وزارة التعليم العالي و البحث العلمي

BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA

جامعة باجي مختار– عنابـــــة

Faculté des Sciences de l'Ingéniorat
Année 2014-2015

Département d'Informatique

THESE

Présenté en vue de l'obtention
du diplôme de Doctorat 3ème Cycle en Informatique

Apports des Approches de Séparation Avancée des Préoccupations :

Une Etude Comparative Fondée sur les Modèles de Conception

Option
Ingénierie des Logiciels Complexes

Par
Mme OTMANE RACHEDI
(DEBBOUB) Soumeya

Directeur de Thèse
Mr **Djamel MESLATI**, Professeur, Université Badji Mokhtar-Annaba

DEVANT LE JURY

Président :	Hassina SERIDI	Prof.	Université Badji Mokhtar-Annaba
Examinateurs :	Abdelkrim AMIRAT	Prof.	Université de Souk Ahras
	Nora BOUNOUR	MCA	Université Badji Mokhtar-Annaba

REMERCIEMENTS

Mes plus sincères remerciements vont à :

Mon directeur de thèse, le Professeur ***Djamel MESLATI***, pour tous ses efforts d'encadrement, sa disponibilité, et ses précieux conseils, pertinents comme constructifs, et cela durant l'intégralité de mon parcours universitaire (Licence, Master et Doctorat). Puisse-t-il trouver dans ce témoignage l'expression de ma profonde gratitude et de mon très grand respect.

Les membres du jury, tout d'abord Professeur ***Hassina SERIDI*** qui nous a honoré en acceptant de présider le jury. Mes remerciements vont également au Professeur ***Abdelkrim AMIRAT*** et au Docteur ***Nora BOUNOUR***, pour le temps précieux qu'ils ont accepté d'accorder à mon travail afin de l'examiner et le juger.

L'Equipe de formation ILC, Pour leurs conseils, encouragements et gentillesses tout au long de ces années de recherche.

Ma famille, mes très chers parents, ma source de bonheur et qui vont surement êtres soulagés après ma soutenance !!, mon mari pour son encouragement absolu, et surtout sa patience, ma sœur, mon frère, ma belle famille, pour leur soutien moral durant l'élaboration de cette thèse.

Mes amis, pour leur encouragement et présence, en particulier, Mme ***Boudjedir*** Avec laquelle j'ai passé de bons et de mauvais moments de fatigue et de stresse. C'était malgré tout une superbe expérience !

Je ne manque pas aussi de remercier tous mes collègues docteurs et doctorants pour leurs conseils, soutiens et sympathies.

Merci à vous tous.

A mes très chers parents

RÉSUMÉ

La présente thèse traite un des problèmes importants du génie logiciel qui celui du choix de l'approche de séparation des préoccupations adéquate lors du développement d'un logiciel donné. En effet, l'augmentation de la taille et de la complexité des projets logiciels a favorisé l'apparition de nombreuses approches de séparation des préoccupations, ce qui en soit rend le choix d'une approche donnée difficile. Cette difficulté s'accentue davantage lorsqu'on se rend compte que ces approches sont sophistiquées et apportent de nouveaux concepts et mécanismes qui sont parfois difficiles à maitriser même pour les développeurs expérimentés.

Ce travail de recherche étudie les nouvelles approches de séparations des préoccupations. La démarche que nous avons adoptée consiste à faire une étude comparative entre trois des plus importantes approches qui existent actuellement.

A cet effet, et vu le degré de sophistication des approches de séparation des préoccupations, nous avons eu recours à l'idée originale d'utiliser les patrons de conception comme des benchmarks hypothétiques et d'étudier comment les trois approches implémentent les patrons de conception. Cette idée a été efficace et nous a permis d'aboutir à des résultats prometteurs, ce que nous attendions, car les patrons de conception couvrent la plus part des problèmes liés à la conception des grands logiciels et donc ils constituent une mise à l'épreuve réelle des approches de séparation des préoccupations.

Nous avons réalisé deux sortes de comparaison : quantitative et qualitative. Dans la première comparaison, nous avons utilisé les mesures structurelles et de performance, le résultat de cette dernière concerne les plus importants facteurs d'analyse des logiciels comme le couplage et la cohésion. La deuxième comparaison est qualitative, elle se base sur quelques remarques observées au cours des phases de compréhension ou d'implémentation des patrons de conceptions.

Mots Clés

AspectJ, JBoss AOP, CaesarJ, patrons de conception, métriques logicielles, séparation des préoccupations.

ABSTRACT

This thesis tackles one of the important problems of software engineering that is the choice of an adequate approach for separation of concerns when developing a specific software. Indeed, the increasing size and complexity of software projects has led to the emergence of many approaches to separation of concerns, which in itself makes the choice of a particular approach difficult. This difficulty is further accentuated when one realizes that these approaches are sophisticated and bring new concepts and mechanisms that can be difficult to master even for experienced developers.

This research explores new approaches for the separation of concerns. The approach we have taken is to make a comparative study between three of the most important approaches that currently exist.

To this end, and given the sophistication of approaches to separation of concerns, we resorted to the original idea of using design patterns as hypothetical benchmarks and study how the three approaches implements design patterns. This idea was effective and allowed us to achieve promising results, as we expected, because the design patterns cover most of the problems related to the design of large software and so they are up-to-the real test approaches to separation of concerns.

We conducted two types of comparison: quantitative and qualitative. In the first comparison, we used structural and performance metrics, the result of the latter concerns the most important analytical software factors like coupling and cohesion. The second comparison is qualitative; it is based on some remarks observed during understanding or implementation phases of design patterns.

Keywords

AspectJ, JBoss AOP, CaesarJ, design patterns, software metrics, separation of concerns.

ملخص

تتناول هذه الأطروحة واحد من أهم المشاكل المتعلقة بهندسة البرمجيات، و هو اختيار المنهج الملائم لفصل الانشغالات عند تطوير برامج معينة. في الواقع، أدت زيادة حجم وتعقيد البرمجيات إلى ظهور الكثير من مناهج فصل الانشغالات ، الأمر الذي يجعل اختيار منهج معين في حد ذاته صعبا. تتفاقم هذه الصعوبة عندما ندرك أن هذه المناهج متطورة وجاءت بمفاهيم وآليات جديدة يصعب إتقان استعمالها بالكامل حتى على المطورين ذوي الخبرة.

هذا البحث يستكشف أساليب جديدة لفصل الانشغالات. الطريقة التي اتبعناها في تحليلنا تتمثل في القيام بمقارنة بين ثلاثة من أهم المناهج الموجودة حاليا. تحقيقا لهذه الغاية، ونظرا لتطور مناهج فصل الانشغالات، لجأنا إلى استخدام أنماط التصميم كمعايير افتراضية لدراسة الطريقة المتبعة من قبل المناهج الثلاثة في تنفيذها. كانت هذه الفكرة فعالة وسمحت لنا بتحقيق نتائج واعدة، كما توقعنا، لأن أنماط التصميم تغطي معظم المشاكل المتعلقة بتصميم البرمجيات الكبيرة ما يسمح بالقيام بدراسة واقعية واختبار حقيقي لمناهج فصل الانشغالات.

في هذا التحليل أجرينا نوعين من المقارنة: مقارنة كمية و مقارنة نوعية. في المقارنة الأولى، استخدمنا المقاييس الهيكلية ومقاييس الأداء، نتائج هذه الأخيرة، تتعلق بأهم عوامل الدراسة التحليلية لبرامج البرمجيات كالاقتران والتماسك. المقارنة الثانية هي نوعية و تقوم على بعض الملاحظات التي ُأخذت خلال فهم أو تنفيذ أنماط التصميم.

كلمات مرشدة

اسبكت جي، جيبوس، كايجزر جي، أنماط التصاميم، قياس البرمجيات، فصل الانشغالات.

TABLE DES MATIERES

INTRODUCTION GENERALE 1

1 CONTEXTE DU TRAVAIL ET OBJECTIF 3
2 PROBLEMATIQUE 3
3 MOTIVATIONS 4
4 METHODOLOGIE DE RECHERCHE 4
5 ORGANISATION DE LA THESE 6

CHAPITRE 1 : LA SÉPARATION AVANCÉE DES PRÉOCCUPATIONS 9

1 LES LIMITES DE LA PROGRAMMATION OBJET 10
1.1 Les préoccupations transversales 10
1.1.1 L'enchevêtrement du code 11
1.1.2 L'éparpillement du code 11
1.2 Conséquences 12
2 LES APPROCHES DE SEPARATION AVANCEE DE PREOCCUPATIONS 12
2.1 La programmation orientée aspect 13
2.2 Les filtres de composition 14
2.3 La séparation multidimensionnelle des préoccupations 15
2.4 La programmation par rôles ou points de vue 16
2.5 La programmation adaptative 16
3 LES PRINCIPES DE BASE DE LA PROGRAMMATION ORIENTEE ASPECT 17
3.1 Le concept d'Aspect 17
3.2 Le concept de point de jonction 17
3.3 Le concept de coupe 18
3.4 Le concept de consigne (Advice) 18
3.4.1 Les consignes before 18
3.4.2 Les consignes after 18
3.4.3 Les consignes around 19
3.5 Le concept d'introduction 19
3.6 Le concept de tissage 19
3.6.1 Les règles de tissage 19
3.6.2 Le tisseur 20
3.6.3 Le tisseur statique (Tissage à la compilation) 20
3.6.4 Le tisseur dynamique (Tissage à l'exécution) 21
4 TRAVAUX SUR L'APPROCHE ORIENTEE ASPECT 21
5 CONCLUSION 22

CHAPITRE 2 : ASPECTJ & JBOSS AOP & CAESARJ 23

1 ASPECTJ 24
1.1 Le concept d'aspect 24
1.2 Le concept de point de jonction 24
1.3 Le concept de coupe 25
1.4 Le concept de consigne (Advice) 26
1.5 Le concept d'introduction 26
1.6 Le concept de tissage 27
2 JBOSS AOP 27
2.1 Le concept d'aspect 27
2.2 Le concept de point de jonction 28
2.3 Le concept de coupe 28
2.4 Le concept de consigne (Advice) 29
2.4.1 Les intercepteurs 29
2.5 Le concept d'introduction 30
2.6 Le concept de tissage 31

3 CAESARJ 31
3.1 Le concept des classes virtuelles 32
3.1.1 Le concept d'héritage implicite 32
3.1.2 Le concept de polymorphisme des familles 33
3.2 Le composant CaesarJ 33
3.2.1 L'interface de collaboration 34
3.2.2 CaesarJ implémentation 34
3.2.3 Binding et wrappers 35
3.2.4 Weavelets 35
3.2.5 Déploiement d'aspect 35
3.3 Le concept de tissage 36
4 BILAN 37
5 CONCLUSION 38

CHAPITRE 3 : RÉALISATION DES PATRONS DE CONCEPTION AVEC L'APPROCHE ORIENTÉE ASPECT 39

1 LES PATRONS DE CONCEPTION DU GANG OF FOUR (GOF) 40
1.1. Présentation des patrons de conception du GoF 40
1.1.1 Origine et histoire 40
1.1.2 Qu'est-ce qu'un patron de conception 41
1.1.3 Les patrons de conception du Gang of four 42
1.2 Application et limites des patrons de conception par objets 45
2 APPROCHE ADOPTEE DANS L'IMPLEMENTATION ORIENTEE ASPECT DES PATRONS 47
3 UTILISATION DES ASPECTS DANS L'IMPLEMENTATION DES PATRONS DE CONCEPTION DU GOF 48
3.1 La notion de rôle 49
3.2 Exemples de réalisation des patrons de conception avec les approches Aspect 49
3.2.1 Le patron de conception Observateur 50
3.2.2 Le patron de conception Mémento 54
3.2.3 Le patron de conception Adaptateur 56
3.2.4 Le patron de conception Fabrique abstraite 59
3.2.5 Le patron de conception Etat 61
3.2.6 Le patron de conception Façade 63
3.3 Application aux autres patrons du GoF 63
3.3.1 Groupe 1(Observateur / Médiateur /Chaine de responsabilités / Composite / Commande) 63
3.3.2 Groupe 2 (Singleton/ Prototype/ Mémento/ Itérateur /Poids-mouche) 64
3.3.3 Groupe 3 (Adaptateur/ Décorateur/ Stratégie/ Visiteur / Proxy) 65
3.3.4 Groupe 4 (Fabrique abstrait/ Fabrique méthode/ Méthode patron/ Constructeur/ Pont) 65
3.3.5 Groupe 5 (Etat / Interpréteur) 65
3.3.6 Groupe 6 (Façade) 65
4 CONCLUSION 65

CHAPITRE 4 : ETUDE COMPARATIVE 67

1 ETUDE DES METRIQUES 68
1.1 Questions sur la mesure des logiciels 68
1.1.1 Pourquoi Mesurer ? 68
1.1.2 Qu'est-ce qu'on peut mesurer dans un logiciel? 68
1.2 Métriques de l'orienté Objet 69
1.2.1 Métriques Structurelles 69
1.2.2 Incidence de l'aspect sur les métriques de l'orienté objet 72
1.3 Métriques de l'Approche Orientée Aspect 74
1.4 Métriques de performance 75
1.5 La qualité des logiciels 76
1.5.1 Les attributs internes de mesure 76
1.5.2 Model de qualité 77
2 ÉTUDE COMPARATIVE 79
2.1 Comparaison quantitative 80
2.1.1 Méthode de travail 80
2.1.2 Résultats obtenus 80
2.2 Interprétation des résultats 89
2.3 Comparaison qualitative 91

2.3.1 Méthode de travail 91
2.3.2 Résultats obtenus 92
2.4 Interprétations des résultats 94
2.4.1 Interprétation des résultats de l'analyse des implémentations 94
2.4.2 Interprétation des résultats des réponses au questionnaire 95
3 SYNTHESE GENERALE 97
4 VERS UN OUTIL GENERIQUE D'EXTRACTION DES METRIQUES 99
4.1 Pourquoi utiliser XML ? 100
4.2 Aspect conceptuel de l'approche 100
4.2.1 Construction des fichiers XML (du code source vers XML) 100
4.2.2 Tissage des fichiers XML 104
4.3 Extraction des métriques 105
5 CONCLUSION 105
CONCLUSION ET PERSPECTIVES … 107
RÉFÉRENCES 111
ANNEXES 117
ANNEXE A 119
ANNEXE B 125

LISTE DES FIGURES

INTRODUCTION GENERALE 1

FIGURE 1 ORGANISATION DU DOCUMENT 7

CHAPITRE 1 : LA SÉPARATION AVANCÉE DES PRÉOCCUPATIONS 9

FIGURE 2 PREOCCUPATIONS TRANSVERSALES 11
FIGURE 3 ENCHEVETREMENT DU CODE 11
FIGURE 4 DISPERSION DU CODE DE LA PREOCCUPATION LOGGING 12
FIGURE 5 INTEGRATION DU CODE DES COMPOSANTS ET DES ASPECTS POUR FORMER LE SYSTEME FINAL 14
FIGURE 6 SCHEMA INTUITIF ILLUSTRANT LE FILTRAGE DES MESSAGES [AMIRAT, 2007] 15
FIGURE 7 EXEMPLE DE REGLE DE COMPOSITION DANS HYPER/J 16
FIGURE 8 TISSEUR POA 20

CHAPITRE 2 : ASPECTJ & JBOSS AOP & CAESARJ 23

FIGURE 9 HERITAGE IMPLICITE DANS CAESARJ 33
FIGURE 10 STRUCTURE GENERALE DU COMPOSANT CAESARJ 36
FIGURE 11 PROCEDURE GENERALE DE TRAVAIL DES TROIS APPROCHES [DEBBOUB, 2013] 38

CHAPITRE 3 : RÉALISATION DES PATRONS DE CONCEPTION AVEC L'APPROCHE ORIENTÉE ASPECT 39

FIGURE 12 SCHEMA GLOBAL DE LA METHODE ADOPTEE 48
FIGURE 13 SOLUTION ASPECTJ DU PATRON DE CONCEPTION OBSERVATEUR 52
FIGURE 14 SOLUTION JBOSS AOP DU PATRON DE CONCEPTION OBSERVATEUR 53
FIGURE 15 SOLUTION CAESARJ DU PATRON DE CONCEPTION OBSERVATEUR 54
FIGURE 16 SOLUTION ASPECTJ DU PATRON DE CONCEPTION MEMENTO 55
FIGURE 17 SOLUTION JBOSS AOP DU PATRON DE CONCEPTION MEMENTO 56
FIGURE 18 SOLUTION CAESARJ DU PATRON DE CONCEPTION MEMENTO 57
FIGURE 19 SOLUTION ASPECTJ DU PATRON DE CONCEPTION ADAPTATEUR 57
FIGURE 20 SOLUTION JBOSS AOP DU PATRON DE CONCEPTION ADAPTATEUR 58
FIGURE 21 SOLUTION CAESARJ DU PATRON DE CONCEPTION ADAPTATEUR 59
FIGURE 22 SOLUTION ASPECTJ DU PATRON DE CONCEPTION FABRIQUE ABSTRAITE 60
FIGURE 23 SOLUTION JBOSS AOP DU PATRON DE CONCEPTION FABRIQUE ABSTRAITE 60
FIGURE 24SOLUTION CAESARJ DU PATRON DE CONCEPTION FABRIQUE ABSTRAITE 61
FIGURE 25 .SOLUTION ASPECTJ DU PATRON DE CONCEPTION ETAT 62
FIGURE 26 . SOLUTION JBOSS AOP DU PATRON DE CONCEPTION ETAT 62
FIGURE 27 SOLUTION CAESARJ DU PATRON DE CONCEPTION ETAT 63

CHAPITRE 4 : ETUDE COMPARATIVE 67

FIGURE 28 METRIQUE PROPOSEE NCI (NUMBER OF COMPONENTS INTRODUCED). 75
FIGURE 29 MODELE DE QUALITE [SANT'ANNA, 2003] 79
FIGURE 30 COMPARAISON QUANTITATIVE 80
FIGURE 31 RESULTATS DE COUPLAGE DU PATRON DE CONCEPTION OBSERVATEUR (AVEC CAESARJ)[DEBBOUB, 2014] 83
FIGURE 32 RESULTATS DES METRIQUES DE TAILLE 84
FIGURE 33 RESULTATS DES METRIQUES DE COUPLAGE 85
FIGURE 34 RESULTATS DE LA METRIQUE DE COHESION 86
FIGURE 35 RESULTATS DES METRIQUES SPECIFIQUES A LA PROGRAMMATION ORIENTEE ASPECT. 87
FIGURE 36 LES DIFFERENTES STRATEGIES EMPLOYEES PAR LES TROIS APPROCHES 93
FIGURE 37 L'ARCHITECTURE GENERALE DE L'APPROCHE 100
FIGURE 38 L'INFLUENCE D'ELEMENTS ASPECT SUR LES ATTRIBUTS DE MESURE 103
FIGURE 39 TISSAGE DES FICHIERS XML 105
FIGURE 40 REQUETE SUR LA METRIQUE NOA 105

LISTE DES TABLEAUX

CHAPITRE 2 : ASPECTJ & JBOSS AOP & CAESARJ

TABLEAU 1 SIGNIFICATION PREDICATIVE DES POINTS DE JOINTURE PRIMITIFS ET LEUR COMPOSITION DANS ASPECTJ 25
TABLEAU 2: LISTE DES POINTS DE JONCTIONS DISPONIBLES DANS JBOSS AOP 28
TABLEAU 3 UTILISATION DES CONCEPTS ASPECT PAR LES TROIS APPROCHES 37

CHAPITRE 3 : RÉALISATION DES PATRONS DE CONCEPTION AVEC L'APPROCHE ORIENTÉE ASPECT

TABLEAU 4LES PATRONS DE CONCEPTION DU GOF 45
TABLEAU 5EXEMPLES ILLUSTRATIFS 50

CHAPITRE 4 : ETUDE COMPARATIVE

TABLEAU 6 LES METRIQUES DE CHIDAMBER ET KEMERER RETENUES DANS NOTRE ETUDE. 69
TABLEAU 7 LES METRIQUES DE LORENZ ET KIDD RETENUES DANS NOTRE ETUDE 72
TABLEAU 8 ASSOCIATION ENTRE LES METRIQUES ET LES ATTRIBUTS DE MESURES 77
TABLEAU 9 RESULTATS DES METRIQUES ETENDUES DE LA PROGRAMMATION OBJET 81
TABLEAU 10 RESULTATS DE LA METRIQUE NCI DANS LE PATRON DE CONCEPTION OBSERVATEUR 88
TABLEAU 11 RESULTATS DES METRIQUES SPECIFIQUES A LA PROGRAMMATION ORIENTEE ASPECT 88
TABLEAU 12RESULTATS DE PERFORMANCE 88
TABLEAU 13 RESULTATS DE LA COMPARAISON QUALITATIVE 93
TABLEAU 14 SYNTHESE GENERALE. 98

INTRODUCTION GENERALE

1 Contexte du travail et objectif

L'apparition de la programmation orientée objet (POO) avait, à ses débuts, bouleversé le monde du génie logiciel. Apportant son lot de concepts et de mécanismes et introduisant des notions clés telles que les classes, l'encapsulation, l'héritage. De plus son efficacité à cette époque, début des années 1980, était relativement avérée.

Par la suite, on remarqua en utilisant cette approche la récurrence de certaines situations, qui donnaient lieu à des solutions de conception quasiment similaires. C'est alors, que le Gang of Four (GoF), un groupe de quatre chercheurs et spécialistes en génie logiciel, mit au point le premier catalogue de patrons de conception (ou modèles de conception), regroupant vingt-trois patrons décrivant les cas les plus connus et les plus fréquemment rencontrés. Ces patrons de conceptions représentaient des solutions génériques, susceptibles de faciliter le processus de conception et de programmation orientée objet (POO), et d'en augmenter le rendement, grâce à une meilleure compréhension des architectures logicielles.

Cependant la taille et la complexité, sans cesse grandissantes, des projets logiciels ont fait apparaitre les limites de l'approche orientée objet et de la conception basée sur les patrons, où seules les préoccupations métiers (ou fonctionnalités) étaient bien isolées. Il y avait là une faille dans la modularité, ce qui laissait présager de nouvelles innovations dans ce sens.

Ce problème de la POO, et plus généralement les problèmes de la séparation des préoccupations trouvèrent une solution puissante avec le paradigme orienté aspect. Plus précisément, la programmation orientée aspect (POA) propose de décomposer les applications non seulement en classes mais aussi en aspects, ce qui offre une implémentation plus propre des préoccupations à caractère transversale telle que la sécurité, la gestion des transactions, etc. Ce paradigme ne remet guère en cause les fondements de la POO, loin de là, il tend plutôt à la servir comme extension, l'enrichissant de concepts complémentaires, pour une meilleure modularité. Ainsi, pour supporter ces nouveaux concepts, plusieurs extensions de langages existants sont proposées comme C++, Java et Eiffel.

Par la suite l'approche orientée aspects s'est généralisée à toute sorte de préoccupations et s'est retrouvée dans un contexte général nommé « Séparation avancée des préoccupations ». La communauté du génie logiciel, a tout de suite saisie l'intérêt d'aller vers une séparation plus poussée des préoccupations et diverses approches ont vu le jour.

Dès lors, bien que le développeur comprenne l'intérêt de ces approches, il se retrouve devant une problématique importante : Quelle approche choisir. C'est dans ce contexte que se situe le travail de cette thèse. Son objectif est de comparer des approches orientées aspects pour en déduire leurs faiblesses et leurs avantages.

2 Problématique

Face à diverses approches de séparation avancée des préoccupations, il est légitime de se poser de nombreuses questions : Quelle approche est plus avantageuse dans tel ou tel contexte ? Existe-il, ou pourrait-il exister une approche convenable dans n'importe quelles circonstances ? Que faut-il faire pour améliorer telle ou telle approche ? Quels sont les concepts clés de la séparation avancée des préoccupations ?etc.

Quel que soit la manière de procéder pour répondre à ces questions, il demeure un point commun qui est celui de la nécessiter de comparer les approches existantes.

Malheureusement, la comparaison elle-même n'est pas triviale. S'il est aisé de comparer, par exemple, deux langages orientés objets, les approches de séparation avancée des préoccupations sont beaucoup plus sophistiquées et ne s'y prêtent pas facilement à la comparaison. Ressortir leurs points forts et leurs points faibles ne peut se faire par une simple comparaison de langages et des concepts qu'ils renferment.

La comparaison des approches de séparation avancée des préoccupations et la manière de mener cette comparaison constitue un défi considérable et une problématique d'une portée importante. Vu l'ampleur du travail, nous avons été obligés de restreindre la problématique de notre thèse à un sous ensemble d'approches et nous avons utilisé une démarche particulière pour mettre à l'épreuve ces approches. On particulier, nous avons opté pour la comparaison de trois approches de la catégorie orientée aspects et nous avons utilisé les patrons de conception du GoF comme des benchmarks de comparaison.

3 Motivations

Comme pour toute nouveauté, la séparation avancée des préoccupations a soulevé, bien sûr, de sérieuses questions quant à la pertinence de ses apports, et sur la manière d'en bénéficier, ainsi que sur sa fiabilité.

Plusieurs études comparatives ont été menées pour mettre en évidence les avantages et les apports du paradigme aspect par rapport à ces prédécesseurs [Hannemann, 2002] [Garcia, 2006] [Hachani, 2005].

Cependant, seuls quelques travaux comparent les approches orientées aspects entre elles. Une telle comparaison est motivée par deux points importants :

- Aider les développeurs à choisir l'approche adéquate pour leurs projets.
- Mettre en évidence les avantages et les inconvénients des différentes approches de telle sorte à contribuer à leur amélioration et, éventuellement, opérer une synergie des différentes approches et parvenir à une approche unifiée qui soit, à l'image d'UML, une approche utilisable dans différents domaines et offrant une multitudes d'avantages tout en réduisant les inconvénients.

4 Méthodologie de recherche

Ayant cerné l'objectif et les motivations (le Quoi et le Pourquoi), il nous reste à déterminer la méthodologie de travail (le Comment). Dans un premier temps, vu la diversité des approches de séparation avancée des préoccupations, nous avons limité notre comparaison à trois approches. Ces trois approches, AspectJ, CaesarJ et JBoss AOP, appartiennent à la famille dite orientée aspects. De plus ces trois approches sont implémentées comme des extensions du langage orienté objet Java. Ce choix délibéré nous permet de se concentrer sur les apports spécifiquement orientés aspects de ces trois approches. Par ailleurs, comme signalé précédemment, les approches orientées aspects ont des concepts sophistiqués qu'il n'est pas facile de comparer directement, nous avons alors opté pour une comparaison à base de patrons de conception.

L'utilisation des patrons de conception comme des benchmarks hypothétiques dans cette étude est une idée pratique et efficace. En effet:

- Les patrons de conception couvrent la plus part des problèmes liés à la conception des grands logiciels. Autrement dit, si une approche implémente efficacement un patron elle répond efficacement au problème de conception que ce patron couvre,
- Les patrons de conception forment des structures complexes de haut niveau dont la mise en œuvre n'est pas triviale. L'utilisation des langages orientés aspects dans la résolution de ces problèmes permet d'extraire des différences et montrer la puissance de chaque langage,
- Plusieurs études comparatives ont utilisé les patrons de conception pour montrer les avantages d'utilisation de la programmation orientée aspect dans leurs implémentations [Hannemann, 2002] [Garcia, 2006] [Hachani, 2005],
- Actuellement, une bonne partie des développeurs sont familiarisés avec les patrons de conception. De ce fait, comprendre les nouveaux concepts des approches orientées aspects à travers la compréhension des solutions orientées aspects de ces patrons émane d'une démarche pratique,
- L'architecture de la plupart des programmes existants est basée sur les patrons de conception, alors fournir les nouvelles solutions aspects avec différentes approches aide à maintenir les anciennes architectures, et basculer vers le monde orienté aspect plus facilement.

Le fait que les trois approches : AspectJ, JBoss AOP et CaesarJ soient basées sur le langage Java et appliquent les concepts de la programmation orientée aspect, nous permet de se concentrer sur les apports des approches spécifiques à l'orienté aspect. En effet, avoir un seul langage de base, nous permet de mettre l'accent sur l'extension de ce langage (c'est à dire ce qui est ajouté pour couvrir les nouveaux concepts de la programmation orientée aspect), plutôt que sur les différences qui viennent du langage lui-même.

AspectJ été le premier langage proposé, il étend le langage Java avec des nouveaux concepts comme les aspects, les consignes et les déclarations inter-type. Ce nouveau langage a prouvé sa vigueur dans plusieurs études expérimentales.

Similairement à AspectJ, JBoss AOP travaille presque avec les mêmes principes, d'ailleurs les deux approches sont connues sous le nom « AspectJ-like languages », cependant JBoss AOP soutient l'idée de ne pas s'éloigner du langage Java, et préfère utiliser des instructions en Java pur pour décrire les préoccupations.

Contrairement, aux deux approches précédentes, CaesarJ supporte de nouveaux concepts qui lui sont propres, comme les classes virtuelles, les bindings et la composition par mix-in.

La proximité relative de ces trois approches nous permet d'envisager deux types de comparaisons. La première est quantitative, elle est basée sur des métriques structurelles et de performance, nous avons pris en compte l'influence de la programmation orientée aspect sur ces métriques, son objectif est :

- d'étudier des principaux facteurs de qualité logicielle comme le couplage et la cohésion.
- de valider quelques remarques qualitatives observées précédemment.

La deuxième est qualitative, et se base sur quelques remarques observées au cours des phases de compréhension et d'implémentation ayant eu lieu durant notre étude.

5 Organisation de la thèse

Ce document est organisé selon le schéma de description de la figure 1 où on distingue trois parties :

- **Partie 1 (Etat de l'art).** Elle contient deux chapitres. Le premier présente un état de l'art sur les différents facteurs qui ont poussé à l'apparition des approches de séparation avancée des préoccupations, décrit ces nouvelles approches, et énumère quelques travaux menés sur l'approche orientée aspect.
  Dans le deuxième chapitre, l'accent est mis sur les trois approches étudiées dans cette thèse que sont : AspectJ, JBoss AOP et CaesarJ.

