وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA

جامعة باجي مختار – عنابـــــــــة

Faculté des Sciences de l'Ingénieur

Département d'Informatique

THÈSE

Présentée en vue de l'obtention du diplôme de DOCTORAT

Une Approche Hybride pour la Séparation des Préoccupations avec Résolution de Conflits durant l'Ingénierie des Besoins

---

Discipline

INFORMATIQUE

Par

Abdelkrim AMIRAT

DEVANT LE JURY

Président	Mr. Med. Benmohamed	Professeur	Université - Mentouri - Constantine
Examinateurs	Mr. Allaoua Chaoui	MC	Université - Mentouri - Constantine
	Mr. Nacereddine Zarour	MC	Université - Mentouri - Constantine
	Mme. Hayet Merouani	MC	Université - Badji Mokhtar - Annaba
Directeur de thèse	Mr. Med. Tayeb Laskri	Professeur	Université - Badji Mokhtar - Annaba
Co-encadrant	Mr. Tahar Khammaci	MC	Université - Nantes - France

Année 2007

Remerciements

Je tiens à remercier vivement :

Mon directeur de thèse, *Le Professeur Mohammed Tayeb Laskri* pour avoir accepté de diriger notre travail de thèse. Nous tenons à le remercier pour sa disponibilité à tout moment et ce parfois au détriment de ses maintes obligations, en tant que recteur de l'université d'Annaba. Nous avons trouvé en lui une oreille attentive à nos préoccupations, un soutien indéfectible face à l'adversité et, par-dessus tout, une confiance absolue. Pour toutes ces raisons, nous lui exprimons à travers ces quelques mots toute notre profonde gratitude.

Mon co-encadrant, *Monsieur Tahar Khammaci, Maître de Conférences à l'Université Nantes*, pour toutes les facilités qui m'ont été accordées au sein du laboratoire LINA, pour l'aide qu'il m'a apportée durant la rédaction de cette thèse et pour son soutien moral durant toute la période de notre collaboration au sein du laboratoire LINA de l'université de Nantes.

Les membres du jury : *A Monsieur Mohamed Benmohamed*, Professeur à l'Université de Mentouri, Constantine, pour m'avoir fait l'honneur d'accepter de présider le jury de cette thèse. *A Monsieur Allaoua Chaoui,* Maître de conférences à l'Université de Mentouri, Constantine, pour l'intérêt qu'il a porté à mes travaux. *A Monsieur Nacereddine Zarour*, Maître de conférences à l'Université de Mentouri, Constantine, pour avoir accepté d'examiner cette thèse. *A Madame Hayet Merouani*, Maître de conférences à l'Université de Badji Mokhtar, Annaba, pour m'avoir honoré de sa présence comme membre du jury et pour s'être intéressée à mes recherches.

Le professeur *Mourad Oussalah*, pour m'avoir accepté au sein de son équipe de recherche MODAL et pour tout ce qu'il m'a inculqué sur le plan scientifique et méthodologique.

Mon collègue *Monsieur Djamel Meslati,* pour son caractère agréable et pour ses qualités humaines exceptionnelles et pour m'avoir accepté au sein de son équipe de recherche et surtout pour m'avoir orienté vers le domaine de l'orienté aspect et m'avoir conseillé et aidé pendant les moments difficiles de ma vie de doctorant.

Les membres de l'équipe ERELOG du laboratoire LRI, Annaba : *Tahar Kimour, Mme Atil, Mme Bounour et Mme Saheb* pour leur esprit d'équipe et leur collaboration scientifique.

Les membres de l'équipe MODAL du laboratoire LINA, Nantes : *Dalila Tamzalit, Adel Smeda, Nassima Sadou, Olivier Le-Goaer et Sylvain Maillard* pour leurs collaborations et leur soutien scientifique.

Le chef du département d'informatique de l'université de Annaba, *Monsieur Djamel Solh Amrane,* pour toutes les facilités administratives auxquelles j'ai eu droit et pour son esprit éclairé par lequel il gère le département d'informatique et que j'ai eu à constater durant tout mon service au sein de ce département et ce depuis déjà une dizaine d'années.

Tous ceux qui m'ont aidé durant la période de thèse « *et spécialement Mr. Nourdine TAMANI* **».**
Enfin, à *ma mère*, à ma femme *Sihem* pour son soutien moral et à *mes enfants* et à mes *amis ...*

Résumé

Les techniques telles que les cas d'utilisation, les points de vue et les buts aident à séparer les préoccupations des utilisateurs, mais elles ne prennent pas en compte leur uniformité avec les conditions et les contraintes globales. La détection prématurée des aspects complète ces approches en fournissant des moyens systématiques pour manipuler de telles préoccupations. Les aspects ne peuvent pas être localisés, ils sont dispersés dans tous les artéfacts du cycle de vie du logiciel. Ceci réduit la modularité des artéfacts et peut, par conséquent, mener à de sérieux problèmes de maintenance et à un faible taux de réutilisabilité. Malheureusement, les approches conventionnelles de développement de logiciel orientées aspect sont principalement conçues pour identifier les aspects au niveau du code et peu de travaux traitent l'impact des préoccupations transversales dans les premières phases du cycle de développement de logiciel. Dans cette thèse, nous présentons une méthodologie pour séparer les « *aspects* » durant les premières phases du cycle de développement de logiciel de manière générale et en particulier pendant la phase de l'analyse des besoins.

Notre contribution se décline en trois volets : Le premier volet concerne la proposition d'une étude comparative entre les approches classiques non-orientées aspect et les approches orientées aspect. Le deuxième volet concerne la définition d'une démarche de description des concepts de base de notre approche. L'approche proposée est une approche hybride qui traite en même temps les besoins fonctionnels en utilisant le modèle de cas d'utilisation et les besoins non-fonctionnels en utilisant le NFR avec une représentation XML inspirée du modèle PREview. Enfin, le dernier volet concerne la proposition d'un modèle de validation des résultats obtenus.

Mots-clés : Ingénierie des Besoins, Préoccupation, Besoin, Aspect, Fonctionnel, Non Fonctionnel, Conflit, Résolution, Composition, Cas d'Utilisation.

Abstract

Techniques such as use cases, viewpoints and goals can help to achieve the separation of stakeholders concerns however ensuring their consistency with the global requirements and the constraints is largely unsupported. Working on early aspects, therefore, complements these approaches by providing a systematic means for handling such concerns. Early aspects cannot be localized and tend to be scattered over a multiple early life cycle modules. This reduces the modularity of the artifacts and might consequently lead to a serious maintenance problems and low degree of reusability. Unfortunately, conventional aspect oriented software development approaches have mainly focused on identifying the aspects at the programming level and less attention has been given to the impact of crosscutting concerns at the early phases of the software development. In this thesis we focus on a methodology to elicit the crosscutting concerns "*aspects*" in the early life phases of the software development in general and especially during the requirements analysis in particular.

Our contribution is situated around three axes: The first axe concerns with the comparative study between the classic approaches (not aspect oriented) and the aspect oriented approaches. The second axe concerns with the definition of a methodology to describe hybrid approach proposition which processes to deal at the same time with the functional requirements using use case models, and with non-functional requirements using the NFR with the XML representation inspired from the PREview model.

Keywords: Requirement Engineering, Concern, Requirement, Aspect, Functional, Non Functional, Conflict, Resolution, Composition, Use Case.

ملخـــــص

التقنيات مثل حالة الاستعمال, وجهات النظر أو الأهداف تساعد على فصل اهتمامات المتفاعلين مع النظام, ولكن ضمان تطابق هذه الاهتمامات المستخلصة مع الاحتياجات العامة والقيود الواجب احترامها يبقى دائما هدفا منشودا وصعب المنال . البحث في الجانب المبكر لاستخلاص الاهتمامات يعتبر عملا مكملا لهذه التقنيات وذلك لتوفير الآليات الموضوعية للتعامل مع هذه الاهتمامات . الجانب المبكر في أغلب الأحيان لا يمكن تحديد موضعه لأنه بطبيعته يكون منتشرا عبر مجموعة كبيرة من العناصر التي تشكل بداية دورة حياة النظـــــام .

بعض النتائج السلبية لهذه الظاهرة تتمثل في النقص في استغلالية عناصر التصميم وهذا قد يؤدي إلى تعقيد عمليات الصيانة والانخفاض في عامل إعادة استعمال البرمـــجيات. ومن سوء الحظ أن الإسهامات المتوفرة في هذا المجال ركزت على تحديد ومعالجة الجانب (Aspect) في مرحلة كتابة البرامج وكثيرا ما أهملت الاهتمامات المتداخلة في المراحل الأخرى من دورة حياة البرمجيات . في هذه الأطروحة نحاول تقديم منهجية نركز من خلالها على التقاط وتعيين الجوانب ابتداء من المراحل الأولى من دورة حياة البرمجيات بصفة عامة وبصفة خاصة سوف نركز على مرحلة هندسة الاحتياجات.

كلمات مرشـــدة :

هندسة الاحتياجات , اهتمام , احتياج , جانب (Aspect) , خواص , خاصية نزاع, حل نزاع, تركيب, حالة الاستعمال .

Table des matières

Introduction ***1***

Contexte de recherche 1

Problématique 1

Motivations et Objectifs 2

Organisation du document 2

Chapitre 1 La séparation des préoccupations : principes et motivations ***4***

1.1. Introduction 4

1.2. La séparation des préoccupations 5

1.3. L'approche objet 7

1.4. Les préoccupations transversales 8

1.5. Les problèmes du crosscutting 9

1.5.1. Dispersion du code 10

1.5.2. Enchevêtrement du code 11

1.5.3. Conséquences 11

1.6. Approches pour séparation des préoccupations 12

1.6.1. La programmation orientée aspects (*PAO*) 13

1.6.2. La composition de filtres (*CF*) 15

1.6.2.1. Concepts de bases 16

1.6.2.2. Fonctionnement des filtres 18

1.6.3. La séparation multidimensionnelle des préoccupations (*MDSOC*) 19

1.7. Conclusion 22

Chapitre 2 Mise en Œuvre des Approches de Séparation des Préoccupations ***24***

2.1. Introduction 24

2.2. AspectJ 25

2.2.1. Les concepts de base d'AspectJ 25

2.2.1.1. Les points de jointure (*Join Point*) 26
2.2.1.2. Les points de coupe (*Pointcut*) 29
2.2.1.3. Les consignes (*Advice*) 31
2.2.1.4. Les Introductions 31
2.2.2. L'héritage d'aspect 32
2.2.3. La recomposition dans AspectJ 32
2.2.4. La représentation UML des aspects 33
2.3. Les Filtres de Composition 34
2.3.1. Implémentations du CF-modèle 34
2.3.2. Types de Filtres 36
2.4. HyperJ – Concepts de bases 37
2.5. Modèle de transformation entre langages orientés aspect 38
2.6. Conclusion 39
Chapitre 3 Ingénierie des Besoins Orientée Aspect 41
3.1. Introduction 41
3.2. Définitions 43
3.2.1. Besoins (*Requirement*) 43
3.2.1. Préoccupation (*Concern*) 44
3.3. Approches non orientées aspects 44
3.3.1. Approches orientées point de vue (*PREview*) 44
3.3.1.1. Méthode PREview 45
3.3.2. Approches orientées but (*Goal*) 46
3.3.2.1. NFR Framework (NFRF) 47
3.3.2.2. KAOS (Knowledge Acquisition in Automated Specification) 50
3.3.2.3. *i** Framework / Méthodologie Tropos 52
3.3.2.4. Conclusion 54
3.3.3. Approche orientée Problem Frames (PF) 54
3.3.4. Approches basées cas d'utilisation (*Use Cases*) 55
3.3.4.1. Mécanisme de séparation des cas d'utilisation 55
3.3.4.2. Mécanisme de composition des cas d'utilisation 56
3.3.5. Conclusion 56
3.4. Approches Orientées Aspect 57

3.4.1. Approches orientées aspect basées point de vue (AORE) 57
3.4.2. Approches orientées aspect basées but 58
3.4.2.1. Approche ARGM (Aspects in Requirements Goal Models) 59
3.4.3. Approche orientée aspect basée cas d'utilisation 60
3.4.3.1. Approche AOSD/UC 60
3.4.4. Approches orientées MDSC 62
3.4.4.1. Méthode CORE (Concern-Oriented Requirements Engineering) 63
3.4.5. Approche orientée aspect basée composant 64
3.4.5.1. Approche AOREC (AORE for CBSD) 65
3.4.6. Approche Theme/Doc 67
3.5. Comparaison 67
3.6. Langages d'expressions des besoins 70
3.7. Conclusion 72

Chapitre 4 HASoCC : Une Nouvelle Approche AORE 73

4.1. Introduction 73
4.2. Motivations 74
4.3. Etapes du processus 75
4.4. Modèle de l'approche HASoCC 76
4.4.1. Capture et identification des besoins 77
4.4.1.1. Identification des besoins fonctionnels 78
4.4.1.2. Spécification des besoins fonctionnels 78
4.4.1.3. Détection des besoins fonctionnels transversaux 79
4.4.1.4. Identification des besoins non-fonctionnels 79
4.4.1.5. Spécification des besoins non-fonctionnels 80
4.4.1.6. Détection des besoins non-fonctionnels transversaux 80
4.4.2. Processus de composition 81
4.4.2.1. Identification des points d'interaction 82
4.4.2.2. Définition des conflits 83
4.4.2.3. Résolution des conflits et intégration 84
4.5. Validation des besoins identifiés 85
4.6. Le Modèle proposé (HASoCC) et les approches existantes 86
4.7. Projection des éléments de l'ingénierie des besoins vers la phase d'architecture 87
4.8. Conclusion 89

Chapitre 5 Expérimentations 91

5.1. Introduction 91

5.2. Etude de cas (*exemple du système métro*) 92

5.2.1. Identification des besoins fonctionnels 93

5.2.2. Spécification des besoins fonctionnels 94

5.2.3. Identification des besoins non-fonctionnels 95

5.2.4. Spécification des besoins non-fonctionnels 97

5.2.5. Identification des besoins transversaux 98

5.2.6. Composition des besoins 99

5.2.6.1. Identification des points de jointures 99

5.2.6.2. Identification des conflits 99

5.2.6.3. Identification de la préoccupation dominante 100

5.2.6.4. Règles de composition 100

5.3. Conclusion 101

Conclusion et perspectives 103

Bilan 103

Perspectives 104

Liste de nos travaux publiés 107

Bibliographie 109

Liste des figures

	Chapitre 1	
1.1	Conceptualisation d'une application	6
1.2	Préoccupations transversales dans une application	9
1.3.a	Modularité : idéal vs. Réalité (Code dispersé - mauvaise modularité)	10
1.3.b	Modularité : idéal vs. Réalité (Code regroupé - bonne modularité)	11
1.4	Encapsulation des préoccupations transversales	14
1.5	Exemple de recoupement de préoccupations	14
1.6	Représentation simplifiée du modèle CF-Objet	17
1.7	Schéma intuitif pour le filtrage de messages	19
1.8	Exemple d'hyper-slices	20
1.9	La composition dans Hyper/J	21

	Chapitre 2	
2.1	Diagramme de classe UML d'un SEF	26
2.2	Points de jointure dans l'application - SEF	27
2.3	Point de coupe nommé sans paramètres	29
2.4	Point de coupe nommé avec paramètres	30
2.5	Points de coupe primitifs et composés	30
2.6	Consigne de type around	31
2.7	Déclaration d'introductions ajoutant des attributs et des méthodes	32
2.8	Syntaxe abstraite du méta modèle spécifique à AspectJ	33
2.9	Processus de transformation entre langages (e.g. AspectJ / HyperJ)	39

Chapitre 3

3.1	Besoins, contraintes, problèmes et solutions dans l'ingénierie des besoins	43
3.2	Relation préoccupation et besoin	44
3.3	Processus du modèle PREview	45
3.4	Le type sécurité dans le catalogue NFR	48
3.5	Décomposition du softgoal sécurité d'un compte bancaire	49
3.6	Représentation des besoins de la préoccupation de sécurité	49
3.7	Les niveaux conceptuels du model KAOS	51
3.8	Relation entre modèles dans la spécification KAOS	52
3.9	Syntaxe graphique du modèle des buts de KAOS	52
3.10	Représentation d'acteur et de cas d'utilisation	56
3.11	Représentation XML des points de vue	58
3.12	V-Graphe	59
3.13	Procédure de décomposition (cas d'un mauvais scénario)	60
3.14	Modèle de cas d'utilisation de réservation de chambres	61
3.15	Description de l'extension du cas d'utilisation	62
3.16	Représentation en hyper-cube des préoccupations dans CORE	63
3.17	Espace méta-préoccupations	64
3.18	Processus de base de l'approche AOREC	65
3.19	Principaux éléments du méta modèle des RDLs	71

Chapitre 4

4.1	Modèle de l'approche proposée	77
4.2	Processus de transformation d'un modèle AOR vers un modèle AOA	88

Chapitre 5

5.1	Diagramme de cas d'utilisation du système métro	95
5.2	Spécification du besoin sécurité pour cas d'utilisation (EnterSubway)	97
5.3	Règle de composition pour JP_A	100

Table des tables

Chapitre 2

2.1	Signification prédicative des points de jointure primitifs et leur composition dans AspectJ	28
2.2	Type de filtres dans le CF-Modèle	36

Chapitre 3

3.1	Type de contribution entre préoccupations	64
3.2	Comparaison des approches étudiées	69

Chapitre 4

4.1	Spécification par « template » des besoins fonctionnels	79
4.2	Spécification par « template » des besoins non-fonctionnels	80
4.3	Cas d'utilisation affectés par les NFRs	81
4.4	Points d'interaction dans le système	82
4.5	Matrice des contributions des besoins transversaux	83
4.6	Différents opérateurs de composition	85
4.7	HASoCC par rapport aux critères définis dans la section 3.5	86
4.8	Règles de passage (mapping) entre RE – Architecture - AspectJ	89

Chapitre 5

5.1	Besoins identifiés pour le système métro	93
5.2	Spécification par « template » du besoin EnterSubway	94
5.3	Besoins non-fonctionnels identifiés pour le système métro	95
5.4	Spécification par « template » du temps de réponse	96
5.5	Spécification par « template » de la disponibilité	96
5.6	Identification des points de jointure	99

Introduction

Contexte de recherche

Cette thèse s'inscrit dans le domaine du génie logiciel et plus précisément dans celui de l'ingénierie des besoins au sein du paradigme de l'orienté aspect initié principalement par Peri L. Tarr et Gregor Kiczales dans deux projets différents [Tarr 1999] [Kiczales 2001].

Le travail que nous présentons dans cette thèse a été élaboré dans un premier temps au niveau de l'équipe **ERELOG** (**E**volution et **RE**utilisation des **LOG**iciels), du **L**aboratoire de **R**echerche en **I**nformatique (**LRI**) de l'Université de Annaba et dans un deuxième temps au niveau de l'équipe **MODAL** (langages de **MOD**élisation d'**A**rchitectures **L**ogicielles) du **L**aboratoire d'**I**nformatique de **N**antes **A**tlantique (**LINA**) de l'Université de Nantes – France.

Les logiciels actuels répondent à diverses exigences. Nous pouvons distinguer les exigences fonctionnelles, décrivant le comportement du système et définissant les fonctions que le système doit remplir et les exigences non-fonctionnelles qui expriment les contraintes imposées sur la manière de satisfaire les exigences fonctionnelles comme la performance, la sécurité, la concurrence, la synchronisation, etc.

Problématique

Le problème des aspects a été largement étudié au niveau de la phase d'implémentation du logiciel et les résultats sont évidents : un nombre important d'approches de programmation orientées aspect (AspectJ, HyperJ, Composition de Filtres, JAsCo, ApectC, JAC, …).

Mais la question qui se pose est la suivante :

Est-ce que les aspects interviennent uniquement au niveau implémentation ?

Il est évident que les aspects peuvent être considérés dans toutes les phases du cycle de développement des logiciels (de la phase d'analyse des besoins jusqu'à la phase de maintenance). Ainsi, ce constat nous a conduit à prendre en charge le concept aspect dès les premières phase du cycle de développement de logiciels

(E*arly Aspect Oriented Software Development*, *EAOSD*), notamment lors de la phase de l'ingénierie des besoins (*Aspect Oriented Requirement Engineering, AORE*) et la phase d'architecture logicielle (*Aspect Oriented Software Architecture, AOSA*) [Amirat 2005a].

Le travail de recherche présenté dans cette thèse est consacré principalement à l'étude du concept aspect durant la phase de l'ingénierie des besoins. Notons que cette phase est cruciale puisqu'elle conditionne la réussite ou l'échec du logiciel produit.

Motivations et Objectifs

- Fournir un support pour les préoccupations entre-coupantes durant la phase de l'ingénierie des besoins (*Requirement Engineering*, RE).

- Offrir les structures et les mécanismes nécessaires pour l'identification et la gestion des conflits qui apparaissent comme une conséquence de la représentation des préoccupations entre-coupées.

- Préserver la traçabilité et identifier l'influence sur les artefacts dans les prochaines phases du cycle de développement.

- Etablir un compromis critique sur la représentation finale des préoccupations avant que l'architecture du système ne soit dérivée.

Organisation du document

Cette thèse est organisée en cinq chapitres, en plus d'une introduction et d'une conclusion générale.

La suite de ce manuscrit est organisée comme suit :

Chapitre 1. Ce chapitre présente le domaine du développement, l'orienté aspect et les concepts de base des différentes approches d'implémentation de l'orienté aspect ainsi que leurs points forts et leurs points faibles par rapport aux approches traditionnelles (e.g. approche orientée objet).

Chapitre 2. Nous décrivons dans ce chapitre, les trois langages orientés aspect qui sont à l'origine de ce paradigme. Ces trois langages sont issus de trois écoles différentes, à savoir :

- La programmation orientée aspect dirigée par Gregor Kiczales de Xerox [Kiczales 2001].
- La programmation multi-dimensionnelles dirigée conjointement par Peri L. Tarr et Harold Ossher d'IBM [Tarr 1999].
- La programmation basée sur la composition de filtres dirigée par Mehmet Aksit de l'université de Twente – Hollande [Aksit 1997].

Chapitre 3. Dans ce chapitre, nous étudions les concepts de base de l'ingénierie des besoins ainsi que les approches développées dans le cadre de l'ingénierie des besoins et nous présentons les modèles les plus représentatifs de description des besoins. Nous décrivons également les approches non orientées aspect qui sont à l'origine des approches orientées aspect. Nous clôturons ce chapitre par une comparaison des principaux modèles développés suivant des critères liés à la l'analyse des besoins.

Chapitre 4. Nous présentons dans ce chapitre, notre modèle hybride HASoCC (*Hybrid Approach for Separation of Crosscutting Concerns*), basé sur les cas d'utilisation pour l'identification des préoccupations fonctionnelles et les NFR pour l'identification des préoccupations non fonctionnelles, afin de décrire les besoins fonctionnels et non-fonctionnels. Il est fait état également dans ce chapitre, d'une démarche pour guider le concepteur dans son processus d'identification des besoins.

Chapitre 5. Dans ce chapitre, nous décrivons une expérimentation que nous avons effectuée en appliquant le modèle HASoCC sur un système réel de gestion de station métro.

En conclusion, nous faisons le bilan de ce travail, pour souligner très précisément notre contribution et nos résultats. Nous procédons ensuite à une critique de ce travail et proposons alors les raffinements et les perspectives possibles pour notre étude.

CHAPITRE 1

La séparation des préoccupations : Principes et Motivations

1.1. Introduction

Dans “Mythical Man-Months” [Brooks 1995] F. Brooks a écrit: “…*the essential difficulties are inherent in the conceptual complexity of the software functions to be designed and built at any time, by any method*…”, i.e. le logiciel a une complexité *inhérente* certes; cependant toute la complexité dans le développement logiciel n’est pas inhérente : la méthode que nous choisissons pour concevoir et implémenter les logiciels possède souvent sa propre complexité inhérente. Cette complexité est appelée complexité *accidentelle.*

Comme l’apport du développement de logiciel orienté aspect (*Aspect Oriented Software Development*, AOSD) consiste à faire promouvoir le raisonnement modulaire et la composition, il est essentiel d'éliminer les éléments de complexité accidentelle dans ces méthodes.

Durant la phase de conception, la plupart des approches se concentrent sur l’élaboration d'une conception orientée objet, c'est-à-dire, la décomposition du système en un ensemble d’objets, où chacun fournit une partie bien définie de la fonctionnalité principale du système. Par conséquent, les fonctionnalités secondaires (e.g. la sécurité, performance, …) sont souvent mal encapsulées. Ce phénomène est considéré par Ossher et Tarr [Ossher 2001] comme la tyrannie de la décomposition dominante “… *the tyranny of the dominant decomposition is the single most significant cause of the failure, to date, to achieve many of the expected benefits of separation of concerns*”. L’orienté aspect est une voie prometteuse pour contourner ce phénomène inhérent à la décomposition.

Le développement de logiciels orientés aspect est initialement apparu comme une technique d’implémentation et récemment devenu un sujet de recherche très abordé dans d'autres disciplines du développement de logiciels. Particulièrement, un

grand intérêt pour l'approche orientée aspect a émergé dans le domaine de l'ingénierie des besoins (*Requirement Engineering,* RE).

L'AOSD est principalement fondée sur le concept de séparation des préoccupations (*Separation of Concerns*, SoC). La séparation des préoccupations est considérée depuis longtemps comme l'une des approches les plus prometteuses du génie logiciel. Selon cette approche, un programme écrit avec la technologie objet contient deux parties principales :

- Une partie fonctionnelle qui implémente les services pour lesquels les objets ont été créés ; c'est la *préoccupation principale* du programme appelée parfois le code de base.

- Une partie qui adapte les objets du code de base à un environnement et/ou à une application particulière. Cette partie regroupe des *préoccupations secondaires* parfois complexes et disséminées dans tout le code source de l'application.

La séparation de ces deux parties permet d'une part, de réduire le travail et l'expertise demandés au programmeur de l'objet, et d'autre part, de mieux réutiliser la partie fonctionnelle des objets dans d'autres environnements et d'autres applications. Pour cela, l'objet ne doit contenir que sa partie fonctionnelle et son adaptation doit être effectuée à l'extérieur de l'objet.

Après cette brève introduction au principe de la séparation des préoccupations, nous considérons dans la suite de ce chapitre les motivations de cette séparation en tant que technique de programmation puis nous mettons en évidence les insuffisances de l'approche orientée objet. Dans la quatrième section, nous présentons le concept préoccupation transversale (*crosscutting concern*) que l'approche objet n'offre pas de moyens pour l'encapsuler. Dans la cinquième section, nous explicitons les principaux problèmes engendrés par les préoccupations transversales. Pour terminer, nous abordons dans la sixième section l'ensemble des travaux actuellement menés autour de la notion d'aspect.

1.2. La séparation des préoccupations

Nous sommes de plus en plus confrontés à des systèmes logiciels extrêmement complexes. Nous avons intérêt à gérer cette complexité, d'une façon ou d'une autre, pour faire face aux contraintes d'adaptabilité, d'évolution et de réutilisation. Plutôt que d'attaquer le développement d'une application de front, les concepts d'abstraction et de modularisation permettent de raisonner sur un problème donné sans se préoccuper de sa solution entière ou encore moins de la connaître dans sa globalité. Cette abstraction permet ainsi l'identification de plusieurs problèmes récurrents et concrets qui peuvent apparaître dans divers contextes semblables [Pawlak 2000]. La conception des systèmes logiciels consiste alors à décomposer un problème donné en

sous-problèmes (de petites unités ou modules) qui sont traités séparément les uns des autres (figure 1.1). Le système final est ensuite construit par l'implémentation des différents modules conceptuels et leur composition. La programmation de ces unités permet de réduire la complexité du problème à implémenter, augmentant, entre autre, la flexibilité et facilitant aussi la compréhension du système [Parnas 1972].

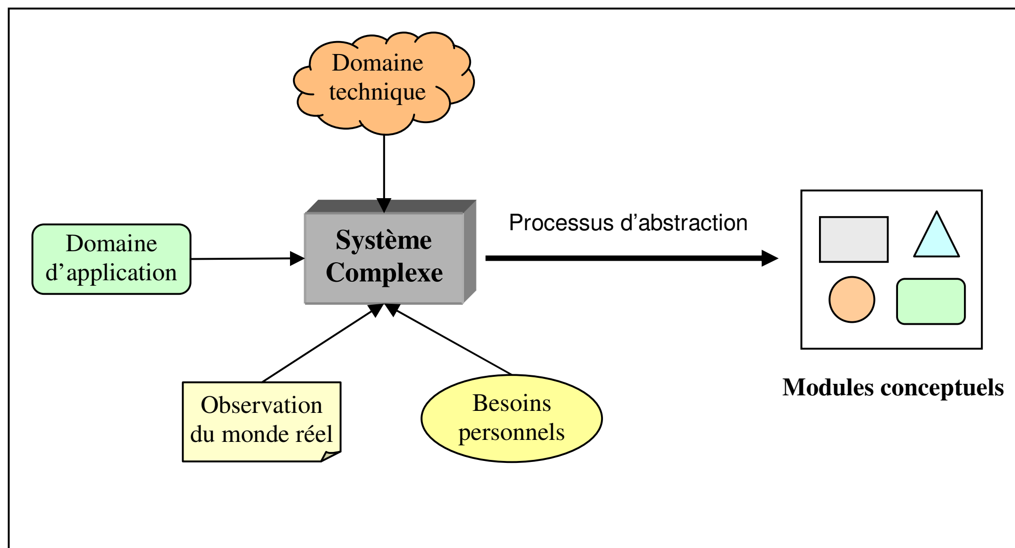


Figure 1.1 : Conceptualisation d'une application

C'est l'approche désormais classique de modularisation évoquée depuis les années 1970 par R. Gauthier [Gauthier 1970] qui a été adoptée par l'ensemble des approches de développement traditionnelles (procédurale, fonctionnelle et objet). Aujourd'hui, toutes ces approches visent à appliquer le principe de séparation des préoccupations dans leur processus de développement [Dijkstra 1976, Hursh 1995]. Une préoccupation est un concept, un but particulier ou une unité d'intérêt. Un système typique admet principalement deux types de préoccupations : fonctionnelles et non-fonctionnelles. Par exemple, l'abstraction d'une carte de crédit admet une préoccupation fonctionnelle concevant le processus de payement et des préoccupations systémiques assurant la gestion de la connexion, l'authentification, l'intégrité de la transaction de paiement, la sécurité, etc.

En adoptant l'idée classique de l'approche de modularisation, le principe de séparation des préoccupations consiste à identifier séparément l'ensemble des préoccupations d'un système logiciel et à les adresser d'une façon relativement indépendante, aussi bien à la phase de conception qu'à la phase d'implémentation. Ce principe est reconnu essentiel au développement des logiciels : il améliore la lisibilité et la flexibilité du système permettant ainsi l'évolution, l'adaptation et la réutilisation de toute ou une partie de l'application [Amirat 2005b].

1.3. L'approche objet

En appliquant le principe de séparation de préoccupations, l'approche orientée objet met en avant la description d'entités du domaine d'application, définies sous la forme d'objets admettant chacun une identité, un état et un comportement particulier. Les langages à objets [Masini 1989, Ducournau, 1998] permettent en effet de décomposer le système en composants paramétrables, indépendants les uns des autres qui s'engagent à fournir chacun un ou plusieurs services au reste du système. Le système est donc conçu comme une composition d'objets collaborant entre eux.

C'est grâce aux nombreux concepts du paradigme orienté objet [Wegner 1990] (l'encapsulation [Snyder 1986], le mécanisme d'héritage, la relation de généralisation/spécialisation, le polymorphisme, etc.) que l'approche orientée objet améliore la réutilisabilité d'un système complexe. L'encapsulation est une technique de mise en oeuvre du principe de décomposition modulaire, produisant ainsi des modules à faible couplage qui, de plus, doivent présenter chacun une et une seule entité logique à forte cohésion. Elle permet aussi de cacher les détails d'implantation au-dessous des interfaces.

Le polymorphisme caractérise la possibilité de définir plusieurs fonctions de même nom mais possédant des paramètres différents. Grâce à de tels concepts, l'approche orientée objet possède un potentiel pour faire accroître la productivité, la flexibilité et la compréhensibilité d'un système informatique, assurant ainsi l'évolution et la réutilisation des logiciels.

Cependant, cette réutilisation n'est pas toujours aisée. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services attendus, mais nécessite également la prise en compte de différentes propriétés non fonctionnelles. En effet, du fait des contraintes d'exécution, il est constamment nécessaire d'y intégrer différentes propriétés non-fonctionnelles comme, par exemple, la gestion de la mémoire, la persistance ou dans un cadre réparti, la répartition des données, la synchronisation, etc. Or, il s'avère que les unités de décompositions fonctionnelles et les propriétés non fonctionnelles sont souvent orthogonales et leur intégration dans un programme unique est compromettante dans le sens où elle conduit à des programmes peu lisibles, difficiles à faire évoluer, à maintenir et à réutiliser.

La décomposition fonctionnelle adoptée par l'approche orientée objet nous donne toujours comme résultat des préoccupations transversales (*crosscutting concerns*) affectant ainsi le développement de l'application de différentes façons. Dans ce qui suit, nous présentons deux problèmes récurrents de l'approche orientée objet ainsi que leurs implications.

1.4. Les préoccupations transversales

Pour illustrer les préoccupations transversales, considérons l'exemple d'une librairie électronique dont la réalisation nécessite, en plus des services attendus, la prise en compte de différentes propriétés non-fonctionnelles affectant l'application dans sa globalité. Dans cette application, les clients utilisent un réseau pour consulter la liste des ouvrages disponibles et éventuellement les commander. Plusieurs clients peuvent être connectés en même temps et plusieurs commandes peuvent être traitées en parallèle.

La conception objet de cette application conduit à définir notamment les classes : *Librairie*, *Client*, *Ouvrage*, *Commande*, *Banque* et *CarteDeCredit* qui représentent les entités du domaine d'application, fournissant les services de base de l'application (e.g. la consultation d'un ouvrage ou la passation d'une commande). Par ailleurs, nous pouvons identifier au moins trois propriétés non fonctionnelles différentes correspondant à des propriétés non-fonctionnelles de cette application : la *distribution*, la *persistance* et la *synchronisation*.

- La *distribution* apparaît du fait que les instances des différentes classes mises en jeu se trouvent sur des sites distants. Cette propriété non fonctionnelle regroupe entre autre la gestion des communications de site à site et la définition du protocole de communication utilisé.

- La *persistance* se justifie parce que des objets comme, par exemple, les ouvrages mis en vente, doivent persister à un arrêt de l'application. La sauvegarde de tels objets sur un support de masse et la gestion des mises à jour de cette sauvegarde font partie de la définition de cette propriété non fonctionnelle, quelle que soit la technique de persistance utilisée.

- La *synchronisation* est mise en évidence, notamment, par le besoin de synchroniser les accès concurrents aux structures des objets. C'est le cas, par exemple, lorsqu'un client demande le prix d'un ouvrage pendant que le libraire modifie ce même prix. Cette propriété non fonctionnelle englobe la gestion de tels accès concurrents.

Nous remarquons ici, que ces trois propriétés non fonctionnelles sont en interaction avec l'ensemble des objets mis en jeu dans l'application, et qu'elles sont transversales à l'ensemble des classes définissant cette application. D'une façon générale, ce phénomène de recoupement des préoccupations (propriétés transversales) est connu sous le nom de *crosscutting*.

Les rectangles, sur la figure 1.2, représentent les classes qui interagissent avec les propriétés transversales représentées par les différentes formes (cercle, triangle et carré). Chaque forme symbolise une propriété donnée. Comme le montre cette figure, une propriété peut interagir avec toutes les classes (par exemple, les carrés) ou seulement quelques-unes (les cercles et les triangles) et vice-versa.

