Ces méthodes peuvent constituer des préoccupations enchevêtrées.

3.5.3 Fan-in

Dans [Mar 04], ***Marin*** et al. ont présenté un processus d'extraction d'aspects, basé sur l'utilisation de la métrique fan-in [You 79]. Fan-in compte le nombre de modules qui requièrent un module donné.

Les auteurs affirment que beaucoup de préoccupations transversales bien connues sont implémentées en utilisant une technique qui exploite un haut degré de charge d'entrée (métrique fan-in). Ils proposent l'utilisation d'une métrique de charge d'entrée fan-in pour découvrir les préoccupations transversales dans le code source. La métrique fan-in d'une méthode m est définie comme étant le nombre de méthodes distinctes qui peuvent invoquer m. A cause du polymorphisme, un appel à une méthode m influe sur fan-in de toutes les méthodes qui raffinent m. L'algorithme d'aspect mining comprend les étapes suivantes:

- Calcul de la métrique fan-in pour toutes les méthodes du système qui est en cours d'analyse.
- Filtrage automatique des résultats : outre le filtrage des méthodes *accessor* et *mutator*, ainsi que les méthodes utilitaires telles que `toString()`, le nombre de méthodes considérées est aussi limité par le fait que seules les méthodes avec une valeur de fan-in supérieure à un certain seuil (fan-in >10) représentent des aspects candidats.
- Filtrage manuel : les méthodes restantes seront analysées manuellement.

Les auteurs présentent une expérience dans laquelle les préoccupations transversales sont identifiées : les méthodes avec fan-in élevé étaient des graines d'information indiquant l'existence d'un aspect. L'aspect avec un petit empreint sera ignoré par cette approche.

Cependant, les méthodes ayant un fan-in inférieur au seuil défini (qui ne peuvent pas être considérées par cette technique comme aspects candidats) peuvent construire des préoccupations enchevêtrées. Par conséquent, cette technique ne détecte que le symptôme de dispersion. En outre, l'ensemble des résultats est lié au seuil de fan-in choisi. Ce dernier nécessite une connaissance préalable et une certaine sémantique.

3.5.4 Modèle récurrent

L'approche est basée sur l'analyse dynamique des traces d'exécution d'un programme. Elle cherche les modèles d'exécution récurrents à partir du comportement d'exécution d'un système [Bre 03]. C'est pourquoi, une notion de relations d'exécution entre les appels de méthodes a été introduite. Considérons l'exemple suivant d'une trace d'exécution de méthodes :

```
B() {
    C() {
        G()
        H()
        }
    }
A() { }
```

Quatre relations d'exécution ont été identifiées:

- *Outside-before* (ou *before*), par exemple : B est appelée avant A.
- *Outside-after* (ou *after*), par exemple : A est appelée après B.
- *Inside-first*, par exemple : G est le premier appel dans C.
- *Inside-last*, par exemple : H est le dernier appel dans C.

En utilisant ces relations d'exécution, l'algorithme d'aspect mining identifie les aspects candidats en se basant sur les modèles récurrents d'appels de méthodes. Si la relation d'exécution se produit uniformément plus d'une fois, par exemple chaque invocation de la méthode B est suivie par une invocation de la méthode A, elle est considérée comme un aspect candidat. Pour s'assurer que l'aspect candidat est suffisamment entrecoupant, il y a une condition supplémentaire exigeant que les rapports périodiques doivent paraître dans différents contextes d'appel. *DynAMiT* [Bre 04] est un outil implémentant cette technique.

Un ensemble de méthodes qui n'appartiennent pas à l'une des quatre relations d'exécution (*Outside-before, Outside-after, Inside-first* ou *Inside-last*) peut contenir des méthodes dont l'appel est dispersé avec fan-in élevée, et ces dernières ne sont pas

détectées comme aspects candidats. Par conséquent, cette technique ne détecte pas le symptôme de dispersion.

3.5.5 Détection de clones

Il a été démontré dans la littérature que les préoccupations transversales se manifestent habituellement sous forme d'un code source dupliqué [Bru 05]. Étant donné que ces préoccupations ne peuvent pas être bien modularisées et encapsulées dans des entités séparées, les développeurs sont souvent obligés d'écrire le même code, et encore en utilisant la pratique copier/coller , et en adaptant légèrement le code aux besoins [Bru 05]. La préoccupation *logging* est un bon exemple, avec une implémentation dupliquée partout dans le système. Puis, on a vu apparaître plusieurs techniques de détection de la duplication du code pour identifier les préoccupations transversales, permettant ainsi d'appliquer une refactorisaion (*refactoring*) orientée aspect, afin de supprimer ce code dupliqué.

Les techniques de détection des clones utilisées dans la littérature varient selon la structure utilisée, tels que le texte et le graphe de dépendances. Ces techniques peuvent être classées comme suit [Bru 05]:

- Techniques à base de texte (par exemple [Duc 99] et [Joh 93]) : elles utilisent directement la représentation textuelle du code source (séquences de lignes de code), pour la recherche des sections du code identiques ou légèrement différentes (l'espace blanc et les commentaires sont ignorés).
- Techniques à base de jetons (par exemple [Bak 95] et [Kam 02]) : approches lexicales, commencent par transformer le code source en une séquence de lexicales "jetons" en utilisant l'analyse lexicale. La séquence est ensuite analysée pour les séquences dupliquées de jetons, et le code original correspondant est retourné comme clones.
- Techniques basées AST (par exemple [Bax 98]) : avant de rechercher les clones, un opérateur analyse le code source et construit une représentation abstraite d'arbre de

syntaxe (*Abstract Syntax Tree AST*) de ce code. Ensuite, l'AST est analysé afin de détecter les sous-arbres similaires.

- Techniques basées PDG (par exemple [Kom 01], [Kri 01] et [Bel 12]) : elles utilisent une représentation plus abstraite du code source, les graphes de dépendance du programme (*Program Dependence Graph PDG*). Ces graphes contiennent non seulement des informations syntaxiques, mais aussi sémantiques. Ensuite, les approches tentent de détecter les sous-graphes similaires, afin d'identifier les clones.
- Techniques basées métriques (par exemple [May 96]) : plusieurs métriques sont calculées afin d'évaluer certains attributs de fragments de code (par exemple fan-in).

Les auteurs dans [Bru 05] ont fait une analyse de l'efficacité de ces techniques de détection de clones. Ils ont utilisé des outils implémentant trois de ces techniques, sur une même étude de cas, et ils ont comparé ensuite leurs résultats (aspects candidats) avec ceux annotés par le développeur original de l'étude de cas. Les résultats obtenus par l'analyse ont montré que les outils ont vraiment identifié les préoccupations caractérisées par la duplication du code.

Parmi les outils implémentant la technique de détection de clones, on peut en rappeler deux : l'outil *ccdiml* contenu dans le projet *Bauhaus* [Bau 11] qui appartient à la technique basée AST, un outil implémentant la technique basée jeton et nommé *CCFinder* [Kam 02] et un autre outil basé sur la technique PDG appelé *PDG-DUP* [Kom 01].

Bien que l'utilisation d'une technique de détection de clones soit efficace pour détecter le code dupliqué, elle reste limitée, car utile seulement pour la détection des préoccupations transversales ayant des fragments dupliqués du code.

3.5.6 Transversalité statique et dynamique

Une autre approche qui exploite les appels entre les méthodes est proposée par ***Bernadi*** dans [Ber 11]. L'auteur a défini deux types de transversalité.

La transversalité statique dûe aux déclarations (dispersées et enchevêtrées) des méthodes et des types substituables générés grâce aux relations statiques (héritage (*inheritance*),

implémentation (*implementation*) ou *containment*) [Ber 09]. Une analyse statique du code pour la hiérarchie de type de système est effectuée, en tenant compte des relations d'héritage, d'implémentation et de *containement*. Avec cette transversalité statique, l'auteur définit les préoccupations du système.

La transversalité dynamique est introduite grâce aux invocations dispersées et enchevêtrées des méthodes. En effet, une invocation d'une méthode affecte le flux du contrôle d'exécution, en ajoutant le comportement de la méthode appelée à celle appelante. Par conséquent, lorsqu'une méthode d'une préoccupation identifiée est invoquée par plusieurs autres méthodes appartenant à différentes autres préoccupations, le comportement de la préoccupation invoquée est dynamiquement dispersé (à l'exécution) dans les préoccupations associées aux appelantes. De la même façon, lorsqu'une méthode fait appel à d'autres méthodes appartenant aux différentes préoccupations, les comportements de telles préoccupations sont enchevêtrés ensemble dans la méthode appelante.

Cette approche n'est pas complète, car les préoccupations transversales qui n'ont pas une structure statique (en termes de modules du système) ne sont pas prises en compte.

3.5.7 Arbre d'appels et traces

Les auteurs dans [Qu 07] proposent une approche d'aspect mining qui utilise l'arbre d'appels de méthodes, afin de dépasser les limites de l'approche dans [Bre 03] qui reste incomplète, car les traces du programme contiennent seulement un ensemble particulier des entrées du programme. Une analyse dynamique complète est impossible pour exécuter tous les chemins possibles, ce qui diminue les aspects candidats.

L'idée de base dans [Qu 07] est d'observer le code source et créer l'arbre d'appels de méthodes, dans le but d'obtenir les traces d'appels des méthodes du système logiciel. L'arbre d'appels de méthodes est un arbre binaire qui décrit les relations d'appels de méthodes. Il contient des nœuds de contrôle et des nœuds d'appels de méthode. Le nœud de contrôle marque le type d'expression (*sentence*) où la méthode est appelée, incluant le nœud de contrôle de séquence, nœud de contrôle if (*if control code*), le nœud de contrôle *if-else*, le nœud de contrôle *switch* et le nœud de contrôle de boucle. Le nœud d'appel de

méthode marque la méthode appelée. Les traces sont ensuite étudiées pour les relations d'appels de méthodes. Ces dernières peuvent être décrites par les relations *inside* et *outside* (comme dans [Bre 03]). Les relations d'appels récurrents de méthodes sont des préoccupations transversales potentielles, qui décrivent les fonctionnalités récurrentes dans le programme, et sont par conséquent des aspects possibles. Cette approche détecte seulement l'enchevêtrement.

3.5.8 Rétro-ingénierie d'UML

Les aspects candidats ont été identifiés au niveau des modèles UML [Omg 13], générés à partir du code source orienté objet.

a. Rétro-ingénierie des diagrammes de séquence et d'activité:

L'identification des aspects à partir des diagrammes UML générés du code source a été discutée dans [Su 06]. L'auteur propose d'intégrer les aspects dans les diagrammes de séquence et d'activités générés à partir du code source orienté objet.

Concernant le diagramme de séquence, les préoccupations qui traversent les séquences des messages sont transversales. Afin de détecter ces préoccupations transversales, les patrons récurrents d'exécution générés à partir des traces d'exécution du programme orienté objet seront considérés comme des aspects, car ils décrivent les fonctionnalités dupliquées dans le programme. Pour intégrer l'aspect dans le diagramme de séquence, l'aspect (patron récurrent) doit être remodularisé en utilisant la structure composite d'UML (collaboration), outre les stéréotypes *<<aspect>>*, *<<advice>>*, *<<call>>* et *<<crosscut>>*.

Les préoccupations transversales du diagramme d'activités sont celles qui traversent multiples procédures ou threads [Su 06]. Les préoccupations transversales les plus importantes peuvent être des opérations de synchronisation ou d'exclusion mutuelle. Ensuite, grâce à la fusion et l'ajout des nœuds dans le diagramme d'activités, ce dernier sera orienté aspect.

Cependant, le seul symptôme de la transversalité détecté est l'enchevêtrement.

b. Rétro-ingénierie des diagrammes de classes et de séquence:

Une autre approche qui identifie les aspects, à partir des diagrammes UML de classes et de séquence générés du code source, est proposée dans [Dah 12]. Une approche hybride à trois passes a été retenue. Dans une première passe, l'analyse des concepts formels a été utilisée afin de regrouper les fonctionnalités dispersées. Dans une deuxième passe, l'ordre de transmissions de messages est analysé afin de détecter les fonctionnalités enchevêtrées. Ensuite, une fusion des résultats de ces deux analyses sera effectuée afin de filtrer l'ensemble des aspects candidats obtenus, et garder ceux jugés pertinents.

Les messages transmis entre les objets dans les diagrammes de séquence sont exploités. L'analyse FCA regroupe les méthodes du diagramme de classes selon les objets qui transmettent les messages dans les diagrammes de séquences (objets qui invoquent ces méthodes). Les aspects candidats sont toutes les méthodes invoquées (dans les concepts du treillis) qui ont le nombre de leurs objets (nombre d'objets appelants) supérieur à 2.

L'analyse de l'ordre d'appels vise à utiliser l'ordre chronologique de transmissions de messages dans les diagrammes de séquence, reflétant l'ordre d'exécution des tâches du système, afin de détecter les modèles récurrents d'appels de méthodes, et par conséquent les préoccupations enchevêtrées.

Concernant le filtrage des résultats, si un aspect candidat dispersé est déjà identifié comme enchevêtré, alors l'aspect dispersé sera éliminé, et le symptôme de dispersion est ajouté à l'aspect enchevêtré.

Cependant, la génération des diagrammes de classes et de séquence à partir du code source n'est pas vraiment facile et précise.

3.6 Discussions des approches

La table 3.1 récapitule et compare la plupart des approches citées dans les sections 3.5. Comme plusieurs approches de la table 3.1 ont été comparées dans [Kil 07] selon différents critères, nous avons défini dans cette table d'autres critères de classification:

1. **En se basant sur** : il existe des approches qui identifient les aspects en utilisant le partitionnement (**Part**) des artefacts du système logiciel, afin de détecter les préoccupations qui s'enchevêtrent et/ou se dispersent. D'autres approches se basent principalement sur le critère de répétition (**Répé**), pour identifier ces préoccupations transversales.
2. **Entrée** : c'est l'ensemble des artefacts nécessaires pour l'application de la technique.
3. **Sortie** : c'est l'ensemble des résultats fournis par la technique.
4. **Transversalité** : pour chaque technique, comment la transversalité est considérée.
5. **Niveau d'abstraction** : c'est le niveau dans lequel la technique d'identification d'aspects opère.
6. **Symptômes détectés** : ce sont les symptômes pouvant être détectés par la technique d'identification d'aspects : dispersion (**Disp**) et enchevêtrement (**Ench**).
7. **Intervention humaine** : l'approche nécessite t-elle une intervention du concepteur ou de l'utilisateur pour le choix des données d'entrée, filtrage des résultats, etc.

L'identification des aspects à partir de l'implémentation se base principalement sur l'analyse du code source. Ce dernier représente une information digne de confiance. En outre, il contient la dernière version des fonctionnalités du système. Or, les approches d'identification d'aspects à ce niveau (d'implémentation) souffrent de certaines limites, nous pouvons citer :

- le code source est difficile à comprendre ;
- les approches qui détectent les préoccupations transversales à partir des traces d'exécution sont incomplètes (une analyse dynamique complète est impossible pour exécuter tous les chemins possibles), ce qui diminue les aspects candidats. Le manque de précision nécessite un filtrage des aspects par l'intervention humaine [Qu 07] ;
- les outils d'aspect mining dépendent de la technologie d'implémentation (tel que le langage de programmation et le système d'exploitation) ;
- la subjectivité de l'interprétation des résultats [Men 08] ;

- un manque de détection simultané des deux symptômes de la transversalité: enchevêtrement et dispersion [Kel 07]. Dans [Men 08], ***Mens*** et al. affirment que presque toutes les techniques d'aspect mining du niveau code se concentrent exclusivement sur la détection de la dispersion, alors que l'enchevêtrement est, lui aussi, un indicateur de la transversalité ;
- la majorité des approches existantes d'identification d'aspects nécessitent l'intervention humaine [Kel 07] (comme il est montré dans la colonne « intervention humaine » de la table 3.1) ;
- les préoccupations transversales se manifestent dans des étapes précoces du cycle de développement logiciel, en raison de leur large influence dans les décompositions de logiciels. Le manque de soutien de cette identification, dans les phases antérieures, est un obstacle pour l'application des approches de la modélisation des aspects.

Tab. 3.1. Table récapitulative et comparative des approches d'identification d'aspects à l'implémentation

N° Section	Technique		En se basant sur:		Entrée	Sortie	Niveau d'abstraction	Symptômes détectés		Transversalité est :	Intervention humaine
			Part	Répé				Disp	Ench		
3.5.1	FCA	a	X		Objets * Attributs	Hiérarchie de concepts	Code source	X		Attribut partagé par plusieurs objets	Navigation des aspects obtenus en utilisant l'intégration IDE [Kel 07]
		b					traces d'exécution + cas d'utilisations	X	X		Sélection des cas d'utilisation
		c					Code source	X			Choix d'un seuil fan-in
3.5.2	Clustering	a	X		Groupes d'objets + Distances de similarité	Groupes d'objets similaires	Code source	X		Objet du même groupe appartient à d'autres objets	Navigation des aspects obtenus en utilisant l'intégration IDE [Kel 07]
		b					traces d'exécution + cas d'utilisations		X		Sélection des cas d'utilisation
		c					Code source + traces d'exécution + cas d'utilisations	X			Interprétation des chaines lexicales obtenues
3.5.3	Fan-in			X	Invocations de méthodes	Méthodes souvent invoquées	Code source	X		Méthode invoquée plusieurs fois par objets de différentes classes	Sélection des aspects candidats à partir d'une liste [Kel 07]
3.5.4	Modèle récurrent			X	Traces d'exécutions	Relations récurrentes d'exécution	Traces d'exécution		X	Relation d'exécution se produisant plusieurs fois	Inspection des patrons récurrents obtenus [Kel 07]
3.5.5	Clone			X	Code source/ PDG/ AST/..	Fragment dupliqué du code	Code source	X		Fragment dupliqué du code	Navigation et inspection manuelle des clones trouvés [Kel 07]

3.5.6	Transversalité statique et dynamique	X		Déclarations dispersées et enchevêtrées	Préoccupations	Code source	X	X	•Méthode invoquée par plusieurs autres méthodes appartenant à différentes préoccupations. •Méthode qui invoque d'autres méthodes appartenant à différentes préoccupations.	Inspection des relations hiérarchiques, afin d'identifier les préoccupations
3.5.7	Arbre d'appels et traces		X	Arbre d'appels de méthodes	Trace d'appels de méthodes	Code source + traces d'exécution		X	relations d'appels récurrents de méthodes	Inspection des patrons récurrents obtenus
3.5.8	Rétro-ingénierie d'UML		X	Diagrammes de séquence et d'activité	Traces d'exécution	Traces d'exécution		X	patrons récurrents d'exécution	Inspection des patrons récurrents obtenus

3.7 Conclusion

Dans ce chapitre, nous avons mis l'accent sur l'intérêt du domaine de l'aspect mining pour la rétro-ingénierie du logiciel, et nous avons ensuite présenté les approches et les techniques les plus connues, qui identifient les aspects au niveau implémentation. Nous avons également extrait les points forts et les points faibles de ces approches.

La complexité d'identification des préoccupations transversales au niveau implémentation, nécessite une prise en charge précoce de la gestion de la transversalité. Dans le prochain chapitre, nous présentons les approches d'identification des préoccupations transversales durant le cycle de développement logiciel (avant la phase de l'implémentation).

CHAPITRE 4

IDENTIFICATION D'ASPECTS DURANT LES PHASES DU DÉVELOPPEMENT DE LOGICIEL: ÉTUDE COMPARATIVE DES APPROCHES

« Aspects are identified and captured mainly in code. But aspects are evident earlier in the life cycle, such as during requirements gathering and architecture development. Identifying these early aspects ensures that you can appropriately capture aspects related to the problem domain (as opposed to merely the implementation). Additionally, it offers opportunities for early recognition and negotiation of trade-offs and allows forward and backward aspect traceability. This makes requirements, architecture, and implementation more seamless and lets you apply aspects more systematically.»

(Baniassad et al. 2006) [Ban 06]

4.1 Introduction

Les intervenants (*stakeholders*) peuvent décrire leur système en termes d'exigences (*requirements*). Une exigence définit une propriété ou une capacité qui doit être présentée par un système, afin de résoudre le problème pour lequel il a été conçu [Chi 05]. Les exigences sont généralement classées en deux catégories: les exigences fonctionnelles et non fonctionnelles. Bien que les exigences fonctionnelles soient liées à des fonctions que le système doit fournir, les exigences non fonctionnelles décrivent la qualité ou les

contraintes du système. En ce sens, les contraintes limitent l'ensemble des solutions possibles du système. Une contrainte peut être de nature technique (par exemple, la bande passante dans une application de réseau) ou liées au domaine du problème (par exemple, la législation ou les normes) [Chi 05].

Chaque exigence peut concerner une ou plusieurs préoccupations. Celles-ci s'enchevêtrent et se dispersent sur les exigences, impliquant l'émergence des préoccupations transversales.

Afin de bénéficier d'un développement orienté aspect, il est nécessaire de gérer ces préoccupations transversales durant les phases du cycle de développement du logiciel, au niveau exigences, architecture ou conception. A cet effet, plusieurs approches d'identification des préoccupations transversales, avant la phase d'implémentation du logiciel, ont été proposées, et un nouveau domaine appelé « *Early aspect mining* » [Mor 11] a vu le jour.

Le terme « *early aspect* » (aspect précoce) est défini comme une préoccupation transversale au début des phases du développement du logiciel [Ban 06, Mor 11]. Les auteurs [Ban 06] ont défini les aspects précoces comme « des préoccupations qui entrecoupent la décomposition dominante d'un artefact, ou les modules de base issus à partir du critère de la séparation dominante des préoccupations ».

En outre, afin d'améliorer la future maintenance, réutilisation et évolution des systèmes logiciels qui ne sont pas encore implémentés, on doit faciliter le processus de leur compréhension. Pour comprendre un logiciel complexe, on doit se baser sur une information digne de confiance. Celle-ci est représentée principalement par son code source. La compréhension de ce dernier est facilitée par l'amélioration de sa structure. Une amélioration possible consiste en la séparation des préoccupations, et le paradigme orienté aspect est apparu pour supporter la structure améliorée des systèmes logiciels. Par conséquent, early aspect mining permet de gérer la transversalité tôt dans le cycle du développement du logiciel, afin d'effectuer une implémentation orientée aspect.

4.2 Définitions de la transversalité

Les techniques d'identification d'aspects, avant la phase d'implémentation, sont basées principalement sur de nouvelles définitions de la transversalité. Nous en présenterons, dans ce qui suit, les plus connues.

4.2.1 Définition basée sur les exigences

La transversalité se produit lorsqu'une exigence est partiellement implémentée dans plus d'une classe [Fox 05]. Dans la figure 4.1, les exigences $r2$ et $r3$ sont des aspects, car elles sont implémentées dans plus d'une classe.

Cette définition, formulée par ***Fox*** [Fox 05], est basée sur une classification des aspects dans deux catégories différentes: systémique (*Systemic*) et granularité fine (*Fine Granular FG*). La catégorie systémique se réfère aux aspects qui influencent les décisions de la mise en forme et de la modification du comportement de l'architecture (exemple de la qualité du service), et peuvent être résolus en utilisant les mécanismes de composition, comme Hyper/J [Oss 01] ou les filtres de composition [Ber 01]. *FG* dénote les aspects qui peuvent être implémentés au niveau des méthodes dans les classes, par exemple la sécurité, la synchronisation, logging, et peuvent être dans certains cas résolus en utilisant par exemple les patrons de conception (*Design Patterns*) [Eri 95] et AspectJ [Kic 01a].

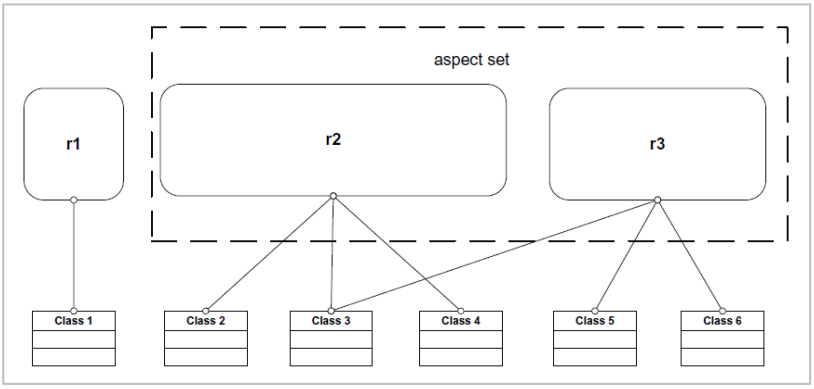


Fig. 4.1. Exigences et aspects en relation avec les classes [Fox 05]

Comme on peut le constater à partir de la définition de la transversalité de ***Fox***, un aspect est défini ici en termes de dispersion, et cette dernière est considérée par l'auteur comme une condition nécessaire et suffisante pour détecter la transversalité.

4.2.2 Définition basée sur le domaine source et cible

La transversalité se produit, dans la situation où deux domaines (source et cible) sont liés grâce à des dépendances de traçabilité, lorsqu'un élément source est dispersé sur différents éléments cibles [Ead 08a] et au moins un de ces éléments cibles est enchevêtré [Con 10].

D'après ***Conejero*** [Con 10], la transversalité ne peut être définie qu'en termes de «quelque chose» en respectant «autre chose». En conséquence, et à partir d'un point de vue mathématique, cela signifie qu'on a deux domaines liés l'un à l'autre. Les termes source et cible sont utilisés pour désigner ces deux domaines, et la relation de traçabilité est le mappage entre ces domaines (figure 4.2).

Les domaines source et cible peuvent représenter deux différents domaines, niveaux ou phases d'un processus de développement du logiciel, et les définitions d'enchevêtrement, dispersion et transversalité sont considérées comme des cas particuliers de la correspondance entre ces deux domaines.

Comme on peut le voir dans la figure 4.2, il y'a une fonction f' multi-valeurs à partir des éléments sources ($s \in Source$, $Source$ est le domaine source) aux éléments cibles ($t \in Cible$, $Cible$ est le domaine cible qui est un ensemble de préoccupations ou éléments du programme, qui ne sont pas déjà dans $Source$). $f' : Source \rightarrow Cible$ tel que si $f'(s) = t$ alors il existe une relation de trace entre$s$ s et t (où s et t sont les éléments source et cible, respectivement).

De manière analogue, on peut définir une autre fonction multi-valeurs g' qui peut être considérée comme l'inverse de f'. $g' : Cible \rightarrow Source$, tel que si $g'(t) = s$ alors il existe une relation de trace entre s et t.

Les fonctions f' et g' peuvent également être représentées comme des fonctions à valeur unique, étant donné que les fonctions sont l'ensemble des parties non vides de la cible et l'ensemble des parties non vides de la source, respectivement :

Soit $f : Source \rightarrow \wp(Cible)$ et $g : Cible \rightarrow \wp(Source)$ ces nouvelles fonctions définies par:

$\forall\, s \in Source,\ f(s) = \{t \in Cible : f'(s) = t\}.$

$\forall\, t \in cible,\ g(t) = \{s \in Source : g'(t) = s\}.$

Les concepts de dispersion, enchevêtrement et transversalité sont définis comme des cas particuliers de ces fonctions. La dispersion se produit lorsque, dans une mise en correspondance entre la source et la cible, un élément source est lié à plusieurs éléments cibles. On dit qu'un élément s est dispersé si $card\,(f(s)) > 1$. L'enchevêtrement se produit quand un élément cible est lié à plusieurs éléments sources. On dit qu'un élément cible est enchevêtré si $card(g(t)) > 1$. ($card$ est la cardinalité).

La transversalité se produit lorsqu'un élément source est dispersé sur différents éléments cibles, et au moins un de ces éléments cibles est enchevêtré [Con 10]. Soient $s1, s2 \in S$, $s1 \neq s2$. On dit que $s1$ entrecoupe $s2$ si $card\,(f(s1)) > 1$ et $\exists\, t \in f(s1) : s2 \in g(t)$.

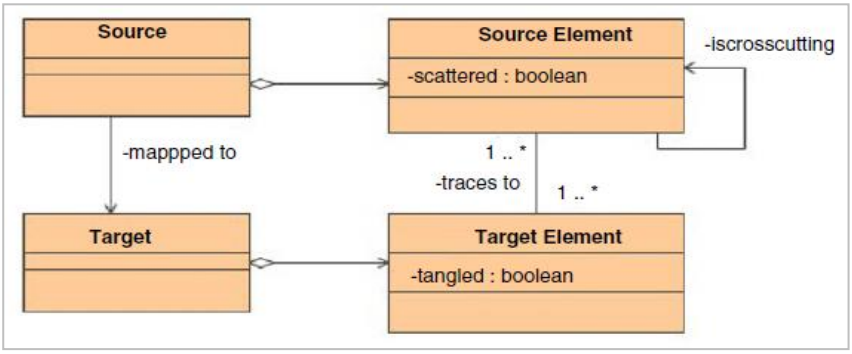


Fig. 4.2. Modèle de transversalité [Con 10]

Dans [Ead 08a], ***Eaddy*** se base sur l'identification des endroits des préoccupations. Il identifie les relations (R) entre deux domaines (figure 4.3) : un domaine de préoccupations (domaine source) et un domaine cible. Une préoccupation transversale est une préoccupation dispersée (sans tenir compte de l'enchevêtrement) dans les éléments cibles. Il considère la dispersion comme une condition nécessaire et suffisante pour détecter la transversalité. La relation entre les préoccupations et les artefacts cibles est utilisée pour trouver le code source lié à un problème particulier.

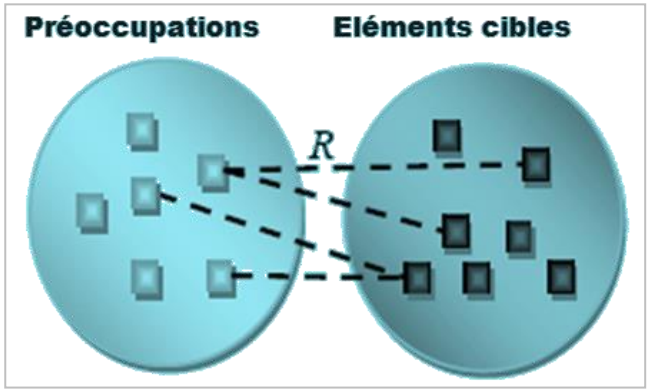


Fig. 4.3. Modèle de préoccupation [Ead 08a]

Dans [Ead 08b], les auteurs ont utilisé quelques métriques afin de valider la modularité du logiciel. La métrique degré de dispersion (*Degree Of Scattering DOS)* fournit une information concernant la manière dont le code d'une préoccupation est distribué entre les éléments. DOS est définie comme la variance statistique de la métrique *concentration* d'une préoccupation sur tous les éléments du programme. La métrique *concentration* [Won 00] est définie comme le quotient des lignes sources dans l'élément t liées à la préoccupation s, par les lignes sources liées à la préoccupation s.

DOS est proportionnelle au nombre d'éléments liés à la préoccupation, et inversement proportionnelle à la concentration. Autrement dit, une préoccupation moins concentrée est une préoccupation plus dispersée.

Dans [Con 09], les auteurs ont identifié trois métriques, selon la technique de traçabilité [Con 10] :

- La métrique *NScattering* d'un élément source s est définie comme le nombre des éléments cibles qui s'adressent à cet élément source.
- La métrique *NTangling* d'un élément cible t est le nombre des éléments sources qui s'adressent à cet élément cible.
- La métrique *NCrosscut* d'un élément source s est le nombre des éléments sources entrecoupés par l'élément source s.

4.3 Synthèse sur les définitions de la transversalité

La table 4.1 récapitule les définitions de la transversalité présentées dans les sections 3.2 et 4.2. Nous constatons que les définitions n'utilisent pas les mêmes artefacts. Par exemple, dans [Mas 03], les artefacts de programmation sont utilisés, tandis que dans [Fox 05] l'utilisation porte sur les exigences. En outre, il y a des définitions qui se ressemblent, comme celles dans [Mas 03] et [Mez 03]. D'autres définitions sont des cas particuliers des autres, par exemple les définitions proposées dans [Mas 03], [Mez 03] et [Ead 08a] sont un cas particulier de celle fournie dans [Con 10].

D'autre part, certaines définitions proposées ne prennent pas en compte simultanément la dispersion et l'enchevêtrement, comme par exemple le cas de [Fox 05], la transversalité est définie en termes de dispersion, en négligeant l'enchevêtrement. D'autres considèrent l'enchevêtrement et la dispersion comme deux conditions nécessaires à la fois pour définir la transversalité (exemple [Con 10]), or l'enchevêtrement ou la dispersion est une condition nécessaire et suffisante pour détecter la transversalité, et l'existence d'une de ces conditions pose un problème au niveau de la compréhension, la maintenance, la réutilisation et l'évolution des systèmes logiciels.

La définition de la transversalité reste vague, imprécise et n'est pas générale, du fait que chaque auteur considère la transversalité de son point de vue, et la définition de la transversalité est fortement liée aux artefacts utilisés (dans la phase du cycle de développement logiciel).