- **Partie 2 (Implémentation)**. Le chapitre constituant cette partie décrit les patrons de conception et leur importance dans le génie logiciel. Ensuite, il présente la première contribution de notre travail qui concerne l'implémentation des patrons de conception avec les concepts aspects en montrant les solutions proposées.

- **Partie 3 (Synthèse)**. Dans cette partie, le chapitre 4 expose notre étude comparative. Il commence par décrire les métriques utilisées dans notre travail, par la suite il présente et interprète les résultats obtenus dans nos deux comparaisons : quantitative et qualitative.
  La fin de cette partie pose les prémisses d'un travail futur qui concerne la réalisation d'un outil générique destiné à l'extraction des métriques logicielles.

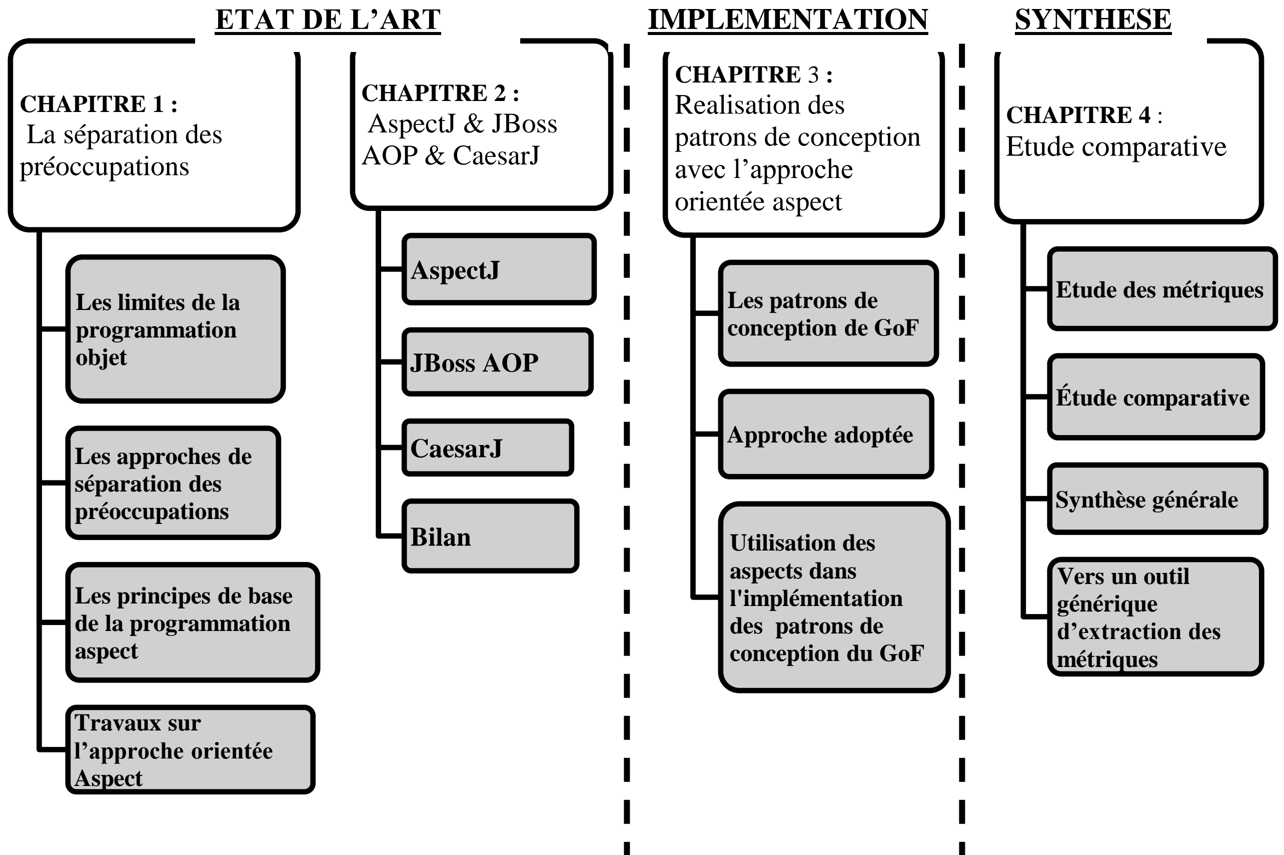

Figure 1 Organisation du document

CHAPITRE 1 : LA SEPARATION AVANCEE DES PREOCCUPATIONS

1 LES LIMITES DE LA PROGRAMMATION OBJET
1.1 LES PREOCCUPATIONS TRANSVERSALES
1.2 CONSEQUENCES
2 LES APPROCHES DE SÉPARATION AVANCÉE DE PRÉOCCUPATIONS
2.1 LA PROGRAMMATION ORIENTEE ASPECT
2.2 LES FILTRES DE COMPOSITION
2.3 LA SEPARATION MULTIDIMENSIONNELLE DES PREOCCUPATIONS
2.4 LA PROGRAMMATION PAR ROLES OU POINTS DE VUE
2.5 LA PROGRAMMATION ADAPTATIVE
3 LES PRINCIPES DE BASE DE LA PROGRAMMATION ORIENTÉE ASPECT
3.1 LE CONCEPT D'ASPECT
3.2 LE CONCEPT DE POINT DE JONCTION
3.3 LE CONCEPT DE COUPE
3.4 LE CONCEPT DE CONSIGNE (ADVICE)
3.5 LE CONCEPT D'INTRODUCTION
3.6 LE CONCEPT DE TISSAGE
4 TRAVAUX SUR L'APPROCHE ORIENTÉE ASPECT
5 CONCLUSION

***Résumé* :**

Armée des concepts de classes et d’objets, la programmation Orientée Object (POO) a indéniablement fait avancer l’ingénierie des applications informatiques et des programmes plus complexes ont pu être conçus de façon plus simple qu’avec la programmation procédurale. Cependant, la POO souffre de quelques problèmes d'organisation, tels que l’enchevêtrement et l’éparpillement du code.

Alors que l’on cherche des palliatifs à la POO pour résoudre ces problèmes, L’approche orientée aspect est venue proposer une nouvelle démarche prometteuse de séparation des préoccupations.

Ce chapitre retrace le cheminement ayant conduit au paradigme orienté aspect, décrit les approches les plus importantes de séparation des préoccupations avant de présenter les concepts de base de la programmation orientée aspect.

1 Les limites de la programmation objet

La programmation orientée objet vise à structurer les objets de manière intelligente, ses principaux buts sont de rendre les applications plus modulables (i.e. concepts de classes et d’objets), mieux réutilisables et davantage extensibles (i.e. concept d'héritage). Néanmoins, nous allons voir que, les solutions fournies par la POO ne sont pas toujours claires et élégantes, et ne sont donc pas très satisfaisantes, surtout en ce qui concerne les problèmes engendrés par les préoccupations transversales.

1.1 Les préoccupations transversales

Idéalement, un projet logiciel propose une isolation quasi totale de ses différentes préoccupations, de telle sorte que chacune d'entre elles puisse évoluer indépendamment des autres. De façon générale, ces différentes préoccupations sont liées à la technique (i.e. configuration, accès aux données, ...), au métier et à la présentation.

Cependant, une application dispose de préoccupations dites transversales (Crosscutting Concerns), c'est-à-dire qu'elles sont présentes dans ses différentes couches et fonctionnalités métier. Classiquement, ces préoccupations transversales sont la gestion de la sécurité, des transactions, des traces applicatives et de la cohérence des traitements par rapport aux spécifications métier. Ainsi, chaque couche de l'application ou fonctionnalité métier sera éventuellement concernée par ces préoccupations.

Le problème est donc que chaque évolution dans ces préoccupations transversales devra être répercutée dans chacun des composants y faisant appel. Si ce n'est pas le cas, il est très probable que des effets de bord apparaissent.

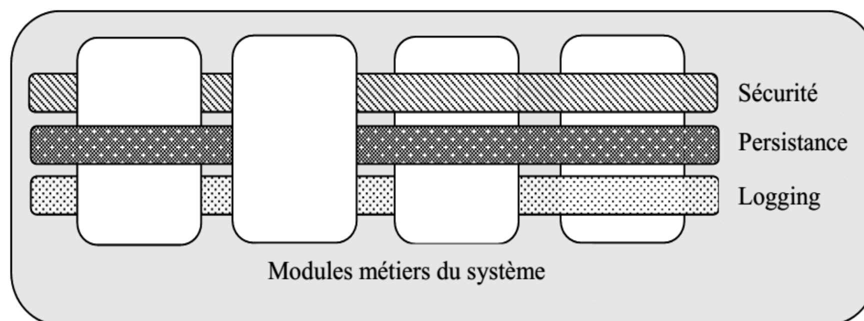

Figure 2 Préoccupations transversales

Il existe en fait deux principaux symptômes liés aux préoccupations transversales : l'enchevêtrement du code et l'éparpillement du code

1.1.1 L'enchevêtrement du code

L'enchevêtrement du code est provoqué quand un module est implémenté pour traiter plusieurs préoccupations en même temps. Un développeur a souvent affaire, pendant qu'il développe un module, à des préoccupations telles que la gestion transactionnelle de la persistance, le logging, la sécurité, etc. (Voir figure 3). Cela conduit à la présence simultanée d'éléments issus de chaque préoccupation et il en résulte un enchevêtrement du code.

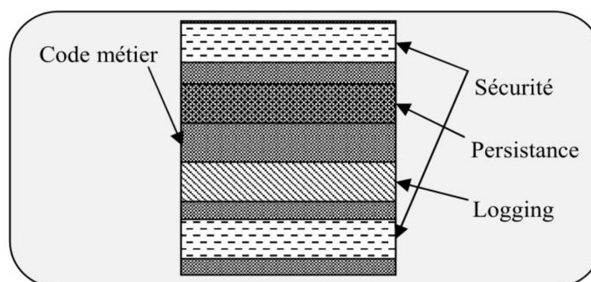

Figure 3 Enchevêtrement du code

1.1.2 L'éparpillement du code

La dispersion de code est une autre limite de la programmation orientée objet. Considérons un langage orienté objet, l'unité fonctionnelle est le package et si nous raffinons encore la découpe, l'unité devient la classe. Il n'est pas rare de voir des méthodes traitant d'une exigence non-fonctionnelle se disperser dans l'implémentation des différentes classes composant le cœur fonctionnel du système.

Par exemple, dans un système de gestion de base de données il se peut que la performance, la journalisation et la synchronisation concernent toutes les classes accédant à la base de données. On voit donc que ces aspects seront implémentés dans plusieurs modules sans être bien circonscrits. Il s'agit très souvent de portions de code très similaires à ajouter un peu partout dans les modules concernés, ce qui pose des problèmes en termes d'évolutivité et de maintenance du code.

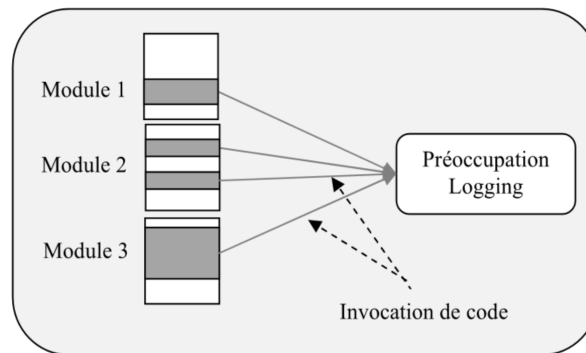

Figure 4 Dispersion du code de la préoccupation logging

1.2 Conséquences

Les deux problèmes cités auparavant, entraînent des conséquences négatives sur le développement d'un logiciel :

- **Traçage difficile.** Les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les exigences et leurs implémentations.
- **Diminution de la productivité.** La prise en considération de plusieurs exigences au sein d'un même module empêche le programmeur de se focaliser uniquement sur son but premier. Le danger d'accorder trop ou pas assez d'importance aux aspects accessoires d'un module en découle directement.
- **Diminution de la réutilisation du code.** Dans les conditions actuelles, un module implémente de multiples exigences. D'autres systèmes nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser le module tel quel, entraînant de nouveau une diminution de la productivité à moyen terme.
- **Diminution de la qualité du code.** Les programmeurs ne peuvent pas se concentrer sur toutes les contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entraîner des effets de bords non-désirés, (i.e. des bugs).
- **Maintenance et évolutivité du code difficile.** Lorsqu'on veut faire évoluer le système, on doit modifier de nombreux modules. Modifier chaque sous-système pour répercuter les modifications souhaitées peut conduire à des incohérences [Pawlak, 2004].

2 Les approches de séparation avancée de préoccupations

Comme nous avons montré dans la section précédente, le principal problème des approches classiques revient à la mauvaise modularisation des préoccupations transversales, pour cette raison plusieurs nouvelles approches sont proposées pour garantir une meilleure séparation et composition de tout type de préoccupations, ces approches sont connues sous la désignation plus générale d'approches de séparation des préoccupations.

Les principales approches pour la séparation des préoccupations sont :

- La programmation orienté aspect,
- Les filtres de composition,
- La séparation multidimensionnelle des préoccupations,
- Programmation par rôles ou points de vue,
- Programmation adaptative.

2.1 La programmation orientée aspect

Parmi les approches les plus connues de séparation des préoccupations, on trouve la programmation par aspects [Kiczales, 1997], Les fondements de ce paradigme ont été définis au centre de recherche (PARC) de Xerox, à Palo Alto au milieu des années 90 [Hachani, 2006].

La programmation par aspect, explore des pistes alternatives visant à corriger les défauts d'adaptation et de modularité des programmes à objets [Denier, 2007]. Elle s'intéresse au problème particulier de la modularité des préoccupations transversales. Suivant cette problématique, la décomposition d'un système informatique en préoccupations ne cadre pas toujours avec un système hiérarchique de classes : certaines préoccupations sont « forcées » dans les classes existantes. En termes de symptômes, les implémentations de ces préoccupations sont dispersées et mélangées avec les autres préoccupations. La programmation par aspects est une technique visant une bonne modularisation de ces préoccupations, en complément des mécanismes préexistant dans les langages classiques [Filman, 2005].

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 [Amirat, 2007].

Et pour mieux comprendre ce principe, nous pouvons reprendre la description originale faite par Gregor Kiczales et ses collègues [Kiczales, 1997]. Une préoccupation qui doit être implémentée est soit:

- **Un composant**. Si elle peut être clairement encapsulée dans un objet, ou un module. Les composants sont donc, par définition, des unités fonctionnelles d'un système. Par exemple, les services d'impression ou d'accès aux bases de données d'une librairie digitale.
- **Un aspect**. Si elle ne peut pas être clairement encapsulée dans un composant. Les aspects correspondent donc souvent aux exigences non-fonctionnelles d'un système. Par exemple, la synchronisation, la performance et la fiabilité d'une librairie digitale.

En utilisant ces termes, on entrevoit le but de la programmation orientée aspect : aider le programmeur à séparer clairement les aspects et les composants les uns des autres en offrant des mécanismes qui permettent de les abstraire et de les composer pour obtenir le système général [Baltus, 2002]. L'indépendance de chaque problématique rend le système final bien structuré et facilite son évolution et maintenance dans l'avenir.

En général, le cycle de développement dans la programmation orientée aspect se fait en trois étapes :

- **La décomposition aspectuelle.** Consiste à décomposer les besoins afin d'identifier et séparer les problématiques transversales et métiers.
- **Implantation des préoccupations.** Consiste à implanter chaque problématique séparément. Les fonctionnalités métiers ont implantées moyennant les techniques conventionnelles de la programmation orientée objet, alors que les préoccupations transversales, représentées par des aspects, peuvent être exprimés selon deux approches :
  - Concevoir un langage spécifique (ou étendre un langage existant par de nouvelles constructions syntaxiques),
  - Fournir un cadre logiciel (framework) ajoutant le support des paradigmes aspects.

- **Recomposition aspectuelle.** On parle ici du *Tissage*, c'est-à-dire la composition du système final comportant les différentes préoccupations, cette étape est basée sur des règles prédéfinies lors de l'implémentation.

La figure 5 résume ces trois étapes.

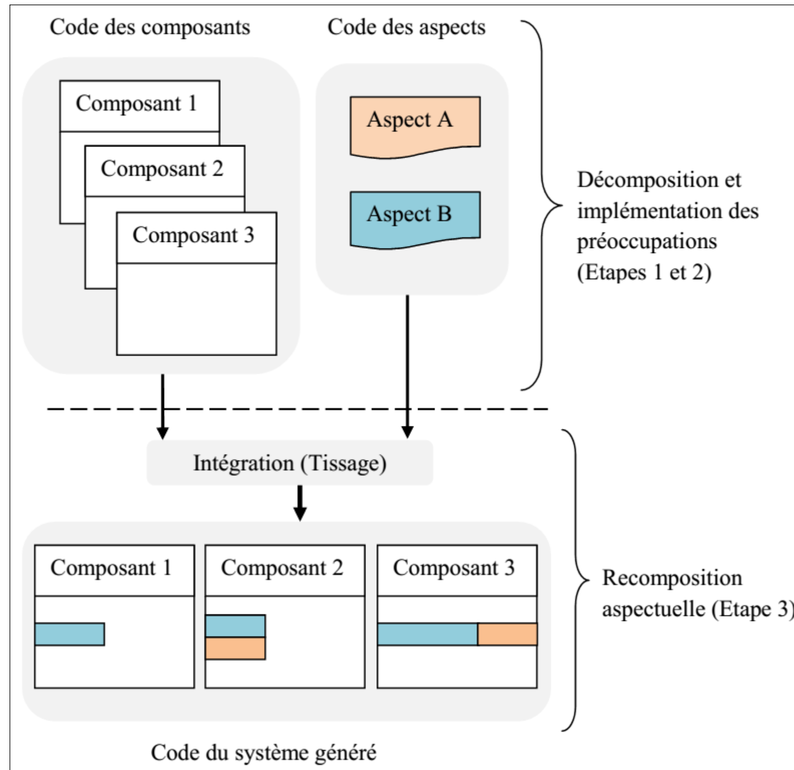

Figure 5 Intégration du code des composants et des aspects pour former le système final

2.2 Les filtres de composition

L'approche par filtres de composition [Aksit, 1998], [Bergmans, 1994] a été développée par le groupe TRESE, au département d'informatique de l'université de Twente, au Pays-Bas [Filters, Web]. Cette approche est motivée par la difficulté d'adresser, de manière indépendante et séparée, la coordination de messages complexes échangés entre objets en collaboration pour réaliser une ou plusieurs préoccupations particulières. Partant de ce constat, les filtres de composition proposent d'étendre le paradigme Objet en ajoutant le concept de filtre de messages [Hachani, 2006], ce nouveau concept ajoute aux objets de l'application des filtres qui interceptent l'envoi et la réception de messages. Les messages entrants (respectivement sortants) sont soumis à l'ensemble des filtres d'entrée (respectivement de sortie). Cette fonctionnalité est obtenue à travers la réification des envois de message.

Un filtre est lui-même un objet qui évalue et manipule des messages réifiés. Un filtre peut accepter ou rejeter un message tout en déclenchant un certain nombre d'actions. Les filtres sont ordonnés dans un ensemble (de filtres d'entrée ou de sortie), c'est leur seule capacité de composition immédiate, ils sont tous orthogonaux deux à deux.

Un message n'est reçu par l'objet destinataire qu'après avoir passé tous les filtres d'entrée. De même, un message ne quitte l'objet émetteur qu'après avoir passé l'ensemble des filtres de sortie [Quintian, 2004].

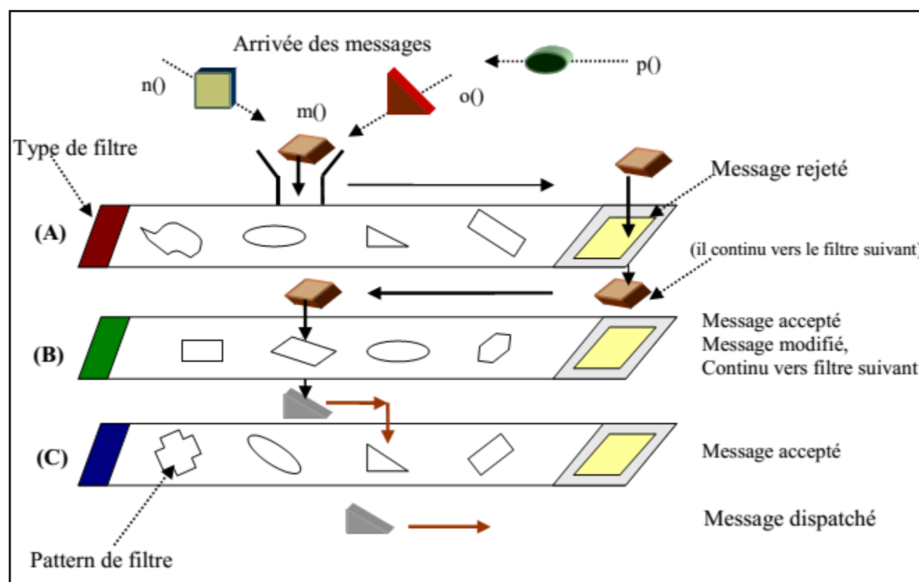

Figure 6 Schéma intuitif illustrant le filtrage des messages [Amirat, 2007]

2.3 La séparation multidimensionnelle des préoccupations

L'approche de séparation multidimensionnelle des préoccupations a été proposée par Peri L. Tarr et Harold Ossher d'IBM [Tarr, 1999]. Cette nouvelle approche s'attaque aux limites des décompositions actuelles, tout en permettant une séparation souple et claire entre les multiples préoccupations.

Le terme « multidimensionnelle » dénote une séparation des préoccupations qui implique une décomposition des programmes en plusieurs dimensions arbitraires, où chaque dimension regroupe un ensemble de préoccupations particulières. L'ensemble des classes d'une application constitue, par exemple, une dimension. Il s'agit de la dimension des préoccupations des classes [Hachani, 2006].

La structure modulaire issue de la décomposition est appelée un hyperslice (hypertranche). Un hyperslice ne contient que les unités pertinentes à une préoccupation particulière, les règles de composition sont décrites dans des HyperModules, enfin, une phase de composition nommée intégration réunit tous les comportements pour former l'application finale.

Les trois activités liées à la séparation des comportements : identification, encapsulation et intégration sont prises en charge par le modèle des HyperSpaces.

- L'identification est la sélection d'une préoccupation, elle contient les classes et les méthodes qui lui sont pertinentes. La préoccupation est alors représentée par un HyperSpace.
- Les préoccupations sont ensuite encapsulées pour être manipulées comme des classes, dans différents HyperSlices. Un HyperSlice peut contenir une ou plusieurs préoccupations.
- Finalement, pour intégrer correctement les différentes préoccupations, les HyperModules définissent les règles de composition. Notons qu'un HyperModule peut servir d'HyperSlice pour un autre HyperModule ; on peut donc le considérer comme un HyperSlice composite. [Quintian, 2004].

Enfin un hyperespace est constitué d'un ensemble d'hypermodules qui réalisent différentes modularisation des mêmes unités. Les systèmes peuvent être composés de plusieurs façons à partir de ces hypermodules, les changements et l'ajout de fonctionnalités sont effectués par l'ajout de nouvelles unités ou/et hypermodules.

```
hypermodule SEE_With_Logged_Eval_And_Check
   hyperslices: Feature.Kernel, Feature.Check,
                Feature.Eval, Feature.Logging
   relationships:
      mergeByName;
      merge Feature.Logging.LoggedClass with *;
      bracket ~_* with
         _logEntry(ClassName, MethodName)
         _logExit(ClassName, MethodName)
```

Figure 7 Exemple de règle de composition dans Hyper/J[1]

2.4 La programmation par rôles ou points de vue

La programmation par rôles [Gottlob, 1996] ou par points de vue [Bardou, 1996] et [Abiteboul, 1991] sont deux paradigmes assez proches ; ils proposent de structurer les programmes en groupes d'objets ou chaque groupe représente un point de vue du système à développer. Ils permettent de déstructurer la notion d'objets par l'introduction de vues subjectives : les rôles qu'ils jouent ou les points de vue qu'ils représentent.

Les rôles ou les vues sont donc les préoccupations dont ces deux modèles formalisent la séparation.

Il est à noter que ce paradigme ne fait pas la distinction explicite entre préoccupations de base et préoccupations transversales. En effet, chaque groupe d'objets représente une préoccupation au sens large, et toutes les préoccupations sont au même niveau. Subséquemment, même si conceptuellement, ces deux paradigmes permettent de supporter la séparation des préoccupations, leurs différentes implémentations ne permettent pas de réconcilier les problèmes de partage de données et de recouvrement de diverses définitions entre les différentes préoccupations [Quintian, 2004] [Hachani, 2006].

2.5 La programmation adaptative

La programmation adaptative a été développée par Karl Lieberherr et son équipe à l'université de Northeastern [Lieberherr, 1994], connue aussi sous le nom de programmation orientée patron.

La programmation adaptative est réalisée par l'utilisation des modèles de propagation, ces derniers sont motivés par la loi de Demeter qui suggère de ne pas relier les détails des structures des classes lors de l'écriture des méthodes. Les modèles de propagation prennent cette idée un peu plus loin en gardant les détails des structures des classes de programmes entiers autant que possible.

En se basant sur ce principe, la programmation adaptative étend le paradigme orienté objet en élevant la programmation à un niveau d'abstraction supérieur, dans une forme plus simple, ou on trouve moins de dépendances. En outre, lorsque des changements à un programme d'adaptation sont nécessaires, ils sont beaucoup plus faciles à intégrer vu la légèreté de ces programmes.

Les avantages ci-après résultent de l'utilisation des programmes d'adaptation :

- Les programmes d'adaptation se concentrent sur le cœur du problème à résoudre ce qui leur permet d'être plus simples et plus courts que les programmes orientés objets classiques.

[1] Hyper/J est une extension du langage Java, supportant la séparation multidimensionnelle des préoccupations.

- Les programmes d'adaptation améliorent la réutilisation, en effet de nombreux comportements nécessitent les mêmes caractéristiques qui sont effectivement réutilisées. [Lieberherr 1996]

3 Les principes de base de la programmation orientée aspect

Comme c'est déjà noté, la programmation orientée aspect prône une décomposition des programmes non seulement en unités modulaires représentant les préoccupations fonctionnelles de base, mais aussi en unités modulaires dédiées à la représentation des préoccupations transversales. Elle offre de plus des solutions adéquates de composition de préoccupations (on parle aussi de tissage), afin de construire des systèmes efficaces.

Dans ce qui suit nous allons présenter les concepts de base de ce nouveau paradigme (La plus part de ces définitions sont tirées des deux livres [Pawlak, 2004] et [Laddad, 2009]).

3.1 Le concept d'Aspect

La notion d'aspect est au cœur de la POA. En programmation procédurale, un découpage est fait selon les traitements à effectuer (i.e. procédures, fonctions). En programmation orientée objet, le découpage se fait en encapsulant les différentes entités manipulées au sein de classes comprenant les données et traitements relatifs à ces entités. En programmation orientée aspect, un aspect est une entité logicielle permettant d'encapsuler une fonctionnalité transversale d'une application, il comprend les données et les traitements nécessaires à l'exécution de cette fonctionnalité.

La définition d'un aspect est quasiment identique à celle d'une classe en programmation orientée objet, cependant la différence est que :

- Les classes manipulent des entités techniques, métier et présentation,
- Les aspects représentent toutes les fonctionnalités transversales de l'application

Le lien entre ces deux types d'entités se fait grâce aux points de jonction et aux coupes (ou points de coupure) que nous présenterons dans ce qui suit.

3.2 Le concept de point de jonction

Un point de jonction est un endroit dans le code du programme où seront utilisés un ou plusieurs aspects. Ce point de jonction est déterminé lors de la programmation par le développeur. L'utilisation d'un aspect pour un point de jonction se fait de manière dynamique, c'est-à-dire une fois le programme en exécution.

Il existe plusieurs types de points de jonction, selon ce que le développeur souhaite intercepter. Ainsi, un point de jonction peut intervenir pour:

- Une classe
- Une interface
- L'exécution d'une méthode
- L'accès à une variable de classe
- L'exécution d'un bloc de code
- L'exécution d'un mot-clé de langage (condition, boucle, ...)

Potentiellement, un programme comporte un nombre important de points de jonction. Il est donc pertinent de sélectionner les plus appropriés au travers de coupes.

3.3 Le concept de coupe

Une coupe (ou un point de coupure) désigne un ensemble concret de points de jonction à prendre en compte dans une application. Une coupe est définie à l'intérieur d'un aspect. Un aspect peut contenir une à plusieurs coupes selon la complexité du projet.

Les notions de point de jonction et de coupe sont étroitement liées. Une coupe, de par sa définition, désigne plusieurs points de jonction. Un point de jonction peut quant à lui être référencé par différentes coupes.

3.4 Le concept de consigne (Advice)

Une consigne (advice en anglais) représente un comportement technique particulier d'un aspect. Concrètement, c'est un bloc de code qui sera greffé dans l'application à l'exécution d'un point de jonction définit dans une coupe.

Un aspect peut contenir une à plusieurs consignes. De même qu'un aspect peut être assimilé à une classe en programmation orientée objet, un code de consigne pourra être comparé à une méthode de l'aspect.

Comme nous l'avons vu précédemment, une coupe peut contenir plusieurs points de jonction, qui eux-mêmes peuvent référencer plusieurs aspects. Une consigne est donc susceptible d'être utilisée à plusieurs endroits dans l'application et un point de jonction peut éventuellement greffer plusieurs consignes au moment de l'interception.

Différents types de consignes peuvent être identifiés, selon la manière dont ils sont exécutés à l'apparition d'un point de jonction définit par une coupe.

3.4.1 Les consignes before

Une consigne de type before, exécute son bloc de code avant l'exécution du code intercepté. Le déroulement d'un type before peut être récapitulé ainsi:

- exécution du programme
- arrivée à un point de jonction
- exécution du code de la consigne
- exécution du code intercepté
- fin du point de jonction, suite de l'exécution du programme

3.4.2 Les consignes after

Le bloc du code, d’une consigne de type ***after*** est exécuté après l'exécution du code intercepté[1].