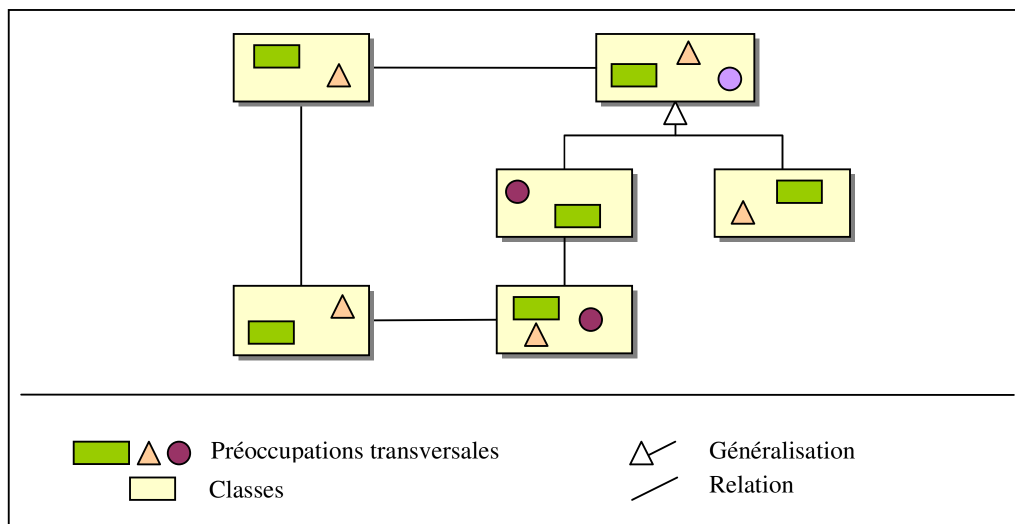


Figure 1.2 : Préoccupations transversales dans une application

Généralement, nous distinguons deux types de préoccupations transversales. Outre les besoins non fonctionnels ou systémiques d'une application, un système admet souvent des fonctionnalités transversales dont l'implantation se trouvent généralement déclinée et éclatée dans la description des différentes classes composant le système. Ce point sera abordé dans le quatrième chapitre.

1.5. Les problèmes du crosscutting

Une utilisation classique de l'approche orientée objet pour traiter ce phénomène de *crosscutting*, impose de mêler le code des préoccupations transversales au code des classes définissant l'application. De tels enchevêtrements introduisent des problèmes récurrents, qui compromettent la réutilisation des implantations de l'ensemble des unités fonctionnelles ainsi que celles des propriétés transversales de l'application, indépendamment les unes des autres [Hursh 1995, Kiczales 1997]. Nous distinguons principalement deux problèmes induits par l'approche orientée objet :

- **L’enchevêtrement du code (*Code Tangling*)**

  L’approche orientée objet conduit à l’enchevêtrement du code source correspondant aux services de base de l’application, traités par les différentes classes, avec celui correspondant aux différentes propriétés transversales.

- **La dispersion du code (*Code Scattering*)**

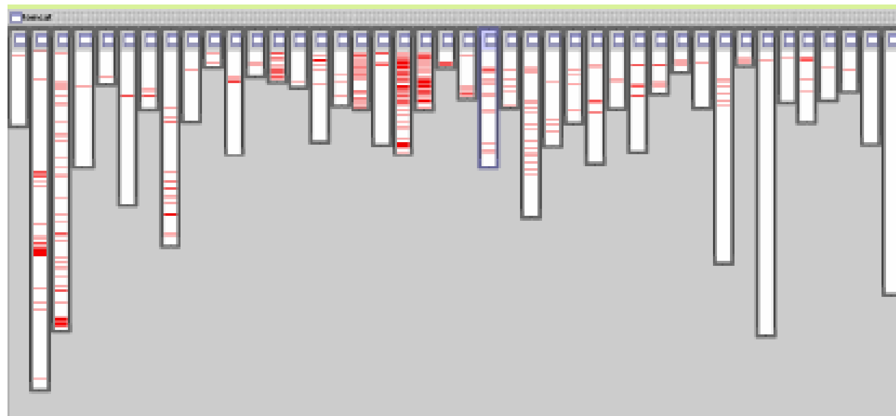
  L’approche orientée objet conduit à la dispersion du code correspondant à chacune des propriétés transversales dans la description des différentes classes concernées par ces propriétés.

Combinés ensemble, ces deux problèmes récurrents contrarient le principe de séparation des préoccupations et affectent ainsi, la conception et l’implantation de l’application de plusieurs façons. Dans ce qui suit, nous illustrons tout d’abord ces deux problèmes et nous détaillons ensuite leurs implications.

1.5.1. Dispersion du code

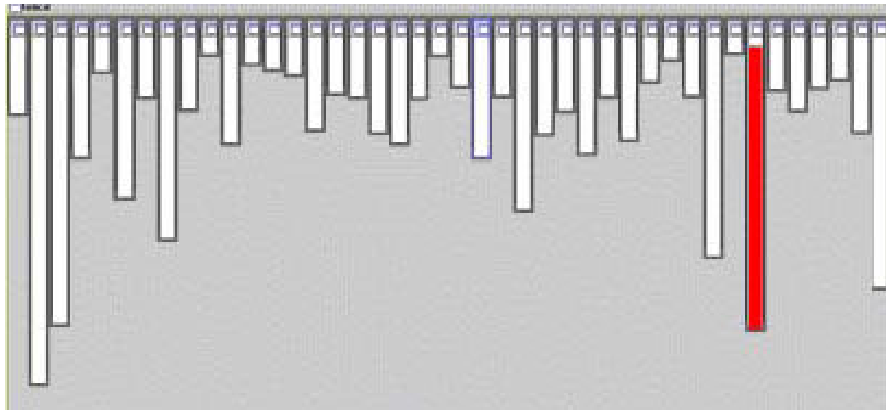
La dispersion du code signifie que la spécification d’une propriété du système n’est pas encapsulée dan un seul module. Par exemple, dans un système de gestion de base de données, il est possible que l’optimisation des performances, la journalisation (*logging*) et la synchronisation puissent concerner toutes les classes accédant à la base de données. On voit donc que ces aspects doivent être implémentés dans plusieurs modules sans être bien circonscrits. Il s’agit très souvent de portions de code similaires à ajouter un peu partout dans les modules concernés par ces propriétés.

Dans la figure 1.3.a, les colonnes illustrent les différents modules et les parties apparaissant en couleur foncée représentent les préoccupations. Ces dernières sont dispersées sur presque tous les modules.



a) Code dispersé (mauvaise modularité)

A l'opposé, la figure 1.3.b donne un exemple d'une préoccupation circonscrite dans un seul module.



b) Code regroupé (bonne modularité)

Figure 1.3 : Modularité: l'Idéal vs. Réalité (extrait de [Kiczales, 2001])

1.5.2. Enchevêtrement du code

Chaque module du système contient la description de plusieurs propriétés ou de fonctionnalités différentes. Un problème qui découle directement du constat précédent est que certaines préoccupations non-fonctionnelles viennent recouper l'implantation des préoccupations fonctionnelles. Il en résulte la présence d'éléments de plusieurs préoccupations dans l'implantation d'un seul et même module.

1.5.3. Conséquences

Les deux problèmes précédents entraînent des conséquences négatives sur le développement du logiciel telles que :

- **Mauvaise traçabilité** : l'implantation simultanée de plusieurs propriétés transversales dans l'ensemble des unités de modularisation fonctionnelle avec lesquelles elles interagissent, rend obscure la correspondance entre une préoccupation et son implémentation. Cette mauvaise traçabilité nuit à la compréhension et par conséquent à l'évolution et à la réutilisation du code source relatif à cette préoccupation.

- **Faible productivité** : l'implantation simultanée de plusieurs propriétés transversales éloigne l'attention du développeur du but final, vers les besoins annexes, limitant d'autant la productivité de l'application. De même, reconsidérer les préoccupations de manière répétitive (i.e. dans chaque module) réduit la productivité.

- **Plus faible réutilisation** : Une unité de décomposition fonctionnelle implémente des propriétés transversales multiples. Les autres systèmes qui nécessitent des services similaires (offerts par cette unité de décomposition fonctionnelle) ne sont pas forcément capables d'utiliser directement cette unité de décomposition modulaire. En effet, les unités de décompositions fonctionnelles regroupent non seulement leurs comportements propres mais en plus, elles y mêlent des propriétés transversales, diminuant ainsi la compréhension et éventuellement la réutilisation de ces unités. Aussi, la réutilisation d'une propriété transversale dans d'autres systèmes est compromise, du fait de la dispersion du code implantant cette propriété à travers l'ensemble des unités concernées par cette dernière.

- **Pauvre qualité du code** : l'enchevêtrement du code final de l'application produit souvent un code mêlant plusieurs préoccupations dissimulées. De plus, en ciblant trop de propriétés transversales en même temps, certaines d'entre elles risquent de ne pas recevoir l'attention nécessaire.

- **Evolution difficile**: une vue limitée et des ressources limitées produisent généralement un modèle de conception qui ne répond qu'aux problèmes actuels. Répondre aux problèmes futurs nécessite souvent de retravailler toute l'implantation. Comme celle-ci n'est pas modularisée, cela veut dire que de nombreux modules de décomposition devront être modifiés. Modifier chaque sous-système pour propager les modifications peut conduire à des incohérences. Cela requiert également des efforts considérables du coté des tests pour s'assurer que de tels changements n'ont pas provoqué des bogues.

Parmi les principaux objectifs de notre travail, nous envisageons d'encapsuler ces préoccupations transversales dans des modules séparés que nous appelons des *aspects*.

1.6. Approches pour la séparation des préoccupations

La séparation des préoccupations est un concept présent depuis de nombreuses années dans l'ingénierie du logiciel [Parnas 1972, Lopes 1995]. Les différentes préoccupations des concepteurs apparaissent comme les premières motivations pour organiser et décomposer une application en un ensemble d'éléments compréhensibles et facilement manipulables. La séparation en préoccupations apparaît dans les différentes étapes du cycle de vie du logiciel et est donc de différents ordres. Il peut s'agir de préoccupations d'ordre fonctionnel (séparation des fonctions de l'application), technique (séparation des propriétés du logiciel système), ou encore liées aux rôles des acteurs du processus logiciel (séparation des actions de manipulation du logiciel).

Par ces séparations, le logiciel n'est plus abordé dans sa globalité, mais par parties. Cette approche réduit la complexité de conception, de réalisation, mais aussi de maintenance du logiciel et améliore la compréhension, la réutilisation et l'évolution. Les principales approches pour la séparation des préoccupations au niveau implémentation sont :

- La programmation orienté aspect [Kiczales 1997]
- Les filtres de composition [Aksit 1998]
- La séparation multidimensionnelle des préoccupations [Ossher 1999]

1.6.1. La programmation orientée aspect (PAO)

Dans le but de permettre une meilleure réutilisation, le paradigme de la programmation orientée aspect considère que les services de base d'une application et ses propriétés ou fonctionnalités transversales, correspondent à des préoccupations indépendantes qui doivent être découplées les unes des autres aussi bien à la phase de conception qu'à la phase d'implémentation.

En effet, tout en appliquant le principe de séparation des préoccupations au niveau implémentation, la programmation orientée aspect prône la possibilité de définir non seulement des composants (procédures, fonctions, objets, etc.) mais également des ***aspects*** comme concepts permettant d'encapsuler les préoccupations transversales d'une application.

Afin de pallier les défauts évoqués précédemment concernant l'approche orientée objet, la programmation par aspects propose de découpler les préoccupations transversales de la structure des objets. Elle met en avant, outre les entités du domaine d'application décrites par des objets, la description des préoccupations transversales, définie sous la forme d'aspects (cf. la figure 1.4).

L'une des techniques les plus étudiées à l'heure actuelle correspond à la programmation par aspects (*Aspect-Oriented Programming,* AOP). Celle-ci a été introduite par des chercheurs du centre XEROX PARC[1] en 1997 [Kiczales 1997]. Il s'agit d'une technique novatrice pour l'ingénierie des applications complexes, telles que les applications distribuées. Elle se fonde sur une séparation claire entre les préoccupations fonctionnelles (services métiers) et non-fonctionnelles (services systèmes) présentes dans les applications. Ce point de vue est similaire à celui présent dans les serveurs d'applications (.Net, EJB, CCM) où les composants fournissent des services métiers et les serveurs d'applications sont une structure d'accueil proposant aux composants des services systèmes.

[1] Le PARC (*Palo Alto Research Center*) nous a entre autre gratifié avant la programmation orientée aspects de Smalltalk, du réseau d'entreprise, de l'interface graphique et de la souris (mise en oeuvre des idées novatrices de Engelbart), de l'imprimante Postscript laser et du premier ordinateur personnel, et ce dès la fin des années soixante.

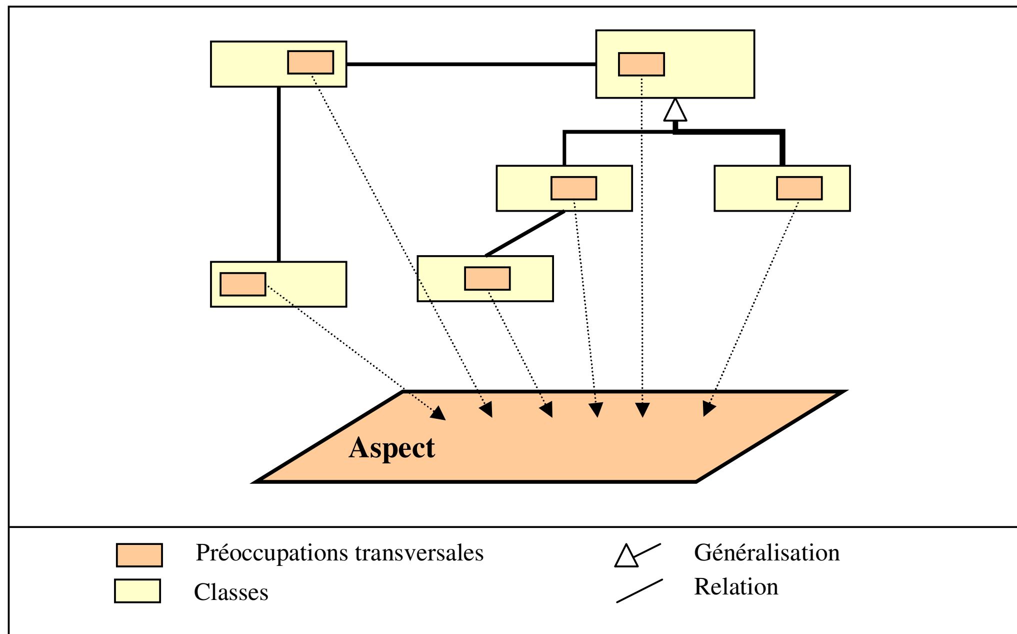


Figure 1.4 : Encapsulation des préoccupations transversales

Néanmoins, l'AOP ne s'arrête pas à ce premier niveau de découpage, elle vise aussi à appliquer la séparation à toutes les préoccupations, qu'elles soient fonctionnelles ou non.

Le principe de la programmation orientée aspect est de coder chaque problématique séparément et de définir leurs règles d'intégration pour les combiner en vue de former un système final. Le tisseur (Weaver) est l'infrastructure qui permet de greffer le code des aspects dans le code des méthodes des classes. La figure 1.5 montre un exemple de recoupement de préoccupations.

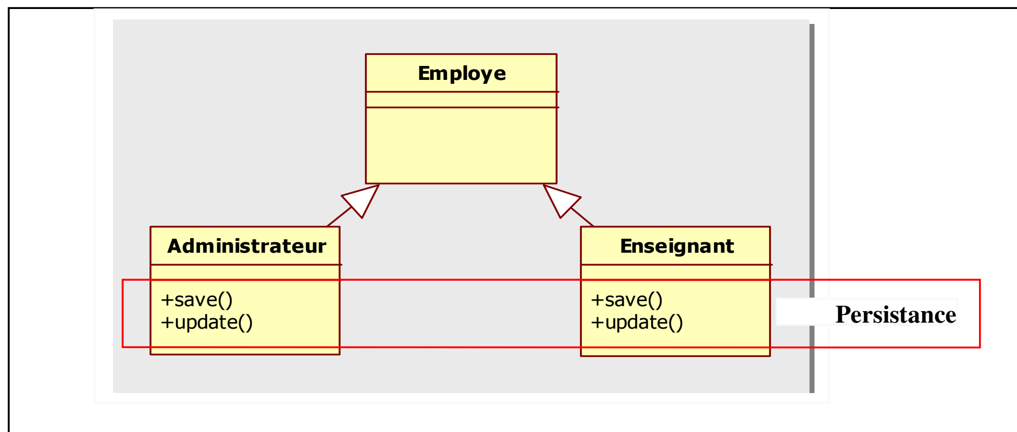


Figure 1.5 : Exemple de recoupement de préoccupations

Sur le diagramme de classes UML de la figure 1.5, on suppose qu'on souhaite enregistrer les informations concernant des employés suite à chaque mise à jour. La persistance est une préoccupation qui concerne les deux classes (administrateur et enseignant) ; on parle alors de recoupement des préoccupations. En général, on dit que des préoccupations se recoupent si plusieurs méthodes sont concernées par ces aspects [Elrad 2001]. La programmation orientée aspect permet d'encapsuler ces préoccupations dans des modules réutilisables [Kiczales 2001].

G. Kiczales dans [Kiczales, 1997] distingue deux types de modules, composants et aspects, en donnant les définitions suivantes :

- Un **composant** est un élément qui peut être clairement encapsulé dans un objet ou dans un module. Les composants sont les unités fonctionnelles d'un système.

- Un **aspect** n'est pas une unité de décomposition du système mais une propriété qui affecte la sémantique des composants du système. C'est une fonctionnalité qui ne peut pas être encapsulée à l'aide des moyens de structuration conventionnels. Les aspects correspondent aux exigences non-fonctionnelles d'un système.

Le but de la POA est de séparer la programmation des composants de celles des aspects et de disposer de moyens de tissage (*weaving*) pour composer le système final sur la base des différents modules et des règles d'intégration.

Chaque aspect est destiné à être développé de façon indépendante puis intégré à une application par le processus de tissage d'aspects. L'une des expérimentations les plus abouties des langages orientés aspect est AspectJ [Kiczales 2001] développé par l'équipe à l'origine de l'AOP.

1.6.2. La composition de filtres (CF)

Le groupe TRESE de l'université de Twente propose depuis 1988 un modèle pour la séparation des préoccupations, basé sur les filtres de composition. Moyennant des interfaces bien construites, la composition de filtres fournit une solution convenable pour résoudre divers problèmes reliés à l'approche objet. Aksit et Bergmans citent, en leur qualité d'auteurs de la composition de filtres, ces problèmes comme les motivations principales de leurs approches [Aksit 1998]. Parmi ces motivations nous pouvons mentionner :

- L'héritage dynamique qui représente la possibilité de changer dynamiquement le lien d'héritage entre classes.
- La modélisation des comportements qui sont liés à l'état d'un objet. Ainsi, l'objet change de comportement selon son état.

- La possibilité d'offrir des vues multiples d'un même objet.
- La synchronisation et coordination des objets.
- L'affichage de traces durant le développement d'un programme.

L'un des avantages de la composition de filtres réside dans l'utilisation d'un mécanisme de filtrage pour résoudre les problèmes cités précédemment.

1.6.2.1. Concepts de base

La composition de filtres ajoute à un objet de base, dit *noyau* (*Kernel*) une ou plusieurs couches enveloppantes appelées interfaces qui interceptent les messages entrants et sortants. Un objet doté d'une interface est appelé CF-objet.

Chaque CF-objet est défini par un CF-modèle. Le CF-modèle est une extension modulaire du modèle orienté objet conventionnel. Cette extension peut être aussi bien utilisée par les langages de programmation conventionnels tels que Java, C++ et Smalltalk que par les modèles à base de composants comme .Net, CORBA et Entreprise JavaBeans. Le concept fondamental de cette extension est l'amélioration du modèle d'objets conventionnels en interceptant et en manipulant tous les messages envoyés et reçus par cet objet. Cela permet d'exprimer plusieurs améliorations au niveau du comportement des objets, puisque dans les systèmes à base d'objet le comportement extérieur visible d'un objet est exprimé par les messages qu'il envoie et qu'il reçoit.

La figure 1.6 illustre cette extension par *l'abstraction* de l'implémentation objet avec une couche d'interface. Cette interface se compose d'un ensemble de filtres. Ces filtres sont groupés dans des modules appelés modules de filtres.

Modules de Filtres : Ces modules sont des sous-composants qui contiennent des filtres pour la manipulation des messages envoyés et reçus. Ces modules de filtres sont des unités pour la réutilisation et l'instanciation du comportement du filtre. Les modules de filtres doivent fournir la spécification des filtres ainsi que des contextes de leurs exécutions. Ces contextes d'exécution sont définis par un ensemble d'objets internes et externes.

Filtres : Les filtres définissent les améliorations possibles du comportement des objets. Chaque filtre spécifie une inspection et une manipulation particulière des messages. Un filtre peut soit accepter, soit rejeter un message et effectuer un certain nombre d'actions sur ce message. Les filtres d'entrées et de sorties peuvent manipuler respectivement les messages reçus et envoyés par un objet. Après la composition des modules de filtres avec les filtres, les messages reçus doivent passer à travers les filtres d'entrées et les messages envoyés à travers les filtres de sorties. Le positionnement de filtres sur un message se fait de manière déclarative. L'ordre d'application des filtres est défini suivant leur ordre de déclaration. La composition

se fait donc par empilement en utilisant des opérateurs de composition prédéfinis [Aksit, 1992].

Objets internes : ce sont des objets dont les méthodes sont utilisées pour étendre le comportement d'un objet de base. Un message reçu par un CF-objet peut être délégué à un objet interne à la place de l'objet noyau auquel il était initialement adressé.

Objets externes : ils sont semblables aux objets internes. Cependant, ils peuvent exister indépendamment des CF-objets. Leurs références sont passées comme paramètres aux constructeurs des CF-objets au moment de leur instanciation. Ces références sont affectées aux variables d'instances correspondantes.

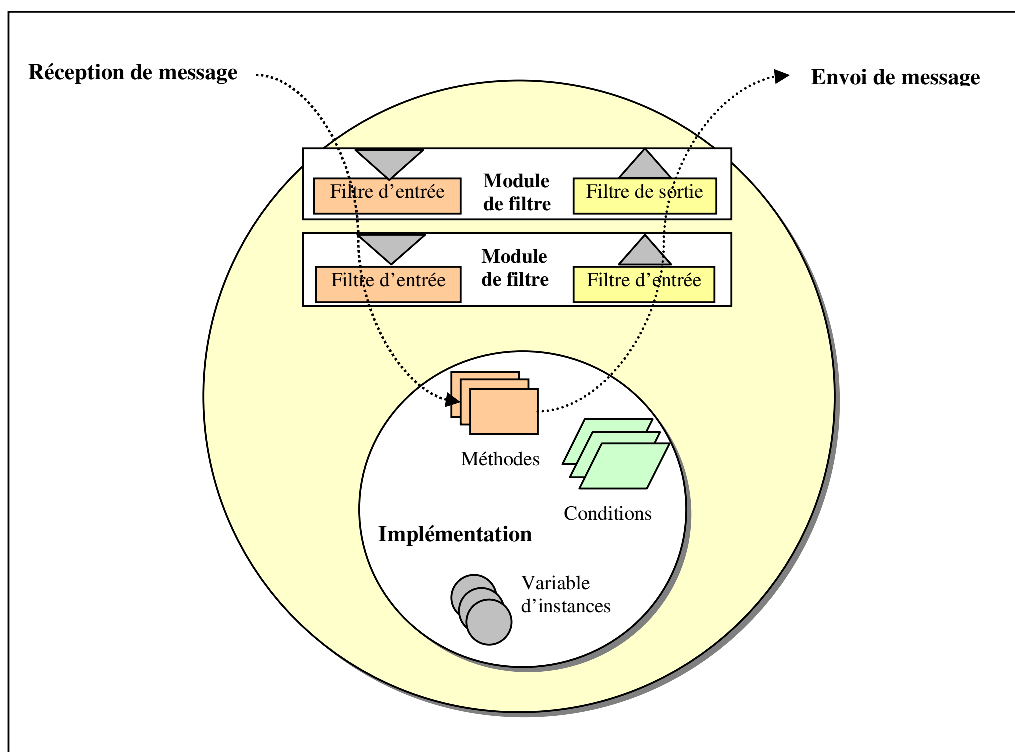


Figure 1.6 : Représentation simplifiée du modèle CF-Objet

L'objet noyau offre une interface d'accès aux méthodes disponibles. Nous pouvons distinguer deux types de méthodes, les méthodes régulières et les méthodes de conditions (appelées conditions) [Aksit, 1992].

- **Méthodes régulières :** ces méthodes implémentent le comportement fonctionnel des objets. Ces méthodes sont invoquées à travers des messages, si les filtres l'autorisent.

- **Méthodes conditions :** ce sont des méthodes spécifiques sans paramètres qui implémentent des expressions booléennes définies pour fournir systématiquement des informations sur l'état de l'objet sans aucun effet de bord. Nous utilisons les conditions pour trois objectifs :

  i. Elles offrent une abstraction de l'état et de l'implémentation de l'objet, permettant aux filtres de considérer seulement les états appropriés.

  ii. Elles permettent aux filtres de rester indépendants des détails d'implémentation de l'objet noyau. Cela a l'avantage de rendre les filtres plus réutilisables.

  iii. Les conditions peuvent être réutilisées par plusieurs modules de filtres.

En conclusion, les méthodes conditions mettent en application la séparation de l'abstraction d'état de l'objet noyau et les préoccupations de filtrage de messages. Ainsi, dans une application donnée, les objets de base représentent le code fonctionnel et les filtres représentent les propriétés non fonctionnelles de cette application [Aksit, 1992].

1.6.2.2. Fonctionnement des filtres

Par le schéma intuitif de la figure 1.7 nous expliquons le processus de filtrage de messages. Notre description se focalise uniquement sur les filtres d'entrées, notons que les filtres de sorties fonctionnent exactement de la même façon. Dans la figure 1.7, nous avons représenté trois filtres (A), (B) et (C), nous supposons qu'ils sont composés séquentiellement. Chaque filtre possède un type de filtre et un pattern de filtre. Le type du filtre détermine la manière de manipuler des messages après qu'ils soient acceptés par le pattern de filtre. Le pattern de filtre est une simple expression déclarative pour accepter et modifier les messages.

Typiquement, les messages sont acheminés séquentiellement à travers les filtres jusqu'à ce qu'ils soient dispatchés. Dispatcher signifie ici commencer l'exécution d'une méthode locale ou déléguer le message à un autre objet. Nous constatons dans la figure 1.7, comment un message est rejeté par le premier filtre (A), ensuite il continue vers le second filtre (B) où il sera accepté et modifié, il continue, enfin, vers le dernier filtre (C) qui l'accepte et le dispatche.

Chaque filtre peut accepter ou rejeter un message. La sémantique associée à une action d'acceptation ou de rejet dépend du type du filtre. Ces actions sont en réalité

des opérations de manipulation (*modification*) de message ou d'exécution de certaines actions.

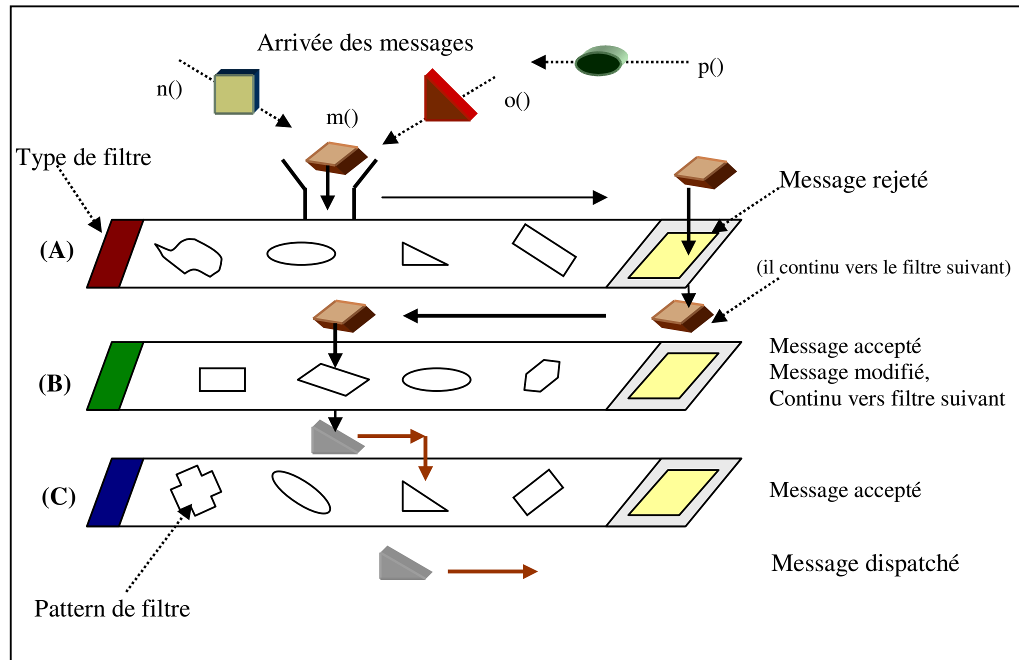


Figure 1.7 : Schéma intuitif illustrant le filtrage de messages

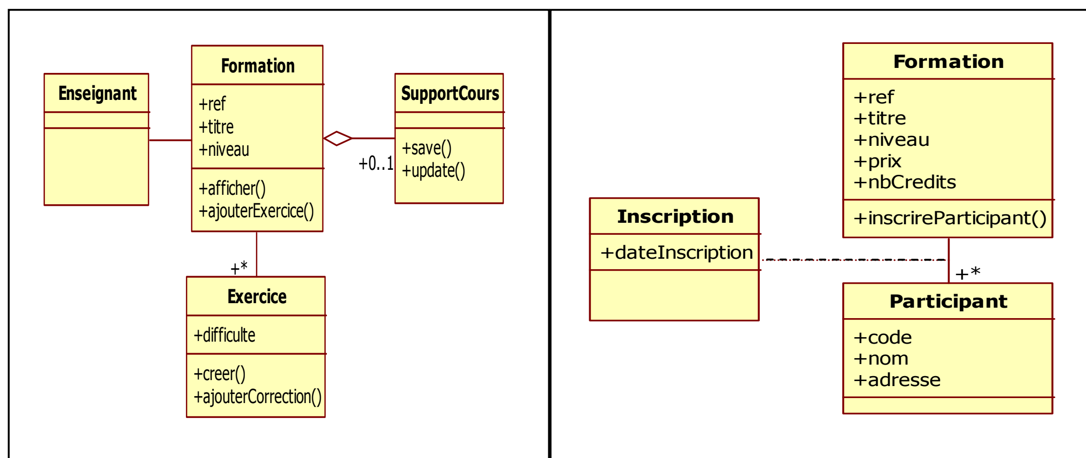
Parmi les types de filtre prédéfinis nous pouvons citer : *Dispatch*, *Substitute,* *Error*, *Wait*, *Meta et realtime* [Aksit, 1992]. Plus de détails seront présentés dans le deuxième chapitre.

1.6.3. La séparation multidimensionnelle des préoccupations (MDSOC)

La séparation multidimensionnelle des préoccupations consiste à décomposer le développement de logiciel selon plusieurs niveaux (fonctionnel, métier, règle de gestion, technologique, etc.) [Tarr 1999]. Ces niveaux sont appelés des dimensions. La structure modulaire issue de la décomposition est appelée un *hyperslice* (hypertranche). Un hyperslice ne contient que les unités pertinentes à une préoccupation particulière. La figure 1.8 illustre un exemple de deux hyperslices.

L'hyperslice « Formation» contient les unités et les propriétés concernant la préoccupation de la formation. Cet hyperslice contient les entités, les propriétés et les méthodes pour la préoccupation de gestion des formations. L'hyperslice

«Inscription» contient les unités et les propriétés concernant la préoccupation d'inscription. Cette décomposition est une décomposition selon une dimension fonctionnelle. Le système peut être décomposé selon d'autres dimensions. Par exemple, une décomposition selon la dimension technologique peut donner lieu aux hyperslices « Persistance », « Sécurité », etc.



HyperSlice – Formation **HyperSlice - Inscription**

Figure 1.8 : Exemple d'hyperslices

Un système est une collection d'hyperslices composés dans un **hypermodule** selon des règles de composition. Pour composer un ensemble d'hyperslices, le développeur doit effectuer un ensemble de déclarations dans un hypermodule en indiquant l'ensemble des hyperslices à composer et les règles de composition. L'outil de composition est appelé Hyper/J [Ossher 2001]). Hyper/J supporte la MDSOC pour le développement JAVA. Il opère sur les « *class file* » pour intégrer les différents hyperslices d'un hypermodule.

La figure 1.9 montre un exemple de déclaration Hyper/J. Cette déclaration montre que les hyperslices à composer sont « Formation » et « Inscription » en utilisant la règle de composition « *Merge by name* ».

```
Hermodule InscriptionPlusFormation
Hyperslices :
     Feature.Inscription,
     Feature.Formation ;
Relationships :
    mergeByName ;
…
End hypermodule
```

Figure 1.9 : La composition dans Hyper/J

Dans le cas général, la séparation des préoccupations s'articule autour d'une préoccupation centrale. Aussi, la programmation orientée aspect considère le code métier d'une application comme la préoccupation centrale. Des travaux plus récents menés au centre de recherche T.J. Watson d'IBM proposent une autre vision de cette approche : la séparation multidimensionnelle des préoccupations. La base de ce travail repose sur les constats suivants :

- De multiples dimensions de séparation peuvent être définies pour le même ensemble de préoccupations.

- Ces multiples dimensions de séparation doivent être utilisables de manière simultanée. Une (ou plusieurs) préoccupation(s) centrale(s) n'est pas imposée aux concepteurs et aux développeurs. Chaque acteur peut donc disposer de sa propre vision d'un système avec ses préoccupations dominantes. Au-delà de l'identification des préoccupations, il est important que l'encapsulation soit suffisante pour limiter l'impact de l'activité liée à l'ascendance d'une préoccupation sur les autres préoccupations. L'absence d'impact entre préoccupations est considérée comme impossible.

- Percevoir les préoccupations comme indépendantes ou orthogonales est attractif, mais rarement totalement applicable en pratique. Il est donc important de permettre la mise en relation des préoccupations tout en offrant une séparation intéressante.

Le projet *Hyperspaces* [Ossher 1999, Tarr 1999] expérimente cette approche au travers de la plateforme *HyperJ*. Un hyperespace est un espace de préoccupations qui structure la séparation multidimensionnelle.

- Les préoccupations sont regroupées au sein de dimensions, ce qui donne à l'hyperespace sa structure multidimensionnelle. Au sein d'une dimension (un hyperespace), les préoccupations sont disjointes (pas de définitions communes).

Deux préoccupations ne peuvent avoir d'intersection au sein d'une dimension, mais peuvent en avoir si elles sont définies dans deux dimensions distinctes.

- Les dimensions (*hyperespace*) sont structurées en modules. Un module contient un certain nombre de concepts représentant une préoccupation ainsi qu'une règle de composition. Ce découpage en concepts et relations provient de la programmation orientée sujet.

- Les modules sont des briques de base et ne représentent pas, en général, des programmes complets ni exécutables. Un système est défini comme un module complet qui peut donc s'exécuter de façon indépendante.

Un hyperespace permet d'identifier explicitement les préoccupations et les dimensions. Il vise à l'inclusion d'artefacts depuis toutes les étapes du cycle de vie d'un logiciel. L'approche par séparation multidimensionnelle des préoccupations est plus ambitieuse que les approches telle que la programmation orientée aspect. Elle est plus générale et ses objectifs sont plus larges. Cependant, beaucoup d'effort reste à fournir dans ce domaine pour atteindre tous ses objectifs.

1.7. Conclusion

La séparation des préoccupations est une approche visant à simplifier la conception et la production d'applications. Les préoccupations sont les motivations premières pour décomposer et organiser une application en éléments compréhensibles et facilement manipulables. Chaque acteur dispose des éléments relatifs à sa préoccupation et uniquement de ceux-ci. Ce découpage facilite les activités des différents acteurs du processus logiciel.