Tab. 4.1. Table récapitulative de définitions de la transversalité

Section	**Travail**	**La transversalité se produit lorsque :**	**Niveau d'abstraction**	**Symptômes**
3.2.1	[Mas 03]	Les projections de deux modules dans un domaine résultat se croisent, et aucune projection n'est sous ensemble de l'autre	Code source	Enchevêtrement

3.2.2	[Mez 03]	Les décompositions des éléments du code source selon des différents critères se croisent	Code source	Enchevêtrement
4.2.1	[Fox 05]	L'exigence est partiellement implémentée dans plus d'une classe.	Exigences	Dispersion
4.2.2	[Ead 08a]	Un élément source est dispersé sur différents éléments cibles	∀ niveau d'abstraction (exigences, conception ou implémentation)	Dispersion
	[Con 10]	Un élément source est dispersé sur différents éléments cibles et au moins un de ces éléments cibles est enchevêtré	∀ niveau d'abstraction	Dispersion & Enchevêtrement

4.4 Approches d'identification d'aspects à partir des exigences

Une exigence définit une propriété ou une capacité qui doit être présentée par un système, afin de lui permettre de résoudre le problème pour lequel il a été conçu. Plusieurs approches d'ingénierie des exigences ont été proposées. Les plus connues dans la littérature sont:

- Les points de vue (*viewpoints*) [Fin 96] ;
- Les buts (*goals*) [Myl 92] ;
- Les cas d'utilisation (*uses cases*) [Jac 92].

Ces approches initiales ne supportent pas l'identification et l'intégration des aspects. Pour cette raison, des extensions de ces approches et des nouvelles approches ont apparu par la suite, supportant la gestion de la transversalité au niveau exigences.

4.4.1 Approches orientées points de vue

Selon ***Sommerville*** [Som 97], les approches d'ingénierie des exigences orientées points de vue visent l'amélioration du processus d'ingénierie des exigences, ainsi que l'organisation et la présentation de la spécification elle-même. L'ingénieur d'exigences

recueille et analyse les exigences d'un système à partir de différentes perspectives ou points de vue. Ces dernières sont les entités qui peuvent être utilisées pour structurer la spécification des exigences. Elles organisent les exigences du système à partir de différents types d'utilisateurs et d'intervenants.

Les approches orientées points de vue considèrent les informations concernant le problème à partir des vues de différents utilisateurs du système, qui peuvent avoir différentes perspectives, selon les différents responsabilités et rôles.

4.4.1.1 Approches non orientées aspects

Les approches orientées points de vue non orientées aspect sont : *PREview* [Som 96, Som 97, Saw 96] et *VIM* (*Viewpoints and Inconsistency Management*) [Nus 94, Nus 03].

- PREView (*Process and Requirements Engineering VIEWpoints*) utilise les préoccupations comme des conducteurs dans la découverte des exigences, et ces préoccupations reflètent les caractéristiques critiques non fonctionnelles du système. Lors de l'utilisation des points de vue pour la découverte des exigences réelles, les préoccupations (qui sont identifiées tout au début du processus d'ingénierie des exigences et sont décomposées en questions, contraintes ou exigences) doivent être traitées par les points de vue. Ainsi, les préoccupations peuvent entrecouper tous les points de vue, et les questions associées aux préoccupations doivent être reliées à tous ces points de vue. L'approche PREView est structurée autour de la reconnaissance de l'importance de l'impact des préoccupations transversales non fonctionnelles. Ces préoccupations sont identifiées au moyen des interviews avec les intervenants, et sont modularisées en modèles de préoccupations. L'impact d'une préoccupation sur d'autres préoccupations, cependant, n'est pas modularisé. L'identification et le traitement des préoccupations non fonctionnelles sont appropriés seulement quand un petit ensemble stable d'exigences non-fonctionnelles bien définies est impliqué. En outre, bien que PREview suggère que la fonctionnalité peut être une préoccupation, de la même façon que les préoccupations non fonctionnelles, il n'y a pas de démonstration de la façon dont ces préoccupations sont identifiées et traitées. Par conséquent, l'approche PREView ne prend pas en charge

l'identification et le traitement de la transversalité pour les préoccupations fonctionnelles.

- VIM permet la spécification d'un système à partir de plusieurs points de vue, sans avoir besoin de prédéfinir une notation pour toutes les spécifications. Différents points de vue (hétérogènes) peuvent définir leurs propres modèles pour la méthode de notation et de traitement. Un modèle peut ensuite être instancié pour de nombreux points de vue. Cette approche ne permet pas d'identifier les préoccupations transversales séparément: toutes les préoccupations sont traitées comme une partie d'un point de vue. Bien que les points de vue puissent se chevaucher et s'entrecouper les uns les autres, cela est traité comme un problème de résolution d'incohérence, plutôt qu'un problème de la séparation des préoccupations.

Les deux approches PREview et VIM semblent traiter les préoccupations transversales. Cependant, PREview est limitée au petit nombre de préoccupations non fonctionnelles prédéfinies, sans pouvoir supporter d'autres, tandis que VIM traite toutes les préoccupations comme parties de points de vue, sans considérer leurs problèmes de modularisation individuelle.

4.4.1.2 Approche orientée aspects (*Arcade*)

Arcade [Ras 03] a été l'une des premières approches introduisant le terme « *early aspect* ». Elle a également été l'une des premières approches relatives à l'identification, la séparation et la composition des préoccupations transversales au niveau des exigences.

Arcade est une extension de l'approche PREview, en proposant des moyens pour séparer les aspects candidats à partir des points de vue. Le processus d'Arcade commence d'abord par l'identification des préoccupations et des exigences des intervenants. Une fois que les préoccupations et les exigences sont identifiées, elles sont représentées dans une matrice où les préoccupations qui contraignent les exigences sont marquées. Les préoccupations qui sont liées à plus d'une exigence des intervenants sont considérées comme des aspects candidats.

L'approche Arcade ne traite que l'identification et la séparation des préoccupations transversales non-fonctionnelles. Or, les préoccupations fonctionnelles peuvent être

également des préoccupations transversales et candidates à être modularisées par les techniques orientées aspect. Ainsi, il n'est pas claire (tâche non automatique) comment la matrice utilisée pour lier les préoccupations et les exigences des intervenants (points de vues) est obtenue.

4.4.2 Approches orientée buts

Les buts (*goals*) ont longtemps été reconnus comme des éléments essentiels dans l'ingénierie des exigences. Un but est défini comme un objectif que le système considéré doit atteindre [Van 01]. Les buts doivent être spécifiés précisément pour supporter l'élaboration, la vérification/validation, la gestion des conflits, la négociation, l'explication et l'évolution des exigences. L'ensemble des exigences est complet s'il satisfait tous les buts considérés. Un but peut couvrir une exigence fonctionnelle ou non fonctionnelle.

Les buts fournissent le rationnel pour le changement des exigences, autrement dit, une base pour leur fondement même. En outre, le raffinement des buts fournit un mécanisme naturel pour structurer les documents des exigences complexes, impliquant une meilleure lisibilité, et fournissant des liens de traçabilité à partir des objectifs stratégiques de haut niveau vers les exigences de bas niveau.

4.4.2.1 Approches non orientées aspects

Le framework *NFR* (*Non-Functional Requirements Framework*) proposé par ***Chung*** [Chu 00], *KAOS* (*Knowledge Acquisition in Automated Specification*) proposé par ***Dardenne*** [Dar 93] et le framework *i** proposé par ***Yu*** [Yu 97], sont des approches basées buts non orientées aspect.

- Le framework NFR fournit des structures de connaissances sur les buts non-fonctionnels basés sur un framework qualitatif, pour l'intégration des buts et leurs raffinements dans les modèles d'exigences. Une fois les buts identifiés, ils seront raffinés, afin de construire un graphe d'interdépendance (père/fils). Les sous buts feuilles doivent être satisfaits ensemble. Le NFR met l'accent sur la clarification au

sens des exigences non-fonctionnelles, et fournit des alternatives pour les satisfaire au plus haut niveau possible.

- KAOS utilise les buts comme le concept central dans l'acquisition des exigences, et comme le principal concept de guide dans le développement des spécifications formelles, qui intègrent les buts et leurs raffinements dans les modèles d'exigences. En d'autres termes, KAOS structure les exigences dans un modèle, pour le raffiner en utilisant un langage formel.
- Le framework i* est une approche de connaissance orientée agent pour la modélisation explicite et l'analyse des multiples relations entre les acteurs, en introduisant l'analyse sociale dans l'analyse des exigences, et en facilitant le dialogue avec les intervenants pendant l'extraction des exigences.

La représentation des exigences par utilisation des buts reste limitée. Les approches basées buts ne précisent pas exactement comment les exigences non fonctionnelles sont identifiées, mais seulement suggèrent qu'elles devraient être obtenues en rassemblant les connaissances sur le domaine pour lequel un système sera construit.

4.4.2.2 Approche orientée aspects (AOV-graph)

Dans [Yu 04], ***Yu*** et al. ont présenté l'une des premières approches pour faire face aux préoccupations transversales dans les modèles basés buts, en particulier dans les graphes V (*V-graph*). V-graph est un type spécifique du modèle but où les buts qui représentent les fonctionnalités, et les buts non fonctionnels qui représentent les préoccupations non fonctionnelles, sont représentés par un graphe ayant la forme globale de la lettre « V ».

Le V-graph orienté aspect (AOV-graph) est une extension des modèles V-graph pour éviter l'enchevêtrement et la dispersion des préoccupations transversales dans les modèles d'exigences [Fer 05]. Cette approche utilise des modèles de but et les concepts définis dans des langages orientés aspects, pour assurer la séparation, la composition et la visualisation des préoccupations transversales, afin de faciliter leur modélisation. Les buts de l'AOV-graph sont utilisés pour modéliser les ensembles des exigences séparément et offrir un moyen pour modéliser les relations entre elles.

Les buts fonctionnels et les buts non fonctionnels sont récursivement décomposés jusqu'à ce qu'ils puissent être réduits à une tâche spécifique. Les préoccupations transversales peuvent alors être identifiées comme les tâches qui contribuent à plusieurs buts fonctionnels et non fonctionnels. Cependant, la plupart des procédures utilisées dans le processus d'identification d'aspects doivent être effectuées manuellement. Cela n'est pas faisable dans les systèmes complexes [Yu 04]. En outre, l'identification des buts fonctionnels et non fonctionnels, ainsi que les relations de contributions, n'est pas claire.

4.4.3 Approches orientées cas d'utilisation

L'approche des cas d'utilisation (*use cases*), proposée initialement par ***Jacobson*** [Jac 92], a émergé comme le moyen le plus populaire pour présenter les exigences des intervenants ou des acteurs du système, et capturer les exigences dans l'industrie. Un acteur est une entité externe, une personne, un autre système ou tout ce qui interagit avec le système en cours d'étude. Un cas d'utilisation définit un ensemble d'interactions entre un acteur et le système, afin d'atteindre un but. Chaque cas utilisation décrit comment un acteur réalise un but ou une tâche.

Bien que les cas d'utilisation soient faciles à comprendre, et sont généralement considérés comme une excellente technique pour capturer les exigences fonctionnelles d'un système, leur utilisation dans l'ingénierie des exigences reste limitée. Les cas d'utilisation ne sont pas bien adaptés pour capturer les exigences non fonctionnelles. Pour cette raison, plusieurs approches visent à identifier les préoccupations transversales à partir des cas d'utilisation.

Au niveau des diagrammes des cas d'utilisation, les préoccupations transversales sont des exigences non fonctionnelles (contraintes que le système doit satisfaire) qui entrecoupent et affectent plus d'un cas d'utilisation [Ara 02]. Un cas d'utilisation transversal peut avoir une relation *<<include>>*, *<<extend>>* ou *<<constrain>>* avec plus d'un cas d'utilisation [Ara 03, Mor 02, Bri 02, Kas 06].

L'approche *AOSD/UC* (*Aspect Oriented Software Development with Use Cases*) est une méthode du développement de logiciel orientée cas d'utilisation [Ara 02]. Les diagrammes des cas d'utilisation représentent les préoccupations séparément. Cette

méthode fournit un processus par lequel la séparation de préoccupations aux diagrammes de cas d'utilisation est maintenue à travers le cycle de vie du développement de logiciel. À un haut niveau de conception, les aspects sont définis comme des cas d'utilisation qui s'étendent aux autre cas d'utilisation. Dans la conception détaillée, les cas d'utilisation sont spécifiés sous forme de paquets. Dans ces paquets, la conception de cas d'utilisation est représentée. Les cas d'utilisation transversaux contiennent les aspects. Un aspect est un classificateur qui est identifié par un stéréotype <<*aspect*>>.

Dans [Chi 05], les auteurs considèrent les cas d'utilisation comme des préoccupations transversales, car la réalisation de chaque cas d'utilisation affecte plusieurs classes.

Les aspects peuvent être, après leur identification, modularisés comme des types spécifiques de cas d'utilisation, en utilisant des stéréotypes UML, tels que <<*wrappedBy*>>, <<*overlapedBy*>>, <<*overridedBy*>>, <<*include*>>, <<*extend*>> et <<*constrain*>>.

Les approches d'identification d'aspects au niveau des diagrammes de cas d'utilisation se basent sur l'analyse syntaxique, et opèrent de façon semi-automatique. En outre, l'identification de la transversalité en se basant sur les cas d'utilisation, manque de précision du fait que le niveau de granularité est grand.

4.4.4 Approches basées composants (AOREC)

Dans [Gru 99], ***Grundy*** a présenté une approche qui introduit les techniques orientées aspects pour le développement du logiciel basé composants. L'approche est appelée *AOREC* (*Aspect Oriented Requirements Engineering for Component Based Systems*), et elle est une extension du développement de logiciel basé composants.

Lors du développement d'une application, les exigences du système sont analysées afin d'identifier les composants. Le développeur a besoin d'identifier tous les composants, et utilise le terme « aspect » pour décrire les fonctionnalités qui entrecoupent les différents composants.

Un aspect dans AOREC est une caractéristique du système pour laquelle les composants fournissent ou nécessitent des services. Ces aspects sont des contraintes non

fonctionnelles, et incluent les interfaces de l'utilisateur, les services du travail collaboratif, la gestion de la persistance et de la distribution, et les services de sécurité.

L'approche AOREC permet d'isoler les aspects afin d'avoir des composants réutilisables. Cependant, l'identification d'aspects qui entrecoupent les composants n'a pas été clairement définie. En outre, AOREC ne fournit aucun support général pour l'identification des préoccupations transversales, et les aspects sont identifiés cas par cas, par l'ingénieur des exigences.

4.4.5 Séparation multidimensionnelle des préoccupations (Cosmos)

La séparation multidimensionnelle des préoccupations (*Multi-Dimensional Separation of Concerns MDSoC*) [Tar 99] est basée sur le principe selon lequel tous les artefacts du développement de logiciel sont multiples, constitués de préoccupations qui se chevauchent, et dont le développement du logiciel bénéficiera si les systèmes logiciels peuvent être, d'une manière flexible, décomposés et composés selon les différentes combinaisons et les organisations possibles des préoccupations [Sut 03]. Les auteurs utilisent le terme « séparation multidimensionnelle des préoccupations » pour désigner une séparation, modularisation et intégration souples et progressives des artefacts du logiciel, en se basant sur un certain nombre de préoccupations. Ces mécanismes permettent une séparation nette des préoccupations multiples, qui potentiellement se chevauchent en même temps, en supportant la remodularisation (à la demande) afin d'encapsuler les nouvelles préoccupations à tout moment [Bri 08].

En se basant sur le principe de la séparation multidimensionnelle de préoccupations [Tar 99], plusieurs approches ont été proposées [Yos 99, Tar 99, Mor 05]. Ces approches ne font pas une distinction forte entre les préoccupations transversales et non transversales dans un système, mais plutôt traitent toutes les préoccupations de façon symétrique. La notion de la multi-dimension supporte ainsi le développement orienté aspect du logiciel, car on peut éventuellement choisir un ensemble arbitraire de préoccupations comme une base sur laquelle on observe l'influence transversale sur un autre ensemble de préoccupations [Yos 99].

Cosmos est un schéma d'usage général pour la modélisation de l'espace multidimensionnel de préoccupations [Sut 02, Hyp 12]. Un espace de préoccupations est une représentation organisée des préoccupations, leurs relations et les prédicats. Les relations et les prédicats sont également classés comme des préoccupations, afin de garantir l'exhaustivité et la cohérence avec une perspective multidimensionnelle.

Les préoccupations sont classées comme logiques (représentant les préoccupations vues conceptuellement, tels que les propriétés et les problèmes), et physiques (unités du logiciel, telles que les classes et les instances). Les relations sont utilisées pour représenter la manière dans laquelle les préoccupations interagissent entre elles. Les prédicats sont utilisés pour établir les conditions d'intégrité sur les relations, par exemple : une préoccupation ne peut pas être à la fois une classe et une instance.

Ainsi, une seule préoccupation considérée dans COSMOS peut à la fois, concerner plusieurs phases du cycle de vie, être liée à plusieurs types d'artefacts et affecter différentes phases et artefacts de différentes manières. Une préoccupation peut changer au fil du temps avec le changement correspondant dans les artefacts de logiciels liés à cette préoccupation. De la même façon que les préoccupations, les aspects dans COSMOS sont considérés comme des représentations de préoccupations qui se produisent également pendant tout le cycle de vie, et peuvent être également liés à des artefacts différents (en fonction de la phase de développement).

Le processus de Cosmos commence par l'analyse des documents des exigences, afin d'identifier les préoccupations et leur catégorie (physique, logique). Il étudie attentivement les documents disponibles et les artefacts d'intérêt afin d'identifier les préoccupations exprimées explicitement, ainsi que les préoccupations impliquées par les documents et les artefacts. Il est nécessaire de sélectionner seulement les préoccupations qui sont raisonnablement pertinentes pour le système logiciel, car le nombre de préoccupations sera très élevé, même pour les petits systèmes.

Cosmos permet une décomposition du système en préoccupations identifiées et une composition ultérieure, en utilisant différents critères. Comme les multiples compositions des préoccupations sont possibles, différentes versions du système peuvent être construites. En outre, en reliant les préoccupations avec les artefacts logiciels qui les

réalisent (dans n'importe quel niveau d'abstraction ou de développement), la traçabilité est réellement améliorée et l'impact de changement peut être facilement estimé.

Les préoccupations partagées (par différentes exigences) doivent être considérées comme des aspects candidats. De même, les préoccupations réutilisées dans différentes parties peuvent être considérées comme des préoccupations transversales. Cependant, l'identification des préoccupations (transversales) n'est pas toujours simple.

4.4.6 Ea-Miner

EA-Miner [Sam 05] est un outil semi-automatique d'identification des préoccupations et des aspects précoces au niveau des exigences, basé sur les techniques du traitement du langage naturel. Il est implémenté comme un plug-in d'Eclipse. Il prend en entrée les documents des exigences du système, et il permet l'identification et la structuration des abstractions des différents modèles des exigences, tels que les cas d'utilisations ou les points de vue.

EA-Miner utilise l'outil WMATRIX [Saw 02] pour faire un prétraitement sur les documents des exigences. WMATRIX implémente différentes analyses de traitement du langage naturel, pour voir quels sont les mots les plus pertinents dans le texte, obtenir la fonction grammaticale de chaque mot (nom, verbe, objectif…) et son sens, et regrouper les mots dans des catégories selon leurs relations de sens (utilise un large dictionnaire de plus de 73984 mots).

Les étapes du traitement de l'EA-Miner sont les suivantes :

- Identifier les principales exigences à partir des interviews ou des documents des exigences (chaque phrase est considérée comme exigence) ;
- Identifier les concepts du modèle des exigences automatiquement grâce au traitement du langage naturel, ainsi que les relations entre ces concepts. Ces derniers peuvent être ajoutés, supprimés ou édités par le développeur. Les exigences associées aux différents concepts sont utilisées pour identifier les relations entre les exigences partagées par plus d'un concept ;

- Identifier les aspects précoces en utilisant la sémantique des mots obtenue par WMATRIX. L'algorithme utilisé consiste en la comparaison des valeurs de la sémantique associée à un mot, avec un catalogue des préoccupations non fonctionnelles où les mots similaires sont définis avec leurs valeurs sémantiques associées. Si une correspondance est trouvée, le mot est identifié comme un aspect candidat, et la phrase de l'exigence où le mot apparait est associée à l'aspect précoce. Puisque les préoccupations fonctionnelles pourraient être également considérées comme des préoccupations transversales candidates, l'outil utilise également une analyse fan-in [Mar 04] pour obtenir les mots les plus fréquemment utilisés, et de les classer comme des aspects candidats ;
- Identifier les relations transversales en établissant les relations entre les concepts (points de vue par exemple) et les aspects précoces, en se basant sur les exigences partagées par ces concepts ;
- Structurer les documents des exigences grâce aux concepts identifiés ;
- Valider le modèle des exigences obtenu et résoudre les conflits, par l'utilisation des autres outils existants (tel que Arcade [Ras 03]).

Ea-Miner identifie les aspects fonctionnels et non fonctionnels, et opère à partir des différents documents des exigences, tels que les manuels des utilisateurs, documents des aperçus des systèmes (*system overview documents*), etc [Sam 07]. Si l'outil est applicable à n'importe quel modèle des exigences en identifiant ses concepts, il ne peut l'être à des artefacts, excepté le texte (ce dernier peut être ambigu et plein de contradictions). Il nécessite une grande interaction humaine (ainsi que le temps et l'effort) après l'identification des résultats, afin de les adapter aux résultats réels. Par exemple: éviter les points de vue qui sont très généraux, appliquer le filtrage des synonymes, éditer les exigences si c'est nécessaire. En outre, le catalogue « *lexicon* » (fichier XML utilisé comme une base pour l'extraction des aspects précoces non fonctionnels) utilisé par l'outil doit être modifié par l'utilisateur à chaque nouvelle préoccupation non fonctionnelle trouvée.

4.4.7 Theme/Doc

Theme/Doc est la première partie d'une large approche appelée Theme [Ban 04a, Ban 05], développée par ***Baniassad*** et **Clarke**. Theme est divisée en deux approches : Theme/Doc [Ban 04b] et Them/UML [Cla 05]. Theme/Doc est dédié à analyser les exigences afin d'identifier les préoccupations transversales, tandis que Theme/UML est responsable de la modélisation des préoccupations transversales au niveau conception, en utilisant les artefacts d'UML.

Au niveau des exigences, le processus utilisé par Theme/Doc commence par une analyse manuelle des documents des exigences par le développeur. Cette analyse fournit une liste des actions clés identifiées dans le texte. Ces actions sont obtenues en détectant les verbes sensibles qui apparaissent dans les exigences. Ensuite, l'approche fait une analyse lexicale afin de construire une vue d'action. Une vue d'action représente les relations entre les actions et les exigences où elles apparaissent, en établissant le regroupement des exigences autour des fonctionnalités principales appelées thèmes (*themes*). Un thème est un ensemble de structures et de comportements qui représentent une fonctionnalité. Toute préoccupation, transversale ou non, peut être encapsulée dans un thème.

Chaque thème doit être conçu séparément, or, les thèmes peuvent être parfois dispersés sur différentes exigences, et enchevêtrés avec les fonctionnalités des autres thèmes. Par conséquent, un thème peut être de base ou un thème transversal. Comme plusieurs actions sont liées à une seule exigence, cette dernière contient les actions liées à plus d'une exigence. Par conséquent, il existe des actions qui partagent des exigences et des actions liées à plusieurs exigences. L'identification des thèmes transversaux est dirigée par l'analyse des exigences partagées. Lorsqu'une exigence est partagée au moins par deux différentes actions, le développeur doit analyser quelle action clé est réalisée dans l'exigence. Le but final est d'avoir une vue d'action où chacune est isolée avec les exigences qui lui sont uniquement liées.

Les exigences non fonctionnelles n'ont aucune action associée, et par conséquent, Theme/Doc ne traite pas ce type d'exigences. L'approche Theme/Doc suggère que, dans ce cas, les exigences peuvent être réécrites afin d'inclure les mots d'actions. Cependant, cela suppose que de telles préoccupations peuvent être identifiées par l'ingénieur des

exigences. En outre, l'approche Thème/Doc est limitée concernant les exigences des systèmes larges, et elle nécessite d'effectuer une recherche préliminaire manuelles des mots clé.

4.4.8 HASoCC

Dans [Ami 08], l'auteur propose une approche hybride et orientée aspects pour la séparation des préoccupations transversales (*Hybrid Approach for Separation of Crosscutting Concerns HASoCC*). Cette approche traite en même temps les exigences fonctionnelles, en utilisant le modèle de cas d'utilisation, et les exigences non fonctionnelles en utilisant NFR [Chu 00], avec une représentation XML inspirée du modèle de PREview [Saw 96]. En outre, HASoCC résout les conflits entre les préoccupations durant la phase de l'ingénierie des exigences.

L'approche commence par l'identification des exigences fonctionnelles, en utilisant les cas d'utilisation et en interagissant avec les intervenants. Ces exigences seront ensuite raffinées, avec les descriptions textuelles des cas d'utilisation, les représentations graphiques et les spécifications formelles. Ensuite, les exigences fonctionnelles transversales seront identifiées. Si un cas d'utilisation apparait dans plusieurs autres cas d'utilisations, alors il est considéré comme transversal.

Les exigences non fonctionnelles sont capturées en parallèle avec les exigences fonctionnelles, en utilisant le catalogue des exigences non fonctionnelles [Chu 00]. Afin de détecter les exigences non fonctionnelles transversales, une matrice d'interaction entre les cas d'utilisations et les exigences non fonctionnelles identifiées est utilisée. Les lignes représentent les cas d'utilisation, tandis que les colonnes sont les exigences non fonctionnelles. Un parcours colonne par colonne de la matrice peut détecter les exigences non fonctionnelles transversales. Une exigence non fonctionnelle est transversale si elle entrecoupe au moins deux cas d'utilisation.

Cette approche identifie les exigences transversales fonctionnelles et non fonctionnelles. Cependant, une préoccupation qui est transversale à travers le code source peut ne pas être détectée comme transversale au niveau des exigences, car ces dernières ne reflètent pas directement l'implémentation du système logiciel.

4.4.9 Clustering (ACE)

La technique du clustering a été utilisée pour regrouper les exigences autour des termes dominants. Dans [Dua 07], les auteurs ont proposé une technique appelée ACE (*Aspect Clustering Engine*) qui utilise un modèle probabiliste pour calculer la similarité entre les différentes exigences, et utilise un algorithme hiérarchique modifié [Jai 88] pour rassembler les groupes similaires. Les exigences sont initialement regroupées autour des termes dominants, qui tendent à représenter les ensembles des fonctionnalités, tâches des utilisateurs, et parfois des préoccupations transversales.

Malheureusement, il existe des préoccupations transversales qui sont dispersées à travers d'autres groupes de termes dominants. A cet effet, une deuxième phase est nécessaire pour identifier ces préoccupations dispersées.

Dans cette deuxième phase, les termes dominants (partagés par plusieurs exigences) sont identifiés et retirés de chaque groupe initial, et les exigences sont regroupées une autre fois sans tenir compte de ces termes. En conséquence, les groupes sont formés autour des termes les plus récessifs, permettant l'identification des préoccupations transversales manquées dans la première étape. Les groupes contenant les exigences qui partagent les mêmes termes dominants sont des ensembles des exigences transversales.

Les auteurs dans [Dua 07] proposent l'exemple suivant, avec R l'ensemble d'exigences, et T l'ensemble des termes dominants. Soit $R = \{r1, r2, \ldots . rn\}$, $T = \{a, b, c, \ldots \ldots, z\}$, tel que : $r1 = \{a, b, c, d\}$, $r2 = \{a, a, b, d\}$, $r3 = \{d, f, g, g\}$, $r4 = \{c, f, f, g\}$, $r5 = \{c, h, i, j, j\}$, et $r6 = \{j, j, k, k\}$.

Selon la phase initiale de l'approche ACE, un ensemble de groupes $C = \{c1, c2, \ldots, cn\}$ est obtenu, tels que $c1 = \{r1, r2\}$, $c2 = \{r3, r4\}$ et $c3 = \{r5, r6\}$.

Le groupe $c1$ est formé autour des termes a, b et d, le groupe $c2$ est formé autour des termes f et g, et le groupe $c3$ est formé autour du terme j. Le terme c se produit dans plusieurs exigences ($r1, r4$ et $r5$), et ces exigences sont dispersées dans les trois groupes. Le retrait temporaire du terme dominant c ramène à la construction du nouveau groupe $c4 = \{r1, r4, r5\}$, qui contient l'ensemble des exigences transversales.

ACE augmente le niveau d'automatisation de l'identification des aspects précoces (*early aspects*). Cependant, ACE identifie uniquement la dispersion comme symptôme de la transversalité. En outre, un terme dominant est défini comme tout terme qui se produit dans plus de 50% des exigences dans un groupe. Ce critère est subjectif.

4.4.10 Réseau de Petri

La théorie des réseaux de Petri (*Petri net*) est née de la thèse de ***Carl Adam Petri*** [Pet 62]. Le réseau de Petri est une méthode basée mathématique pour modéliser et vérifier les artefacts du logiciel. Le réseau de Petri fournit des sémantiques précises et claires, une notation graphique intuitive, et plusieurs techniques et outils pour leur analyse, simulation et exécution.

Un réseau de Petri est un moyen de modélisation du comportement des systèmes dynamiques à événements, et de description des relations existantes entre des conditions et des évènements.

La méthode proposée dans [Abd 10] définit les exigences et les préoccupations d'une manière formelle par les réseaux de Petri, en les considérant respectivement, comme des réseaux des exigences et des réseaux de préoccupations. Chaque exigence est représentée avec un réseau de Petri, et par conséquent une préoccupation sera un ensemble de réseaux des exigences, et le système sera, à son tour, un réseau de Petri. L'entité logique qui appartient à plusieurs réseaux des exigences est enchevêtrée, et le réseau de préoccupation qui comprend ce réseau d'exigence, est considéré comme une préoccupation transversale.

Différentes opportunités peuvent être associées à l'utilisation des réseaux de Petri comme méthode formelle pour l'identification des aspects sont [Abd 10] :

- Le réseau de Petri constitue un langage de modélisation exécutable, qui peut identifier les aspects en exécutant le modèle construit pour le système ;
- La compréhensibilité des réseaux de Petri est plus grande que celles des autres méthodes formelles ;

- Le réseau de Petri a des outils pour le modéliser et l'exécuter, et on peut utiliser ces outils afin d'implémenter la méthode sans désigner de nouveaux outils.

Cependant, la spécification de toutes les exigences pour chaque préoccupation n'est pas facile à première vue, et certaines exigences sont spécifiées par une ré-consultation (*reviewing*).

4.5 Approches d'identification d'aspects au niveau architecture

Les architectures du logiciel sont les représentations de la conception à un niveau élevé d'abstraction. Elles facilitent la communication entre différents intervenants, et permettent le partitionnement efficace et le développement parallèle du système logiciel. En outre, elles fournissent des moyens pour indiquer les décisions de conception, et les opportunités pour la réutilisation [Bas 98, Sha 96].

Les approches de l'architecture sont classées comme suit [Chi 05] :

- Approches de modélisation de l'architecture (visuelles et textuelles) : ce sont des langages de description architecturaux (*Architectural Description Languages ADL*) représentés par les approches ACME [Gar 00], Chiron-2 [Med 96a, Med 96b], Aesop [Kom 11, Gar 94], Darwin [Dar 11, Mag 94], Rapide [Rap 11, Luc 02], et Wright [All 97], et les approches qui utilisent l'UML représentées par [Med 02].
- Approches pour le processus de conception du logiciel, telles que la conception de l'architecture orientée exigences (*Requirements-driven Architecture Design*) [Jac 99] et la conception de l'architecture orientée domaine (*Domain-driven Architecture Design*).
- Approches de l'évaluation de l'architecture, telle que la méthode d'analyse de l'architecture du logiciel (*Software Architecture Analysis Method SAAM*) [Dob 02] et la méthode d'analyse *Trade-Off* de l'architecture (*The Architecture Trade-Off Analysis Method ATAM*) [Kaz 00].

Ces approches de l'architecture ne font pas, cependant, une distinction explicite entre les préoccupations architecturales conventionnelles qui peuvent être localisées à l'aide des abstractions architecturales, et les préoccupations architecturales qui recoupent multiples composants architecturaux. Le risque est que les aspects potentiels pourraient être facilement négligés lors de la conception de l'architecture du logiciel, et cela n'est pas toujours résolu au niveau conception et programmation. Cela peut conduire à un code enchevêtré et/ou dispersé dans le système où les facteurs de qualité, que les méthodes d'analyse de l'architecture tentent de vérifier, seront encore restreints. Semblable à la notion d'aspect au niveau de la programmation, ces préoccupations sont transversales et désignent les aspects architecturaux.

Puisque la propriété de transversalité des aspects architecturaux est inhérente, elle ne peut pas être annulée simplement en redéfinissant l'architecture du logiciel et en utilisant des abstractions architecturales classiques. En fait, comme pour les diverses abstractions de la programmation orientée aspect, on a besoin de mécanismes explicites pour identifier, préciser et évaluer les aspects au niveau de l'architecture.

A cet effet, des approches orientées aspect de la conception d'architecture ont apparu. Cependant, ces approches n'identifient pas les aspects d'une manière claire, mais introduisent plutôt la notion d'aspect dans la modélisation architecturale. Les approches de l'architecture orientées aspect sont peu nombreuses. Dans ce qui suit, nous allons en citer quelques-unes.

4.5.1 DAOP-ADL

DAOP-ADL [Pin 03] est un langage de description (modélisation) de l'architecture basé XML, pour décrire l'architecture d'une application en termes d'un ensemble de composants (décrivant le comportement fonctionnel) et un ensemble d'aspects et leurs interconnections. Les dépendances entre les préoccupations transversales sont exprimées à travers les propriétés partagées.

Le processus pour la conception de l'architecture basée DAOP-ADL commence par la description de tous les composants et les aspects qui peuvent être instanciés dans une application. Ensuite, les informations sur la composition entre les composants, les

relations entre les aspects et la composition entre les composants et les aspects sont fournies. Après cette phase, ces informations peuvent être validées afin de vérifier si certaines disparités existent.