Le bloc de code de ce type est exécuté si l'exécution du code intercepté s'est correctement effectuée. Le déroulement peut être récapitulé ainsi:

- exécution du programme
- arrivée à un point de jonction
- exécution du code intercepté
- exécution du code de la consigne
- fin du point de jonction, suite de l'exécution du programme

[1] On peut distinguer deux types de consignes after:
- After returning : ce type est exécuté si l'exécution du code intercepté s'est correctement effectuée.
- After error : est exécuté si l'exécution du code intercepté a provoqué une erreur.

3.4.3 Les consignes around

Une consigne de type ***around,*** exécute son bloc de code avant et après l'exécution du code intercepté. Ainsi, une consigne ***around*** est composée de deux parties avant et après. Le déroulement d'une greffe around peut être récapitulé ainsi:

- exécution du programme
- arrivée à un point de jonction, exécution de la partie avant du code de la consigne
- Si certains traitements de la partie avant imposent la non-exécution du code intercepté (par exemple, pour des raisons de sécurité), le point de jonction ne fera pas appel à la méthode qui permet d'exécuter le code intercepté. Cette dernière si elle n'est pas appelée, le code intercepté est remplacé par le code de la consigne.
- exécution du code intercepté
- exécution de la partie après du code advice
- fin du point de jonction, suite de l'exécution du programme

3.5 Le concept d'introduction

Les mécanismes de coupes et de consignes que nous avons vus jusqu'à présent permettent de définir des aspects qui étendent le comportement d'une application. Les coupes désignent des points de jonction dans l'exécution de l'application (appel de méthode, exécution, lecture d'un attribut, etc.), tandis que les codes des consignes ajoutent du code avant ou après ce point.

Dans tous les cas, il est nécessaire que les codes correspondant aux points de jonction soient exécutés pour que les codes des consignes le soient également. Si les points de jonction n'apparaissent jamais, les codes des consignes ne sont jamais exécutés, c'est-à-dire ne seront jamais ajoutés à l'application.

Le mécanisme d'introduction permet d'étendre le comportement d'une application en lui ajoutant des éléments, essentiellement des attributs ou des méthodes. Contrairement aux codes des consignes, qui étendent le comportement d'une l'application si et seulement si certains points de jonction sont exécutés. Le mécanisme d'introduction est sans condition : l'extension est réalisée dans tous les cas.

3.6 Le concept de tissage

Le tissage (weaving en anglais) est une opération automatique qui est absolument nécessaire pour obtenir une application opérationnelle, cette étape sert à intégrer les fonctionnalités des classes et celles des aspects.

Le programme qui réalise cette opération est appelé un tisseur d'aspects. L'application obtenue à l'issue du tissage est dite tissée.

3.6.1 Les règles de tissage

Les règles de tissage précisent la façon d'intégrer les préoccupations pour former le système final. Ces règles sont définies dans les aspects et spécifiées par le concepteur de l'application afin de recouper correctement les préoccupations entre-elles.

Le langage utilisé pour spécifier ces règles peut être une extension d'un langage standard, ou bien une chose totalement différente. Par exemple utiliser un langage basé sur XML.

Exemple :

Avant chaque ***opération*** sur le système bancaire, le client doit être ***authentifié.***

Règle de tissage unité fonctionnelle une préoccupation

3.6.2 Le tisseur

Le tisseur est un outil spécial utilisé pour greffer le code des aspects dans le code des méthodes des classes suivant les règles de tissage prédéfinies.

Figure 8 Tisseur POA

On distingue deux types de tisseurs, en fonction du moment de leur utilisation. Nous les expliquons ci-après.

3.6.3 Le tisseur statique (Tissage à la compilation)

Dans ce cas le tisseur est un programme qui, avant l'exécution, prend en entrée un ensemble de classes et un ensemble d'aspects et fournit en sortie une application augmentée des aspects. Un tel tisseur est donc très proche d'un compilateur, on le distingue parfois sous le terme de compilateur d'aspect.

Les classes fournies en entrée du tisseur peuvent se présenter soit sous forme de code source, soit sous forme de code intermédiaire (*bytecode*). Il y a aussi des solutions mixtes, dans lesquelles certaines parties de code sont du code intermédiaire. Tandis que d'autres sont du code source.

Un tisseur admettant en entrée du code intermédiaire est généralement plus intéressant qu'un tisseur n'acceptant que du code source. En effet, on est alors capable de tisser des applications, commerciales ou fournies par des tiers, dont le code source n'est pas disponible. De plus, le tisseur est plus simple à réaliser. En effet, l'analyse du code en entrée est alors une analyse de code intermédiaire plutôt qu'une analyse de code source, plus complexe. De plus, le code intermédiaire est exempt d'erreur de syntaxe, ce qui n'est pas forcément le cas de code source, une conséquence de cette simplification est que le tisseur est aussi plus performant.

La nature du code fourni en sortie par le tisseur dépend de celle des classes en entrées. Si le code en entrée est du code source, la sortie peut être soit du code source, soit du code intermédiaire. Par contre, si le code en entrée est du code intermédiaire, la sortie est nécessairement sous forme de code intermédiaire.

Lorsque la sortie est sous forme de code source, les applications produites sont lisibles. Les développeurs peuvent facilement consulter ce code source et étudier les effets des aspects sur l'application. Ce code source doit toutefois être compilé avec un compilateur traditionnel, ce qui ralenti la chaine de production et rend difficile une compilation incrémentale.

A titre d'exemple, les premières versions d'AspectJ [AspectJ, Web], généraient du code source. Depuis la version 1.1, AspectJ ne génère plus que du code intermédiaire. A moins d'utiliser un décompilateur de code intermédiaire, il n'est donc plus possible de consulter le code produit par AspectJ.

Avec le tissage à la compilation, les aspects sont pris en compte lors de la compilation pour produire une application tissée. Ils ne subsistent pas en tant que tel au moment de l'exécution. Leurs effets sont certes présents dans le code final, mais il n'y a aucun moyen de distinguer le reste du code. Il n'est donc pas possible de retirer des aspects, d'en ajouter de nouveaux ou de modifier la façon dont ils ont été tissés. Ce type de tissage est dit statique.

3.6.4 Le tisseur dynamique (Tissage à l'exécution)

Dans le cas du tissage à l'exécution, le tisseur est un programme qui permet d'exécuter à la fois l'application et les aspects qui lui ont été ajoutées.

La principale différence avec le tissage statique réside dans la manière dans les aspects sont associés au code métier. Dans AspectJ, par exemple, l'approche statique spécifie dans la définition de l'aspect avec quelles classes il doit être tissé. Mais les solutions dynamiques, comme JBoss AOP [JBoss AOP, Web], AspectWerkz [AspectWerkz, Web] ou encore AspectJ 5, préfèrent utilisés un document XML pour associer les aspects au code métier.

Avec un tel tisseur, application et aspects doivent nécessairement se présenter sous forme de code intermédiaire. Ils doivent donc avoir été préalablement compilés. L'avantage d'une telle technique est que les aspects peuvent être ajoutés, enlevés ou modifiés à chaud, pendant que l'application s'exécute. Cette caractéristique s'avère intéressante pour les applications qui ont de fortes contraintes de disponibilité et qui ne peuvent être arrêtées pendant de longues périodes. Tel est le cas, par exemple, des serveurs d'applications. Un tel mode de tissage est dit dynamique.

Dans la plupart des cas, le tissage à l'exécution doit être précédé d'une phase d'adaptation. Cette dernière a pour rôle de préparer l'application en vue de l'exécution des aspects. Concrètement, les parties de l'application qui sont susceptibles d'être modifiées par des aspects le sont, et le fil d'exécution normal du programme est redirigé vers le tisseur. Ce dernier détermine si une instruction d'aspect doit être exécutée. La phase d'adaptation est réalisée juste avant l'exécution, lors du chargement des classes dans la machine virtuelle.

Si l'étape d'adaptation n'était pas mise en œuvre, l'application doit être exécutée dans un mode de supervision proche de ce qui se passe avec un débogueur. La supervision aurait alors pour fonction de déterminer, à chaque pas d'exécution de l'application, si un aspect doit être ajouté ou non. Le temps d'exécution d'un tel processus serait prohibitif en termes de coût. La phase d'adaptation évite cet inconvénient en identifiant préalablement dans les applications les emplacements ou les instructions d'aspects sont applicables.

4 Travaux sur l'approche orientée aspect

L'apparition de l'approche aspect, a apporté de fortes motivations au monde du génie logiciel, plusieurs travaux couvrant tout les phases de cycle de développement de systèmes logiciels, ont ainsi vu le jour.

- Analyse des besoins,
- Conception,
- Développement de logiciels

- …

D’autres se sont intéressés à l’influence et les conflits causés par les interactions entre les nouveaux concepts de la programmation orientée aspect, et les classes de bases. Ces études se basent sur des méthodes formelles, plus particulièrement, la vérification formelle de ces interactions [Mostefaoui, 2007], [Chen, 2010], [Katz, 2003].

La réingénierie est aussi présente, où plusieurs travaux proposent d’analyser les applications à base d’objets déjà existantes, pour extraire des aspects en vue de les réécrire dans une approche Aspect, comme par exemple le travail de [Ettinger, 2004].

Un autre domaine de recherche qui est très actif, et qui touche un des axes de notre recherche, concerne l’implémentation des patrons de conception, comme ceux de GoF [Gamma, 1995] avec les concepts aspect. Les premiers travaux dans ce domaine ont eu comme objectif, d’éclairer et tester les apports de la programmation orientée aspect dans la réalisation des différents patrons [Hannemann, 2002], [Garcia, 2006] d’autres pour offrir aux développeurs des nouveaux patrons basés sur les nouveaux concepts de programmation [Lorenz, 1998], [Noda, 2001], [Nordberg, 2001]. Cependant, la majorité de ces travaux ont pris AspectJ comme le représentant majeur de la programmation orientée aspect, dans la plus part de leurs études.

5 Conclusion

Les préoccupations transversales sont le cœur des principaux problèmes observés lors de la conception et l’implémentation des logiciels avec les approches classiques. Elles se dispersent dans les différents modules constituant le système, ce qui perturbe la compréhension, et rend l’évolution et la maintenance plus complexes.

La programmation orientée aspect, les filtres de composition, la séparation multidimensionnelle des préoccupations, la programmation par rôles ou la programmation adaptative, sont toutes des nouvelles approches riches et ambitieuses, chacune d’entre elles tente de résoudre les problèmes des préoccupations transversales en offrant des modules de décomposition et d’organisation permettant d’obtenir des applications compréhensibles et facilement manipulables.

Grâce au concept d’aspect, la programmation orientée aspect permet avec élégance d’intégrer à une application orientée objet des fonctionnalités transversales en s’appuyant sur les notions complémentaires de coupes, de points de jonction et de code de consigne. L’opération d’intégration est désignée sous le terme de tissage, elle peut être réalisée au moment de la compilation, ou pendant l’exécution de l’application.

L’importance des concepts proposés par la programmation orientée aspect, ont fait apparaitre plusieurs langages pour les mettre en œuvre dans le monde réel. Le chapitre suivant, présente trois différents projets créés pour satisfaire les concepts de base de la programmation orientée aspect.

Chapitre 2 : AspectJ & JBoss AOP & CaesarJ

1 ASPECTJ

1.1 Le concept d'aspect
1.2 Le concept de point de jonction
1.3 Le concept de coupe
1.4 Le concept de consigne (Advice)
1.5 Le concept d'introduction
1.6 Le concept de tissage

2 JBOSS AOP

2.1 Le concept d'aspect
2.2 Le concept de point de jonction
2.3 Le concept de coupe
2.4 Le concept de consigne (Advice)
2.5 Le concept d'introduction
2.6 Le concept de tissage

3 CAESARJ

3.1 Le concept des classes virtuelles
3.2 Le composant CaesarJ
3.3 Le concept de tissage

4 BILAN

5 CONCLUSION

***Résumé* :**

La notion d'aspect n'est attachée à aucun langage particulier, de même que nous pouvons programmer avec des objets en C++, Java ou Smalltalk, nous pouvons utiliser des aspects dans n'importe quel langage.

Les premières expériences de la programmation orientée aspect (POA) ont été réalisées autour de Java, qui est le langage le plus populaire dans la communauté des chercheurs en informatique. Entre ces nombreux projets on trouve les trois approches AspectJ, JBoss AOP et CaesarJ, où chacune à une façon propre pour implémenter les concepts de base de la programmation orientée aspect.

Ce chapitre traite de la mise en œuvre des concepts aspect avec les trois approches : AspectJ, JBoss AOP et CaesarJ.

1 AspectJ

Conçu et développé par Gregor Kiczales, l'inventeur du concept de la programmation orientée aspect, et son équipe du centre de recherche PARC (Palo Alto Research Center) de Xerox. Les premières versions d'AspectJ datent de 1998. Depuis décembre 2002, le projet AspectJ a quitté le PARC et rejoint la communauté Open Source Eclipse. Il est ainsi devenu un plug-in (AJDT) standard pour la plate-forme de développement Eclipse.

AspectJ met on œuvre les concepts de base de la programmation orientée aspect comme suit.

1.1 Le concept d'aspect

AspectJ étend la syntaxe de Java avec de nouveaux mots-clés. Le premier que nous rencontrons ici est *aspect*.

Pour AspectJ un aspect est une entité logicielle très similaire à une classe (Le mot-clef *class* est juste remplacé par *aspect*). Un aspect a un nom, et peut être définit dans des packages, il peut aussi être étendu par héritage. Son rôle est de définir une préoccupation comme fait une classe. Bien que les différences se situent dans le caractère transversal de la préoccupation modularisée et dans la logique d'intégration liée a un aspect, classes et aspects sont des entités de même niveau, qui doivent obéir aux mêmes règles, dans la mesure du possible.

Les aspects AspectJ comportent des coupes et des codes advice. Les sections qui suivent détaillent ces éléments.

1.2 Le concept de point de jonction

Les types de points de jonction fournis par AspectJ peuvent concerner des méthodes, des attributs, des exceptions, des constructeurs ou des blocs de code statiques. Un dernier type concerne les exécutions de codes advice.

Le tableau suivant résume les différents types de point de jonction offerts par AspectJ [Meslati, 2006].

Tableau 1 Signification prédicative des points de jointure primitifs et leur composition dans AspectJ

Point de jointure	Valeur du prédicat correspondant (cjp : signifie point de jointure courant)	Commentaire
Call(S)	Vrai si cjp correspond à un appel de *S*	*S ::= ResultatType e.g. ClassName.MethodName (Paramètres)*
Execution(S)	Vrai si cjp correspond à une exécution de *S*	
Get(S)	Vrai si cjp correspond à un accès à *S*	*S ::= Type ClasseName.FieldName*
Set(S)	Vrai si cjp correspond à une affectation de *S*	
Initialisation(S)	Vrai si cjp correspond à l'exécution de l'initialisation d'un objet de *S*	*S ::= ClassName(Parametres)*
Preinitialiation(S)	Vrai si cjp correspond à l'exécution du pré initialisation d'un objet de *S*	
Staticinitialisation(S)	Vrai si cjp correspond à l'exécution de l'initialisation de la classe *S*	*S ::= ClassName*
Handler(TP)	Vrai si cjp correspond à la manipulation de l'exécution TP dans un ploc de capture d'exécution	*TP* spécifie le type d'exécution
Within(TP)	Vrai si cjp correspond l'exécution d'un code appartient à *TP*	*TP* est le nom d'une classe
Withincode(TP)	Vrai si cjp correspond l'exécution d'un code défini dans une méthode ou un constructeur spécifie par *S*	S est une méthode ou une signature d'un constructeur
Cflow(P)	Vrai si cjp est dans le flot de contrôle du point de jointure défini par *P* (incluant *P* lui-même)	*P* est un point de coupure
Cflowbelow(P)	Vrai si cjp est dans le flot de contrôle bas du point de jointure défini par *P* (excluant *P* lui-même)	
Adviceexecution()	Vrai si le corps en exécution appartient à une consigne	
This (TP or Id)	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)	*TP* est un nom de classe et *Id* un identificateur '..' remplace n'importe quel nombre de paramètres
Target(TP or Id)	Vrai si la cible du cjp est un objet spécifie par TP ou Id	
Args(TP or Id or '..')	Vrai si les arguments du cjp sont des instances dont le type est spécifié par TP ou Id	
If(*BoolExp*)	Vrai si *BoolExp* est vrai	*BoolExp* est une expression booléenne
! P	Vrai si P n'est pas satisfait	P, P1 et P2 sont des points de coupures
P1 && P2	Vrai si P1 et P2 sont satisfaits	
P1 || P2	Vrai si P1 ou P2 ou les deux sont satisfaits	
(P)	Vrai si P est satisfait	

1.3 Le concept de coupe

À l'intérieur d'un aspect, le mot-clé *pointcut* définit une coupe. D'une façon générale, un aspect peut comporter autant de coupes que nécessaire.

Une coupe peut être anonyme et associée directement à un code advice. Il est en effet inutile de nommer une coupe si elle n'est utilisée que dans un seul code advice.

Avec AspectJ, une coupe a la syntaxe suivante :

```
1   pointcut nomDeLaCoupe(paramètres) :
2   définitionDeLaCoupe
```

Chaque coupe est associée à une expression, qui définit son ensemble de points de jonction. Les points de jonction sont typés, et chaque type est associé à un mot-clé. Une expression de coupe peut comporter des points de jonction de types différents.

Exemple d'un point de coupure qui contient un seul point de jonction de type call

```
1    pointcut nomDeLaCoupe (): // Coupe
2    call(public int nomClasse. nomMethode()); //point de jonction de type
     call
```

***Remarque** : Il est à noter que les déterminations des point de jonctions, utilisent des caractères spéciaux appelés wildcards*[1].

1.4 Le concept de consigne (Advice)

Nous avons vu que les coupes permettaient de décrire ou les instructions d'aspect devaient être ajoutées dans un code métier. Les codes advice définissent quant à eux les instructions d'aspect proprement dites.

De la même façon que les méthodes définissent le comportement d'une classe, les codes advice définissent le comportement d'un aspect. Plusieurs codes advice peuvent être définis dans un aspect. Chaque code advice fournit une série d'instructions écrites dans un bloc de code. Il possède également un type et est attaché à une coupe.

AspectJ fournit cinq types de code advice, les trois types principaux, *before*, *after* et *around*,, et deux types supplémentaires, *after returning* et *after throwing*. Ces derniers sont des raffinements du type *after*.

En plus de son type, un code advice est associé à une coupe. Par rapport aux points de jonction désignés par la coupe, le type d'un advice définit le moment auquel nous souhaitons que le bloc de code soit exécuté. Globalement, le bloc de code peut être exécuté avant ou après les points de jonction.

1.5 Le concept d'introduction

Comme nous l'avons expliqué dans le chapitre précédent, le mécanisme d'introduction permet d'introduire de nouveaux éléments structuraux au code d'une application.

En AspectJ ce mécanisme est connu aussi sous la désignation « déclarations inter-types », il permet de déclarer des membres dans des classes prédéfinies, ou de changer les relations d'héritages entre les classes de l'application.

Les six catégories d'éléments suivantes peuvent être ajoutées par le mécanisme d'introduction d'AspectJ : attribut, méthode, constructeur, classe héritée, interface implémentée et exception.

Le code suivant montre un exemple de déclarations inter-type. Il s'agit d'ajouter et initialiser un attribut *disabled* de type *Boolean* à la classe *Server*:

```
1    boolean Server.disabled = false;
```

L'instruction suivante permet de déclarer que les deux classes *Point* et *Line* héritent de la classe *GeometricObject* :

```
1    declare parents : (Point || Line) extends GeometricObject ;
```

[1] AspectJ fournit un mécanisme permettant, à l'aide des symboles *, .. et +, de créer des expressions englobant plusieurs méthodes. Ce mécanisme produit une syntaxe riche permettant de décrire facilement de nombreuses coupes.

1.6 Le concept de tissage

L'opération de tissage en AspectJ peut être effectuée en trois étapes différentes dans le processus de développement d'une application. Ces trois possibilités sont listées ci-dessous :

- **Lors de la compilation (Compile-time weaving)** Cette méthode est la plus simple à mettre en œuvre c'est une méthode statique qui utilise uniquement les fichiers sources du programme. Le compilateur ajc d'AspectJ procède en deux opérations, la première consistant à compiler l'application, la deuxième consistant à effectuer l'opération d'injection du code correspondant aux aspects.

- **Après la compilation (Post-compile weaving)** Cette méthode permet d'injecter du code associé à un aspect lorsque les fichiers sont déjà compilés. Les fichiers peuvent êtres contenus soit dans des .class soit dans des fichiers .jar.

- **Lors du chargement du programme (Load time weaving)**[1] Cette méthode permet d'injecter le code associé aux aspects, durant la phase de chargement de classe dans le class loader. Le chargeur de classe est adapté par un agent gérant la modification des classes lorsqu'elles sont chargées. AspectJ a toujours supporté un mode loadtime weaving dans son framework, cependant depuis la version 1.5, l'intégration est plus aisée et le développement se base uniquement sur un fichier de configuration aop.xml. Ce fichier contient la déclaration des aspects devants êtres appliqués sur les classes. Il est aussi possible de réaliser des aspects abstraits directement dans le fichier XML. Cette fonctionnalité permet par exemple d'adapter un aspect en modifiant uniquement la définition des Pointcut dans le fichier XML, sans toucher au code Java [AspectJ, Web].

2 JBoss AOP

Conçu et développé par Bill Burck avec la collaboration de contributeurs, dont Marc Fleury. Il peut s'utiliser de façon autonome ou conjointement avec le serveur d'application J2EE JBoss, dans le premier cas, la version autonome est appelée Standalone. Dans le deuxième cas, à partir de la version 4.0, le serveur d'application JBoss inclut en standard le framework JBoss AOP. Il est distribué librement et gratuitement selon les termes de la licence GNU LGPL (cette licence autorise une utilisation de JBoss AOP dans des produits commerciaux) [Pawlak, 2004].

JBoss AOP met en œuvre les concepts de base de la programmation orientée aspect comme suit (La plus part de ces définitions sont tirées des documents officiels de JBoss AOP [JBoss AOP, Document1] et [JBoss AOP, Document 2] [Pawlak, 2004]).

2.1 Le concept d'aspect

Le concept aspect en JBoss AOP comporte deux fichiers, le premier est un fichier XML qui définit les différents attachements, les introductions, les coupes et

[1] Cette méthode de tissage est spécifique à AspectJ 5, ce dernier est le fruit de la fusion entre les deux projets AspectJ et AspectWerkz en 2005 (Dans la présente thèse nous nous concentrons sur la version d'AspectJ qui étend le langage Java)

les métadonnées, et le second est un fichier Java qui fournit les codes advice associés à ces coupes.

2.2 Le concept de point de jonction

Les types de points de jonction fournis par JBoss AOP peuvent concerner des classes, des méthodes, des attributs et des constructeurs.

Le tableau 2 résume les différents points de jonction :

Tableau 2: Liste des points de jonctions disponibles dans JBoss AOP

mot-clé	(signature)
execution	méthode ou constructeur
construction	Constructeur
all	Type
get/set/field	Attribute
within	Type
has	Méthode ou constructeur
hasfield	attribut

2.3 Le concept de coupe

Si AspectJ définit les coupes à l'aide de mots-clés, la déclaration des coupes en JBoss AOP peut se faire selon deux manières:

- Dans un document XML dédie, ou
- Dans la classe implémentant l'aspect, en tant qu'annotation.

La première variante est celle qui semble être la manière recommandée par JBoss. L'utilisation d'un fichier de configuration des coupes respecte la philosophie des serveurs d'application ou l'on met en place un maximum de choses dans des fichiers de configuration, pour diminuer la complexité de l'implémentation. De plus, un des objectifs de JBoss étant de pouvoir fournir des aspects prédéfinis à greffer sur une application tierce, le but est de pouvoir le faire sans modifier l'aspect lui-même.

Dans le reste de cette section, l'accent sera porté sur la configuration des aspects par XML. D'une part, parce que les possibilités de l'une et de l'autre manière sont les mêmes et d'autre part parce que la technique XML semble être préférée.

En JBoss AOP Chaque application est associée à un fichier XML de définition de coupe appelé habituellement *jboss-aop.xml*.

Plusieurs coupes peuvent être définies dans un même fichier *jboss-aop.xml*. La balise principale de ce dernier est *<aop>*.

Tout fichier XML doit avoir une structure similaire à celle-ci :

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <aop>
3     <bind pointcut=" ... expression de coupe ... " >
4     <interceptor class=" ... classe d'interception ... " />
5     </bind>
6     ....
7    </aop>
```

La première ligne est un en-tête standard en XML. Les définitions de coupe sont comprises entre les balises *<aop>* et *</aop>*. La balise *<bind>* définit une coupe à l'aide d'une expression fournie par l'attribut *pointcut*. Un fichier *jboss-aop.xml* contient autant de balises *<bind>* qu'il y a de coupes dans l'application.

À l'intérieur d'une balise *<bind>* on trouve la balise *<interceptor> ou <aspect>* qui définit l'intercepteur ou la classe aspect associé à la coupe, ces deux concepts seront expliqués dans la prochaine section.

2.4 Le concept de consigne (Advice)

Dans JBoss AOP, l'aspect est une classe java. Les méthodes de la classe sont les *Advices* ; ces méthodes sont appelées lorsqu'un certain point de jonction est exécuté. Une analogie utilisée dans la documentation officielle de JBoss AOP est que l'advice est un gestionnaire d'événement, l'événement étant l'occurrence d'un point de jonction dans le flux du programme [Bodmer, 2006].

JBoss AOP fournit cinq types d'advices: Around, before, after, after-throwing et finally.

On trouve aussi un autre type d'advice dans JBoss AOP qui est l'intercepteur.

2.4.1 Les intercepteurs

Un intercepteur est un type particulier d'aspect qui ne contient qu'une seule consigne. La signature de cette consigne est définie par une interface, org.jboss.aop.advice.Interceptor:

```
1    package aop.jboss;
2    import org.jboss.aop.advice.Interceptor;
3    import org.jboss.aop.joinpoint.Invocation;
4     public class MyInterceptor implements Interceptor {
5     public String getName() { return "unNom"; }
6     public Object invoke(Invocation invocation) throws Throwable {
7      System.out.println("Code avant");
8      Object rsp = invocation.invokeNext();
9      System.out.println("Code après");
10     return rsp;
11   } }
```

L'interface *org.jboss.aop.advice.Interceptor* définit deux méthodes

- **getName.** retourne une chaîne de caractères, cette dernière représente le nom qu'on souhaite attribuer à l'intercepteur. Par convention, il s'agit souvent du nom de la classe, mais n'importe quelle autre valeur peut être choisie.
- **Invoke.** est appelée par le Framework JBoss AOP juste avant un des points de jonction de la coupe associée à l'intercepteur. La signature de la méthode *invoke* est imposée : un seul paramètre de type Invocation est autorisé et le type de retour est Object. Le type d'exception *Throwable* doit être présent. où :
  - Le paramètre de type Invocation permet d'introspecter le point de jonction.
  - Le type Object constitue la valeur retournée par l'intercepteur à l'appelant.
  - Throwable est le type racine de toutes les exceptions et erreurs en Java. Sa mention indique que la méthode invoke est susceptible de lever n'importe quelle exception ou erreur.

Dans un intercepteur, l'appel de la méthode *invokeNext* permet de délimiter les parties de code qui s'exécutent avant et après le point de jonction. La méthode *invokeNext* doit être appelée sur l'objet de type Invocation passé en paramètre de la méthode *invoke*. La méthode *invokeNext* joue pour JBoss AOP le même rôle que *proceed* pour AspectJ.

Lorsqu'un point de jonction ne comporte qu'un seul intercepteur, la méthode *invokeNext* exécute le code correspondant au point de jonction. Lorsque plusieurs intercepteurs sont greffés autour du même point de jonction, *invokeNext* invoque

l'intercepteur suivant ou le code correspondant au point de jonction si le dernier intercepteur est atteint.

L'appel de la méthode *invokeNext* est facultatif. Si elle n'est pas appelée, le code correspondant au point de jonction n'est pas exécuté. Ce comportement correspond à des aspects, par exemple un aspect de sécurité, qui ne souhaité pas exécuter ce code ou qui souhaité le remplacer par un autre comportement.

La méthode *invokeNext* peut être appelée plusieurs fois. Dans ce cas, le code correspondant au point de jonction est exécuté à plusieurs reprises. Ce comportement correspond à des aspects qui tentent d'exécuter à plusieurs reprises l'application, par exemple en cas de défaillance.

Comme nous l'avons déjà mentionné, la méthode *invoke* retourne une valeur de type Object. Ce type correspond à la valeur retournée par le code du point de jonction. Il est nécessaire de propager cette valeur à l'appelant.

2.5 Le concept d'introduction

Le mécanisme mix-in de JBoss AOP permet d'étendre le comportement d'une application. Ce mécanisme est identique au mécanisme d'introduction d'AspectJ. Concrètement le mécanisme mix-in consiste à ajouter aux classes existantes de l'application des interfaces ou des attributs et des méthodes provenant d'une classe.

Le mécanisme mix-in atteint donc, par d'autres biais, le même objectif que l'héritage : il permet d'étendre une classe. Contrairement à l'héritage, cependant, il ne permet pas de redéfinir ou de surcharger des méthodes existantes.

Le mécanisme mix-in se définit dans le fichier *jboss-aop.xml* au moyen de la balise *<introduction>* associée à l'attribut class. Cet attribut est une expression régulière Java qui indique la ou les classes étendues par le mécanisme mix-in. La balise est suivie d'une balise *<mixin>*.

Le fragment de fichier XML suivant illustre la définition d'un mécanisme mix-in concernant la classe aop.jboss.Order :