Aussi, la séparation des préoccupations fournit un support méthodologique de modélisation. Elle permet de décomposer la modélisation d'un système en plusieurs modèles dédiés à des préoccupations différentes. Associé à ce découpage, un processus d'intégration des différents modèles est requis. L'association du découpage et du processus d'intégration facilite la collaboration des différents acteurs d'un processus logiciel.

Au niveau implémentation, la programmation orientée aspect apporte une solution élégante et simple à ce problème. Elle permet d'implémenter chaque problématique indépendamment des autres, puis de les assembler selon des règles bien définies. La programmation orientée aspect promet donc une meilleure productivité, une meilleure réutilisation du code et une bonne adaptation du code aux changements.

La séparation des préoccupations fournit un support méthodologique de modélisation et de programmation. Elle doit, bien sûr, être accompagnée d'un processus d'intégration des différents artifacts[1] générés pour les différentes préoccupations. C'est le processus dit de composition ou de tissage (*weaving*). Ce support permet de gérer plus naturellement, et d'une façon modulaire, des intégrations complexes. Comme le montre Meslati, il est communément admis qu'une bonne séparation des préoccupations réduit la complexité des systèmes logiciels, facilite la réutilisation, améliore la compréhension et simplifie l'intégration des artefacts [Meslati 2006].

Remarque : Nous nous sommes limités, dans ce chapitre, à l'introduction des techniques de séparation des préoccupations ainsi que la programmation orientée aspect. Ces techniques répondent bien aux problèmes de prise en charge des aspects au niveau implémentation. Dès lors, une question importante est soulevée :

Est ce que les aspects interviennent uniquement au niveau implémentation ?

Plus précisément :

- Est ce que le concept aspect apparaît uniquement durant l'implémentation et non pas dans les phases premières comme l'analyse des besoins, l'architecture et la conception et les dernières phases comme les tests, maintenance et évolution.

- Si la réponse est non, alors comment prendre en charge ce concept durant ces autres phases ?

- Quelle est la structure de ce concept en dehors de l'implémentation ?

- Quelles sont les approches et les modèles à développer pour la prise en charge de ce concept ?

Dans le chapitre suivant, nous allons essayer d'apporter des éléments de réponse pour prendre en considération ce concept durant la phase de l'ingénierie des besoins.

CHAPITRE 2

Mise en Œuvre des Approches de Séparation des Préoccupations

2.1. Introduction

Une des solutions avancées pour résoudre le problème de l'enchevêtrement et de la dispersion du code des propriétés non-fonctionnelles rencontrées avec la programmation par aspects, consiste à découpler leur définition en suivant le principe de la séparation des préoccupations. Partant de ce principe, G. Kiczales et son équipe [Kiczales 1997] ont formalisé cette séparation ainsi que les différentes étapes de réalisation de l'application pour aboutir au paradigme de la programmation par aspect [Kiczales 1997]. Ce paradigme de programmation consiste à structurer une application en modules indépendants représentant chacun les définitions des différents aspects ou propriétés non-fonctionnelles.

Une application se décompose en deux parties. La première correspond à l'aspect de base. Celui-ci définit l'ensemble des fonctionnalités offertes par l'application. Spécifiquement, il décrit l'application. La seconde partie représente un ensemble d'aspects techniques où chacun définit les mécanismes qui régissent l'exécution de l'application. Ils sont tous indépendants les uns des autres et définissent chacun un mécanisme d'exécution particulier.

Ce type de construction modulaire d'application, nécessite bien évidemment une étape d'assemblage. En effet, chaque module étant indépendant des autres modules, il est nécessaire de les assembler pour construire l'application. Ce mécanisme d'assemblage est appelé composition ou encore le tissage (*weaving*).

Dans ce chapitre, nous présentons trois types de langages qui implémentent le concept aspect et assurent la séparation de préoccupations au niveau code, à savoir, AspectJ, HyperJ et ComposeJ.

2.2. AspectJ

Etant un langage de programmation orienté aspect, AspectJ est une spécification d'un langage aussi bien qu'un outil assurant le tissage des aspects et la compilation des codes sources [Xerox 2002]. AspectJ est une extension de Java permettant la programmation par aspects. Ainsi, un aspect en AspectJ n'est qu'une classe Java qui modélise une préoccupation transversale. En choisissant Java comme langage de base, AspectJ profite de l'ensemble des avantages de ce dernier, ce qui le rend facile à utiliser pour des programmeurs Java. Kiczales qualifie AspectJ d'extension «compatible» du langage Java dans la mesure où [Kiczales 2001] :

- Chaque programme valide Java est un programme valide AspectJ,
- Tous les programmes valides AspectJ sont capables de s'exécuter sur n'importe quelle machine virtuelle Java,
- Il est possible d'étendre l'ensemble des outils Java pour supporter AspectJ, ceci inclut les IDE, les outils de documentation et les outils de conception,
- Le style de programmation avec AspectJ est très proche de celui de la programmation avec Java.

Dans ce qui suit, nous détaillons tout d'abord, l'ensemble des concepts fondamentaux de ce langage à aspects. Pour concrétiser ces notions et mieux comprendre comment réaliser une application en utilisant des aspects, nous considérons des exemples de programmation par aspects avec AspectJ. Nous présentons ensuite, les aspects et leurs structures. Pour terminer, nous décrivons brièvement le principe de recomposition adopté par ce langage à aspect.

2.2.1 Les concepts de base d'AspectJ

AspectJ étend Java tout en supportant la modularisation, dans des aspects, de deux types d'implémentation de transversalité. La première implémentation assure la définition des perfectionnements, qui doivent être exécutées à des points bien définis, du flot d'exécution de l'ensemble des composants du système. Il s'agit ici, des transversalités dynamiques *(dynamic crosscutting)*. La deuxième implémentation, permet de définir des nouvelles structures et méthodes sur des types déjà existants. Il s'agit ici, des transversalités statiques *(static crosscutting)* du fait qu'elles affectent les signatures statiques des différents composants constituant le système. Pour ce faire, AspectJ offre un certain nombre de concepts dont, les points de jointures (*Join Points*), les points de coupe (*Pointcuts*) et les consignes (*Advices*) pour l'implémentation des transversalités dynamiques, et les introductions (*Introductions*) pour l'implémentation des transversalités statiques.

Pour mieux comprendre toutes ces notions, considérons comme exemple d'application, un *Simple Editeur de Figures* (SEF). Une figure consiste en un certain nombre d'éléments de figure, qui peuvent être des points ou des lignes. Pour éditer ces figures, notre éditeur met en jeu un afficheur d'images. La conception objet de cette application nous conduit à définir les classes : Figure, ElementDeFigure, Point et Ligne. Sans oublier, bien sûr, la classe Display représentant l'afficheur. La figure 2.1 montre le diagramme de classes UML de cet éditeur de figures [Booch 1999].

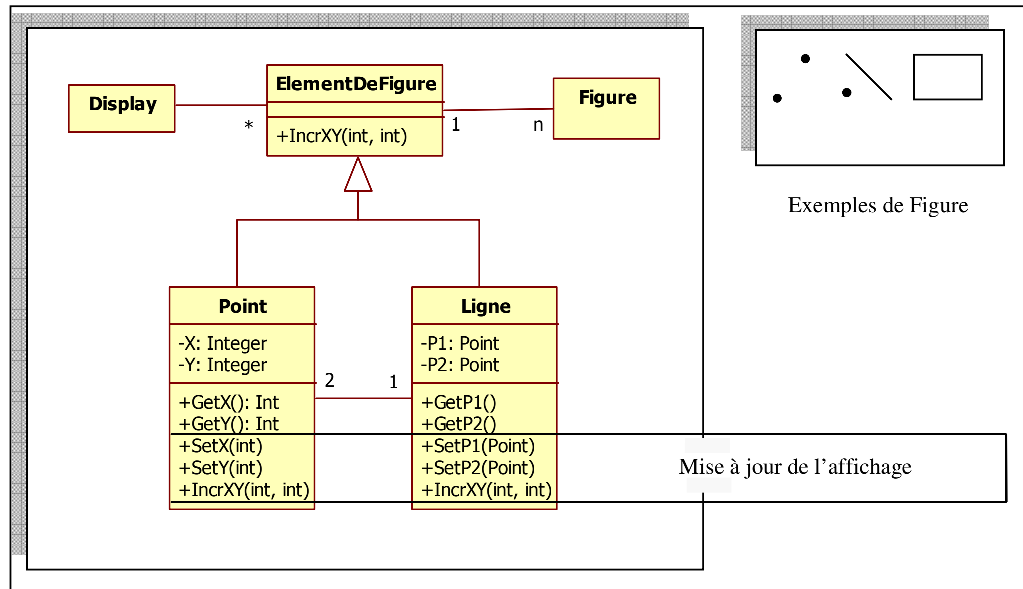


Figure 2.1 : Diagramme de classes UML d'un SEF

La méthode IncrXY() définit par la classe ElementDeFigure permet de déplacer un élément d'une figure en incrémentant ses coordonnées. La zone intitulée « *Mise à jour de l'affichage* » définit une préoccupation transversale en interaction avec l'ensemble des méthodes de déplacement, appartenant aux composants fonctionnels Point et Ligne de cette application. Il s'agit ici d'une préoccupation de transversalité dynamique. Nous traitons cet aspect en détail dans les prochaines sections.

2.2.1.1. Les points de jointures (*Join Point*)

A tout moment durant l'exécution, le système passe d'un état à un autre et génère des faits et des évènements tels que l'accès ou la modification d'un champ d'un objet, l'appel d'une méthode, son exécution et le retour de valeurs, etc. Tous ces points précis observables durant l'exécution d'un programme sont appelés *points de jointures* (*join points*) et représentent des points d'interruption et d'exploitation où l'aspect peut intervenir. Tout se passe comme si l'exécution du code fonctionnel est

interrompue aux points de jointures pour laisser la possibilité au code de l'aspect de s'exécuter et de réaliser le but de la préoccupation qu'il implémente.

Nous illustrons ces points à travers notre exemple d'application. Supposons que nous avons à exécuter le programme suivant [Kiczales 2001, Xerox 2002] :

```
Point p1 = new Point ( 0, 0 ) ;
Point p2 = new Point ( 4, 4 ) ;
Ligne ln = new Ligne ( p1, p2 ) ;
ln.incrXY ( 3, 6 ) ;
```

L'exécution des trois premières lignes de ce code crée les trois objets p1, p2 et ln montrés sur la figure 2.2 (représentés par les grands cercles). Dans cette figure, les petits rectangles représentent l'ensemble des méthodes offertes respectivement par p1, p2 et ln.

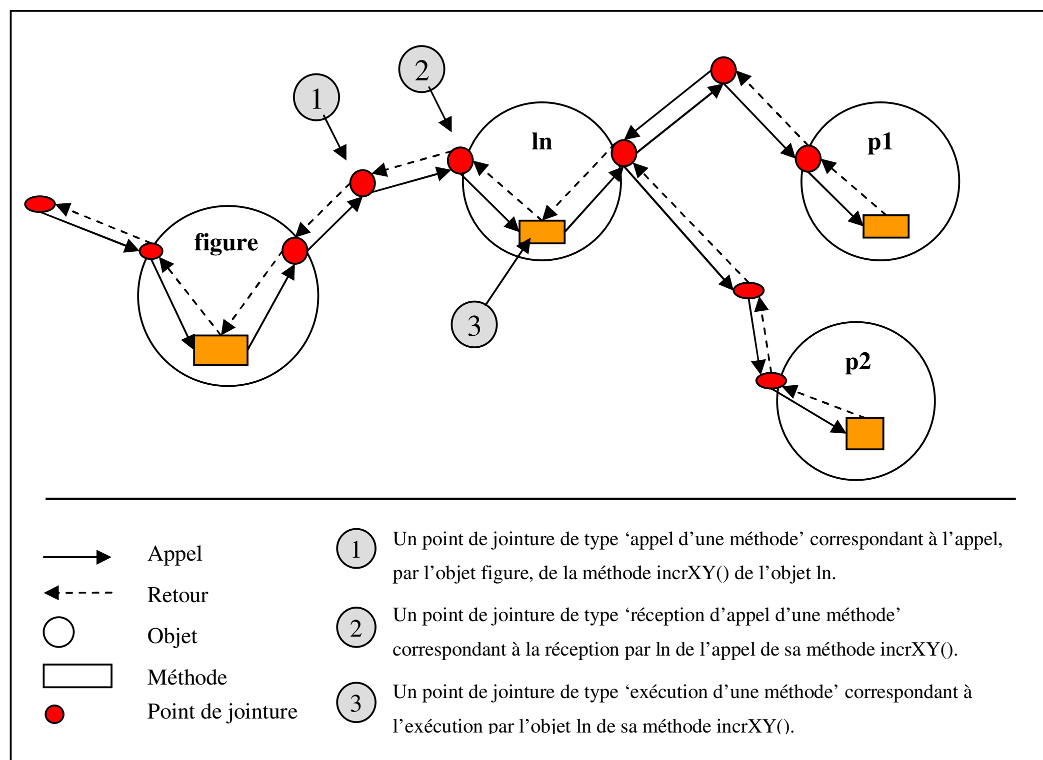


Figure 2.2 : Points de jointure dans l'application SEF

L'exécution de la dernière ligne de ce code enchaîne des appels et des exécutions de plusieurs méthodes, propres à la ligne ln et à ses deux points p1 et p2, et qui correspondent à une séquence de points de jointure, dont la figure 2.2 montre quelques-uns. Comme illustré sur la même figure, un point de jonction peut être un

simple appel à une méthode (1), une réception d'un appel à une méthode (2) ou encore une exécution d'une méthode (3). Le tableau suivant (table 2.1) résume les différents types de point de jointure offerts par AspectJ [Meslati 2006].

<table>
<tr><th>Point de jointure</th><th>Valeur du prédicat correspondant
(cjp : signifie point de jointure courant)</th><th>Commentaire</th></tr>
<tr><td>Call(S)</td><td>Vrai si cjp correspond à un appel de S</td><td rowspan="2">S ::= ResultatType e.g. ClassName.MethodName (Paramètres)</td></tr>
<tr><td>Execution(S)</td><td>Vrai si cjp correspond à une exécution de S</td></tr>
<tr><td>Get(S)</td><td>Vrai si cjp correspond à un accès à S</td><td rowspan="2">S ::= Type ClasseName.FieldName</td></tr>
<tr><td>Set(S)</td><td>Vrai si cjp correspond à une affectation de S</td></tr>
<tr><td>Initialisation(S)</td><td>Vrai si cjp correspond à l'exécution de l'initialisation d'un objet de S</td><td rowspan="2">S ::= ClassName(Parametres)</td></tr>
<tr><td>Preinitialiation(S)</td><td>Vrai si cjp correspond à l'exécution du pré initialisation d'un objet de S</td></tr>
<tr><td>Staticinitialisation(S)</td><td>Vrai si cjp correspond à l'exécution de l'initialisation de la classe S</td><td>S ::= ClassName</td></tr>
<tr><td>Handler(TP)</td><td>Vrai si cjp correspond à la manipulation de l'exécution TP dans un ploc de capture d'exécution</td><td>TP spécifie le type d'exécution</td></tr>
<tr><td>Within(TP)</td><td>Vrai si cjp correspond l'exécution d'un code appartient à TP</td><td>TP est le nom d'une classe</td></tr>
<tr><td>Withincode(TP)</td><td>Vrai si cjp correspond l'exécution d'un code défini dans une méthode ou un constructeur spécifie par S</td><td>S est une méthode ou une signature d'un constructeur</td></tr>
<tr><td>Cflow(P)</td><td>Vrai si cjp est dans le flot de contrôle du point de jointure défini par P (incluant P lui-même)</td><td rowspan="2">P est un point de coupure</td></tr>
<tr><td>Cflowbelow(P)</td><td>Vrai si cjp est dans le flot de contrôle bas du point de jointure défini par P (excluant P lui-même)</td></tr>
<tr><td>Adviceexecution()</td><td>Vrai si le corps en exécution appartient à une consigne</td><td></td></tr>
<tr><td>This (TP or Id)</td><td>Vrai si cjp correspond l'exécution d'un code appartenant à l'objet défini par TP ou Id (l'objet étant l'objet courant référencé par This dans Java)</td><td rowspan="3">TP est un nom de classe et Id un identificateur

'..' remplace n'importe quel nombre de paramètres</td></tr>
<tr><td>Target(TP or Id)</td><td>Vrai si la cible du cjp est un objet spécifie par TP ou Id</td></tr>
<tr><td>Args(TP or Id or '..')</td><td>Vrai si les arguments du cjp sont des instances dont le type est spécifié par TP ou Id</td></tr>
<tr><td>If(BoolExp)</td><td>Vrai si BoolExp est vrai</td><td>BoolExp est une expression booléenne</td></tr>
<tr><td>! P</td><td>Vrai si P n'est pas satisfait</td><td rowspan="4">P, P1 et P2 sont des points de coupures</td></tr>
<tr><td>P1 && P2</td><td>Vrai si P1 et P2 sont satisfaits</td></tr>
<tr><td>P1 || P2</td><td>Vrai si P1 ou P2 ou les deux sont satisfaits</td></tr>
<tr><td>(P)</td><td>Vrai si P est satisfait</td></tr>
</table>

Table 2.1 : Signification prédicative des points de jointure primitifs et leur composition dans AspectJ

2.2.1.2 Les points de coupes (*Pointcut*)

Parmi tous les points de jointures, seul un sous-ensemble peut intéresser un aspect à un moment donné. Le code source de ce dernier, spécifie les points de jointures qui lui sont significatifs en utilisant des *points de coupes* (*pointcuts*). Les points de jointures désignés par les points de coupes sont utilisés pour composer chaque aspect avec les composants. Ainsi, un point de coupes est une collection de points de jointure et, optionnellement, de quelques paramètres précisant le contexte d'exécution au niveau de ces points de jointures [Kiczales, 2001]. Pour identifier ces points de jointures, un point de coupes utilise plusieurs désignations connues sous le nom de désignateur de points de coupes (*pointcut designators*). Il combine ensuite ces points de jointures grâce à un ensemble d'opérateurs booléens : *and* ('&&'), *or* ('||') et *not* ('!'), pour former sa collection. Nous distinguons, principalement, deux types de points de coupes :

- **Point de coupes nommé :** il s'agit, dans ce cas, d'un point de coupes défini par un utilisateur (figures 2.3 et 2.4).

- **Point de coupes anonyme :** comme, par exemple, les points de coupes primitifs de la figure 2.5.

Pour mieux comprendre les notions introduites ci-dessus, examinons quelques exemples de points de coupes.

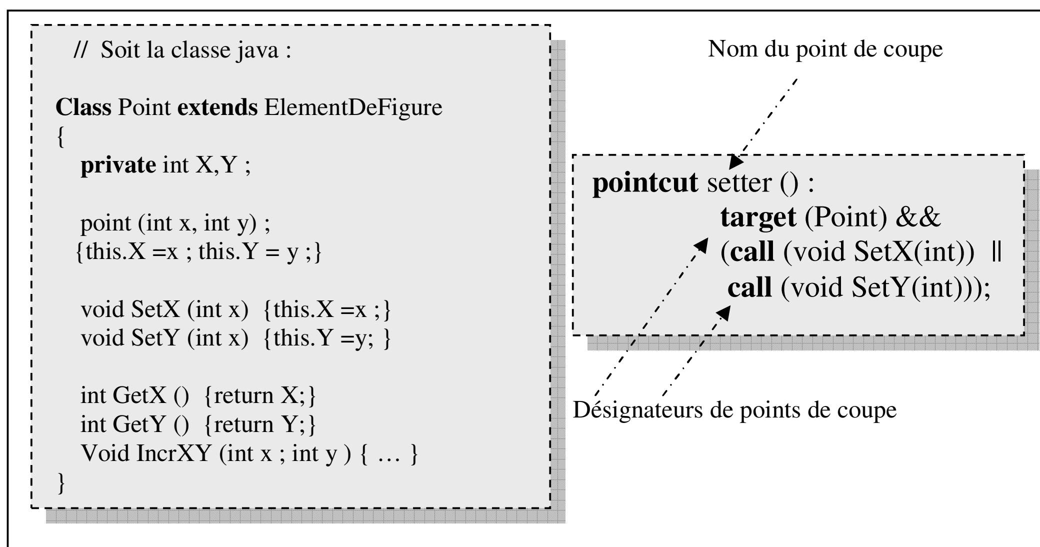


Figure 2.3 : Point de coupe nommé sans paramètres

Dans l'exemple de la figure 2.3, il s'agit d'un point de coupe nommé. Ce point de coupe, appelé *setter*, identifie deux points de jointures de type appel de méthodes SetX( ) ou SetY( ), de n'importe quelle instance de la classe Point.

Nous distinguons aussi deux types de désignateurs de points de coupes : *Call* pour identifier les deux points de jointures correspondants aux appels et *Target* pour spécifier le type de l'appelé (dans notre cas, il s'agit du type Point).

La figure 2.4 reprend le même exemple, mais cette fois avec des paramètres. Les points de jointures y sont définis dans un contexte d'exécution bien déterminé. Il s'agit ici d'appels à des méthodes d'une instance particulière 'p' avec le paramètre '*newval*' de type *int*.

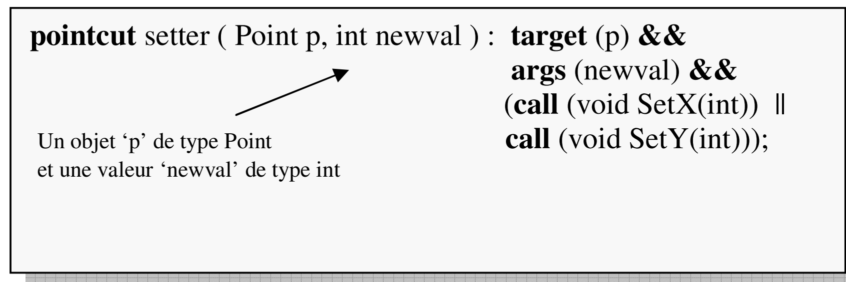


Figure 2.4 : Point de coupe nommé avec paramètres

Nous distinguons aussi des points de coupes primitifs qui peuvent être composés par les opérateurs '&&', '||' ou '!', donnant lieu ainsi à des points de coupes plus complexes comme le montrent les exemples de la figures 2.5. Il est possible aussi d'utiliser des '*wildcards*' comme par exemple '*', remplaçant n'importe quel caractère.

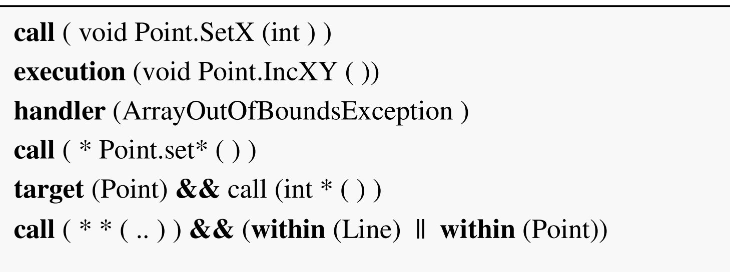


Figure 2.5 : Points de coupes primitifs et composés

Plusieurs aspects peuvent exister à un moment donné dans un système et être concernés par les mêmes points de jointures et les mêmes points de coupes ; ce qui engendre des situations de conflits entre aspects. Dans toutes les implémentations orientées aspects, ces conflits doivent être résolus à l'avance pour ne pas créer des situations d'exécutions insolubles (*deadlock*). Dans AspectJ, les aspects en conflit sont exécutés suivants des règles de précédences prédéfinies à l'avance par le programmeur.

2.2.1.3 Les consignes (*Advice*)

Le code de l'aspect est divisé en plusieurs blocs dits *consignes*. Ces blocs ressemblent aux méthodes implémentées dans une classe Java et sont utilisées pour déclarer un code qui ne doit s'exécuter qu'une fois que les points de jointures d'un point de coupe soient atteints.

Il existe trois types de relations entre points de coupes et consignes : *before*, *after* et *around* (qu'on peut traduire respectivement par : *avant*, *après* et *autour*). Si un point de jointure est un appel à une méthode *m()*, les aspects concernés doivent contenir un point de coupe qui spécifie ce point de jointure (appel de méthode) et une consigne qui s'exécutera avant ou après l'appel de *m()* ou même autour de cet appel. Ce dernier cas signifie que la consigne peut remplacer l'appel de la méthode comme elle peut ajouter un traitement à effectuer avant et un autre à effectuer après cet appel. Pour cette dernière situation, une instruction spéciale appelée *proceed(),* disponible uniquement à l'intérieur d'une consigne *around,* peut relancer l'exécution du traitement associé aux points de jointures (cf. figure 2.6).

```
around ( ) : call (Display.update ( ) )
{
        if  ( ! Display.disabled ( ) )
                proceed ( ) ;
}
```

Figure 2.6 : Consigne de type around

L'appel à la méthode *proceed ()* dans le corps de la consigne *around* exécute le code original défini au niveau du point de jointures approprié (il s'agit ici du code défini par la méthode update ( ) de la classe Display).

2.2.1.4 Les Introductions

Les introductions dans AspectJ sont des constructions qui permettent de déclarer, dans un aspect, de nouveaux membres qui seront insérés dans des classes déjà définies. Ils permettent d'ajouter des attributs et des méthodes à des classes existantes [Xerox 2002]. La figure 2.7 montre quelques exemples de déclaration d'introductions.

```
Public String Point.name ;
Public void Point.setName (String name)
{
    this.name = name ;
}
public String ( Point || Ligne ). GetName ( )
{
    return name ;
}
```

Figure 2.7 : Déclaration d'introductions ajoutant des attributs et des méthodes

Les déclarations d'introductions permettent aussi de modifier la hiérarchie d'héritage des classes.

2.2.2. L'héritage d'aspect

L'héritage d'aspect a été introduit pour deux raisons :

- Il peut être intéressant de construire des hiérarchies d'aspects avec par exemple des aspects abstraits, pour les mêmes raisons que dans l'approche objet, à savoir, la capitalisation et la réutilisation.
- Les concepteurs d'AspectJ se sont rendus compte que leur langage ne permettait pas la séparation entre les points de coupes et les consignes d'aspects. De ce fait, il est intéressant de disposer de bibliothèques de consignes d'aspects indépendantes et intégrables avec n'importe quel point de coupes. L'héritage d'aspect permet de créer des aspects abstraits ne contenant que des consignes d'aspects. Ces aspects abstraits sont, ensuite, spécialisés par héritage et complétés par la définition des points de coupes.

2.2.3. La recomposition dans AspectJ

Après avoir défini l'ensemble des concepts fondamentaux d'AspectJ ainsi que la structure d'un aspect dans AspectJ, nous nous intéressons ici à la manière dont ce langage à aspects recompose l'ensemble des aspects et des classes pour obtenir l'application finale.

Pour assurer la recomposition dans la programmation orientée aspects, nous avons besoin de mécanismes assurant le tissage des aspects et la compilation des

codes sources, pour obtenir le code exécutable de l'application finale. Particulièrement, dans le cas d'AspectJ, la recomposition fusionne ces deux activités [Kiczales 2001]. En effet, l'implémentation d'AspectJ fournit, uniquement, un compilateur qui assure le tissage et la compilation en même temps. Il procède en trois étapes pour aboutir au code final et transformer le tout en un code exécutable. Dans un premier temps, et après avoir compilé chaque partie du programme qui n'est pas affectée par les aspects (en utilisant le compilateur standard Java), il effectue la compilation des corps des consignes de l'ensemble des aspects ; ensuite il fait correspondre les différentes méthodes des aspects (i.e. leurs consignes) et des composants. Enfin, il produit le code exécutable en respectant la composition des points de coupes utilisant des désignateurs de points de coupes comme *cflow* spécifier par la table 2.1.

2.2.4. La représentation UML des aspects

Plusieurs tentatives d'intégration des concepts orientés aspect dans l'univers d'UML ont été entreprises, soit par la définition d'opérations d'extension du modèle UML lui-même [Mostefaoui 2005], soit par la définition de profil UML propre à l'orienté aspect [Aldawud 2003] quoique cette dernière solution n'a pas réussi à faire monter en surface les concepts de l'approche orientée aspect. Dans cette optique, en considérant qu'un aspect n'est, en réalité, qu'une simple classe dans le sens du paradigme objet. Ainsi nous avons opté pour la première alternative afin intégrer les concepts orientés aspect dans un méta modèle comme extension de celui défini dans UML. La figure 2.8 montre un extrait dans lequel nous trouvons les éléments les plus importants.

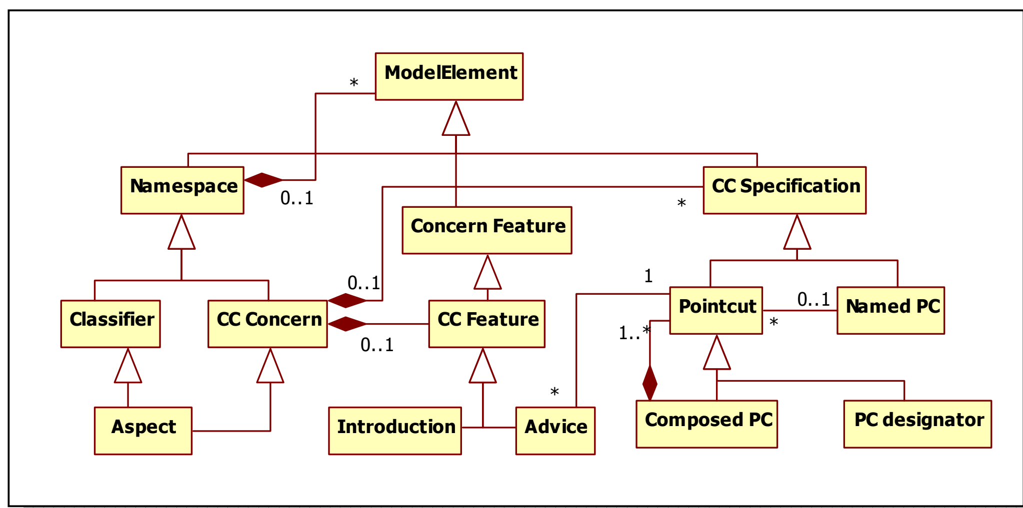


Figure 2.8 : Syntaxe abstraite du méta-modèle spécifique à Aspect J

Dans ce méta-modèle, un aspect est la classe des préoccupations transversales d'AspectJ, c'est une spécialisation de (*Crosscutting Concern,* CC). AspectJ définit un aspect d'une façon similaire aux classes en Java : un aspect peut contenir des définitions de propriétés (structurelles et/ou comportementales) qui lui sont propres, mais aussi des définitions de propriétés d'entrecroisement (*Crosscutting Feature*). Un aspect peut contenir de plus, des spécifications transversales (*Crosscutting Specification*). Par ailleurs, un aspect peut hériter d'un autre aspect ou d'une classe, il peut être abstrait comme il peut implémenter une interface. De plus, un aspect peut jouer le rôle d'une préoccupation de base concernée par un ou plusieurs aspects. Comme illustré sur la figure 2.8, la classe Aspect est également une spécialisation de la classe Classifier [Amirat 2007b].

Ce méta-modèle permet de faire la projection (*mapping*) entre AspectJ et les différents méta-modèles qui peuvent exister (e.g. méta-modèle de HyperJ). Bien évidemment, ce transfert est toujours conditionné par le passage via le méta-méta modèle MOF (*Meta Object Facility*) d'UML en utilisant les outils définis dans le cadre de l'OMG comme QVT (*Query, View, Transformation*) [Bezivin 2006].

2.3. Les filtres de composition

L'approche des filtres de composition est construite autour du paradigme orienté objet. Elle consiste à ajouter à un objet une couche enveloppante appelée interface qui intercepte les messages entrants et sortants. Un objet doté d'une interface est appelé CF-objet. Chaque CF-Objet est défini par un CF-Modèle (voir chapitre 1).

Chaque filtre (d'entrée ou de sortie) peut accepter ou rejeter un appel selon la sémantique associée à son type. Accepter un appel peut signifier dispatcher ou simplement ignorer le message. Dans ce dernier cas, il passe au filtre suivant.

Rejeter un appel peut signifier écarter le message, le mettre en attente dans une file tant que l'expression associée au filtre résulte en un rejet, ou alors simplement l'ignorer (i.e. le message passe alors au filtre suivant).

2.3.1. Implémentations du CF-modèle

Depuis la publication du modèle de composition de filtres plusieurs implémentations ont été réalisées. Mais elles défèrent les unes des autres suivant :

- Le langage de base adopté pour décrire l'implémentation,
- Le niveau d'intégration avec le langage de base,

- La nature de la traduction (interpréteur pur, interpréteur du byte code ou compilateur)

Parmi les implémentations les plus importantes, nous pouvons citer :

❖ **Sina.**

Le langage *Sina* représente une implémentation partielle des filtres de composition (seul un ensemble restreint de filtres est supporté – *Dispatch* et *Error*). Ce langage génère un byte code interprété par une machine virtuelle construite comme une couche sur Smalltalk [Tripathi 1989].

❖ **C++/CF.**

Le langage *C++/CF* est le résultat de l'intégration des filtres de composition avec le langage C++. Il est basé sur la réification des invocations de messages. Cette réification est implémentée sous forme de Macros [Glandrup 1995]. Notons que cette implémentation est aussi partielle.

❖ **ComposeJ.**

Le langage *ComposeJ* est le résultat de la combinaison du modèle des filtres de composition et du langage Java. Ce langage peut être utilisé très facilement comme add-on du compilateur java standard [Wichman 1999].

❖ **ConcernJ.**

Le langage *ConcernJ* est la mise en oeuvre des filtres de composition en utilisant un mécanisme de super-imposition[1] des filtres. Comme ConcernJ est considéré comme un pré-processeur pour ComposeJ alors il supporte le même ensemble de filtres de composition que ce dernier. ConcernJ permet de résoudre les conflits et génère la spécification des filtres correspondants pour ComposeJ [Salinas 2001].

---

[1] Cette technique permet, de manière modulaire, d'imposer des filtres à des ensembles de classes, plutôt qu'à une classe seulement.

2.3.2. Types de Filtres

Pour qu'une implémentation soit complète, elle doit prendre en compte les différents types de filtres définis par le CF-modèle. Six types de filtres sont définis dans le CF-Modèle, à savoir (*Error*, *Meta, Wait*, *Dispatch, Substitute* et *Realtime*). Nous trouvons plus de détails sur leur description ainsi que leur utilisation dans [Meslati 2006]. Chaque type est utilisé pour des besoins spécifiques dans l'expression des préoccupations. En générale, un ensemble de filtres contient des filtres de deux ou plusieurs types. Chaque filtre peut accepter ou rejeter un message. La table 2.2 résume l'ensemble de ces filtres avec leur description [Aksit 1992].

Type de Filtre	Actions exécutées	Description
Error	Acceptation	Le message accepté passe au filtre suivant dans l'ensemble des filtres
	Rejet	Une exception est générée (déclenchée)
Meta	Acceptation	Si le message est accepté alors il sera réifié (i.e. le sélecteur de la méthode et l'objet cible deviennent accessibles) et envoyé comme un paramètre d'un nouveau message à un objet déterminé
	Rejet	Le message doit continuer au filtre suivant. L'objet qui reçoit le message peut observer et manipuler le message réifié et réactiver son exécution
Wait	Acceptation	Le message accepté passe au filtre suivant, Aucune substitution ou délégation n'aura lieu.
	Rejet	Le message rejeté est bloqué jusqu'à ce que la condition, correspondant à la *matching part* qui s'est appariée avec le message, devienne vrai.
Dispatch	Acceptation	Si le message est accepté, il est expédié à la cible courante du message
	Rejet	Le message continue au filtre suivant (s'il n'y a rien, une exception sera déclenchée)
Realtime	Acceptation	Les attributs temporels du message accepté sont chargés et le message passe au filtre suivant
	Rejet	Le message rejeté passe au filtre suivant dans l'ensemble des filtres
Substitute	Acceptation	Certaines propriétés du message accepté sont modifier ou substituer
	Rejet	Le message rejeté passe au filtre suivant dans l'ensemble des filtres

Table 2.2 : Type de filtres dans le CF-Modèle

Bien que le modèle des filtres de composition avec la super-imposition apparaît comme prometteur, aucune implémentation actuelle ne mette complètement en œuvre ce modèle. Ainsi, aucune utilisation concrète pour la séparation des préoccupations n'est à priori envisageable et par conséquent aucun langage de programmation ne sera pas abordé en détail dans ce chapitre.