4.5.2 AOGA

Les AOGA (*Aspect-Oriented Generative Approaches*) [Kul 04a, Kul 04b] sont des approches pour le processus de conception du logiciel. Elles sont centrées architecture, et ont été initialement décrites dans [Gar 04] et ensuite étendues dans [Kul 04a, Kul 04b, Kul 04c], afin de supporter les systèmes multi-agents (*Multi-Agent Systems MAS*). Bien que ces approches aient été initialement appliquées au domaine du MAS, les concepts introduits sont généraux et non limités à ce domaine. L'idée de base de l'approche AOGA est de promouvoir l'intégration des technologies génératrices et orientées aspect pour faciliter la modélisation du domaine, la spécification de l'architecture et la génération du code des fonctionnalités transversales, dès les premiers stades de développement du logiciel. Les aspects peuvent être capturés et spécifiés dans les étapes préliminaires du développement, avant même l'étape de l'architecture. En ce sens, AOGA a défini des extensions à des modèles de fonction [Cza 00] et un nouveau langage spécifique au domaine (*Domain-Specific Language DSL*), outre la notation basée UML, afin d'exprimer les aspects architecturaux. L'objectif de l'approche AOGA est d'offrir aux développeurs de logiciels les moyens de modularisation des fonctionnalités transversales de façon progressive. Cet objectif de niveau supérieur est, à son tour, décomposé en trois sous-objectifs:

- soutenir l'identification et la spécification des aspects du domaine;
- permettre l'identification et la spécification des aspects architecturaux;
- automatiser la génération du code de l'architecture orientée aspect.

Dans l'approche AOGA, un nouveau type de relation est défini, appelé transversal. Une fonctionnalité *A* entrecoupe une fonctionnalité *B*, lorsque *A* ou une de ses sous-fonctionnalités dépend de *B* ou de l'une de ses sous-fonctionnalités. Les aspects architecturaux (ou composants aspectuels) sont les aspects au niveau architecture. Chaque

composant aspectuel est lié à plus d'un composant architectural, et par conséquent, il représente ainsi sa nature transversale.

4.5.3 ASAAM

ASAAM (*Aspectual Software Architecture Analysis Method*) est une approche de l'évaluation de l'architecture, visant à identifier et spécifier explicitement les aspects architecturaux au début du cycle de vie du logiciel [Tek 04]. L'approche s'appuie sur les méthodes d'analyse de l'architecture basées scénario. L'avantage d'ASAAM est le soutien systématique de la gestion des aspects architecturaux de manière explicite.

Les deux artefacts principaux dans ASAAM sont les scénarios et les composants architecturaux. Il existe trois types de scénarios: directs, indirects et aspectuels. Un scénario direct peut être directement effectué. Un scénario indirect nécessite un changement d'un composant. Un scénario aspectuel représente un aspect potentiel, et peut être soit direct ou indirect, et il est dispersé à travers plusieurs composants. Il existe quatre types de composants:

- Un composant cohésif est un élément qui est bien défini, et exécute des scénarios sémantiquement proches ;
- Un composant mal défini est un composant constitué de plusieurs sous-composants, chacun effectue un ensemble de scénarios sémantiquement proches ;
- Un composant enchevêtré est un composant qui effectue un scénario aspectuel qui est, directement ou indirectement, effectué par le composant ;
- Un composant composite comprend des scénarios sémantiquement distincts mais qui ne peuvent pas être décomposés, ou qui ne comprend pas un scénario aspectuel.

Les aspects architecturaux sont les artefacts de sortie, et peuvent être utilisés pour refactoriser l'architecture.

4.6 Approches d’identification d’aspects au niveau conception

L'activité de conception d'un processus de développement du logiciel donne au concepteur la possibilité de raisonner sur un système logiciel, défini par un ensemble d'exigences. Ce processus de raisonnement sur le système implique l’étude du comportement nécessaire pour le système, afin d’atteindre ses objectifs, et une structure correspondante pour supporter ce comportement. Le résultat de l'activité de conception est un ensemble de modèles qui caractérisent et spécifient le comportement et la structure du système. Ces modèles peuvent être à différents niveaux d'abstraction, selon le niveau de détails du raisonnement du concepteur.

Bien que le système soit divisé en unités modulaires (qui peuvent être utilisées séparément), une conception est créée lors du développement, qui peut se disperser et s’enchevêtrer sur différentes préoccupations du système. Comme les préoccupations ne sont pas séparées, les changements dans une préoccupation auront un impact sur d'autres préoccupations liées.

Les approches non orientées aspect de la conception du logiciel ne prennent pas en compte la modularisation des préoccupations dispersées ou/et enchevêtrées. Les approches orientées aspect de la conception sont apparues afin de permettre une séparation des préoccupations au niveau conception. Les préoccupations transversales peuvent être conçues séparément comme des aspects.

Le langage de modélisation unifié (*Unified Modeling Language UML*) [Omg 13] est le langage de conception standard de l'OMG pour la conception orientée objet. C'est le langage de conception orientée objet le plus largement utilisé. L'UML a également été la base à partir de laquelle la plupart des langages de conception orientés aspect ont été développés.

Selon notre étude faite sur l’identification d’aspects au niveau conception, rares sont les approches qui font l’identification des aspects. Cependant, plusieurs approches existent visant l’intégration des aspects dans les diagrammes UML. Parmi ces approches, nous citons les suivantes :

- L'approche *SUP* (*State charts and UML Profile*) [Ald 01] supporte un processus pour l'analyse et la conception orientée aspect, basée sur les diagrammes d'états, complémentée par un langage de développement orienté aspect basé sur un profile UML [Ald 01, Ald 03]. Un aspect est spécifié dans les diagrammes de classes en utilisant le stéréotype « *aspect* », et dans les diagrammes d'états comme un ensemble d'états qui sont connectés à travers une série d'évènements. La spécification du comportement transversal est effectuée à travers l'utilisation des diagrammes d'états, et modélisée comme un évènement qui déclenche une transition d'états. Au niveau des diagrammes de classes, l'intégration est spécifiée à travers un type d'association qui indique la transversalité, noté par le stéréotype « *crosscut* ».

- Une autre approche propose l'utilisation des aspects dans les modèles d'activité [Bar 02]. Les diagrammes d'activité peuvent modéliser un grand nombre de flux de contrôle. Si on ajoute ou on modifie une sorte de comportement dans le système existant, on doit ajouter quelques flux de contrôle supplémentaires et augmenter la taille du modèle. Pour réduire cette augmentation de la complexité, les exigences transversales devraient être implémentées comme des activités. L'auteur propose l'utilisation des aspects dans les modèles d'activité. Chaque aspect prend la forme d'une nouvelle activité qui est fusionnée avec les activités existantes à travers les nœuds d'interface, comme une invocation d'activité. L'implémentation des aspects comme activités a l'avantage d'éviter l'introduction des constructions UML supplémentaires, puisque les diagrammes d'activité peuvent supporter les exigences transversales à travers l'utilisation de simples invocations d'activité. Ceci permet un couplage minimal et simplifie la traçabilité [Bar 02].

- La méthode Theme/UML [Cla 05] fait la conception et la modélisation en utilisant l'UML et en introduisant un moyen pour écrire les aspects. Elle étend le méta-modèle UML. Chaque préoccupation, transversale ou non, peut être encapsulée dans un thème [Cla 99]. A la conception, les thèmes sont des vues partielles des exigences. Un enchevêtrement entre les thèmes se produit lorsque les exigences pour différents thèmes décrivent partiellement un concept de domaine. Cet enchevêtrement est considéré dans différentes aspects, dans des thèmes qui

représentent les concepts du domaine. Theme/UML supporte la représentation des thèmes de base et thèmes aspects. Tous ces thèmes sont spécifiés dans des packages, et contiennent les diagrammes de classes qui représentent la structure d'aspect. Le diagramme de classe dans le package thème représente les concepts de conception qui sont nécessaires pour la réalisation des exigences liées au thème.

- L'approche *UFA* (*UML For Aspects*) est une approche de développement orienté aspect, basée sur le modèle aspectuel de collaboration (*Aspectual Collaborations Model ACM*) [Her 02]. C'est une autre approche d'extension d'UML. Les aspects réutilisables ou de haut niveau sont modélisés, indépendamment du contexte dans lequel ils seront appliqués. Un aspect est spécifié comme un package abstrait. Les packages UML sont utilisés pour les parties encapsulées d'un système, qui contribuent à un comportement complexe. Un package d'aspect peut être spécialisé pour entrecouper une application spécifique.

4.7 Discussion des approches

La table 4.2 récapitule les approches d'identification d'aspects au niveau exigences, présentées dans la section 4.4, selon les mêmes critères utilisés dans la table 3.1.

La complexité d'identification des préoccupations transversales au niveau code source a nécessité une prise en charge précoce de l'identification d'aspects, au niveau exigences, architecture ou conception. Les approches d'identification d'aspects durant le cycle du développement de logiciel présentent plusieurs avantages, nous citons les suivants:

- L'identification précoce des préoccupations transversales permet l'application des techniques du développement orientés aspect ;
- La prise en charge précoce des préoccupations transversales permet à l'architecte de concevoir un système qui satisfait mieux les exigences des intervenants, au lieu de se concentrer uniquement sur les préoccupations fonctionnelles ;
- En absence de la détection précoce des préoccupations transversales, ces dernières tendent à passer inaperçues, et deviennent étroitement liées (tissées) dans la

conception, empêchant les développeurs d'identifier une conception architecturale optimale, d'où la nécessité ultérieure des efforts de la refactorisation.

Cependant, plusieurs insuffisances jalonnent les approches d'identification d'aspects durant le cycle du développement de logiciel:

- La plupart des approches du niveau exigences identifient ces dernières en se basant sur les interviews avec les intervenants. Ces dernières sont souvent imprécises, pleins de contradictions et manquent d'information essentielle [Ras 05] ;
- L'application des approches analysant les exigences a été limitée au seul traitement du texte ;
- Certaines approches souffrent de la subjectivité, du fait de l'identification à priori des préoccupations transversales, à partir de l'expérience du concepteur ou du développeur, outre la modélisation de préoccupations transversales connues d'avance par la communauté AOSD (*Aspect Oriented Software Development*) ;
- Il y a méconnaissance des modalités de détermination des dimensions d'aspect, ainsi que leur impact sur d'autres phases de développement [Con 10] ;
- Le manque d'une définition précise d'aspect rend les préoccupations transversales identifiées durant les phases du cycle de développement logiciel, ne peuvent pas toutes être refactorisées dans des aspects ;
- L'utilisation des documents des exigences comme entrée pour l'identification des préoccupations transversales conduit à des faux aspects, car les préoccupations transversales dépendent de la décomposition des exigences selon le modèle sélectionné (un modèle des exigences choisi peut enlever l'enchevêtrement ou la dispersion) ;
- Des préoccupations transversales détectées dans les phases précoces du cycle de développement du logiciel peuvent être changées dans des phases ultérieures: disparaissent ou émergent, car les artefacts considérés peuvent ne pas refléter le code source.

Tab. 4.2. Table récapitulative et comparative des approches d'identification d'aspects au niveau exigences

N° Section	Technique	Entrée	Sortie	Symptômes détectés		Transversalité est :	Intervention humaine
				Disp	Ench		
4.4.1.2	Arcade	Préoccupations non fonctionnelles + Exigences	Préoccupations*Exigences	X		Préoccupation liée à plusieurs exigences	Identification des préoccupations et des exigences des intervenants
4.4.2.2	AOV-graph	Buts fonctionnels + Buts non fonctionnels	Relations entre les buts	X		Tâche qui contribue à plusieurs buts	Raffinement des buts
4.4.3	Cas d'utilisation	Diagramme de cas d'utilisation	Relations entre exigences et cas d'utilisation	X		Exigences non fonctionnelles/ Cas d'utilisation qui affecte plusieurs cas d'utilisations	Identification des exigences non fonctionnelles
4.4.4	AOREC	Exigences	Composants et leurs exigences	X		Caractéristique (contrainte non fonctionnelle) du système pour laquelle les composants fournissent ou nécessitent des services	Analyse des exigences et identification des composants et les aspects qui les entrecoupent
4.4.5	Cosmos	Exigences	Préoccupations et leurs relations + Prédicats	X		Préoccupation partagée ou réutilisée	Identification des préoccupations et sélection des préoccupations pertinentes
4.4.6	Ea-Miner	Exigences	Mots regroupés dans des catégories selon leurs sens	X		Exigence ayant un mot qui a des correspondances	Identification des principales exigences des intervenants
4.4.7	Theme/Doc	Exigences	Exigences regroupées selon des thèmes	X	X	Exigence partagée par différents thèmes	Analyse manuelle des documents des exigences pour identifier les actions clés
4.4.8	HASOCC	Cas d'utilisations + catalogue des exigences non fonctionnelles	• Relations entre les cas d'utilisations • Cas d'utilisations *Exigences non fonctionnelles	X		•Cas d'utilisation lié à d'autres cas d'utilisation •Exigence non fonctionnelle entrecoupe plusieurs cas d'utilisation	Identification des exigences fonctionnelles en interaction avec les intervenants

4.4.9	Clustering	Termes dominant des exigences	Groupes des exigences qui partagent les mêmes termes dominants.	X		Exigences appartenant à différents groupes, partagent des termes dominants dispersés dans différentes exigences	Inspection des termes dominants
4.4.10	Réseau de Petri	Réseaux des exigences + Réseaux de préoccupations	Dépendances entre les réseaux des exigences et les réseaux de préoccupations		X	Réseau de préoccupation qui comprend le réseau d'exigence enchevêtrée (contenant l'entité logique qui appartient à plusieurs réseaux d'exigences)	Consultation des réseaux des exigences

4.8 Conclusion

Dans ce chapitre, nous nous sommes attachés à étudier l'importance du domaine de l'identification d'aspects durant le cycle du développement de logiciel (*early aspect mining*). Nous avons ensuite explicité les avantages et les limites des approches existantes d'identification précoce d'aspects.

Afin de remédier aux insuffisances des approches actuelles, nous proposons dans le prochain chapitre, une nouvelle approche d'aspect mining.

CHAPITRE 5

NOUVELLE APPROCHE D'IDENTIFICATION D'ASPECTS BASÉE SUR LE MODÈLE D'INVOCATIONS DE MÉTHODES

« When you first start off trying to solve a problem, the first solutions you come up with are very complex, and most people stop there. But if you keep going, and live with the problem and peel more layers of the onion off, you can often times arrive at some very elegant and simple solutions.»

(Steve Jobs 2006)

5.1 Introduction

Notre étude faite sur les approches d'identification d'aspects nous a permis de constater que l'identification peut être faite sous une double perspective, à savoir, celles de la rétro-ingénierie et du développement orienté aspect.

La première perspective englobe les approches qui identifient les aspects au niveau implémentation, en utilisant le code source, ou les modèles générés à partir de ce dernier, tandis que la seconde perspective représente les approches qui identifient les aspects durant les phases du cycle de développement du logiciel (avant l'implémentation).

L'identification d'aspects, selon ces perspectives, connait des points forts et des points faibles majeurs:

- Identifier les aspects durant les phases du cycle de développement du logiciel a l'avantage de gérer tôt la transversalité, et par conséquent, de bénéficier d'un développement orienté aspect. Or, cette identification présente un risque, selon lequel le système logiciel n'est pas dans sa version finale (certains aspects peuvent disparaitre et d'autres émerger).
- Identifier les aspects à partir de l'implémentation présente l'avantage d'utiliser la version finale du système logiciel. Or, cette identification est difficile, car les aspects sont fortement intégrés avec le code métier, et ce dernier est lié à la technologie d'implémentation.

Nous proposons une nouvelle approche d'identification d'aspects à partir des systèmes logiciels orientés objet, en essayant de garder les points forts et dépasser les points faibles cités ci-dessus. D'autre part, il s'agit d'unifier le processus d'identification d'aspects, afin qu'il soit applicable pour les deux perspectives (perspective de la rétro-ingénierie et celle du développement orienté aspect).

Notre approche utilise le diagramme UML de séquence [Omg 13]. A cet effet, il s'agit d'une approche qui exploite les relations d'interactions internes des éléments du système, car ces relations portent une information utile (indépendante de la technologie d'implémentation et de la syntaxe de programmation). Ces relations sont disponibles au niveau conceptuel, représentées par les diagrammes d'interactions d'objets, ou au niveau implémentation représentées par les relations d'invocations de méthodes.

Nous utilisons dans notre approche un niveau plus abstrait que celui du code source. Ceci permet de rendre l'approche d'identification d'aspects plus générale et indépendante de la technologie d'implémentation. En outre, nous utilisons un niveau d'abstraction plus proche du code source, car l'identification d'aspect, durant les phases du cycle de développement, s'avère meilleure si elle est faite sur un niveau plus proche de la version finale qui représente l'ensemble des fonctionnalités du système.

5.2 Présentation générale de notre processus unifié d'identification d'aspects

Nous proposons le modèle de notre processus d'identification d'aspects, illustré par la figure 5.1, que ce soit au niveau du système en cours de développement, ou au niveau d'un système déjà implémenté (*legacy system*). L'entrée de notre processus est un ensemble de modèles du système. Ces modèles sont construits dans la phase du développement (par exemple les diagrammes de séquence UML), ou obtenus par la rétro-ingénierie des systèmes existants (par exemple les diagrammes UML générés à partir du code source) ou encore en utilisant les graphes d'appels de méthodes à partir du code source. Le processus analysera les interactions représentées par les différents modèles, afin d'obtenir les préoccupations dispersées et celles enchevêtrées.

Une préoccupation qui a été identifiée comme dispersée (ou enchevêtrée) ne doit pas être identifiée une autre fois comme enchevêtrée (ou dispersée). A cet effet, nous devons filtrer (éliminer et fusionner) les aspects candidats obtenus, afin d'éviter l'identification dupliquée. Ainsi, afin de déterminer les préoccupations ayant un niveau élevé de transversalité, des métriques doivent être calculées. Cela permet de détecter les erreurs possibles dans le processus d'identification d'aspects, et aider à choisir les aspects les plus pertinents à refactoriser.

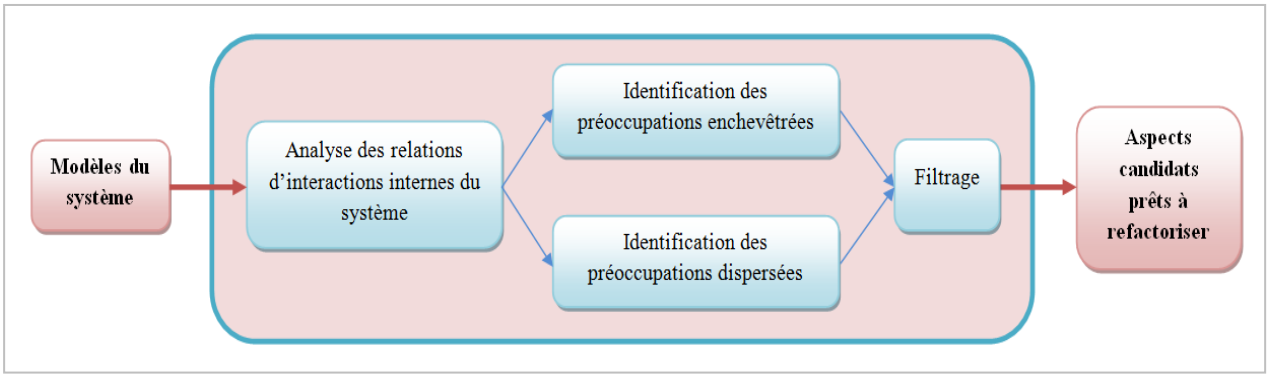


Fig. 5.1. Notre processus général d'identification d'aspects

5.3 Présentation des modèles du système logiciel

L'UML (*Unified Modeling Language*) [Omg 13] est devenu une référence importante dans l'ingénierie du logiciel orienté objet. Les diagrammes de séquence sont des diagrammes UML utilisés pour décrire la communication entre les différentes parties d'un système (composants ou objets), ou entre un système et ses utilisateurs. Le diagramme de séquence met l'accent sur l'ordre chronologique dans lequel se produit l'échange de messages entre les objets. Dans la plupart des cas, la réception d'un message est suivie par l'exécution d'une méthode de classe [Uml 14].

Un message peut comporter une garde (*guard*), ce qui signifie que le message n'est pas envoyé, sauf si certaines conditions sont remplies. Une garde est une expression conditionnelle qui contrôle un opérande d'interaction dans un fragment combiné (garde). Grâce à l'utilisation des fragments combinés, l'utilisateur sera capable de décrire un certain nombre de traces, d'une manière compacte et concise. Nous distinguons les gardes suivantes qui ont un impact sur la transversalité :

- **Alt**: elle représente les fragments multiples alternatifs (équivalent à : *« si...alors...sinon...* »), et une seule des deux branches sera réalisée dans un scénario donné (elle désigne un choix). L'utilisation de l'équivalent de l'opérateur *alors* permet d'indiquer que la branche est exécutée si la condition de « Alt » est vraie. L'utilisation de l'opérateur *sinon* permet d'indiquer que la branche est exécutée si la condition de « Alt » est fausse [Seq 14].
- **Opt**: elle représente un fragment optionnel, un comportement qui peut se produire ou non. Un fragment optionnel est équivalent à un fragment « Alt » qui ne posséderait pas de *sinon* (qui n'aurait qu'une seule branche). Un fragment optionnel est donc une sorte de « *si...alors...* » [Seq 14].
- **Par**: cette garde est utilisée pour représenter des interactions ayant lieu en fragments parallèles (traitement compétitif) [Seq 14].
- **Loop**: cette garde est utilisée pour décrire un ensemble d'interactions qui s'exécute en boucle (fragment répétitif). En général, une contrainte indiquant le nombre de

répétitions (minimum et maximum), ou une condition booléenne, doit être respectée [Seq 14].

- **Critical**: cette garde désigne une section critique (seulement un *thread* à la fois). Cette dernière permet d'indiquer que les interactions décrites dans cette garde ne peuvent pas être interrompues par d'autres interactions décrites dans le diagramme. On dit que l'opérateur impose alors un traitement atomique des interactions qu'il contient [Seq 14].

5.4 Principe de l'approche proposée

L'approche proposée utilise les diagrammes conceptuels de séquence qui sont détaillés et finaux, afin qu'ils puissent refléter le code source du système logiciel. Elle est basée sur les informations portées par ces diagrammes: la plupart des interactions (transmissions de messages) entre les différents objets du diagramme de séquence reflètent les invocations de méthodes dans l'implémentation, et l'ordre chronologique des transmissions de messages reflète l'ordre d'exécution des tâches du système.

A partir des diagrammes de séquence, l'approche proposée s'intéresse seulement aux messages qui invoquent les méthodes, et les entités de base à analyser sont les méthodes et leurs invocations. Ces entités dureront au niveau implémentation. A cet effet, les aspects détectés apparaissent dans l'implémentation comme attendus.

Un diagramme de séquence capture le comportement d'un seul cas d'utilisation. Pour cette raison, l'approche proposée considère tous les diagrammes de séquence du système logiciel.

L'identification des préoccupations transversales proposée est basée sur un modèle du système, selon les transmissions de messages (figure 5.2). Le système logiciel est modélisé avec une collection des diagrammes de séquence décrivant son comportement. Chacun de ces diagrammes représente un ensemble d'objets qui transmettent des messages. Une méthode est invoquée dans des messages transmis par des objets, et ces derniers sont des instances de classes, composées par un ensemble d'attributs et de méthodes (ligne rouge). La méthode invoquée est définie (appartient) dans une classe

(ligne bleue). Les messages peuvent être guidés par des gardes. Une garde est faite sur des attributs et/ou des méthodes. Chaque méthode est invoquée avant/après une autre.

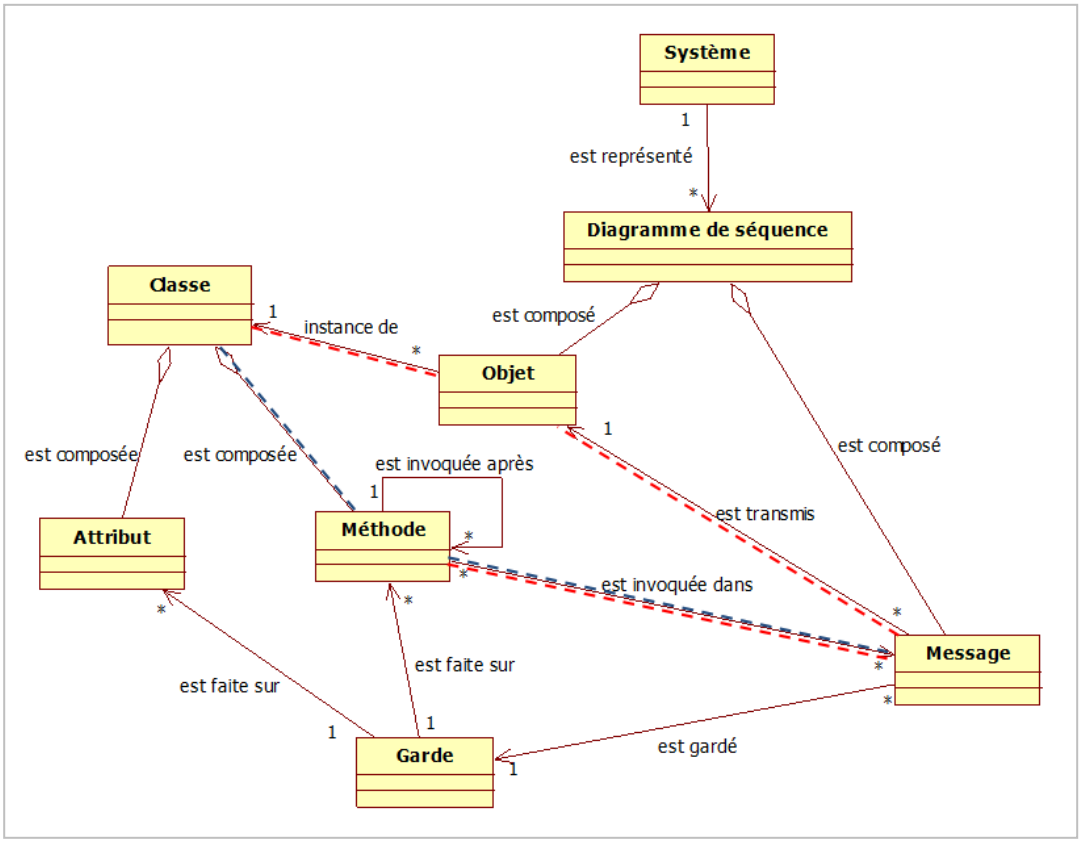


Fig. 5.2. Modèle du système via les transmissions de messages

5.5 Modèles de la transversalité

La transversalité est définie comme la présence de la dispersion et/ou l'enchevêtrement. La dispersion se produit lorsque les invocations d'une méthode sont dispersées dans les transmissions de messages, tandis que l'enchevêtrement marque la présence d'un modèle récurrent d'invocation (*Recurrent Invocation Model RIM*) ou la garde « Alt » (figure 5.3). Afin de faciliter la notation, nous utilisons la notation *O.m* pour dénoter le message transmis par l'objet *O* qui invoque la méthode *m*.

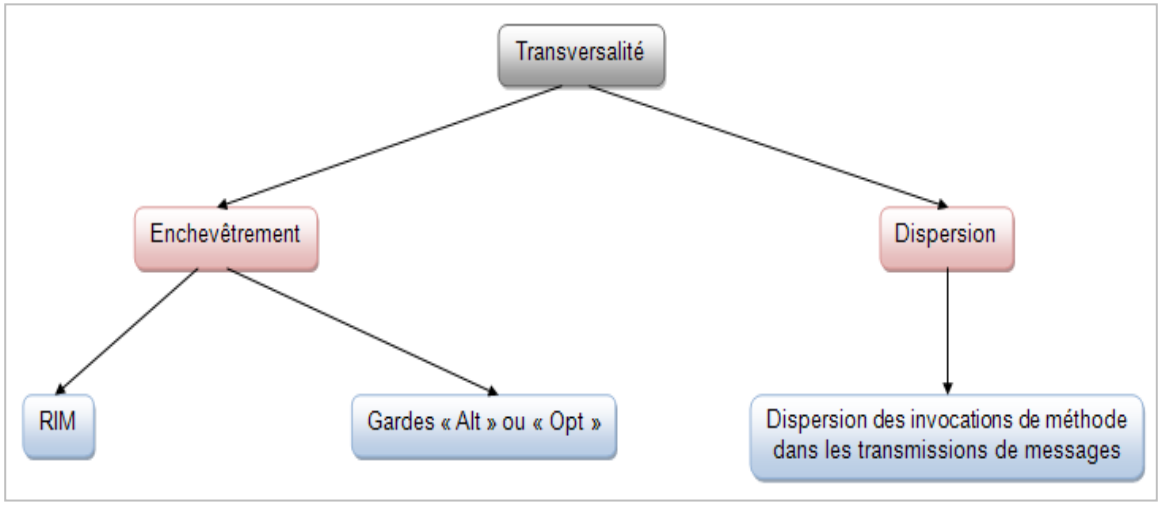


Fig. 5.3. Modèle de transversalité basé sur les invocations de méthodes

5.5.1 Modèle d'enchevêtrement

Deux situations peuvent indiquer l'existence de l'enchevêtrement dans les diagrammes de séquence. La première consiste en la présence d'un modèle récurrent d'invocation (RIM), tandis que le second marque la présence de la garde « Alt ». Un RIM consiste, au moins, en deux invocations successives de méthodes (à travers les transmissions de messages) qui sont répétées en un nombre de fois supérieur à un certain seuil.

Nous considérons un RIM comme un symptôme d'enchevêtrement, si les méthodes qui le composent appartiennent au moins à deux classes différentes. L'existence de méthodes qui n'appartiennent pas à la même classe et toujours exécutées successivement, signifie qu'il existe une tâche (composée par toutes ces méthodes) qui est enchevêtrée dans différentes classes (classes où ces méthodes sont définies). Par conséquent, cela augmente le couplage entre ces classes.

Un RIM peut être un de ces types :

- Type *Before* : un message dans le diagramme de séquence, invoquant la méthode *n*, est toujours transmis avant un autre qui invoque la méthode *m* (figure 5.4.a).
- Type *After* : un message invoquant la méthode *m*, est toujours transmis après celui qui invoque la méthode *n* (figure 5.4.b).

- Type *Symetric* : un message invoquant la méthode *n* est toujours transmis avant un autre qui invoque la méthode *m*, et dans le même temps, celui invoquant la méthode *m* est toujours transmis après celui qui invoque la méthode *n* (figure 5.4.c).

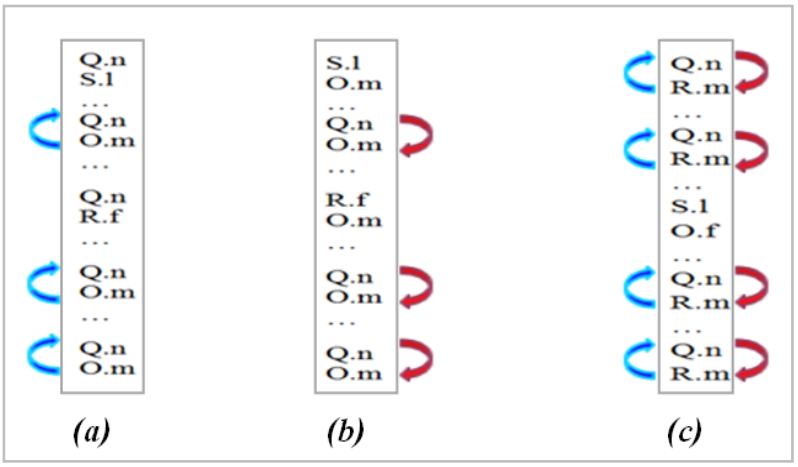


Fig. 5.4. Exemples de types de RIM

Selon le modèle du système présenté dans la figure 5.2 (ligne bleue), la figure 5.5 définit le modèle d'enchevêtrement à partir de RIM. La méthode m_a est définie dans la classe $class_f$. L'invocation de la méthode m_a est suivie par celle de m_b (la flèche signifie la succession d'invocations de méthodes).

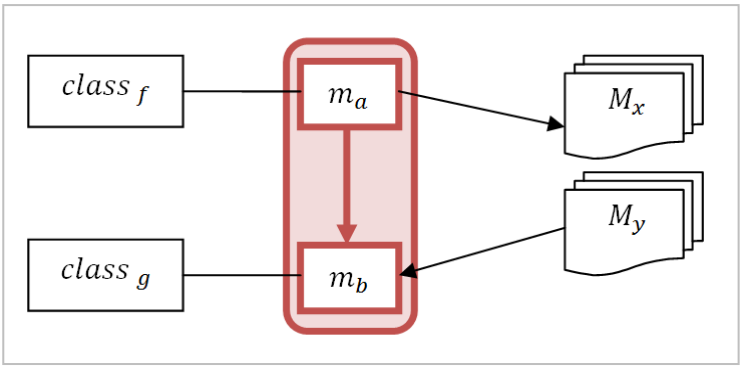


Fig. 5.5. Modèle d'enchevêtrement à partir de RIM

Dans la figure 5.5, la forme avec le font rouge désigne un RIM. Chaque méthode de l'ensemble M_x est invoquée après la méthode m_a, et chaque méthode de l'ensemble M_y

est invoquée avant la méthode m_b. $|M_x|$ indique le nombre de méthodes non dupliquées qui composent l'ensemble M_x.