```
1    <introduction class="aop.jboss.Order">
2     <mixin>
3     ...
4     </mixin>
5     </introduction>
```

Les balises *<introduction>* cohabitent au sein des fichiers *jboss-aop.xml* avec les balises de définition de coupe que nous avons vues jusqu'à présent. Elles sont écrites entre les balises *<aop>* et *</aop>*, qui délimitent le début et la fin d'un fichier *jboss-aop.xml*.

Deux types d'éléments peuvent être ajoutés : des interfaces et des classes. Chacun de ces éléments est associé à une balise XML.

La balise *<interfaces>* fournit les noms de la ou des interfaces ajoutées. Lorsqu'il y a plusieurs interfaces, leurs noms sont séparés par une virgule.

La balise *<class>* donne le nom de la classe à ajouter. Tous les attributs et toutes les méthodes de cette classe sont ajoutés à la ou aux classes initiales.

Conceptuellement, le mécanisme mix-in étend une classe existante. Cependant, en pratique, deux instances subsistent lorsque le programme s'exécute : celle de la classe initiale, non étendue, et celle de la classe correspondant à l'extension.

Une troisième balise *<construction>* est disponible dans la définition d'un mécanisme mix-in pour indiquer comment cette seconde instance doit être créée.

Concrètement, cette balise fournit, à l'aide de l'instruction *new*, la ligne de code qui permet de créer l'instance.

Par exemple, le fragment XML suivant :

```
1    <construction> new aop.jboss.Calendar(this) </construction>
```

Ce code indique que l'instance étendue doit être créée en appelant le constructeur de la classe *aop.jboss.Calendar* et en lui passant en paramètre la référence *this*.

La création de l'instance étendue s'effectue dans le contexte de l'instance initiale. La référence *this* correspond à la référence de l'instance initiale. Nous sommes de la sorte à même de transmettre à l'instance étendue la référence de l'instance initiale. L'instance étendue connaît la référence de l'instance qu'elle étend.

2.6 Le concept de tissage

JBoss AOP applique les aspects au code du programme en modifiant le bytecode java. Cette opération peut avoir lieu à trois moments:

- **Lors de la compilation.** Les classes sont instrumentées préalablement à l'exécution par un compilateur d'aspect.

- **Lors du chargement du programme.** Les classes sont instrumentées lors du premier chargement de celles-ci.

- **Durant l'exécution (Hot Deployment).** JBoss AOP permet d'ajouter et de supprimer des advices de manière dynamique, lors de l'exécution. Cette possibilité peut s'avérer très intéressante pour appliquer des aspects de test sur un système en production. Il est particulièrement simple de procéder de la sorte dans le cadre du serveur d'application.
  Les points de jointure sont "détectés" lors de l'exécution et le programme et alors modifié à ce moment là. Cette manière de faire a évidemment des incidences sur les performances d'exécution, en revanche, elle ouvre la voie au déploiement "à chaud" des aspects. Ce mode de fonctionnement fait probablement partie d'un des objectifs initiaux de l'implémentation AOP de JBoss, car elle a tout son sens dans le cadre de l'utilisation des aspects dans le serveur d'application J2EE JBoss AS. On peut ainsi enclencher et déclencher des aspects sur un service en production.
  Pour la modification du bytecode, c'est la librairie Javassist qui est utilisée par le tisseur pour manipuler la classe compilée. Cet environnement permet de manipuler du bytecode sans jamais devoir se préoccuper de sa forme. En effet, le programme est abstrait dans une structure de données que l'on peut manipuler à l'aide d'une API.

3 CaesarJ

CaesarJ, conçu et développé par l'université de technologie de Darmstadt en Allemagne, il suit une nouvelle démarche de programmation qui combine les nouveaux concepts de la programmation orientée aspect, comme les points de coupure et les advices, avec les mécanismes avancés de la programmation objet [Aracic, 2006].

CaesarJ prend en charge des concepts supplémentaires tels que des classes virtuelles, la composition mixin et le binding.

La structure de cette section est différente des deux précédentes, cela est parce que :

- CaesarJ utilise les mêmes syntaxes de points de jonctions et de coupures d'AspectJ.
- CaesarJ a des nouveaux mécanismes et concepts propre à lui-même, et qui sont la base de son fonctionnement.

Nous nous concentrons dans ce qui suit sur les particularités de CaesarJ.

3.1 Le concept des classes virtuelles

CaesarJ prend en charge deux types de classes : des classes Java et des classes CaesarJ, ces dernières offrent des fonctionnalités supplémentaires et spécifiques à CaesarJ, et elles ont une sémantique légèrement différente des anciennes classes.

Les classes CaesarJ sont déclarées en utilisant le nouveau mot clé *cclass*. Ces classes ont la capacité de traiter des classes internes (i.e. classes virtuelles) de la même manière que les méthodes et les attributs dans les classes Java [Braz, 2009], cette capacité permet à ces classes d'êtres polymorphiquement redéfinies par ses sous-classes, ainsi on peut avoir une préoccupation très importante en assemblant plusieurs ensembles de cclasses.

Dans l'exemple suivant les deux classes internes *Nœuds* et *Arêtes* sont imbriquées dans la classe CaesarJ *Graphes* :

```
1   public cclass Graphes {
2    public cclass Nœuds {
3     ...
4    }
5    public cclass Arêtes {
6     ...
7    }
8   }
```

Toutefois, la relation entre les classes CaesarJ et celles de Java n'est pas très ouverte, où on trouve plusieurs restrictions comme :

- Une classe Java ne peut pas être imbriquée dans une classe CaesarJ (Le contraire est vrai aussi),
- Les interfaces ne peuvent pas être imbriquées dans une classe CaesarJ,
- Une classe Java ne peut pas étendre une classe CaesarJ (Le contraire est vrai aussi)

3.1.1 Le concept d'héritage implicite

Les familles des classes détiennent un ensemble de collaborations entre leurs classes internes (virtuelles), ces collaborations doivent être assurées et accessibles par les objets de ces classes. CaesarJ permet de définir une variation d'abstractions représentées comme des classes virtuelles en raffinant les supers classes dans les sous-classes de la famille. La différence entre le raffinement des classes virtuelles dans CaesarJ et le raffinement conventionnel des sous-classes, est que les références relatives à une classe virtuelle sont liées dynamiquement par les objets des familles. La figure suivante illustre un exemple d'héritage entre deux classes virtuelles de CaesarJ.

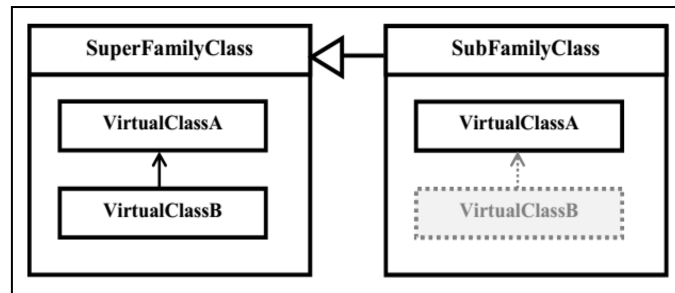

Figure 9 Héritage implicite dans CaesarJ

La figure 9, présente deux classes de la même famille, la super classe *SuperFamilyClass* et sa sous classe *SubFamilyClass*. Dans *SuperFamilyClass* on trouve deux classes virtuelles, *VirtualClassA* et *VirtualClassB* où *VirtualClassB* fait une référence à *VirtualClassA*. Dans la sous classe *SubFamilyClass* il ya une seule classe virtuelle *VirtualClassA* qui raffine *VirtualClassA* définie dans *SuperFamilyClass*.

A la différence des mécanismes d'héritage classiques, toutes les références relatives à *VirtualClassA* des *VirtualClassB* des objets de *SubFamilyClass* sont liées à la *VirtualClassA* raffinée. Cet effet est indiqué par les ombres grises dans *VirtualClassB*.

La classe *VirtualClassA* est aussi raffinée et reflète un autre mécanisme prévu par les classes virtuelles, c'est l'héritage implicite, en effet, les deux classes *SuperFamilyClass* et *SubFamilyClass* définissent une classe virtuelle nommée *VirtualClassA*, et *SubFamilyClass* étends *SuperFamilyClass*, ce qui crée une relation d'héritage implicite entre les deux classes de la famille. *VirtualClassA* dans *SubFamilyClass* remplace la *VirtualClassA* dans *SuperFamilyClass*, cela rend l'instanciation de *VirtualClassA* dépends de la famille des objets dans laquelle elle a été consultée [Braz, 2009].

3.1.2 Le concept de polymorphisme des familles

Le polymorphisme des familles traite le problème d'expression des familles des classes virtuelles et la gestion de leurs relations polymorphes, il permet aux développeurs la flexibilité de redéfinir les classes en garantissant que celles des différentes familles ne seront pas mélangées.

Grâce au polymorphisme des familles, il est possible de déclarer statiquement et de gérer les relations entre plusieurs classes polymorphes, d'une manière où un ensemble donné de classes est connu pour constituer une famille et les relations entre ses membres, sans préciser la relation statique entre elles.

Le paradigme objet manque de mécanismes pour représenter les familles des classes liées explicitement, cela pose des problèmes de cohérence, car les objets issus de familles des classes indépendantes peuvent être mélangés, cette limitation force les développeurs à prendre une décision entre la flexibilité et la sécurité. En conséquence, il arrive qu'un grand nombre de combinaisons incompatibles soient autorisées, alors que, plusieurs combinaisons correctes soient bloquées. Pour faire face à cette limitation, le motif abstrait Fabrication [Gamma, 1995] est parfois utilisé.

3.2 Le composant CaesarJ

Conceptuellement, un programme en CaesarJ peut être vu comme un composant comportant plusieurs modules. Ces modules sont : l'interface de collaboration, la partie implémentation, CaesarJ Binding et Weavelets. Chacune de ces parties à un

rôle et un niveau d’abstraction différent des autres. Nous les présenterons dans ce qui suit.

3.2.1 L’interface de collaboration

Comme nous l’avons déjà mentionné, le composant CaesarJ contient plusieurs parties qui interagissent les une avec les autres pour réaliser une tâche particulière. L’interface de collaboration, représente le niveau le plus abstrait dans le composant CaesarJ, elle contient les déclarations des rôles abstraits des participants ainsi que leurs opérations, ce qui définit la façon que ces participants doivent suivre pour interagir et collaborer entre eux.

Les rôles de cette interface sont définis par des collaborations de classes représentées par des classes virtuelles et sont souvent mutuellement récursives, dans le sens où chaque classe virtuelle se réfère à une autre classe virtuelle dans sa propre définition.

Dans la programmation orientée composant, la structure logique d’un composant est constituée de deux parties : la partie fonctionnelle qui représente le code métier du composant, et la partie non fonctionnelle qui représente les services dont le composant à besoin pour s’exécuté [Belhanafi, 2005]. En d’autre terme la première partie représente ce que le composant doit fournir à ses clients, et la deuxième, ce que le composant attend de son contexte. En anglais ces deux parties sont nommées provided et expected (i.e. Fournie / prévue en français).

Initialement, dans CaesarJ, les deux facettes fournie et prévue ont été explicitement citées par les deux mots-clés *required* et *expected*, par la suite, les développeurs sont arrivés à la conclusion que chaque utilisation concrète de ces deux facettes représente seulement un cas parmi beaucoup d'autres possibles, cette conclusion a conduit à la disparition de ces deux mots-clés [Aracic, 2006]. Néanmoins, la division entre les deux facettes est toujours présente au niveau conceptuel, même si elle n’est pas explicitement représentée dans le code source de CaesarJ, elle est représentée par les deux parties *implémentation* et *Binding* (i.e. Fixation) dans le composant CaesarJ.

La partie implémentation comprend la mise en œuvre de la partie fournie tandis que binding intègre le composant dans l'application de base pour mettre en œuvre la partie attendue. Bien qu’implémentation et binding soient placées dans des modules différents, elles sont reliées par leur interface de collaboration commune qui sert de moyen de communication [Mezini, 2003].

3.2.2 CaesarJ implémentation

CaesarJ implémentations mettent en œuvre les méthodes contextuelles héritées de l’interface de collaboration. Elles correspondent aux aspects abstraits d’AspectJ, car elles sont indépendantes de tout cas particulier.

Les classes virtuelles de CaesarJ implémentation peuvent étendre et affiner les classes virtuelles définies dans l’interface de collaboration en ajoutant des données ou méthodes nécessaires pour la mise en œuvre de la facette fournie.

Le caractère réutilisable des implémentations vient du fait qu'elles ne sont pas liées à un scénario particulier, de sorte qu'elles ont un certain niveau d'abstraction du système où on peut crée plusieurs implémentations avec différents membres de données et méthodes, selon les besoins et les objectifs du développeur, ainsi que les caractéristiques des systèmes.

3.2.3 Binding et wrappers

La partie Binding représente la colle qui relie le composant CaesarJ à une application spécifique. Elle correspond à la partie attendue du composant et joue le même rôle que les aspects concrets d'AspectJ.

La partie Binding complète la partie implémentation, bien que cette dernière implémente la partie abstraite de l'interface, binding implémente les méthodes qui enveloppent la logique spécifique du contexte du composant en concrétisant les différents rôles abstraits déclarés précédemment. Pour achever son travail, binding peut utiliser les points de coupure et les advices d'AspectJ, elle peut aussi envelopper les classes d'application pour assurer la correspondance entre les classes virtuelles déclarées dans l'interface de collaboration et les classes concrètes de l'application. Dans CaesarJ, les mécanismes primaires pour effectuer une telle correspondance sont les wrappers (i.e. en français emballages).

En effet, les classes virtuelles peuvent envelopper un ou plusieurs objets de types arbitraires, et qui deviennent ses wrappees, en ayant accès aux méthodes et champs publics de ces derniers. Les classes virtuelles peuvent étendre leur comportement afin de les adapter au composant et les associer à un ou plusieurs rôles définis dans l'interface

Les wrappers sont le mécanisme de CaesarJ qui remplace les déclarations inter-type d'AspectJ et qui représente le concept d'introduction.

3.2.4 Weavelets

Différents modules sont utilisés pour définir les facettes complémentaires fournie et attendue de l'interface de collaboration, toutefois les deux parties sont incomplètes, ce qui ne leur permet pas d'être instanciées. Une classe concrète CaesarJ appelée Weavelet permet de résoudre ce problème en utilisant une composition *Mixin* qui combine les deux parties et termine la création du composant.

La composition Mixin peut être considérée comme une forme d'héritage multiple, car elle combine plusieurs modules différents qui mettent en œuvre des éléments complémentaires d'un composant. Elle est obtenue en utilisant l'opérateur (&) et elle est caractérisée par la syntaxe suivante :