2.4. HyperJ : Concepts de base

HyperJ est une extension de Java qui implémente le principe des hyperespaces en opérant au niveau des bytecodes Java. Nous décrivons ci-après les concepts clés des hyperespaces [Ossher 1999].

Les hyperespaces (*Hyperspace*) est une approche qui permet l'identification explicite de chaque dimension et de chaque préoccupation, à n'importe quelle étape du développement, l'encapsulation de ces préoccupations, la gestion des relations entre ces préoccupations et leur intégration. Dans cette approche, l'accent et mis sur quatre concepts principaux : préoccupation (*concern*), hyperespace (*Hyperspace*), hypermodule (*hypermodule*) et hypertranche (*hyperslice*).

Un système logiciel se compose de plusieurs parties qui sont formulées dans un langage donné. Une unité est une construction syntaxique dérivée d'un tel langage. Elle peut être, par exemple, une méthode, une variable d'instance, une procédure, une fonction, une règle, une classe, une interface, etc. Cependant, on distingue les unités simples non décomposables et les unités composées qui sont des regroupements d'unités simples.

L'espace des préoccupations renferme toutes les unités d'un système logiciel donné. L'objectif est alors de pouvoir organiser les unités de manière à former et à séparer les préoccupations importantes, décrire plusieurs types de relations et indiquer comment le système, et éventuellement ses composants, peuvent être construits et intégrés à partir des unités de ces préoccupations.

Le processus de construction d'un système comprend trois étapes :

- **Identification :** c'est le processus de sélection des préoccupations ainsi que les unités que ces préoccupations renferment.

- **Encapsulation :** consiste à regrouper des unités pour former des préoccupations manipulables en tant qu'entités de première classe. Une classe Java est un exemple d'une préoccupation encapsulée.

- **Intégration :** Une fois les préoccupations encapsulées, on les intègre pour créer le système final.

L'identification des préoccupations se fait à partir d'une matrice multidimensionnelles dont chaque axe représente une dimension de préoccupation et chaque position sur cet axe une préoccupation donnée. Les dimensions permettent une répartition de toutes les préoccupations en groupes. Chaque élément de la matrice est une unité. Ceci rend explicite l'identification de toutes les dimensions d'intérêt, les préoccupations qui appartiennent à chaque dimension et quelles sont les préoccupations qui sont affectées et par quelles unités. Les coordonnées d'une unité indiquent toutes les préoccupations qu'elle affecte.

L'expérimentation du langage HyperJ est restée toujours dans un cadre académique est beaucoup plus limitée aux laboratoires d'IBM qui sont l'origine de ce langage basé sur la séparation multi-dimensionnelles des préoccupations et c'est aussi la raison pour laquelle nous avons décidé de ne pas entrer plus dans les détails de ce langage de programmation (HyperJ) [Ossher 2001].

2.5. Modèle de transformation entre langages orientés aspect

Les mécanismes et concepts introduits dans le cadre de l'approche Aspect permettent une représentation explicite et séparée des préoccupations (*SoC*) : ils améliorent leur traçabilité et augmentent la réutilisation des programmes. Mais ils restent des solutions dépendantes d'un langage de programmation par aspects. Par contre, actuellement nous avons besoin d'un cadre conceptuel pour exprimer des structures par aspects qui soient indépendantes d'un langage de programmation donné. Ainsi, plusieurs modèles et langages de programmation associés sont apparus qui sont différents de par leurs concepts et mécanismes orientés aspect.

Le besoin d'un modèle général à l'approche Aspect exige :

- Un rapprochement des concepts et mécanismes des divers modèles et langages de programmation par aspects, par la définition de méta-modèles spécifiques à ces langages.
- Définition d'un méta-modèle général pour l'approche aspect.

L'idée de base pour réaliser la transformation, est de faire une projection d'un modèle instance du méta-modèle général vers un modèle instance d'un méta-modèle spécifique (voir section 2.2.4, figure 2.8). Ainsi, par cette projection nous obtenons des concepts très proches, mais trop spécifiques. Par conséquent, le besoin d'une spécification de transformation pour une projection complète est indispensable [Bezivin 2006].

La transformation dirigée par les modèles est l'une des possibilités qui permet de passer d'un langage orienté aspect vers un autre. Nous proposons dans cette section,

un processus de transformation inspiré d'une étude que nous avons effectuée dans le cadre de l'évolution des architectures logicielles à base de composants (figure 2.9). Ce processus de transformation par annotation basée sur les trois méta-modèles suivants :

1. **Modèle général :** modèle instance du méta-modèle général, il représente la structure par aspects du logiciel, partie (a) figure 2.9.

2. **Modèle intermédiaire :** basé sur le premier modèle, mais doté en plus d'annotations (*tagged values*) spécifiques à la projection cible, partie (b) figure 2.9.

3. **Modèle spécifique :** modèle instance du méta-modèle spécifique cible, il représente la structure par aspect du logiciel, partie (c) figure 2.9.

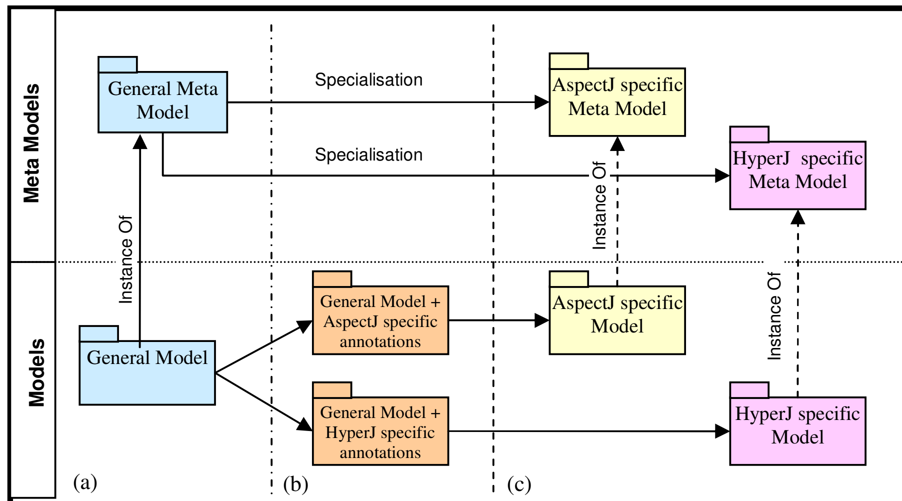


Figure 2.9 : Processus de transformation entre langages (e.g. AspectJ / HyperJ)

2.6. Conclusion

Dans ce chapitre, nous avons montré que la complexité du code source d'un système à objets, résulte, principalement, de l'enchevêtrement et de la dispersion du code des aspects d'une application, dans la description de ses composants. Les composants représentent les propriétés fonctionnelles du système, dont l'implémentation peut être proprement encapsulée dans une procédure généralisée. Alors que, les aspects correspondent à l'ensemble des propriétés transversales du

système, dont l'implémentation ne peut pas être proprement encapsulée dans une procédure généralisée. Ils représentent des préoccupations transversales qui touchent à l'ensemble des composants fonctionnels de l'application.

Ce paradigme de programmation garantit une meilleure abstraction de l'application en modules indépendants, qui représentent les définitions des composants et des aspects. L'application finale est produite par la composition de ces modules, grâce à un préprocesseur particulier appelé tisseur d'aspects. La différence principale entre ce nouveau paradigme et les autres approches de programmation traditionnelles, est que l'AOP fournit des unités de modularisation et des langages à aspects avec différents niveaux d'abstraction et différents mécanismes de recomposition. Comme exemple de mise en oeuvre de ce nouveau paradigme de programmation par aspects, nous avons expérimenté dans le cadre de ce travail, une extension compatible du langage Java par aspects, AspectJ.

A ce jour, la programmation par aspects a été expérimentée pour réaliser plusieurs applications réelles. Bien que ces expérimentations aient montré l'aptitude de l'AOP à faciliter l'évolution et la réutilisation de ces applications, ils n'en reste pas moins que les mises en oeuvre existantes n'ont pas résolu toutes les difficultés que soulève ce paradigme. De plus, de nombreux problèmes sont apparus, comme, par exemples, la composition des différents aspects en *conflits* (avec un composant fonctionnel à travers un même point de jointure) ou encore la réutilisabilité des aspects d'une application. Même si de nombreuses questions restent encore ouvertes, il n'en demeure pas moins que nous sommes convaincus que la programmation par aspects connaîtra dans les années à venir un essor semblable à celui qu'a connu la programmation par objet.

Ainsi, dans le chapitre trois, nous allons étudier tous les concepts et approches de l'orientés aspect vis-à-vis de la phase de l'ingénierie des besoins. Nous qualifions cette étude d'importante du moment qu'elle conditionne dès le départ le succès ou l'échec des futurs systèmes à développer.

CHAPITRE 3

Ingénierie des Besoins Orientée Aspect

3.1. Introduction

Actuellement, dans le domaine de l'ingénierie des besoins, nous constatons qu'un certain nombre d'approches récentes non orientées aspect (non-AO) ont reconnu que les besoins non-fonctionnels sont caractérisés par leur large influence sur les autres besoins, alors que ces approches ne considèrent pas la large influence de certains besoins fonctionnels sur les autres besoins.

Toutes les approches non-AORE discutées par R. Chitchyan et *al.* dans [Chitchyan 2005a] ont une structure de modularisation rigide dès le début du processus d'analyse des besoins. Par exemple, les cas d'utilisation [Jacobson 1992] sont limités aux modules de cas d'utilisation, PREview [Sawyer 1996] est limitée aux points de vue et besoins organisationnels, Problem Frames est limité aux Frames, etc. Une telle rigidité peut être restrictive à certaines structures de modularisation. Par exemple, supposons qu'un analyste ait un ensemble de points de vue, des besoins non fonctionnels et des cas d'utilisation à traiter. Il doit se décider pour l'utilisation d'une seule structure de modularisation et limiter son analyse, ne prenant pas en compte les avantages offerts par les autres structures.

Chaque approche identifie, classifie et raffine les besoins à partir de différentes perspectives. Dans ce chapitre, nous allons étudier les approches les plus connues et les plus utilisées dans les lignes de productions de logiciels. Notons que la plus part des ces approches sont issues à partir de certains approches qualifiées de base dans ce domaine, à savoir, l'approche orientée points de vue, l'approche basée cas d'utilisation/scénario et l'approche orientée but (*Goal*). Dans la cinquième section de ce chapitre, nous présentons une étude comparative entre ces approches suivant un nombre de critères présélectionnés. La dernière section de chapitre sera consacrée à la définition des concepts fondamentaux des langages de description des besoins.

D'une manière générale, la phase d'ingénierie des besoins se préoccupe de la détection des buts, des fonctions et des contraintes réelles d'un système logiciel.

Cependant, l'ingénierie des besoins orientée aspect est concernée par la capture précoce des besoins transversaux au début du cycle de vie du développement logiciel. L'identification précoce des besoins a pour objectif d'améliorer la modularité dès la première phase du cycle de développement et évite l'enchevêtrement au niveau des phases ultérieures (conception et l'implémentation).

Les travaux de recherche sur l'utilisation des aspects au niveau de l'analyse des besoins sont toujours immatures. Il n'y a pas de consensus, ni sur la définition du concept aspect durant cette phase, ni sur comment il doit être transformé (*mapping*) vers des artefacts liés aux activités de développement postérieures. Très peu d'approches traitent efficacement les préoccupations transversales pendant l'analyse des besoins. Par conséquent, l'identification précoce améliore la qualité du produit et réduit les coûts de maintenance et augmente l'évolutivité du produit logiciel.

Les besoins sont toujours synthétisés à partir de certains problèmes métiers (e.g. on demande d'améliorer les systèmes de sécurité d'un modèle de voiture, ajouter une nouvelle fonctionnalité ou valeurs ajoutées aux téléphones portables, etc.). Les projets peuvent consister en un développement de nouveaux produits ou une évolution d'anciens systèmes. Dans tous les cas, le système logiciel sera incorporé dans un contexte opérationnel et un autre organisationnel. Donc, il aura des interfaces utilisateurs, des éléments du processus commercial ou autres systèmes logiciels ou matériels en interaction (voir figure 3.1).

Actuellement, nous pouvons représenter les besoins selon les trois modes suivants :

1. ***Informel*** : sous forme de texte généralement structuré suivant un canevas.
   e.g. Canevas – IEEE- std – 830.

2. ***Semi-formel*** : la représentation des besoins doit suivre un modèle.
   e.g. i* , KAOS, NFR Framework.

3. ***Formel*** : utilisation de notations mathématiques qui permettent de représenter en plus le raisonnement sur les besoins.

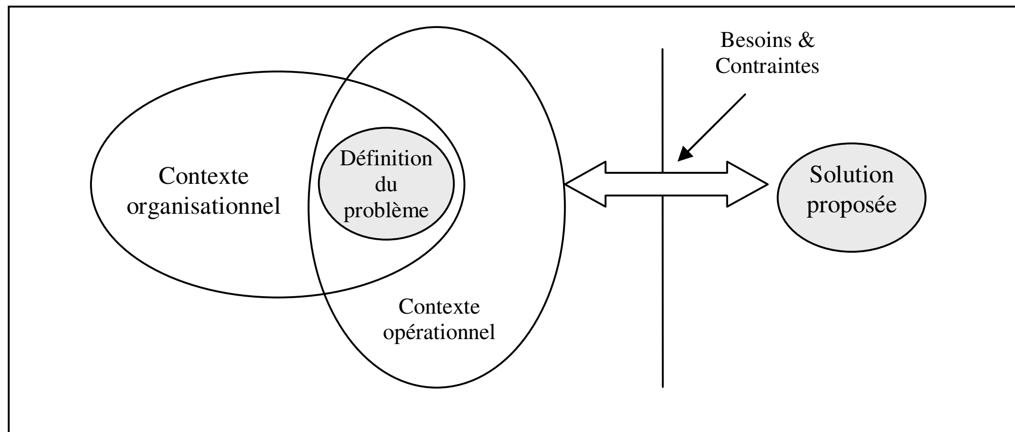


Figure 3.1 : Besoins, contraintes, problèmes et solutions dans l'ingénierie des besoins.

Vu que la représentation informelle utilise un langage naturel dont l'ambiguïté est inhérente, elle ne permet pas de détecter les conflits entre besoins. Par conséquent, cette représentation n'est pas très utilisée dans le développement de logiciels. Aussi la représentation formelle est difficile à appliquer dans un contexte dont l'ambiguïté est inhérente. Par conséquent, les académiciens et les praticiens se retrouvent, d'une manière logique, à développer et à appliquer beaucoup plus la représentation semi-formelle. Ceci se confirme par le nombre important et la variété des approches développées dans ce cadre. Ceci nous a conduit à se focaliser sur les approches semi-formelles de représentation et à s'en inspirer pour modéliser et développer notre travail dans ce domaine.

3.2. Définitions

Dans cette section, nous allons présenter les concepts et les définitions nécessaires pour la compréhension du domaine de l'ingénierie des besoins.

3 .2.1. Besoin (*Requirement*)

Un besoin représente la description des services attendus par le système pour répondre aux exigences de ses intervenants (*Stakeholders*). Chaque besoin possède un identificateur (id) unique dans sa portée (dans la préoccupation). Aussi, il doit avoir une partie de description textuelle[1] exprimant le besoin. D'une manière similaire, un besoin peut avoir des sous-besoins (Figure 3.2).

[1] Un besoin peut contenir une ou plusieurs phrases, qui ne sont pas numérotés, une phrase peut contenir une ou plusieurs clauses.

1.2.2. Préoccupation (*Concern*)

Tout système informatique peut être vu comme un ensemble de préoccupations. Une préoccupation est un module qui encapsule un ou plusieurs besoins (Figure 3.2). Une préoccupation peut être simple (contenant uniquement des besoins), ou composite (contenant d'autres préoccupations et des besoins).

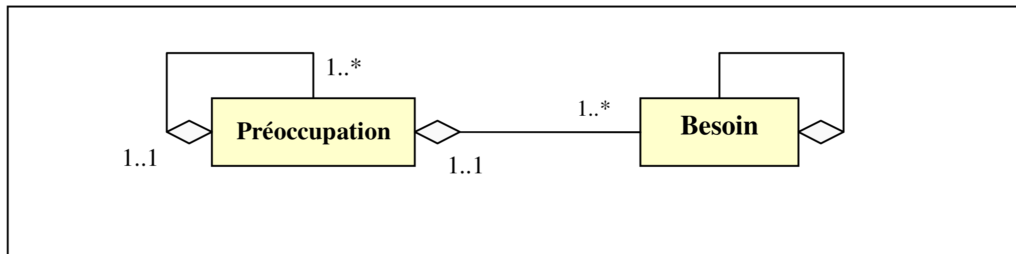


Figure 3.2 : Relation préoccupation et besoin

Nous présentons, dans les sections suivantes, les deux catégories d'approches développées dans le cadre de l'ingénierie des besoins, à savoir les approches non orientées aspect et les approches orientées aspects.

3.3. Approches non orientées aspects

Dans cette section, nous étudions le travail représentatif de plusieurs classes d'approches de l'ingénierie des besoins. Ces classes ont été choisies parce qu'elles sont à l'origine des travaux actuels dans ce domaine et elles ont servi de support de base pour les techniques émergentes de AORE. Pour chaque classe choisie, nous discutons quelques approches représentatives, examinons leurs méthodes générales, leurs artefacts et leurs processus. Parmi ces approches, nous nous intéressons aux approches suivantes :

- Approches orientées points de vue
- Approches orientées but
- Approches basées cas d'utilisation / scénarios
- Approches basées Problem Frames

3.3.1. Approche orientée points de vue

L'approche orientée points de vue considère les informations concernant le problème à partir des vues d'agents différents (e.g. utilisateurs du système logiciel) qui peuvent avoir différentes perspectives, souvent valables et incomplètes sur le problème. Ces perspectives partielles apparaissent en raison des responsabilités

différentes, des rôles, des buts ou des interprétations des sources de l'information. La combinaison de l'agent et de la vue qu'il a sur le système est désignée par point de vue (*viewpoint*). Cette approche ajoute la notion de préoccupation organisationnelle à la notion standard de point de vue.

3.3.1.1. Méthode PREview

Une préoccupation dans PREview est une généralisation de la notion de but. Elle inclut aussi bien les buts organisationnels que les contraintes qui limitent le système ou le processus à analyser. PREview a été développée pour capturer une large gamme de préoccupations comme le temps de réponse et la sécurité [Sommerville 1996] [Sommerville 1997].

Le processus PREview a été développé dans le cadre de cette approche. Les grandes lignes de ce processus sont présentées dans la figure 3.3. Le processus commence par l'établissement des préoccupations de haut niveau (e.g. la discussion avec la direction). Ces préoccupations sont alors élaborées sous une forme plus spécifique et sont présentées comme des besoins spécifiques (besoins externes), des contraintes ou des questions. Les questions aident à rassembler l'élément essentiel à partir des perspectives de chaque point de vue. Ce processus a été déjà utilisé dans des systèmes de petite et moyenne taille avec un niveau critique de sécurité.

L'approche a l'avantage de proposer un ensemble stable de concepts et une méthode fiable. Malgré l'avantage de la détection de l'inconsistance qui peut exister entre les besoins des points de vue et les préoccupations, l'approche n'identifie pas le transfert (*mapping*) ou l'influence des propriétés entrecoupantes sur les artefacts durant les phases ultérieures du développement.

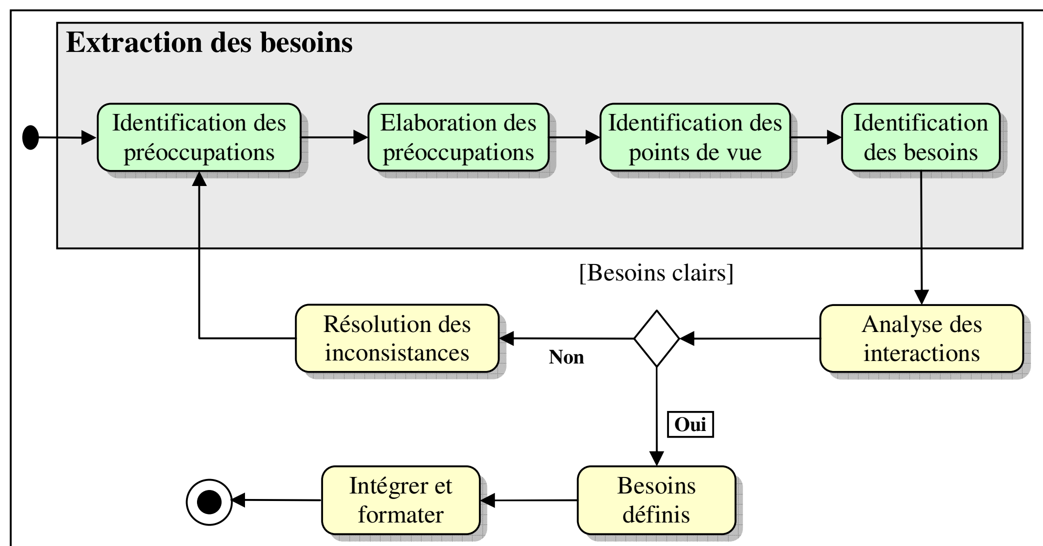


Figure 3.3 : Processus du modèle PREview

Malgré les efforts déployés pour raffiner l'approche PREview, certains problèmes sont encore non résolus. Parmi ces problèmes nous pouvons citer :

- Seulement un nombre réduit de préoccupations (non supérieur à 5) peut être efficacement traité dans chaque projet. Un grand nombre de préoccupations rend la quantité d'informations produites ingérables;
- De la même façon, seulement un nombre réduit de points de vue (inférieur à 6) doit être utilisé;
- Absence de directives claires pour la décomposition des préoccupations;
- Absence d'un mécanisme pour la gestion de l'inconséquence, l'analyse de compromis et le support pour la décision ;
- Cette approche peut-elle vraiment identifier les besoins fonctionnels entrecoupants?

Nous trouvons aussi, dans la littérature, une approche qui n'est pas très connue appelée VIM (*Viewpoints and Inconsistency Management*). Elle propose d'utiliser les points de vue comme mécanisme pour la gestion de l'inconsistance entre les spécifications partielles des besoins. Certainement, ces deux approches peuvent être complémentaires, avec PREview pour identifier les points de vue et VIM pour gérer les éventuelles inconsistances.

3.3.2. Approches Orientées But (Goal)

Un but (*goal*) est un objectif que le système en construction doit réaliser [Lamsweerde 2001]. Selon ce dernier, nous pouvons classer les buts selon deux niveaux :

- Buts de haut niveau : comme les buts stratégiques, globaux ou les buts couvrant l'organisation.
- Buts de bas niveau : comme les buts opérationnels, locaux ou spécifiques à la conception.

Les buts représentent les propriétés attendues et peuvent couvrir aussi bien les besoins fonctionnels que le nouveau système doit fournir que les besoins non-fonctionnels liés à sa qualité de service, comme la sécurité, les performances, le coût, l'adaptabilité, etc. À la différence des besoins, la réalisation d'un but peut exiger la coopération de plusieurs agents responsables.

Les buts peuvent être utilisés pour fournir une analyse sur les besoins, en évaluant la complétude des besoins et leur pertinence aussi bien que leur identification. Par exemple, un besoin est justifiable et pertinent s'il mène à la satisfaction d'un but. L'ensemble des besoins est complet s'il satisfait tous les buts considérés.

Dans les sections suivantes, nous présentons les approches principales développées dans le cadre de l'ingénierie des besoins orientée but (*Goal Oriented Requirement Engineering*, GORE).

3.3.2.1. NFR Framework (NFRF)

Cette approche d'analyse des besoins non-fonctionnels (NFR) considère qu'un besoin est une description d'un service du système ou une contrainte que l'utilisateur doit prendre en compte pour réaliser un but. Ainsi, un besoin spécifie comment un but doit être accompli par un système donné [Mylopoulos, 1992].

Chung dans [Chung 2000] considère que cette approche est destinée à représenter et à analyser des buts non-fonctionnels. Le concept le plus important dans cette approche est celui de « *softgoal* » qui représente un but qui n'a ni une définition claire, ni un critère précis pour déterminer s'il a été satisfait ou non.

Les *Softgoals* sont utilisés pour représenter les besoins non-fonctionnels. Les softgoals sont dépendants les uns des autres. Ces rapports de dépendance sont utilisés pour déduire comment un softgoal est satisfait. Les Softgoals peuvent être raffinés en utilisant les opérateurs de raffinements (AND/OR) avec leurs sémantiques évidentes. Aussi, les interdépendances des softgoals peuvent être représentées avec des signes de contribution positive ("+") ou négative ("-").

L'approche NFR est constituée de cinq composants majeurs : softgoals, interdépendances, procédure d'évaluation, méthodes et corrélations [Chung 2000].

Les *Softgoal* sont utilisés pour représenter les besoins. Une fois que tous les softgoals sont identifiés, ils sont reliés entre eux par une relation d'interdépendance (père - fils) en formant un graphe d'interdépendance (*Softgoal Interdependance Graph*, SIG) qui relie chaque softgoal à son père avec une contribution positive ou négative. Dans le cas d'une contribution positive, la satisfaction du fils mène à la satisfaction du père aussi, tandis que dans le cas d'une contribution négative, la satisfaction du fils mène à l'insatisfaction du père.

Dans l'approche NFR, les softgoals identifiés doivent être catalogués et arrangés sous forme de types et d'hiérarchies de relations « *IsA* » qui raffinent le softgoal initial. Ces catalogues sont destinés à une future réutilisation et évitent l'oubli de besoins importants dans d'autres applications. On distingue trois types de catalogues :

- **Le catalogue NFR**

  Ce type de catalogue inclut des informations sur les types particuliers de NFRs (e.g. performance, sécurité, etc.).

- **Le catalogue méthode**

  Dans ce catalogue, on enregistre les connaissances qui aident à raffiner un softgoal et l'opérationnalisation. Ce catalogue peut avoir une méthode générique qui déclare qu'un NFR softgoal appliqué à un élément peut être décomposé en NFR softgoals pour tous les composants de cet élément.

- **Le catalogue de règles de corrélation**

  Il contient des données qui aident à la détection des interdépendances implicites entre les softgoals. Par exemple, le catalogue peut indiquer que l'indexation contribue positivement au temps de réponse.

Dans la figure 3.4, nous présentons un exemple sur le premier type de catalogue qui porte sur le softgoal de sécurité.

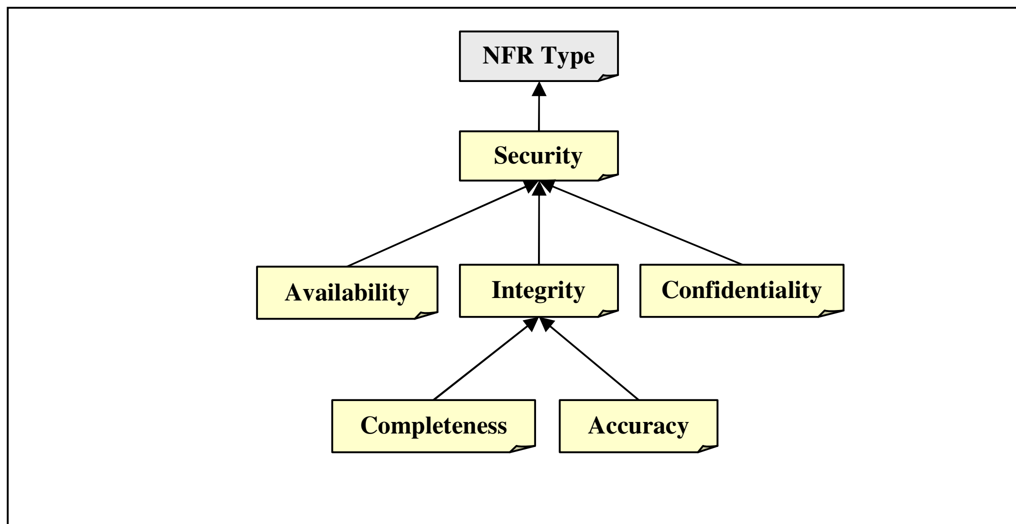


Figure 3.4 : Le type Sécurité dans le Catalogue NFR

Le catalogue de types et les méthodes de décomposition sont utilisés pour décomposer un softgoal spécifique. La figure 3.5 montre la décomposition du softgoal sécurité d'un compte bancaire.

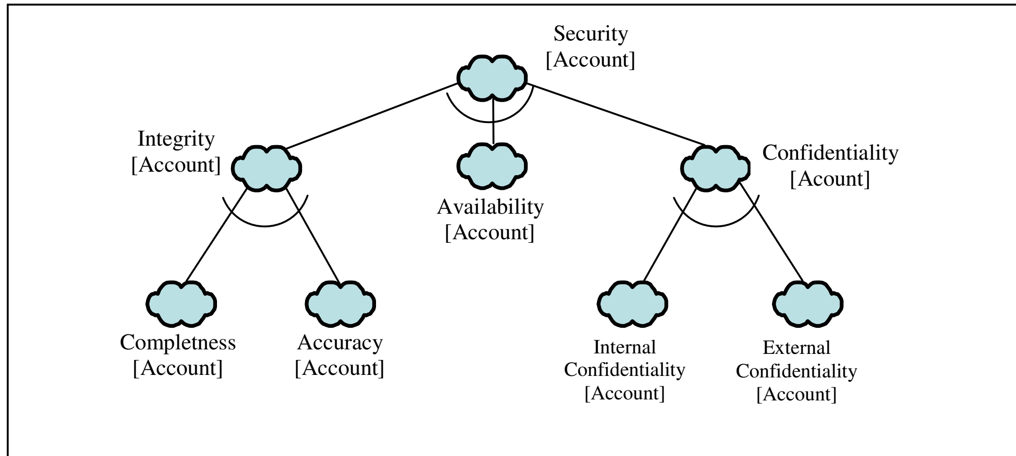


Figure 3.5 : Décomposition du softgoal sécurité d'un compte bancaire

Les softgoals décomposés peuvent être maintenant opérationnalisés et augmentés pour produire un SIG (graphe d'interdépendance de sofgoals) encore plus rafiné, comme l'illustre la figure 3.6. Le but *InternalConsistency* est opérationnalisé via *Access Autorisation* qui est décomposé en sous-buts (*authentification*, *validation des règles d'accès* et *l'identification*). Tous ces sous-buts doivent être satisfaits ensemble, en utilisant l'opérateur *AND,* pour satisfaire *Access Autorisation*. Au niveau du SIG, l'opérateur AND et représenté par un arc et l'opérateur OR par deux arcs.

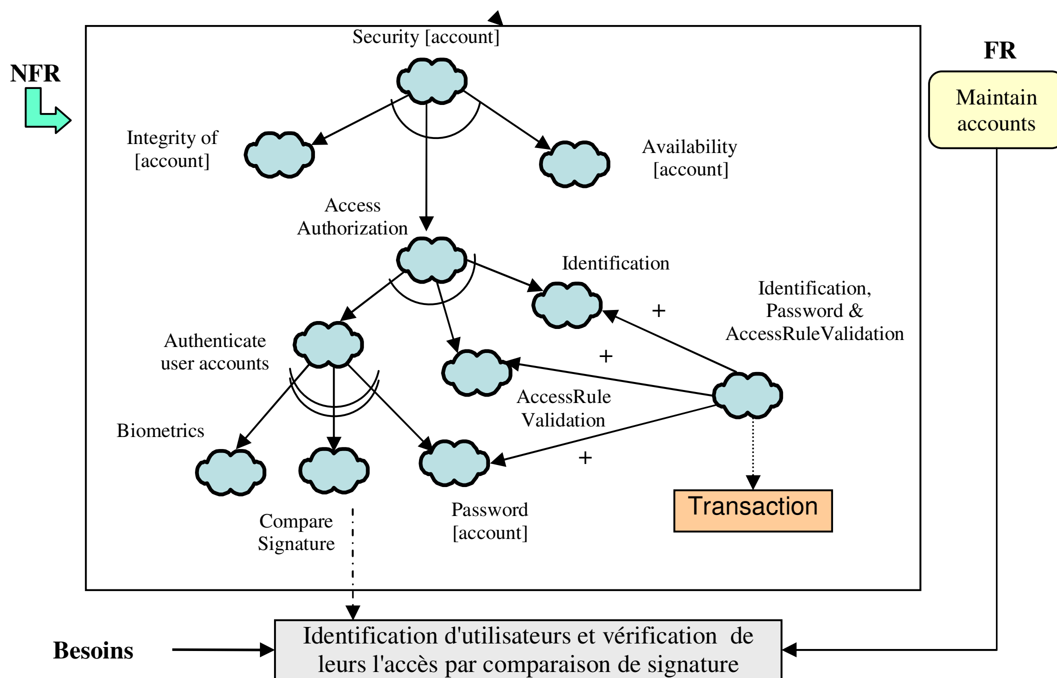


Figure 3.6 : Représentation des besoins de la préoccupation de Sécurité

NFR souligne l'importance des besoins non-fonctionnels, qui sont, par leur nature, entrecoupants. L'approche ne clarifie pas avec exactitude comment les besoins non-fonctionnels sont identifiés, suggérant seulement qu'ils doivent être obtenus en recueillant les connaissances du domaine pour lequel le système est construit. Le SIG du NFR encapsule le traitement de chaque besoin non-fonctionnel entrecoupant, contient les détails de sa décomposition, transforme les décisions et les choix nécessaires à la réalisation durant l'étape suivante de conception.

3.3.2.2. KAOS (Knowledge Acquisition in Automated Specification)

L'approche de KAOS est apparue suite à l'application des idées issues du domaine d'apprentissage des machines au domaine de l'ingénierie des besoins [Lamsweerde 1991]. L'approche KAOS est une référence pour les approches orientées but dans le domaine de l'ingénierie des besoins. KAOS considère l'analyse des besoins comme la coordination de deux tâches : l'acquisition des besoins et leur spécification formelle. La tâche d'acquisition se focalise sur la structuration des besoins dans un modèle préliminaire (le modèle des besoins) en utilisant un langage riche de modélisation. La tâche de spécification formelle se focalise sur le raffinement du modèle des besoins pour obtenir un formalisme plus précis approprié pour des opérations de preuves formelles et de génération de prototypes. Aussi, parmi les principaux avantages de cette approche, nous pouvons citer son langage formel (logique de premier ordre) avec des contraintes de type temps réel pour spécifier les parties critiques du système et la possibilité d'une modélisation informelle. KAOS utilise les buts pour détecter et gérer les conflits parmi les besoins. Elle a été utilisée dans plusieurs projets industriels [Lamsweerde 2001].

L'ontologie de KAOS contient :

1- Des *objets* qui sont les éléments d'intérêt dans le système composé dont les instances peuvent évoluer d'un état à un autre. Les objets peuvent être des *entités*, des *relations* ou des *événements*.

2- Des *opérations* qui sont des relations d'entrée et de sortie sur les objets. L'application d'une opération détermine l'état de transition. Chaque opération possède des pré-conditions, des post-conditions et des conditions de déclenchement.

3- Des *agents ;* est un agent est objet qui agit comme un processeur pour les opérations. Les agents sont des composants actifs qui peuvent être des personnes, des dispositifs, du logiciel, etc. KAOS donne la possibilité aux analystes de spécifier les objets qui sont observables ou contrôlables par les agents.