A partir de la figure 5.5, l'enchevêtrement est détecté selon l'algorithme du listing 5.1. Dans cet algorithme, le type symétrique du code advice signifie que l'aspect détecté est de type *before* (comme dans le listing 5.1), ou de type *after* (code advice= $\{m_b\}$ et Pointcuts= $\{m_a\}$). Nous avons choisi le premier cas, afin de détecter les contraintes d'intégrité référentielle.

```
if (class_f = class_g) then ∄ tangling
else {
if (|M_x| = |M_y| = 0) then {
          RIM type= {symmetric};
          Advice type= {before};
          Code advice= {m_a};
          Pointcuts= {m_b};
          }
else if (|M_x| = 0) then {
          RIM type= {after};
          Advice type= {after};
          Code advice= {m_b};
          Pointcut= {m_a};
          }
else if (|M_y| = 0) then {
          RIM type= {before};
          Advice type= {before};
          Code advice= {m_a};
          Pointcut= {m_b};
          }
else ∄ tangling;
 }
```

Listing 5.1. Algorithme de détection de l'enchevêtrement par l'analyse des RIM

Concernant la garde « Alt », elle relie les transmissions de messages (invocations de méthodes) avec les conditions correspondantes. Si au moins une condition (attribut ou méthode), et la méthode invoquée juste après la condition, appartiennent à deux différentes classes, alors ces classes ne sont pas indépendantes (elles sont liées), cela étant dû à l'existence d'une tâche qui est enchevêtrée.

Soit $Meths$ un ensemble de méthodes sur lesquelles la condition de la garde est faite, et $meth_i \in Meths$. Soit $Attrs$ un ensemble d'attributs sur lesquels la condition de la garde

est faite, et $attr_i \in Attrs$. m_a est la méthode invoquée lorsque les conditions de la garde « Alt » sont vérifiées (valides ou invalides). $Meths'$ est un ensemble de méthodes (non conditions) telle que chaque méthode $meth'_i$ de $Meths'$ est invoquée avant m_a. Classes' est l'ensemble de classes (non dupliquées) où $Meths$ et $Attrs$ sont définis. La figure 5.6 dérivée du modèle du système (présenté dans figure 5.2) illustre le modèle d'enchevêtrement à partir de la garde « Alt », tandis que le listing 5.2 présente l'algorithme de détection d'enchevêtrement par l'analyse de la garde « Alt », selon le modèle de la figure 5.6.

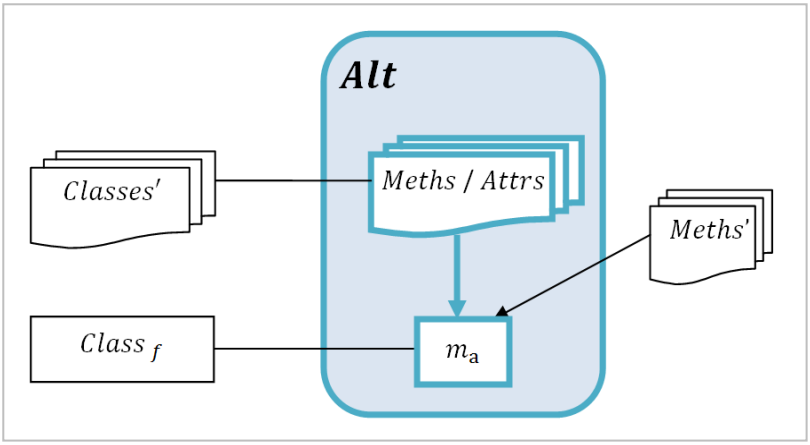


Fig. 5.6. Modèle d'enchevêtrement à partir de la garde « Alt »

```
if (∃ cl1 ∈ Classes') ∧ (cl1 ≠ class_f ) then {
  if (Meths' = ∅) then {
       Advice type= {before};
       Code advice= {meth_1, … , meth_n, attr_1, … , attr_n};
       Pointcuts= {m_a };
  }
  else {
       Advice type= {after};
       Code advice= {m_a };
       Pointcuts= {meth_1, … , meth_n, attr_1, … , attr_n, meth'_1, … , meth'_n};
       }
}
else ∄ tangling;
```

Listing 5.2. Algorithme de détection de l'enchevêtrement par l'analyse de la garde « Alt »

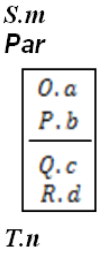
Concernant l'aspect candidat détecté à partir de la garde « Alt », si la méthode m_a est invoquée seulement lorsque les conditions de « Alt » sont vérifiées, alors il existe une contrainte d'intégrité référentielle (cas du deuxième *if* dans le listing 5.2).

Nous utilisons dans cette thèse la notation avec des parenthèses « (, , …) » afin de dénoter le code advice qui contient plus d'une méthode, et les accolades « { , , … } » pour dénoter l'ensemble des points de coupure. Par exemple, le code advice (A, B) signifie que ce fragment contient l'invocation de la méthode A qui est suivie par l'invocation de la méthode B, tandis que l'ensemble des points de coupure $\{C, D\}$ (tel que le code advice est la méthode E et son type est *after*) signifie qu'après l'invocation de C ou D, la méthode E sera invoquée.

Nous considérons la garde « Opt » comme un cas particulier de la garde « Alt », car « Opt » utilise également des conditions. Avec la garde « Opt », une méthode (qui suit les conditions) est invoquée seulement lorsque les conditions de « Opt » sont valides, tandis que avec la garde « Alt » la méthode est invoquée lorsque les conditions sont valides (fragment *if*) ou invalides (fragment *else*).

L'approche proposée prend en compte également d'autres gardes du diagramme de séquence, qui ont un impact sur la détection de l'enchevêtrement :

- ***Par*** : si par exemple nous avons la séquence suivante des invocations de méthodes, identifiée à partir d'une partie du diagramme de séquence :



  Nous ne disons pas, dans ce cas, que la méthode c est invoquée après la méthode b, car les deux fragments séparés par une ligne, se déroulent en parallèle. Nous disons plutôt que la méthode b est invoquée juste après la méthode a, et la méthode c est invoquée juste avant la méthode d.

- ***Loop*** : ici, le même fragment est répété. Si la transmission d'un message (invocation d'une méthode) est faite *n* fois, cela n'indique pas l'existence d'un aspect candidat, car le contexte de cette transmission est le même.

- ***Critical*** : les interactions décrites dans cette garde ne peuvent pas être interrompues par d'autres interactions du diagramme de séquence. Cela implique l'existence d'un aspect enchevêtré, si les méthodes invoquées juste à l'intérieur de la garde appartiennent, au moins, à deux classes différentes (tâche liée et enchevêtrée sur au moins ces deux classes). Son point de coupure est la méthode invoquée juste avant la garde « Critical », et le code advice sera la séquence de méthodes invoquées à l'intérieur du fragment de cette garde.

5.5.2 Modèle de dispersion

La dispersion se produit lorsqu'une méthode est invoquée plusieurs fois dans des messages transmis par des objets de différentes classes (au moins deux classes différentes de la classe où la méthode invoquée est définie). Cela s'explique par le fait que la refactorisation orientée aspect d'une méthode invoquée une seule fois n'améliore pas la structure du logiciel (la modification de la manière de son invocation n'est pas coûteuse). Pour la même raison, une méthode dont ses invocations ne sont pas dispersées (invoquée par la même classe ou par l'objet qui la contient) n'est pas prise en compte.

A partir du modèle de système présenté précédemment dans la figure 5.2 (ligne rouge), nous dérivons dans la figure 5.7 le modèle de dispersion. La flèche désigne la succession d'invocations de méthodes. La méthode m_a est invoquée dans le message $message_i$ transmis par l'objet $object_k$. Ce dernier est une instance de la classe $class_f$. Dans la figure 5.7, $meth'_i \in Meths'$ l'ensemble de méthodes invoquées, telle que chacune est invoquée juste avant la méthode m_a. L'algorithme du listing 3.3 détecte la dispersion. Dans cet algorithme, $class_a$ est la classe où la méthode m_a est définie.

Dans le reste du chapitre, nous utilisons « type » ou « type d'aspect » pour se référer du type du code advice de l'aspect.

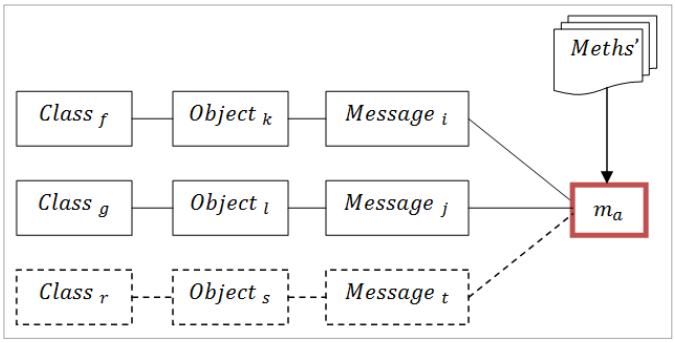


Fig. 5.7. Modèle de dispersion

```
if (class_f ≠ class_g ≠ class_a) then {
    Type= {after};
    Code advice= {m_a};
    Pointcuts= {meth'_1, …, meth'_n};
}
else ∄ scattering;
```

Listing 5.3. Algorithme de détection de la dispersion

5.6 Concrétisation de l'approche

Afin de concrétiser notre approche d'identification d'aspects, nous utilisons une matrice, nommée matrice d'ordre de transmission de messages (*Matrix of Message Transmission Order MMTO*). *MMTO* est construite en exploitant toutes les transmissions de messages qui invoquent des méthodes, à partir des diagrammes de séquence. Cette matrice est binaire et carrée $(n * n)$. Ses lignes et ses colonnes représentent les méthodes invoquées durant les transmissions de messages au niveau des diagrammes de séquence. La relation binaire entre les lignes et les colonnes est l'ordre des transmissions de messages (invocations de méthodes). En d'autres termes, si la méthode j est invoquée juste après la méthode i, la valeur de la cellule $MMTO[i,j]$ sera incrémentée. Soit l_i la méthode située à la ligne i de la matrice $MMTO$ et c_j la méthode située à la colonne j.

Concernant les transmissions des messages avec la garde « Alt », la matrice $MMTO$ est remplie selon le type de sa condition. Si la condition est une méthode, alors nous mettons $MMTO[i,j] = 100$, telle que l_i est la méthode sur laquelle la condition est faite, et c_j est la méthode invoquée lorsque la condition est vérifiée (valide ou invalide). Sinon, si la

condition est un attribut, une autre matrice appelée $MMTO'$ sera construite, telles que ses colonnes représentent les méthodes (comme dans la matrice $MMTO$) et ses lignes sont l'ensemble des attributs qui existent dans les gardes « Alt ». Nous mettons $MMTO'[i,j] = 100$ si l_i est l'attribut sur lequel la condition est faite, et c_j est la méthode invoquée lorsque la condition est vérifiée (valide ou invalide). La valeur élevée ($= 100$) choisie pour les cellules des gardes « Alt » signifie l'existence d'une tâche enchevêtrée (méthodes fortement liées). Initialement, les deux matrices $MMTO$ et $MMTO'$ sont initialisées à zéro.

Si la condition est composée (méthodes et attributs), alors chaque méthode ou attribut de la condition est traité indépendamment, et les aspects candidats obtenus de la condition composée seront fusionnés dans l'étape ultérieure de l'élimination et fusion des aspects dupliqués (section 5.7.1). La valeur de chaque cellule de ces matrices sera utilisée pour détecter l'enchevêtrement.

Nous ajoutons un tableau, de type liste, appelé $TClass$. Ce dernier est ensuite utilisé pour détecter la dispersion à partir des transmissions de messages. Il est initialisé comme suit : pour chaque cellule j du tableau $TClass$, $TClass[j]$ accueille la classe définit la méthode c_j de la matrice $MMTO$. Ce tableau est ensuite rempli, en parallèle, avec la matrice $MMTO$: à chaque fois qu'une méthode est invoquée (à partir des diagrammes de séquence), la classe de l'objet qui transmet le message (où la méthode c_j est invoquée) sera insérée dans la cellule $TClass[j]$ si elle n'est pas déjà insérée. La valeur de chaque cellule j de ce tableau est la liste des classes non dupliquées.

5.6.1 Détection de l'enchevêtrement via RIM

En partant de la matrice $MMTO$, l'enchevêtrement est identifié à partir des cellules ayant les valeurs supérieurs à 1 et inférieurs à 100, comme suit :

1. Nous cherchons une valeur de cellule dans la matrice $MMTO$ tel que : $2 \leq MMTO[i,j] < 100$ (soit dans la cellule $MMTO[i,j]$). Nous mettons : $m_a = l_i$, $m_b = c_j$ (du listing 5.1) ;

2. Pour toutes les cellules de la ligne i (exceptée la cellule $MMTO[i,j]$), nous calculons $|M_x| = \sum_{k=1}^{n} | MMTO[i,k] \neq 0 , k \neq j |$ le nombre des cellules qui ont la valeur différente à zéro ;
3. Pour toutes les cellules de la colonne j (exceptée la cellule $MMTO[i,j]$), nous calculons $|M_y| = \sum_{k=1}^{n} | MMTO[k,j] \neq 0 , k \neq i |$;
4. Nous appliquons l'algorithme du listing 5.1.

5.6.2 Détection de l'enchevêtrement via la garde « Alt »

L'enchevêtrement, via la garde « Alt », est détecté à partir des cellules de la matrice $MMTO$ qui ont la valeur supérieure ou égale 100 (soit la cellule $MMTO[i,j]$) et à partir des cellules de la matrice $MMTO'$. Si les méthodes l_i et c_j appartiennent à deux classes différentes, alors l'enchevêtrement est identifié via la garde « Alt » comme suit :

1. $\forall\, k = 1..n$, si $MMTO[k,j]\ mod\ 100 = 0$ (c'est-à-dire c_j est invoquée seulement quand des conditions sont vérifiées, et par conséquent il existe une contrainte d'intégrité référentielle) alors :
   - Point de coupure = $\{c_j\}$;
   - Code advice = $\{l_i\}$;
   - Type= $\{before\}$.
2. Sinon :
   - Points de coupure = L , tel que $\forall\, l_k \in L,\ MMTO[k,j] \neq 0,\ k = 1..n$ ($l_k \in \{meth_1, \dots, meth_n,\ attr_1, \dots, attr_n,\ meth'_1 \dots, meth'_n\}$ du listing 5.2);
   - Code advice = $\{c_j\}$;
   - Type = $\{after\}$.

L'enchevêtrement est identifié à partir de la matrice $MMTO'$ selon le premier cas (aspect détecté est de type *before*), car la matrice $MMTO'$ ne contient que des méthodes liées à des conditions.

5.6.3 Détection de la dispersion

En partant du tableau $TClass$, la dispersion est identifiée à partir des cellules où $|TClass[j]| > 2$. Chacune de ces cellules indique l'invocation d'une méthode dans deux classes différentes de la classe où elle est définie.

Pour chaque cellule du tableau $TClass$, si $|TClass[j]| > 2$ alors :

1. A partir de la matrice $MMTO$, nous mettons $m_a = c_j$ (du listing 5.3) ;
2. L est l'ensemble des méthodes l_k, tel que $\forall\, l_k \in L,\ MMTO[k,j] \neq 0,\ k = 1..n$, (équivalent à $\{meth'_1, \dots, meth'_n\}$ du listing 5.3) ;
   - Points de coupure = L ;
   - Code advice = $\{m_a\}$ (équivalent à m_a du listing 5.3) ;
   - Type = $\{after\}$.

5.7 Filtrage des aspects candidats

Afin d'évaluer notre approche proposée, et détecter les erreurs possibles dans le processus d'identification d'aspects (faux aspects candidats), les aspects candidats obtenus seront filtrer. Il s'agit d'un filtrage à deux passes :

1. Première passe : élimination et fusion des aspects dupliqués ;
2. Deuxième passe : sélection des aspects pertinents, en utilisant des métriques.

5.7.1 Elimination et fusion des aspects dupliqués

Les aspects qui sont détectés comme enchevêtrés (ou dispersés) ne devront pas être détectés une autre fois comme dispersés (ou enchevêtrés). Afin d'éliminer les aspects dupliqués (faux aspects positifs) qui ont été obtenus à partir de l'identification de l'enchevêtrement et la dispersion, les aspects candidats doivent être filtrés (éliminés et/ou fusionnés). Soit Pt_i l'ensemble des points de coupure de l'aspect candidat $aspect_i$, Ad_i le code advice de $aspect_i$, $type_i$ le type de Ad_i et $Sympt_i$ le symptôme de $aspect_i$.

- Pour les aspects candidats ayant le même ensemble du code advice et le même type, nous gardons l'ensemble du code advice et le type, et nous fusionnons les ensembles des points de coupure et les symptômes (si ces derniers sont différents). Soient les aspects candidats : $aspect_1$, $aspect_2$, … $aspect_{\mathrm{n}}$, tel que $Ad_1 = Ad_2 = \cdots = Ad_{\mathrm{n}}$, et $type_1 = type_2 = \cdots = type_n$, nous fusionnons ces aspects dans un aspect candidat $\boldsymbol{aspect_{fusion}}$, tel que $Ad_{fusion} = Ad_1$, $type_{fusion} = type_1$, $Pt_{fusion} = Pt_1 \cup Pt_2 \cup \ldots \cup Pt_n$, $Sympt_{fusion} = Sympt_1 \cup Sympt_2 \cup \ldots \cup Sympt_n$.

- Pour les aspects candidats ayant le même ensemble de points de coupure et le même type, nous gardons l'ensemble de points de coupure et le type, et nous fusionnons les ensembles du code advice et les symptômes (si ces derniers sont différents). Soient les aspects candidats : $aspect_1$, $aspect_2$, … $aspect_{\mathrm{n}}$, tel que $Pt_1 = Pt_2 = \cdots = Pt_{\mathrm{n}}$, et $type_1 = type_2 = \cdots = type_n$, nous fusionnons ces aspects dans un aspect candidat $\boldsymbol{aspect_{fusion}}$, tel que $Pt_{fusion} = Pt_1$, $type_{fusion} = type_1$, $Ad_{fusion} = Ad_1 \cup Ad_2 \cup \ldots \cup Ad_n$, $Sympt_{fusion} = Sympt_1 \cup Sympt_2 \cup \ldots \cup Sympt_n$.

- Pour un aspect candidat dispersé et un autre enchevêtré ayant le même type de code advice et le même ensemble des points de coupure, si l'ensemble du code advice de l'aspect dispersé est inclus dans, ou égale à, celui de l'aspect enchevêtré, alors nous supprimons l'aspect dispersé, car il est déjà encapsulé dans l'aspect enchevêtré, et nous fusionnons les ensembles des symptômes. Soient les aspects candidats $aspect_1$ et $aspect_2$, tel que $Pt_1 = Pt_2$, $type_1 = type_2$, $sympt_1$ =dispersé et $Sympt_2$ = enchevêtré, et $Ad_1 \subseteq Ad_2$, alors nous fusionnons ces aspects dans un aspect candidat $\boldsymbol{aspect_{fusion}}$, tel que $Ad_{fusion} = Ad_2$, $type_{fusion} = type_2$, $Pt_{fusion} = Pt_2$, $Sympt_{fusion} = Sympt_1 \cup Sympt_2$.

- S'il existe les aspects candidats $aspect_1$ et $aspect_2$, tel que $Ad_1 = Pt_2$, $Ad_2 = Pt_1$ et $type_1 \neq type_2$, alors l'aspect de type *after* sera supprimé (soit $aspect_2$), les symptômes seront fusionnés (s'ils sont différents), et donc, l'aspect résultat sera $\boldsymbol{aspect_{fusion}}$, tel que $Ad_{fusion} = Ad_1$, $type_{fusion} = type_1$, $Pt_{fusion} = Pt_1$, $Sympt_{fusion} = Sympt_1 \cup Sympt_2$.

Cette phase de l'élimination et de la fusion des aspects dupliqués permet de prendre en compte le cas où, plus de deux méthodes sont impliquées dans l'enchevêtrement, ou la dispersion.

5.7.2 Métriques orientées aspects

Les métriques permettent de détecter les erreurs possibles dans le processus d'identification d'aspects, et aident l'utilisateur à choisir les aspects candidats les plus pertinents. A cet effet, nous proposons trois nouvelles métriques : métrique du couplage des classes (*Metric of Classes Coupling MCC*), métrique de dispersion (*Metric of Scattering MS*) et métrique d'enchevêtrement (*Metric of Tangling MT*).

La métrique *MCC* est calculée pour chaque aspect candidat détecté, après la phase d'élimination et fusion des aspects dupliqués, tandis que les métriques *MS* et *MT* sont calculées respectivement pour l'aspect candidat dispersé et enchevêtré, cela après chaque identification d'aspect et avant la phase d'élimination et fusion des aspects dupliqués, afin de ne pas perdre les données de chaque aspect candidat détecté.

5.7.2.1 Métrique du couplage des classes (*MCC*)

L'existence des préoccupations transversales dispersées ou enchevêtrées incrémente le couplage entre les classes, et rend ces dernières dépendantes, ce qui détruit la modularité orientée objet. Concernant l'aspect dispersé, la modification de la manière d'invoquer une méthode relie la classe où cette méthode est définie, avec les classes qui invoquent cette méthode. Concernant l'aspect enchevêtré, il relie la classe où une condition est définie (cette condition est une méthode ou attribut), avec celle de la méthode invoquée lors de la vérification de la condition.

La métrique *MCC* mesure le degré du couplage des classes, causé par l'existence d'un aspect candidat. Ce couplage résulte des relations de ces classes avec cet aspect candidat.

Soit Asp un aspect candidat obtenu par notre approche. Soit Pt l'ensemble de points de coupure de Asp et Ad est l'ensemble du code advice de Asp. *MCC* est calculée comme suit :

$$MCC\ (Asp) = |classes\_couplées|$$

Concernant un aspect enchevêtré Asp, $\forall\, cl \in classes\_couplées$, cl est une classe qui contient au moins une méthode appartient à Ad ou Pt. La valeur de *MCC* est supérieure à 1 pour l'aspect enchevêtré.

Concernant un aspect dispersé Asp, $classes\_couplées$ est l'ensemble de classes où la méthode du code advice Ad a été invoquée, ainsi que la classe qui définit cette méthode. $classes\_couplées = TClass[Ad]$. *MCC* est supérieure à 2 pour l'aspect dispersé. La valeur élevée de *MCC* signifie que l'aspect candidat Asp a plus de chance d'être refactorisé.

5.7.2.2 Métrique d'enchevêtrement (*MT*)

La métrique *MT* mesure l'enchevêtrement d'un aspect candidat Asp. Si Asp est détecté à partir d'une cellule $MMTO[i,j]$ ou $MMTO'[i,j]$, la valeur élevée de cette cellule signifie que cet aspect candidat Asp a plus de chance d'être refactorisé, car la préoccupation qu'il encapsule est plus enchevêtrée à travers des classes. *MT* est calculée comme suit :

$MT(Asp) = MMTO[i,j]$, si Asp est détecté à parir de la matrice $MMTO$.

$MT(Asp) = MMTO'[i,j]$, si Asp est détecté à parir de la matrice $MMTO'$.

La valeur de *MT* est supérieure à 1.

5.7.2.3 Métrique de dispersion (*MS*)

La métrique *MS* mesure la dispersion d'un aspect candidat Asp. Elle représente le nombre de classes (non dupliquées) où la méthode du code advice Ad a été invoquée.

$$MS\ (Asp) = |TClass[Ad]| - 1 = MCC(Asp) - 1.$$

La valeur élevée de *MS* signifie que la préoccupation encapsulée dans Asp est plus dispersée, ce qui donne plus la chance à l'aspect candidat Asp d'être refactorisé. La valeur de *MS* est supérieure à 1.

5.7.2.4 Métriques des aspects fusionnés

Si les aspects candidat $Asp1$ et $Asp2$ sont fusionnés (dans l'étape de la section 5.7.1) dans $Asp3$, alors:

$$MT(Asp3) = MT(Asp1) + MT(Asp2)$$

$$MS(Asp3) = MS(Asp1) + MS(Asp2)$$

$MCC(Asp3)$ est calculée directement à partir de l'aspect $Asp3$, et $classes\_couplées$ est l'union des ensembles des classes liées pour les deux aspects (selon la section 5.7.1).

5.7.2.5 Discussion

En comparaison avec les définitions de la section 2.6.4, le couplage entre les classes du système orienté objet est mesuré dans notre approche avec la métrique *MCC*. Si les classes sont couplées (c'est-à-dire *MCC* est élevée), cela signifie que ces classes ne sont pas bien modularisées, à cause de l'existence d'un aspect. La solution que nous avons proposée consiste à garder les classes non couplées, et encapsuler séparément celles couplées, dans les aspects correspondants. Un système logiciel (orienté aspect) d'une structure améliorée sera obtenu.

Notre approche utilise également la métrique fan-in (section 3.5.3), afin de mesurer le degré de la dispersion (la métrique *MS*). En encapsulant les méthodes ayant fan-in élevée (à condition qu'elles soient invoquées dans différentes classes) dans des aspects séparément, le système logiciel (orienté aspect) obtenu sera plus modulaire.

5.8 Autre variante du processus d'identification d'aspects appliquée au code

Le processus général d'identification d'aspects, présenté dans la section 5.2, peut être appliqué différemment, en gardant les mêmes définitions de la transversalité, d'enchevêtrement et de dispersion, mais en changeant le niveau d'abstraction et le contexte, afin d'identifier les préoccupations transversales au niveau des systèmes déjà implémentés (*legacy systems*). Nous identifions les aspects au niveau implémentation, et d'une manière statique, à la différence de l'approche de la section 5.4, où l'identification a été faite en utilisant les scénarios des cas d'utilisation exprimés par les diagrammes de séquence.

La variante proposée dans cette section est également basée sur l'ordre des invocations de méthodes, ainsi que les répétitions de ces invocations. Les modèles du système (de la figure 5.1) sont présentés par les graphes d'invocations de méthodes, que nous générons à partir du code source du système logiciel. Ces graphes sont: GMIO (*Graph of Method Invocation Order*) et GMIS (*Graph of Method Invocation Structure*). Cela permet à rendre l'identification des aspects indépendante de la technologie d'implémentation. Les graphes sont construits simultanément, à partir du code source du système logiciel.

Dans la première variante de notre approche, nous avons utilisé une matrice et un tableau, du fait de la facilité de leur implémentation. Le choix de l'utilisation des graphes, développé dans un deuxième temps, se justifie par le fait que, au niveau code source, l'insertion de plusieurs méthodes dans la matrice augmente la complexité (en termes de temps) de notre approche. Aussi, l'utilisation de deux graphes s'avère plus appropriée, compte tenu de l'existence des algorithmes de parcours optimisé.

5.8.1 Graphe GMIO

Le graphe GMIO est construit en exploitant l'ordre d'invocations de méthodes. Ses nœuds sont les noms de méthodes, tandis que les arcs représentent les relations de succession d'invocations de méthodes. Les arcs sont dirigés (des flèches) et établis à partir des successions de toutes les invocations de méthodes, selon la structure du code source.

Durant la construction du graphe GMIO, une valeur est attribuée à chaque arc, indiquant le nombre de fois où deux méthodes sont invoquées successivement dans le code source. Par exemple, si la méthode N est invoquée juste après la méthode M, alors l'arc entre M et N est établi (de M vers N) avec la valeur 1 (nous pouvons ne pas mettre la valeur de l'arc qui est égale à 1), et cette valeur sera incrémentée à chaque fois qu'une invocation de la méthode M est suivie par l'invocation de N.

Soit la séquence suivante d'invocations de méthodes, apparaissant dans la classe Cl :

```
Class Cl {
        A() {
            C()
            D()
            E()
            }
        B() {
            K()
            L()
            }
       ...
       }
```

Dans le graphe GMIO, nous disons que la méthode C est suivie par D qui est suivie, à son tour, par E, et K est suivie par L. Nous ne pouvons pas dire que la méthode A est suivie par B, car l'ordre de définitions des méthodes dans une classe n'est pas significatif. La figure 5.8.(a) présente la partie du graphe GMIO de cette séquence d'invocations de méthodes.

Soit la séquence suivante d'invocations de méthodes :

```
M() {
    A () {
        C()
        D()
        E()
        }
    B() {
        K()
        L()
        }
    ...
     }
```

Nous disons, dans ce cas, que la méthode A est suivie par B, la méthode C est suivie par D qui est suivie par E, et K est suivie par L. La figure 5.8.(b) présente la partie du graphe GMIO de cette séquence d'invocations de méthodes.

Les gardes des diagrammes de séquence (utilisées dans la variante de la section 5.4) sont remplacées au niveau implémentation par les fragments conditionnels *« IF »* du code (*« if…then…else… »* ou *« if…then… »*). Concernant ces derniers, les arcs situés entre la méthode du fragment *« if »* et celle dans les fragment *« then »* et *« else »*, sont établis avec une valeur égale à 100 (et elle sera ensuite incrémentée en ajoutant la valeur 100) afin de désigner deux invocations de méthodes qui sont fortement dépendantes (méthodes liées). L'arc qui relie la condition avec une méthode dans le fragment *« else »*, est établi

avec une ligne discontinue. La figure 5.8.(c) présente la partie du graphe GMIO de l'exemple : $if\ B()\ then\ C()\ else\ D()$.

Si le fragment « *then* » (ou « *else* ») contient plusieurs méthodes, alors un arc est établi entre la condition et la première méthode du fragment « *then* » (et avec la première méthode du fragment « *else* » si ce dernier existe). La figure 5.8.(d) présente la partie du graphe GMIO de l'exemple : $if\ A()\ then\ \{B()\ \ C()\}\ else\ \{D()\ \ E()\}$.

Si la condition est un attribut, alors nous ajoutons un nœud contenant cet attribut. Un arc sera établi entre ce nœud et la méthode du fragment *« then »* (et un autre arc avec la méthode du fragment « *else* » si ce dernier existe). Soit at un attribut. La figure 5.8.(e) présente la partie du graphe GMIO de l'exemple : $if\ at\ then\ B()\ else\ C()$.

La boucle est traitée comme une condition « *IF* » répétée dans le même contexte.

Soit M une méthode appartenant à la classe $Cl1$, et les classes $Cl2$ et $Cl3$ étendent (*extend*) la classe $Cl1$. Si M a des invocations dans $Cl2$ et $Cl3$, alors ces invocations ne sont pas dispersées.

Dans la figure 5.9, nous avons pris l'exemple présenté dans la section *"Experimental Evaluation"* de [Qu 07]. Nous supposons que les conditions (*condition2*) et (*condition4*) sont les méthodes $Cond2$ et $Cond4$ respectivement, tandis que les conditions (*condition1*), (*condition5*) et (*condition6*) sont respectivement les attributs $Cond1$, $Cond5$ et $Cond6$. Le graphe GMIO, construit à partir du code source de la figure 5.9, est présenté dans la figure 5.10.

Afin de parcourir tous les chemins possibles des exécutions, nous exploitons dans le graphe GMIO (figure 5.10) toutes les séquences possibles des invocations de méthodes. Par exemple, la première invocation (dans la figure 5.9) de la méthode B est suivie par l'invocation de A, si la condition $Cond1$ est valide, et elle est suivie par F si la condition $Cond1$ n'est pas valide.