```
1   public cclass C extends A & B{ }
```

L'opérateur & définit une chaîne d'héritage entre les opérandes, l'ordre des opérandes définit dans la composition mixin indique l'ordre de la chaîne d'héritage linéaire, où l'opérande sur le côté gauche est plus spécifique que celle sur le côté droit [Aracic, 2006].

La composition mixin peut être obtenue en faisant passer un mixin comme paramètre superclasse à une autre classe CaesarJ.

3.2.5 Déploiement d'aspect

Après la définition de Weavelet et que les deux facettes fournie et attendue sont composées, la Weavelet doit encore être instanciée et déployée afin d'activer ses points de coupure et ses advices. En effet, et contrairement à AspectJ, les aspects en CaesarJ peuvent être instanciés explicitement, cela correspond à l'instanciation de plusieurs Weavelets. De cette façon, il est possible de créer des instances de plusieurs aspects dans la même application et de les gérer comme des objets ayant des responsabilités particulières.

En CaesarJ, le déploiement d'aspect peut être fait à la fois statiquement et dynamiquement, cela constitue une autre différence par rapport à AspectJ, où ce dernier ne supporte que le déploiement d'aspect statique.

En CaesarJ, le déploiement d'aspect statique est obtenu par le modificateur *deployed*, ce qui permet le déploiement d'aspect en moment de la compilation. Le déploiement dynamique peut être soit le déploiement local ou à base de déploiement fil. Pour se faire, un aspect doit d'abord être instancié par l'instanciation d'une Weavelet puis utiliser l'instruction de *deploy* pour préciser la portée de cet aspect.

Dans le déploiement local, le mode d'activation d'un aspect est défini par les deux des mots-clés *deploy* et *undeploy*, qui active et désactive l'aspect, respectivement.

Le mode d'activation dans le déploiement à base fil est définit par un bloc *deploy,* où l'aspect est déployé suivant un contrôle de flux à l'intérieur du bloc et qui n'a aucune influence sur les exécutions simultanées.

La figure 10 illustre la structure générale du composant CaesarJ, ainsi que la relation entre ses modules et les classes de base d'une application particulière.

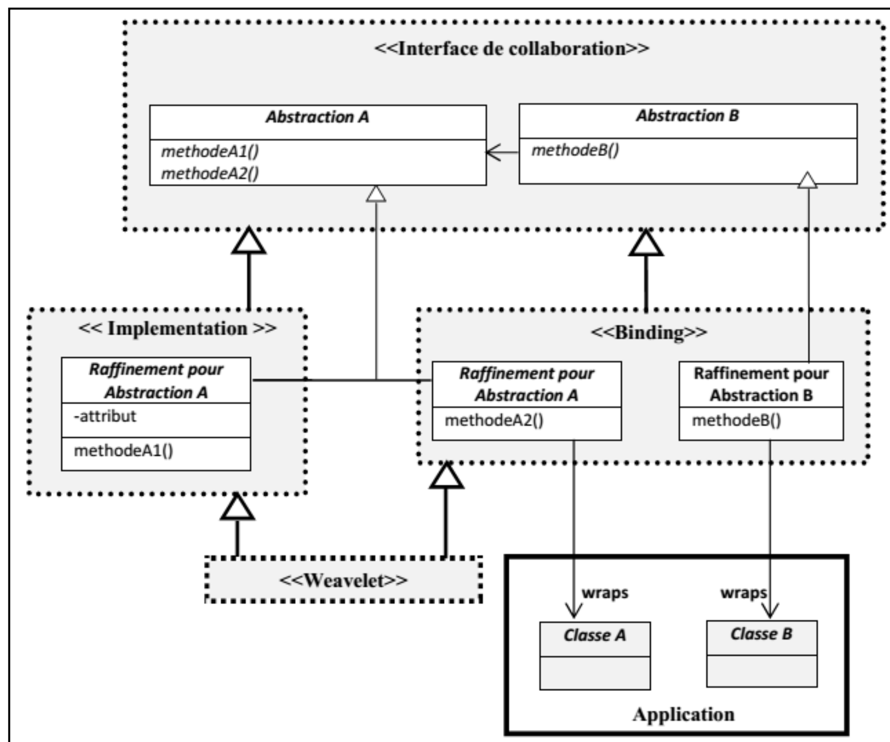

Figure 10 Structure générale du composant CaesarJ

Il est intéressant de noter que la présence de toutes les parties n'est pas toujours nécessaire, à part la partie binding car c'est elle qui assure la liaison entre la partie la plus concrète de l'application et la plus abstraite qui est l'interface de collaboration.

3.3 Le concept de tissage

Le tissage en CaesarJ, est comme la plupart des approches supportant l'activation dynamique des aspects, tels que JAC [Pawlak, 2001], JBoss AOP [JBoss AOP, Web] et AspectWerkz [AspectWerkz, Web], doit être précédé d'une phase d'adaptation. Cette dernière a pour rôle de préparer l'application en vue de l'exécution des aspects.

Il y a deux possibilités pour la préparation : soit d'insérer des crochets qui se joignent à tous les points d'une classe chargée ou de limiter la zone d'activation à un ensemble fixe de points de jointure connus. Alors que la première option provoque une surcharge significative sur les performances, la seconde option nécessite une connaissance initiale sur les aspects, qui peuvent êtres activés [Aracic, 2006].

4 Bilan

Même si l'objectif des trois approches est le même, la façon qu'ils suivent pour l'atteindre diffère d'une approche à l'autre.

AspectJ et JBoss AOP ont montré une certaine ressemblance dans la mise en œuvre des concepts de base de la programmation orientée aspect, avec une légère différence au niveau de la syntaxe et le principe d'extension du langage Java.

CaesarJ, même s'il utilise les points de coupure et les advices d'AspectJ, présente une démarche totalement différente des autres. Il propose de nouvelles notions et concepts et améliore d'autres inspirés des anciens paradigmes tels que le paradigme objet.

Le tableau suivant résume quelques notions présentées auparavant :

Tableau 3 Utilisation des concepts aspect par les trois approches

	AspectJ	**JBoss AOP**	**CaesarJ**
Langage de base	Java	Java	Java
Extension du langage de base	Oui[1]	Non	Oui
Utilisation des points de coupure	Oui	Oui	Oui
Composant comportant les préoccupations	Aspect[1]	Classe java	Cclass
Modules réutilisables	Aspect abstrait	Ceux de java	Interface de collaboration et CaesarJ implémentation
Mécanisme d'introduction	Déclaration Inter-type	Mécanisme Mix-in	Wrappers
Moments du tissage	Compilation Chargement[2]	Compilation Chargement Exécution	Compilation Chargement

Après avoir présenté les différents concepts d'AspectJ, JBoss AOP et CaesarJ nous pouvons montrer la méthode suivie par chacun d'entre eux afin de définir et exécuter les différentes préoccupations.

Le schéma suivant illustre d'une façon générale la procédure de travail des trois approches. Noter que suivant l'IDE supporté, le tissage peut prendre place dans

[1] AspectJ 5 permet de travailler avec le langage Java pur.
[2] Ce moment de tissage est spécifique à AspectJ 5.

différents moment : au moment de la compilation, avant/après le chargement de classes ou au moment de l'exécution (figure 11).

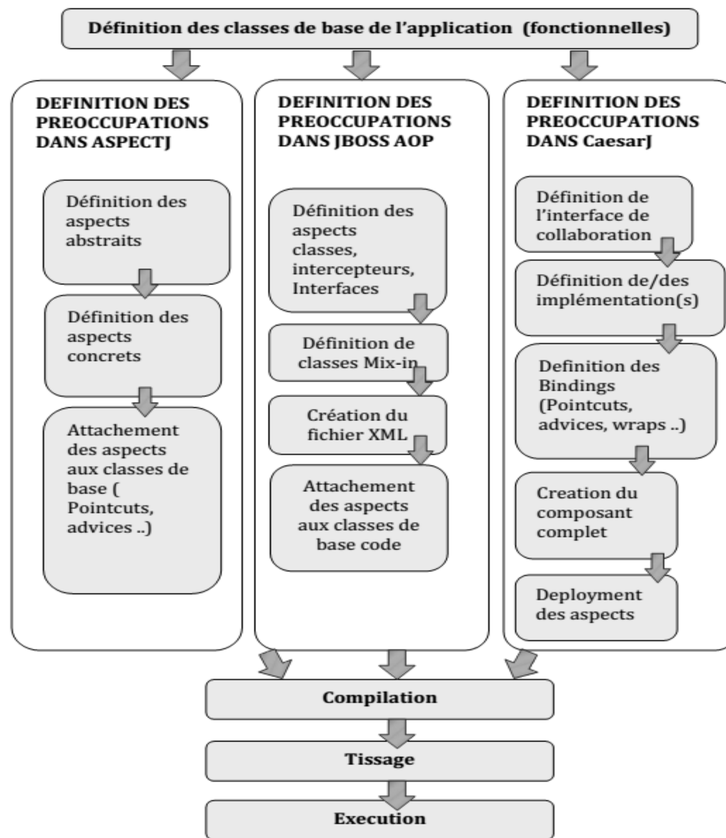

Figure 11 Procédure générale de travail des trois approches [Debboub, 2013]

5 Conclusion

Dans ce chapitre nous avons présenté les concepts, et les méthodes de travail suivis par les trois approches : AspectJ, JBoss AOP et CaesarJ, visant à garantir une meilleure séparation des préoccupations.

AspectJ, vu qu'il présente le premier projet émergé de la programmation orientée aspect, réalise avec une façon concrète touts ses concepts et notions comme : les aspects, les points de coupure et jointure, le mécanisme d'introduction et le tissage.

A la différence d'AspectJ, JBoss AOP n'étend pas le langage Java et réalise les notions de la programmation orientée aspect, en utilisant des fichiers XML ou des annotations. Cependant, et même si la syntaxe de JBoss AOP et le choix de rester en Java pur peuvent le distinguer d'AspectJ, on trouve beaucoup de ressemblances entre les deux approches.

CaesarJ, résout le problème de séparation des préoccupations en utilisant des concepts et mécanismes qui lui sont propre. Il est parfois considéré comme une approche qui unifie la programmation orientée aspect et celle des composants.

La compréhension et la mise en œuvre des différents concepts et mécanismes présentés dans ce chapitre peuvent être cernées plus facilement en lisant le chapitre suivant, qui discute sur de la réalisation des patrons de conception de Gang of four

[Gamma, 1995] avec les trois approches.

CHAPITRE 3 : REALISATION DES PATRONS DE CONCEPTION AVEC L'APPROCHE ORIENTEE ASPECT

1 LES PATRONS DE CONCEPTION DU GANG OF FOUR (GOF)

1.1. PRESENTATION DES PATRONS DE CONCEPTION DU GOF

1.2 APPLICATION ET LIMITES DES PATRONS DE CONCEPTION PAR OBJETS

2 APPROCHE ADOPTÉE DANS L'IMPLÉMENTATION ORIENTÉE ASPECT DES PATRONS

3 UTILISATION DES ASPECTS DANS L'IMPLÉMENTATION DES PATRONS DE CONCEPTION DU GOF

3.1 LA NOTION DE ROLE

3.2 EXEMPLES DE REALISATION DES PATRONS DE CONCEPTION AVEC LES APPROCHES ASPECT

3.3 APPLICATION AUX AUTRES PATRONS DU GOF

4 CONCLUSION

Résumé :

Un patron de conception désigne une solution générale à problème de conception qui a prouvé sa vigueur et qui a mérité d'être préservée pour être réutilisée dans différentes situations.

Le catalogue du GoF couvre 23 patrons de conception qui sont les plus connus et les plus populaires, ils sont la base de nombreuses recherches dans différents domaines du génie logiciel.

La réalisation des patrons de conception avec la programmation objet a engendré quelques problèmes de conception comme la dispersion du code des patrons dans les différentes classes de l'application ce qui gêne leur évolution et maintenance.

Dans le contexte de la programmation orientée aspect l'utilisation des patrons de conception comme des benchmarks hypothétiques, permet non seulement d'exposer et exploiter les apports du paradigme aspect, mais aussi de comparer les caractéristiques et les puissances des approches aspect à travers la résolution des différents problèmes de conception.

Dans le présent chapitre nous allons nous intéresser à la réalisation des patrons de conception du GoF avec les trois approches AspectJ, JBoss AOP et CaesarJ. Afin d'être plus explicite nous présentons quelques exemples illustratifs de ces implémentations.

1 Les patrons de conception du Gang of Four (GoF)

1.1. Présentation des patrons de conception du GoF

1.1.1 Origine et histoire

Développer au premier abord pour l'architecture et l'urbanisme, le concept de « *Design Pattern* » a était imaginé par l'architecte Christopher Alexander à la fin des années 70, celui-ci proposa de résoudre les problèmes récurrents en constructions par des solutions dites classiques. Il en établi un langage de 253 patrons couvrant l'ensemble de ces problèmes.

Inspirés par les écrits d'Alexandre, Ken Beck et Ward Cunningham ont mené les premiers travaux appliqués à l'informatique dans le domaine. Ils ont conçu un ensemble de modèles de développement d'IHM élégantes en Smalltalk qu'ils ont présenté en 1987 lors d'une conférence à Orlando.

En1995, est né du fruit de la collaboration des chercheurs Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides, surnommés affectueusement « Gang of Four », l'ouvrage phare intitul**é : «** *Design Patterns : Elements of Reusable Object-Oriented Software* » [Gamma, 1995], présentant les 23 « Design

Patterns » qui font aujourd'hui office de référence dans le monde de l'informatique.

1.1.2 Qu'est-ce qu'un patron de conception

Christopher Alexander dit que : «Chaque patron décrit un problème qui se produit encore et encore dans notre environnement, et ensuite décrit le cœur d'une solution à ce problème de manière à ce qu'on puisse utiliser cette solution plus d'un million de fois, sans jamais le faire deux fois exactement de la même manière.» [Christopher, 1977].

Bien qu'Alexander ait fait état de modèles en matière d'édifices et de villes, ce qu'il propose s'applique aux modèles de conception orienté-objet. Ces derniers s'expriment en termes d'objets et d'interfaces, plutôt qu'en termes de murs et de portes, mais au cœur des deux types de modèles, réside la solution d'un problème pour un contexte spécifique.

Les patrons de conception décrivent des organisations de classes objets, sous une forme abstraite. Ils ne s'attachent pas aux détails du problème, au même titre qu'ils sont basés sur des expériences passées et des solutions qui ont prouvé leur efficacité.

En général un modèle possède quatre éléments essentiels :

- **Le nom** de modèle est un moyen de décrire en quelques mots un problème de conception, ses solutions, et leurs conséquences. Donner un nom à un modèle accroît immédiatement le vocabulaire de conception. Cela permet de travailler à un degré d'abstraction plus élevé. Disposer d'un vocabulaire pour les modèles, permet de les citer aux collègues, dans une documentation par exemple. Le nom facilite par la même occasion la recherche de solutions, ainsi que leur communication à des tiers, nanties de leurs variantes de compromis.

- **Le problème** décrit les situations où les modèles s'appliquent. Il expose le sujet à traiter et son contexte. Il peut s'agir de problèmes spécifiques de conception comme, la représentation d'algorithmes en termes d'objets. Il peut décrire des structures de classes ou d'objets, typiques d'une conception immutable. Le problème comportera parfois une liste de conditions à satisfaire pour que le modèle s'applique valablement.

- **La solution** décrit les éléments qui constituent la conception, les relations entre eux et leur part dans la solution et la coopération. La solution ne décrit pas un modèle précis, ni une implémentation, puisqu'un modèle est une sorte de patron qui s'applique dans diverses situations. Le modèle fournit plutôt la description générique d'un problème de conception, et indique comment un agencement d'éléments (dans notre cas, des classes, des objets, …) peut le résoudre.

- **Les conséquences** sont les effets résultants de la mise en œuvre du modèle et les variantes de compromis que celle-ci entraîne. Bien que ces conséquences soient rarement évoquées lors de la description des choix de conception, elles sont déterminantes pour l'évaluation des alternatives de conception et pour l'appréciation des avantages et des inconvénients de l'application du modèle.
  Dans le cas des logiciels, les conséquences tiennent fréquemment de compromis de volumes d'encombrement mémoire et de rapidité d'exécution. Elles peuvent concernées également des aspects du langage et du codage. Etant donné que la réutilisation est souvent le facteur déterminant d'une

conception orientée-objet, les conséquences d'un modèle incluent son impact sur la flexibilité d'un système, son extensibilité, ou sa portabilité. Faire la liste explicite de ces conséquences permet de les comprendre et de les évaluer. [Gamma, 1995]

Il ne nous reste plus maintenant qu'à citer quelques avantages d'utilisation des patrons de conception, alors voici une liste non-exhaustive de ces avantages :

- Ils fournissent un vocabulaire partagé pour communiquer à propos des conceptions logicielles,
- Ils réduisent la complexité,
- Ils capitalisent l'expérience, car ils constituent une base d'expérience pour la construction de logiciels sous forme de briques de base de conception à partir desquels des conceptions plus complexes peuvent être élaborées,
- Ils forment un catalogue de solutions,
- Ils permettent l'élaboration de constructions logicielles de meilleure qualité par un niveau d'abstraction plus élevé,
- Ils permettent d'augmenter la productivité,
- Ils facilitent la réutilisation et la maintenance du code.

1.1.3 Les patrons de conception du Gang of four

Les avantages indéniables qu'apportent les patrons de conception, ont poussé les chercheurs à travailler intensément dans ce domaine, plusieurs catalogues ont été rédigés.

La contribution principale à ce domaine a été la publication en 1995 du livre Design Patterns: Elements of Reusable Object Oriented Software par la désormais célèbre « bande des quatre »: Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides [Gamma, 1995]. Ce livre, qui est l'un des plus vendus de toute l'histoire de l'informatique, a constitué le premier catalogue de patrons de conception applicable à une grande variété d'applications. Dans ce premier catalogue, la plupart des patrons de conception se focalisent sur un aspect particulier de la conception de logiciel, à savoir sa flexibilité. Il s'agit donc pour l'essentiel de solutions de conception visant à rendre le logiciel conçu plus évolutif, que ce soit dynamiquement ou tout simplement pour faciliter de futures opérations de maintenance. L'idée clé des patrons de conception du GoF est de permettre de faire varier certains aspects de la structure du système indépendamment des autres aspects, afin de rendre le système robuste à certains types de changements qui peuvent alors être effectués sans ré-conception.

1.1.3.1 Formalisme du GoF

Les patrons de conception peuvent être présentés soit sous forme narrative, soit sous forme structurée, Cette dernière peut améliorer la lisibilité du modèle, et facilite la communication entre les concepteurs de logiciels.

Le format choisi par GoF comporte les éléments suivants :

- **Nom.** Le nom du patron est choisi de manière à véhiculer succinctement l'essence du patron. Un nom bien choisi est absolument vital, car il devient ensuite un élément clé du vocabulaire de conception.
- **Intention.** Courte description de :
  - ce que fait le patron de conception,
  - sa raison d'être ou son but,
  - cas ou problème particulier de conception concerné.
- **Alias.** Autres noms connus pour le patron.

- **Motivation.** La motivation ou le contexte dans lequel ce patron s'applique. Ceci peut être concrétisé par un scénario qui illustre un problème de conception et comment une structure de classe et d'objets aide à résoudre le problème. Le scénario a pour objectif d'aider à comprendre les descriptions plus abstraites qui suivront.
- **Indications d'utilisation.** Permet de répondre aux questions suivantes :
  - Quels sont les cas qui justifient l'utilisation du patron ?
  - Quelles situations de conception peuvent tirer avantage du patron ?
  - Comment reconnaître ces situations ?
- **Structure.** Une description structurelle (en termes de classes et de leurs relations exprimées en UML) d'un exemple du type de solution proposée par ce patron.
- **Constituants.** Explication des différentes classes/objets qui interviennent dans la structure du patron, avec leurs responsabilités
- **Collaborations.** Comment les participants collaborent pour mener à bien leurs responsabilités.
- **Conséquences.** Courte description des :
  - compromis induits par l'utilisation du patron,
  - impacts sur l'architecture de conception,
  - gains en termes de diminution du couplage dans la solution.
- **Implémentation.** Quels pièges, astuces, ou techniques concernent l'implantation de ce patron de conception ? Y a-t-il des aspects qui sont spécifiques à un langage de programmation particulier ?
- **Exemple de code.** Des fragments de code qui illustrent comment on peut implanter ce patron de conception dans des langages de programmation à objets classiques.
- **Utilisations remarquables.** Exemples documentés d'application de ce patron de conception dans des systèmes réels. L'effort de documentation d'une solution de conception n'est en effet valable que si cette solution peut être appliquée de manière récurrente pour résoudre le problème dans un contexte donné. Il est donc d'usage de considérer qu'une solution de conception n'est en réalité un patron de conception que si elle a été identifiée dans au moins trois applications totalement différentes.
- **Modèles apparentés.** Quels patrons de conception ont un rapport avec celui qu'on est en train de décrire ? Quelles sont les différences importantes ? Avec quels autres patrons celui-ci peut-il être utilisé en synergie ?

1.1.3.2 Classification GoF

Selon GoF, les patrons de conception se divisent en trois grandes classes : modèles créateurs, modèles structuraux et modèles de comportement. Dans ce qui suit nous allons faire une petite présentation pour chacune d'entre elles.

- **Modèles créateurs**
  Les modèles créateurs sont des modèles de conception spécialisés dans la création, ils expriment le processus d'instanciation, et permettent de rendre le système indépendant de la façon dont ses objets ont était créés, combinés et concrétisés.
  Un modèle créateur de classe utilise l'héritage pour instancier des variantes de classe, alors qu'un modèle créateur d'objet délègue l'instanciation à un autre objet.
  On trouve dans cette classe les modèles suivants :

Abstract Factory (Fabrique abstraite)**:** elle sert à encapsuler un groupe de fabriques ayant une thématique commune. L'utilisateur ne se préoccupe pas de savoir quelle fabrique a servi à donner l'objet.
Builder (Monteur) **:** il dissocie la construction d'un objet complexe de sa présentation, de sorte que le même processus de construction permette des représentations différentes.
FactoryMethod (Fabrique) **:** définit une interface pour la création d'un objet, mais en laissant à des sous-classes le choix des classes à instancier. Elle permet à une classe de déléguer l'instanciation à des sous-classes.
Prototype**:** il est utilisé lorsque la création d'une instance est complexe ou consommatrice en temps. Pour obtenir une nouvelle instance on copie une instance déjà existante puis on modifie la copie de façon appropriée.
Singleton **:** son but est de restreindre l'instanciation d'une classe à un seul objet. Ce patron est à utiliser lorsqu'il ne doit y avoir qu'une seule instance de la classe.

- **Modèles structuraux**

Les modèles de conception structuraux étudient la façon de composer des classes et des objets pour réaliser des structures plus importantes. Les modèles de classes structuraux utilisent l'héritage pour composer interfaces et implémentations. Par contre les modèles objets structuraux décrivent les moyens de composer des objets pour réaliser de nouvelles fonctionnalités. La souplesse supplémentaire apportée par composition d'objets, résulte de la possibilité de modifier la composition à l'exécution, ce qui est impossible avec la composition statique de classe.
Dans cette classe on trouve les modèles suivants :

Adapter (adaptateur) **:** Il est utilisé pour modifier l'interface d'un objet afin de le rendre utilisable avec une interface qu'il ne possède pas.
Bridge (Pont): il permet de découpler l'interface d'une classe et son implémentation et ainsi de les faire varier séparément.
Composite**:** il sert à représenter une hiérarchie d'objets. Les objets composites sont constitués de plusieurs objets similaires.
Decorator (Décorateur)**:** il permet d'attacher dynamiquement de nouvelles responsabilités à un objet.
Facade (Façade)**:**elle a pour but de cacher une conception et une interface complexe en fournissant une interface simple.
Flyweight (Poids-mouche)**:** il sert lorsque de nombreux petits objets doivent être manipulés alors qu'ils ne différent que par quelques paramètres. Seul un petit nombre d'objets est alors instancié, leur différenciation se fait par le passage de paramètres dans les méthodes.
Proxy**:** défini une classe qui se substitue à une autre. il implémente alors la même interface que la classe à laquelle il se substitue. Son utilisation ajoute une indirection à l'utilisation de la classe substituée. Il peut jouer un rôle d'ambassadeur (remote proxy) ou de contrôle d'accès (protection proxy).

- **Modèles de comportement**

Les modèles de comportement traitent des algorithmes et de l'affectation des responsabilités entre les objets. Ils ne décrivent pas seulement, des modèles d'objets ou de classes, mais également des modèles de communication entre ceux-ci. Les modèles de classes de comportement utilisent l'héritage pour répartir les comportements entre les classes, les modèles d'objets de comportement utilisent la composition des objets pour décrire comment un groupe d'objet se constitue pour accomplir une tâche qu'aucun d'entre eux n'aurait pu réaliser seul, ou de s'occuper de l'encapsulation de

comportement à l'intérieur d'un objet, et de la délégation de requêtes à ce dernier.
Dans cette classe on trouve les modèles suivants :

***Chain of responsability (Chaîne de responsabilité)*:** elle évite le couplage de l'émetteur d'une requête à ses récepteurs en donnant à plus d'un objet la possibilité d'entreprendre la requête. Chaîner les objets récepteurs et faite passer la requête tout au long de la chaîne, jusqu'à ce qu'un objet la traite.
***Command (Commande)*:** elle encapsule l'appel d'une méthode sur un objet pour présenter une interface uniforme à l'objet qui y fera appel. Ce qui lui permet de fournir un comportement polymorphe d'exécution d'une requête aux instances d'une même classe
***Interpreter (Interpréteur)*:** employé lorsqu'il faut interpréter un langage sur une grammaire simple.
***Iterator (Itérateur)*:** il fournit un moyen d'accès séquentiel, aux éléments d'un même agrégat d'objets, sans mettre à découvert la représentation interne de celui-ci.
***Mediator (Médiateur)* :** c'est la seule classe qui a connaissance des interfaces des autres classes. Cela permet de résoudre les problèmes de complexité de communication qui peuvent survenir dans les logiciels composés d'un grand nombre de classes.
***Memento (mémento)* :** fournit le moyen de renvoyer un objet à un état précédent sans violer le principe d'encapsulation.
***Observer (observateur)* :** est utilisé pour envoyer un signal à des modules qui sont les observateurs et qui effectuent des actions en fonction des informations qui leur parviennent depuis les modules qu'ils observent.
***State (Etat)*:** permet à un objet de modifier son comportement, quand son état interne change. Tout se passera comme si l'objet changeait de classe.
***Strategy(stratégie)* :** permet de sélectionner des algorithmes à la volée en cours d'exécution selon certaines conditions.
***Template Method (Patron de méthode)*:** il permet de définir le squelette d'un algorithme par des opérations abstraites et dont le comportement concret se trouvera dans les sous-classes qui définiront ces opérations.
***Visitor (Visiteur)*:** il fait la représentation d'une opération applicable aux éléments d'une structure d'objet. Il permet de définir une nouvelle opération sans, qu'il soit nécessaire de modifier la classe des éléments sur lesquels elle agit.

Le tableau 4 présente la classification des 23 patrons de conception du GoF :

Tableau 4 Les patrons de conception du GoF

Domaine	Créateur	Structurel	Comportement
Classe	Fabrique	Adaptateur	Interprète
			Patron de méthode
Objet	Fabrique	Adaptateur	Chaîne de
	Monteur	Pont	Commande
	Prototype	Composite	Itérateur
	Singleton	Décorateur	Médiateur
		Façade	Mémento
		Poids	Observateur
		Procuratio	Etat
			Stratégie
			Visiteur

1.2 Application et limites des patrons de conception par objets

Jusqu'à présent, nous avons parlé des origines des patrons de conception, leurs avantages, ainsi que le premier et le plus important catalogue des patrons, mais il

reste à signaler qu'il existe des limites et des problèmes liés à leur utilisation. Plus particulièrement, lors de leur implémentation. Les patrons ont le défaut de disparaître dans les conceptions et les implémentations des applications qui les utilisent. Il devient ainsi difficile de retracer et d'isoler l'imitation d'un patron, afin de la maintenir et de le faire évoluer.

La mise en œuvre des patrons de conception nécessite des techniques d'implémentation appropriées, pour une meilleure utilisation de ceux ci dans le développement de systèmes de qualité. Les techniques fondées sur le paradigme Objet ne sont pas satisfaisantes. Elles se basent principalement sur le mécanisme d'héritage, sur la composition et sur la délégation explicite, qui posent plusieurs problèmes et limites, comme :

- **Code non réutilisable**. Le code d'une implémentation d'un patron est dispersé et éclaté dans l'ensemble des classes qu'il impacte. Il est non isolé, et donc difficilement réutilisable indépendamment de l'application. En effet, de manière générale, les patrons de conception permettent la réutilisation de solutions conceptuelles et non pas d'implémentation.

- **Maintenance et évolution contraignantes**. La dispersion et l'enchevêtrement du code des implémentations des patrons pénalisent la maintenance et l'évolution de celui ci, mais aussi du code relatif à l'application. En effet, étant dispersé et éclaté dans plusieurs classes de l'application, le code relatif à un patron est difficilement localisé. Il est par conséquent difficile à maintenir et à faire évoluer. Par ailleurs, l'enchevêtrement du code des imitations des patrons, avec le reste du code de l'application, résulte en un code non lisible et nuit par conséquent à la maintenance et à l'évolution de toute l'application devant respecter les contraintes d'implémentation. Comme le confirme G. Florijn [Florijn, 1997] l'implémentation d'un patron doit respecter certaines contraintes d'implémentations dictées le plus souvent par leurs spécifications. Toute modification d'une application, utilisant des patrons, doit ainsi prendre en compte et respecter toutes les contraintes de leurs implémentations.

En revanche, les mécanismes et concepts introduits dans le cadre de l'approche Aspect offrent de nouvelles alternatives. Ils permettent ainsi de réaliser les mêmes adaptations attendues par les patrons, tout en palliant aux problèmes et limites liés à l'utilisation de ces mécanismes et concepts Objet.

Il est possible, par exemple, d'établir une relation d'héritage entre deux classes sans avoir à les lier véritablement ; aucune dépendance directe n'est ainsi établie entre la super classe et la classe fille. Celles ci restent indépendantes et donc facilement réutilisables dans plusieurs autres contextes d'utilisation. De plus, les nouvelles définitions des propriétés de la super classe enrichissant la sous classe restent isolées de cette dernière classe qui reste inchangée. Aussi, grâce aux mécanismes et concepts Aspect, il est possible de simuler l'héritage multiple même si le langage de base ne le permet pas (tel que Java, par exemple) [Hachani, 2005].

Cependant, et comme nous l'avons déjà montré dans le chapitre précédent, chaque approche à sa propre façon d'introduire et de mettre en œuvre les divers concepts et mécanismes de la programmation orientée aspect.

Dans la section trois nous exposons de façon concrète ces dissemblances à travers l'implémentation des six patrons de conception de GoF : Observateur, Mémento, Adaptateur, Fabrique abstraite, Etat et Façade en utilisant AspectJ, JBoss AOP et CaesarJ. Nous présentons dans cette même section quelques remarques sur l'implémentation aspects des autres patrons de conception.

2 Approche adoptée dans l'implémentation orientée aspect des patrons

L'approche que nous avons choisie pour effectuer notre travail consiste à implémenter les 23 patrons de conception de GoF avec les trois approches : AspectJ, JBoss AOP, CaesarJ. Ensuite de faire une comparaison quantitative et qualitative.

Afin d'être équitable avec les trois approches, nous avons demandé l'aide d'un groupe de 12 étudiants de Master II Option : Ingénierie des logiciels complexe. Pour ce faire Nous avons commencé par rassembler tous les travaux qui traitent l'implémentation des patrons de conception avec les approches aspect. Ensuite nous les avons trié et gardé ceux qui nous semblaient fiables et corrects. Pour AspectJ nous avons grader deux travaux : de [Hannemann, 2002] [Hachani, 2006]. Pour CaesarJ nous avons trouvé les codes source de 8 patrons et l'explication de 5 autres [Sousa, 2008] [Braz, 2009]. Pour JBoss AOP nous avons trouvé un seul patron de conception qui est le Singleton du livre de Pawlak et al [Pawlak, 2004].

Les résultats de la phase d'implémentation étaient satisfaisants, et les patrons de conceptions ont pu être conçus avec les trois approches. Toutefois et afin de rester équitable, nous avons pris les meilleures solutions. Le choix dans cette étape reposait sur quelques bases de génie logiciel comme la réutilisation, la taille. De ce fait, nous avons grader les solutions qui utilisent plus de composants réutilisables : comme les interfaces, les aspects abstrait, les interfaces de collaboration, etc. Aussi les solutions qui réalisent les tâches avec le strict minimum de composants ou éléments.

L'utilisation des patrons de conception comme des benchmarks hypothétiques nous a permis par la suite de faire deux sortes de comparaison : quantitative et qualitative. Dans la première comparaison, nous avons utilisé les métriques structurelles et de performance. Le manque d'outils complets pour recueillir toutes les métriques nous a obligés, dans certains cas, de les collecter manuellement.

Suite à ces métriques nous avons effectué trois sous- comparaisons :

- **Une sous-comparaison basée sur des métriques objets influencées par la programmation Aspect :** les résultats de cette comparaison reflètent trois critères importants que sont : le couplage, la taille et de la cohésion des programmes.
- **Une sous-comparaison basée sur des métriques propre à la programmation orientée aspect:** les résultats de cette comparaison concernent les nouveaux concepts de la POA, et comment ils sont utilisés par AspectJ, JBoss AOP et CaesarJ.
- **Une sous-comparaison basée sur une métrique de performance :** les résultats de cette comparaison donne une idée globale sur les outils supports des trois approches.

La comparaison qualitative vise à donner des remarques sur certains aspects observés lors de l'apprentissage des approches et la mise en œuvre des 23 modèles de conception, et pour lesquels il n'existe pas des mesures quantitatives.

Le schéma global de notre étude est illustré par la figure 12.

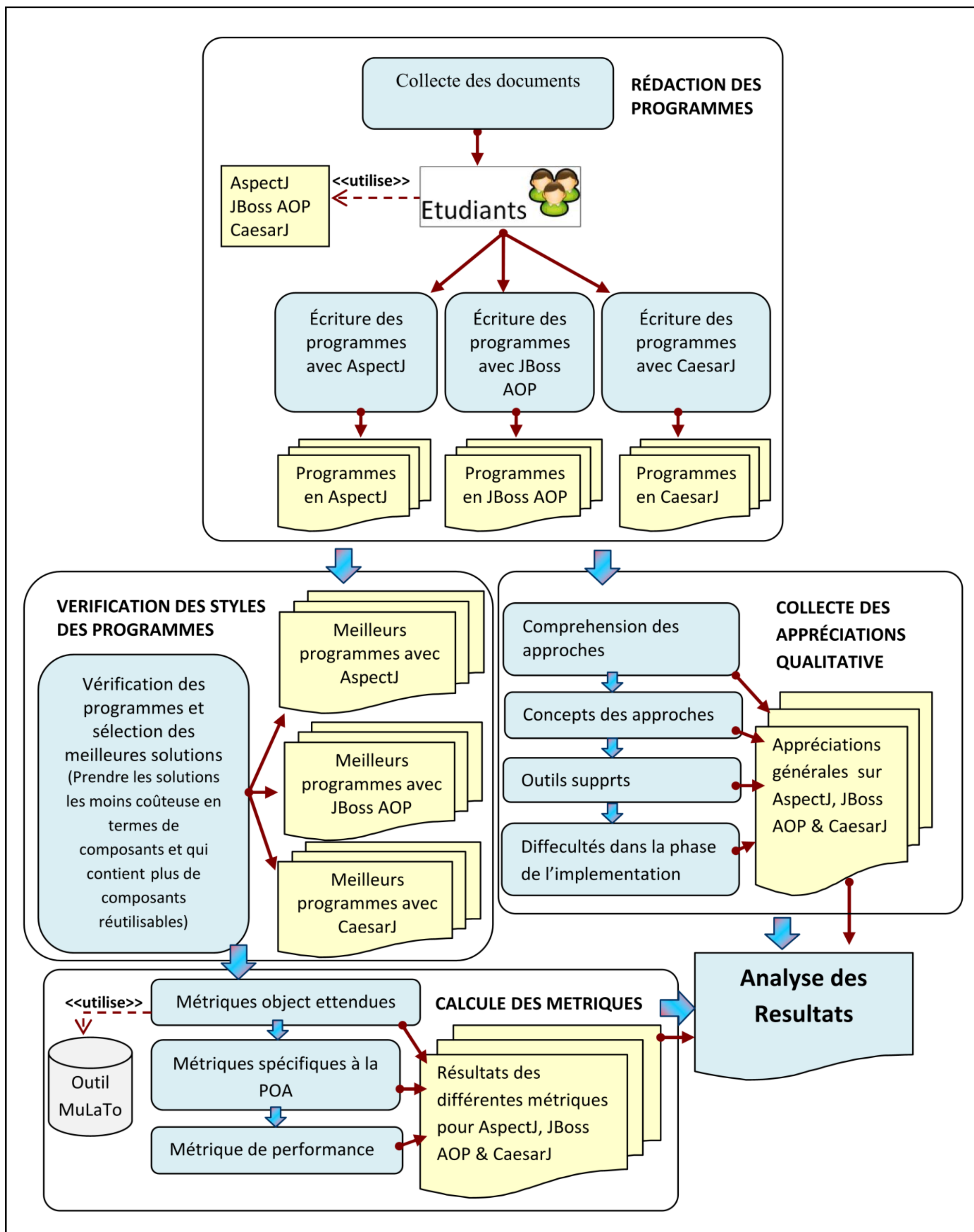

Figure 12 Schéma global de la méthode adoptée

3 Utilisation des aspects dans l'implémentation des patrons de conception du GoF

Avant d'exposer les différentes solutions aspect proposées il est intéressant de discuter du principal objectif de ces solutions. Ce qui nous oblige aussi de définir la notion de rôle et sa relation avec les patrons de conception.

3.1 La notion de rôle

Chaque patron de conception à des rôles qui définissent les comportements et les fonctionnalités de ses participants. Des exemples de ces rôles sont : composant, composé, feuille dans le modèle composite, sujet et observateur dans le modèle observateur. Ces rôles peuvent être divisés en deux catégories :

- **Première catégorie**. Dans cette catégorie, les participants n'ont pas de fonctionnalités en dehors du patron de conception. Autrement dit, ils définissent uniquement les rôles du patron de conception dont ils se trouvent. Exemple : les objets qui jouent le rôle de façade, fournissent une interface unifiée pour un sous-système et ils n'ont pas un autre comportement qui leur est propre.
- **Deuxième catégorie**. Dans cette catégorie, les rôles sont affectés à des classes qui ont déjà des fonctionnalités et des responsabilités en dehors du patron de conception. Dans le modèle Observateur par exemple, les classes qui jouent les deux rôles sujet et observateur ne remplissent pas uniquement les exigences du patron de conception mais ils ont d'autres responsabilités à accomplir. Par exemple dans un contexte graphique, les sujets pourraient être des widgets. Donc le rôle sujet est un renforcement d'une classe qui existe déjà.

Les préoccupations transversales sont causées par les différents types de rôles et leur interaction avec les classes participantes [Hannemann, 2002]. Par exemple la deuxième catégorie peut conduire à un type de préoccupation où les rôles peuvent entrecouper les participants, Autrement dit, pour un rôle, il peut y avoir N classes, et une classe peut avoir plusieurs rôles. Si on prend le modèle observateur comme exemple, une classe peut jouer le rôle d'un sujet d'observation, et le rôle d'observateur (d'une autre classe), à la fois.

Alors si on revient à l'objectif des implémentations aspect des patrons de conception on peut dire qu'elles consistent en premier lieu à capturer ces rôles dans des aspects, afin de pouvoir les réutiliser et d'améliorer la modularisation du code.

Hannemann et Kiczales regroupent les 23 patrons de conception selon les caractéristiques communes des rôles, la structure des patrons, ou l'implémentation AspectJ en six groupes que sont : (1) Observateur, Médiateur, Chaine de responsabilités, Composite et Commande, (2) Singleton, Prototype, Mémento, Itérateur et Poids-mouche, (3) Adaptateur, Décorateur, Stratégie, Visiteur et Proxy, (4) Fabrique abstrait, Fabrication, Patron de méthode, Constructeur, Pont, (5) Etat et Interpréteur, (6) Façade.

3.2 Exemples de réalisation des patrons de conception avec les approches Aspect

Nous présentons dans cette section les nouvelles implémentations orientées aspect proposées pour les six patrons de conception suivant : Observateur, Mémento, Adaptateur, Fabrique abstraite, Etat et Façade. En utilisant les nouveaux concepts de la programmation orientée aspect et en se basant sur les trois langages aspect ; AspectJ, JBoss AOP et CaesarJ. Notons que les patrons de conception pris dans cette section appartiennent aux six différents groupes de Hannemann and Kiczales.

Le tableau 5 donne les intentions des patrons de conception choisis, ainsi que les exemples appliqués sur ces patrons.

Tableau 5 Exemples illustratifs

Patron de conception	**Intention**	**Exemple d'application**
Observateur	Définir une interdépendance entre objets, telle sorte que lorsqu'un objet change d'état interne, tous ceux qui en dépendent en soient notifiés et automatiquement mis à jour.	Bruce Eckel, Thinking in patterns http://www.mindviewinc.com/downloads/TIPatterns-0.9.zip
Mémento	Sans violer l'encapsulation, il capture et externalise l'état interne d'un objet de telle sorte à ce que l'objet puisse être remis ultérieurement dans cet état.	Jan Hannemann and Gregor Kiczales http://hannemann.pbwiki.com/Design+Patterns
Adaptateur	Converti l'interface d'une classe en une autre que le client souhaite. Ce patron permet aux classes de travailler ensemble, chose qu'elles n'auraient pas pu faire à cause des leurs interfaces incompatibles.	DP Java companion http://www.patterndepot.com/put/8/JavaPatterns.htm
Fabrique abstraite	Fournit une interface pour créer une famille d'objets ou des objets interdépendants sans spécifier leurs classes.	DP Java companion http://www.patterndepot.com/put/8/JavaPatterns.htm
Etat	Permet à un objet de changer son comportement quand son état interne change. L'objet donnera l'impression de changer de classe.	Jan Hannemann and Gregor Kiczales http://hannemann.pbwiki.com/Design+Patterns
Façade	Fournit une interface unifiée à un ensemble d'interfaces dans un sous-système. Facade défini une interface de haut niveau qui rend le sous-système plus simple à utiliser.	Jan Hannemann and Gregor Kiczales http://hannemann.pbwiki.com/Design+Patterns

3.2.1 Le patron de conception Observateur

3.2.1.1 Description

L'un des plus connus et plus utilisés des patrons de conception est l'observateur. Ce patron permet de mettre en place un mécanisme d'abonnement-publication d'événements entre un sujet observable et d'autres objets intéressés par les changements d'état du sujet (les observateurs).

L'observateur a un caractère systématique: il s'agit toujours de gérer une liste d'observateurs dans la classe de l'observable (ainsi que l'ajout et la suppression d'observateurs dans cette liste), puis de notifier tous les observateurs.

3.2.1.2 Indications d'utilisation

On utilise l'Observateur lorsque :

- Une abstraction à plusieurs représentations non indépendantes,
- Une modification d'un objet nécessite la mise à jour d'autres objets,
- Un objet doit prévenir d'autres objets sans pour autant les connaitre.

3.2.1.3 Solutions orientées Aspect

- **Solution AspectJ**

Suivant l'intention de ce patron on trouve que l'observateur définit deux rôles : Le *Sujet* qui génère un événement d'intérêt (en général un changement d'état), et les *observateurs* qui observent cet événement, où chaque observateur définit sa propre réaction.

La solution AspectJ du patron de conception observateur repose sur un protocole de communication qui reste le même pour toutes ses imitations, elle consiste alors à isoler la partie générale dans un aspect abstrait noté *ObserverProtocol*, et projette les principes de ce protocole sur une application particulière en utilisant des aspects concrets.

Le listing suivant présente le code d*'ObserverProtocol*

```
1    public abstract aspect ObserverProtocol {
2    protected interface Subject{}
3    protected interface Observer {}
4
5    private WeakHashMap perSubjectObservers;
6    protected List getObservers(Subject subject) { … }
7
8    public voidaddObserver(Subject subject, Observer observer) { … }
9    public voidremoveObserver(Subject subject, Observer observer) { … }
10
11   protected abstract pointcut subjectChange(Subject s);
12
13   after (Subject subject): subjectChange(subject) { … }
14   protected   abstract   void   updateObserver(Subject   subject,   Observer
     observer);
15   }
```

L'aspect abstrait *ObserverProtocol* déclare deux interfaces internes protégées (*Subject* et *Observer*) et détient l'ensemble des introductions nécessaires, pour ces deux interfaces, spécifiant respectivement le comportement minimal des classes jouant le rôle de sujet ou d'observateur. Ces deux interfaces sont définies pour des raisons de typage, et sont déclarées comme étant protégées car non utilisées par les clients des classes jouant le rôle de sujet ou d'observateur. En effet, ces deux interfaces sont utiles uniquement dans le cadre des implémentations de l'Observateur, mais pas en dehors de ce cadre.

ObserverProtocol déclare aussi tous les éléments nécessaires au fonctionnement du protocole de communication de façon générale : il s'agit du point de recouvrement (*pointcut*) abstrait *subjectChange* qui conditionne l'exécution d'un *advice* réalisant la notification aux observateurs.

La représentation UML de la solution AspectJ pour le patron de conception Observateur est donnée en figure 13.

Figure 13 Solution AspectJ du patron de conception Observateur

- **Solution JBoss AOP**

La solution JBoss AOP, ne diffère pas beaucoup de celle d'AspectJ, néanmoins JBoss AOP et différent d'AspectJ, il n'étend pas le langage Java, ce qui l'oblige à utiliser des composants purs Java comme les classes abstraites ou les interfaces pour remplacer les aspects abstraits d'AspectJ.

Nous trouverons alors dans notre exemple, deux interfaces représentants les deux rôles du patron observateur, la première contient les méthodes nécessaires pour le rôle sujet d'observation et l'autre pour le rôle observateur.