Le modèle conceptuel de KAOS est représenté en trois niveaux d'abstraction (figure 3.7) :

1. **Le niveau méta modèle (M2)**
   C'est le niveau le plus haut de la pyramide, il fournit un ensemble de constructions pour la représentation des abstractions indépendantes d'un domaine en général, ainsi que les relations qui peuvent exister entre les abstractions.
2. **Le niveau classe (M1)**
   A ce niveau, on concrétise les abstractions disponibles au niveau M2 sous forme d'un ensemble de concepts relatifs à un domaine particulier. La figure 3.7 montre le domaine de librairie qui est modélisé par un emprunteur comme agent, un livre comme une entité, …
3. **Le niveau instance (M0)**
   Au niveau instance, nous pouvons instancier d'une manière individuelle chaque concept à partir du modèle du domaine (types disponibles au niveau M1).

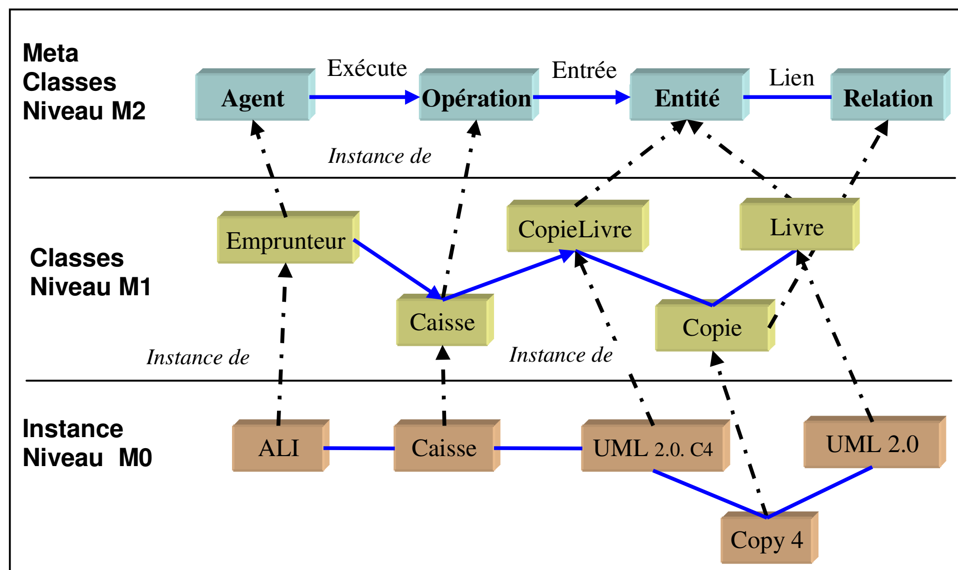


Figure 3.7 : Les niveaux conceptuels du modèle KAOS

En général, une spécification KAOS (voir figure 3.8) est constituée de l'ensemble des modèles fondamentaux suivants :

- Le *modèle but* dans lequel les buts sont identifiés et représentés.
- Le *modèle objet* qui est un modèle UML et qui peut être dérivé à partir de la spécification formelle des buts du moment qu'ils font référence aux objets et à leurs propriétés.
- Le *modèle des responsabilités* qui représente l'affectation des actions aux agents.
- Le *modèle opération* qui définit des services divers qui sont fournis par des agents logiciels.

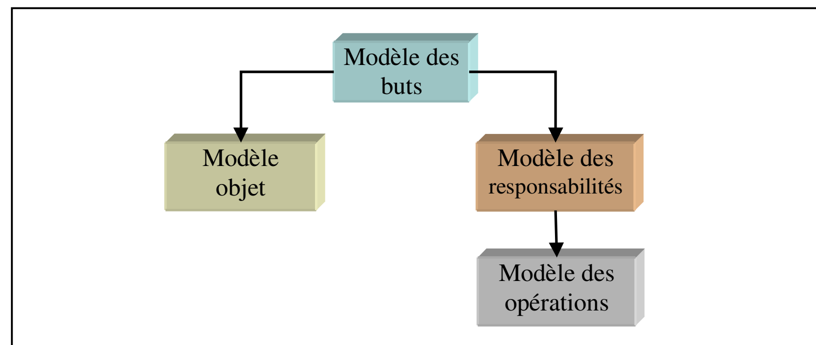


Figure 3.8 : Relations entre modèles dans la spécification KAOS

La figure 3.9 montre la syntaxe graphique utilisée par le modèle des buts dans KAOS.

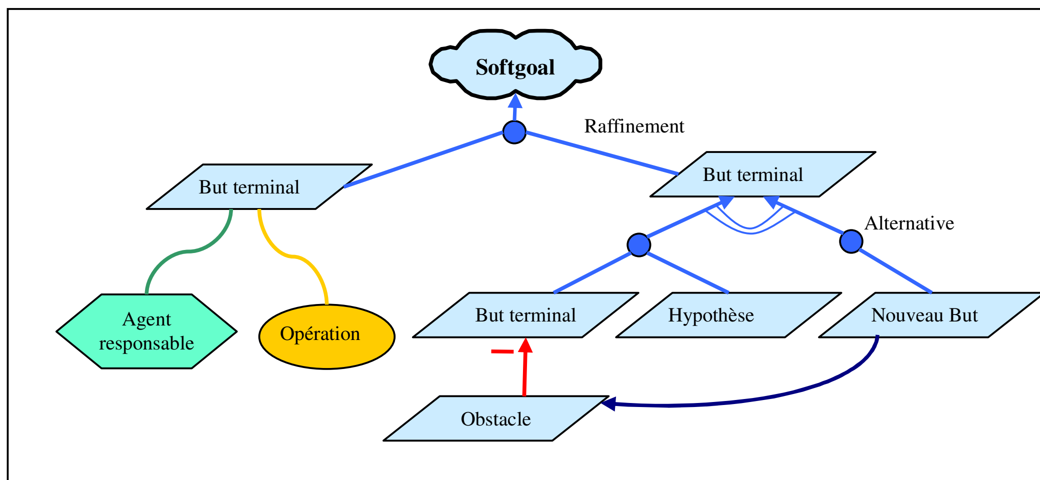


Figure 3.9 : Syntaxe graphique du Modèle des buts de KAOS

3.3.2.3. *i** Framework / Méthodologie Tropos

*i** est une approche basée sur la notion d'agent pour l'ingénierie des besoins. Le terme agent indique que cette approche est centré autour d'un système d'acteurs et des relations qui existent entre eux. Ces relations reflètent comment les acteurs dépendent les uns des l'autres pour réaliser leurs buts, effectuer certaines tâches et acquérir les ressources [Yu 1997].

L'approche *i** est construite sur la base des concepts *d'acteur intentionnel* et de *dépendance intentionnelle*. Les acteurs sont décrits dans leur cadre organisationnel et ont des attributs comme des buts, des capacités, des croyances et des engagements.

Les acteurs peuvent être des agents, des rôles et des positions. Les agents sont des acteurs concrets, des systèmes ou des personnes, ayant des capacités spécifiques. Un rôle est un acteur abstrait qui encapsule des attentes et des responsabilités. Une position est un ensemble de rôles socialement reconnus et typiquement joués par un agent. Cette division est particulièrement utile dans l'analyse du contexte social des systèmes logiciels.

Les dépendances entre les acteurs sont identifiées comme intentionnelles si elles apparaissent à la suite des agents poursuivant leurs buts. Nous distinguons quatre types de dépendances dans i*. Elles sont classifiées sur la base du sujet de la dépendance : but, softgoal, tâche et ressource. Aussi, cette approche fournit un ensemble de représentations graphiques intuitives et simples pour les acteurs et la modélisation des dépendances qui facilite le dialogue entre acteurs pendant l'identification des besoins et encourage l'engagement des acteurs et le retour d'expériences. Parmi les graphes importants de cette méthode nous trouvons :

- **Dépendance stratégique (*Strategic Dependencies*, SD)**

Le SD est élaboré sous forme de graphe permettant d'étudier les acteurs et leurs dépendances, ainsi nous pouvons obtenir une bonne compréhension du processus métier du problème.

- **Raisonnement stratégique (*Strategic Rational*, SR)**

Le SR est aussi un graphe qui explore les différentes alternatives possibles de configuration d'acteurs et leurs dépendances. Ceci permet explorer systématiquement l'espace de nouvelles de conceptions possibles du processus. Chaque alternative peut avoir des implications différentes pour les agents et le système dans l'ensemble.

Parmi les insuffisances de cette approche, on peut citer l'absence de supports de transformations (*mapping*) des artefacts aux activités ultérieures du cycle de vie du logiciel [Yu 1997].

Tropos [Castro 2002] est une méthodologie d'analyse des besoins orientée agent dont les bases sont fondées sur l'infrastructure i*. Cette méthodologie est utilisée pour guider le développement de la phase d'analyse des besoins jusqu'à l'implémentation en passant par l'architecture et la conception. La force de *Tropos* réside dans son langage spécification appelé *« Le Formalisme Tropos »* [Fuxman 2001]. Ce langage est utilisé pour spécifier les contraintes, les invariants, les pré-conditions et les post-conditions. *Propos* génère à la fin des modèles graphiques avec la notation de i*. Ces modèles peuvent être validés par des modèles de vérification (*model_checking*).

3.3.2.4. Conclusion

Les approches à base de but les plus représentatives et discutées dans cette section illustrent le large spectre d'utilisation des buts. L'infrastructure NFR se concentre sur les buts non-fonctionnels et leur décomposition. i* souligne l'importance des agents, leurs dépendances et leurs caractéristiques dans un environnement d'analyse orienté but. Finalement, KAOS est spécialement développé dans un but de formalisation et d'acquisition de connaissances. Ainsi, les trois approches sont fortement complémentaires : NFR fournit les structures de connaissances des buts non-fonctionnels, i* fournit des connaissances sur les agents et KAOS aide à formaliser et raisonner ces connaissances. Les trois approches utilisent des structures de décomposition de buts qui se ressemblent.

3.3.3. Approche orientée Problem Frames (PF)

L'approche Problem Frames développée par Jackson [Jackson 2001] propose dans le cadre du développement de logiciels de traiter les problèmes complexes en les décomposant en plusieurs parties structurées de sous-problèmes. Ces parties interagissent entre elles d'une manière simple en utilisant des interfaces compréhensibles. Une combinaison correcte des solutions aux sous-problèmes est considérée comme la description de la solution du problème original.

L'idée essentielle de l'approche Problem Frames est la décomposition des problèmes complexes en sous-problèmes plus au moins connus, puisqu'il est facile de résoudre un problème qui ressemble à un autre qu'on a traité dans le passé. Ainsi, l'approche Problem Frames a pour objectif d'identifier les problèmes simples communs qui peuvent être utilisés comme des modèles (*pattern*s) à suivre dans le processus de décomposition de problèmes complexes.

Le principal artefact construit en utilisant l'approche Problem Frames est le diagramme de contexte qui représente simplement le problème et les domaines du monde réel qui agissent avec ce dernier. Bien que, l'approche de PF se concentre sur le traitement des besoins fonctionnels, elle reconnaît aussi l'importance des besoins non-fonctionnels. Alors, aucun type des besoins entrecoupants n'est identifié ou traité explicitement par cette approche.

Comme l'auteur de ce travail ne fournit aucune approche systématique pour l'utilisation des concepts, alors l'approche peut être difficile à comprendre et son application n'est pas évidente. Notons qu'un travail récent tente de combiner les Problem Frames avec des architectures logicielles à base de composants [Grundy 2000].

3.3.4. Approche orientée Cas d'Utilisation (Use Cases)

Les scénarios et les cas d'utilisation sont apparus récemment comme les approches les plus utilisées dans l'industrie des logiciels pour l'extraction des besoins. Un scénario est défini comme séquence d'actions effectuées par agents intelligents [Alexander 2004]. Les scénarios (essentiellement des histoires courtes) exploitent la capacité humaine de raisonner à partir des histoires. Ainsi, quand les besoins sont représentés sous forme de scénarios, alors il est plus facile de détecter les inconsistances, les omissions et les conflits dans le système à construire. Les scénarios avec des options multiples, des conditions et des branches peuvent être organisés sous forme de cas d'utilisation.

Le processus de cas d'utilisation commence par l'identification des acteurs qui vont utiliser le système. Ensuite, on cherche pour chaque acteur l'ensemble des fonctionnalités qui lui sont fournies par le système. Cet ensemble de fonctionnalités devient un cas d'utilisation pour cet acteur. L'ensemble des cas d'utilisation forme le modèle de cas d'utilisation.

Chaque cas d'utilisation décrit une partie du système utilisée par quelques acteurs pour obtenir un résultat désiré. Les cas d'utilisation explorent systématiquement la façon avec laquelle un système est utilisé par les intervenants, montrent les différents scénarios d'utilisations, aident à la compréhension entre les utilisateurs et les développeurs et capturent les besoins de haut niveau qui sont importants pour un utilisateur [Jacobson 1992]. Les concepts de base de cette approche sont :

- *Les acteurs :* représentent les entités qui interagissent avec le système
- *Cas d'utilisation :* représentent ce qui doit être réalisé par le système
- *Scénarios :* représentent les instanciations des cas d'utilisation
- *Relations :* représentent les interactions entre les acteurs et le système.

Pour réaliser la modularité des cas d'utilisation, nous avons besoin de deux mécanismes :

3.3.4.1. Mécanisme de séparation des cas d'utilisation

Les cas d'utilisation sont conçus pour partitionner un système en plusieurs parties séparées et reliées entre elles. Le partitionnement fonctionne bien pour les cas d'utilisation qui sont égaux ou pairs (*peers*). Aucun des pairs n'est un cas de base plus que les autres et aucun n'est plus obligatoire ou plus facultatif que les autres.

Exemple : Dans la gestion d'un appel téléphonique d'un système de télécommunications, nous pouvons distinguer trois cas d'utilisation d'appel (local, national, international). Les trois cas d'utilisation sont différents mais aucun cas n'est plus important ou plus prioritaire que l'autre. Cependant, il y a des cas d'utilisation

qui dépendent d'autres cas d'utilisations, ceci est fondamental pour le fonctionnement des cas d'utilisation. Pour séparer ces cas d'utilisation, nous avons besoin d'un mécanisme d'extension.

Ce mécanisme permet à un système complexe d'être développé en commençant avec un cas d'utilisation de base et puis en l'étendant avec des comportements et ce sans le modifier ; ce mécanisme est appelé extension (*extend)*.

3.3.4.2. Mécanisme de composition des cas d'utilisation

Pour que le système puisse fonctionner, nous devons composer, intégrer ou tisser les parties précédentes, c'est-à-dire les cas d'utilisation séparés, dans un tout cohérent pour obtenir le code exécutable. Nous avons plusieurs options pour réaliser cet objectif : on peut effectuer le tissage (*weaving*) avant la compilation, durant la compilation ou durant le temps d'exécution [Jacobson 2003].

Les relations entre les cas d'utilisation comme la généralisation, l'extension et l'inclusion sont aussi modélisées. L'inclusion est utilisée pour factoriser un comportement commun entre les cas d'utilisation. La généralisation est utilisée pour raffiner la séquence d'actions du cas d'utilisation le plus général à partir de cas très spécifiques. Finalement, la relation d'extension permet l'ajout d'un comportement additionnel au cas d'utilisation de base sans changer son comportement initial (figure 3.10).

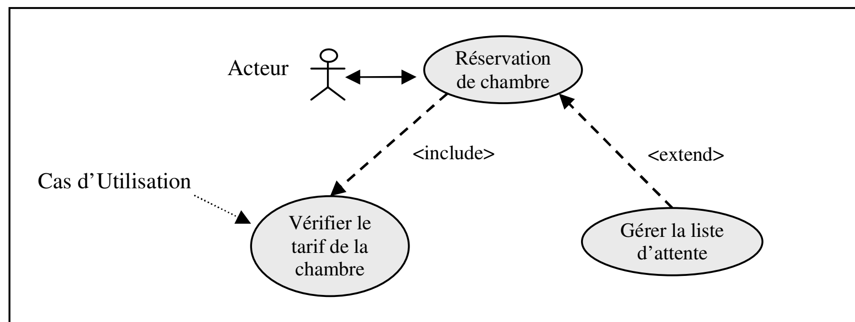


Figure 3.10 : Représentation d'acteurs et de cas d'utilisation

3.3.5. Conclusion

Nous avons présenté dans les sections précédentes un ensemble sélectionné d'approches contemporaines développées dans le cadre de l'ingénierie des besoins. Nous n'avons pas l'intention de présenter une étude exhaustive de ces approches. Mais nous avons essayé d'étudier les travaux qui ont beaucoup influencés les travaux dans ce domaine et qui sont, en principe, à l'origine des approches et méthodes émergentes dans le domaine de l'ingénierie des besoins orientée aspect qui seront étudiées dans les sections suivantes.

3.4. Approches Orientées Aspects

Les techniques de l'ingénierie des besoins qui reconnaissent explicitement au même titre l'importance des préoccupations entrecoupantes fonctionnelles et non-fonctionnelles en plus des préoccupations de base (non entrecoupantes) sont appelées approches d'ingénierie des besoins orientées aspect (*Aspect Oriented Requirement Engineering*, AORE). L'émergence de ces approches est incitée par les trois facteurs suivants : besoin de composition, traçabilité et développement de nouvelles technologies.

- Le premier facteur (composition) est la réalisation de l'intégration des besoins séparés (e.g. les points de vue, les buts ou les cas d'utilisation). Ce facteur représente le goulot d'étranglement de plusieurs méthodologies d'analyse des besoins.
- Le deuxième facteur représente la traçabilité des propriétés entrecoupantes à travers le cycle de vie d'un logiciel.
- Le troisième facteur est l'émergence de la programmation orientée aspect qui a fait apparaître de nouveaux concepts pour encapsuler les préoccupations transversales, et de nouveaux mécanismes de composition (comme les points de jointure et les points de coupure). L'identification et le traitement de ces préoccupations entrecoupantes dès le début du cycle de vie d'un logiciel assurent l'homogénéité, dans un processus orienté aspect, du développement de logiciel et aident à réaliser le facteur traçabilité des aspects mentionnés ci-dessus.

Ainsi, les approches de l'ingénierie des besoins orientée aspect se focalisent sur le traitement systématique et modulaire, le raisonnement, la composition et la traçabilité des besoins transversaux fonctionnels et non fonctionnels à travers des abstractions appropriées, la représentation et un mécanisme de composition conçu pour l'ingénierie des besoins. Dans les sections suivantes, nous étudions les approches les plus importantes qui ont été développées dans le cadre de L'AORE.

3.4.1. Approche Orientée Aspect basée points de vue (AORE)

L'approche AORE (Aspect Oriented Requirement Engineering) est l'une des premières approches à présenter le concept orienté aspect au niveau de l'analyse des besoins. En fait, [Araujo 2002] est le premier article présentant le terme d'aspect précoce (*Early Aspect*) qui est maintenant un terme clé (*de facto*) dans le domaine de l'ingénierie des besoins et de l'architecture du logiciel. Cette approche est généralement appelée «*AORE with Arcade* » pour la distinguer du domaine AORE lui-même. A l'origine, cette approche est basée sur la méthode PREview présentée précédemment. Elle propose une séparation de la composition des besoins orientés

aspects et des besoins de base. Ainsi, ces besoins sont fortement indépendants les uns des autres, ce qui permet d'avoir une meilleure séparation des besoins et d'améliorer la réutilisabilité des classes d'aspects par le fait de les instancier plusieurs fois. Puisque les règles de composition opèrent sur une granularité individuelle des besoins, il est possible d'identifier et de gérer les conflits à une granularité très fine. Cependant, l'approche est dans sa version la plus basique et nécessite une validation à travers des projets réels.

De même cette approche propose un modèle pour séparer les besoins de base (*non aspectual requirements*) et les besoins entrecoupants (*aspectual requirement*), ainsi que leurs règles de composition. Ce modèle représente un processus général pour l'analyse des besoins qui peut être instancié avec n'importe quelle technique d'analyse des besoins. Une instanciation concrète de ce modèle est proposée par A. Rashid [Rashid 2003] en utilisant PREview comme point de vue et un mécanisme de composition basé sur XML. La représentation XML des points de vue est donnée dans la figure 3.11.

```
<Viewpoint name="V1">
     <Requirement id="1">...</Requirement>
     ….
     <Viewpoint name="SubVP_V1">
          <Requirement id="1">...</Requirement>
          …
     </Viewpoint>
</Viewpoint>
```

Figure 3.11. Représentation XML des points de vue

Cette approche est caractérisée par son mécanisme générique qui peut être facilement adapté pour traiter aussi bien les besoins fonctionnels que les besoins non-fonctionnels. Cependant, il n'est pas clair comment les besoins fonctionnels transversaux sont identifiés. L'identification des besoins non-fonctionnels transversaux n'est pas aussi complètement satisfaisante. Nous constatons aussi l'absence de moyens supports (e.g. directives de développement, outils, etc.) qui peuvent aider l'ingénieur des besoins dans la détection de ces préoccupations.

D'autre part, un travail très récent sur cette approche qui tente de lui donner une autre dimension caractérisée par l'analyse linguistique sémantique du langage naturel utilisé dans les documents des besoins pour l'identification des aspects [Sampaio 2005a] [Sampaio 2005b].

3.4.2. Approche Orientée Aspect Basée But

Pour les approches à base de but, il est commun d'avoir un but ou un *softgoal* qui contribue à l'accomplissement d'autres buts ou d'autres *softgoals.* De telles relations

s'accordent clairement au pattern de préoccupations entrecoupantes (*scattering and tangling*). Cependant, trop peu de travaux sont disponibles actuellement sur l'ingénierie des besoins orientés aspect basée but. Nous présentons dans la section suivante l'approche la plus connue.

3.4.2.1. Approche ARGM (*Aspects in Requirements Goal Models*)

Dans cette approche, développée par Y. Yu [Yu 2004], les aspects sont découverts à partir des relations qui existent entre les buts fonctionnels et non fonctionnels en décomposant ces derniers en sous-buts, sous-softgoals et leurs opérationnalisations.

Quand les buts et les softgoals sont décomposés, des liaisons de corrélation entre softgoals et les buts fonctionnels sont établies. A partir de ce graphe d'interdépendance entre les buts et les softgoals, les aspects peuvent être détectés comme des tâches communes pour satisfaire les nœuds en relation. Un type spécifique de graphe, appelé le V-graphe, est utilisé pour représenter les relations but-softgoal-aspect (voir figure 3.12).

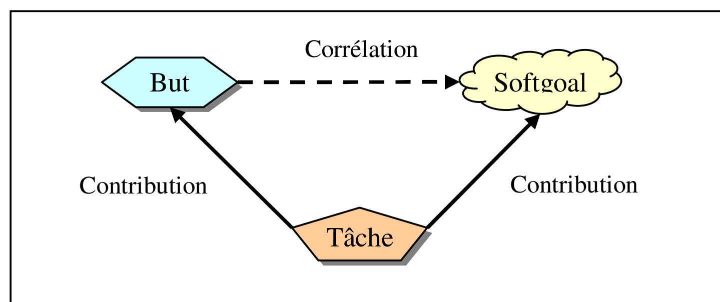


Figure 3.12 : V-Graphe

L'approche ARGM est très similaire à l'approche NFRF puisque chaque but a un sujet et un type. Le type reflète le besoin générique fonctionnel ou non-fonctionnel, tandis que le sujet capture les informations contextuelles relatives au but. Dans cette approche, les sujets sont utilisés comme points de coupe auxquels les buts fonctionnels et les tâches sont reliés aux softgoals non-fonctionnels. Les tâches associent au but un sujet donné. Les tâches aspect sont des tâches d'opérationnalisation pour les besoins non fonctionnels. De plus, l'approche propose le concept de but-aspect : le but que la tâche aspect et les tâches fonctionnelles réalisent.

ARGM propose un processus clair pour décomposer les buts et les softgoals et ensuite identifier les aspects. La référence [Yu 2004] contient plus de détails sur les différents algorithmes utilisés par ce processus. Selon ARGM, les buts (besoins fonctionnels) et les softgoals (besoins non-fonctionnels) sont récursivement décomposés jusqu'à ce qu'ils puissent être réduits à des tâches spécifiques. Les

besoins transversaux (fonctionnels et non-fonctionnels) peuvent alors être identifiés comme les tâches qui contribuent dans la réalisation de plusieurs buts et softgoals. Cependant, le mécanisme de décomposition ne permet pas la décomposition en sous-buts, un but fils qui contribue négativement à un but parental (figure 3.13).

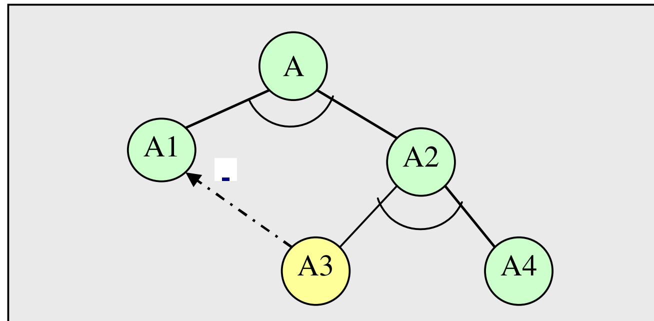


Figure 3.13 : Procédure de décomposition (cas d'un mauvais scénario)

Comme illustré sur la figure 3.13, A3 contribue négativement à A1, ainsi la procédure de décomposition enlèvera A3, ce qui ne permet pas de réaliser A2 (puisque A2 = A3 AND A4) et par conséquent, A ne sera pas réalisé (puisque A = A1 AND A2).

3.4.3. Approche Orientée Aspect Basée Cas d'Utilisation

Les approches basées cas d'utilisation et scénarios ont été à l'avant-garde des approches classiques vu leurs mécanismes de composition réalisés à travers les relations *extend* et *include*. Cependant, la nature orientée fonctionnalité de ces techniques a laissé à l'écart la prise en charge des préoccupations non-fonctionnelles. Par conséquent, des travaux récents dans le domaine de l'ingénierie des besoins orientés aspect (AORE) ont étendu le mécanisme de composition disponible pour inclure les préoccupations non-fonctionnelles.

3.4.3.1. Approche AOSD/UC

AOSD/UC (*Aspect Oriented Software Development with Use Cases*) étend l'approche traditionnelle basée cas d'utilisation avec trois concepts principaux :

- Points de coupure (*pointcuts*) : ce sont les points de jointure représentés par les points d'extension des cas d'utilisation. Ces points sont nommés dans la spécification des cas d'utilisation.
- Tranches de cas d'utilisation (*use case slice*) : qui contiennent les détails d'un cas d'utilisation à une phase donnée du développement.
- Module de cas d'utilisation (*use case module*) : qui contient tous les détails liés à un cas d'utilisation, à travers toutes les phases du développement.

L'approche distingue deux types de cas d'utilisation :

- Cas d'utilisation égaux (*peer)* : ils sont distincts et indépendants les uns des autres, chacun peut être utilisé séparément sans référence à l'autre; ils représentent les besoins de base. La composition de ces cas d'utilisation déclenche la composition de leurs opérations sans aucune extension. Cependant, la composition de ce type de cas d'utilisation doit prendre en considération le comportement entrecoupant et les conflits entre les classes impliquées dans la réalisation des cas d'utilisation.
- Cas d'utilisation extension (*extension*) : ce sont des caractéristiques supplémentaires dans un cas d'utilisation de base. Malgré que les extensions puissent être définies indépendamment des cas de base, elles doivent être normalement utilisées avec les cas de base. Quand les extensions sont composées avec les cas de base, leurs opérations interfèrent durant l'exécution avec celles du cas d'utilisation de base.

L'approche AOSD/UC encourage la capture des besoins non fonctionnels comme des cas d'utilisation. Dans cette approche, le classifieur UML pour les cas d'utilisation est enrichi par les labels /basic/, /alt/, et /sub/ respectivement pour les flux des cas d'utilisation de base, alternatif et extension. Le label *basic* indique que le cas d'utilisation peut être lancé par un acteur, tandis que le label *sub* indique que le flux peut être référencé ou inclus dans un autre flux. Le flux alternatif (*alt*) peut être défini comme un cas d'utilisation extension.

La définition du flux dans un cas d'utilisation extension peut aussi être complétée par une condition d'extension avec les mots clés *before*, *after* et *around* qui reflètent la condition et la séquence d'exécution d'une extension dans le flux du cas d'utilisation de base. Les figures 3.14 et 3.15 présentent une version très simplifiée d'un cas d'utilisation de réservation de chambres dans un système de gestion d'hôtel.

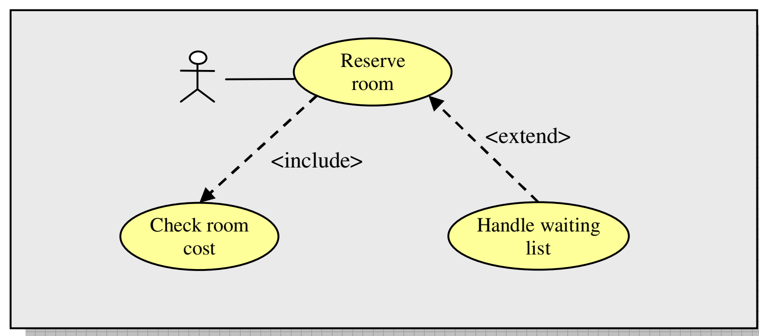


Figure 3.14 : Modèle du cas d'utilisation de réservation de chambres

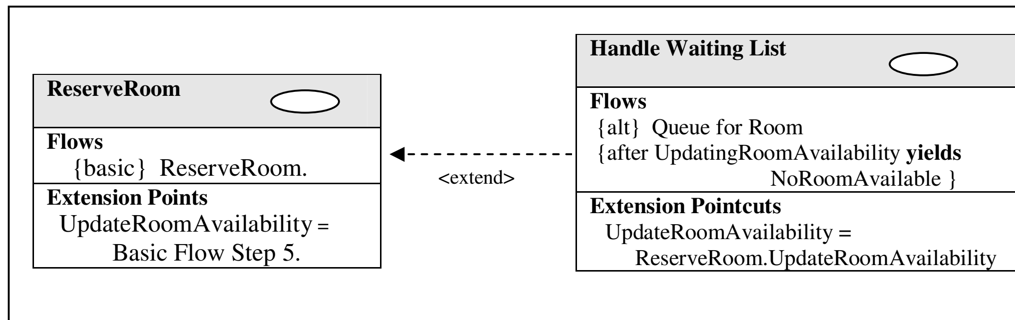


Figure 3.15 : Description de l'extension du cas d'utilisation

AOSD/UC est une approche de développement qui couvre tout le cycle de vie de développement de logiciels (de l'analyse des besoins jusqu'à l'implémentation). Pour la phase de l'ingénierie des besoins, le processus d'AOSD/UC est très similaire à celui du processus de Cas d'Utilisation traditionnel (section 3.3.5). La seule différence significative est l'introduction de cas d'utilisation séparés pour les besoins non-fonctionnels. Ceux-ci doivent être identifiés, représentés et enregistrés en parallèle avec les besoins fonctionnels en utilisant la même infrastructure de cas d'utilisation. Un autre point secondaire de différence est l'encapsulation de chaque cas d'utilisation et de ses éléments liés séparément dans un slice de cas d'utilisation.

Deux autres approches qui ne sont pas très citées dans la littérature sont :

- Modélisation des scénarios avec aspect : elle traite l'influence qu'un aspect peut avoir sur la modélisation du comportement. Alors, on considère que le comportement qui se répète souvent comme une aspect [Whittle 2004].

- Approche dirigée par cas d'utilisation orientés aspect : elle est très proche de AOSD/UC par le fait qu'elle sépare les fonctionnalités transversales en cas d'utilisation en inclusion et en extension. Elle propose en plus de modéliser les attributs de qualités sous forme de cas d'utilisation [Araujo 2003].

3.4.4. Approches orientées MDSC

La séparation multidimensionnelle des préoccupations est basée sur la prémisse que tous les artefacts de développement logiciels sont composés de multiples préoccupations entrecoupantes. Il n y a aucune préoccupation qui est plus importante que les autres, alors nous obtenons des (*peers*) préoccupations [Tarr 1999]. Actuellement, on distingue deux méthodes qui traitent les besoins suivant les concepts avancés dans le cadre de la séparation multidimensionnelle des préoccupations ; à savoir méthode *Cosmos* et la méthode *CORE*. Dans cette thèse, nous avons choisi d'étudier la méthode CORE parce qu'elle est plus documentée et plus récente que la méthode Cosmos. [Sutton 2002], [Sutton 2003] et [Sutton 2004] fournissent plus de détails sur la méthode Cosmos.

3.4.4.1. Méthode CORE (Concern-Oriented Requirements Engineering)

La méthode CORE est une réelle adaptation de la méthode AORE/Arcade (section 3.4.1). Plusieurs concepts et techniques de représentation développés dans AORE/Arcade sont réutilisés dans CORE. Par contre, la principale différence entre les deux méthodes est que CORE adopte une vue uniforme sur toutes les préoccupations. Dans CORE une préoccupation est définie comme une collection uniforme de besoins [Moreira 2005a] [Moreira 2005b].

La figure 3.16 illustre ce modèle par la représentation de l'espace des préoccupations CORE au niveau de l'analyse des besoins. Chaque face de l'hyper-cube représente une préoccupation particulière. Puisque tous les besoins sont considérés comme des *peers* alors n'importe quel sous-ensemble de besoins peut être choisi comme base pour projeter l'influence des autres besoins. Par conséquent, nous pouvons dire que CORE supporte parfaitement la séparation multidimensionnelle des préoccupations. En plus, ce modèle définit le concept espace de méta-préoccupations.

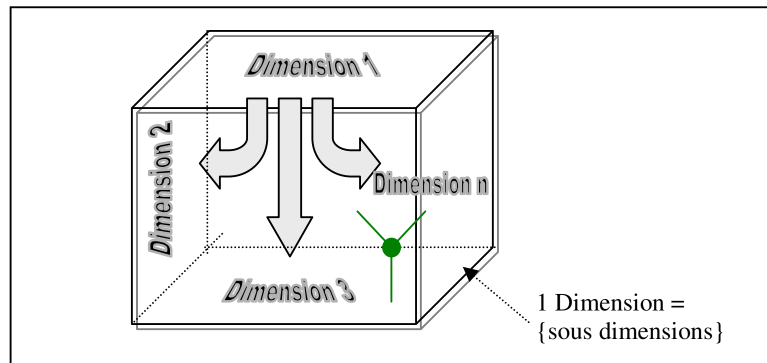


Figure 3.16 : Représentation en hyper-cube des préoccupations dans CORE

L'espace méta-préoccupation contient toutes les préoccupations abstraites (fonctionnelles et non-fonctionnelles) qui peuvent se manifester dans des systèmes logiciels différents (figure 3.17). Ainsi, l'espace méta-préoccupation peut être vu comme un catalogue de préoccupations qui peuvent apparaître d'un moment à l'autre dans des systèmes logiciels. Par conséquent, une préoccupation abstraite de cet espace peut être utilisée comme base de classification des besoins d'un système donné. Cette base permet d'identifier les besoins concrets correspondants dans l'espace du système.

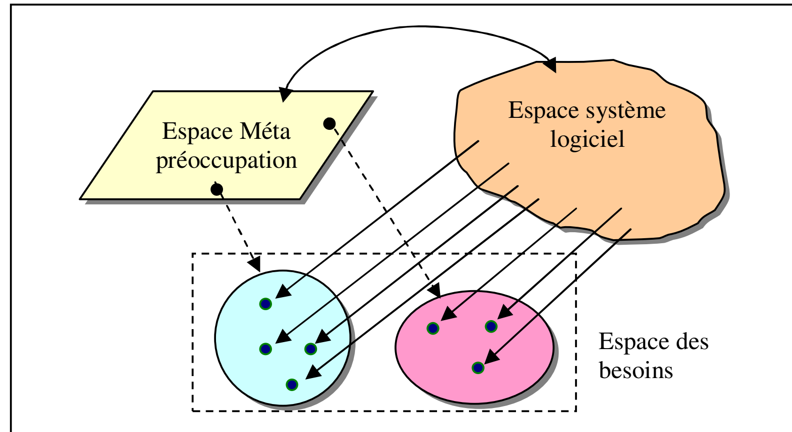


Figure 3.17 : Espace méta-préoccupations

Les préoccupations dans CORE sont représentées par des documents XML en utilisant le tag <*concern*> pour encapsuler toutes les préoccupations en une structure hiérarchique qui contient tous les besoins et les sous-besoins avec un identificateur unique pour chaque élément de la hiérarchie. CORE utilise aussi des matrices symétriques pour illustrer l'influence et le type de contribution de chaque préoccupation par rapport aux autres. Notons qu'une contribution négative entre deux préoccupations est un conflit qui doit être résolu (Table 3.1). La référence [Sampaio 2005] contient plus de détails sur un outil développé comme support pour cette approche.