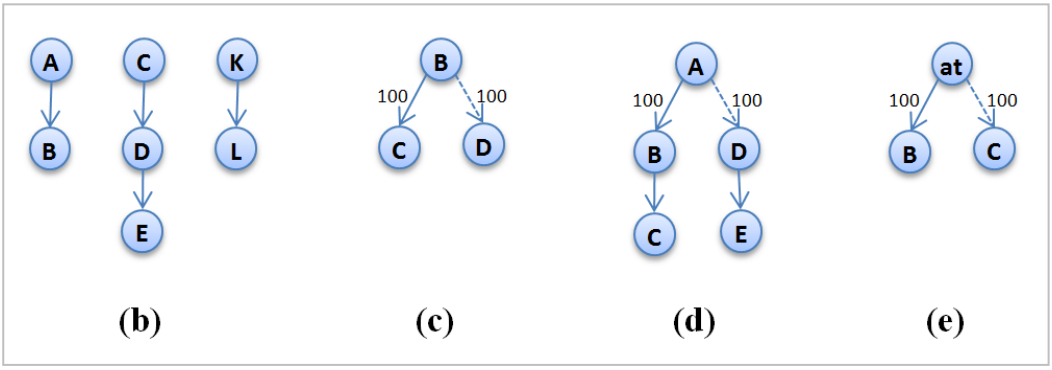


Fig. 5.8. Exemples des parties du graphe GMIO

```
main(){
        B();
        while (condition1){
                A();
                B();
        }
        F();
        J();
        G();
        H();
        A();
        B();
        D();
}
A(){
}
B(){
        C();
        if (condition2)
           J();
        if (condition3) {
           G();
           if (condition4)
              F();
           else L();}
}
C(){
        if (condition5) {
           G();
           H();
        }
}
```

```
D(){
        C();
        A();
        B();
        K();
        I();
        G();
        E();
}
E(){
}
F(){
        K();
        I();
}
G(){
}
H(){
}
I(){
        if (condition6)
           J();
}
J(){
}
K(){
}
L(){
        K();
}
```

Fig. 5.9. Exemple d'un code source [Qu 07]

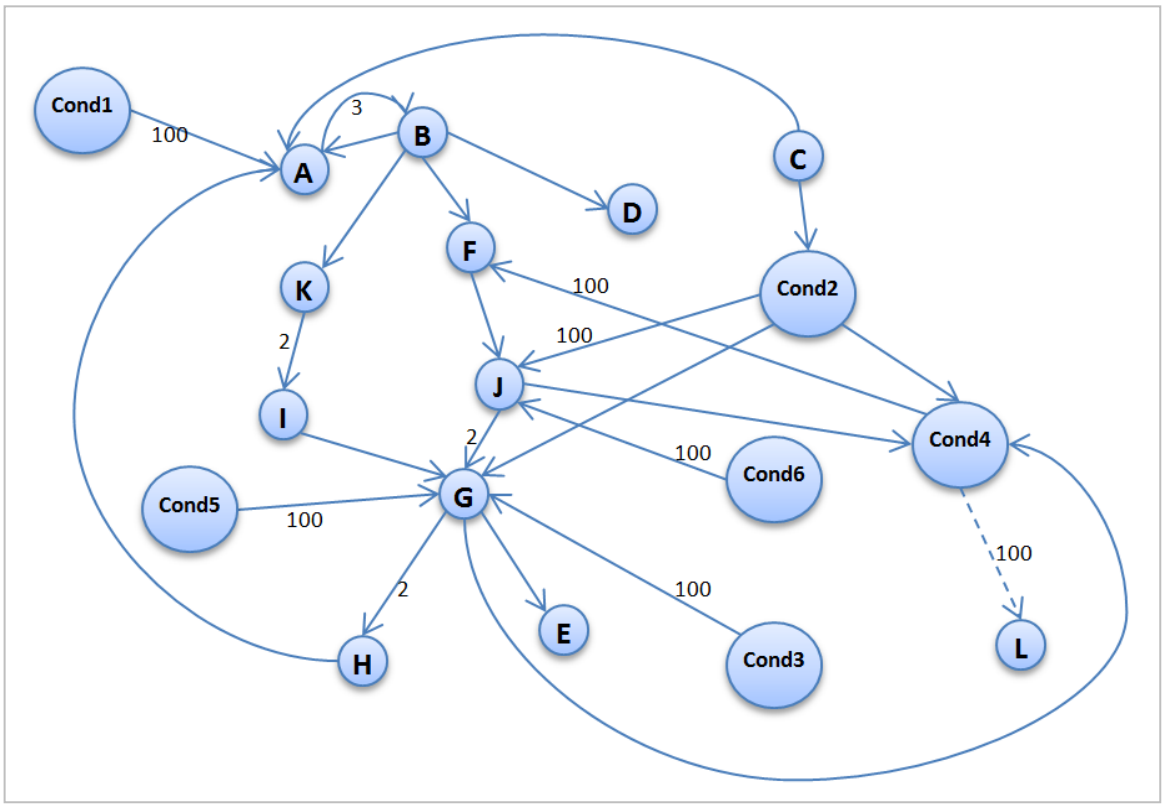


Fig. 5.10. Graphe GMIO du code source de la figure 5.9

5.8.2 Graphe GMIS

Le graphe GMIS est construit en respectant la structure du code source du logiciel. Il permet d'identifier les invocations dispersées des méthodes. Ses nœuds sont les noms des méthodes, soit M et N deux nœuds (méthodes) du graphe GMIS. Si le corps de la méthode M contient au moins une invocation de la méthode N, alors un arc de M vers N sera établi dans le graphe GMIS. A partir de ce dernier, les invocations dispersées d'une méthode sont représentées par les méthodes invoquantes qui appartiennent à différentes classes. Le graphe GMIS construit à partir du code source de la figure 5.9 est présenté dans la figure 5.11.

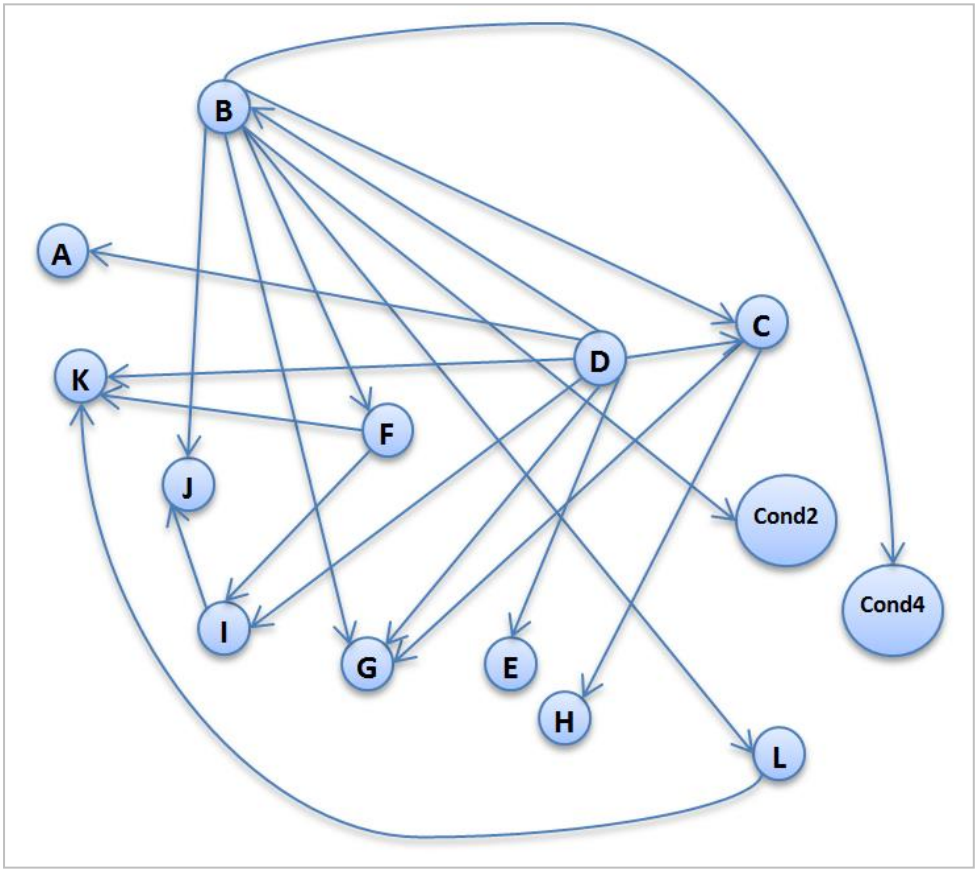


Fig. 5.11. Graphe GMIS du code source de la figure 5.9

5.8.3 Identification de la transversalité

Nous utilisons la notation NdI_{M} pour dénoter l'ensemble de tous les nœuds entrants du nœud M à partir du graphe $GMIO$, et NdI'_{M} l'ensemble des nœuds entrants du nœud M à partir du graphe $GMIS$. NdO_{M} dénote l'ensemble des nœuds sortants du nœud M à partir du graphe $GMIO$. Concernant l'exemple de la figure 5.10, $NdI_{\mathrm{A}} = \{B, C, H, Cond1\}$, $NdI'_{\mathrm{A}} = \{D\}$ et $NdO_{\mathrm{A}} = \{B\}$. $|NdI_{\mathrm{A}}| = 4$ est le nombre de nœuds de l'ensemble NdI_{A}. $AV_{\mathrm{J,G}} = 2$ est la valeur de l'arc entre le nœud J vers le nœud G. Les nœuds attributs sont ignorés dans l'identification de l'enchevêtrement via RIM.

Le fragment répétitif (« *For* », « *While* », « *Repeat* ») est traité comme le fragment conditionnel « *IF* », car le contexte des invocations de ses méthodes ne change pas avec les répétitions.

5.8.3.1 Identification de l'enchevêtrement

L'enchevêtrement est identifié à partir du graphe GMIO. L'arc ayant une valeur supérieure à 1 et inférieure à 100 indique que les deux méthodes, reliées par cet arc, sont invoquées successivement (existence d'un RIM), et par conséquent elles sont liées. Cela signifie qu'il existe une tâche enchevêtrée (composée par ces deux méthodes).

Nous supprimons l'aspect candidat qui est enchevêtré seulement sur deux classes, telle que l'une de ces dernières est interne (appartiennent au langage d'implémentation), car cet aspect ne cause pas une dépendance de deux classes.

L'enchevêtrement est détecté (via RIM) à partir du graphe GMIO, pour chaque arc entre les nœuds M et N du graphe GMIO, tel que: $1 < AV_{\mathrm{M,N}} < 100$, ($M$ et N sont deux méthodes liées par un arc de M vers N). Soit les méthodes M et N, appartenant respectivement à la classe $class_M$ et $class_N$. L'identification de l'enchevêtrement se fait selon le listing 5.4, en suivant les mêmes étapes de la section 5.6.1, avec :

- $MMTO[M,N] = AV_{\mathrm{M,N}}$
- $l_i = M$
- $c_j = N$
- $M_x = NdO_{\mathrm{M}}$
- $M_y = NdI_{\mathrm{N}}$

Afin de vérifier les conditions de l'enchevêtrement et de la dispersion, nous supposons que les méthodes et les attributs du code source de la figure 5.9 appartiennent à des classes $Class1$, $Class2$, $Class3$ et $Class4$ comme suit :

***Class1**: A, K, Cond1, Cond2*

***Class2**: C, D, E, F, Cond3, Cond4*

***Class3**: G, H, I, Cond5*

***Class4**: B, J, L, Cond6*

A partir du graphe GMIO de la figure 5.10, les aspects candidats enchevêtrés identifiés via RIM sont présentés dans la table (Tab) 5.1.

```
if (class_M = class_N) then ∄ tangling
else {
if (|NdO_M| = |NdI_N| = 0) then {
      RIM type= {symmetric};
      Advice type= {before};
      Code advice= {M};
      Pointcut= {N};
}
else if (|NdO_M| = 0) then {
      RIM type= {after};
      Advice type= {after};
      Code advice= {N};
      Pointcut= {M};
}
else if (|NdI_N| = 0) then {
      RIM type= {before};
      Advice type= {before};
      Code advice= {M};
      Pointcut= {N};
}
else ∄ tangling;
}
```

Listing 5.4. Algorithme d'identification de l'enchevêtrement via RIM

Tab. 5.1. Liste des aspects candidats enchevêtrés identifiés via RIM du code source de la figure 5.9

Aspect candidat	Symptôme	Code advice	Points de coupure	Type
1	Tangling	{ A }	{ B }	before
2	Tangling	{ K }	{ I }	before

L'arc ayant une valeur supérieure ou égale à 100 signifie que l'invocation de la méthode du nœud cible dépend de la condition du nœud source (existence du fragment conditionnel). Par conséquent, il existe une tâche enchevêtrée.

Soit M (M une méthode ou un attribut) et N appartiennent respectivement à la classe $class_M$ et $class_N$. A partir du graphe GMIO, pour chaque arc entre le nœud M vers N, tel que $AV_{\mathrm{M,N}} \geq 100$ et M une méthode ou un attribut, nous appliquons l'algorithme du listing 5.5. Nous mettons dans le listing 5.5 « $(M = valid)$ » si l'arc est avec une ligne continue, et « $(M = invalid)$ » si l'arc est avec une ligne discontinue.

Si la méthode N est invoquée seulement lorsque des conditions sont vérifiées, alors il existe une contrainte d'intégrité référentielle (cas du deuxième *if* dans le listing 5.5).

A partir du graphe GMIO de la figure 5.9, les aspects candidats enchevêtrés identifiés à partir des fragments conditionnels sont présentés dans la table 5.2.

```
if (class_N = class_M) then ∄ tangling
else {
if (∀ K ∈ NdI_N , AV_K,N mod 100 = 0) then {
      Advice type= {before};
      Code advice= {(M = valid)};
      Pointcut= {N};
}
else {
      Advice type= {after};
      Code advice= {N};
      Pointcuts= NdI_N ;
}
}
```

Listing 5.5. Algorithme d'identification de l'enchevêtrement via les fragments conditionnels

Tab. 5.2. Liste des aspects candidats enchevêtrés identifiés via les fragments conditionnels du code source de la figure 5.9

Aspect candidat	Symptôme	Code advice	Points de coupure	Type
3	Tangling	{ J }	{ (Cond2= valid) , (Cond6=valid), F }	after
4	Tangling	{ G }	{ (Cond3= valid), (Cond2=valid), (Cond5=valid), I, J }	after
5	Tangling	{ (Cond4=invalid) }	{ L }	before

5.8.3.2 Identification de la dispersion

La dispersion est identifiée à partir du graphe GMIS. Un nœud N ayant plus d'un nœud entrant, signifie que la méthode N est invoquée plus d'une fois, et ces invocations peuvent être dispersées, si elles appartiennent à différentes classes. Les méthodes invoquantes du nœud N sont celles des nœuds entrants du nœud N. La modification de la manière d'invocation de la méthode N nécessite des changements dans celles invoquantes.

Nous supprimons l'aspect candidat dispersé, telles que:

- toutes les méthodes de l'ensemble de points de coupure soient des méthodes internes ;

- toutes les méthodes de l'ensemble de son code advice soient internes ;

Cela s'explique par le fait que la modification d'une méthode interne, ou sa signature, est impossible.

A partir du graphe GMIS, la dispersion est identifiée selon le listing 5.6, tel que N est un nœud du graphe GMIS, avec K ($K \in class_k$) et L ($L \in class_L$), deux méthodes de l'ensemble NdI_{N}, et $N \in class_N$.

```
if (|NdI'N| ≥ 2) then {
  if (∃ K,L ∈ NdI'N: classL ≠ classK ≠ classN) then {
      Advice type= {after};
      Code dvice= {N};
      Pointcuts= NdIN ;
      }
  else ∄ Scattering
}
else ∄ Scattering
```

Listing 5.6. Algorithme d'identification de la dispersion

A partir du graphe GMIS de la figure 5.10, les aspects candidats dispersés identifiés sont présentés dans la table 5.3.

Tab. 5.3. Liste des aspects candidats dispersés du code source de la figure 5.9

Aspect candidat	Symptôme	Code advice	Point de coupure	Type
6	Scattering	{ G }	{ (Cond3=valid), (Cond2=valid), (Cond5=valid), I, J }	after
7	Scattering	{ K }	{ B }	after

5.8.4 Filtrage des aspects candidats

Nous éliminons et nous fusionnons les aspects dupliqués qui résultent de l'identification de l'enchevêtrement et la dispersion, selon les mêmes étapes présentées dans la section 5.7.1.

A partir de la table 5.2 et 5.3, les aspects candidats 4 et 6 ont le même code advice (la méthode G) et le même type (*after*). Par conséquent, nous fusionnons les ensembles de points de coupure des aspects 4 et 6, ainsi que les symptômes. L'aspect candidat résultant

est l'aspect 6 de la table 5.4. Les aspects candidats finaux obtenus sont présentés dans la table 5.4.

Dans la section 5.7.2, nous avons proposé trois métriques orientées aspect : métrique de couplage des classes (*MCC*), métrique d'enchevêtrement (*MT*) et métrique de dispersion (*MS*), afin de valider la qualité des aspects candidats identifiés au niveau conceptuel (diagrammes UML). Dans cette variante de l'approche (de la section 5.8), les mêmes métriques ont été adaptées pour qu'elles soient applicables au niveau implémentation.

Les résultats de l'application des métriques sur les aspects candidats finaux de la table 5.4 sont présentés dans la table 5.5. En appliquant ces métriques, nous constatons que les aspects candidats obtenus apparaissent pertinents, et chacun d'eux rend au moins deux classes dépendantes.

Tab. 5.4. Liste des aspects candidats finaux identifiés à partir du code source de la figure 5.9

Aspect candidat	**Symptôme**	**Code advice**	**Point de coupure**	**Type**
1	Tangling	{ A }	{ B }	before
2	Tangling	{ K }	{ I }	before
3	Tangling	{ J }	{ (Cond2= valid) , (Cond6=valid), F }	after
4	Tangling	{ (Cond4=invalid) }	{ L }	before
5	Scattering	{ K }	{ B }	after
6	Tangling & Scattering	{ G }	{ (Cond3=valid), (Cond2=valid), (Cond5=valid), I, J }	after

Tab. 5.5. Résultats de l'application des métriques sur les aspects candidats finaux de la table 5.4

Aspect candidat	***MCC***	***MT***	***MS***
1	2	3	
2	2	2	
3	3	100	
4	2	100	
5	3		2
6	4	100	2

5.8.5 Comparaison

Dans cette section, nous allons comparer les résultats de l'application de notre approche (variante de la section 5.8) sur le code source de la figure 5.8, avec ceux obtenus en appliquant les approches [Bre 03] et [Qu 07] sur le même exemple.

Dans l'approche présentée dans la section 3.5.4 [Bre 03], l'analyse des traces d'exécution du code source a été utilisée afin de détecter la transversalité à partir des modèles récurrents. Cependant, les approches qui détectent les préoccupations transversales à partir des traces d'exécution sont incomplètes, car les traces du programme contiennent seulement un ensemble particulier des entrées du programme. Une analyse dynamique complète est impossible pour exécuter tous les chemins possibles, ce qui diminue les aspects candidats.

Les auteurs de l'approche présentée dans la section 3.5.7 [Qu 07] proposent l'identification d'aspects en utilisant les traces d'exécution, ainsi que l'arbre d'appels de méthodes, afin de dépasser les limites de l'approche dans [Bre 03].

Afin de démonter la contribution de notre approche présentée dans la section 5.8 par rapport aux méthodes similaires, nous allons procéder par comparaison des résultats. La table 5.6 présente les aspects candidats identifiés à partir du code source de la figure 5.8, en utilisant l'approche [Qu 07] et l'approche [Bre 03].

Les caractères « → » « ← » « ∇ » et « Δ » désignent respectivement les relations : *before*, *after*, *Inside-first* et *Inside-last* (présentées dans la section 3.5.4).

Tab. 5.6. Aspects candidats identifies par l'approche [Qu 07] et par l'approche [Bre 03] (extrait à partir de [Qu 07])

Résultats de [Qu 07]	**Résultats de [Bre 03]**
G→E G→H C ▽ B C ▽ D K ▽ F K ▽ L	G→E G→H C ▽ B C ▽ D

En comparant les résultats obtenus par notre approche (table 5.4) avec ceux obtenus par l'approche de [Qu 07] et l'approche de [Bre 03], nous obtiendrons ce qui suit :

- L'aspect candidat 1 de la table 5.6 n'a pas été identifié par notre approche.
- L'aspect candidat 2 de la table 5.6 a été identifié et supprimé par notre approche (détection de RIM), car la méthode G et H sont supposées appartenir à la même classe (*Class3*).
- Les aspects candidats 5 et 6 de la table 5.6 qui encapsulent la méthode K ont été identifiés par notre approche dans la table 5.4 comme l'aspect 5, tandis que les aspects 3 et 4 de la table 5.6 qui encapsulent la méthode C n'ont pas été détectés par notre approche (identifiés comme aspect dispersé et ensuite supprimé) du fait que les méthodes B, C et D sont supposées appartenir à seulement deux classes différentes.
- Le reste des aspects candidats de la table 5.4 n'ont pas pu être identifiés par les approches [Qu 07] et [Bre 03].
- Les points de coupure des aspects candidats possibles de type de relation *Inside-first* (ainsi que *Inside-last*) sont difficiles de les placer dans la refactorisation des aspects.
- Les approches [Qu 07] et [Bre 03] n'identifient pas les aspects à partir des fragments conditionnels, pourtant ces aspects reflètent bien des tâches fortement liées.

5.9 Conclusion

Au cours de ce chapitre, nous avons présenté le principe de notre approche proposée, qui vise l'identification d'aspects au niveau conceptuel, en utilisant les diagrammes de séquence. En outre, afin de montrer l'applicabilité de notre approche dans un niveau d'abstraction plus bas et dans un contexte différent, nous avons présenté une autre variante de l'approche proposée, visant l'identification d'aspects au niveau implémentation d'une manière statique.

En comparant notre approche (les deux variantes) avec les approches existantes, en utilisant les mêmes critères de comparaison utilisés dans les tables 3.1 et 4.2, nous constatons le suivant :

- **En se basant sur :** répétition ;
- **Entrée :** modèles d'interactions entre les objets (invocations de méthodes) ;
- **Sortie :** successions d'invocations de méthodes ;
- **Niveau d'abstraction :** modèles d'interactions ;
- **Symptômes détectés :** dispersion et enchevêtrement ;
- **Transversalité est :** la transversalité se produit lorsque :
  - les méthodes qui composent un RIM appartiennent au moins à deux classes différentes ;
  - une condition (attribut ou méthode) et la méthode invoquée lors de la vérification de cette condition, appartiennent à deux différentes classes ;
  - une méthode est invoquée plusieurs fois dans au moins deux classes différentes de sa classe où elle est définie.
- **Intervention humaine :** l'utilisateur peut intervenir pour choisir des aspects parmi l'ensemble des aspects candidats obtenus par notre approche, selon les résultats des métriques. Or, ce choix n'est pas nécessaire, du fait que tous les aspects obtenus doivent être refactorisés, afin d'améliorer la modularité du système logiciel, pour les raisons suivantes :
  - l'existence de chaque aspect obtenu par notre approche rend au moins deux classes liées ;
  - les aspects enchevêtrés relient deux classes ;
  - les aspects enchevêtrés de type *before*, détectés à partir des conditions, encapsulent des contraintes d'intégrité référentielle qui doivent être respectées ;
  - Chaque méthode du code advice des aspects dispersés est invoquée dans, au moins, deux classes différentes de la classe qui la définit, et les modifications

dans la manière d'utiliser cette méthode peuvent être nombreuses, coûteuses et sujettes à erreur.

Nous avons concrétisé notre approche dans des étapes claires et précises, afin de faciliter son implémentation, qui sera présentée dans le chapitre suivant.

CHAPITRE 6

IMPLÉMENTATION

« To me, ideas are worth nothing unless executed. They are just a multiplier. Execution is worth millions.»

(Steve Jobs)

6.1 Introduction

Afin de valider notre approche proposée, nous avons développé un outil qui l'implémente. Deux études de cas sont choisies : système de gestion de bibliothèque (*Library Management System)* et système de péage des autoroutes portugaises (*Portuguese Highways Toll System*). Dans ce chapitre, nous présentons notre outil, ainsi que les résultats de son application sur ces études de cas.

6.2 Schéma de l'outil

Dans le but de faciliter l'implémentation, nous utilisons le format XML, afin d'exploiter les transmissions de messages des diagrammes de séquence. A partir de la figure 6.1, les étapes d'obtention des aspects candidats par notre outil sont les suivantes :

1. A partir des fichiers XML (générés à partir des diagrammes de séquence), toutes les invocations de méthodes sont extraites, en spécifiant pour chaque méthode le contexte d'appel (sa classe et la classe qui l'invoque) ;
2. A partir de ces invocations, la matrice $MMTO$ est construite (et éventuellement $MMTO'$), ainsi que le tableau $TClass$;

3. Identification, d'une part, des aspects enchevêtrés et leurs métriques, à partir de la matrice $MMTO$ (et $MMTO'$) ; d'autre part, des aspects dispersés et leurs métriques à partir du tableau $TClass$.

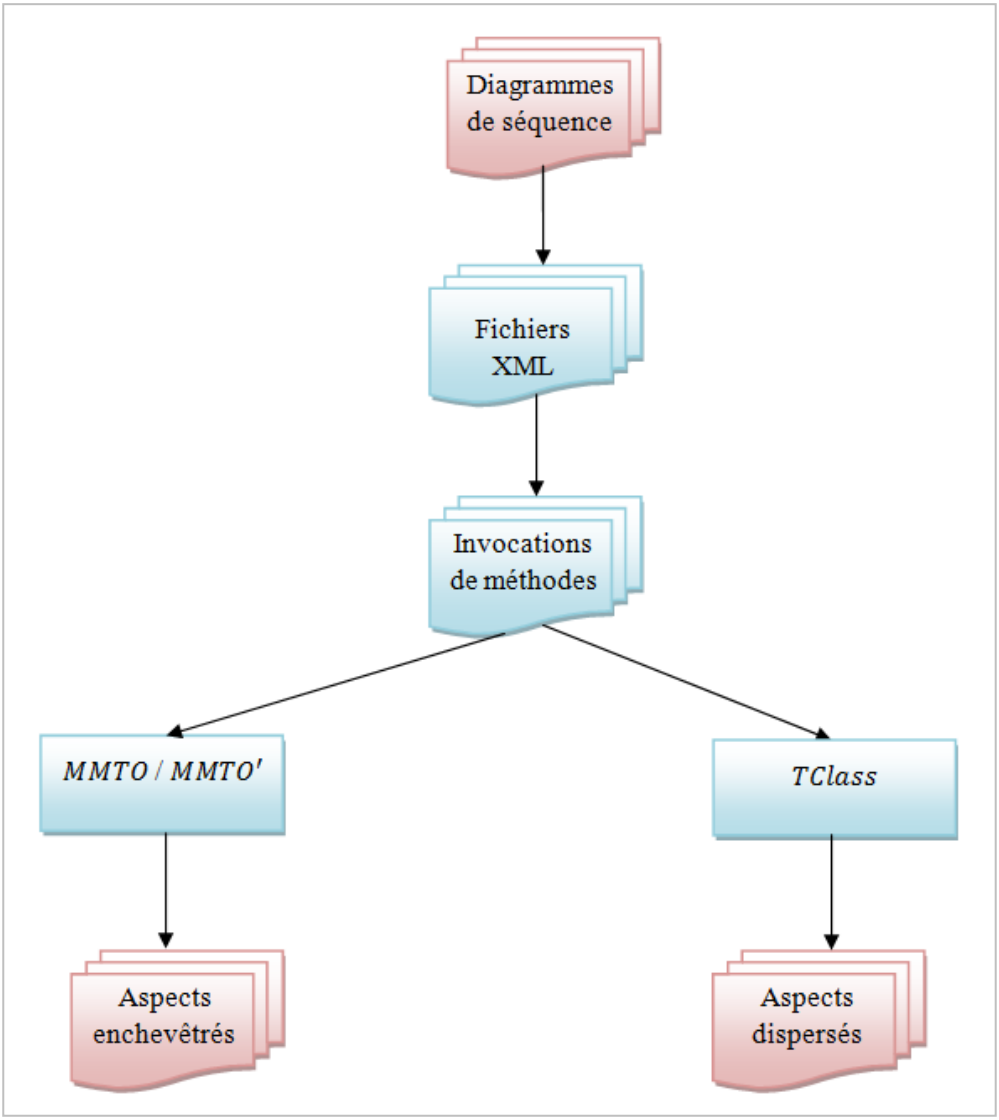


Fig. 6.1. Schéma de l'outil d'identification d'aspects

6.3 Etude de cas : Système de gestion de bibliothèque

Un système de gestion de bibliothèque est responsable des inscriptions des étudiants, la mise à jour de leurs informations, les enregistrements des nouveaux livres, ainsi que les emprunts et les remises des livres.

A partir des diagrammes conceptuels modélisant ce système (présentés dans l'annexe), la figure 6.2 présente toutes les méthodes du système de gestion de bibliothèque, obtenues par notre outil.

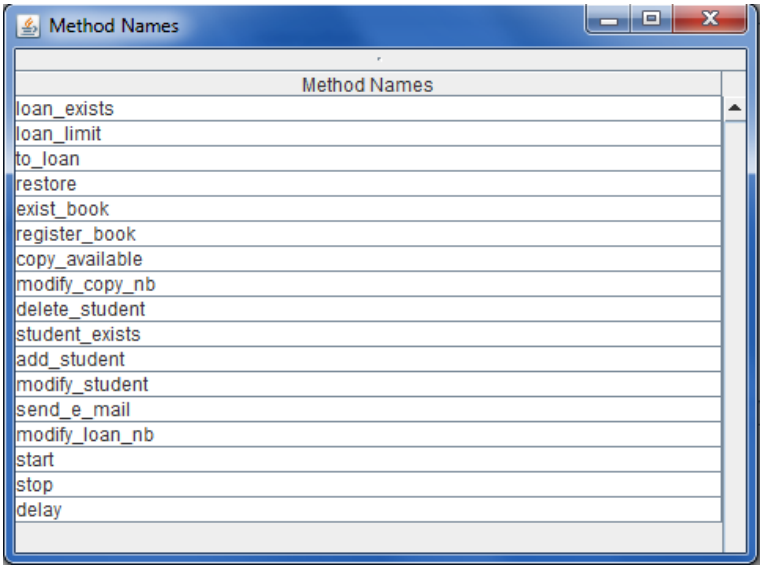


Fig. 6.2. Méthodes du système de gestion de bibliothèque

Matrix MMTO

	restore	to_loan	loan_limit	loan_exists	exist_book	register_book	modify_copy_nb	copy_available	student_exists	add_student	send_e_mail	delete_stud..	modify_student	modify_loan_nb	start	stop	delay
restore	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
to_loan	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
loan_limit	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0
loan_exists	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0
exist_book	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0
register_book	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
modify_copy_nb	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
copy_available	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
student_exists	100	0	100	100	0	0	0	0	0	100	0	0	100	0	0	0	0
add_student	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
send_e_mail	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
delete_student	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
modify_student	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
modify_loan_nb	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0
start	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
stop	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
delay	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0

Fig. 6.3. Matrice $MMTO$ du système de gestion de bibliothèque

Array TClass

restore	to_loan	loan_limit	loan_exists	exist_book	register_book	modify_copy_nb	copy_available	student_exists	add_student	send_e_mail	delete_student	modify_student	modify_loan_nb	start	stop	delay
Loan, Library	Loan, Library	Loan, Library	Loan, Library	Book, Library	Book, Library	Book_copy, Book, Loan	Book_copy, Library	Student, Library	Student, Library	Student	Student, Library	Student, Library	Student, Loan	Timer, Loan	Timer, Loan	Timer, Library

Fig. 6.4. Tableau $TClass$ du système de gestion de bibliothèque

La matrice $MMTO$ est construite dans la figure 6.3. Pour la cellule $MMTO[5,6]$ de la figure 6.3, les méthodes $exist\_book$ et $register\_book$ sont liées (pour un enregistrement d'un nouveau livre, si ce livre existe déjà, on ne doit pas l'ajouter à nouveau, mais plutôt enregistrer seulement son nouveau exemplaire), mais elles appartiennent à la même classe ($Book$). Par conséquent, ces deux méthodes ne composent pas un aspect candidat (qui n'est pas détecté par l'algorithme du listing 5.2). De même pour les cellules $MMTO[9,10]$ et $MMTO[9,13]$, où les méthodes appartiennent à la même classe ($Student$).

Le tableau $TClass$ est illustré dans la figure 6.4. A partir de ce tableau, la cellule $TClass[7]$ (méthode $modify\_copy\_nb$) contient trois classes, un aspect candidat dispersé est détecté. La table 6.1 présente la liste des aspects candidats qui résultent avant la phase de l'élimination et fusion des aspects dupliqués (section 5.7.1).

Tab. 6.1. Liste des aspects candidats du système de gestion de bibliothèque avant l'élimination et la fusion des aspects dupliqués

Aspect candidat	**Cellule de *MMTO***	**Algorithme appliqué**	**Symptôme**	**Code advice**	**Points de coupure**	**Type**	**Métriques**		
							MCC	***MS***	***MT***
1	[14,7]	Listing 5.1	Tangling	{modify_copy_nb}	{modify_loan_nb}	after	2		2
2	[4,12]	Listing 5.2	Tangling	{loan_exists}	{delete_student }	before	2		100
3	[3,8]	Listing 5.2	Tangling	{loan_limit}	{copy_available}	before	2		100
4	[8,2]	Listing 5.2	Tangling	{copy_available}	{to_loan}	before	2		100
5	[9,3]	Listing 5.2	Tangling	{student_exists}	{loan_limit}	before	2		100
6	[9,4]	Listing 5.2	Tangling	{student_exists}	{loan_exits}	before	2		100
7	[9,1]	Listing 5.2	Tangling	{student_exists}	{restore}	before	2		100
8	[18,7]	Listing 5.3	Scattering	{modify_copy_nb}	{modify_loan_nb, register_book}	after	3	2	

A partir de la table 6.1, les aspects 5, 6 et 7 sont fusionnés, selon la phase de l'élimination et fusion des aspects dupliqués (présentée dans la section 5.7.1). L'aspect résultant de cette fusion est l'aspect 5 de la figure 6.5. De la même façon, les aspects 1 et 8 (de la table 6.1) ont été fusionnés en aspect 1 de la figure 6.5.