```
1    public interface Observable {
2    public void addObserver(Observer observer);
3    public void removeObserver(Observer observer);
4      public void notify();}
5    ……
6    public interface Observer {
7    public void update(boolean isOpen);
8    }
```

La première interface est totalement indépendante de toute application spécifique, par contre la méthode *update* dans l'interface Observer contient le paramètre isOpen qui indique l'ouverture et la fermeture de la classe Flower.

Afin de projeter cette logique sur une application spécifique, nous devons créer des classes qui implémentent les différentes méthodes de ces deux interfaces et les relier avec des classes d'une application particulière en utilisant pour cela le mécanisme MixIn de JBoss AOP.

Afin d'intercepter l'attribut qui indique l'ouverture et la fermeture de la classe Flower, nous utilisons un point de coupure qui saisit tout changement de cet attribut pour déclencher l'exécution d'un aspect qui force la classe jouant le rôle du sujet d'observation à notifier ses différents observateurs.

Le listing suivant présente le code de cet aspect :

```
1    import org.jboss.aop.advice.annotation.Target;
2    public class ObsreverAspect {
3    public void updateObservers(@Target Observable observer){
4    observer.notify();}}
```

La représentation UML de la solution JBoss AOP pour le patron de conception Observateur est donnée par la figure 14.

Figure 14 Solution JBoss AOP du patron de conception Observateur

- **Solution CaesarJ**

Dans le composant CaesarJ nous trouverons différentes parties qui interagissent les unes avec les autres pour réaliser le patron observateur. Le niveau le plus abstrait est représenté par une interface de collaboration qui contient les déclarations des rôles abstraits ainsi que leurs opérations.

Le code de l'interface de collaboration est présenté dans le listing suivant :

```
1    public abstract cclass ObserverProtocol {
2     public abstract cclass Subject {
3     public abstract void addObserver(Observer obs);
4     public abstract void removeObserver(Observer obs);
5     public abstract void notifyObservers();
6     public abstract Object getState();}
7
8    public abstract cclass Observer {
9     public abstract void update(Subject s);}}
```

Les trois méthodes *addObserver (), removeObserver()* et *notifyObservers()* de Subject sont abstraites, car la façon d'ajouter, de supprimer ou de notifier un observateur n'est pas nécessairement la même dans les différents contextes. Par conséquent, ces méthodes peuvent être raffinées dans une implémentation comme suit :

```
1    public abstract cclass ObsImpl2 extends ObserverProtocol {
2    public cclass Subject {
3     private HashMap observers = new HashMap();
4    public void addObserver(Observer obs){…}
5    public void removeObserver(Observer obs){…}
6    public void notifyObservers(){…}
7     public Object getState() {…} }}
```

Les méthodes abstraite *update()* dans *Observer* doivent être appliquées suivant un cas spécifique, pour cette raison elle est raffinée dans un *CJBinding* qui assure la projection du patron observateur sur les différentes classes de l'application en utilisant le mot clé ***wraps***. *CJBinding* utilise aussi un point de coupure qui intercepte les événements d'intérêts. Notre exemple contient un seul événement qui est le changement de l'attribut *isOpen* de la classe *Flower*.

La dernière étape dans la réalisation d’observateur avec CaesarJ, consiste en l’utilisation de la composition Mixin qui rend la communication entre les différents modules du composant possible, pour se faire nous devons créer une nouvelle cclass non abstraite *FlowerObserverDeplo* qui hérite des deux cclass *ObsImpl* et *ObsBinding*. Par la suite *FlowerObserverDeploy* peut être instanciée afin de pouvoir déployer le composant CaesarJ.

La représentation UML de la solution CaesarJ pour le patron de conception Observateur est donnée par la figure 15.

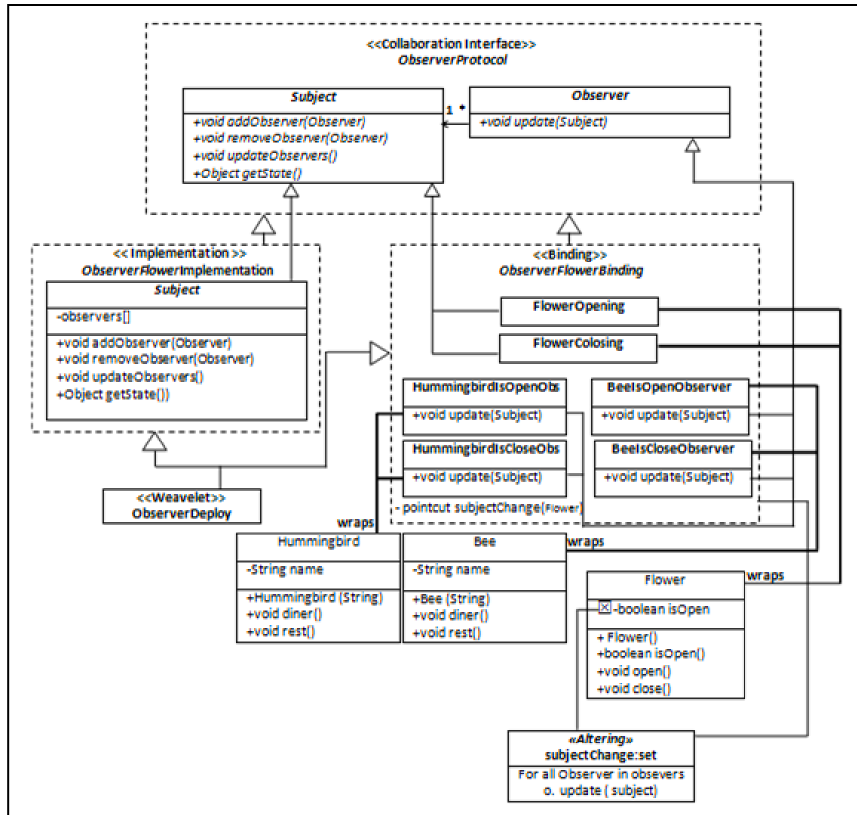

Figure 15 Solution CaesarJ du patron de conception Observateur

3.2.2 Le patron de conception Mémento

3.2.2.1 Description

Le patron de conception Mémento sert à mémoriser et transmettre l’état interne d’un objet en respectant l'encapsulation, afin de le restaurer plus tard.

3.2.2.2 Indications d’utilisation

- Une sauvegarde de tout ou partie d’un objet doit être faite,
- Isoler la conservation de l'état d'un objet.

3.2.2.3 Solutions orientées Aspect

- **Solution AspectJ**

Comme dans la solution de l’Observateur, on trouve un aspect abstrait général qui déclare une interface spécifiant le rôle joué par la classe contexte. Ainsi que deux opérations abstraites nécessaires à la réalisation de ce patron. Ces différents éléments sont par la suite concrétisés par les classes participantes dans le patron, aussi par le biais d’introductions, au niveau de l’aspect concret.

```
1    public abstract aspect MementoProtocol {
2      protected interface Originator {}
3      public abstract Memento createMementoFor(Originator o);
4      public abstract void setMemento(Originator o, Memento m);}
```

L'aspect concret de ce patron permet de projeter l'aspect abstrait sur une application particulière. Pour se faire il attribue le rôle Originator à la classe participante. Ensuite il raffine les opérations abstraites*createMementoFor*() et *setMemento(Originator o, Memento m*). La première méthode est considérée comme une Factory qui crée un objet Memento pour un Originator et la deuxième méthode, restore l'objet Memento associé à un certain objet de type Originator.

Dans le code client l'appel de ces méthodes est fait par l'utilisation de l'attribut *aspectOf*

```
1    storedState = CounterMemento.aspectOf().createMementoFor(counter);
2     …
3    CounterMemento.aspectOf().setMemento(counter, storedState
```

La figure 16 donne la représentation UML de la solution AspectJ pour le patron de conception Mémento.

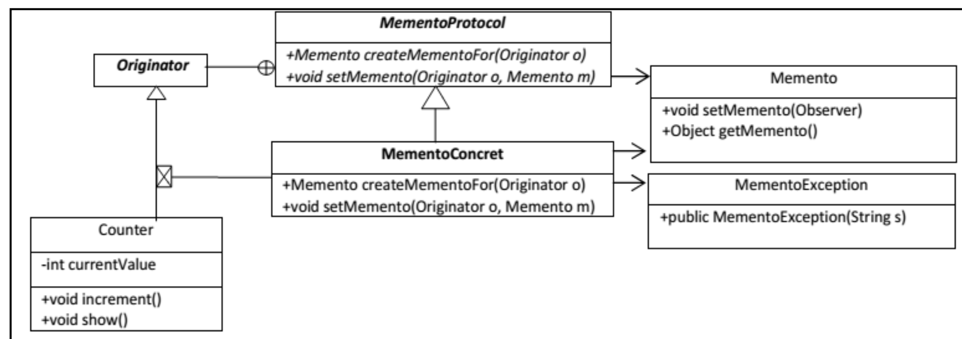

Figure 16 Solution AspectJ du patron de conception Mémento

- **Solution JBoss AOP**

Dans la solution JBoss AOP on trouve une interface représentant le rôle Originator du patron de conception Mémento.

```
1    public interface Originator {
2     publicMemento createMementoFor(Originator o);
3     publicvoid setMemento(Originator o, Memento m);}
```

Une classe implémentant cette interface et concrétisant les différentes méthodes déclarées est par la suite ajoutée à une classe d'application en utilisant le mécanisme MixIn,

```
1    public class OriginatorMixIn implements Originator{
2     private Counter counter;
3     public OriginatorMixIn(Counter counter){
4     this.counter=counter;}
5    public Memento createMementoFor(Originator o) {… }
6    public void setMemento(Originator o, Memento m) {… }}
```

La classe d'application, qui est dans cet exemple Counter, sera capable de travailler avec les différentes méthodes à partir du code Client en utilisant le casting.

```
1    Counter counter = new Counter();
2    …
3     storedState
     =((Originator)counter).createMementoFor((Originator)counter);
4    …
5     ((Originator)counter).setMemento((Originator)counter, storedState);
```

La figure 17 donne la représentation UML de la solution JBoss AOP pour le patron de conception Mémento.

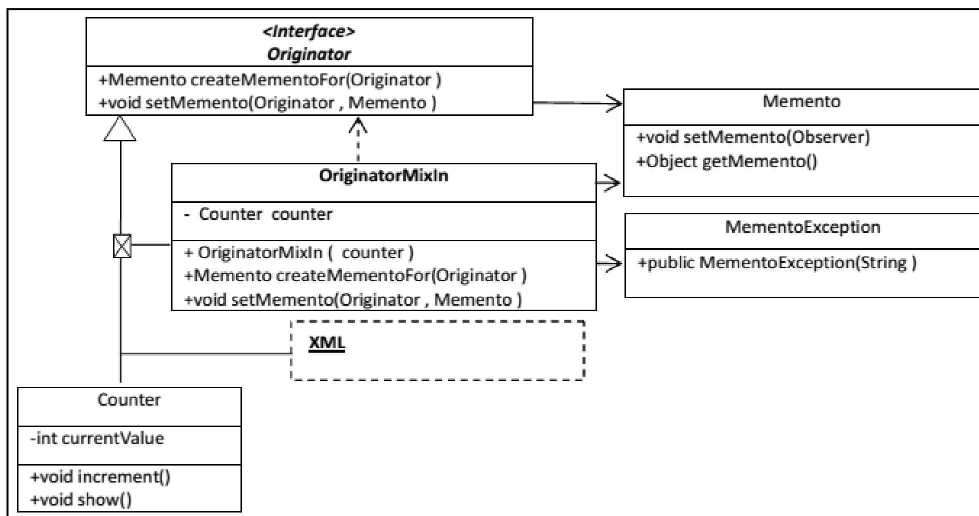

Figure 17 Solution JBoss AOP du patron de conception Mémento

- **Solution CaesarJ**

Dans la solution CaesarJ la partie générale se trouve dans l'interface de collaboration ce qui permet son utilisation dans des différentes applications.

```
1   publicabstractcclass MementoProtocole {
2     publicabstractcclass Originator {
3             publicabstract Memento createMementoFor(Originator o);

4             publicabstractvoid setMemento(Originator o, Memento m);}}
```

La projection de ce protocole sur les classes de bases est assurée par la partie CJBinding qui enveloppe une ou plusieurs classes de l'application. On remarque dans ce patron l'absence de la partie CJImplementation car les différentes méthodes du protocole dépendent de l'application.

Dans le code client on assure les relations entre la classe enveloppante et enveloppée afin de pouvoir accéder aux différentes méthodes du patron.

```
1   Counter counter = new Counter();
2   MementoBinding asp = new MementoBinding();
3   MementoProtocole.Originator O1 = asp.Originator1(counter);
4    …
5   storedState = O1.createMementoFor(O1);
6    …
7   O1.setMemento(O1, storedState);
```

La figure 18 donne la représentation UML de la solution CaesarJ pour le patron de conception Mémento.

3.2.3 Le patron de conception Adaptateur

3.2.3.1 Description

Le patron de conception Adaptateur permet de convertir l'interface d'une classe en une autre conforme à l'attente d'un client. Il permet ainsi de construire une collaboration d'instances dont les classes ont des interfaces incompatibles.

3.2.3.2 Indications d'utilisation

- Utilisation d'une classe existante dont l'interface ne correspond pas au nouveau domaine d'utilisation,
- Permettre à des classes de fonctionner et collaborer ensemble, ce qui n'aurait pas été possible autrement (à cause de leurs interfaces incompatibles).

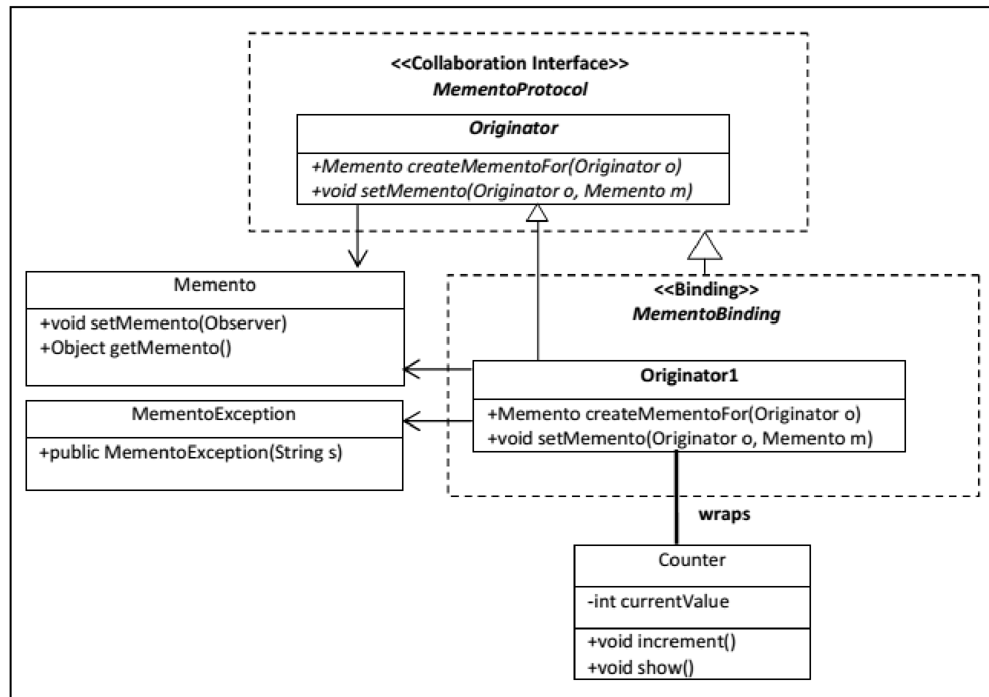

Figure 18 Solution CaesarJ du patron de conception Mémento

3.2.3.3 *Solutions orientées Aspect*

- **Solution AspectJ**

L'implémentation du patron de conception Adaptateur avec AspectJ est directe, elle se base sur le mécanisme d'introduction qui force la classe adaptée à implémenter une ou plusieurs interfaces. Par la suite, il implémente les différentes méthodes de l'interface en utilisant les déclarations inter-type.

```
1    Public aspect PrinterAdapter {
2      declare parents: Adaptee implements Target;
3      Public void Adaptee.request() {
4     …
5       specificRequest();}}
```

La figure 19 donne la représentation UML de la solution AspectJ pour le patron de conception Adaptateur.

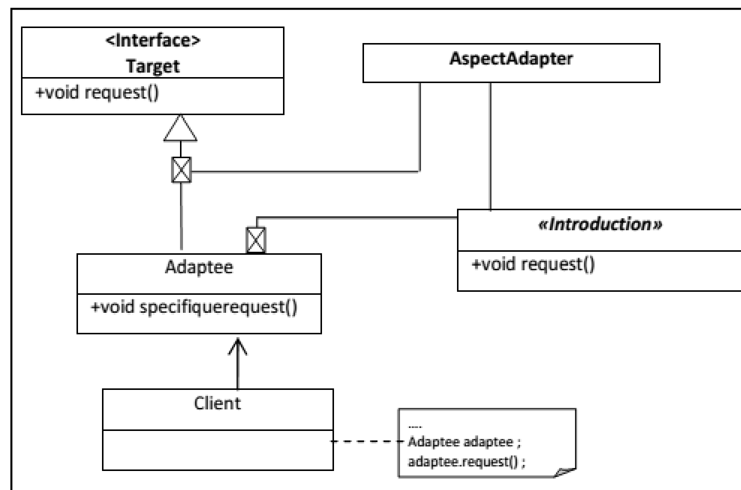

Figure 19 Solution AspectJ du patron de conception Adaptateur

- **Solution JBoss AOP**

Comme la solution AspectJ, JBoss AOP se base sur le mécanisme mix-in dans l'implémentation du patron de conception Adaptateur. Il utilise pour cela une classe mix-in.

Dans JBoss AOP, un constructeur de classe mix-in prend comme paramètre l'objet courant du point de jonction, ce paramètre permet à la classe mix-in d'accéder aux membres publics de l'objet, qui représente dans notre exemple la classe *Adaptee*, et la force à implémenter une ou plusieurs interfaces à travers un fichier XML.

```
1    public class Adapter implements Target{
2    private Adaptee adaptee;
3    public Adapter(Adaptee adaptee){
4      this.adaptee=adaptee;
5      }
6    public void request () {
7    …
8       adaptee.specificRequest();
9      }}
```

```
1    <aop>
2    <introduction class="Adapter.Adaptee">
3    <mixin>
4    <interfaces>Adapter.Target</interfaces>
5    <class>Adapter.Adapter</class>
6    <construction>new Adapter.Adapter(this)</construction>
7    </mixin>
8    </introduction>
9    </aop>
```

La figure 20 donne la représentation UML de la solution JBoss AOP pour le patron de conception Adaptateur.

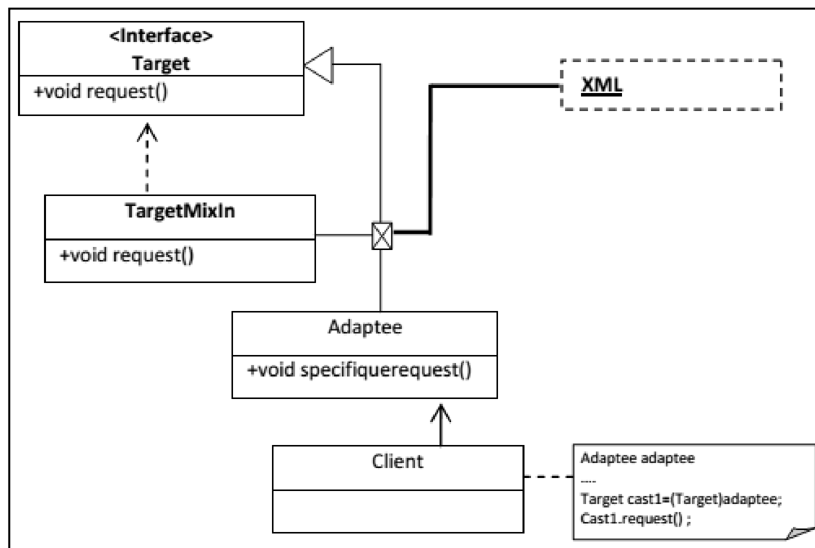

Figure 20 Solution JBoss AOP du patron de conception Adaptateur

- **Solution CaesarJ**

Dans l'implémentation du patron de conception Adaptateur, CaesarJ ne produit aucun élément réutilisable, d'ailleurs c'est le cas pour les deux autres langages orientés aspect.

Il utilise une seule classe interne qui enveloppe une classe de l'application et la force à implémenter une ou plusieurs interfaces. Il utilise le mot clé wraps pour spécifier la classe adaptée.

```
1   publiccclass PrinterAdapter{
2     publiccclass Adapter implements Target wraps Adaptee {
3             publicvoid request (){
4   …
5             wrappee.specificRequest(); }}}
```

La figure 21 donne la représentation UML de la solution CaesarJ pour le patron de conception Adaptateur.

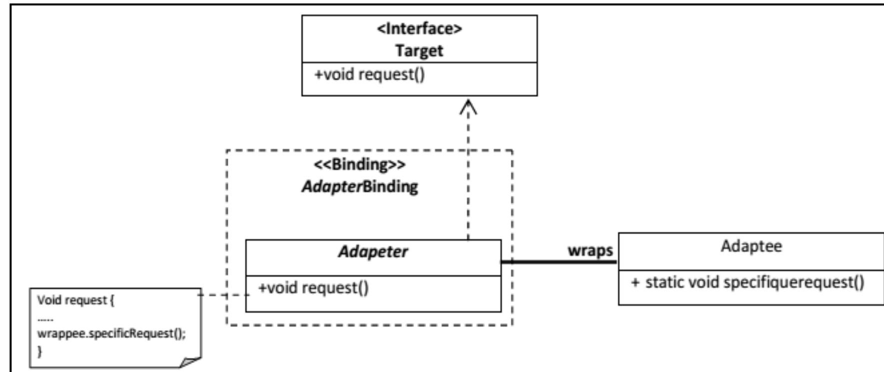

Figure 21 Solution CaesarJ du patron de conception Adaptateur

3.2.4 Le patron de conception Fabrique abstraite

3.2.4.1 Description

Le patron de conception Fabrique abstraite permet de fournir une interface pour la création de familles d'objets apparentés ou interdépendants, sans qu'il soit nécessaire de spécifier leurs classes concrètes.

3.2.4.2 Indications d'utilisation

- Un système doit être indépendant de la façon dont ses produits sont créés, combinés et représentés,
- Un système est constitué à partir d'une famille de produits, parmi plusieurs,
- On veut renforcer le caractère de communauté d'une famille de produits conçus pour être utilisés ensemble.

3.2.4.3 Solutions orientées Aspect

- **Solution AspectJ**

La solution AspectJ proposée pour Fabrique abstraite sert à remplacer les classes abstraites dans le code Java par des interfaces, ensuite d'utiliser les déclarations inter-types pour former des classes concrètes afin d'implémenter les méthodes de ces interfaces.

L'aspect qui fait cette implémentation est propre à chaque cas, c.à.d. il ne produit pas un code réutilisable. Cependant cette nouvelle solution a l'avantage de libérer des fabriques concrètes d'hériter d'une autre classe.

L'inconvénient de cette solution c'est qu'elle ne satisfait pas l'objectif du patron de conception Fabrique abstraite, mais juste elle améliore son implémentation en créant un genre d'héritage multiple, qui n'est pas supporté par le langage de base Java.

La figure 22 donne la représentation UML de la solution AspectJ pour le patron de conception Fabrique abstraite

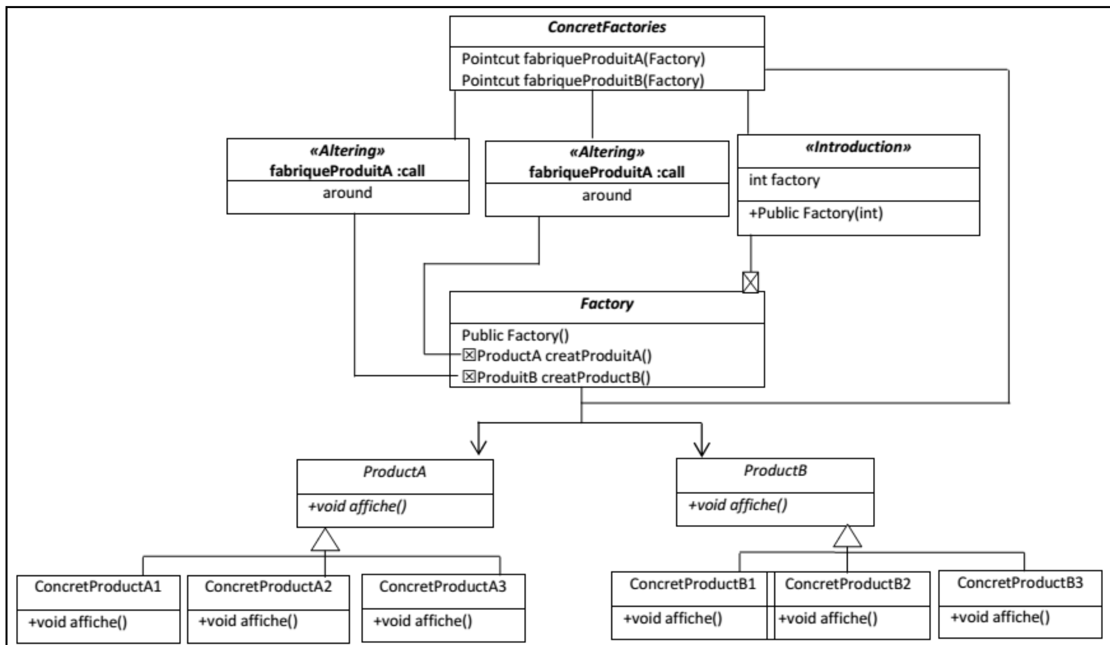

Figure 22 Solution AspectJ du patron de conception Fabrique abstraite

- **Solution JBoss AOP**

JBoss AOP résout aussi avec le mécanisme d'introduction le problème de l'héritage multiple en introduisant l'implémentation des méthodes de création des Concret Factory à travers un fichier XML qui assure les relations entre l'interface Factory et les différentes classes concrètes

La figure 23 donne la représentation UML de la solution JBoss AOP pour le patron de conception Fabrique abstraite.

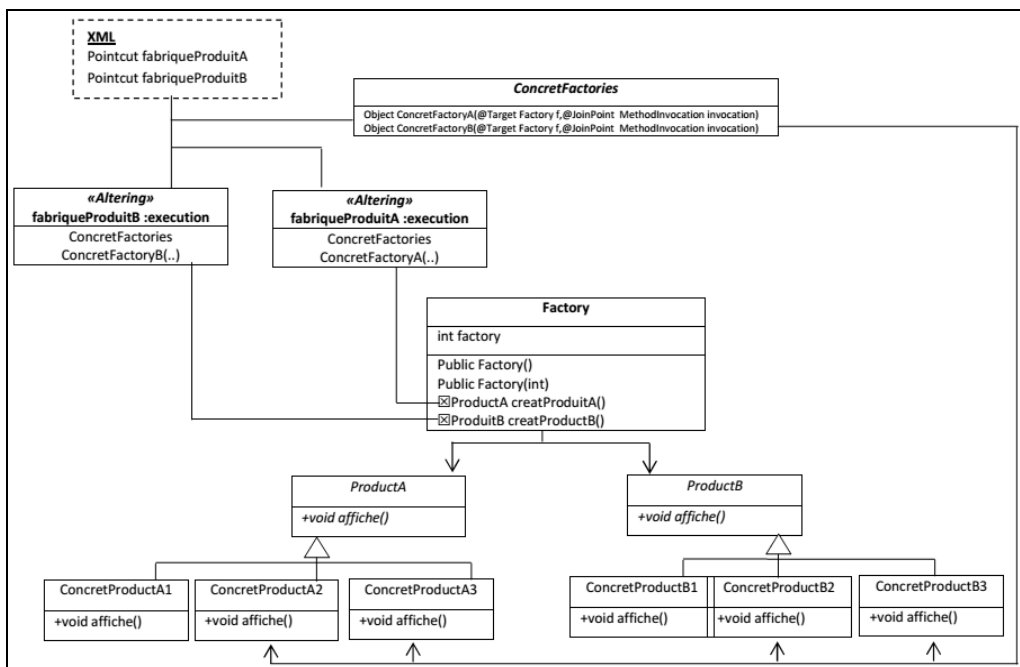

Figure 23 Solution JBoss AOP du patron de conception Fabrique abstraite

- **Solution CaesarJ**

Puisque l'objectif de Fabrique abstraite est d'empêcher les objets de différentes classes de la famille à être mélangées, les classes virtuelles et les mécanismes de

polymorphisme de CaesarJ permettent la création de familles bien définies et limitées, il est possible alors de d'implémenter directement ce patron avec CaesarJ.

La figure 24 donne la représentation UML de la solution CaesarJ pour le patron de conception Fabrique abstraite.

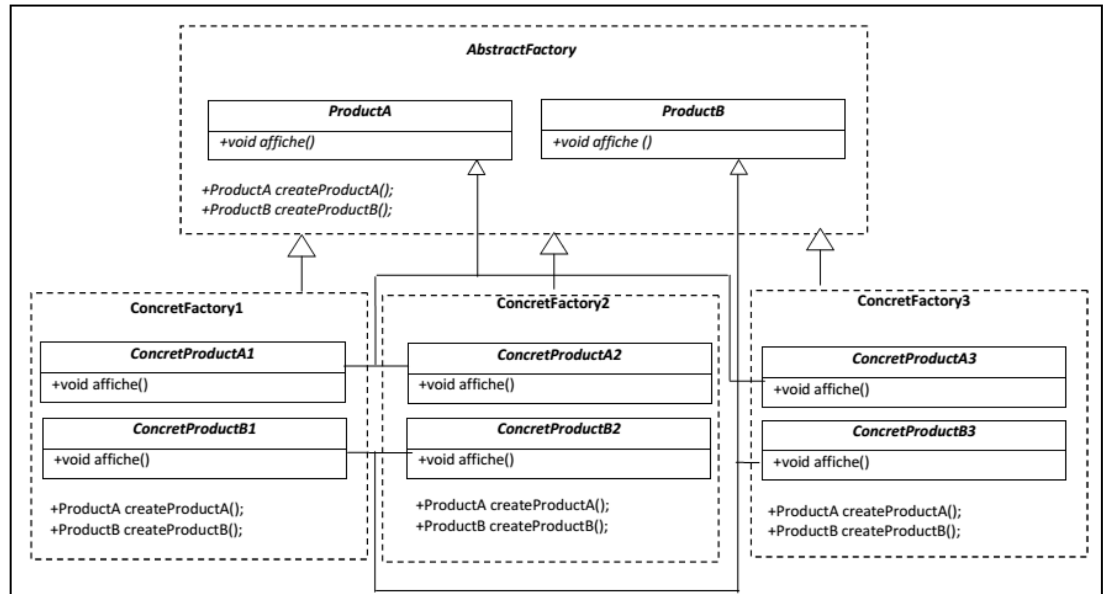

Figure 24Solution CaesarJ du patron de conception Fabrique abstraite

3.2.5 Le patron de conception Etat

3.2.5.1 *Description*

Le patron de conception Etat permet à un objet de changer de comportement quand son état interne change.

3.2.5.2 *Indications d'utilisation*

- Le comportement d'un objet dépend étroitement de son état et doit donc changer dynamiquement,
- La réalisation des opérations de l'objet est fonction de son état (même structure conditionnelle pour un ensemble d'opérations).

3.2.5.3 *Solutions orientées Aspect*

- **Solution AspectJ**

La solution AspectJ pour le patron de conception Etat ne produit aucun module réutilisable, elle se base seulement sur un aspect concret qui permet de gérer la communication entre les différentes classes participantes en utilisant des points de coupure et des consignes.