	C1	C2	….	Cn
C1		+		-
C2	+			
….				
Cn	-			

Table 3.1 : Type de contribution entre préoccupations

3.4.5. Approche orientée aspect basée composant

Les approches orientées aspect pour l'ingénierie des besoins présentées dans les sections précédentes n'ont pas, jusqu'ici, spécifié la granularité des modules qui doivent encapsuler les aspects. Les techniques orientées aspect à base de composants soulignent l'utilité des aspects dans un système à base de composants et de ce fait, appliquent la notion d'aspects spécifiquement aux modules de large granularité.

3.4.5.1. Approche AOREC (AORE for CBSD)

Un aspect dans AOREC est une caractéristique d'un système pour laquelle les composants fournissent ou requièrent des services. Les aspects aident à identifier, catégoriser et raisonner sur les besoins de chaque composant [Grundy 2000]. Son inconvénient majeur est le manque de clarté sur sa méthode d'identification des aspects pour chaque composant. Aussi, elle ne fournit pas de mécanismes pour traiter les conflits et elle ignore le processus d'intégration entre composants et aspects (voir figure 3.18) [Grundy, 1999]. Les aspects peuvent être produits sous forme de nouveaux composants développés ou par adaptation d'anciens composants déjà existants.

Le processus de base de cette approche est illustré sur la figure 3.18.

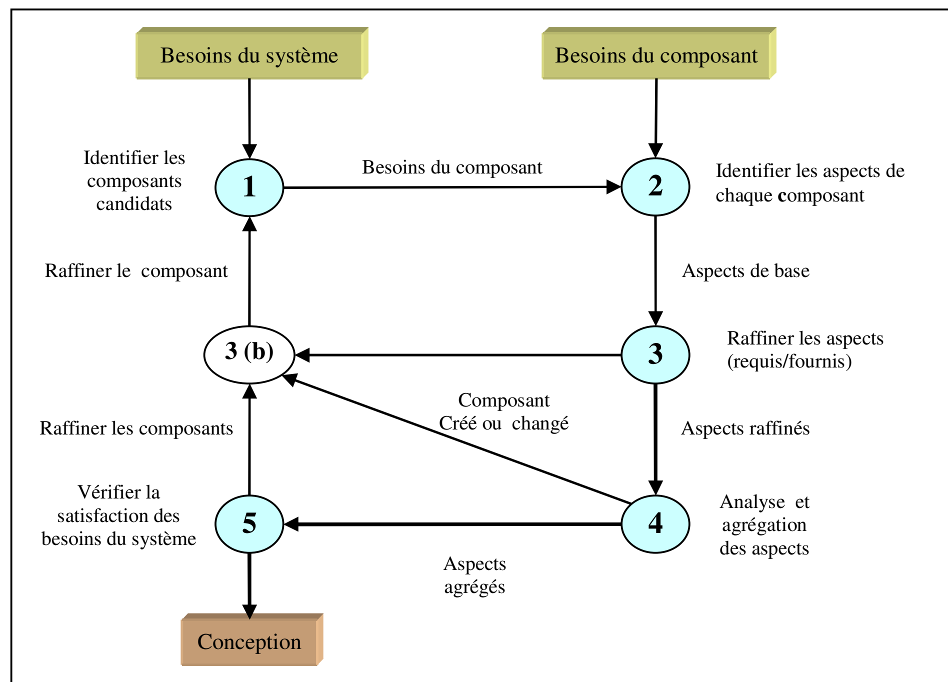


Figure 3.18 : Processus de base de l'approche AOREC

Ce processus commence par l'analyse des besoins généraux de d'application.

- **Etape1 :** Les besoins systèmes sont utilisés pour identifier les composants, les besoins de chaque composant sont raffinés.
- **Etape2 :** Les aspects associés à chaque composant sont identifiés.

- **Etape3 :** Les aspects sont raffinés pour déterminer les informations requises et fournies par chaque aspect. Les aspects identifiés sont utilisés dans le processus de composition et de configuration de composants.
- **Etape4 :** Les aspects raffinés et associés à chaque classe de composants sont regroupés. Chaque agrégation d'aspects peut être identifiée comme un nouveau composant.
- **Etape3b :** Les nouveaux composants qui apparaissent suite à l'agrégation peuvent remettre en cause la configuration globale des composants, ceci nous conduit à vérifier la nouvelle configuration des composants.
- **Etape5 :** Les composants et les aspects produits sont vérifiés par rapport aux besoins du système. Si les besoins sont complètement satisfaits par l'ensemble de composants et aspects produits, alors la phase de conception peut être lancée.

Par ailleurs, le CBSE (Component Based Software Engineering) vise à réduire les dépendances entre les différentes entités logicielles (composants) lors de la compilation et lors de l'exécution. Dans le cadre de la programmation par composants, les industriels ont abordé le problème de la séparation des préoccupations, en proposant des modèles de composants tels que les Enterprise Java Beans (EJB/J2EE) [Blevins 2001] et le modèle de composants de CCM/CORBA [CORBA]. Ces modèles proposent le concept de « *conteneur* » gérant des services non-fonctionnels indépendamment de la programmation des composants.

L'objectif des EJB et de CCM est de répondre à des problèmes importants du monde industriel. L'un de ces problèmes concerne la gestion de différents services non-fonctionnels sur les composants tels que la persistance, les transactions, la distribution, etc., de manière transparente pour le programmeur des composants. Ces services sont gérés par des conteneurs qui encapsulent les instances de composants.

Cette approche permet d'isoler différents métiers dans le développement d'applications à base de composants, chaque métier ayant son domaine d'expertise. Le fournisseur de la plate-forme est un expert dans le domaine de la distribution, de la sécurité, des transactions, etc. Il offre les outils nécessaires à la gestion de ces services. Le programmeur de composants, quant à lui, s'intéresse particulièrement à la fonctionnalité du composant. L'assembleur d'application connecte des composants et paramètre les services offerts par la plate-forme. Enfin, une personne est chargée du déploiement de l'application ainsi créée [Amirat 2007b].

3.4.6. Approche Theme/Doc

L'approche Theme/Doc est une partie de l'approche Theme conçue dans le cadre de l'ingénierie des besoins. Le noyau de l'approche est le concept de thème qui représente une unité significative de fonctionnalité cohésive. La méthode est centrée autour d'une représentation, sous forme de graphe, des thèmes potentiels [Baniassad 2004a] [Baniassad 2004a].

L'approche est, en principe, applicable durant une étape très avancée du processus de l'ingénierie des besoins, quand au moins un document initial est prêt pour l'analyse lexicale. En effet, l'approche utilise l'analyse lexicale, dans un premier temps, pour l'identification des interdépendances entre les activités (i.e. les verbes) à partir des besoins exprimés en langage naturel, et, dans un second temps, les verbes sont structurés sous forme d'ensembles pour construite les thèmes.

En raison de l'intérêt qu'accorde cette approche aux aspects comportements d'un système, les préoccupations non-fonctionnelles ne sont pas considérées. En effet, l'objectif principal de l'approche est de découvrir les préoccupations fonctionnelles entrecoupantes en se basant sur les comportements du système, ce qui n'est pas possible pour les préoccupations non-fonctionnelles car aucun comportement ne se manifeste [Baniassad 2004b].

L'approche est supportée par l'outil Theme/Doc qui fournit un ensemble de vues qui aident aussi bien dans l'analyse des besoins que dans le transfert (*mapping*) direct des vues vers des artefacts UML (Theme/UML). En réalité, l'outil Theme/Doc est l'élément le plus important de l'approche parce que l'analyse et les étapes de l'approche sont basées sur les visualisations graphiques de l'outil.

3.5. Comparaison

La comparaison entre les différentes approches peut être basée sur différents critères que nous énumérons ci-après. Ces critères sont définis par *Filman et al.* pour évaluer les modèles conçus pour la phase de l'ingénierie des besoins [Filman 2004].

- *Support de l'orienté aspect :*
  L'approche soutient-elle directement les concepts de base AOSD ?
- *Activités dans le cycle de vie :*
  Quelles activités du processus de développement sont prises en charge par l'approche ?

- *Règles de composition :*

  L'approche fournit-elle des règles explicites pour composer les aspects avec les besoins ?

- *Traitement des conflits :*

  L'approche fournit-elle une méthode pour identifier et résoudre les conflits entre aspects ?

- *Maturité :*

  L'approche a-t-elle été utilisée dans des projets réels ?

- *Transformation vers des artefacts postérieurs :*

  L'approche prévoit-elle comment les besoins orientés aspect sont transformés en artefacts de conception et d'implémentation ?

- *Support Outil :*

  Y - a t- il des outils qui supportent l'approche ?

Le tableau suivant (table 3.2) représente une synthèse sous forme de résultats obtenus suite à l'application des critères ci-dessus sur chaque approche présentée dans ce chapitre.

Approches			Critères						
			Support des Aspects	Activité dans le cycle de vie	Règles de composition	Traitement des conflits	Maturité	Passage aux artefacts postérieurs	Outils de supports
Approches	Approches Non Orientées Aspect	**PREview**	X	RE	X	X	√	X	JPREView
		NFRF	X	RE	√	√	√	√	NFR Catalogue
		KAOS	X	RE, Spec.	X	√	√	√	Grail
		I*	X	Org.Req., RE	X	√	√	X	OME
		ProblemFrames	X	RE, Archit.	√	X	√	√	X
		Use Cases	X	RE, Arch.Impl.	√	X	√	√	UML-Tools
	Approches Orientées Aspect	**AORE / Arcade**	√	RE	√	√	X	√	Arcade
		ARGM	√	RE	√	X	√	√	X
		AOSD / UC	√	RE, Arch., Impl.	√	X	√	√	UML Tools
		CORE	√	RE	√	√	√	X	X
		AOCRE	√	RE for CBS	X	X	Partiel	√	JComposer
		Theme/Doc	√	Analyse, conception	√	X	X	√	Theme/Doc

(Légende : RE (Requirement Engineering), Arch.(Architecture),CBS(Component-Based System), Impl.(Implementation),Spec.(Specification), Org.(Organisation))

Table 3.2 : Comparaison des approches étudiées

3.6. Langages d’expressions des besoins

Une fois les besoins identifiés, représentés et raffinés sans aucun conflit, ils doivent être rédigés sous forme de documents qui sont décrits dans un langage dédié à cette phase. Ces documents représentent la seule source d’information pour la phase suivante dans le cycle de vie des logiciels. Or, actuellement on ne trouve aucun langage qui a été développé spécialement pour l’expression des besoins.

Un langage d’expressions des besoins (*Requirement Description Langage, **RDL***) est un élément principal, souvent implicite, qui affecte n'importe quel processus d’analyse des besoins. Il représente le langage dans lequel les besoins sont exprimés, c’est-à-dire, le langage naturel employé par les intervenants dans les documents, dans les normes utilisées et dans toutes les autres sources de besoins.

La majorité des approches développées dans le contexte de l’ingénierie des besoins néglige cet élément (langage), ou le limite à un sous-ensemble (semi-formel ou formel) du langage naturel à cause de l’ambiguïté et de l’imprécision inhérentes qui peuvent compliquer la spécification des besoins.

Les langages de description des besoins ne sont pas très communs dans le domaine d’ingénierie des besoins. Il est plus habituel de développer des notations de représentation visuelle (e.g. les buts (*goals*) [Chung 2000], les cas d'utilisation [Jacobson 1992]) et les template (e.g. points de vue [Sommervile 1997]) pour décrire les besoins. Bien évidemment, toutes les notations visuelles sont aussi des langages, mais elles sont utilisées pour la représentation visuelle, plutôt que pour un objectif de description.

Les besoins et les préoccupations sont décrits à partir d’un ensemble de phrases et chaque phrase est composée à partir d’une ou de plusieurs clauses. Une préoccupation peut être simple (contenant seulement des besoins), ou composite (contenant des besoins et aussi d’autres préoccupations). Ceci permet d’avoir une organisation hiérarchique des besoins. La figure 3.19 montre les principaux éléments du méta-modèle des RDLs.

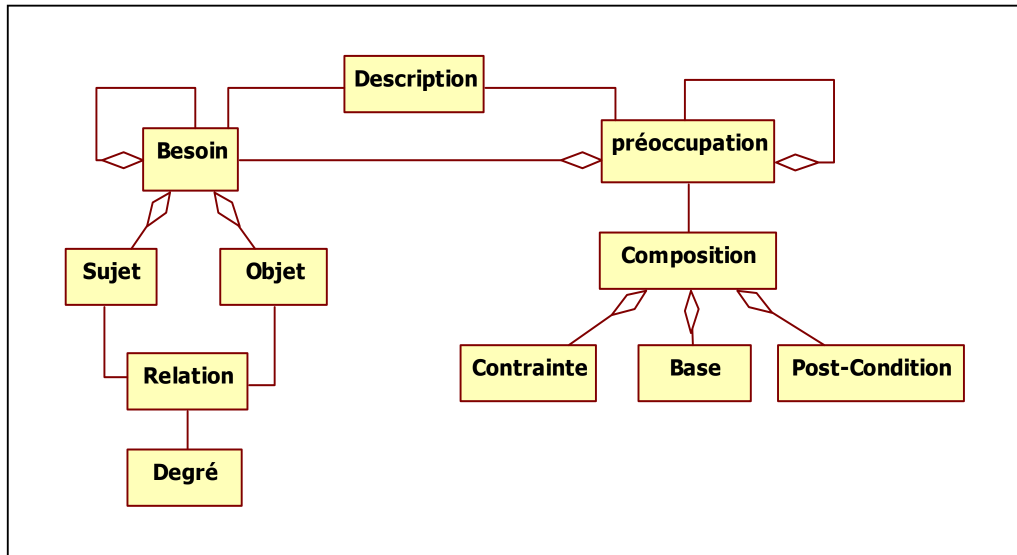


Figure 3.19 : Principaux éléments du méta modèle des RDLs

Dans ce méta modèle, un sujet est l'entité qui entreprend des actions décrites dans une clause. Un objet est l'entité qui est affectée par les actions entreprises par le sujet de la phrase. La classe relation représente l'action exécutée par le sujet sur l'objet (ou les objets). Ces relations peuvent être exprimées par n'importe quel verbe. Les verbes sont exprimés dans des phrases écrites en langage naturel. La structure sujet-relation-verbe (S-R-V) véhicule la principale sémantique que porte la phrase. La classe degré représente l'importance de la relation entre le sujet et l'objet (e.g. maximiser, minimiser, facultative, exclusive, modale, approximative, etc.)

R. Chitchyan et al. [Chitchyan 2006, Chitchyan 2007] ont développé l'outil MRAT (*Multidimensional Requirements Analysis Tools*) qui représente la première tentative dans ce contexte. Cet outil est basé sur le méta modèle précédent et exploite en grande partie les techniques disponibles du traitement des langages naturels.

La classe contrainte spécifie les contrôles et les restrictions qui doivent être imposés sur un ensemble de besoins fournis par un élément de base. Aussi, la clause contrainte exprime sous quelles contraintes une action doit être entreprise et la clause post-condition définit la manière d'imposer des contraintes sur les besoins de base qui doivent être traités.

3.7. Conclusion

Durant les dernières années, plusieurs approches ont été développées dans le cadre de la phase d'ingénierie des besoins. Avant l'apparition du paradigme orienté aspect, ces approches traitent les besoins sous forme d'un ensemble de propriétés fonctionnelles et non-fonctionnelles. Plusieurs approches sont ainsi apparues. Après l'apparition du paradigme relatif à la programmation orienté aspect initiée par Kiczales et al. vers la fin de 1997, plusieurs équipes de recherche, qui travaillent dans le domaine de l'ingénierie des besoins, sont très vite intéressées soit pour adapter les méthodes existantes ou pour développer de nouvelles approches pour la prise en charge du concept aspect dès les premières phases du cycle de vie de développement logiciel.

Dans ce chapitre, nous avons introduit les approches développées durant les deux périodes citées précédemment (avant et après l'apparition du concept aspect) et qui sont consacrées à l'étude et l'analyse des besoins. Nous avons fourni aussi une étude comparative des approches les plus connues dans ce domaine selon un ensemble sélectionné de critères tirés d'une structure méthodologique pour le développement logiciel. Bien que les différentes approches fournissent un processus systématique pour l'identification des besoins, les règles heuristiques qui sont appliquées ne sont pas toujours explicites. Par conséquent, nous pouvons dire qu'un modèle de préoccupation uniforme pour représenter les aspects dans les premières phases qui soit standardisé et applicable dans la chaîne de production de logiciels, n'existe pas encore. Nous avons consacré la dernière section de ce chapitre pour introduire le concept de langage de description d'architecture avec les principaux éléments du méta modèle RDL.

Dans le chapitre 4, nous proposons un modèle pour l'identification, la représentation et la composition des préoccupations. Ensuite, dans le chapitre 5, nous appliquons ce modèle sur un exemple concret.

CHAPITRE 4

HASoCC : Une Nouvelle Approche AORE

4.1. Introduction

Les approches de l'analyse des besoins non orientées aspect existantes ont été développées principalement pour traiter un type de besoins bien précis. Par exemple, PREview [Sawyer 1996, Sommerville 1996] et NFR [Chung 2000] ont souligné l'importance des besoins non-fonctionnels et ont proposé des moyens pour assurer leur accomplissement dans un système, vu qu'elles sont caractérisées par une large influence sur les autres besoins. D'autre part, Problem Frames [Jackson 2001] et les cas d'utilisation (*Use Cases*) [Jacobson 1992], se sont concentrés beaucoup plus sur les besoins fonctionnels d'un système puisqu'ils représentent le noyau fonctionnel du futur système. Ainsi, on peut dire qu'il y a une prise en charge inégale de chaque type de besoins dans les approches précédentes.

Dans ce chapitre, nous présentons notre approche que nous avons baptisée HASoCC (***H****ybrid* ***A****pproach for* ***S****eparation of* ***C****rosscutting* ***C****oncerns, HASoCC*), une approche hybride et orientée aspect pour la séparation des préoccupations transversales avec résolution de conflits entre les préoccupations durant la phase de l'ingénierie des besoins. Dans notre approche, nous avons opté pour les cas d'utilisation pour représenter les besoins fonctionnels, les NFR pour représenter les besoins non-fonctionnels et la démarche de PREview pour décrire les besoins sous forme XML pour un éventuel traitement automatique des besoins et notamment pour valider les résultats obtenus. Aussi, nous insistons sur d'idée que tous les types de besoins sont d'une importance égale et doivent être traités sans discrimination. Nous pensons que cette manière s'impose puisque chaque type de besoins a une influence sur l'autre type.

Cette approche est construite principalement autour de deux phases. Nous avons utilisé le concept de phase comme moyen de classification pour les activités de notre approche.

- Phase d'identification et de spécification des besoins : cette phase s'occupe de la capture, l'identification, la spécification des besoins fonctionnels et non-fonctionnels avec la détermination des cas transversaux dans les deux types de besoins.

- Phase de composition (*weaving*) des besoins : cette phase concerne la détection des éventuels conflits qui peuvent exister entre les besoins transversaux durant les opérations de composition et de résolution des conflits s'ils existent et l'intégration finale des besoins pour construire une image complète et cohérente du futur système. Les artefacts de cette image finale du système seront transformés en artefacts architecturaux qui seront transformés à leur tour en éléments de conception prêt à être implémentés par le développeur de l'application.

Une fois toutes les étapes du modèle seront présentées. Nous présentons dans la sixième section de ce chapitre, une démarche pour transformer les artefacts obtenus dans la phase de l'ingénierie des besoins vers leurs équivalences dans les phases ultérieures du cycle de vie du logiciel, nous insistons tout particulièrement sur leurs équivalents au niveau de la phase qui suis directement l'ingénierie des besoins (phase de description d'architectures logiciels). Nous terminons ce chapitre par une conclusion.

4.2. Motivations

Une approche de développement de logiciels efficace doit harmoniser le besoin de construire le comportement fonctionnel d'un système avec un besoin clair de construire un modèle associé aux NFRs. La plupart des approches actuelles adoptent l'AOSD comme un mécanisme efficace pour traiter les NFR à une étape précoce du processus de développement. Puisqu'on considère les NFR comme les propriétés globales du système, ces propriétés s'entrecoupent dans plusieurs points différents. Ainsi, elles doivent être traitées dans le contexte d'AOSD en tant qu'approche de séparation des préoccupations transversales améliorant la modularité dans des artefacts des systèmes logiciels. Dans ce travail, notre but est de développer un processus orienté aspect, systématique et bien défini qui supporte la capture des besoins aussi bien que l'analyse et la conception des différents besoins fonctionnels et non-fonctionnels. Notre approche vise à réaliser les objectifs suivants :

- Éliminer les lacunes produites à cause de la nature différente des FR et NFR.

- Établir une stratégie systématique pour identifier et résoudre les conflits parmi les préoccupations transversales.

- Identifier la correspondance et l'influence sur les artefacts durant les phases suivantes du cycle de vie du logiciel en identifiant exactement la nature de chaque besoin (e.g. fonctionnel, propriété, contrainte, décision, choix, etc.).

4.3. Etapes du processus

Nous pouvons résumer le processus global de l'ingénierie des besoins dans les étapes suivantes :

Etape 1: Capture

Durant cette phase, on capture les préoccupations telles que l'ingénieur des besoins établit les points d'intérêt de chaque intervenant, c'est-à-dire, leurs intérêts en ce qui concerne un système logiciel donné. Cela peut être obtenu à l'aide des discussions avec les intervenants, des interviews, des observations, des textes, des images, etc., puis complété par des directives comme l'espace des méta préoccupations décrit dans [Moreira 2005b]. Le résultat de cette phase est la liste initiale des besoins généraux des intervenants (par exemple : recherche de données, sécurité, etc.).

Etape 2: Identification

Durant cette phase, l'identification des besoins qui ont été extrait précédemment est effectuée en utilisant par exemple : les catalogues NFR qui nous permettent de définir les besoins engendrés par la préoccupation de sécurité etc. L'utilisation d'outils, comme les outils d'identification des besoins basés sur le traitement des langages naturels (*Natural Language Processing*, NLP) [Sampaio 2005] ou Theme/Doc [Clarke 2005] qui peuvent être très utiles pour cette phase aussi.

Etape 3: Représentation

Durant cette phase, nous représentons séparément les préoccupations identifiées dans la phase précédente. La représentation de ces dernières se fait, généralement, sous forme de graphe de softgoal, de points de vue, de cas d'utilisation, XML, etc.

Etape 4: Raffinement

Chaque préoccupation est raffinée, extraite, identifiée et représentée itérativement. Cette phase s'arrête lorsqu'on obtient un nombre suffisant de préoccupations et de détails sur les besoins des intervenants.

Etape 5: Composition

Durant cette phase, nous composons les préoccupations obtenues dans la phase précédente avec les autres modules du système. Le résultat de la composition servira de base pour identifier les éventuels conflits qui peuvent apparaître entre les préoccupations qui sont représentées d'une manière transversale.

Etape 6: Résolution de conflits

La résolution de conflits entre préoccupations se fait en commun accord avec les intervenants du système (*Trade-off Resolution*) et en parallèle avec le cycle itératif de raffinement (extraction, identification et représentation).

Phase 7: Transformation des préoccupations (Mapping)

Quand un ensemble relativement stable de préoccupations est identifié et représenté et que les opérations de composition sont réalisées avec succès, alors nous pouvons commencer l'étape de transformation de ces préoccupations vers des représentations architecturales et de conception en utilisant un ensemble de directives pour la mener à bien. Il est essentiel que le choix de l'architecture finale satisfasse les attentes des intervenants du système.

4.4. Modèle de l'approche HASoCC

Dans cette section, nous allons présenter l'architecture générale, sous forme graphique, de notre modèle proposé HASoCC pour la séparation des préoccupations. Cette architecture est illustrée par la figure 4.1. Cette architecture comporte quatre modules, à savoir :

Module pour l'identification des besoins

- *Entrée :* l'ensemble des besoins du client dans leurs formes brutes,
- *Sortie :* une liste des besoins fonctionnels et une autre pour les besoins non-fonctionnels (préoccupations).

Module d'analyse et identification des préoccupations transversales

- *Entrée :* Liste complète des besoins identifiés dans l'étape précédente,
- *Sortie :* Liste des préoccupations transversales

Module de tissage (composition des besoins)

- *Entrée :* Liste complète des besoins identifiés dans l'étape précédente,
- *Sortie :* Composition des besoins du système final attendu.

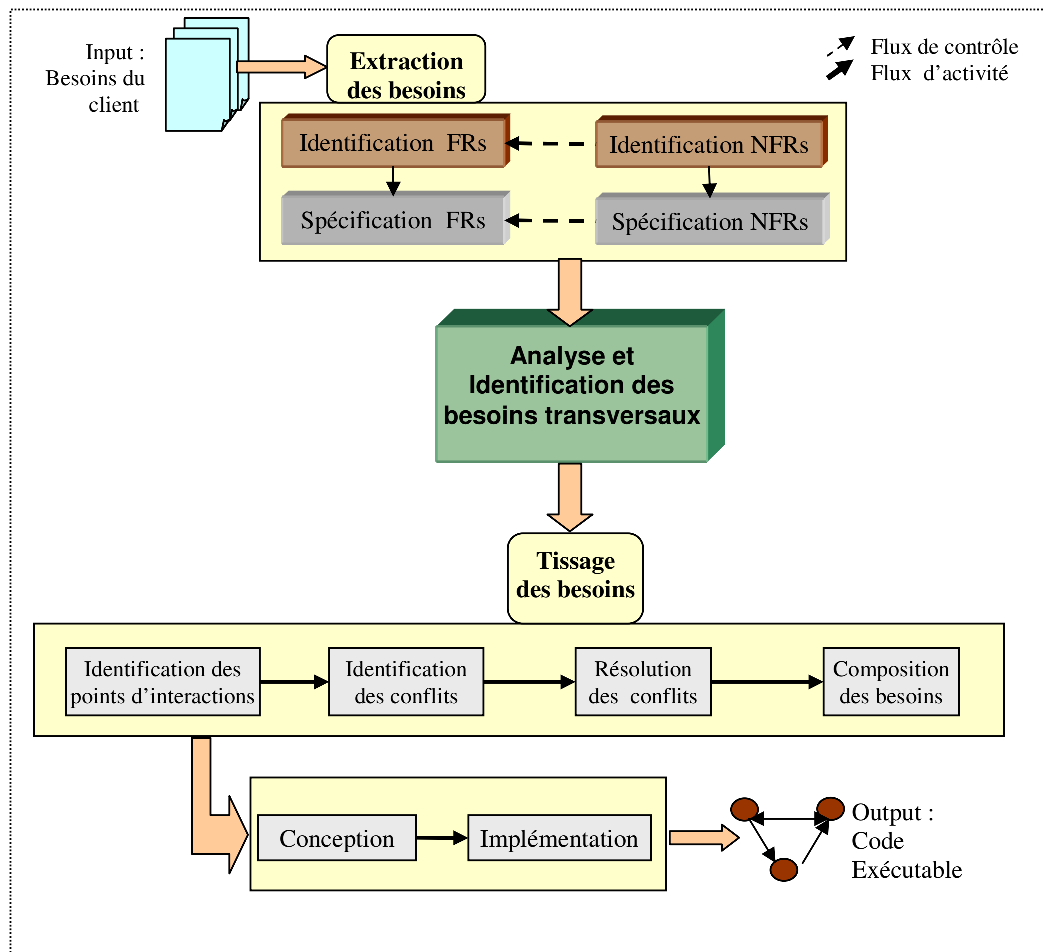


Figure 4.1: Modèle de l'approche proposée

Nous détaillons dans la suite, les étapes de chaque module.

4.4.1. Capture et identification des besoins

Durant cette phase, la capture et l'identification des besoins d'une manière globale sont effectuées (voir figure 4.1). Etant donné que les besoins sont exprimés par les intervenants comme un tout non divisible (besoins fonctionnels et besoins non-fonctionnels mélangés), alors nous avons décidé de prévoir deux tâches parallèles pour identifier en même temps les besoins fonctionnels et les besoins non-fonctionnels. Cette phase peut être décomposée selon les étapes suivantes [Amirat 2006a] :

- Identification des besoins fonctionnels,

- Spécification des besoins fonctionnels,
- Détection des besoins fonctionnels transversaux,
- Identification des besoins non-fonctionnels,
- Spécification des besoins non-fonctionnels,
- Détection des besoins non-fonctionnels transversaux.

4.4.1.1. Identification des besoins fonctionnels

Les besoins fonctionnels expriment le comportement fonctionnel attendu du système à développer. Ce comportement peut être exprimé comme des services, des tâches ou des fonctions que le système doit effectuer. Cette étape peut être effectuée en utilisant l'une des approches existantes comme les points de vue utilisés par PREview, les cas d'utilisations ou les buts. Notons que ces approches ont fait leurs preuves dans le domaine de l'identification des besoins fonctionnels.

Dans notre approche, nous avons opté pour les cas d'utilisation comme approche pour identifier ce type de besoins en construisant les diagrammes des cas d'utilisation en interaction avec des acteurs pour représenter le comportement de haut niveau du système. Via cette image graphique sous forme de diagramme, nous pouvons apporter des éléments de réponses à des questions du genre :

- Que fait le système (comme une boîte noire) ?
- Quel est l'environnement du système ?
- Comment le système est-il utilisé ?

L'identification des besoins fonctionnels est un processus qui engage des discussions avec les intervenants du système, passant en revue des propositions, construisant des prototypes et arrangeant des rencontres consacrées à l'extraction de type de besoins. A la fin de cette phase, nous obtenons un ensemble de cas d'utilisation (*black box*) en interaction avec des acteurs identifiés qui décrivent tout le côté fonctionnel du système.

4.4.1.2 Spécification des besoins fonctionnels

Dans cette étape, nous raffinons encore plus le comportement fonctionnel détaillé de chaque cas d'utilisation avec la description textuelle, la représentation graphique et la spécification formelle. Le résultat de cette activité est l'achèvement de la description de chaque cas d'utilisation sous forme d'un modèle à trous (*template*) (table 4.1). A la fin de cette activité, nous obtenons une représentation graphique complète des interactions entre les acteurs et le système sous forme d'un ensemble de diagrammes de séquence. Ainsi, nous pouvons dire que la représentation contient tous les artefacts de grosse granularité du système [Amirat 2006c].

Attribut	Désignation
UC N°.	Numéro unique pour chaque cas d'utilisation
Nom	Nom du cas d'utilisation
Priorité	Importance du cas d'utilisation
Acteurs	Acteurs primaires et secondaires
Pré-condition (Textuel →Formel)	Description des conditions qui doivent être satisfaites avant l'exécution du cas d'utilisation
Scénario Principale	Séquence d'étapes complètes et uniformes décrivant les interactions entre un utilisateur et le système
Scénario Alternatif	Extensions ou alternatifs du scénario principal
Post-condition (Textuel →Formel)	Description des conditions à satisfaire après l'exécution du cas d'utilisation courant
UC Reliés	Cas d'utilisation dont dépend le cas d'utilisation actuel

Table 4.1. Spécification par « template » des besoins fonctionnels

4.4.1.3. Détection des besoins fonctionnels transversaux

Pour détecter la nature transversale de certains besoins fonctionnels, nous devons examiner pour chaque cas d'utilisation les informations contenues dans la ligne «*Related Use Cases* » de la table 4.1. Si un cas d'utilisation apparaît dans plusieurs autres cas d'utilisation alors il est transversal. Notons que nous pouvons rencontrer un cas d'utilisation qui dépend de plusieurs autres cas d'utilisation via les mécanismes d'extension et d'inclusion, mais il n'est pas de nature transversale.

Le résultat de cette activité est l'ensemble des cas d'utilisation transversaux, s'il existe dans un système donné. Bien évidemment, nous devons résoudre les éventuels conflits qui peuvent exister entre ces cas d'utilisation. Notons que de telles situations peuvent exister, mais elles ne sont pas très courantes.

4.4.1.4. Identification des besoins non-fonctionnels

Les besoins non-fonctionnels qui sont propres au domaine du problème sont capturés en parallèle avec les besoins fonctionnels. Bien que l'extraction des besoins non-fonctionnels puisse être accomplie en utilisant un certain nombre de techniques existantes, nous constatons que la technique la plus reconnue dans ce contexte est celle du catalogue NFR [Chung 2000]. Dans cette technique, nous vérifions l'applicabilité de chaque entrée du catalogue (chaque entrée contient une propriété) pour le système en question.

4.4.1.5. Spécification des besoins non-fonctionnels

La spécification des besoins non-fonctionnels consiste à donner une présentation non ambiguë et formelle aux besoins identifiés dans l'étape précédente. De ce fait, nous pouvons utiliser la représentation XML ou la représentation sous forme de modèle à trous (template). Pour des raisons d'uniformité de la méthode, nous optons pour le modèle de template (cf. table 4.2). Aussi, nous utilisons le concept de préoccupation que celui de besoins pour rester plus général dans le sens de pouvoir représenter les préoccupations qui sont composites. Cette représentation est aussi utilisée avec les préoccupations élémentaires qui sont des besoins simples non fonctionnels [Amirat 2006c].

Attribut	Désignation
Nom	Nom de la préoccupation non-fonctionnelle
Description	Courte description de la préoccupation
Priorité	Importance de la préoccupation
Décomposition	Décomposition des préoccupations composites
Point de Jointure	Point d'appel de la préoccupation au niveau du modèle global associé au système
Besoins requis	Liste des besoins décrivant la préoccupation
Contribution	Nature de la contribution de cette préoccupation vis-à-vis des autres

Table 4.2. Spécification par « template » des besoins non-fonctionnels

Notons que les tables 4.1 et 4.2 doivent être remplies d'une façon incrémentale et itérative au fur et à mesure des opérations d'identification et de raffinement des différents besoins.

4.4.1.6. Détection des besoins non-fonctionnels transversaux

Pour détecter les besoins non-fonctionnels transversaux (*crosscutting concerns*), nous utilisons une matrice d'intersection entre les cas d'utilisation et les NFR identifiés et spécifiés dans les étapes précédentes (cf. table 4.3).

Un simple parcours, colonne par colonne, de la table 4.3 nous permet de détecter les NFR transversales. Une NFR_i est dite transversale si elle entrecoupe au moins deux cas d'utilisation. Le résultat de cet algorithme est un sous ensemble de NFR transversales. Durant les étapes suivantes du modèle, nous allons faire un zoom sur cet ensemble de préoccupations transversales en termes d'analyse et de prise en charge, parce qu'en réalité, ce sont ces préoccupations qui nous intéressent le plus [Amirat 2006b].

	NFR1	NFR2	….	NFR_m
$Use\text{-}Case_1$	✓	✓		
$Use\text{-}Case_2$		✓		✓
….				
$Use\text{-}Case_n$	✓			

Table 4.3 : Cas d'utilisation affectés par NFR

4.4.2. Processus de composition

L'objectif de cette phase est d'assembler tous les besoins identifiés dans la phase précédente dans le but de former un système complet et cohérent au niveau de l'analyse des besoins (cf. figure 4.1). Cet assemblage dans la terminologie de l'ingénierie des besoins est appelé composition des besoins. Par contre dans la terminologie de l'implémentation orientée aspect, il est appelé tissage (*weaving*) ; on dit souvent tisser les aspects (préoccupations non fonctionnels) avec le code base (préoccupations fonctionnels). Actuellement, nous avons même des outils de pré-compilation appelés des weavers.