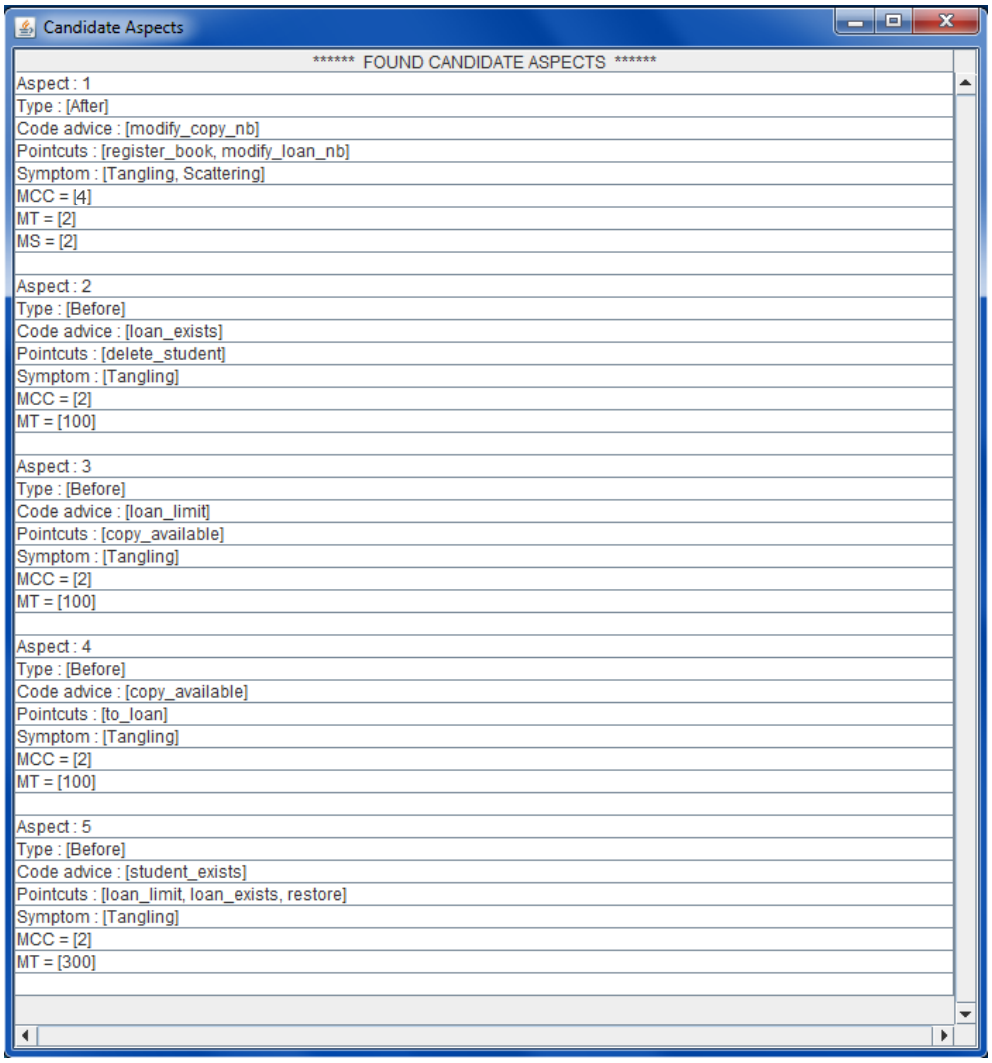


Fig. 6.5. Liste des aspects candidats du système de gestion de bibliothèque obtenus par notre outil

La figure 6.5 présente les aspects candidats obtenus par notre outil, ainsi que leurs métriques. A partir de cette figure, les aspects candidats du système de gestion de bibliothèque obtenus sont :

- ✓ Aspect 1 : Cet aspect candidat est une préoccupation de mise à jour enchevêtrée, et elle doit être refactorisée dans un aspect, car elle inclut des tâches qui sont fortement liées : le changement du nombre d'emprunt d'un livre (grâce à l'emprunt ou la remise) et le changement du nombre de ces exemplaires disponibles, ainsi que l'enregistrement du nombre des exemplaires disponibles d'un livre suite à sa nouvelle inscription.

- ✓ Aspect 2 : Les deux méthodes (*delete_student* et *loan_exists*) qui composent cet aspect candidat n'appartiennent pas à la même classe, or elles sont liées : avant de permettre une désinscription d'un étudiant, ce dernier doit remettre tous ses livres empruntés, sinon s'il a un emprunt en cours, sa désinscription est impossible.
- ✓ Aspect 3 : les méthodes *copy_available* et *loan_limit* sont liées : si un étudiant dépasse le nombre d'emprunts autorisé, il ne peut pas effectuer un nouvel emprunt, et par conséquent une copy de son livre demandé ne doit pas être recherchée.
- ✓ Aspect 4 : Pour les deux méthodes liées (*to_loan* et *copy_available*), l'étudiant ne peut pas emprunter avant qu'un exemplaire de son livre désiré soit disponible.
- ✓ Aspect 5 : Cet aspect présente une préoccupation de vérification : avant de faire une transaction pour un étudiant, il faut d'abord vérifier que cet étudiant est déjà inscrit.

En appliquant nos métriques (proposées dans la section 5.7.2), nous constatons que chaque aspect obtenu dans la figure 6.5 relie au moins deux classes (métrique $MCC \geq 2$). En outre, les aspects candidats 2, 3 et 4 ont une valeur élevée d'enchevêtrement (métrique $MT = 100$), indiquant l'existence des contraintes d'intégrité référentielle qui doivent être respectées. L'aspect candidat 1 a les métrique *MT* et *MS* supérieures ou égales à 2. Par conséquent, tous ces aspects candidats obtenus par notre outil méritent d'être refactorisés dans des aspects. En outre, tous les aspects obtenus respectent bien les contraintes du système de la gestion de bibliothèque étudié.

6.4 Etude de cas : Système de péage des autoroutes portugaises

La deuxième étude de cas que nous avons choisie est une version simplifiée du système de péage des autoroutes portugaises (*Portuguese Highways Toll system*) [Cla 97].

"Dans un système de tarification de la circulation, les conducteurs de véhicules autorisés avancent devant les barrières de péage automatique. Les ports sont placées sur des voies spéciales appelées voies vertes. Un conducteur doit installer un dispositif dans son véhicule appelé *gizmo*. L'enregistrement des véhicules autorisés inclut les données personnelles du propriétaire, son numéro de compte bancaire et les détails du véhicule. Le

gizmo est donné au client pour être activé en utilisant un guichet automatique (*ATM*) qui informe le système lors de l'activation du gizmo. Un gizmo est lu par les capteurs de barrière du péage. Les informations lues sont stockées par le système, et utilisées pour débiter les comptes correspondants. Lorsqu'un véhicule autorisé traverse une voie verte, le feu vert est allumé, et le montant étant débité est affiché. Si un véhicule non autorisé passe, un feu jaune est allumé, et une caméra prend une photo de la plaque d'immatriculation (pour trouver le propriétaire du véhicule). Il existe trois types de barrière de péage : péage unique où le même type du véhicule paye un montant fixe, péage d'entrée pour entrer dans un péage et péage de sortie pour le quitter. Le montant payé sur les autoroutes dépend du véhicule et de la distance parcourue. " [Cla 97]

Selon les auteurs dans [Mor 02], il existe cinq cas d'utilisations : l'enregistrement du véhicule, l'entrée à l'autoroute, la sortie de l'autoroute, le péage unique et le paiement de factures. Les classes du système sont : *System*, *Vehicle*, *Account*, *Gizmo*, *Light*, *Amount*, *Camera*, *Entrance*, *ATM* et *Billing* . Leurs méthodes sont illustrées dans la figure 6.6.

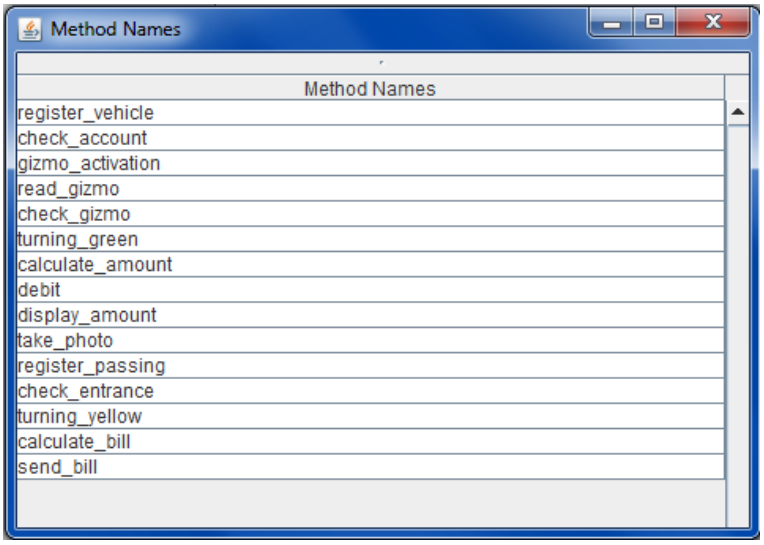
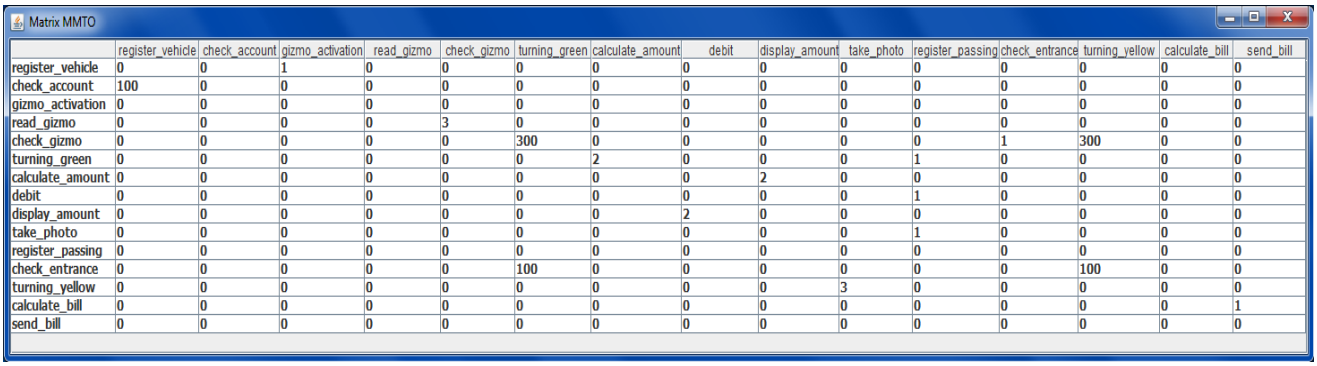
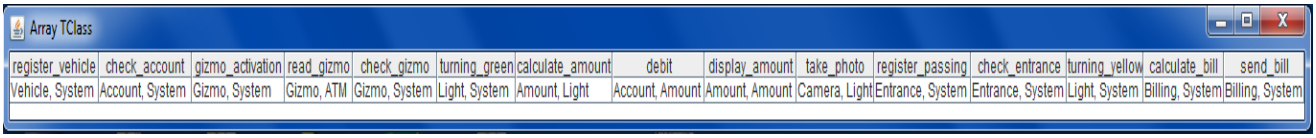


Fig. 6.6. Méthodes du système de péage des autoroutes portugaises


Matrix MMTO

	register_vehicle	check_account	gizmo_activation	read_gizmo	check_gizmo	turning_green	calculate_amount	debit	display_amount	take_photo	register_passing	check_entrance	turning_yellow	calculate_bill	send_bill
register_vehicle	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
check_account	100	0	0	0	0	0	0	0	0	0	0	0	0	0	0
gizmo_activation	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
read_gizmo	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0
check_gizmo	0	0	0	0	0	300	0	0	0	0	0	1	300	0	0
turning_green	0	0	0	0	0	0	2	0	0	0	1	0	0	0	0
calculate_amount	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0
debit	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
display_amount	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0
take_photo	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
register_passing	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
check_entrance	0	0	0	0	0	100	0	0	0	0	0	0	100	0	0
turning_yellow	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0
calculate_bill	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
send_bill	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 6.7. Matrice $MMTO$ du système de péage des autoroutes portugaises


Array TClass

register_vehicle	check_account	gizmo_activation	read_gizmo	check_gizmo	turning_green	calculate_amount	debit	display_amount	take_photo	register_passing	check_entrance	turning_yellow	calculate_bill	send_bill
Vehicle, System	Account, System	Gizmo, System	Gizmo, ATM	Gizmo, System	Light, System	Amount, Light	Account, Amount	Amount, Amount	Camera, Light	Entrance, System	Entrance, System	Light, System	Billing, System	Billing, System

Fig. 6.8. Tableau $TClass$ du système de péage des autoroutes portugaises

Tab. 6.2. Liste des aspects candidats du système de péage des autoroutes portugaises avant l'élimination et la fusion des aspects dupliqués

Aspect candidat	**Cellule de *MMTO***	**Symptôme**	**Code advice**	**Point de coupure**	**Type**	**Métriques**		
						MCC	***MS***	***MT***
1	[2,1]	Tangling	{check_account}	{register_vehicle}	before	2		100
2	[5,6]	Tangling	{check_gizmo}	{turning_green}	before	2		300
3	[5,13]	Tangling	{check_gizmo}	{turning_yellow}	before	2		300
4	[13,10]	Tangling	{turning_yellow}	{take_photo}	before	2		3
5	[6,7]	Tangling	{turning_green}	{calculate_amount}	before	2		2
6	[12,6]	Tangling	{check_entrance}	{turning_green}	before	2		100
7	[12,13]	Tangling	{check_entrance}	{turning_yellow}	before	2		100
8	[9,8]	Tangling	{display_amount}	{debit}	before	2		2

A partir des diagrammes de séquence modélisant le système de péage des autoroutes portugaises, la matrice $MMTO$ est construite dans la figure 6.7. Le tableau $TClass$ est illustré dans la figure 6.8. A partir de ce tableau, nous pouvons constater qu'aucune méthode n'est dispersée. La table 6.2 présente la liste des aspects candidats qui résultent avant la phase de l'élimination et fusion des résultats aspects (section 5.7.1). A partir de cette table, les aspects 2, 3, 6 et 7 sont fusionnés, selon la phase de l'élimination et fusion des aspects dupliqués (présentée dans la section 5.7.1). L'aspect résultant de cette fusion est l'aspect 2 de la figure 6.9.

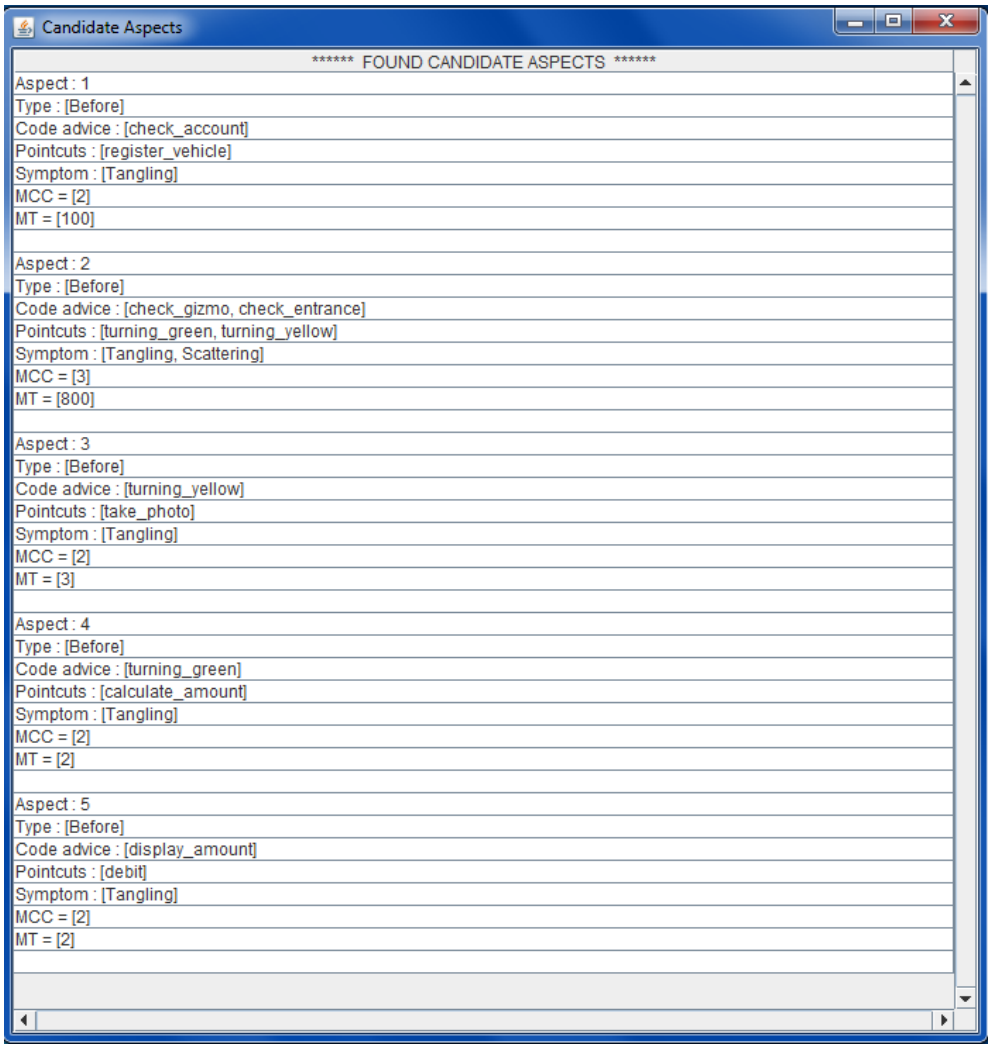


Fig. 6.9. Résultats de l'application de notre approche sur le système de péage des autoroutes portugaises

La figure 6.9 présente les aspects candidats obtenus par notre outil, ainsi que les résultats de l'application de nos métriques. A partir de cette figure, nous pouvons constater :

- ✓ Aspect 1: avant d'enregistrer le véhicule, il faut d'abord vérifier si le compte de son propriétaire est bien activé.
- ✓ Aspect 2: avant d'allumer le feu vert ou jaune, il faut d'abord vérifier si le gizmo est bien activé, et si l'entrée est enregistrée.
- ✓ Aspect 3: la photo est prise seulement lorsque le feu jaune est allumé.
- ✓ Aspect 4: le montant est calculé lorsque le feu vert est allumé.

- ✓ Aspect 5: le montant à débiter est affiché puis débité.

Les aspects identifiés par notre approche prennent en compte les contraintes de l'étude de cas (système de péage des autoroutes portugaises), et améliorent la modularité du système logiciel. Concernant les aspects 1 et 2, ils encapsulent des contraints d'intégrité référentielle, et relient au moins deux différentes classes. Les aspects 3, 4 et 5 présentent des tâches successives et liées, et relient, par conséquent, deux différentes classes.

Cette étude de cas a été utilisée dans d'autres travaux, afin d'illustrer l'identification des préoccupations transversales. Par exemple, dans [Mor 02], les auteurs commencent par l'identification des exigences du système, et la sélection, parmi ces exigences, les attributs de qualité, en rapport avec le domaine d'application et les intervenants. Les exigences fonctionnelles sont spécifiées avec les cas d'utilisation, et les attributs de qualité sont décrits en utilisant des modèles (*templates*) spéciales. Les attributs de qualité peuvent être des hypothèses, contraintes ou buts des intervenants. Ensuite, les attributs de qualité qui entrecoupent les exigences fonctionnelles sont identifiés. Par exemple, si le propriétaire du véhicule doit indiquer, durant son enregistrement, ses détails de banque, alors la sécurité est une contrainte que le système doit respecter. Les autres attributs de qualité sont identifiés d'une façon similaire : temps de réponse, système multi-utilisateur, compatibilité, questions juridiques, exactitude et disponibilité. Si un attribut de qualité affecte plusieurs cas d'utilisation, alors il est transversal (aspect).

Dans [Ras 03], les préoccupations sont d'abord identifiées en analysant les points de vue (exigences) des intervenants. Par exemple, comme le propriétaire du véhicule doit indiquer ses détails de la banque durant l'enregistrement, alors la préoccupation « sécurité » est identifiée. De la même façon, les autres préoccupations sont identifiées : temps de réponse, multi-accès au système, compatibilité, questions juridiques, exactitude et disponibilité. Les préoccupations qui entrecoupent les exigences des intervenants (points de vue) sont considérées comme aspects candidats. Dans ce cas, toutes les préoccupations identifiées sont transversales.

Dans [Con 13], d'abord les mots clé du système sont spécifiés en utilisant le catalogue des documents des exigences. Les préoccupations non fonctionnelles sont ensuite identifiées, en analysant les mots clé. Les diagrammes de cas d'utilisation sont utilisés

pour représenter les exigences des intervenants, et la dépendance entre les préoccupations non fonctionnelles et les artefacts des exigences sont identifiées, en comparant les mots du catalogue avec le nom des différents artefacts des exigences, en utilisant l'analyse syntaxique. Par exemple, dans le cas d'utilisation « vérifier le véhicule autorisé », le mot « autorisé » est défini, avec la préoccupation « sécurité », et par conséquent, une relation de dépendance entre « vérifier le véhicule autorisé » et « sécurité » est identifiée. Dans cette approche, toutes les préoccupations non fonctionnelles sont considérées comme transversales (aspects) : sécurité, persistance, représentation des données, notification d'événements.

Comme nous pouvons le constater par ces trois approches, les aspects détectés sont généraux, initialement identifiés à partir des connaissances du domaine, et ils peuvent ne pas apparaitre dans l'implémentation du système comme attendu.

6.5 Conclusion

Dans ce chapitre, nous avons démontré la facilité d'implémenter notre approche, ainsi que son applicabilité. Nous avons utilisé deux différentes études de cas. Les résultats obtenues, avec l'utilisation des métriques, sont satisfaisants, et respectent les contraintes imposées par les systèmes étudiées.

CHAPITRE 7

CONCLUSION GÉNÉRALE ET PERSPECTIVES

« At that time I had a dream. I dreamt that decomposing the whole system into blocks with clearly defined signal interfaces would be an accepted technology…I still have another dream. I dream that we will get a component marketplace where different players can work. Some will play the role of selling components; others will buy components.»

(Ivar Jacobson)

Nous assistons, depuis fort longtemps, à une évolution vers des systèmes logiciels de plus en plus complexes. Les paradigmes de programmation ont pour objectif principal de faciliter l'implémentation de ces systèmes. La programmation orientée objet est le paradigme de choix pour plusieurs projets logiciels. Elle a apporté une meilleure encapsulation et une meilleure décomposition, en unissant les données et les traitements. En outre, elle est efficace dans l'expression des fonctionnalités dites verticales, fonctionnalités exprimant les aspects métiers du système. Inversement, elle s'avère particulièrement limitée dans le cas de l'existence des préoccupations transversales, du fait qu'elle ne permet pas une encapsulation propre à ces préoccupations transversales. Ceci a rendu la compréhension, la maintenance, l'évolution et la réutilisation des systèmes logiciels plus difficiles.

Pour remédier à ces insuffisances, le paradigme orienté aspect est apparu afin de supporter la modularité améliorée des systèmes logiciels. Il complète la modularité

orientée objet par l'encapsulation des préoccupations transversales dans des aspects séparément.

L'émergence du paradigme orienté aspect a nécessité ensuite l'apparition d'un nouveau domaine : identification d'aspects. Nous distinguons l'identification d'aspects selon deux perspectives : la rétro-ingénierie (qui concerne la maintenance du logiciel et la migration vers le paradigme orienté aspect), et le développement orienté aspect.

Cependant, la plupart des approches d'identification d'aspects souffrent de nombreuses limites, tel que le manque d'une définition précise d'aspect, ce qui empêche les préoccupations transversales identifiées d'être toutes refactorisées dans des aspects, outre le manque de détection simultanée des deux symptômes de la transversalité : enchevêtrement et dispersion.

Ainsi, identifier les aspects à l'implémentation a l'avantage d'utiliser la version finale du système logiciel. Or, cette identification est difficile, car les aspects sont fortement intégrés avec le code métier, et ce dernier est lié à la technologie d'implémentation. En outre, identifier les aspects durant des étapes du cycle de développement du logiciel (avant l'implémentation) a l'avantage de gérer la transversalité tôt, et par conséquent de bénéficier d'un développement orienté aspect. Cependant, cette identification présente le risque que le système logiciel n'est pas dans sa version finale (des aspects peuvent disparaitre et d'autres peuvent émerger).

Dans cette thèse, nous avons proposé une nouvelle approche d'identification d'aspects au niveau des systèmes logiciels orientés objet, qui exploite les relations d'interactions internes des éléments du système, car ces relations véhiculent une information utile. Ces relations sont disponibles au niveau conceptuel, représentées par les diagrammes d'interactions d'objets, ou au niveau implémentation représentées par les relations d'invocations de méthodes.

Nous avons utilisé un niveau plus abstrait que celui du code source des systèmes existants (*legacy systems*), ce qui permet de rendre l'approche d'identification d'aspects plus générale et indépendante de la technologie d'implémentation, et un niveau plus proche de la version finale du système logiciel, ce qui permet d'avoir l'ensemble des fonctionnalités du système.

Notre approche proposée identifie les aspects, en utilisant les diagrammes UML de séquence. Elle est basée sur les interactions (transmissions de messages) entre les différents objets du diagramme de séquence qui reflètent les invocations de méthodes, ainsi que l'ordre chronologique des transmissions de messages qui reflète l'ordre d'exécution des tâches du système.

Au niveau des diagrammes de séquence, nous avons exploité les transmissions répétées et dispersées des messages afin de détecter les préoccupations dispersées, ainsi que l'ordre chronologique des transmissions des messages pour détecter celles qui sont enchevêtrées.

7.1 Résumé des contributions

Le travail réalisé dans cette thèse nous a permis d'apporter des contributions. Il s'agit de proposer une nouvelle approche d'identification d'aspects, qui apporte plusieurs avantages par rapport aux travaux existants dans le domaine d'identification d'aspects :

- Exploiter les deux symptômes de la transversalité à la fois : enchevêtrement et dispersion ;
- Unifier le processus d'identification d'aspects pour que l'approche soit applicable, aussi bien sur le code source que sur les modèles ;
- L'approche proposée utilise les diagrammes conceptuels de séquence qui sont détaillés et finaux. Par conséquent, ils reflètent directement le code source du système logiciel. A partir du diagramme de séquence, les entités de base analysées par notre approche sont les méthodes et leurs invocations. Ces entités resteront au niveau implémentation. Pour ces raisons, les aspects détectés par notre approche apparaissent dans l'implémentation comme attendus.
- Notre identification d'aspects est plus générale, indépendante de la technologie d'implémentation, de la syntaxe du programmeur, ou des interviews avec les intervenants.
- Nous fournissons des aspects candidats prêts à la refactorisation, en les spécifiant en termes de points de coupures et de code advice.

- Afin d'avoir une évaluation de l'approche proposée, et de détecter les erreurs possibles dans le processus d'identification d'aspects (aspects candidats faux), nous avons proposé trois nouvelles métriques. Celles-ci aident également l'utilisateur de choisir les aspects candidats les plus pertinents.
- Outre les métriques (pour valider notre approche proposée), nous avons développé un outil qui implémente notre approche. Cet outil a été appliqué sur des études de cas. Les résultats ont été satisfaisants.

7.2 Perspectives de recherche

Notre travail ouvre trois perspectives :

- La première concerne la généralisation de notre approche, autrement dit, son applicabilité pour d'autres types d'artefacts, outre les méthodes et leurs invocations ;
- La seconde cherche l'existence éventuelle d'autres symptômes de la transversalité ;
- La troisième perspective concerne l'amélioration de la structure de données (matrice et graphes), utilisée par l'approche proposée, afin de rendre cette dernière plus optimale.

RÉFÉRENCES

[Abd 10] Abdelzad, V., Aliee, F.S., A Method Based on Petri Nets for Identification of Aspects, In Proc. of Workshop on Early Aspects in AOSD (2010)

[Ald 01] Aldawud, O., Elrad, T., Bader, A., A UML Profile for Aspect Oriented Modeling," in Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA), Tampa Bay, Florida, USA (2001)

[Ald 03] Aldawud, O., Bader, A., Elrad, T., UML Profile for Aspect Oriented Software Development, presented at Workshop on Aspect-Oriented Modelling with UML, Boston, Massachusetts, USA (2003)

[All 97] Allen, R., Garlan, D. A formal basis for architectural connection, ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 6, pp. 213-249 (1997)

[Ami 08] Amirat, A., Laskri, M., Khammaci, T., Modularisation of crosscutting concerns in requirements engineering, The international arab journal of information technologie, Vol. 5, No. 2 (2008)

[Ape 06] Apel, S., Leich, T., Saake, G., Aspectual mixin layers: aspects and features in concert, 28th International Conference on Software Engineering (ICSE), Shanghai, pp. 122-131, China (2006)

[Ara 02] Araujo, J., Moreira, A., Brito, I., Rashid, A., Aspect-Oriented Requirements with UML, Workshop on Aspect-Oriented Modelling with UML at 5th International Conference on Unified Modelling Language, Dresden, Germany (2002)

[Ara 03] Araújo, J., Moreira, A., An Aspectual Use Case Driven Approach, Proc. 8th Jornadas de Ingenieria de Software y Base de Datos, Spain (2003)

[Aré 05] Arévalo, G., Ducasse, S., Nierstrasz, O., Lessons Learned in Applying Formal Concept Analysis to Reverse Engineering, conference ICACF (2005)

[Bak 95] Baker, B.S., On finding duplication and near-duplication in large software systems, second Working Conference on Reverse Engineering (WCRE), Toronto, pp. 86-95, Canada (1995)

[Ban 04a] Baniassad, E., Clarke, S., Theme: An Approach for Aspect-Oriented Analysis and Design, 26th International Conference on Software Engineering (ICSE), pp. 158-167, Edinburgh, Scotland (2004)

[Ban 04b] Baniassad E., Clarke, S., Finding Aspects in Requirements with Theme/Doc, presented at Workshop on Early Aspects, Lancaster, UK (2004)

[Ban 05] Baniassad, E., Clarke, S., Aspect-Oriented Analysis and Design: The Theme Approach, Addison-Wesley, Reading (2005)

[Ban 06] Baniassad, E., Clements, P., Araújo, P., Moreira, A., Rashid, A., Tekinerdogan, B., Discovering early aspects, IEEE Software 23(1), pp. 61-70 (2006)

[Bar 02] Barros, J.P., Gomes, L., Activities as Behaviour Aspects, presented at Workshop on Aspect-oriented Modelling, Dresden, Germany (2002)

[Bas 98] Bass, L. Clements, P. Kazman, R., Software Architecture in Practice, Addison-Wesley (1998)

[Bau 11] Project Bauhaus, http://www.bauhaus-stuttgart.de/, 03/05/2011

[Bax 98] Baxter, I.D., Yahin, A., Moura, L., Anna, M.S., Bier, L., Clone Detection Using Abstract Syntax Trees, International Conference on Software Maintenance (ICSM), pp. 368-377, Maryland, USA (1998)

[Bel 12] Belmabrouk, A., messabih, B., The Reverse Engineering in Oriented Aspect: Detection of semantics clones, International Journal of Scientific and Engineering Research, Vol. 3, issue 5 (2012)

[Bel 76] L. Belady, L. Lehman, A model of large program development, IBM Systems Journal, Vol. 15, issue 3, pp 225–252 (1976)

[Ber 01] Bergmans, L., Aksit, M., Composing Crosscutting Concerns Using Composition Filters, Communications of the ACM, Vol. 44, No. 10, pp. 51-57 (2001)

[Ber 05a] Berg, K. van den, Conejero, J.M., A Conceptual Formalization of Crosscutting in AOSD, in Desarrollo de Software Orientado a Aspectos (DSOA), Granada, Spain (2005)

[Ber 05b] Berg, K. van den, Conejero, J., Chitchyan, R., AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation, AOSD Europe Network of Excellence (2005)

[Ber 09] Bernardi, M.L., Lucca, G.A., Analysing Object Type Hierarchies to Identify Crosscutting Concerns, Springer-Verlag Berlin Heidelberg, pp. 216–22 (2009)

[Ber 11] Bernardi, M.L., Lucca, G.A., Identifying the Crosscutting among Concerns by Methods' Calls Analysis, Vol. 257, pp.147-158, Springer (2011)

[Bou 06] Bounour, N., Ghoul. S., Atil, F. A, comparative classification of aspect mining approaches, Journal of Computer Science, Vol. 2, No. 4, ISSN 1549-3636, pp. 322-325 (2006)

[Bou 07] N. Bounour, Une approche dirigée par les modèles pour l'Aspect Mining, Thèse de doctorat d'état en informatique, Université de Badji Mokhtar Annaba (2007)

[Bre 03] Breu, S., Jens, J., Aspect mining using dynamic analysis, In GI-Software technik-Trends, Mitteilungen der Gesellschaft f¨ur Informatik, Vol. 23, Bad Honnef, Germany, pp. 21–22 (2003)

[Bre 04] Breu, S., Krinke, J. Aspect Mining Using Event Traces, 19th International Conference on Automated Software Engineering (ASE), IEEE Computer Society, Linz, Austria, pp. 310-315 (2004)

[Bri 02] Brito, I., Moreira, A., Araújo, J., A requirements model for quality attributes, Proc. Aspect-Oriented Requirements Engineering and Architecture Design, Amsterdam (2002)

[Bri 08] Brito, I., Aspect-Oriented Requirements Analysis, PhD Thesis (2008)

[Bru 05] Bruntink, M., Deursen, A., Engelen, R., Tourwé, T., On the Use of Clone Detection for Identifying Crosscutting Concern Code, IEEE Transactions of Software Engineering, 31(10), pp. 804-818 (2005)

[Bus 08] Busyairah, S., Kasirun, Z., Crosscutting Concern Identification at Requirements Level, Malaysian Journal of Computer Science, ISSN 0127-9084, Vol. 21, No. 2(2008)

[Cec 04] Ceccato, M., Tonella, P., Measuring the Effects of Software Aspectization, First Workshop on Aspect Reverse Engineering, Delft University of Technology, the Netherlands (2004)

[Cec 06] Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwe, T., Applying and combining three different aspect Mining Techniques, Software Quality Journal, Vol. 14, Issue 3, pp. 209-231 (2006)

[Cec 07] Ceccato, M., Migrating Object Oriented code to Aspect Oriented Programming, IEEE International Conference on Software Maintenance ICSM, Paris (2007)

[Chi 05] Chitchyan, R., Rashid, A., Sawyer, P., and al., Survey of Analysis and Design Approaches. AOSD Europe Network of Excellence (2005)

[Chu 00] Chung, L., Nixon, B., Yu, E., Mylopoulos, J., Non-Functional Requirements in Software Engineering, 0-7923-8666-3, Kluwer Academic Publishers (2000)