```
1    …
2    after(Queue queue): initialization(new()) && target(queue) {
3              queue.setState(empty);
4    …}
```

La figure 25 donne la représentation UML de la solution AspectJ pour le patron de conception Etat.

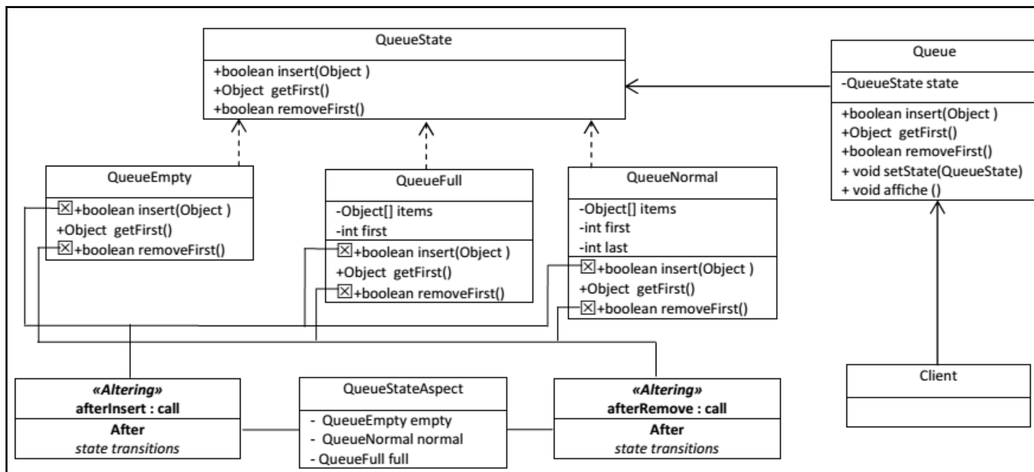

Figure 25 .Solution AspectJ du patron de conception Etat

▪ **Solution JBoss AOP**

La solution de JBoss AOP utilise deux éléments pour réaliser le patron de conception Etat. Une classe jouant le rôle d'un aspect et implémente les méthodes nécessaires de la communication entre les classes participantes (les advices) et un fichier XML regroupant tous les points de coupures déclenchant ces méthodes.

Afin de capturer les éléments des points de jointure comme les classes *called* et *target* et les arguments, le langage JBoss AOP offre des annotations spécifiques.