A son tour, cette phase est composée de quatre étapes qui sont :

- Identification des points d'interaction des NFRs avec les cas d'utilisation,
- Identification des éventuels conflits entre besoins au niveau de chaque point d'interaction,
- Résolution des situations de conflits détectées dans l'étape précédente,
- Intégration finale des besoins raffinés sans aucun conflit.

4.4.2.1. Identification des points d’interaction

En se basant sur les besoins fonctionnels transversaux (définis dans 4.4.1.3) et la correspondance définie entre NFRs et les cas d'utilisation (définis dans 4.4.1.6) qui constitue une autre forme d’entrecoupage, nous pouvons identifier des points d'interaction dans le système où les besoins transversaux se manifestent d’une manière directe.

Nous définissons les ensembles suivants :

UC = {Use Cases}

CCR = {crosscutting FRs} Union {crosscutting NFRs}

Nous définissons aussi la fonction f pour assigner les besoins transversaux à chaque cas d'utilisation affecté. Notons que cette définition est basée sur l'analyse des Tables 4.1, 4.2 et 4.3.

Soit **u** un cas d’utilisation appartenant à **UC** et **cc** un besoin inclus dans **CCR**

f **(u) = Ø** s’il n’ y a aucun besoin transversal lié à **u** et

f **(u) = cc** sinon

L’ensemble des points d’interaction **I** est défini comme suit:

$$I = UC - \{ u \mid f(u) = \emptyset \}$$

Nous pouvons illustrer la transformation (*mapping*) de chaque élément dans **I** vers une liste d’éléments transversaux (table 4.4), ceci est fourni par la fonction f.

Point d’interaction (i_1)	Point d’interaction (i_2)	Point d’interaction (i_3)	….	Point d’interaction (i_n)
ccr_1	*ccr2*	*ccr3*		*ccr1*

Table 4.4 : Points d’interaction dans le système

Ainsi, dans la table 4.4 chaque ccr_i représente un sous-ensemble de l’ensemble des besoins transversaux au niveau du système complet. Chaque sous-ensemble est caractérisé par son point d’interaction avec le reste du système.

4.4.2.2. Définition des conflits

Dans les cas pratiques, il est très rare de trouver un besoin qui se manifeste d'une manière isolée. En principe, la prévision d'un besoin transversal doit affecter le niveau de prévision des autres. Cette dépendance mutuelle est appelée la non orthogonalité des besoins. Pour définir les conflits possibles parmi les besoins, nous déployons une matrice de contribution des préoccupations (table 4.5) basée sur l'ensemble (CCRi x CCRj) d'une façon semblable à celle proposée dans [Rashid 2003]. Pour les besoins qui coexistent au même point d'interaction, nous assignons un signe positif pour une contribution positive et un signe négatif si leur contribution est négative. Nous définissons la fonction « g » pour réaliser la transformation des paires de (CCRi x CCRj) aux valeurs "+", "-", "?" ou " ".

$$g(\mathbf{CCRi \times CCRj}) = \alpha \quad \text{tel que} \quad \alpha \in \{ \text{'+'}, \text{'-'}, \text{'?'}, \text{' '} \}$$

g: function	CCR_1	CCR_2	….	CCR_N
CCR_1		+		-
CCR_2	+			?
….				
CCR_N	-	?		

Table 4.5 : Matrice des contributions des besoins transversaux

Les règles d'assignation des signes aux paires de CCRS sont comme suit :

- Les valeurs '-' , '+' ou ' ' sont assignées à une paire de CCR qui contribue respectivement négativement, positivement ou n'agit pas réciproquement au même point d'interaction. L'attribution de ces valeurs est basée sur le jugement des experts et des développeurs [Amirat 2007a].

- La valeur ' ?' indique un manque d'information sur la contribution, cela peut être mis à jour dans les phases ultérieures du cycle de vie de développement logiciel, ou durant une itération suivante.

Notons que la table 4.5 est une représentation matricielle symétrique de la fonction de transformation g définie précédemment. Il est important de souligner que n'importe quel conflit défini à cette étape peut s'avérer ne pas être un conflit réel quand le système sera raffiné plus loin pendant des phases ultérieures.

4.4.2.3. Résolution des conflits et intégration

Pour chaque point d'interaction ' i ', nous analysons l'ensemble des besoins transversaux et nous étudions la contribution de chacun de ses éléments. Nous nous sommes essentiellement intéressés aux éléments qui représentent des besoins qui ont une interaction négative mutuelle. Nous résolvons le conflit entre ces éléments, en raffinant encore ces besoins dans le but d'éliminer leurs contributions négatives, ou assignons des priorités entre ces éléments pour déterminer l'ordre de leur exécution.

Dans cette étape, nous intégrons les besoins fonctionnels et non-fonctionnels pour obtenir le système entier. A ce haut niveau d'abstraction, nous pouvons utiliser le diagramme de cas d'utilisation d'UML pour modéliser l'interaction entre les besoins et leur composition. Dans le nouveau diagramme de cas d'utilisation structuré, nous utilisons le stéréotype << include >> pour chaque NFR qui a un ensemble de cas d'utilisation transversaux qui sont inclus dans d'autres nouveaux cas d'utilisation.

Nous utiliserons les règles de composition pour définir l'ordre dans lequel les besoins seront appliqués dans un point de jointure particulier. Généralement, une règle de composition prend la forme suivante :

<Besoin$_1$> <Opérateur> <Besoin$_2$>

La sémantique des opérateurs déployés entre les besoins est inspirée du langage de spécification LOTOS [Brinksma 1988]. La liste des opérateurs que nous avons utilisé est donnée dans la table 4.6.

OPERATEUR	DESIGNATION
Before	Indique que besoin$_1$ doit être satisfait avant la satisfaction de besoin$_2$
After	Indique que besoin$_1$ doit être satisfait après la satisfaction de besoin$_2$
Override	Indique que le comportement décrit par besoin$_1$ se substitut au comportement défini par besoin$_2$
Parallel	Indique que le besoin$_1$ doit être satisfait en parallèle (synchronisé) avec le besoin$_2$
Wrap	Indique que le besoin$_1$ encapsule le besoin$_2$, c'est-à-dire le comportement associé à besoin$_1$ s'exécute avant et après le comportement de besoin$_2$
Interleave	Indique que besoin$_1$ sera satisfait durant un laps de temps qui ne chevauche pas celui de besoin$_2$

Disable	Signifie que le comportement du $besoin_2$ interrompt le comportement du $besoin_1$
Enable	Signifie que l'activité associée à $besoin_2$ commence lorsque l'activité associée à $besoin_1$ se termine avec succès

Table 4.6: Différents opérateurs de composition des besoins

4.5. Validation des besoins identifiés

Dans cette section, nous proposons un modèle pour la validation des besoins identifiés dans le modèle précédent. La validation visée dans ce travail est une validation par comparaison des résultats obtenus en appliquant notre approche avec les résultats obtenus en appliquant l'approche orientée but et l'approche orientée cas d'utilisation. L'idée principale de ce modèle de validation se base sur un graphe de validation.

Pour préparer les besoins à un traitement automatique, on doit les spécifier dans le format XML. Ceci nous conduit à prévoir un module (M1) pour générer un fichier XML contenant les besoins identifiés par chaque approche. Un module (M2) reçoit en entrée ces trois fichiers XML à base desquels un fichier XML commun est synthétisé par comparaison.

En réalité, la comparaison consiste à vérifier le nombre des besoins identifiés pour chaque préoccupation. Si les fichiers contiennent des besoins similaires, ils sont alors fusionnés pour générer un fichier commun. Notons que ce fichier résultat ne doit pas contenir des besoins redondants. Si le module (M2) est incapable de fusionner ces trois fichiers, alors ces derniers fichiers sont passés comme entrée à un autre module (M3). Le module (M3) applique un nombre de règles heuristiques pour résoudre les conflits qui peuvent exister entre les besoins et génère un fichier XML pour chaque préoccupation. Les règles heuristiques manipulées par M3 sont très basiques de types (fixer un besoin, supprimer un besoin, déplacer un besoin, etc.).

Pour implémenter ces trois modules, nous pouvons utiliser deux techniques : soit l'API XML disponible au niveau de Java, soit la technique DOM (*Document Object Model*).

- L'API – XML Java nous permet de lire le fichier XML et de générer un évènement à chaque balise rencontrée dans le texte. Nous pouvons exploiter chaque évènement en lui associant le traitement désiré dans chaque module. Principalement, nous devons prévoir des routines de génération de graphes pour transformer le fichier XML en graphe. L'avantage de cette technique est que le nombre de besoins à traiter n'est pas limité.

- La technique DOM nous permet de générer directement un graphe d'objet en mémoire. Chaque objet est associé à une balise dans le fichier XML. Ensuite, il est facile de manipuler et de parcourir ces objets en mémoire pour réaliser les traitements nécessaires pour chaque module. La limite de cette technique est que le graphe généré est limité à la taille de la mémoire dynamique disponible.

4.6. Le Modèle proposé (HASoCC) et les approches existantes

Nous avons présenté dans la section 3.5 du troisième chapitre, un tableau comparatif (table 3.2) entre les différentes approches développées dans le cadre de l'ingénierie des besoins. Cette comparaison est fondée sur un certain nombre de critères définis par *Filman et al.* pour l'évaluation des approches d'analyse des besoins [Filman 2004]. Il est intéressant de placer notre modeste contribution par rapport aux approches existantes. Cette initiative est illustrée dans table 4.7. Ce qui manque à notre modèle, ce sont les outils supports qui doivent être développés.

Ces outils donneront, en principe, une dimension réelle à notre travail. Notons que ces outils ne peuvent être que semi-automatiques. L'intervention de l'utilisateur est toujours nécessaire durant les premières phases du cycle de développement de logiciels. Ces outils sont prévus pour l'identification, la représentation et la composition. Ces outils doivent inclure, entre autres, des modules pour la détection et la résolution des éventuels conflits, avec une phase de validation des résultats obtenus.

Approche HASoCC « Approche Orientée Aspect » Approche Hybride						
Support des Aspects	Activité dans le cycle de vie	Règles de composition	Traitement des conflits	Maturité	Passage aux artefacts postérieurs	Outils de supports
√	Ingénierie des besoins, Architecture	√	√	X	√	---

Table 4.7 : HASoCC par rapport aux critères définis dans la section 3.5

Nous avons des travaux en cours pour le passage des artefacts de l'ingénierie des besoins vers la phase d'architecture, et l'introduction de la notion d'aspect au niveau d'un langage de description d'architectures (l'ADL COSA [Smeda 2006]) en utilisant les concepts architecturaux (composants, connecteurs et configuration) et les mécanismes de l'orienté objet (l'héritage, l'instanciation et la généralisation).

4.7. Projection des éléments de l'ingénierie des besoins vers la phase d'architecture

Actuellement, il existe plusieurs approches orientées aspect disponibles que nous pouvons appliquer séparément dans les différentes phases de développement des logiciels. Ces approches ont conduit à un large fossé entre des propositions à différents niveaux d'abstractions. Par conséquent, on constate l'apparition de nouveaux problèmes lorsqu'on essaie de faire transiter les artefacts d'une phase donnée à la phase suivante dans le cycle de développement du logiciel.

La conséquence de ce constat se ressent lors de la dérivation d'une conception architecturale orientée aspect (*Aspect Oriented Architecture*, AOA) à partir d'une spécification de besoins orientée aspect (*Aspect Oriented Requirements*, AOR) du moment que certains éléments de la spécification des besoins ne peuvent pas être projetés directement vers des éléments architecturaux [Amirat 2007b].

Dans cette section, nous expliquons comment nous pouvons réduire ce fossé en utilisant des modèles de transformation pour obtenir des modèles d'architectures orientés aspects à partir des modèles de l'ingénierie des besoins orientés aspect. Notons que ces transformations doivent préserver au maximum la sémantique de la spécification des besoins. Pour supporter l'automatisation de cette transformation, on doit exploiter les avantages du développement dirigé par les modèles (*Model Driven Development*, MDD) [OMG 2001b].

Etape 1 :

Construire un modèle UML qui correspond à la spécification des besoins raffinés obtenus. Notons qu'il y a aussi une autre possibilité qui consiste à utiliser le profil UML développé par Aldawud et *al.* [Aldawud 2003] pour modéliser les besoins orientés aspect. En utilisant ce profil, les besoins orientés aspect sont modélisés par un ensemble de scénarios et chaque scénario peut avoir plusieurs besoins non-fonctionnels. Ainsi, le résultat de cette étape est un modèle de scénarios.

Etape 2 :

Le modèle scénario obtenu dans l'étape précédente est transformé en un modèle d'architecture orienté aspect en utilisant des transformations prédéfinies. Le modèle d'architecture est construit d'une manière incrémentale par la transformation de chaque scénario séparément.

Le modèle architectural peut être décrit par le profil UML 2.0 pour CAM (*Component and Aspect Model*) développé par Pinto et *al.* [Pinto 2005], l'une des rares approches disponibles actuellement. Le modèle de transformation est décrit par le standard QVT (*Query, View, Transformations*) [Bezivin 2006]. Toutes ces étapes sont décrites par le processus représenté dans la figure 4.2.

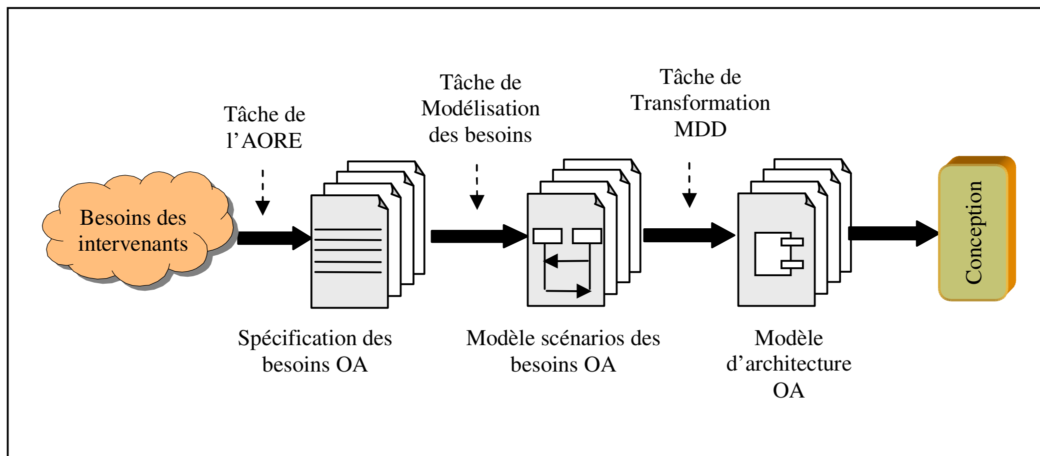


Figure 4.2 : Processus de transformation d'un modèle AOR vers un modèle AOA

Pour maître en œuvre le processus précédent, nous avons besoin de certaines règles de projection de concepts déployés au niveau de l'ingénierie des besoins vers d'autres concepts équivalents au niveau d'une architecture à base de composants. Les concepts utilisés par l'architecture à base de composants seront à leur tour projetés vers les éléments disponibles au niveau du langage AspectJ ou un autre. Nous avons choisi AspectJ vu sa popularité au sein de la communauté des développeurs.

La table 4.8 montre les règles de passage qui nous permettent d'effectuer la transition entre les concepts de l'analyse des besoins et les concepts architecturaux d'un côté, et de l'autre côté, entre les concepts architecturaux et AspectJ [Amirat 2007b].

INGENIERIE DES BESOINS	**ARCHITECTURE**	**ASPECTJ**	**EXPLICATION**
Préoccupation	Configuration	Classe	Une préoccupation est réifiée en une classe Java.
Besoin fonctionnel	Composant	Classe	Un besoin est implémenté par une classe.
Besoin non-fonctionnel (NFR)	Composant aspectuel	Aspect	Les NFR sont implémentés par des aspects.
Relation entre besoins fonctionnels	Connecteur	Classe	Les relations entre besoins sont implémentées par des classes.
Relation entre besoins non-fonctionnels	Connecteur aspectuel	Classe	
Activité	Service	Méthode	Une méthode simule une activité.
Point de jointure	Port/Rôle	Joinpoint	Les points de jointure représentent des ports et des rôles.
Point de coupe	Glue	Pointcut	Les points de coupe sont implémentés par des méthodes.

Table 4.8 : Règles de passage (*mapping*) entre RE – Architecture - AspectJ

4.8. Conclusion

L'enchevêtrement et la dispersion sont des symptômes qui n'affectent pas uniquement l'implémentation, mais ils apparaissent initialement dans les premières phases du processus de développement du logiciel.

L'identification et la modélisation précoce des besoins transversaux ont un grand impact sur l'amélioration de la qualité générale du système par l'augmentation de la compréhensibilité et de la réutilisabilité et par la réduction du temps de développement, en réduisant la complexité par le fait d'améliorer le processus de détection et de résolution de conflits.

Dans ce chapitre, nous avons présenté l'approche HASoCC sous forme d'une séquence ordonnée d'activités systématiques pour prendre en compte de manière précoce l'identification, la spécification et la séparation des besoins transversaux qui doivent être modularisés pour permettre une bonne traçabilité durant toutes les phases du cycle de développement.

Nous nous sommes intéressés aux deux types de préoccupations fonctionnelles et non-fonctionnelles comme des besoins candidats qui peuvent être transversaux. Nous avons aussi proposé la spécification des besoins fonctionnels par extension de la spécification des cas d'utilisation avec la précision d'une définition formelle pour la description des assertions de pré-conditions et de post-conditions (table 4.1).

Pour composer les besoins, nous avons fourni une approche à granularité fine pour définir les points d'interaction et les relier aux cas d'utilisation. En résumé, notre approche permet une reconnaissance précoce des besoins et permet de résoudre les éventuels conflits qui peuvent surgir durant l'activité de composition des besoins. Dans le chapitre suivant, nous montrons la mise en œuvre du modèle et les expérimentations que nous avons effectuées.

CHAPITRE 5

Expérimentations

5.1. Introduction

Le développement de logiciels orientés aspect (AOSD) vise à pallier aux conséquences négatives dues à l'existence des préoccupations transversales au niveau des besoins. Ce nouveau paradigme (AOSD) a comme objectif le développement de méthodologies, de modèles et d'outils qui permettent une bonne prise en charge de la phase de l'ingénierie des besoins. Notons que le succès ou l'échec du produit final dépend en grande partie des résultats de cette phase. Les différentes activités concernées par cette phase sont :

- Identification systématique des besoins,
- Séparation des besoins,
- Représentation des besoins et
- Composition.

Au terme de cette phase, les besoins transversaux sont encapsulés dans des modules séparés, connu sous le nom d'aspects. Ainsi, la localisation et la manipulation de ces besoins, sous forme de modules, sont désormais une tâche facile à réaliser. Nous pouvons justifier ce constat par le fait d'avoir actuellement toute une bibliothèque de langages orientés aspect (AspectJ, HyperJ, AspectualC, etc.). Parmi les résultats positifs attendus de cette modularisation, nous pouvons citer : la réduction des coûts des opérations de maintenance, la simplification de l'évolution globale des besoins du système et la promotion des lignes de production de logiciels (*Software Product Lines*).

Il est évident que faire évoluer un système revient simplement à faire évoluer les besoins initiaux qui déclenchent une série d'opérations d'évolution dans le reste des phases du cycle de vie du système. Actuellement, l'utilisation des styles d'évolution

et les modèles d'évolution que nous pouvons appliquer durant la phase de l'ingénierie des besoins, reste un des points forts de ce domaine.

Notons aussi qu'il y a d'autres styles d'évolutions applicables durant les phases d'architecture (C*lient Server*, *Pipe and Filter*, *Blackboard*) [Shaw 1997] et de conception (*Design pattern de Gang of For*) [Gamma 1999]. Aussi, des styles d'évolution sont définis pour faire évoluer les modèles d'évolution des modèles d'analyse des besoins et d'architectures.

Actuellement, plusieurs tentatives de développement de langages, de mécanismes, de méthodologies basés sur des approches différentes essayent d'introduire le concept d'aspect dans les paradigmes existants comme le développement de logiciels à base d'objets (OOSD) et le développement de logiciels à base de composants (SDCB) et prendre en charge les besoins transversaux par de nouvelles approches ou bien les intégrer dans des approches existantes.

Mais le problème constaté jusqu'à ce jour est le manque de maturité de toutes ces approches. Ceci retarde encore l'apparition de normes et de standards ainsi que des consensus pour fédérer les différents concepts et mieux les encadrer pour pouvoir généraliser et automatiser leur utilisation et développer des outils supports pour ces derniers.

Dans la deuxième section de ce chapitre, nous allons appliquer les phases de notre modèle HASoCC présenté dans le chapitre 4 sur une version simplifiée du système de gestion du trafic des passages dans les stations métro [Araujo 2003]. Le suivie de notre méthodologie va nous permettre d'extraire toutes les préoccupations pertinentes de ce système. La dernière section de chapitre sera consacrée à une conclusion sur cette expérimentation.

5.2. Etude de cas (*exemple du système métro*)

Notre objectif dans cette section est de valider l'approche et d'illustrer la représentation des concepts en utilisant une version simplifiée du système de transport métro [Araujo 2003].

Les exigences au plus haut niveau pour le système métro sont comme suit :

- Pour utiliser le métro, un client doit posséder une carte qui doit être créditée avec une certaine somme d'argent.

- Une carte est achetée et créditée en utilisant des machines d'achat spéciales, disponibles dans n'importe quelle station de métro.

- Un client introduit cette carte dans une machine disponible à l'entrée de la station métro pour initialiser son voyage.

- Quand le client arrive à sa destination, la carte est insérée dans une machine disponible à la sortie de la station. Cette machine débite la carte avec une somme d'argent qui dépend de la distance parcourue « voyage ».

- Si la carte ne contient pas assez de crédits, les portes ne s'ouvrent pas à moins que le client réapprovisionne sa carte avec la somme d'argent nécessaire.

- Le client peut demander un remboursement du montant d'argent disponible dans la carte en la réintroduisant dans une machine distributrice d'argent.

5.2.1. Identification des besoins fonctionnels

En se basant sur les besoins exprimés ci-dessus, nous prenons en considération une simple situation où uniquement l'intervenant *Client* est pris en compte et nous supposons que le système doit offrir les besoins fonctionnels suivants :

N°	Besoin	Description
1	*BuyCard*	Achat de cartes
2	*LoadCard*	Chargement de cartes dans la machine
3	*RefundCard*	Remboursement de l'argent disponible dans le compte de la carte
4	*EnterSubway*	Entrée dans une station métro
5	*ExitSubway*	Sortie d'une station métro
6	*ValidateCard*	Validation d'une carte

Table 5.1 : Besoins fonctionnels identifiés pour le système métro

Pour chacun de ces besoins fonctionnels nous devons, en principe, remplir les entrées de la table 4.1 « *template* » du chapitre 4. Dans notre cas, nous monterons seulement le template « *EnterBubway* » (table 5.2) qui nous permet d'illustrer par la suite les préoccupations transversales et les situations de conflits possibles.

Besoin N°. 4	
UC N°.	Numéro 4
Nom	EnterSubway
Priorité	Très importante
Acteurs	Client, le propriétaire du système
Pré-condition (Textuel →Formel)	- Le client dispose d'une carte - La machine est libre
Scénario Principal	Pour commencer le voyage le client insère sa carte dans la machine disponible à l'entrée du métro. Le système vérifie la carte et enregistre une entrée.
Scénario Alternatif	<Néant>
Post-condition (Textuel →Formel)	La carte client soit enregistrée et contient une somme d'argent
Besoins Reliés	**FR :** ValidateCarte **NFR :** ResponceTime, Accuracy, Security, Availability.

Table 5.2 : Spécification par « template » du besoin EnterSubway

5.2.2. Spécification des besoins fonctionnels

Plusieurs approches peuvent être utilisées pour spécifier les besoins fonctionnels (voir chapitre 3). Nous avons choisi l'approche dirigée par les cas d'utilisation [Jacobson 1992]. Les cas d'utilisation décrivent la manière dont un utilisateur utilise un système donné, c'est pourquoi le diagramme de cas est souvent utilisé pour capturer les besoins fonctionnels. Le diagramme de cas d'utilisation correspondant à ces besoins est illustré sur la figure 5.1.

Dans ce diagramme, nous avons divisé le système métro en deux sous-systèmes avec uniquement (*Client)* comme acteur. Le premier sous-système contient les deux cas d'utilisation (*BuyCard* et *RefundCard*) et le deuxième sous-système contient les trois cas d'utilisation (*EnterSubway*, *ExitSubway* et *LoadCard*) qui sont assez raffinés dans le but de factoriser la fonctionnalité commune (*ValidateCard*).

Notons que les différents acteurs du système sont :

1. Client,
2. Machine,
3. CarteData.

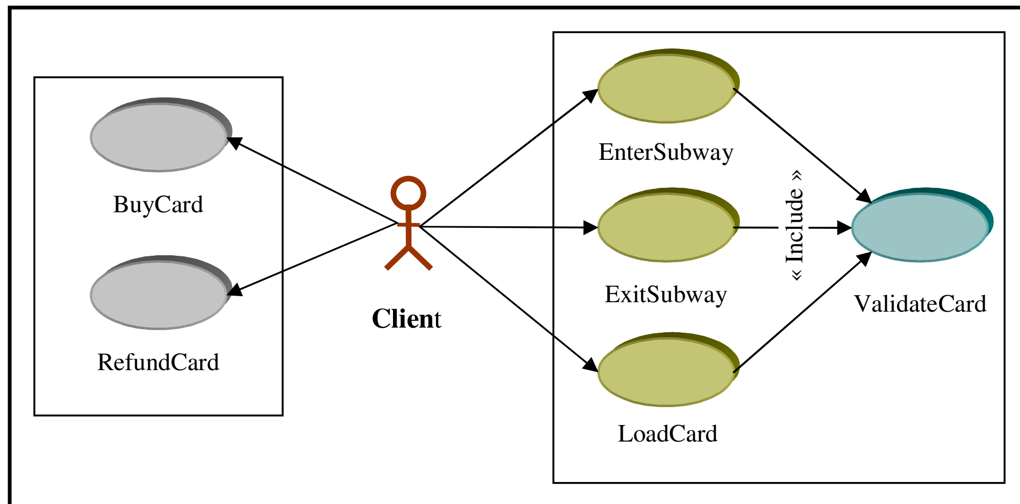


Figure 5.1 : Diagramme de cas d'utilisation du système métro

5.2.3. Identification des besoins non-fonctionnels

Si nous raisonnons en termes de cas d'utilisations, par exemple, les besoins inscrits dans la table 5.1 ci-dessus ne sont pas trop difficiles à identifier. Par contre, d'autres préoccupations peuvent être identifiées si on utilise une approche basée sur le catalogue NFR [Chung 2000]. Pour chaque entrée dans le catalogue, nous devons décider si elle est utile dans notre système ou non. La table 5.2 illustre les entrées utiles pour notre cas d'étude. Ainsi, les préoccupations représentées dans cette table ne sont autres que les besoins non-fonctionnels du système.

N°	Besoin	Description
1	*ResponseTime*	Temps de réponse du système : le système doit réagir aux actions du client dans un temps court et raisonnable
2	*Accuracy*	Exactitude : le système doit débiter la somme exacte (sans erreur) d'une carte
3	*Multi-Access*	Accès multiple : plusieurs passagers peuvent utiliser simultanément le système
4	*Availability*	Disponibilité : le système et les machines doivent être disponibles aux entrées et aux sorties des stations métro.
5	*Security*	Sécurité : les informations de la carte doivent être protégées contre des actions illicites

Table 5.3 : Besoins non-fonctionnels identifiés pour le système métro

Pour chacune de ces préoccupations nous devons, en principe, remplir les entrées de la table 4.2 « *template* » du chapitre 4. Dans notre cas, nous montons seulement les templates qui nous aident à illustrer les préoccupations transversales et en situation de conflit possible.

Les préoccupations non-fonctionnelles sélectionnées sont : le temps de réponse (table 5.2) et la disponibilité (table 5.3).

Préoccupation N°. 1	
Nom	**Response time**
Description	Le système doit réagir à temps et dans des situations spécifiques ; (e.g. entrer dans une station, quitter une station)
Priorité	Très importante
Décomposition	<Néant>
Point de jointure	A l'entrée et à la sortie d'une station métro
Besoins requis	*EnterSubway, ExitSubway*
Contribution	(-) *Security*, (-) *Multi-access*.

Table 5.4 : Spécification par « template » du temps de réponse

Préoccupation N°. 4	
Nom	**Availability**
Description	Le système et les machines doivent être disponibles aux entrées et aux sorties des stations métro
Priorité	Très importante
Décomposition	Néant
Point de jointure	A l'entrée et à la sortie d'une station métro, achat de carte chargement de carte et remboursement de carte
Besoins requis	<Néant>
Contribution	(+) *Multi-access*, (-) *Response Time*

Table 5.5 : Spécification par « template » de la disponibilité

Durant cette tâche, nous devons aussi identifier les différentes contributions entre les préoccupations, les priorités de chacune et la liste des sous-besoins pour accomplir le comportement d'une préoccupation donnée. A la fin de cette tâche, le modèle de template de chaque besoin doit être, en principe, complet.

5.2.4. Spécification des besoins non-fonctionnels

Pour spécifier ces besoins, nous pouvons utiliser un graphe hiérarchique montrant leur interdépendance (*Softgoal Independency Graph*, SIG) [Chung 2000]. Le SIG est développé dans le cadre de l'approche NFR Framework section 3.3.2.1 du chapitre 3. Dans la figure 5.2, nous présentons la spécification de la préoccupation sécurité en relation avec les autres préoccupations et nous mettons entre crochets le sujet adressé par chaque préoccupation à savoir *EnterSubway, Machine* et *CardData*. Ceci signifie que la sécurité est prise en compte pour le besoin fonctionnel *EnterSubway*, *Integrity*, *Completeness* et *Accuracy* sont garanties pour *CardData* et *Availability* doit être garantie pour *Machine.* Enfin, *Accuracy* d'une carte de données peut être opérationnalisée par une procédure de validation des données de la carte.

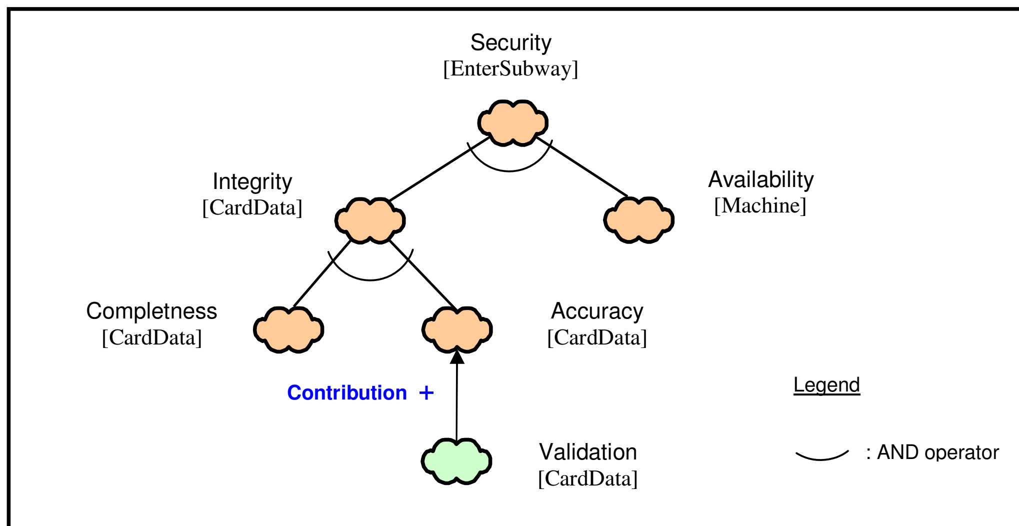


Figure 5.2 : Spécification du besoin sécurité pour le cas d'utilisation *(*EnterSubway*)*

Les nœuds du SIG sont alors des préoccupations non-fonctionnelles représentées par des nuages. Les lignes entre les nœuds représentent la décomposition de chaque préoccupation en sous-préoccupations (filles) [Chung 2000].

Quand toutes les sous-préoccupations filles d'une préoccupation mère donnée sont nécessaires pour réaliser le comportement de cette dernière (mère), une relation booléenne de type AND est définie avec un arc connectant toutes les lignes de décomposition de cette préoccupation mère (figure 5.2).

Dans la situation où toutes les sous-préoccupations filles sont pas nécessaires pour réaliser le comportement de la préoccupation mère, une relation booléenne OR est définie avec deux arcs liant les lignes de décomposition

Dans le système métro, *Integrity* et *Availability* sont les deux sous-préoccupations nécessaires pour réaliser la préoccupation *Security*, ainsi un opérateur AND est défini entre elles (*Integrity*, *Availability*). En outre, *Accuracy* et *Completeness* sont nécessaires pour accomplir la préoccupation *Integrity*.

5.2.5. Identification des besoins transversaux

Les préoccupations transversales peuvent être identifiées de deux manières :

- En examinant les entrées « besoins requis » et « description » dans les tables de type templates dressées pour l'identification des préoccupations non-fonctionnelles,

- En examinant les diagrammes des cas d'utilisation relatifs aux besoins fonctionnels.

Parmi les préoccupations transversales dans le système métro, nous trouvons :

- ***ResponseTime*** est une préoccupation non-fonctionnelle transversale parce qu'elle est requise pour l'accomplissement des besoins fonctionnels suivants : *EnterSubway* et *ExitSubway* (table 5.4).

- ***ValidateCard*** est une préoccupation fonctionnelle transversale du moment qu'elle est requise dans la réalisation de *EnterSubway*, *LoadCard* et *ExitSubway* (figure 5.1).

- *Availability, Multi-Access*, *Security* et *Accuracy* sont aussi des préoccupations non-fonctionnelles transversales.

5.2.6. Composition des besoins

L'objectif de cette étape est de composer tous les besoins pour obtenir le système complet. Lors de cette opération de composition, des conflits entre préoccupations peuvent apparaître. Alors, s'il y a des conflits qui surgissent, nous devons les résoudre dans l'étape suivante :

5.2.6.1. Identification des points de jointures

Les points de jointures sont identifiés en utilisant la table suivante :

STAKEHOLDER	CONCERN			
	EnterSubway	*BuyCard*	*ValidateCard*	…
<Client>	RT,S,AC,AV,V **(JP_A)**	S,AC,AV **(JP_B)**	AC	

(*AC:* Accuracy*; RT:* Response Time*; S:* Security*; AV:* Availability*; V:* ValidateCard; JP: Join Point)

Table 5.6 : Identification des points de jointures

5.2.6.2. Identification des conflits

Quand une case (table 5.6) contient plus d'un élément (préoccupation transversale) alors nous devons vérifier la nature des contributions qui peuvent exister entre ces derniers. Ainsi, et d'après la table 5.6, nous pouvons identifier, à titre d'exemple, les points jointures suivants :

- **MP_A :** À ce point de jointures, nous avons cinq préoccupations transversales RT, S, AC, AV et V. Notons qu'entre ces préoccupations, la disponibilité et le temps de réponse contribuent mutuellement d'une manière négative (i.e. ils contraignent le comportement de chacune et elles peuvent produire des conflits).

- **MP_B :** À ce point de jointures, nous avons trois préoccupations transversales, à savoir S, AC et AV.

5.2.6.3. Identification de la préoccupation dominante

Pour résoudre les conflits identifiés ci-dessus, nous devons vérifier la priorité de chaque préoccupation impliquée dans ce conflit. Prenons comme exemples, le cas des préoccupations : disponibilité et temps de réponse qui ont la même priorité (*Très Importante*).

Nous recourons dans cette situation à la négociation d'un compromis entre les intervenants du système. Ainsi, nous supposons que la priorité du temps de réponse est diminuée par rapport à la disponibilité. Par conséquent, la disponibilité devient la préoccupation la plus dominante au niveau du point de jointure $\mathbf{JP_A}$.