[Cla 05] Clarke, S., Walker, R. J., Generic Aspect-Oriented Design with Theme/UML, in Aspect-Oriented Software Development: Addison-Wesley (2005)

[Cla 97] Clark, R., Moreira, A., Constructing Formal Specifications from Informal Requirements, In: Software Technology and Engineering Practice, pp. 68–75. IEEE Computer Society Press, Los Alamitos (1997)

[Cla 99] Clarke, S., Harrison, W., Ossher, H., Tarr, P., Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code, presented at Proc. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Denver, Colorado, USA (1999)

[Coj 09] Cojocar, G. S., Czibula, G., Czibula, I. G., A Comparative Analysis of Clustering Algorithms in Aspect Mining, Studia Universitatis Babes-Bolyai, Informatica, Issue 1, pp. 75-84. 10p. 1 Chart (2009)

[Con 09] Conejero, J., Figueiredo, E., Garcia, A., Hernández, J., Jurado, E., Early cross-cutting metrics as predictors of software instability, in: Proc. of the 47th International Conference Objects, Models, Components, Patterns, TOOLS Europe, LNBIP 33, Zurich, Switzerland, pp. 136-156 (2009)

[Con 10] Conejero, J.M., Hernándeza, J., Jurado. E., Berg, K., Mining early aspects based on syntactical and dependency analyses, journal: Science of Computer Programming, Vol 75, pp. 1113-1141 (2010)

[Con 13] Conejero, J.M., Hernándeza, J., Jurado. E., Berg, K., Early Aspect Mining in the Portuguese Highways Toll System, http://www.google.fr/url?sa=t&rct=j&q=early%20aspect%20mining%20in%20the%20portuguese%20highways%20toll&source=web&cd=1&ved=0CDMQFjAA&url=http%3A%2F%2Fquercusseg.unex.es%2Fchemacm%2Fresearch%2Fanalysisofcrosscutting%2F%3Fdownload%3DPortugueseHighwaysTollSystem.pdf&ei=ov2fUY7FF8iw0AXehYCADg&usg=AFQjCNGA-ZK9DW_LwjZ6W_--r9m0ckZJJQ , 21/05/2013

[Cza 00] Czarnecki, K., Eisenecker, U., Generative Programming: Methods, Tools and Applications: Addison-Wesley (2000)

[Dah 11a] Dahi, F., Bounour, N. Identification d'aspects par l'analyse des concepts formels, 1st International Conference on Information Systems and Technologies ICIST'11, Tebessa, Algeria (2011)

[Dah 11b] Dahi, F., Bounour, N., Etude critique des approches de découverte d'aspect à travers le cycle du développement de logiciels', The Second International Conference on Complex Systems CISC'11, Jijel, Algeria (2011)

[Dah 12] Dahi, F., Bounour, N., Détection des préoccupations transversales au niveau architectural, 11th African conference on research in computer science and applied mathematics CARI'12, Algiers, Algeria (2012)

[Dah 72] Dahl, O. J., Dijkstra, E. W., Hoare, C. A. R., Structured programming, book, ISBN: 0-12-200550-3 (1972)

[Dar 11] Darwin Architecture Description Language, http://www.doc.ic.ac.uk/~igeozg/Project/Darwin/, 12/10/2011

[Dar 93] Dardenne, A., Van Lamsweerde, A., Fickas, S., Goal-Directed Requirements Acquisition, Science of Computer Programming Journal, ACM Press. 20(1-2): pp. 3-50 (1993)

[Dem 87] Law Of Demeter, http://c2.com/cgi/wiki?LawOfDemeter, 30/05/2014

[Deu 03] Deursen, A., Marin, M., Moonen, L., Aspect Mining and Refactoring, In Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE), University of Waterloo (2003)

[Deu 99] Deursen, A., Kuipers, T., Identifying Objects using Cluster and Concept Analysis, 21ème conférence internationale sur l'ingénierie logicielle CSE99, Californie (1999)

[Dij 76] Dijkstra, E., A Discipline of Programming. Prentice Hall (1976)

[Dob 02] Dobrica, L., Niemel, E., A survey on software architecture analysis methods, IEEE Trans. Softw. Eng, vol. 28, pp. 638- 653 (2002)

[Dua 07] Duan, C., Cleland-Huang, J. A., Clustering Technique for Early Detection of Dominant and Recessive Cross-Cutting Concerns, In Proceedings of the Early Aspects at Workshops in Aspect-Oriented Requirements Engineering and Architecture Design, USA (2007)

[Duc 99] Ducasse, S., Rieger, M., Demeyer, S., A Language Independent Approach for Detecting Duplicated Code, 15th International Conference on Software Maintenance (ICSM), IEEE, pp. 109-118 (1999)

[Ead 08a] Eaddy, M., An Empirical Assessment of the Crosscutting Concern Problem, Ph.D. Dissertation, Columbia University (2008)

[Ead 08b] Eaddy, M., Zimmermann, T., Sherwood, K., Garg, V., Murphy, G., Nagappan, N., Aho, A., Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering,Volume 34, Issue 4 , pp. 497-515 (2008)

[Eri 95] Erich, G.,Helm, R., Johnson, R.,Vlissides, J.,Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, ISBN 0-201-63361-2 (1995)

[Fer 05] Fernandes, L., An Aspect-Oriented Approach to Model Requirements, In 13th Requirements Engineering Conference (RE'05), Doctoral Consortium, Paris, France (2005)

[Fil 05] Filman, R., Elrad, T., Clarke, S., Aksit, M. Aspect-Oriented Software Development, Addison-Wesley (2005)

[Fil 12] Fillus, E.K., Vergilio, S.R., A Clustering Based Approach for Aspect Mining and Pointcut Identification, 6th Latin American Workshop on Aspect Oriented Software Development Advanced Modularization Techniques, Brazil (2012)

[Fin 96] Finkelstein, A., Sommerville, I., The Viewpoints FAQ, Software Engineering Journal: Special Issue on Viewpoints for Software Engineering, IEE/BCS. 11(1): 2-4 (1996)

[Fox 05] Fox, J., A Formal Foundation for Aspect Oriented Software Development. 14th Congreso Internacional de Computación CIC, pp. 434-444, ISBN 970-36-0266-5, Mexico (2005)

[Gal 01] Gal, A., Schröder-Preikschat, W., Spinczyk,O., AspectC++: Language Proposal and Prototype Implementation, German Research Council (DFG), grant no. SCHR 603/1-1 and SCHR 603/2 (2001)

[Gan 99] Ganter, B., Wille, R., Formal Concept Analysis: Mathematical Foundations, Springer-Verlag (1999)

[Gar 00] Garlan, D. Monroe, R. T., Wile, D., Acme: Architectural Description of Component-Based Systems," in Foundations of Component-Based Systems, M. S. Gary T. Leavens, Ed.: Cambridge University Press, pp. 47-68 (2000)

[Gar 04] Garcia, A., From Objects to Agents: An Aspect-Oriented Approach, vol. PhD Rio de Janeiro, Brazil: PUC-Rio (2004)

[Gar 94] Garlan, D., Allen, R., Ockerbloom, J., Exploiting style in architectural design environments, presented at Symposium on Foundations of Software Engineering, New Orleans, Louisiana, United States (1994)

[God 95] Godin, R., Mineau, G., Missaoui, R., Mili, H., Méthodes de classification conceptuelle basées sur les treillis de Galois et applications, Revue d'Intelligence Artificielle, pp. 105-137 (1995)

[Goe 99] Goebel, M., Gruenwald, L., A Survey of Data Mining and Knowledge Discovery Software Tools, SIGKDD Explorations, Vol. 1, Issue 1, pp. 20-33 (1999)

[Gra 03] Gradecki, J.D., Lesiecki, N., AspectJ: Aspect-Oriented Programming in Java, John Wiley and Sons (2003)

[Gru 99] Grundy, J., Aspect-Oriented Requirements Engineering for Component-based Software Systems, 4th IEEE International Symposium on Requirements Engineering, Limerick, Ireland (1999)

[Han 01] Hannemann, J. and Kiczales, G. Overcoming the Prevalent Decomposition in Legacy Code, Workshop on Advanced Separations of Concerns at 23rd International Conference on Software engineering (ICSE), Toronto, Canada (2001)

[He 06] He, L., Bai, H. Aspect Mining Using Clustering and Association Rule Method, International Journal of Computer Science and Network Security, Vol. 6, No. 2, pp. 247-251 (2006)

[Her 02] Herrmann, S., Composable Designs with UFA, presented at Workshop on Aspect-oriented Modelling (held with AOSD 2002), Enschede, The Netherlands (2002)

[Hil 07] Hill, E., Pollock, L., Vijay-Shanker, K., Exploring the Neighborhood with Dora to Expedite Software Maintenance, in Automated Software Engineering (ASE), Atlanta, Georgia, USA, pp. 14-23 (2007)

[Hür 95] Hürsch, W., Lopes, C., Separation of Concerns, Technical report by the College of Computer Science, Northeastern University (1995)

[Hyp 12] MDSOC: Software Engineering Using Hyperspaces, IBM Research, http://www.research.ibm.com/hyperspace/, 22/12/2012

[Jac 92] Jacobson, I., Chirsterson, M., Jonsson, P., Overgaard, G., Object-Oriented Software Engineering - a Use Case Driven Approach, 978-0201544350, Addison-Wesley (1992)

[Jac 99] Jacobson, I. Booch, G., Rumbaugh, J., The Unified Software Development Process: Addison-Wesley (1999)

[Jac 14] JAC Project, http://jac.ow2.org/, 30/05/2014

[Jai 88] Jain, A.K., Dubes, R.C., Algorithms for Clustering Data, Englewood Cliffs, N.J.: Prentice Hall (1988)

[Joh 93] Johnson, J., Identifying Redundancy in Source Code Using Fingerprints, IBM Centre for Advanced Studies Conference, pp. 171-183 (1993)

[Kam 02] Kamiya, T., Kusumoto, S., Inoue, K., CCFinder: a multilinguistic token-based code clone detection system for large scale source code, IEEE Transsactions of Software Engineering, vol. 28, no 7, pp. 654 - 670 (2002)

[Kas 06] Kassab, M., Ormandjieva, O., Towards an Aspect-Oriented Software Development Model with Tractability Mechanism, Proc. Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Germany (2006)

[Kaz 00] Kazman, R. Klein, M., Clements, P., ATAM: Method Architecture Evaluation, Technical Report CMU/SEI-2000-TR-004 ESC-TR-2000-004 (2000)

[Kel 06] Kellens, A., Aspect Mining, An Introduction, Université Brussel, Belgium (2006)

[Kel 07] Kellens, A., Mens, K., Tonella, P., A survey of automated code-level aspect mining techniques, Transactions on Aspect-Oriented Software Development IV, LNCS Vol. 4640, pp. 143-162 (2007)

[Kic 01a] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G. An overview of AspectJ. 15th European Conference on Object-Oriented Programming (ECOOP), LNCS 2072, Springer, pp. 327-353 (2001)

[Kic 01b] Kiczales, G. and Hilsdale, E. Aspect-oriented programming, Foundations of Software Engineering, Tutorial (2001)

[Kic 97] Kiczale, G., Aspect oriented programming, European Conference on Object-Oriented Programming (ECOOP), Lecture notes in computer science, Vol.1241, pp. 220-242 (1997)

[Kom 01] Komondoor, R., Horwitz, S., Using Slicing to Identify Duplication in Source Code. International Symposium Static Analysis (SAS), Lecture Notes In Computer Science; Vol. 2126, pp. 40-56 (2001)

[Kom 11] Kompanek, A., Proposed Aesop System Architecture, http://www-2.cs.cmu.edu/afs/cs/project/able/www/aesop/html/design_docs/aesop-newdesign, 19/09/2011

[Koo 95] Koopmans, P. S., On the Definition and Implementation of the Sina/st Language, Master of Science thesis, Twente university (1995)

[Kri 01] Krinke, J. Identifying Similar Code with Program Dependence Graphs, 8th Working Conference on Reverse Engineering (WCRE'01), pp. 301-309 (2001)

[Kul 04a] Kulesza, U., Garcia, A., Lucena, C., Generating aspect-oriented agent architectures, presented at Workshop on Early Aspects (2004)

[Kul 04b] Kulesza, U., Garcia, A., Lucena, C., Towards a method for the development of aspect-oriented generative approaches (2004)

[Kul 04c] Kulesza, U., Garcia, A., Lucena, C., Staa, A., Integrating Generative and Aspect-Oriented Technologies," presented at 19th ACM SIGSoft Brazilian Symposium on Software Engineering, Brasília, Brazil (2004)

[Luc 02] Luckham, D., The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems: Addison-Wesley (2002)

[Mad 86] Madsen, O. L., Block Structure and Object Oriented Languages, OOPWORK '86 Proceedings of the 1986 SIGPLAN workshop on Object-oriented program (1986)

[Mag 94] Magee, J. Dulay, N., Kramer, J., A constructive development environment for parallel and distributed programs, presented at Workshop on Configurable Distributed Systems, Pittsburgh, USA (1994)

[Mar 04] Marin, M., Deursen, A.V., Moonen, L. Identifying Aspects Using Fan-In Analysis. 11th Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, pp. 132-141 (2004)

[Mas 03] Masuhara, H., Kiczales, G., Modelling Crosscutting in Aspect-Oriented Mechanisms,17th European Conference on Object Oriented Programming (ECOOP), Darmstadt (2003)

[May 96] Mayrand, J., Leblanc, C., Merlo, E., Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics, 12th International Conference on Software Maintenance (ICSM), 244-254, Monterey, USA (1996)

[Med 02] Medvidovic, N. Rosenblum, D. S., Redmiles, D. F., Robbins, J. E., Modeling software architectures in the Unified Modeling Language, ACM Transactions on Software Engineering Methodologies, vol. 11, pp. 2-57 (2002)

[Med 96a] Medvidovic, N., Oreizy, P., Robbins, J. E., Taylor, R. N., Using Object-Oriented Typing to Support Architectural Design in the C2 Style, presented at Symposium on the Foundations of Software Engineering, San Francisco, CA, USA (1996)

[Med 96b] Medvidovic, N. Taylor, R. N., Whitehead, E. J., Formal Modeling of Software Architectures at Multiple Levels of Abstraction., presented at California Software Symposium, Los Angeles, CA, USA (1996)

[Men 05] Mens, K., Tourwé, T., Delving source code with formal concept analysis, Elsevier Journal on Computer Languages, Systems & Structures, 31(3-4), Special Issue: Smalltalk, Elsevier, pp. 183-198 (2005)

[Men 08] K. Mens , A. Kellens, J. Krinke, Pitfalls in Aspect mining , Proceeding WCRE '08 Proceedings of the15th Working Conference on Reverse Engineering (2008)

[Mes 07] Meslati, D., Ingénierie des logiciels: les nouveaux paradigmes, Master cours, Badji-Mokhtar university, Annaba Algeria (2007)

[Mez 03] Mezini, M. and Ostermann, K. Modules for Crosscutting Models. Ada-Europe 2003, LNCS, vol. 2655, Springer, pp. 24–44, Heidelberg (2003)

[Mor 02] Moreira, A., Araújo, J., Brito, L., Crosscutting Quality Attributes for Requirements Engineering, Proc. 14th International conference on Software engineering and knowledge engineering, Ischia (2002)

[Mor 05] Moreira, A. Araujo, J., Rashid, A., A Concern-Oriented Requirements Engineering Model, International Conference on Advanced Information Systems Engineering (CAiSE) (2005)

[Mor 11] Moreira, A., Araújo, J., The Need for Early Aspects, LNCS 6491, Springer-Verlag Berlin Heidelberg, pp. 386-407 (2011)

[Mul 91] Müller, B., Reinhardt, J., Neural Networks, An introduction, Springer Verlag (1991)

[Myl 92] Mylopoulos, J., Chung, L., Nixon, B., Representing and Using Non-Functional Requirements Engineering: A Process-Oriented Approach, IEEE Transactions on Software Engineering, IEEE Computer Society, Vol. 18, issue 6, pp. 483-497 (1992)

[Ngu 04] Nguifo, E. M., Fouille de Données et Treillis de Galois, Tutoriel, Clermont-Ferrand France (2004)

[Nus 03] Nuseibeh, B., Kramer, J., Finkelstein, A., ViewPoints: Meaningful Relationships Are Difficult! Invited paper, Proceedings of International Conference on Software Engineering, presented at International Conference on Software Engineering (ICSE'03), Postland, Oregon, USA (2003)

[Nus 94] Nuseibeh, B., Kramer, J., Finkelstein, A., A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification, Transactions on Software Engineering, IEEE CS Press, Vol. 20, pp. 760-773 (1994)

[Omg 13] OMG, http://www.uml.org/, 20/05/2013

[Oss 01] Ossher, H., Peri Tarr,P., Hyper/J: multi-dimensional separation of concerns for Java, ICSE'01 Proceedings of the 23rd International Conference on Software Engineering (2001)

[Pam 72] Parnas, D., On the Criteria to Be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15 (1972)

[Paw 04] Pawlak, R., Retaillé, J.P., Seinturier, L., Programmation orientée aspect pour Java/J2EE, chap. 1-2, book, ISBN: 2-212-11408-7, EYROLLES (2004)

[Pet 62] Petri, C.A., Kommunikation mit Automaten, Rheinisch-Westfaliches Institut fur Intrumentelle Mathematik an der Universitat Bonn, Schrift Nr 2 (1962)

[Pin 03] Pinto, M. Fuentes, L. Troya, J. M., DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development, presented at International Conference on GPCE, Erfurt, Germany (2003)

[Poa 11] POA, http://jac.ow2.org/docs/slides/JAC_10_2002fr.pdf , 24/04/2013

[Poa 12] POA,http://xpose.avenir.asso.fr/viewxpose.php?site=36&subpage=/contents/presentation.html, 18/04/2012

[Poa 13] POA, http://membres.multimania.fr/tonysoueid/divers/poa.pdf , 20/03/2013

[Poa 15] POA, http://igm.univ-mlv.fr/~dr/XPOSE2005/gmasquelier/contents/vocabulaire.html, 14/04/2015

[Qu 07] Qu, L., Liu, D., Aspect Mining Using Method Call Tree, International Conference on Multimedia and Ubiquitous Engineering, Korea (2007)

[Rap 11] Rapide™ Project, http://pavg.stanford.edu/rapide/, Stanford University, 20/12/2011

[Ras 03] Rashid, A., Moreira, A., Araujo, J., Modularisation and Composition of Aspectual Requirements, 2nd Aspect Oriented Software Conference (AOSD), Boston, USA (2003)

[Ras 05] Rashid, A., Moreira, A., Araujo, J., Sawyer P., Sampaio, A., A Multi-Dimensional, Model-Driven Approach to Concern Identification and Traceability, 1st Workshop on Models and Aspects, 19th European Conference on Object-Oriented Programming, Glasgow, Scotland (2005)

[Rob 02] Robillard, M., Murphy, G., Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies, 24th International Conference on Software Engineering (ICSE), Orlando, pp. 406-416, USA (2002)

[Sam 05] Sampaio, A., Chitchyan, R., Rashid, A., Rayson, P., EA-Miner: A Tool for Automating Aspect-Oriented Requirements Identification, 20th International Conference on Automated Software Engineering (ASE), California, USA (2005)

[Sam 07] Sampaio, A., Rashid, A., Ruzanna Chitchyan, R., Rayson, P., EA-Miner: Towards Automation in Aspect-Oriented Requirements Engineering, Transactions on Aspect-Oriented Software Development III, Lecture Notes in Computer Science Vol. 4620, pp. 4-39 (2007)

[San 03] Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A., On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework, Brazilian Symposium on Software Engineering (SBES), Manaus, Brazil (2003)

[Saw 02] Sawyer, P., Rayson, P., and Garside, R., REVERE: Support for Requirements Synthesis from Documents. Information Systems, Frontiers 4, 3 (2002)

[Saw 96] Sawyer P., I. Sommerville, and S. Viller, Preview: Tackling the Real Concerns of Requirements Engineering, CSEG Technical Report, Computing Department, Lancaster University (1996)

[Seq 14] Diagramme de séquence, http://cian.developpez.com/uml2/tutoriel/sequence/, 08/08/2014

[Sha 96] Shaw, M., Garlan, D., Software Architectures: Perspectives on an Emerging Discipline, Englewood Cliffs, NJ: Prentice-Hall (1996)

[She 04] Shepherd, D., Pollock, L., Interfaces, Aspects and Views: The discoveries of a clustering Aspect Miner and Viewer, Internal Report, Computer and Information Sciences, University of Delaware, Newark, De 19716, http://webcache.googleusercontent.com/search?q=cache:4oWLfCgF73oJ:glaciers.wlu.edu/pubs/bitstream/handle/id/67/shepherdLate.ps%3Fsequence%3D1+&cd=1&hl=fr&ct=clnk&gl=fr

[She 05] Shepherd D., Pollock, L., Interfaces, aspects and views, In Linking Aspect Technology and Evolution (LATE) Workshop (2005)

[Som 96] Sommerville, I., Sawyer, P., PREview Viewpoints for Process, Requirements Analysis, Lancaster University, Lancaster REAIMS/WP5.1/LU060 (1996)

[Som 97] Sommerville, I., Sawyer, P., Viller, S., Viewpoints: Principles, Problems and a Practial Approach to Requirements Engineering, Lancaster University, TR: CSEG/15/1997 (1997)

[Ste 74] Stevens, W., Myers, G., Constantine, L., Structured design, IBM Systems Journal, Vol. 13, pp.115-139 (1974)

[Su 06] Su, Y., Li, F., Hu, S., Chen, P., Aspect-oriented software reverse engineering, Journal of Shanghai University, Vol. 10, number 5 (2006)

[Sut 02] Sutton Jr, S., Rouvellou, I., Modeling of Software Concerns in Cosmos, In 1st Aspect-Oriented Software Development Conference (AOSD'02), ACM. pp. 127-133, Enschede, Netherlands (2002)

[Sut 03] Sutton, S. M., Concerns in a Requirements Model - A Small Case Study, presented at Early Aspects 2003 Workshop: Aspect-Oriented Requirements Engineering and Architecture Design (held with AOSD 2003), Boston, USA (2003)

[Tar 99] Tarr, P. L., Ossher, H., Harrison, W. H., Sutton, S. M., N degrees of separation: multi-dimensional separation of concerns , presented at Proc. 21st International Conference on Software Engineering (ICSE) (1999)

[Tek 04] Tekinerdogan, B., ASAAM: Aspectual software architecture analysis method, 4th Working IEEE/IFIP Conference on Software Architecture WICSA (2004)

[Tie 05] Tien, M., Programmation Orientée Aspect, Rapports des Travaux Personnels Encadrés (2005)

[Tom 12] Tomcat, http://tomcat.apache.org/, 12/12/2012

[Ton 04] Tonella, P., Ceccato, M., Aspect Mining through the Formal Concept Analysis of Execution Traces, 11th Working Conference on Reverse Engineering (WCRE), Delft, the Netherlands (2004)

[Tou 04] Tourw'e, T., Mens, K., Mining aspectual views using formal concept analysis, département d'Ingénierie Informatique, université catholique de Louvain (2004)

[Tre 14] Treillis de Galois, http://www.iro.umontreal.ca/~valtchev/Research/Projet-Rech.html, 02/02/2014

[Uml 14] UML basics: The sequence diagram, http://www.ibm.com/developerworks/rational/library/3101.html, 14/01/2014

[Van 01] VanLamsweerde, A., Goal-Oriented Requirements Engineering: A Guided Tour, In 5th Requirements Engineering Conference (RE'01), IEEE Computer Society, pp. 249-262, Toronto, Canada (2001)

[Wic 99] Wichman, J. C., ComposeJ –The development of a preprocessor to facilitate Composition Filters in the Java Language, Master of Science thesis, Twente university (1999)

[Won 00] Wong, W., Gokhale, S., Horgan, J. Quantifying the Closeness between Program Components and Features. Journal of Systems and Software, (2000)

[Won 95] Wong, K., Tilley, S. R., Müller, H. A., Storey, M.-A. D., Structural Redocumentation: A Case Study, IEEE Software, 12(1): pp. 46-54 (1995)

[Yos 99] Yoshikiyo, W., Griswold, W., Y, Y., Yuan, J., Aspect Browser: Tool Support for Managing Dispersed Aspects, 1st Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems at 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Denver, USA (1999)

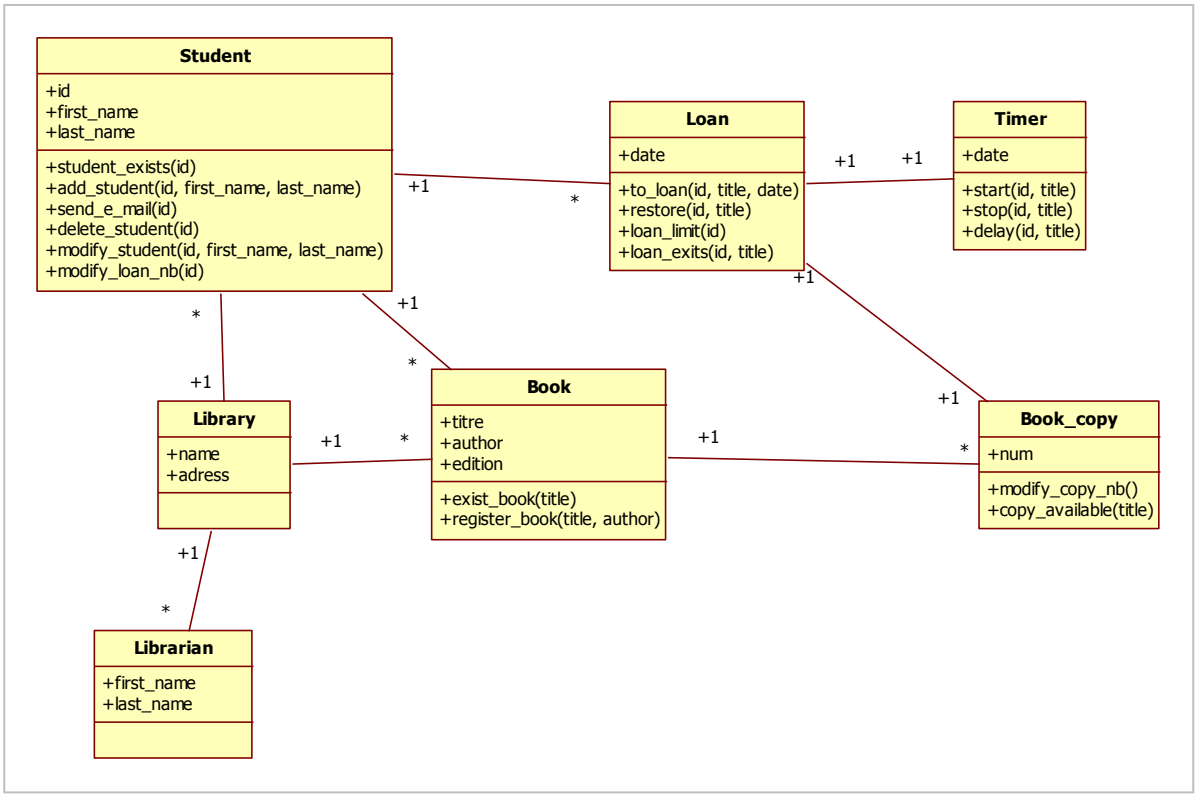
[You 79] Yourdon, E., Constantine, L., Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall (1979)

[Yu 04] Yu, Y., Leite, J. and Mylopoulos, J., From Goals to Aspects: Discovering Aspects from Requirements Goal Models, 12th International Requirements Engineering Conference, IEEE Computer Society, pp. 38-47 (2004)

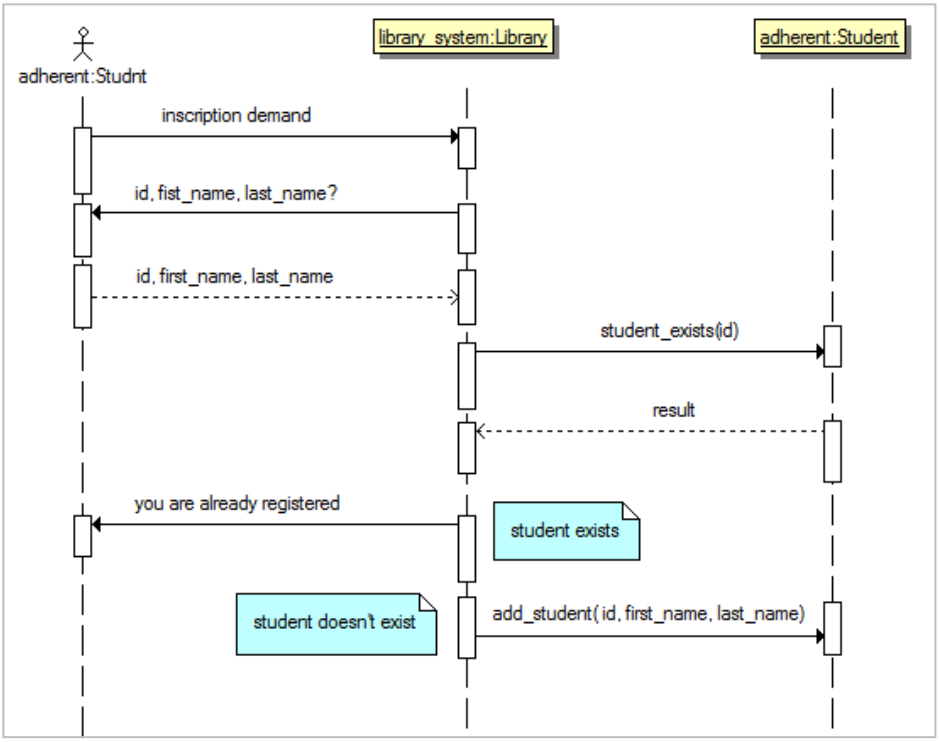
[Yu 97] Yu, E., Towards Modelling and Reasoning Support for Early-Phase Requirements Engineerin, In 3rd Requirements Engineering Conference (RE'97), IEEE Computer Society, pp. 226-235, Annapolis, USA (1997)

[Zha 03] Zhang, C. and Jacobsen, H., A Prism for Research in Software Modularization Through Aspect Mining, Technical Communication, Middleware Systems Research Group, University of Toronto (2003)

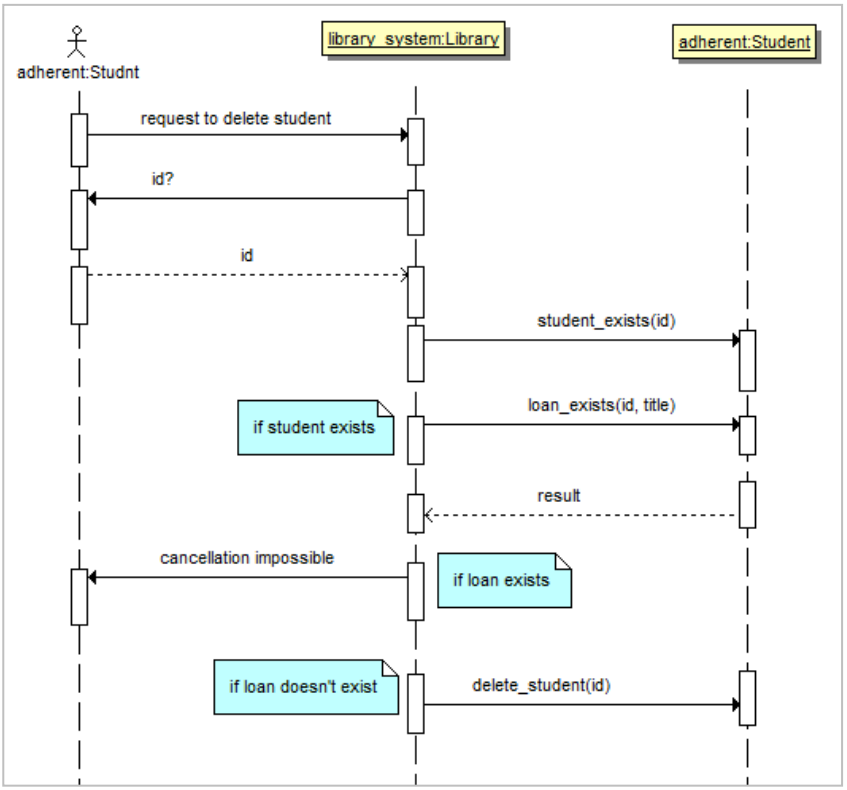
ANNEXE



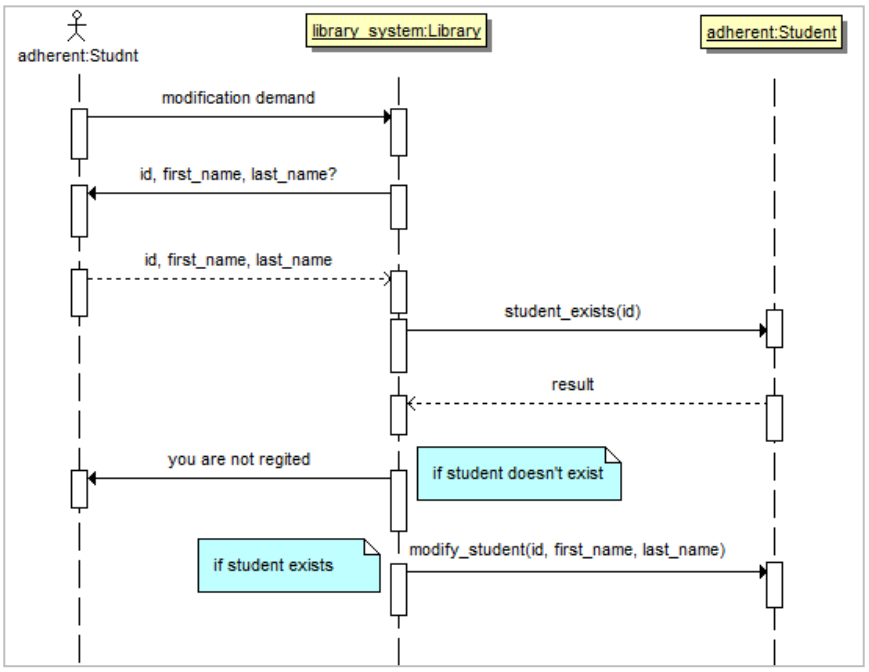
An. 1. Diagramme de classes du système de gestion de bibliothèque