Cependant, comme la disponibilité est une sous-préoccupation de la préoccupation sécurité, nous devons éclaircir si la sécurité est aussi dominante par rapport au temps de réponse. Du moment que la sécurité est composée de deux sous-préoccupations (*Availability* et *Integrity*), alors nous devons garantir dans un premier temps *Availability* et ensuite *Integrity*. Le temps de réponse est une préoccupation dont nous avons besoin entre *Availability* et *Integrity*.

5.2.6.4. Règles de composition

Une règle de composition au point de jointures $\mathbf{JP_A}$ (figure 5.3) doit être définie en utilisant les opérateurs de composition définis dans la table 4.6 du chapitre 4.

```
(Security. Availability
   before
   (
     (ValidateCard  before  EnterSubway )
      Parallel
     ResponseTime
      Parallel
     Security.Integrity.Accuracy
   )
  before
Security. Integrity )  Disable  ErrorHandling.
```

Figure 5.3 : Règle de composition pour $\mathbf{JP_A}$

Nous utilisons le point "." comme notation de sélecteur pour désigner une sous-préoccupation d'une préoccupation donnée. Ainsi, nous pouvons lire cette règle de composition d'une façon informelle et suivant l'ordre indiqué par les points suivants :

1. Avant la satisfaction du besoin *Availability*, ce sont tous les autres besoins qui doivent d'être satisfaits, cette priorité et dûe à l'opérateur **«Before ».**

2. *EnterSubway*, *ResponseTime* et *Accuracy* doivent être synchronisées et satisfaites en parallèle (en utilisant l'opérateur **«Parallel** »).

3. Nous remarquons que *EnterSubway* peut seulement être satisfait après la satisfaction avec succès de *ValidateCard* (à cause de l'opérateur **« Before** »).

4. Lorsque tous les autres besoins sont satisfaits, *Integrity* doit être satisfait.

5. Si le comportement d'une préoccupation ne se termine pas d'une manière normale durant le processus de composition, le système doit être capable de générer les erreurs qui peuvent apparaître. Dans notre cas, nous modélisons cette situation par la préoccupation *ErrorHandling* qui permet d'interrompre n'importe quel ensemble de préoccupations suivies par l'opérateur *Disable*.

5.3. Conclusion

Dans ce chapitre, nous avons expérimenté le modèle HASoCC défini dans le chapitre 4. Nous avons utilisé ce modèle pour effectuer une séparation très avancée des préoccupations durant la phase de l'ingénierie des besoins du cycle de développement de logiciels. Principalement, et dans un premier temps, nous avons appliqué une approche dirigée par les cas d'utilisation dans l'identification et la spécification des besoins fonctionnels du système. Dans un second temps, nous avons utilisé le catalogue de l'infrastructure NFR pour identifier et spécifier les préoccupations non-fonctionnelles.

Une fois toutes les préoccupations identifiées et représentées d'une manière formelle, alors nous avons appliqué une méthodologie simple de composition basée sur un ensemble prédéfini d'opérateurs de compositions, pour obtenir enfin le système complet. Comme des conflits entre préoccupations peuvent apparaître au cours de cette opération de composition. Si tel est le cas, nous appliquons l'algorithme de sélection de la préoccupation dominante parmi celles qui sont en conflit.

Nous constatons que la séparation entre les informations relatives à la composition et les préoccupations d'un coté et les cas d'utilisation de l'autre coté, rend fortement indépendants les éléments de base représentants les besoins des intervenants. Ceci fournit, en soi, une séparation très avancée des préoccupations. Cela améliore aussi la réutilisabilité des préoccupations dans certains projets de développement de logiciels. Par exemple, la préoccupation temps de réponse de l'exemple précédent n'est pas spécifique uniquement au système métro. Elle peut être réutilisée pour spécifier des besoins de temps de réponse dans d'autres systèmes de logiciels. Aussi, la préoccupation disponibilité peut être réutilisée dans d'autres systèmes.

Puisque les règles de composition agissent à une granularité individuelle sur chaque besoin, alors il est possible d'identifier et de gérer les conflits à une granularité fine. Nous constatons aussi, que les opérateurs aident à maintenir une spécification non formelle des besoins tout en fournissant une sémantique de composition bien définie.

Conclusion et Perspectives

Bilan

Les travaux décrits dans ce manuscrit portent sur le développement d'une méthodologie basée sur un sur processus d'identification et la représentation des besoins fonctionnels et non-fonctionnels, transversaux ou de base. Analysons ce qui a été fait au niveau de ces travaux.

Au chapitre 1, nous avons exposé les approches de prise en charge des préoccupations transversales, à travers les travaux les plus significatifs dans le domaine. La programmation par aspects est un paradigme très récent qui a été conçu pour combler certaines limites du paradigme objet (voire même procédural ou fonctionnel). Nous avons montré à travers ce chapitre que l'orienté aspect apporte une solution aux problèmes de dispersion et d'enchevêtrement du code. Ces deux problèmes sont la conséquence de l'intégration des propriétés non-fonctionnelles dans le code. Dans cette perspective, trois propositions principales sont apparues à savoir la programmation par aspect, la programmation multi-dimensionnelles et la composition de filtres.

Par ailleurs, plusieurs langages de programmation orientés aspect conçus à base des idées présentées dans les trois approches pour la prise en charge des préoccupations transversales, en les modélisant sous forme de modules appelés parfois Aspects. Bien que les travaux dans ce domaine soient relativement récents, quelques réalisations sont apparues ces dernières années. Dans ce contexte, nous avons décrit certains langages et nous avons donné leurs principaux concepts et leurs caractéristiques.

Par conséquent, nous avons consacré le chapitre 2 à la description des trois langages les plus connus et développés respectivement par les concepteurs de ces approches à savoir AspectJ, HyperJ et ComposeJ. A travers ce chapitre, nous avons essayé de montrer les principaux concepts que doit supporter un langage de description de programmation orientée aspect.

Le chapitre 3 est consacré à l'étude des différentes approches de séparation de préoccupations. Nous avons classé ces approches en deux catégories différentes : la catégorie des approches non orientées aspect et la catégorie d'approches orientées

aspect. Il est alors évident de constater que la majorité des approches orientées aspect est inspirée des approches de la première catégorie.

Une étude comparative est présentée à la fin ce chapitre. Cette étude est basée sur les critères les plus pertinents dans le domaine du développement de logiciel orienté aspect (AOSD).

Ensuite, le chapitre 4 a été consacré à la présentation de notre approche hybride HASoCC. Dans ce chapitre nous avons présenté toute une méthodologie à suivre pour une bonne prise en charge des préoccupations non-fonctionnelles. La découpe fonctionnelle des besoins a montré ses limites face à l'encapsulation des besoins non-fonctionnels.

La prise en charge des préoccupations transversales, qui sont généralement de nature non-fonctionnelle permet une bonne traçablibité des besoins durant les autres phases du cycle de développement. Aussi, l'encapsulation des préoccupations dans des modules séparés réduit considérablement les efforts nécessaires à fournir lors de la réutilisation de leurs propriétés en les adaptant à d'autres contextes ou à d'autres problèmes. De plus, nous avons constaté si nous projetons ces besoins dans une architecture à base de composants que la réutilisation de leurs propriétés est quasi-parfaite et sans effort.

Nous avons, enfin, présenté dans le chapitre 5 une expérimentation de notre approche sur un exemple réel (système de gestion des stations métro). A travers cette expérimentation, nous avons montré comment peut-on spécifier les besoins fonctionnels et les besoins non-fonctionnels à l'aide d'une démarche de localisation des préoccupations transversales.

Une fois les différentes préoccupations raffinées et représentées séparément, vient la phase de composition. Par le biais de la composition, nous pouvons détecter tous les conflits qui peuvent apparaître. Des algorithmes simples sont prévus pour résoudre ces éventuels conflits. Les résultats obtenus de cette expérimentation sont encourageants.

Toutefois, beaucoup d'efforts restent à fournir pour explorer les différents horizons de l'approche proposée. Un travail de validation de cette approche est aussi nécessaire afin de lui faire atteindre la maturation nécessaire à son exploitation réelle dans les futurs processus de développement de logiciels.

Perspectives

Le travail de thèse a permis la mise en place d'un cadre conceptuel pour l'identification et la spécification des préoccupations transversales et de base. Il s'agit d'une approche de modélisation des besoins logiciels, d'une démarche

d'élaboration d'un processus de l'ingénierie des besoins fonctionnels et non-fonctionnels.

Plusieurs voies de recherche peuvent être suivies en prolongement du travail présenté dans cette thèse. En effet, les perspectives que nous dégageons peuvent se décliner en quatre points.

Composition basée sur la sémantique pour l'AORE

Un des points faibles de toutes les approches développées, y compris celle proposée dans cette thèse, réside dans le mécanisme de décomposition. En effet, ce dernier est basé sur des références syntaxiques pour les besoins (e.g. Identificateurs -id- d'un besoin comme dans notre approche, points d'extension dans les cas d'utilisation [Jacobson, 2005]) ou l'utilisation des points joker pour repérer les identificateurs.

Par conséquent, les modèles générés à la fin de cette phase sont des modèles forcement structurels. Ainsi, toutes les informations comportementales exprimées dans les besoins initiaux des intervenants sont perdues dans les phases suivantes d'analyse.

L'une des pistes que nous pouvons envisager pour satisfaire ce point de vue est d'utiliser la richesse sémantique des langages naturels pour exprimer les points de recoupement (*pointcuts*) dans un langage de description des besoins. Notons que le langage naturel lui-même possède un ensemble bien défini de règles syntaxiques suffisamment précises pour supporter la définition du mécanisme de composition pour l'analyse des besoins.

Langage de description des besoins (RDL)

Ce point important concerne la définition d'un profil UML pour la description des préoccupations et des besoins basés sur le travail présenté dans la section 6 du chapitre 3 et ensuite l'intégration de ce profil dans la démarche MDA [Aldawud 2003]. Les profils UML existants sont dédiés à un type d'application (systèmes distribués, temps réel, etc.), alors que le nôtre sera indépendant et spécialisé dans une activité particulière de modélisation des langages de description des besoins. Dans le cadre de ce travail, nous pouvons beaucoup nous inspirer des travaux qui ont été faits au niveau des langages de descriptions d'architectures (ADLs) [Alti 2007].

Projection des concepts

Cette perspective concerne l'extension de notre travail pour la prise en compte de la projection des concepts de l'ingénierie des besoins vers la phase d'architecture logicielle. L'architecture visée est une architecture orientée aspect et à base de composants, de configurations et de connecteurs. Ce travail fait l'objet actuellement d'investigations au sein de notre équipe. Une autre partie de cette étude concerne la projection des éléments architecturaux orientés aspect vers AspectJ. De cette manière, nous pouvons avoir une traçabilité plus au moins complète des concepts orientés aspect depuis la première phase du cycle de vie jusqu'aux langages d'implémentation.

Validation

Enfin, ce dernier point concerne la validation de l'approche proposée et du modèle HASoCC. Certes, un déroulement manuel sur un exemple concret (station métro) nous a apporté des éléments de réponse quant à l'application de notre approche sur des applications réelles. Toutefois, ces expérimentations restent insuffisantes pour valider l'ensemble de nos propositions.

La pratique de notre approche et de notre modèle sur des cas réels nous permettra d'avoir de plus amples retours d'expérience. De ce fait, il serait possible d'étudier de manière pratique l'efficacité de notre approche en implémentant le modèle proposé dans la section 5 du chapitre 4. Cette validation pratique nous renseignera plus en profondeur des dimensions réelles de l'approche proposée.

Liste de nos travaux publiés

I- Revues Internationales (avec comité de lecture)

1. Abdelkrim Amirat "*Towards a Requirements Model for Crosscutting Concerns*" Information Technology Journal (ITJ), Volume 6, Issue 3, pp 332-337, ISSN 1812-5638, ANSI-Journals, Pakistan 2007.

2. Abdelkrim Amirat, Mohamed Tayeb Laskri and Tahar Khammaci, "*Modularization of Crosscutting Concerns in Requirements Engineering*", To appear in International Arab Journal of Information Technology (IAJIT), Jordan, Amman.

II- Conférences Internationales (avec comité de lecture)

1. Abdelkrim Amirat, Adel Smeda and Olivier Le-Goaer, "*Integrating Aspects in the COSA Model*", International Symposium on Programming and Systems", (ISPS'07), May 7-9, 2007 Alger, Algérie, Pages 79-87, ISSN 1112-5853.

2. Abdelkrim Amirat, Mohamed Tayeb Laskri, and Djamel Meslati, "*Identification and Separation of Crosscutting Concerns: A Case Study*" Proceeding of the 8th African Conference on Research in Computer Science (CARI'06), November 6-9, 2006, Cotonou, Benin, Pages 43-50, ISBN 2-7851-1291 9.

3. Abdelkrim Amirat and Mohamed Tayeb Laskri, "*Towards A Unified Model For Crosscutting Requirements*" in Proceedings of the 16th International Conference on Computer Theory and Applications (ICCTA'06), IEEE Computer Society Press Alexandria, Egypt, 5-7 September 2006.

4. Abdelkrim Amirat and Mohamed Tayeb Laskri "*Encapsulating Crosscutting Concerns in Requirements Engineering*", Conférence Internationale sur l'Informatique et ses Applications, (CIIA'06), Centre Universitaire de Saida, Algérie, 15-16 Mai, 2006.

5. Abdelkrim Amirat, Mohamed Tayeb Laskri, and Djamel Meslati, M.T. "*Elicitation of crosscutting aspects at the early phases of software development*", Proceedings of the 2nd IEEE International Conference on Information & Communication Technologies: From Theory to Application (ICTTA'06), ISBN: 0-7803-9521-2, Vol. 2. Pages 1303- 1304, Damascus, Syria, April 24-28, 2006.

6. Abdelkrim Amirat, Djamel Meslati, and Mohamed Tayeb Laskri, "*An Aspect-Oriented Approach in Early Requirements Engineering*", Proceedings of the 4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2006), IEEE Computer Society Press, March 8-11, 2006, Sharjah, UAE, Pages 224-227.

7. Abdelkrim Amirat and Mohamed Tayeb Laskri, "*Modular Implementation of Aspectual Requirements*", Proceedings of the International Arab Conference on Information and Technology (ACIT'05), Amman, Jordan, December 6-8, 2005, Pages 159-163, ISSN 1812-0857.

8. Abdelkrim Amirat "Modular Implementation of Requirements using MDSOC", Actes du Congrès International en Informatique Appliquée, (CIIA-05), Bourdj Bou Arréridj, Algérie, 19-21 Novembre 2005, Pages 247-252.

Bibliographie

[Aksit 1998] M. Aksit and B. Tekinerdogan, "Aspect-oriented programming using composition-filters", In ECOOP Workshops, page 435, 1998.

[Aksit 1997] M. Aksit, "Issues in Aspect-Oriented Programming", in Workshop on Aspect Oriented Programming (ECOOP 1997), 1997.

[Aksit, 1992] M. Aksit, L. Bergmans and S. Vural. "An Object-Oriented Language-Database Integration Model: The Composition Filters Approach", Proceedings of ECOOP '92, LNCS 615, Springer-Verlag, 1992, pp. 372-395.

[Aldawud 2003] O. Aldawud, T. Elrad and A. Bader, "UML Profile for Aspect-Oriented Software Development", In Proceedings of Third International Workshop on Aspect-Oriented Modeling, March 2003.

[Allen 1983] J. F. Allen, "Maintaining Knowledge about Temporal Intervals", Communications of the ACM, vol. 26, pp. 832-843, 1983.

[Alexander 2004] I. Alexander and N. Maiden, "Scenarios, Stories, Use Cases Through the Systems Development Life-Cycle", John Wiley & Sons, 2004.

[Alti 2007] A. Alti, T. Khammaci and A. Smeda, "Representing and formally Modelling COSA Software Architecture with UML 2.0 Profile", the International REview on COmputer and Software (IRECOS), Volume 2, Number 1, January 2007, pp. 30-37. ISSN 1828-6003.

[Ambriola 1997] V. Ambriola and V. Gervasi, "Processing natural language requirements", presented at International Conference on Automated Software Engineering, Los Alamitos, 1997.

[Amirat 2007a] A. Amirat "Towards a Requirements Model for Crosscutting Concerns" Information Technology Journal (ITJ), Volume 6, Issue 3, pp 332-337, ISSN 1812-5638, ANSI-Journals, Pakistan 2007.

[Amirat 2007b] A. Amirat, A. Smeda and O. Le-Goaer, "Integrating Aspects in the COSA Model", International Symposium on Programming and Systems", (ISPS'07), May 7-9, 2007 Alger, Algérie, Pages 79-87, ISSN 1112-5853.

[Amirat 2006a] A. Amirat, M.T. Laskri, and D. Meslati, "Identification and Separation of Crosscutting Concerns: A Case Study" Proceeding of the 8th African Conference on Research in Computer Science (CARI'06), November 6-9, 2006, Cotonou, Benin, Pages 43-50, ISBN 2-7851-1291 9.

[Amirat 2006b] A. Amirat and M.T. Laskri, "Towards A Unified Model For Crosscutting Requirements" in Proceedings of the 16th International Conference on Computer Theory and Applications (ICCTA'06), IEEE Computer Society Press Alexandria, Egypt, 5-7 September 2006.

[Amirat 2006c] A. Amirat, D. Meslati, and M. T. Laskri, "An Aspect-Oriented Approach in Early Requirements Engineering", Proceedings of the 4th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 2006), IEEE Computer Society Press, March 8-11, 2006, Sharjah, UAE, Pages 224-227.

[Amirat 2005a] A. Amirat and M.T. Laskri, "Modular Implementation of Aspectual Requirements", Proceedings of the International Arab Conference on Information and Technology (ACIT'05), Amman, Jordan, December 6-8, 2005, Pages 159-163, ISSN 1812-0857.

[Amirat 2005b] Abdelkrim Amirat "Modular Implementation of Requirements using MDSOC", Actes du Congrès International en Informatique Appliquée, (CIIA-05), Bourdj Bou Arréridj, Algérie, 19-21 Novembre 2005, Pages 247-252.

[Araujo 2002] J. Araujo, A. Moreira, I. Brito, and A. Rashid, "Aspect-Oriented Requirements with UML", presented at Workshop on Aspect-Oriented Modelling with UML, (held in conjunction with the International Conference on Unified Modelling Language UML 2002), 2002.

[Araujo 2003] J. Araujo and A. Moreira, "An Aspectual Use Case Driven Approach", presented at VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD), Alicante, Spain, 2003.

[Baniassad 2004a] E. Baniassad and S. Clarke, "Finding Aspects in Requirements with Theme/Doc", Workshop on Early Aspects (held with AOSD), Lancaster, UK, 2004.

[Baniassad 2004b] E. Baniassad and S. Clarke, "Theme: An Approach for Aspect-Oriented Analysis and Design", Inter. Conference on Software Engineering, 2004.

[Bezivin 2006] J. Bezivin, F., Buettner, M. Gogolla, F. Jouault, I. Kurtev, and A. Lindow "Model Transformations ? Transformation Models", ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS 2006, Italy.

[Blevins 2001] D. Blevins, W.Vanderpereen, and V.Jonckers, "JAsCo: An Aspect-Oriented Approach Tailored for Component Based Software Development," In proc. Second Int. Conf. on AOSD, Boston, MA March 17-21, pp21-29, ACM Press, 2001.

[Booch 1999] G. Booch, J. Rumbaugh, I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1999.

[Brichau 2006] J. Brichau, M. Mezini, J. Noy, W. Havinga, L. Bergmans, V. Gasiunas, and C. Bockisch, "An Initial Metamodel for Aspect-Oriented Programming Languages (D39)", AOSD-Europe Project Deliverable AOSD-Europe-VUB-12, 2006.

[Brinksma 1988] Brinksma E. (ed): Information Processing Systems – Open Systems Interconnection , "LOTOS - A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour", ISO 8807, 1988.

[Brooks 1995] F. P. Brooks, "The Mythical Man-Month: Essays on Software Engineering", Reading: Addison-Wesley, 1995.

[Castro 2001] J. F. Castro, J. Mylopoulos, F.M. R. Alencar, and G.A. Cysneiros Filho, "Integrating organizational requirements and object-oriented modelling", In 5th Int'l Symp. Requirements Engineering (RE), (Toronto). IEEE, 146153, 2001.

[Castro 2002] J. Castro, M. Kolp, and J. Mylopoulos, "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", Information Systems, 27(6), September 2002.

[Chitchyan 2005a] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Pinto, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson, "Survey of (Aspect-Oriented) Analysis and Design Approaches", Lancaster University, Lancaster, Survey Report AOSD-Europe-ULANC-9, 2005.

[Chitchyan 2005b] R. Chitchyan, A. Rashid, and P. Sawyer, "Initial Definition of the Requirements Description Language", Lancaster University, Lancaster, AOSD-Europe Project Milestone M 6.2: AOSD-Europe-ULANC-16, 2005.

[Chitchyan 2006] R. Chitchyan and A. Rashid, "Tracing Requirements Interdependency Semantics", submitted to Workshop on Early Aspects (to be held with ASOD 06), Bonn, Germany 2006.

[Chitchyan 2007] R. Chitchyan, A. Rashid, P. Rayson, and R. W. Waters (2007) "Semantics-based Composition for Aspect-Oriented Requirements Engineering", International Conference on Aspect-Oriented Software Development (AOSD), Vancouver, Canada. ACM. Pages 36-48. 2007.

[Chung 2000] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, "Non-Functional Requirements in Software Engineering", Kluwer Academic Publishers, 2000.

[Clarke 2005] S. Clarke and E. Baniassad, "Aspect-Oriented Analysis and Design: the Theme Approach", Addison-Wesley, 2005.

[CORBA] The CORBA Component Model. http://www.omg.org/docs/formal/02-06-65.pdf.

[Dijkstra 1976] E.W. Dijkstra, "A Discipline of Programming", Prentice-Hall, Englewood Cliffs, N.J. 1976.

[Dixon 2005] R. M. W. Dixon, "A Semantic Approach to English Grammar", 2 ed. Oxford, Oxford University Press, 2005.

[Ducournau 1998] R. Ducournau, J. Euzenat, G. Masini and A. Napoli, "Langages et modèles à objets", chapitre8, volume 19, INRIA, juillet 1998.

[Elrad 2001] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher, "Discussing Aspects of AOP", Communications of the ACM 44(10), pp. 33-38, October, 2001.

[Filman 2004] R. E., Filman, T. Elrad, S. Clarke, and M. Aksit "Aspect-Oriented Software Development", Addison Wesley Professional, ISBN0-321-21976-7, 2004.

[Fuxman 2001] A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso, "Model Checking Early Requirements Specifications in Tropos", Proc. 5th International Symposium on Requirements Engineering (RE'01), Toronto, Canada, August 2001.

[Gamma 1999] E.Gamma, R. Helm, R. Johnson, and J.Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley 1999,

[Garside 1997] R. Garside, G. Leech, and A. McEnery, "Corpus Annotation: Linguistic Information from Computer Text Corpora", London & New York: Longman, 1997.

[Gauthier 1970] R. Gauthier and S. Pont, "Designing Systems Programs", Prentice-Hall, Englewood Cliffs, N.J., 1970.

[Glandrup 1995] M. Glandrup, "Extending C++ using the Concepts of Composition Filters", MSc. thesis, Dept. of Computer Science, University of Twente, 1995.

[Goldin 1997] L. Goldin and D. Berry, "AbstFinder: A Prototype Natural Language Text Abstraction Finder for Use in Requirements Elicitation", Automated Software Engineering, vol. 4, 1997.

[Goguen 2000] J. Goguen and G. Malcolm, "Software Engineering with OBJ: algebraic specification in action", Boston: Kluwer Academic Publishers, 2000.

[Grundy, 1999] J. Grundy, "Aspect-Oriented Requirements Engineering for Component based Software Systems", 4th IEEE International Symposium on RE,1999.

[Grundy, 2000] J. Grundy, "Multi-perspective specification, design and implementation of software components using aspects", International Journal of Software Engineering and Knowledge Engineering, vol. 20, 2000.

[Hursh 1995] W. Hursh and C. Lopes, "Separation of concerns", College of Computer Sciences, Northeastern University, rapport n° NU-CCS-95-03, Février 1995.

[Jackson 2001] M. Jackson, "Problem Frames: Analyzing and Structuring Software Development Problems", Addison-Wesley, 2001.

[Jacobson 1992] I. Jacobson, M. Chirsterson, P. Jonsson, and G. Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", 4 ed: Addison-Wesley, 1992.

[Jacobson, 2003] I., Jacobson, "Use Cases and Aspects–Working Seamlessly Together" JOT Journal Vol. 2, No. 4, July-August 2003.

[Jacobson 2005] I. Jacobson, P.-W. Ng, "Aspect-Oriented Software Development with Use Cases", Addison Wesley, 2005.

[Katz 2004] S. Katz and A. Rashid, "From Aspectual Requirements to Proof Obligations for Aspect-Oriented Systems", presented at International Conference on Requirements Engineering (RE), Kyoto, Japan, 2004.

[Kiczales 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming", in 11th European Conf. Object-Oriented Programming, vol. 1241, *LNCS*, M. A. a. S. Matsuoka, Ed., 1997, pp. 220-242.

[Kiczales 2001] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, et W. Griswold, "An Overview of AspectJ", European Conference on Object-Oriented Programming (ECOOP'01), 2001.

[Krechetov 2006] I. Krechetov, B. Tekinerdogan, M. Pinto, and L. Fuentes, "Initial Version of Aspect-Oriented Architecture Design Approach", University of Twente AOSD-Europe-UT-D37, 2006.

[Lamsweerde 2001] A. Lamsweerde, “Goal-oriented requirements engineering: A guided tour”. In 5th Int'l Symp. Requirements Engineering (RE), (Toronto). IEEE, 249261, 2001.

[Levin 1993] B. Levin, “English verb classes and alternations: a preliminary investigation”, Chicago: The University of Chicago Press, 1993.

[Lieberherr 97] K. J. Lieberherr, “Demeter and Aspect-Oriented Programming”, presented at Smalltalk and Java in Industry and Education: STJA 1997, Erfurt, Germany, 1997.

[Lopes 1995] C. Lopes and W. Hursch, “Separation of Concerns”, technical rapport, College of Computer Science, Northeastern University, Boston, MA, Etats-Unis, Février 1995.

[Loughran 2005] N. Loughran, G. Coulson, L. Seinturier, and F. S. e. al, “Requirements and Definition of AO Middleware Architecture”, Lancaster University, AOSD-Europe Project Deliverable D21: AOSD-Europe-ULANC-15, 2005.

[Masini 1989] G. Masini, A. Napoli, D. Colnet, D. Léonard and K. Tombre, “Les Langages à objets”, InterEdition, Paris, 1989.

[Mason 2000] O. Mason, “Programming for Corpus Linguistics: How to Do Text Analysis with Java”, Edinburgh University Press, 2000.

[McEnery 1996] T. McEnery and A. Wilson, “Corpus Linguistics”, Edinburgh University Press, 1996.

[Medvidovic 2000] N. Medvidovic, and R.N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Language,” IEEE Transactions on Software Engineering Vol. 26, No1, January 2000.

[Meslati 2006] D. Meslati, “MAGE : Une Approche Ontogénétique de l’Evolution dans les Systèmes Logiciels Critiques et Embarquées”, Thèse de doctorat d’état, Université de Annaba, 2006.

[Moreira 2005a] A. Moreira, J. Araujo, and A. Rashid, “Multi-Dimensional Separation of Concerns in Requirements Engineering”, presented at Requirements Engineering Conference (RE 05), Paris, France, 2005.

[Moreira 2005b] A. Moreira, J. Araujo, and A. Rashid, "A Concern-Oriented Requirements Engineering Model”, presented at Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005.

[Mostefaoui 2005] F. Mostefaoui and J.Vachon. «Modélisation et vérification formelle de la composition des aspects», In Actes de la Journée Francophone sur le développement de logiciels par aspects (JFDLPA'05), Lille, France, 2005.

[Mylopoulos 1992] J. Mylopoulos, L. Chung, and B. Nixon, "Representing and Using Non–Functional Requirements: A Process-Oriented Approach". IEEE Transactions on Software, Engineering, Vol. 18, No. 6, June 1992, pp. 483-497.

[Nuseibeh 2001] B. Nuseibeh, "Weaving Together Requirements and Architectures", EEE Computer, vol. 34, pp. 115-117, 2001.

[OMG 2000] OMG, "Meta Object Facility (MOF) Specification", Version 1.3, Object Management Group, Mars 2000. OMG TC Document formal / 2000-04-03.

[OMG 2001a] OMG, "Unified Modeling Language Specification", Object Management group, September 2001, OMG TC Document formal / 2001-09-67.

[OMG 2001b] OMG, "Model Driven Architecture (MDA)", Object Management Group, Juillet 2001. OMG TC Document formal/2001-07-01.

[Ossher 1999] H. Ossher and P. Tarr, "Multi-dimensional Separation of Concerns using Hyperspace", IBM Research Report 21452, IBM T.J. Watson Research Center, April, 1999.

[Ossher 2001] H. Ossher and P. Tarr, "Using multidimensional separation of concerns to (re)shape evolving software", Communications of the ACM, October 2001/Vol. 44, No. 10, pp. 43-50.

[Parnas 1972] D. Parnas, "On the criteria to be used in decomposing systems in modules", Communication on the ACM, 15(12):1053{1058, 1972.

[Pawlak 2000] R. Pawlak, "Nature and Benefits of Aspect-Oriented Programming", ECOOP 2000, workshop on Aspects and Dimensions of Concerns, Sophia-Antipolis, June 2000.

[Pinto 2005] M. Pinto, L. Fluentes, and M. Troya, "A Dynamic Component and Aspect-Oriented Platform", The Computer Journal Vol.48 No. 4, Pp. 401-420, 2005.

[Popovici 2002] A. Popovici, T. Gross, and G. Alonso, "Dynamic weaving for aspect-oriented programming". In proc. First Int. Conf. on AOSD, Enshede, the Netherlands, April 22-26. pp. 141-147. ACM. Press, 2002.

[Rashid 2002] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo, "Early Aspects: a Model for Aspect-Oriented Requirements Engineering", presented at International Conference on Requirements Engineering (RE), Essen, Germany, 2002.

[Rashid 2003] A. Rashid, A. Moreira, and J. Araujo, “Modularisation and Composition of Aspectual Requirements”, presented at 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston, USA, ACM, pp 11-20, 2003.

[Ryan 2000] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe, “The Modelling and Analysis of Security Protocols: The CSP Approach”, Addison-Wesley, 2000.

[Salinas 2001] P. Salinas, “Adding Systemic Crosscutting and Super-Imposition to Composition Filters”, EMOOSE MSc. thesis, Vrije Universiteit Brussel, 2001.

[Sampaio 2005a] A. Sampaio, N. Loughran, A. Rashid, and P. Rayson, "Mining Aspects in Requirements," Aspect-Oriented Requirements Engineering and Architecture Design Workshop (held with AOSD 2005), Chicago, Illinois, USA, 2005.

[Sampaio 2005b] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, “EA-Miner: a Tool for Automating Aspect-Oriented Requirements Identification”, presented at Automated Software Engineering (ASE 2005), Long Beach, California, USA, 2005.

[Sawyer 1996] P. Sawyer, I. Sommerville, and S. Viller, “PREview: tackling the real concerns of requirements engineering”, Cooperative Systems Engineering Group, Computing Department, Lancaster University, Lancaster, Technical Report CSEG/5/96, 1996.

[Sawyer 2002] P. Sawyer, P. Rayson, and R. Garside, “REVERE: Support for Requirements Synthesis from Documents”, Information Systems Frontiers, vol. 4, pp. 343-353, 2002.

[Sawyer 2005] P. Sawyer, P. Rayson, and K. Cosh, “Shallow Knowledge as an Aid to Deep Understanding in Early Phase Requirements Engineering”, Transactions on Software Engineering, vol. 31, pp. 969-981, 2005.

[Schauerhuber 2006] A. Schauerhuber, W. Schwinger, W. Retshitzegger, and M. Wimmer, “A Survay on Aspect-Oriented Modeling Approaches,” Vianna University of Technology, Austria, 2006.

[Shaw 1997] M. Shaw and P. Clements. “A field guide to boxology: Preliminary classification of architectural styles for software systems”. In Proceedings of COMPSAC 1997, August 1997.

[Smeda 2006] A. Smeda, “Contribution à l’élaboration d’une méta-modélisation de description d’architecture logicielle”, Thèse de Doctorat, Université de Nantes, France, 2006.

[Snyder 1986] A. Snyder, "Encapsulation and Inheritance in Object-Oriented Programming Languages", Proc. of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), nov 1986, pp. 38-45.

[Sommerville 1996] I. Sommerville and P. Sawyer, "PREview Viewpoints for Process and Requirements Analysis", Lancaster University, Lancaster REAIMS/WP5.1/LU060, 29 May 1996.

[Sommerville 1997] I. Sommerville and P. Sawyer, "Viewpoints: Principles, Problems and a Practical Approach to Requirements Engineering", Annals of Software Engineering, vol. 3, pp. 101-130, 1997.

[Sommerville 2004] I. Sommerville, "Software Engineering", 7 ed: Addison-Wesley, 2004.

[Spivey 1992] M. Spivey, "The Z Notation", A Reference Manual, Prentice Hall, 1992.

[Sutton 2002] S. M. Sutton and I. Rouvellou, "Modeling of Software Concerns in Cosmos", presented at International Conference on Aspect-Oriented Software Development, 2002.

[Sutton 2003] S. M. Sutton, "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.

[Sutton 2004] S. M. Sutton and I. Rouvellou, "Concern Modeling for Aspect-Oriented Software Development", in Aspect-Oriented Software Development, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds.: Addison-Wesley, 2004, pp. 479-505.

[Tarr 1999] P. Tarr, H. Ossher, and S.M. Sutton, "N Degrees of Separation: Multi-dimensional Separation of Concerns", Dans Proceedings of the International Conference on Software Engineering (ICSE'99), Mai 1999.

[Tripathi 1989] A. Tripathi, E. Berge and M. Aksit, "An Implementation of the Object-Oriented Concurrent Programming Language Sina", Software: Practice and Experience, Vol. 19(3), pp 235-256, March 1989.

[Van-der-Berg 2005] K. van-der-Berg, J. M. Conejero, and R. Chitchyan, "AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation", University of Twente, Twente, AOSD-Europe Project Deliverable D9: AOSD-Europe-UT-01, 2005.

[Whittle 2004] J. Whittle and J. Araujo, "Scenario Modeling with Aspects", IEE Proceedings - Software, vol. 151, pp. 157-172, 2004.

[Wichman 1999] J. C. Wichman, “The Development of a Preprocessor to Facilitate Composition Filters in the Java Language”, MSc. thesis, Dept. of Computer Science, University of Twente, 1999.

[Xerox 2002] Xerox Corporation, “The AspectJ TM Programming Guide”, 1998-2002.

[Yu 1997] E. Yu, "Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering," presented at Requirements Engineering,Washington D.C., USA., 1997.

[Yu 2004] Y. Yu, J. C. S. d. P. Leite, and J. Mylopoulos, “From Goals to Aspects: Discovering Aspects from Requirements Goal Models”, presented at International Conference on Requirements Engineering, Kyoto, Japan, 2004.

Web Site: British National Corpus (BNC), http://www.natcorp.ox.ac.uk/, 2006.

Web Site: WMATRIX, http://www.comp.lancs.ac.uk/ucrel/wmatrix/, Paul Rayson, Lancaster University, URL: http://www.comp.lancs.ac.uk/ucrel/wmatrix/, maintained by P. Rayson, 2005.

Web Site: W3C XML Query (XQuery), http://www.w3.org/XML/Query/, W3C, 2005.

[Wegner 1990] P. Wegner, “Concepts and Paradigms of Object-Oriented Programming”, volume 1, N°1, ACM OOPS Messenger, pp. 7-87, aug 1990.