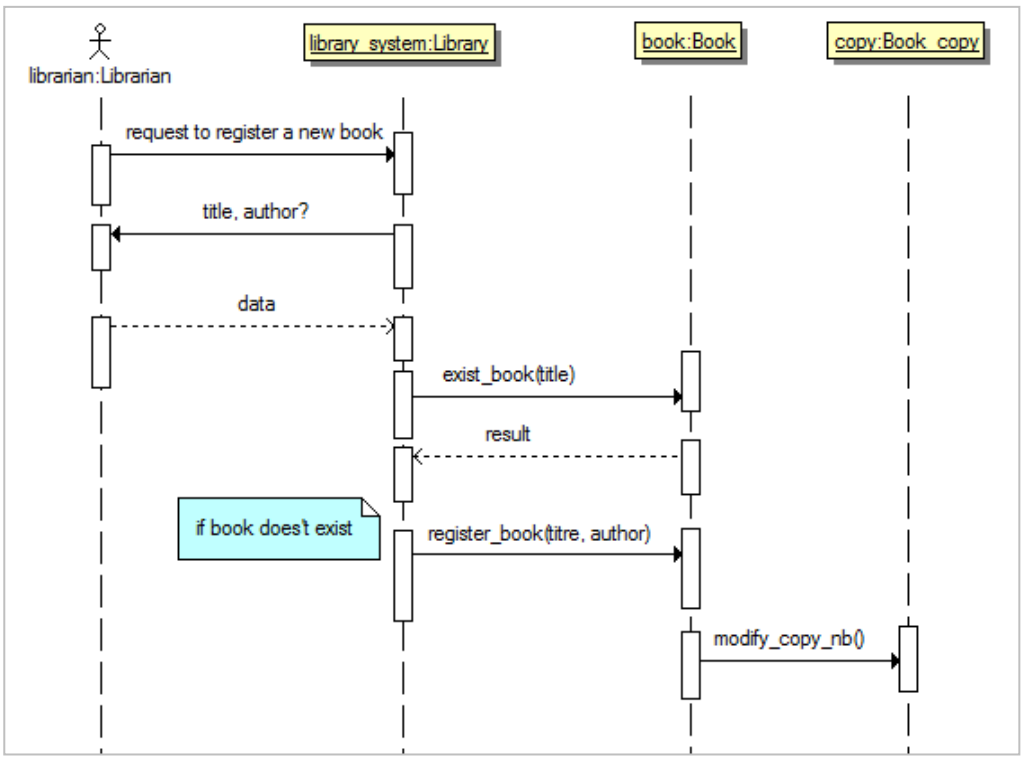
An. 2. Diagramme de séquence du cas d'utilisation inscription d'un étudiant



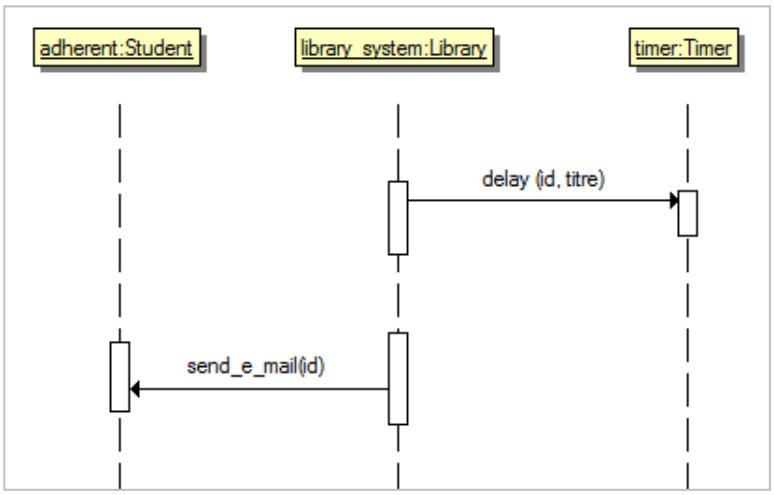
An. 3. Diagramme de séquence du cas d'utilisation désinscription d'un étudiant



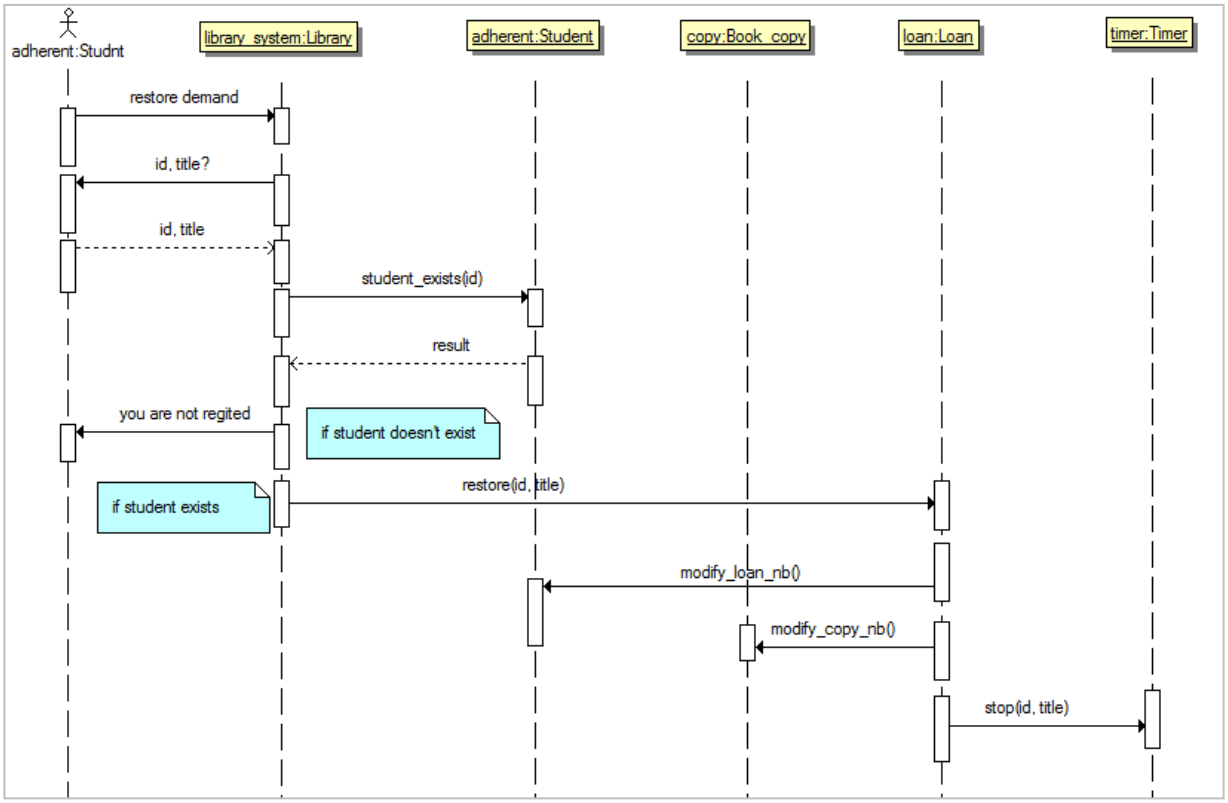
An. 4. Diagramme de séquence du cas d'utilisation modification d'un étudiant



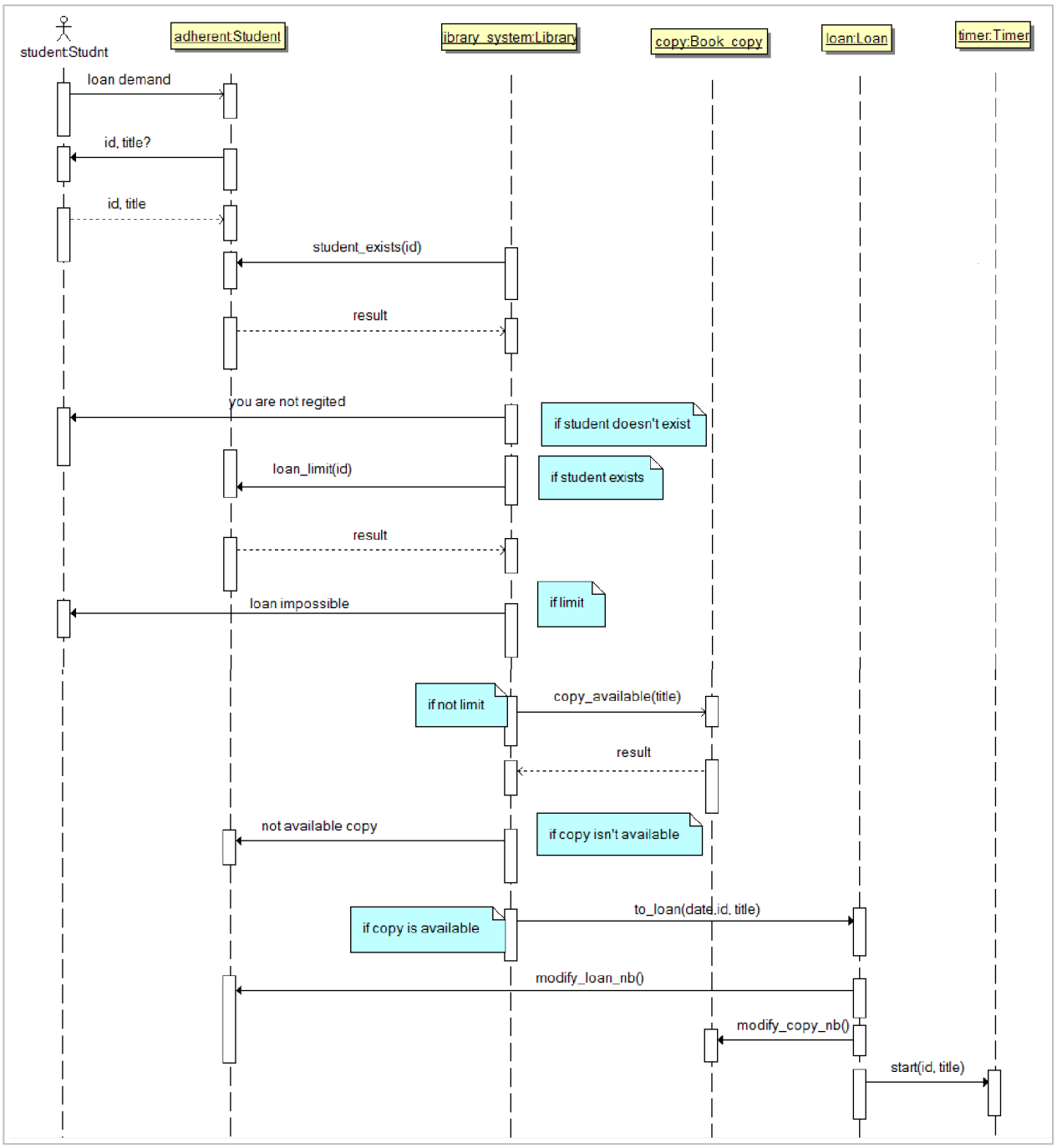
An. 5. Diagramme de séquence du cas d'utilisation enregistrement d'un nouveau livre



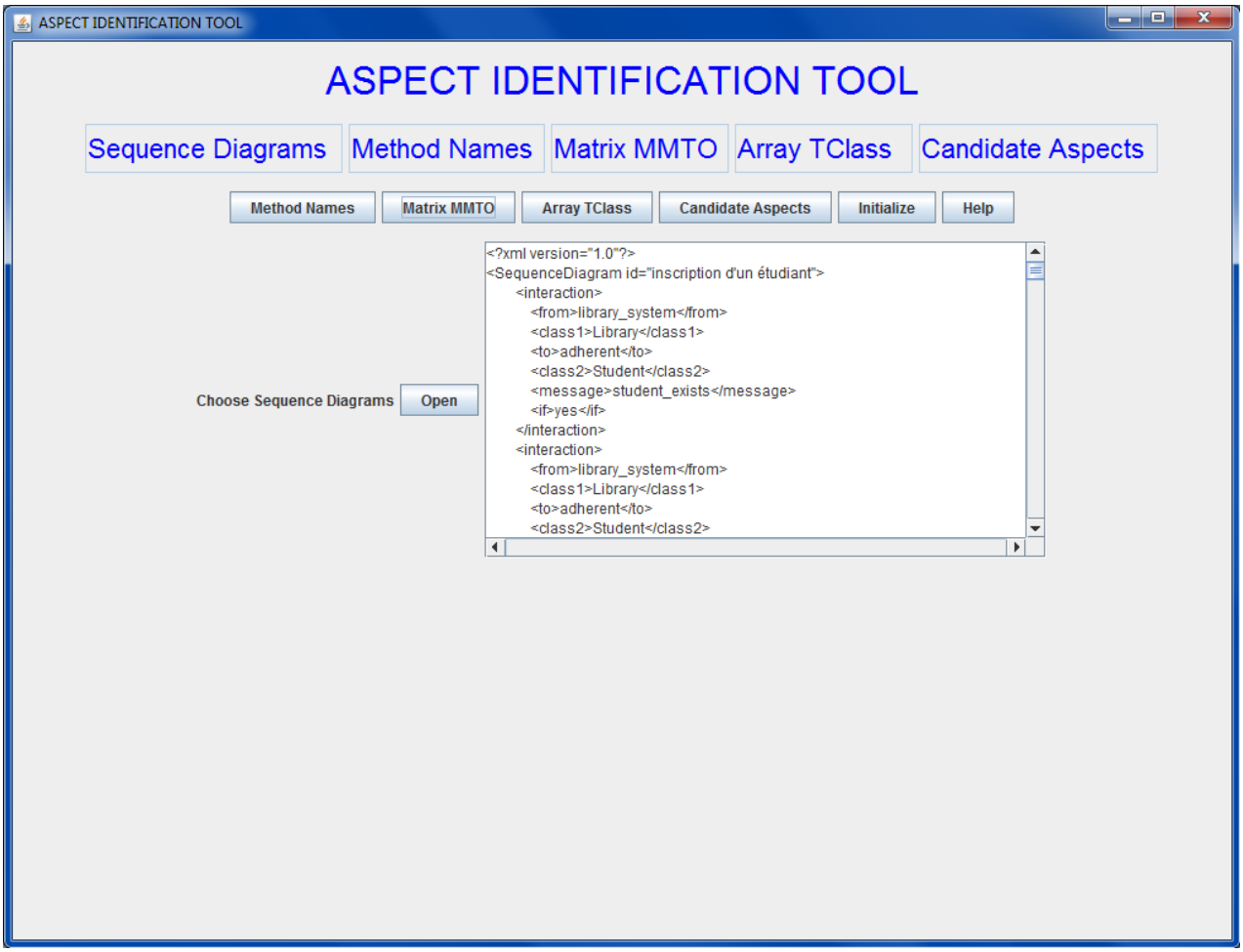
An. 6. Diagramme de séquence du cas d'utilisation délai



An. 7. Diagramme de séquence du cas d'utilisation remise d'un livre



An. 8. Diagramme de séquence du cas d'utilisation emprunt d'un livre



An. 9. Interface de notre outil d'identification d'aspects

GLOSSAIRE DES TERMES UTILISÉS DANS CETTE THÈSE

Dans ce glossaire, nous présentons les définitions de quelques termes utilisés dans ce mémoire, extraites et traduites à partir du glossaire « *IEEE Standard Glossary of Software Engineering Terminology* 1990 », ainsi que la traduction de chaque terme en anglais, en utilisant l'abréviation « Ang ».

A

Abstraction(Ang. ***Abstraction***): Une vue d'un objet qui se base sur l'information concernant un but particulier et ignore le reste d'information.

Algorithme(Ang. ***Algorithm***): Un ensemble fini des règles bien définies pour la solution d'un problème dans un nombre fini d'étapes.

Analyse statique (Ang. ***Static Analyze***): Le processus d'évaluation d'un système ou composant basé sur de sa forme, structure, contenu ou documentation.

Analyse d'exigences (Ang. ***Requirement analysis***):Le processus d'étude des besoins de l'utilisateur afind'arriver à la définition du système, exigences matérielles ou logicielles.

Anomalie (Ang. ***Anomaly***): quelque chose observé dans la documentation ou l'opération du logiciel quidévie des attentes basées sur les produis ou les documents du logiciel de référence précédemment vérifiés.

Architecture (Ang. ***Architecture***): Structure organisée d'un système ou d'un composant.

Attribut (Ang. ***Attribut***): Une caractéristique d'un élément, par exemple la couleur, la taille ou le type de l'élément.

C

Call (Ang. ***Appel***): Une instruction d'ordinateur qui transfère le contrôle à partir d'un module de logiciel à un autre, et souvent spécifie les paramètres à transmettre à et à partir d'un module.

Caractéristique du logiciel (Ang. ***Software feature***): une caractéristique distinguée d'un élément du système (par exemple performance, portabilité ou fonctionnalité).

Code source (Ang. ***Source code***): Instructions d'ordinateur et définitions de données exprimées dans une forme convenable à l'entrée à un assembleur, compilateur ou autre traducteur.

Cohésion (Ang. ***Cohesion***): La manière et le degré auxquels les tâches effectuées par un seul module du logiciel sont liées à un autre.

Complexité (Ang. ***Complexity***): Le degré auquel un système ou un composant a une conception ou une implémentation qui est difficile à comprendre et vérifier.

Composant (Ang. ***Component***):un des parties qui compose un système. Un composant peut être matériel ou logiciel et peut être divisé en d'autres composants.

Conception (Ang. ***Design***):Le processus de définition d'architecture, composants, interfaces et autres caractéristiques d'un système ou d'un composant.

Couplage (Ang. ***Coupling***): la manière et le degré d'interdépendance entre les module du logiciel.

Cycle de développement du logiciel (Ang. ***Software development cycle***):La période de temps qui commence avec la décision de développer un produit logiciel et finit lorsque le logiciel est délivré. Ce cycle typiquement comprend la phase des exigences, phase du teste, et parfois la phase d'installation et de la vérification.

Cycle de vie du logiciel (Ang. ***Software life cycle***):La période de temps qui commence lorsqu'un produit logiciel est conçu et finit lorsque le logiciel n'est plus valable à utiliser. Le cycle de vie du logiciel comprend typiquement laphase des exigences, phase de conception, phase d'implémentation, phase du teste, phase d'installation et de vérification, phase d'opération et de maintenance, et parfois, phase de retraite.

D

Décomposition hiérarchique (Ang.***Hyerarchical decomposition***): Un type de la décomposition modulaire dans lequel un système est répartidansune hiérarchie des composants, à travers une série de raffinements de haut vers le bas.

Dynamique (Ang. ***Dynamic***): Relative à un évènement ou processus qui se produit durant l'exécution d'un programme ordinateur.

E

Etat (Ang.***State***): Une condition ou un mode d'existence, dans lequel un système, un composant ou une simulation peut être.

Efficacité (Ang. ***Efficiency***): Le degré auquel un système ou un composant effectue ses fonctions désignées avec une consommation minimale de ressources.

Encapsulation (Ang. ***Encapsulatio*n**): Une technique de développement du logiciel qui consiste d'isoler une fonction du système ou un ensemble de donnés et opérations sur ces donnés,dans un module et fournir les spécifications précises d'un module.

Entité (Ang. ***Entity***): En programmation ordinateur, un élément qui peut être nommé ou dénoté dans un programme. Par exemple, un élément donné, déclaration du programme ou sous programme.

Exécuter (Ang. ***Execute***): Réaliser une instruction, processus ou programme d'ordinateur.

Exécution (Ang. ***Run***): Dans l'ingénierie du logiciel, une seule, parfois continue, exécution d'un programme ordinateur.

Exception (Ang. ***Exception***): Un évènement qui cause suspension d'une exécution normale du programme. Les types comprennent exception d'adressage, exception de donnée, exception d'opération, exception de débordement,exception de protection.

Exigences (Ang. ***Requirement***): Une condition ou capacité nécessaire par un utilisateur pour résoudre un problème ou accomplir un objectif.

F

Flexibilité (Ang. ***Flexibility***):La facilité avec laquelle un système ou un composant peu être modifié pour l'utilisation dans des applications ou environnements autre que ceux pour lesquels il a été spécifiquement désigné.

Fichier (Ang. ***File***): Un ensemble des enregistrements reliés traités comme unité. Par exemple dans le contrôle de stock, un fichier peut se composer d'un ensemble d'enregistrements de facture.

Fonction (Ang. ***Fonction***):Une action définie d'objectif ou de caractéristique d'un système ou composant. Par exemple, un système peut être peut avoir un contrôle inventaire comme ses fonctions primaires

G

Graphe (Ang. ***Graph***): Un diagramme qui représente la variation d'une variable en comparaison avec celle d'une ou d'autres variables, par exemple, un graphe montrant une courbe bathtub.

Graphe d'appel (Ang. ***Call graph***): Un diagramme qui identifie les modules dans un système ou programme informatique et montre quelles modules appellent un module.

H

Hiérarchie (Ang. ***Hierarchy***): Une structure dans laquelle des composants sont rangés dans des niveaux de subordination, chaque composant a zéro, un, ou plus de subordination , et aucun composant a plus d'un composant subordonné.

I

Implémentation (Ang. ***Implementation***): Le processus de traduction d'une conception en des composants matériels, composants logiciels, ou des deux.

Ingénierie du logiciel(Ang. ***Software engineering***): L'application d'une approche systématique, disciplinaire et quantifiable au développement, opération et maintenance du logiciel, qui est l'application de l'ingénierie au logiciel.

Instanciation (Ang. ***Instanciation***): Le processus de substitution de données ou instructions spécifiques, ou les deux, dans une unité générique de programme, afin de la rendre utilisable dans un programme ordinateur.

Intégrité (Ang. ***Integrity***): Le degré auquel un système ou un composant empêche l'accès non autorisé ou la modification des programmes ou données d'ordinateur.

Intégration (Ang. ***Integration***): Le processus de combinaison des composants logiciels, composants matériels, ou les deux, dans un système total.

Instruction de programme (Ang. ***Program instruction***):Une instruction d'ordinateur dans un programme source.

Instruction d'ordinateur (Ang. ***Computer instruction***): Une déclaration dans un langage de programmation, spécifiant une opération à être effectuée par un ordinateuret les adresses ou les valeurs des opérandes associés, par exemple : *Move A to B* (déplacer A vers B).

Interface (Ang. ***Interface***):Un composant logiciel ou matériel qui relie deux ou plusieurs autres composants afin de passerl'information de l'un à l'autre.

Interrompre (Ang. ***Interrupt***): La suspension d'un processus de manipuler un événement externe au processus.

L

Langage de programmation (Ang. ***Programming language***): Un langage utilisé pour exprimer des programmes d'ordinateur.

Langage formel (Ang.***Formal Laguage***): Un langage dont les règles sont explicitement établies avant son utilisation. Par exemple les langages de programmation et les langages mathématiques.

Langage orienté objet (Ang. ***Object oriented language***):Un langage de programmation qui permet l'utilisateur d'exprimer un programme en termes d'objets et messages entre ces objets.

Liste (Ang. ***List***): Un ensemble d'éléments de données, chacun a la même définition de données.

Logiciel (Ang. ***Software***): Programmes d'ordinateur, procédures, et documentation éventuelles associées et données concernant l'opération d'un système d'ordinateur.

M

Maintenabilité (Ang. ***Maintainability***): La facilité avec laquelle un système logiciel ou un composant peut être modifiéafin de corriger les fautes, améliorer la performance ou autre attribut, ou l'adapter à l'environnement changé.

Maintenance (Ang. ***Maintenance***): Le processus de modification d'un système logiciel ou composant après livraison, afin de corriger les fautes, améliorer la performance ou autres attributs, ou l'adapter à l'environnement changé.

Métrique (Ang. ***Metric***):Une mesure quantitative du degré auquel un système, composant ou processus possède un attribut donné

Modularité (Ang. ***Modularity***): Le degré auquel un système ou programme d'ordinateur est composé des composantsséparés tel que un changement d'un composant a un impact minimal sur les autres composants

Module (Ang. ***Module***): Une unité de programme qui est dséparée et identifiable en respectant la compilation, la combinaison avec d'autres unités, et chargement.

O

Objet (Ang. ***Object***): Concernant le résultatd'unprocessus d'assemblageou de la compilation.

Opérande (Ang. ***Operand***): Une variable, constante, ou fonction sur laquelle une opération est sera effectuée.

Opérationnel (Ang. ***Operational***):Concernant un système ou composant qui est prêt à l'utilisation dans son environnement désiré.

P

Parallèle (Ang. ***Parallel***): Concernant le transfert simultané, occurrent, ou traitement des parties individuellesde l'ensemble, tels que les bits d'un caractère, en utilisant des équipements séparés pour les différentes parties.

Paramètre (Ang. ***Parameter***):Une variable qui donne une valeur constante pour une application précise.

Partitionnement (Ang. ***Partitioning***):La décomposition, la séparation de l'ensemble dans des parties.

Performance (Ang. ***Performance***):Le degré auquel un système ou un composant accomplit ses fonctions désignées dans les contraintes données, telle que la vitesse, la justesse, ou l'usage de la mémoire.

Procédure (Ang. ***Procedure***):Un ensemble d'actions à prendre pour accomplir une tâche donnée.

Processus (Ang. ***Process***):Une séquence d'étapes réalisées pour un but donné, par exemple le processus de développement du logiciel.

Programme d'ordinateur (Ang. ***Computer program***): Une combinaison d'instructions d'ordinateur et de définitions de données qui permettent le matériel d'ordinateur d'effectuer des fonctions de calculou de contrôle.

Q

Qualité (Ang. ***Quality***): Le degré auquel un système, composant ou processus rencontre des exigences spécifiées.

R

Réseau de Pérti (Ang. ***Petri net***):Un modèle abstrait et formel de flux d'informations, illustrant les propriétés statiques et dynamiques d'un système. Un réseau de Petri est d'habitude représenté comme un graphe ayant deux types de nœuds (appelés places et transitions) connectés par des arcs, et des marques indiquant les propriétés dynamiques.

Réutilisabilité (Ang. ***Reusability***): Le degré auquel un module du logiciel ou autre produit du travail peut être utilisé dans plus d'un programme ordinateur ou système logiciel.

Réutilisable (Ang. ***Reusable***): Concernant un module du logiciel ou autre produit du travail qui peut être utilisé dans plus d'un programme ordinateur ou système logiciel.

S

Sémantique (Ang. ***Semantic***): Les relations de symboles ou groupes de symboles avec leurs sens dans un langage donné.

Séquentiel (Ang. ***Sequential***): Concernant l'existence de deux ou plusieurs évènements ou activités tel que le comportement de chacune doit se terminer avant le débuts des prochaines.

Si-Alors-Sinon (Ang. ***If-Then-Else***):Une entrée unique,une sortieuniqueà deux voies de branche qui définit une condition, spécifiant le traitement à réaliser si la condition est trouvée, et optionnellement dans si ce n'est pas le cas, et retourne un contrôledans les deux casàl'instruction immédiatement aprèsla constructionglobale.

Spécification formelle (Ang. ***Formal specification***): Une spécification écrite et approuvée en conformément à des standards établis.

Simplification (Ang. ***Simplicity***):Le degré auquel un système ou un composant a une conception et implémentation qui est simple et facile à comprendre.

Simulation (Ang. ***Simulation***):Un modèle qui se comporte ou opère comme un système donné lorsqu'un ensemble des entrées controlées sont fournies.

Standard (Ang. ***Standard***): Les exigences obligatoirement employées et appliquées pour prescrire une approche disciplinaire et uniforme au développement du logiciel, qui est, les conventions et les pratiques obligatoires sont enfait standards.

Statique (Ang. ***Static***):Concernant un évènement ou processus qui apparait sans l'exécution du programme d'ordinateur, par exemple, l'analyse statique.

Syntaxe (Ang. ***Syntax***): Les règles structurelles et grammaticales qui définissent comment les symboles dans un langage doivent être combinées sous forme des mots, phrases, expressions et autres constructions allowable possibles.

Système (Ang. ***System***): Une collection des composants organizés pour accomplir une fonction spécifiée ou un ensemble de fonctions.

T

Tâche (Ang. ***Task***): Une séquence d'instructions traitées comme une unité de base du travail par un programme de contrôled'un système opérant.

Taxonomie (Ang. ***Taxonomy***): Un schéma qui divise un corps de connaissance et définit les relations entre les pièces. Elle est utilisée pour classifier et comporendre le corps de connaissance.

Techniques (Ang. ***Techniques***): Les procédures tequniques et gérantes qui aident dans l'évaluationet l'amélioration du processus de développement du logiciel.

Traçabilité (Ang. ***Traceability***): Le degré auquel une relation peut être établie entre deux ou plusieurs produits du processus de développement, plus particulièrement les produits ayant les relations prédecesseur-successeur ou maitre-subordonné, par exemplele degré auquel les exigences et la conception d'un composant donné du logiciel correspondant.

Traces d'exécution (Ang. ***Execution trace***): Un enregistrement de la séquence des instructions exécutées durant l'exécution d'un programme ordinateur. Souvent prend la forme d'une liste desétiquettesrencontrées du codecomme les exécutions du programme.

U

Unité (Ang. ***Unit***): Un élément séparément testable, spécifié dans la conception d'un composant du logiciel.

Utilisabilité (Ang. ***Usability***):La ficilité avec laquelle un utilisateur peut apprendre à opérer, préparer les entrées, et interpèter les résultats d'un système ou d'un composant.

V

Validation (Ang. ***Validation***): Le processus d'évaluation d'un système ou composant durant ou à la fin du processus du développement, afin de déterminer s'il satisfait les exigences spécifiées.

Variable (Ang. ***Variable***): La quantité ou élement de données dont la valeur peut changer, par exemple la variable temps_actuel.

Vérification (Ang. ***Verification***): Le processus d'évaluation d'un système ou composant, afin de déterminer si les produits d'une phase donnée du développement satisfaient les conditions imposées au début de la phase.

Version (Ang. ***Version***):Une sortie initiale ou nouvelle dortied'un produit de configuration de logiciel, associé avec une compilation complète ou recompilation de cet élément.

A PROPOS DE L'AUTEUR

Biographie

Fairouz Dahi a obtenu son baccalauréat en science de la nature et de la vie en 2005. Elle rejoignit l'université de Badji Mokhtar – Annaba (UBMA) la même année, pour suivre une formation en informatique et obtenir une licence option système d'information en 2008. Elle obtint son master option ingénierie des logiciels complexes en 2010. Depuis cette date, elle a commencé à préparer sa thèse sous la direction du Dr. Bounour, pour l'obtention du diplôme de doctorat de 3ème cycle.

Actuellement, elle est membre du laboratoire d'ingénierie des systèmes complexes LISCO, qui s'active autour des thèmes d'évolution et de réutilisation du logiciel. Ses intérêts de recherche comprennent: l'aspect mining, la maintenance des logiciels, la migration orientée aspect et la rétro ingénierie des logiciel.

Courriel : fairouz_dahi@yahoo.fr

Adresse : Laboratoire LISCO, Département d'informatique, Université Badji Mokhtar – Annaba, P.O. Box 12, 23000 Annaba, Algeria.

Publications internationnales

- ✓ Dahi, F., Bounour, N., **Identification of Crosscutting Concerns at Design Level**, International Journal of Computer Applications in Technology (IJCAT), Special Issue on: "Current Trends and Improvements in Software Engineering Practices", ISSN: 0952-8091 (*in press*).
- ✓ Dahi, F., Bounour, N., **Détection des préoccupations transversales par l'analyse formelle de concepts des diagrammes de séquence**, Revue africaine de la recherche

en informatique et mathématiques appliquées ARIMA, Vol.18, pp. 19-35, ISSN : 1638-5713 (2014).

Communications internationales

- ✓ Dahi, F., Bounour, N., **Crosscutting Concerns Identification Approach Based on the Sequence Diagram Analysis**, 2nd International Conference on Model and Data Engineering MEDI'12, Poitiers, France, LNCS Vol. 7602, pp. 141-152, Springer (2012).
- ✓ Dahi, F., Bounour, N., **Détection des préoccupations transversales au niveau architectural**, 11th African conference on Research in computer science and applied mathematics CARI'12, Algiers, Algeria (2012).
- ✓ Dahi, F., Bounour, N., **Aspect Identification: Approaches and Challenges**, 2nd International Conference on Information Systems and Technologies ICIST'12, Sousse, Tunisia, ISBN: 978-9938-9511-2-7 (2012).
- ✓ Dahi, F., Bounour, N., **Etude critique des approches de découverte d'aspect à travers le cycle du développement de logiciels**, 2nd International Conference on Complex Systems CISC'11, Jijel, Algeria (2011).
- ✓ Dahi, F., Bounour, N., **Identification d'aspects par l'analyse des concepts formels**, 1st International Conference on Information Systems and Technologies ICIST'11, Tebessa, Algeria, pp. 515-321, ISBN: 978-9931-9004-0-5 (2011).