```
1    public void afterInitialisation(@Target Queue queue){
2      queue.setState(empty);}
3    ………
4    <bind pointcut="construction(public State.Queue->new())">
5    <after name="afterInitialisation" aspect="State.StateAspect"/>
6    </bind>
```

La figure 26 montre la représentation UML de la solution JBoss AOP pour le patron de conception Etat.

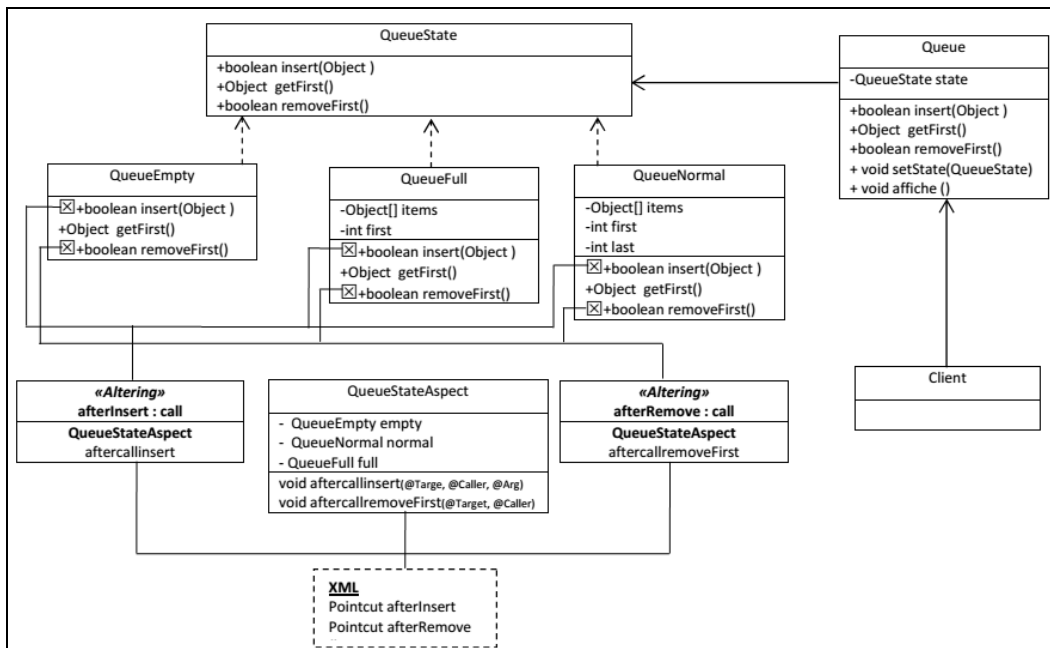

Figure 26 . Solution JBoss AOP du patron de conception Etat

▪ **Solution CaesarJ**

CaesarJ de son côté ne produit aucun élément réutilisable et se base dans son implémentation sur les points de coupures et les consignes.

La figure 27 montre la représentation UML de la solution CaesarJ pour le patron de conception Etat.

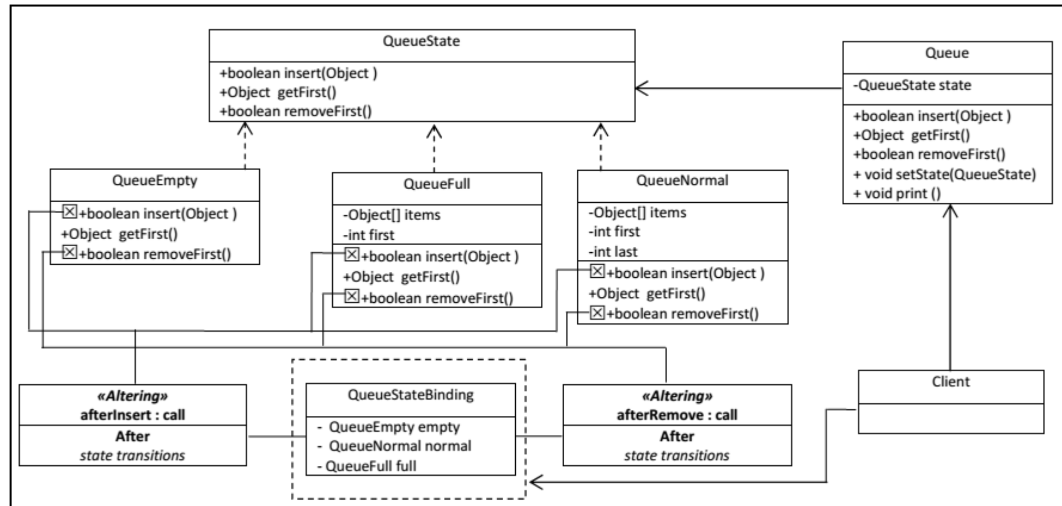

Figure 27 Solution CaesarJ du patron de conception Etat

3.2.6 Le patron de conception Façade

3.2.6.1 Description

L'objectif du patron de conception Façade est de limiter le nombre de couplages entre classes de deux sous-systèmes en fournissant une interface unifiée de haut niveau simplifiant ainsi l'usage de ce sous-système.

3.2.6.2 Indications d'utilisation

- Disposer d'une interface simple avec un package,
- Relations inter-packages trop nombreuses,
- Structuration de packages en niveaux (une façade par niveau).

3.2.6.3 Solutions orientées Aspect

Le patron de conception Façade ne pose pas de problème de séparation des préoccupations, les solutions aspect sont identiques à celle d'objet. Ce qui fait que les solutions Aspect ne font que renforcer l'encapsulation en utilisant les deux Attributs *déclare avertissement* et *déclarer erreur* qui ne permettent pas d'accéder aux méthodes enveloppées par Façade sans passer par elle. CaesarJ ne supporte pas ce genre de déclaration.

3.3 Application aux autres patrons du GoF

Les trois langages orientés aspect essayent de mettre le code de patrons de conception dans des éléments distincts et parfois réutilisables, ce qui libère les classes de l'application des rôles imposés par les patrons.

Dans cette sous section, nous allons décrire les solutions aspect suivant les six groupes proposés par Hannemann et Kiczales [Hannemann, 2002] :

3.3.1 Groupe 1(Observateur / Médiateur /Chaine de responsabilités / Composite / Commande)

Tous les patrons de conception de ce groupe introduisent des rôles supplémentaires sur différentes classes de l'application.

Avec AspectJ et comme nous l'avons vu dans l'exemple illustratif ces rôles sont réalisés avec des interfaces vides (protégées) dans l'aspect abstrait, qui contient aussi toutes les méthodes générales nécessaires à la réalisation de ce patron. Le raffinement de ces méthodes est ensuite assurer par un aspect concret afin de projeter le patron sur une application particulière. Cette logique est peut être considérée comme une bonne pratique mais dans certains cas elle n'est pas suffisante, par exemple si on prend le patron de conception observateur on trouve que l'utilisation des aspects abstraits comme un seul module réutilisable ne donne pas une meilleure conception car on doit raffiner la gestion des observateurs dans chaque aspect concret. Ce problème sera discuté dans le chapitre suivant.

JBoss AOP suit presque la même logique que celle d'AspectJ, seulement, il utilise des éléments en Java pur. L'utilisation des interfaces et des classes aspect comme le seul moyen de réutilisation gêne, en quelques sortes, les concepteurs. On peut même confondre le code de base et le code non fonctionnel dans certains cas, ce qui nécessite une lecture minutieuse du fichier XML.

CaesarJ a été créé après AspectJ, il tente donc de remédier à quelques problèmes observés dans les programmes implémentés avec AspectJ. Avec CaesarJ le programme est peut être vu comme un composant qui se base sur trois parties essentielles : une interface de collaboration qui définit des méthodes générales indépendantes de toute application particulière, une implémentation qui raffine une partie de l'interface de collaboration et qui peut être considérée à son tour comme générale (i.e. comme la partie de gestion et de sauvegarde des observateurs dans le patron de conception Observateur). La dernière partie est le Binding qui relie l'interface de collaboration à une application particulière en utilisant une des présentations offertes par la partie implémentation.

La séparation du code du patron en trois niveaux a l'avantage de minimiser la répétition des codes dans les différents modules concrets de l'application.

3.3.2 Groupe 2 (Singleton/ Prototype/ Mémento/ Itérateur /Poids-mouche)

Tous les patrons de conception de ce groupe offrent aux clients des méthodes de fabrication afin de créer des instances suivant des arguments donnés, comme la création du clone dans le patron de conception Prototype, ou encore la gestion d'une seule instance dans le patron de conception Singleton.

Dans la solution AspectJ, ces patrons (voir Mémento) ont permis de produire des aspects abstraits contenant les méthodes de fabrication ou d'ajouter cette méthode sur les classes participantes à travers le mécanisme d'introduction.

Dans JBoss AOP les méthodes Fabrication sont réalisées avec des classes MixIn, les classes de bases sont par la suite ajoutées durant leur création à ces classes, ce qui leur permet de bénéficier de ces nouvelles méthodes

CaesarJ contrairement aux patrons de conception du premier groupe ou les solutions CaesarJ génèrent des parties implémentation pour les patrons de conception dans ce groupe, la solution CaesarJ peut ne pas avoir une partie réutilisable pour certain cas d'application. C'est le cas de l'exemple de la solution CaesarJ pour le patron de conception Mémento qui ne comporte pas une partie implémentation. En effet, les deux méthodes du protocole dépendent forcément d'une application spécifique.

Par contre dans le patron de conception Poids mouche la gestion du tableau est mise dans une partie implémentation ce qui donne plus d'avantage de réutilisation pour CaesarJ par rapport aux autres langages pour ce patron.

3.3.3 Groupe 3 (Adaptateur/ Décorateur/ Stratégie/ Visiteur / Proxy)

Les nouveaux concepts introduits par la programmation par aspects peuvent faire disparaître les patrons de ce groupe.

Le mécanisme d'introduction d'AspectJ montre sa puissance dans la mise en œuvre des modèles Adaptateur et de Visiteur, où avec un simple aspect concret nous avons pu réaliser ces modèles.

JBoss AOP fournit un mécanisme d'introduction moins souple que celui d'AspectJ. Il nécessite trois éléments différents: l'interface, une classe Mix-in et un fichier XML.

La mise en œuvre des trois modèles de conception Décorateur, proxy et de stratégie avec trois approches, sont basées sur des advices et des points de coupure.

Dans le modèle de conception décorateur, le déploiement dynamique soutenu par JBoss AOP et CaesarJ permet de contrôler les moments où un décorateur est actif. Ce type de déploiement donne plus d'avantage à JBoss AOP et CaesarJ.

3.3.4 Groupe 4 (Fabrique abstrait/ Fabrique méthode/ Méthode patron/ Constructeur/ Pont)

Les modèles de conception de ce groupe sont structurellement similaires: L'héritage est utilisé pour distinguer des mises en œuvre différentes mais connexes [Hannemann, 2002].

L'approche suivie par AspectJ et JBoss AOP est de remplacer la classe abstraite avec une interface et l'utilisation respectivement des déclarations inter-types et les classes Mix-in pour composer les classes concrètes de l'interface. À cet égard, AspectJ et JBoss AOP offrent une forme limitée d'héritage multiple. Cependant, cette solution n'est pas la réponse directe aux objectifs des intentions déclarées pour ce groupe.

Fabrication et Fabrique abstraite sont directement prises en charge par les mécanismes CaesarJ. Mais avoir tout le code dans le composant CaesarJ peut empêcher la réutilisation des classes de base comme par exemple le modèle de conception Pont.

3.3.5 Groupe 5 (Etat / Interpréteur)

Les trois solutions proposées par AspectJ, JBoss AOP et CaesarJ pour les modèles de conception de ce groupe ne produisent pas de composants réutilisables. Ils sont essentiellement basés sur un élément concret qui modularise le code produit. Ce qui leur permet de gérer la communication entre les différentes classes participantes à l'aide de points de coupure et de consignes.

3.3.6 Groupe 6 (Façade)

Le modèle de conception de façade constitue un groupe distinct, parce que la structure de l'objet de ce modèle ne pose pas de problème pour la séparation des préoccupations.

4 Conclusion

Ce chapitre a présenté les patrons de conceptions en général et ceux du Gang of Four en particulier, il a mis l'accent sur les problèmes rencontrés lors de l'implémentation de ces patrons avec la programmation objet.

Par la suite, nous avons présenté les nouvelles solutions aspect en utilisant trois approches modernes que sont : AspectJ, JBoss AOP et CaesarJ, à travers six

exemples illustratifs et une discussion bien détaillée sur les 17 autres patrons de conception.

Dans le chapitre suivant, nous allons exposer notre étude comparative qui se base essentiellement sur les implémentations réalisées dans ce chapitre.

CHAPITRE 4 : ETUDE COMPARATIVE

1 ETUDE DES MÉTRIQUES

1.1 QUESTIONS SUR LA MESURE DES LOGICIELS
1.2 METRIQUES DE L'ORIENTE OBJET
1.3 METRIQUES DE L'APPROCHE ORIENTEE ASPECT
1.4 METRIQUES DE PERFORMANCE
1.5 LA QUALITE DES LOGICIELS

2 ÉTUDE COMPARATIVE

2.1 COMPARAISON QUANTITATIVE
2.2 INTERPRETATION DES RESULTATS
2.3 COMPARAISON QUALITATIVE
2.4 INTERPRETATIONS DES RESULTATS

3 SYNTHÈSE GÉNÉRALE

4 VERS UN OUTIL GÉNÉRIQUE D'EXTRACTION DES MÉTRIQUES

4.1 POURQUOI UTILISER XML ?
4.2 ASPECT CONCEPTUEL DE L'APPROCHE
4.3 EXTRACTION DES METRIQUES

5 CONCLUSION

Résumé :

Dans le monde du génie logiciel, l'apparition d'un nouveau paradigme, langage ou même méthode nécessite souvent l'étude et la comparaison avec les autres concepts afin de prendre des idées claires sur son efficacité.

L'objectif de ce chapitre et de faire une comparaison entre les trois approches : AspectJ, JBoss AOP et CaesarJ. Pour atteindre cet objectif nous commençons par présenter les métriques nécessaires dans notre comparaison quantitative, ensuite nous présentons et interprétons les résultats obtenus dans les deux comparaisons : quantitative et qualitative.

L'interprétation des résultats dévoile les dissemblances entre ces trois approches et présente les points forts et faibles de chacune d'elle.

La fin de ce chapitre pose les prémisses d'un travail futur qui concerne la réalisation d'un outil générique destiné à l'extraction des métriques logicielles.

1 Etude des métriques

1.1 Questions sur la mesure des logiciels

1.1.1 Pourquoi Mesurer ?

Galilée disait à propos de la mesure : « *Compter ce qui peut être compté, mesurer ce qui est mesurable et rendre mesurable ce qu'il ne l'est pas*.», Lord Kelvin de son coté, a dit : « *Lorsque vous pouvez mesurer ce qui vous intéresse et l'exprimer avec des nombres, vous savez quelque chose à son sujet, lorsque vous ne pouvez pas le mesurer et l'exprimer numériquement, ce que vous en savez est insatisfaisant et insuffisant.*». Cette perspective prend le nom de pythagorisme : tout est mesurable et tout doit être mesuré. C'est ainsi que grâce aux succès importants qu'obtenait la science quantitative on en vient à soutenir, à partir du Moyen âge, que la science ne pouvait être sans l'existence de la mesure [Bertrand, 2004].

Dans le génie logiciel la plupart des chercheurs soutiennent cette idée, et affirment que l'analyse quantitative du logiciel est indispensable pour l'évaluation des systèmes, et par conséquent, leur amélioration.

1.1.2 Qu'est-ce qu'on peut mesurer dans un logiciel?

Un système logiciel peut être mesuré de deux manières différentes, statiquement par des mesures effectuées sur ses attributs de conception ou son code source, ou bien dynamiquement au cours de son exécution [Munson, 2003]. Cette distinction a

donné lieu à l'apparition de deux types de métriques : des métriques structurelles concernant le coté statique du programme (i.e conception, architecture, ...), et des métriques de performance, aussi appelées "profiling", afin d'évaluer le temps d'exécution ou encore l'utilisation de la mémoire par un programme donné.

1.2 Métriques de l'orienté Objet

1.2.1 Métriques Structurelles

Les métriques structurelles appliquées dans le développement traditionnel comme le nombre de lignes de code, ou la complexité cyclomatique sont inadaptées avec les nouvelles notations de la programmation orientées objets (i.e. classes, héritage, encapsulation etc.). De ce fait, la conception orientée objet impose des métriques à la fois récentes et différentes, comme le soulignent Edward V. Berard dans "Metrics for Object Oriented Software Engineering" [Edward, 1998] et Linda Rosenberg dans "*Applying and Interpreting Object Oriented Metrics*" [Rosenberg, 1998].

Parmi les travaux les plus cités dans le domaine des métriques pour le « monde objet » on trouve ceux de Chidamber & Kemerer [Chidamber, 1994] et Lorenz & Kidd [Lorenz, 1994].

1.2.1.1 Les métriques de Chidamber et Kemerer

Ces deux chercheurs proposent des métriques, qui peuvent être qualifiées comme les métriques les plus connues et les plus étudiées dans le domaine de la qualité en génie logiciel.

Le tableau 6 présente les cinq métriques de Chidamber et Kemerer que nous avons choisis dans notre étude.

Tableau 6 Les métriques de Chidamber et Kemerer retenues dans notre étude.

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

Nous détaillerons dans ce qui suit chaque métrique en présentant, sa méthode de calcul ainsi que son impact et influence sur la qualité des programmes.

- ***Weighted Methods per Class (WMC).*** Elle permet de donner une indication sur la complexité d'une classe à partir de la complexité de ses méthodes.
  - ***Méthode de calcul.*** Soit c une classe, $m_1...m_n$ les méthodes déclarées par cette classe et $mc_1...mc_n$ les valeurs numériques

correspondantes à la complexité des méthodes alors la valeur de WMC pour la classe c peut être calculée grâce à l'équation suivante :

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

Comme précisé par Chidamber et Kemerer, si la complexité des méthodes d'une classe est considérée comme unitaire, alors le calcul de WMC pour une classe c correspond à la cardinalité de l'ensemble MDc, avec MDc l'ensemble des méthodes déclarées par c :

$$WMC(c) = |MD_c|$$

- ***Impact :***
  - Les classes qui ont un grand nombre de méthodes complexes sont en général plus spécifiques à l'application et donc moins réutilisables.
  - La complexité des méthodes d'une classe est un bon indicateur du temps à passer pour développer et maintenir une classe.

- ***Depth of Inheritance Tree*** (***DIT***)**.** Elle correspond à la valeur du plus long chemin depuis le nœud (la classe étudiée) jusqu'à la racine de l'arbre d'héritage. Elle est mesurée par le nombre de classes ancêtres.

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

$$DIT(root) = 0$$

$$DIT(c) = 1 + \max(\{DIT(\mathrm{super}) | \mathrm{super} \in PARENTS_c\})$$

  - ***Impact :***
    - Plus une classe se situe profondément dans l'arbre d'héritage, plus le nombre de méthodes et de comportements, dont elle hérite, sera grand. Ceci rendra son comportement d'autant moins prévisible.
    - Plus un arbre est profond, plus la conception est complexe.
    - Plus une classe se situe profondément dans l'arbre d'héritage, plus il est probable qu'elle réutilise des méthodes héritées.

- ***Number Of Children*** (***NOC***)**.** Il permet de calculer le nombre de sous-classes immédiates qui héritent de la classe étudiée.

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

$$NOC(c) = |SCI_c|$$

  - ***Impact :***
    - Plus le nombre d'enfants est élevé plus la réutilisation effective est importante, car l'héritage est une forme de réutilisation.
    - Plus le nombre d'enfants est élevé, plus le risque d'une utilisation inadéquate du mécanisme de l'héritage est grand. On pourra alors

se trouver face à soit une mauvaise abstraction pour la classe de base, soit un cas de mauvaise utilisation de l'héritage.

- ***Coupling Between Object*** (***CBO***)**.** Elle permet de mesurer le couplage d'une classe par rapport aux autres classes du programme. Deux classes sont dites couplées si une méthode de l'une utilise une méthode ou un attribut de l'autre.

  – ***Méthode de calcul.*** Soit c une classe, c est considérée comme couplée à une classe x à partir du moment où elle utilise au moins une des méthodes de la classe x. Soit A l'ensemble des classes du programme, Oc l'ensemble des classes o1, o2...on tel que Oc = A \ {c}, MDoi l'ensemble des méthodes déclarées par la classe oi, et MAc l'ensemble des méthodes appelées par la classe c. Il est alors possible de calculer CBO grâce à l'équation suivante :

$$CBO(c) = \left| \left\{ o_i \in O \middle| MD_{o_i} \cap MA_c \neq \Phi \right\} \right|$$

  – ***Impact :***
  - Un couplage excessif entre classes se fait au détriment de la modularité et empêche la réutilisation. Plus le CBO est faible pour une classe, plus cette classe est facilement réutilisable,
  - Pour promouvoir l'encapsulation, le couplage entre classe devrait être minimal. Plus une classe est couplée à d'autres classes, plus une modification de cette classe pourrait influencer d'autres classes. Par conséquent, la maintenabilité s'en trouverait diminuée,
  - Une mesure du couplage est importante pour déterminer la complexité du test des diverses parties d'un logiciel. L'effort de test d'une classe devrait être d'autant plus élevé que le CBO de cette classe est élevé.

- ***Lack of Cohesion in Methods*** (***LCOM***). Elle permet de mesurer le manque de cohésion d'une classe. Où la cohésion est la mesure dans laquelle les méthodes au sein d'une classe sont liées l'une à l'autre et travaillent ensemble pour fournir un comportement uni.

  – ***Méthode de calcul.*** Afin de mesurer le manque de cohésion d'une classe, on se base sur l'utilisation des variables d'instance de la classe par ses méthodes. Considérons une classe c, et MDc l'ensemble des méthodes m1,m2...mn déclarées par c. Soit Ij l'ensemble des variables d'instances utilisées par la méthode mj. Nous avons donc n ensembles de ce type, I1, I2...In.
  Soit Pc, l'ensemble des paires de méthodes de la classe c qui ne partagent pas l'utilisation d'une même variable d'instance et Qc l'ensemble des paires de méthodes de la classe c qui utilisent au moins une même variable d'instance suivante :

$$P_c = \left\{ \left(I_i, I_j \right) \middle| I_i \cap I_j = \Phi \right\} \text{ Et } Q_c = \left\{ \left(I_i, I_j \right) \middle| I_i \cap I_j \neq \Phi \right\}$$

  Alors la valeur de LCOM(c) peut être calculée via l'équation

$$LCOM(c) = |P_c| - |Q_c| \text{ Si } |P_c| > |Q_c|$$

  – ***Impact :***
  - Un LCOM élevé indique une grande disparité de la fonctionnalité de la classe qui est donc susceptible de se comporter de manière moins prévisible qu'une classe avec un LCOM faible.

1.2.1.2 *Les métriques de Lorenz et Kidd*

Lorenz et Kidd de leur côté, ont ajouté une liste de métriques qui peut être considérée comme une référence importante pour l'évaluation, et la détermination de la qualité des logiciels.

Dans notre étude nous avons choisi de travaillé avec deux métriques de taille, ces dernières sont présentée dans le tableau suivant :

Tableau 7 Les métriques de Lorenz et Kidd retenues dans notre étude

Acronymes	Noms	Descriptions
NOA	*Number of Attributes*	Le nombre d'attributs déclarés, pour une classe donnée.
NOM	*Number of Methods*	Le nombre de méthodes déclarées, pour une classe donnée.

Comme pour les métriques de Chidamber et Kemerer, nous présentons dans ce qui suit les méthodes de calcul, ainsi que l'impact de ces deux métriques sur la qualité des logiciels.

- ***Number of Attributes (NOA).*** Elle permet de calculer le nombre d'attributs déclarés, pour une classe donnée.
  - ***Méthode de calcul.*** Soit c une classe et ADc l'ensemble des attributs déclarés par la classe c. La valeur de la métrique NOA pour la classe c sera donnée par la cardinalité de l'ensemble ADc. Ce calcul est défini par l'équation suivante :

$$NOA_c = |AD_c|$$

  - ***Impact :***
    - Plus on a d'attributs dans une classe, plus l'impact pourrait être grand sur les classes enfants via l'héritage.
- ***Number of Methods (NOM).*** Elle permet de calculer le nombre de méthodes déclarées, pour une classe donnée.
  - ***Méthode de calcul.*** Soit c une classe, et MDc l'ensemble des méthodes déclarées par la classe c. La valeur de la métrique NOM pour la classe c sera donnée par la cardinalité de l'ensemble MDc. Ce calcul est défini par l'équation suivante :

$$NOM(c) = |MD_c|$$

  - ***Impact :***
    - Plus on a de méthodes dans une classe, plus l'impact pourra être grand sur les classes enfants via l'héritage.

1.2.2 Incidence de l'aspect sur les métriques de l'orienté objet

Comme nous l'avons déjà mentionné dans les chapitres précédents, L'objectif principal de la POA n'est pas de remplacer les paradigmes actuellement utilisés, mais de les améliorer en essayant de résoudre certaines problématiques. Ainsi elle peut être considérée comme une extension des autres paradigmes notamment le paradigme objet. Pour cette raison, la plupart des paramètres utilisés pour évaluer les systèmes orientés objet sont toujours applicables sur les systèmes orientés aspect avec des légères modifications.

Dans cette section, les métriques présentées auparavant sont révisées afin d'être utilisées avec les systèmes de la programmation orientée aspect.

Comme les métriques s'appliquent aux classes comme aux aspects, dans ce qui suit, nous allons utiliser le mot module/composant pour désigner ces deux unités, de même nous allons utiliser le mot opération pour définir les méthodes des classes ou les advices des aspects.

1.2.2.1 *Incidence de l'orientation aspect sur les métriques de Chidamber et Kemerer*

- **Sur la métrique WMC. S**on abréviation devient **WOM** (Weighted Operations in Module). Comme dans la programmation objet, cette métrique calcule la complexité du module suivant la complexité de ses opérations.

- **Sur la métrique DIT.** La programmation orientée aspect peut avoir une incidence sur le calcul de la métrique DIT dans la mesure où il est possible de modifier la hiérarchie d'une classe en utilisant les mécanismes d'introduction.

- **Sur la métrique NOC.** Comme dans la programmation objet, la métrique NOC compte le nombre de fils d'un module donné.
  Certains avis proposent de prendre en compte l'influence de la programmation orientée aspect sur cette métrique, car les aspects peuvent modifier la métrique NOC grâce au mécanisme d'introduction. Une classe X, orpheline ou bénéficiant de l'héritage, peut être déclarée comme sous-classe de c. Le NOC de la classe c est alors augmenté de 1.

- **Sur la métrique CBO.** Son abréviation devient **CBC** (Coupling Between Components). Comme dans la programmation objet, cette métrique compte le nombre de classes ou aspects avec lesquels le composant est couplé.
  Certains avis proposent de prendre en compte l'influence de la programmation orientée aspect sur cette métrique, car la POA peut intervenir de deux manières sur le comportement de la métrique CBO. Il lui est possible d'introduire de nouvelles méthodes ou d'utiliser des points de coupure afin d'ajouter des codes advice aux méthodes déjà déclarées. Dans les deux cas, la valeur de CBO peut croître en fonction des classes utilisées par les mécanismes introduits. Dans un cas particulier, la valeur de CBO pour une classe donnée peut être réduite. En effet, l'utilisation du code advice de type around peut remplacer complètement le corps d'une méthode. Par exemple, dans AspectJ, l'utilisation du prédicat proceed() permet au code advice de demander l'exécution de la méthode capturée, cependant, son utilisation n'est pas obligatoire, il est donc possible que le < masquage > de cette méthode réduise la valeur de CBO de la classe modifiée.

- **Sur la métrique LCOM**. Son abréviation devient **LCOO** (Lack of Cohesion in Operations). Comme dans la programmation objet, cette métrique compte le manque de cohésion dans un module donné, suivant les interactions nulles entre les pairs d'opérations (i.e. ceux qui n'utilisent pas les mêmes instances de variables).
  Certains avis proposent de prendre en compte l'influence de la programmation orientée aspect sur cette métrique, car l'impact de ses mécanismes sur la métrique LCOM est important. Tout d'abord, l'introduction de nouveaux champs et de nouvelles méthodes aura un impact direct sur la cohésion de la classe. Les méthodes introduites peuvent utiliser les champs originaux de la classe et ainsi en augmenter la cohésion ou, tout au contraire, réduire cette cohésion en n'utilisant pas ces champs. L'introduction de nouveaux champs (qui peuvent être utilisés seulement par les méthodes introduites ou les greffons) peut aussi augmenter la cohésion,

mais non la réduire telle qu'elle a été définie par Chidamber et Kemerer. Enfin, les greffons jouent un rôle important dans cet impact. Les greffons de type before et after peuvent augmenter la cohésion en associant l'utilisation de champs, introduits ou non, à des méthodes de la classe. Les greffons de type around, quant à eux, ont un effet encore plus lourd. Leur habilité à < masquer > ou à étendre une méthode leur permet de réduire ou d'augmenter à volonté la cohésion d'une classe.

1.2.2.2 *Incidence de l'orientation aspect sur les métriques de Lorenz et Kidd*

- **Sur la métrique NOA.** Comme dans la programmation objet, cette métrique compte le nombre d'attributs dans un module donné.
  Certains avis proposent de prendre en compte l'influence de la programmation orientée aspect sur cette métrique, car la POA peut modifier cette métrique, dans la mesure où un aspect est capable, via le mécanisme d'introduction, de déclarer de nouveaux attributs dans une classe donnée.

- **Sur la métrique NOM.** Son abréviation devient **NOO** (Number of operations)**.** Comme dans la programmation objet, cette métrique compte le nombre d'opérations dans un module donné.
  La modification de cette métrique se justifie par le fait que la POA peut altérer cette métrique, dans la mesure où un aspect est capable, via le mécanisme d'introduction, de déclarer de nouvelles méthodes dans une classe donnée.

1.3 Métriques de l'Approche Orientée Aspect

La programmation orienté aspect a introduit de nouvelles abstractions et techniques de composition qui n'existaient pas auparavant. Par conséquent, elles exigent de nouvelles métriques pour leurs études et évaluations. Parmi ces métriques on trouve :

- **Coupling on Intercepted Modules (CIM)**
  CIM est le nombre de modules ou des interfaces explicitement cités dans les points de coupure appartenant à un aspect donné.
  Cette mesure se concentre sur l'aspect qui intercepte les opérations des autres modules. Cependant, la CIM ne prend en compte que les modules et interfaces qui sont explicitement mentionnés dans les points de coupure. Les sous-modules, les modules mentionnés par des noms d'interfaces ou par les wildcards ne sont pas pris en compte.
  Des valeurs élevées de CIM indiquent un couplage élevé de l'aspect de l'application donnée et de la généralité de faible réutilisation.

- **Crosscutting Degree of an Aspect (CDA)**
  CDA est le nombre de modules affectés par les points de coupure ou par les introductions dans un aspect donné.
  Il s'agit d'une nouvelle métrique, spécifique aux logiciels AOP, qui doit être présentée comme l'achèvement de la métrique CIM. CIM n'estime que les modules nommé explicitement, CDA mesures tous les modules qui peuvent être touchés par un aspect. Cela donne une idée de l'impact global d'un aspect sur les autres modules.
  En outre, la différence entre CDA et CIM donne le nombre de modules qui sont touchés par un aspect sans qu'ils doivent être explicitement référencés par ce dernier, ce qui pourrait indiquer le degré de généralité d'un aspect, en termes de son indépendance de classes/aspects spécifiques.

Il est toujours souhaitable d'avoir des valeurs élevées de CDA et faibles de CIM.

- **Number of Pointcuts (NP)**

Cette métrique compte le nombre de points de coupure dans une application. On distingue deux sortes de points de coupure abstraits (NAP) et concrets (NCP).

- **Number of components introduced (NCI)**

Nous proposons une nouvelle métrique appelée NCI qui compte le nombre de composants introduits par une approche donnée lors de l'application d'un modèle de conception.

Ces composants peuvent être des simples interfaces comme celles utilisées avec JBoss AOP ou des composants de nature différente comme les aspects dans AspectJ, les fichiers XML dans JBoss AOP ou les Cclass dans CaesarJ.

Cette métrique nous permet de comprendre, et calculer combien de composants contribuent dans l'implémentation des préoccupations (i.e. les rôles) imposées par les patrons de conception.

La figure 28 présente cette métrique

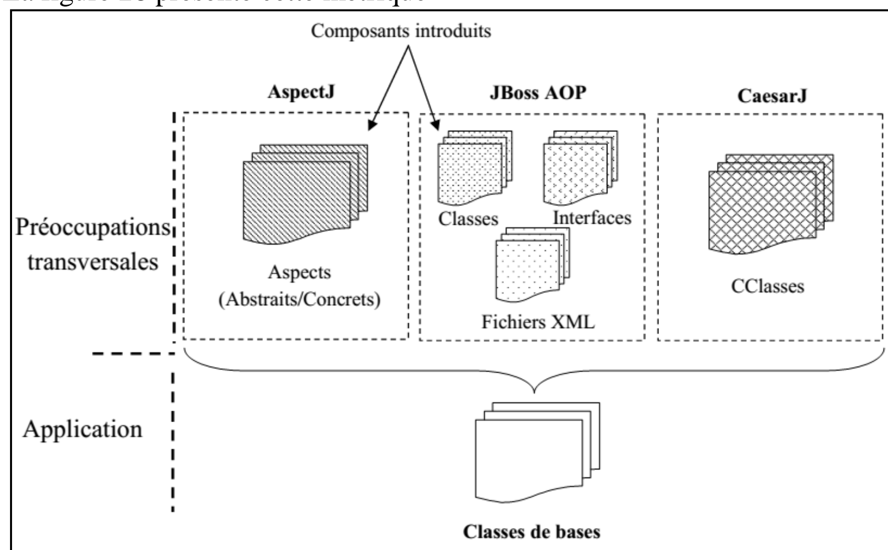

Figure 28 Métrique proposée NCI (Number of components introduced).

Évidemment, nous cherchons à trouver l'approche qui utilise moins de composants et qui satisfait la modularité dans la mise en œuvre des différents patrons de conception.

1.4 Métriques de performance

Les tests opérés sur les performances, permettent aux développeurs de mesurer, d'évaluer et de cibler les problèmes de performance de leur code, exemple de la détermination des temps d'exécution, les nombres d'appels, les occupations CPU par méthodes, la mémoire consommée, etc.

Plusieurs outils, dit de profilage existent (payants ou gratuits), et ont pour fonction d'effectuer ce genre de test. Ces outils permettent de suivre l'exécution du programme et faire des statistiques concernant les différentes mesures de performances. Toutes ces informations sont par la suite enregistrées dans un seul

fichier appelé Profile, à partir duquel on peut déduire des erreurs algorithmiques ou même les points critiques à optimiser dans une application.

Dans notre travail nous avons calculé le temps d'exécution de chaque programme mettant en œuvre des modèles de conception. Afin d'obtenir des résultats précis, nous avons répété l'exécution de ces programmes pour plusieurs milliers de fois à l'aide de boucles répétitives.

Cette mesure permet d'évaluer principalement les outils qui soutiennent une approche.

1.5 La qualité des logiciels

1.5.1 Les attributs internes de mesure

Les métriques présentées auparavant saisissent des informations sur la conception et le code des logiciels en termes d'attributs fondamentaux tels que le couplage, la cohésion et la taille.

- **Couplage**
  Le couplage est une indication du degré d'interconnexions entre les composants d'un système. Les systèmes fortement couplés ont des interconnexions fortes, où les unités de programme dépendent les unes des autres [Sommerville, 2001]. Dans notre étude, CBC et DIT sont les deux métriques qui reflètent l'attribut de couplage.

  – **Impact**
    Le couplage fait référence aux liens entre deux unités distinctes d'un programme.
    - Deux classes qui dépendent chacune fortement des détails de l'autre sont dites fortement couplées.
    - Le couplage faible permet de comprendre le fonctionnement d'une classe sans en analyser d'autres et modifier une classe sans affecter les autres

- **La cohésion.**
  La cohésion mesure le degré de collaboration fonctionnelle entre les éléments d'un même module pour répondre à la préoccupation (i.e. objectif) de ce dernier. La métrique LCOO mesure le manque de cohésion dans notre étude.

  – **Impact**
    - La forte cohésion facilite la compréhension de ce que fait une classe ou une méthode (ou un aspect).

- **La taille**
  La taille mesure la longueur physique de la conception et du code d'un système de logiciel [Fenton, 1997]. Dans notre étude nous avons travaillé avec cinq métriques de taille que sont : LOC, WOM, NOC, NOO, NOA.

  – **Impact**
  – En générale, la réduction de la taille du programme diminue la probabilité d'introduire des erreurs lors du développement d'un système.

Le tableau 8 résume les relations entre les métriques et les attributs de mesure :

Tableau 8 Association entre les métriques et les attributs de mesures

Attributs	**Couplage**		**Cohésion**	**Taille**				
Métriques	CBC	DIT	LCOO	LOC	NOO	NOA	WOC	NOC

1.5.2 Model de qualité

Plusieurs modèles de qualité existent [ISO/IEC 9126, 2001], [Bertoa, 2002], [Sharma, 2008], [Chang, 2008], chaque modèle suggère des caractéristiques différentes afin de comprendre et évaluer la qualité globale des logiciels.

Sant'Anna et al. ont suggéré un framework pour évaluer la réutilisation et la maintenance des logiciels orientés aspect [Sant'Anna, 2003]. Ils ont proposé un modèle qui spécifie la relation entre la réutilisabilité et la maintenabilité et les caractéristiques internes tels que la cohésion, le couplage et la taille.

En effet, d'après Sant'Anna et al, la mesure d'un attribut interne, n'est utile que si elle est liée à la mesure des attributs externes qui ne peuvent pas être mesurés directement comme la réutilisation. Dans ce contexte, ils ont développé un framework pour relier la mesure des attributs internes à deux éléments de qualité, que sont la maintenabilité et la réutilisabilité, en d'autre terme ce framework présente les attributs internes comme des prédicateurs de qualité. L'objectif de leur framework est de fournir un cadre d'évaluation de la réutilisabilité et la maintenabilité des systèmes orientés aspect en se basant sur quelques métriques structurelles.

Les composants de base de ce framework sont les suivants: les métriques structurelles, et le modèle de qualité. Ce dernier clarifie les relations entre la réutilisabilité, la maintenabilité et les métriques structurelles, il guide les ingénieurs dans d'interprétation des données.

Les modèles de qualité sont construits de façon arborescente car la qualité est en fait une composition de nombreuses autres qualités [Fenton, 1997]. Le modèle de qualité Sant'Anna et al est basé sur un examen approfondi d'un ensemble de modèles de qualité existants [Fenton, 1997] [Garcia, 2003], sur des définitions classiques des attributs de qualité [Meyer, 1997] [Sommerville, 2001], sur des théories traditionnelles de conception, telle que celle de Parnas [Parnas, 1972], qui sont communément admises par les chercheurs et les praticiens et enfin sur les attributs de logiciels touchés par les abstractions orientées aspect. Ce modèle a été construit et raffiné en utilisant la méthodologie de GQM de Basili [Basili, 2001].

Le modèle de qualité Sant'Anna et al est composé de trois éléments différents: les qualités, les facteurs et les attributs internes. En outre, le modèle de qualité relie les attributs internes avec les différentes métriques.

- **Les qualités** de la maintenabilité et de la réutilisabilité sont au cœur de ce modèle. Ces deux dernières représentent les deux qualités principalement observées dans le système logiciel.
  - **La réutilisabilité** est la capacité de réutiliser des éléments logiciels pour la construction des différents éléments dans le même système logiciel ou dans des autres [Meyer, 1997]. Ce modèle s'intéresse alors à l'évaluation de la réutilisation des éléments de conception des systèmes orientés aspect.

- **La maintenabilité** indique la facilité avec laquelle les composants logiciels peuvent être modifiés. Les activités de maintenance sont classés en quatre catégories [Fenton, 1997] [Sommerville, 2001] : la maintenance corrective, la maintenance perfective, l'entretien et l'évolution adaptative. Puisque le but principal de la POA est d'améliorer l'évolution des systèmes orienté objet, ce modèle met l'accent sur l'aspect d'évolution dans les systèmes orientée aspect.

- **Les facteurs.** La notion de qualité des logiciels, est généralement capturée dans un modèle qui représente d'autres qualités intermédiaires, ces dernières sont appelées les facteurs. Les facteurs sont alors, les attributs de qualité secondaires qui influent sur les qualités primaires définies auparavant. La flexibilité et la compréhensibilité sont les facteurs centraux pour promouvoir la réutilisation et la maintenabilité [Fenton, 1997] [Meyer, 1997] [Parnas, 1972] [Sommerville, 2001].
  - **La flexibilité** indique le niveau de difficulté pour faire des changements drastiques sur un composant dans un système sans avoir besoin de changer les autres [Parnas, 1972]. un système logiciel doit être suffisamment souple pour supporter l'ajout et le retrait des fonctionnalités et de la réutilisation de ses composants avec un minimum d'effort.

  - **La compréhensibilité** indique le niveau de difficulté pour l'étude et la compréhension de la conception et le code d'un système [Parnas, 1972]. Un système compréhensible a une maintenabilité et une réutilisabilité améliorées; en effet, la plupart des activités de maintenance et de réutilisation exigent que les ingénieurs essaient d'abord de comprendre les composants du système avant d'entreprendre toute modification ou extension.
- **Les attributs internes** sont liés aux métriques structurelles des systèmes logiciels, et qui sont à leur tour essentiels pour l'accomplissement des facteurs et leurs qualités [Fenton, 1997]. Comme indiqué précédemment, chaque attribut interne est lié à un ensemble de métriques proposées.
  - **Couplage.** Les deux métriques CBC et DIT mesurent le couplage avec différents points de vue. La compréhension d'un composant implique la connaissance des autres composants auxquels il est couplé. De ce fait, plus le nombre de couples d'un composant est élevé, plus il est difficile de comprendre le système. En outre, plus le nombre de couples est élevé, plus la sensibilité aux changements est importante, par conséquent, la maintenance est plus difficile. Le couplage entre les composants excessifs est préjudiciable à la conception modulaire et empêche la réutilisation. Plus le composant est indépendant, plus il est facile de le réutiliser dans une autre application.

  - **Cohésion.** La mesure de LCOO détecte le degré auquel un composant implémente une fonction logique unique. Plus le degré auquel les différentes actions réalisées par un composant contribuent à des fonctions distinctes, plus il est difficile de réutiliser et maintenir le composant ou l'une de ses fonctionnalités.

- **Taille.** Les métriques de taille concernent les différents aspects de taille du système logiciel. En général, plus la taille est élevée, plus il est difficile de comprendre le système. Par exemple, la métrique LOC mesure les lignes de code d'un logiciel, plus le nombre de lignes de code est élevé, plus il sera difficile de comprendre le système. En effet, plus le nombre de lignes de code est élevé, plus il sera difficile de trouver les lignes qui doivent être changées au cours des activités d'évolution ou comprendre la mise en œuvre des fonctionnalités nécessaires au cours des activités de réutilisation.

De ce fait, on trouve une relation très forte entre les attributs internes et les facteurs de base. Le couplage et la cohésion affectent la compréhension car et comme nous avons déjà expliqué, un composant du système ne peut pas être compris sans faire référence aux autres composants avec les quelles il est lié. La flexibilité est influencée par les attributs de couplage et la cohésion. Une haute cohésion, et un faible couplage sont souhaités car ils signifient que les composants du système sont indépendants ou presque indépendants. Il est à noter que le facteur de flexibilité n'est pas influencé par la taille. En outre, L'attribut de taille peut indiquer l'effort nécessaire pour comprendre les différents composants d'un logiciel.

La figure 29 présente le modèle de qualité, proposé par Sant'Anna et al. Les branches supérieures contiennent les qualités de haut niveau. Les attributs internes sont plus faciles à mesurer que les qualités et les facteurs car ils sont reliés aux métriques réelles du logiciel.

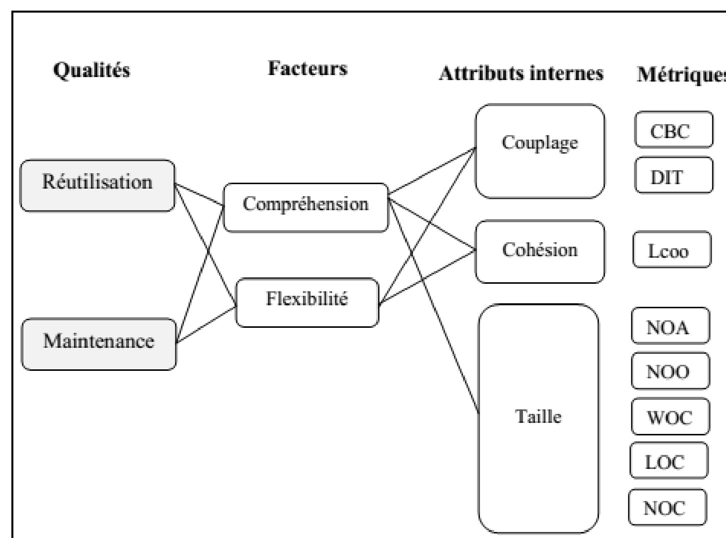

Figure 29 Modèle de qualité [Sant'Anna, 2003]

2 Étude comparative

Après la présentation des patrons de conception, de la programmation orientée aspect, ses principes, ses nouveaux concepts, ses langages AspectJ, JBoss AOP et CaesarJ, et enfin les métriques les plus importantes d'évaluation de la qualité, nous allons présenter notre étude comparative sur AspectJ, JBoss AOP et CaesarJ, et qui

comporte deux comparaisons, quantitative et qualitative, en utilisant les modèles de conception de GoF comme des benchmarks hypothétiques.

2.1 Comparaison quantitative

Après avoir implémenté les 23 modèles de conception avec AspectJ, JBoss AOP et CaesarJ, nous avons effectué une comparaison quantitative basée essentiellement sur des métriques structurelles et de performance, cette comparaison permet de donner des résultats exacts sur certains facteurs importants du génie logiciel comme la compréhension, la réutilisation et la maintenance.

2.1.1 Méthode de travail

La méthode adoptée dans notre étude exige de calculer les différentes métriques structurelles présentées auparavant. Au début de notre étude nous avons essayé de faire les calculs automatiquement, cependant et après avoir testé plusieurs outils de calcul, nous avons constaté qu'aucun d'entre eux n'est suffisamment mature pour supporter les trois approches.

Par exemple, AJATO [Figueiredo, 2006] qui est un outil d'évaluation pour Java et AspectJ ne prend pas en charge les concepts CaesarJ. MuLaTo [Bartolomei, 2006] semble le plus approprié, c'est une application Java conçue pour collecter des mesures des programmes écrits en plusieurs langages. Le module de base de MuLaTo fournit des analyseurs pour les programmes Java, AspectJ et CaesarJ et peut être étendue à d'autres langages. Malheureusement, la version actuelle de MuLaTo ne supporte que quelques mesures de la taille, ce qui nous a obligés à calculer les autres paramètres de couplage et la cohésion manuellement.

Par la suite et selon la nature des mesures utilisées, nous avons effectué trois comparaisons, la première est basée sur les métriques étendues de la programmation objet, la deuxième sur les métriques propres à la programmation orientée aspect et la dernière sur la métrique de performance.

Dans la première comparaison nous avons analysé les résultats obtenus suivant l'attribut qu'ils représentent. Ces attributs sont : le couplage, la cohésion et la taille.

Le schéma de la figure 30 résume la méthode de travail suivie dans notre comparaison quantitative :

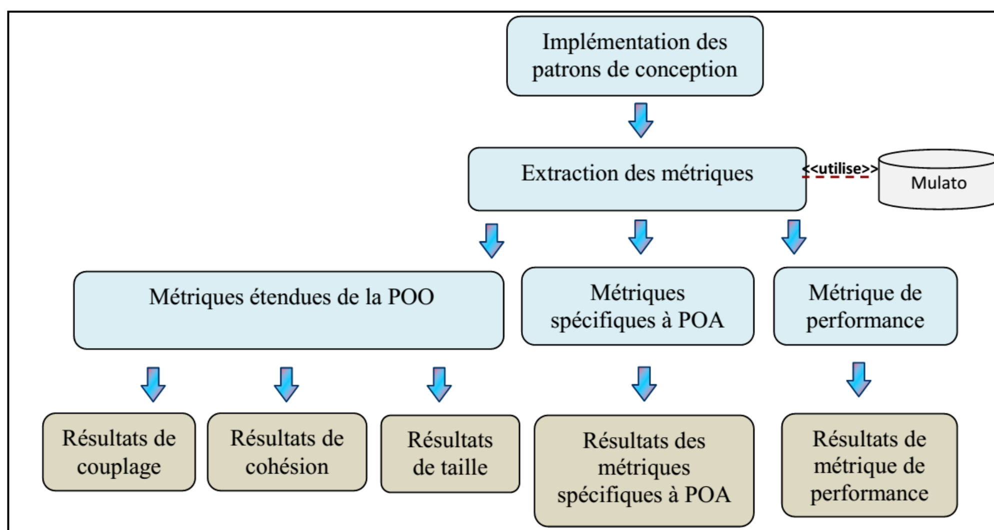

Figure 30 Comparaison quantitative

2.1.2 Résultats obtenus

Suivant la nature des métriques collectées nous avons eu trois types de résultats que nous présentons ci-après.

2.1.2.1 *Résultats des métriques étendues de la programmation objet*

Comme indiqué précédemment, les différentes métriques étendues de la programmation objet touchent trois facteurs importants du génie logiciel, que sont : la taille, le couplage et la cohésion. Le tableau 9 présente les résultats obtenus pour ces trois attributs :

Tableau 9 Résultats des métriques étendues de la programmation objet

<table>
<tr><th></th><th colspan="5">METRIQUES DE TAILLE</th><th colspan="2">METRIQUES DE COUPLAGE</th><th>METRIQUES DE COHESION</th></tr>
<tr><th></th><th>LOCC</th><th>WOM</th><th>NOC</th><th>NOO</th><th>NOA</th><th>CBC</th><th>DIT</th><th>LCOO</th></tr>
<tr><td>AspectJ</td><td>1568</td><td>540</td><td>31</td><td>334</td><td>62</td><td>452</td><td>2</td><td>111</td></tr>
<tr><td>JBoss AOP</td><td>2107</td><td>605</td><td>18</td><td>367</td><td>79</td><td>676</td><td>3</td><td>27</td></tr>
<tr><td>CaesarJ</td><td>1710</td><td>612</td><td>62</td><td>367</td><td>60</td><td>498/872</td><td>4</td><td>20</td></tr>
<tr><td>Meilleur résultat pour la métrique</td><td>AspectJ</td><td>AspectJ</td><td>JBoss</td><td>AspectJ</td><td>CaesarJ</td><td>AspectJ</td><td>AspectJ</td><td rowspan="2">CaesarJ</td></tr>
<tr><td>Meilleur résultat pour l'attribut (taille, couplage, cohésion)</td><td colspan="5">AspectJ</td><td colspan="2">AspectJ</td></tr>
</table>

Dans ce qui suit nous allons présenter les résultats de chaque attribut.

- **Résultats de taille**

  La figure 32 illustre graphiquement les résultats des métriques de taille pour AspectJ, JBoss AOP et CaesarJ (pour cette métrique un résultat faible est préférable).

  Pour l'attribut de taille on considère l'approche qui donne de bons résultats pour la majorité des métriques de taille comme la meilleure. Par exemple, avec 13 patrons de conception AspectJ a donné les meilleurs résultats avec la majorité des métriques de taille par rapport à JBoss AOP et CaesarJ. Ces patrons sont : Commande, Composite, COR, Médiateur, Observateur, Poids mouche, Itérateur, Prototype, Stratégie, Visiteur 2, Constructeur, Patron de méthode, Etat. La plus part de ces patrons sont des patrons de comportement ou il y a plusieurs interactions entre les composants constituant leurs solutions.

  Le patron de conception Mémento donne des résultats différents pour les trois approches. AspectJ prend la première place pour le nombre de lignes de code, JBoss AOP dans le nombre des enfants et CaesarJ dans le nombre d'opérations. Dans les deux autres métriques les trois approches donnent des résultats similaires. Cela nous a empêché de classifier les approches pour ce patron.

  Visiteur1 (La première variante de l'implémentation du patron Visiteur) donne des bons résultats pour AspectJ et JBoss AOP avec les trois métriques : WOM, NOC, NOA, Cependant, sa réalisation a pris peu de lignes de code avec AspectJ comparé à celle de JBoss AOP, en revanche peu de méthodes sont utilisées avec la solution JBoss AOP.

Prototype, Décorateur, Fabrique abstraite, Pont, Fabrique, et Interpréteur donnent des résultats meilleurs avec CaesarJ. La majorité de ces patrons sont des patrons de structure, ce qui prouve l'efficacité de CaesarJ dans la résolution des problèmes de structure.

- **Résultats du couplage**

Comme nous avons déjà mentionné, CaesarJ présente une nouvelle méthode pour la modularisation, il offre une nouvelle approche de conception, ainsi il utilise un mécanisme de classes virtuelles qui permet aux cclasses de contenir d'autres cclasses de la même manière que d'avoir des méthodes et des attributs. Ces classes internes sont elles aussi des classes virtuelles et elles ont des propriétés différentes de celles de Java. Conceptuellement, la relation entre les classes internes affecte l'attribut de couplage.

Dans notre étude nous avons pris en compte deux types de couplage avec CaesarJ, pour cette raison nous allons avoir deux résultats pour l'attribut de couplage dans le diagramme de résultats (i.e. Prendre en compte le couplage produit entre les classes internes dans le premier calcul, et négliger ce genre de couplage dans le deuxième)

La Figure 31 résume les résultats de couplage avec le patron de conception Observateur, la première ligne et colonne représentent les cclasses de la solution où chaque cclasse peut contenir d'autres cclasses internes.

Nous avons deux résultats, le premier est obtenu en prenant en charge le couplage produit entre les classes internes. Par exemple dans le compte de la métrique CBC_I d'*ObserverProtocol* cclass les deux classes internes sont couplées ensemble. En effet, la classe virtuelle *Observer* appelle la classe interne *Subject* et vice versa, ce qui donne un résultat de 2. Par contre la cclass *ObserverProtocol* ne fait aucun appel à une autre cclass ce qui donne un résultat négatif pour cette cclasse.

La figure 33 illustre graphiquement les résultats des métriques de couplage pour : AspectJ, JBoss AOP et CaesarJ (un couplage faible est préférable)

La majorité des patrons de conception implémentés avec AspectJ donnent des bons résultats pour l'attribut de couplage. Ces patrons sont : Commande, Composite, COR, Médiateur, Observateur, Poids mouche, Itérateur, Mémento, Prototype, Adaptateur, Stratégie, Visiteur1, Visiteur2, Pont, Constructeur, Patron de méthode, Interpréteur et Etat.

Quatre patrons de conception ont donné les meilleurs résultats pour JBoss AOP ces patrons sont : Singleton, Décorateur, Proxy et Etat. Tous ces patrons utilisent des intercepteurs dans leurs réalisations ce qui réduit le couplage produit entre les éléments de leurs solutions.

Dans le cas où on prend en compte le couplage produit entre les classes internes, CaesarJ donne des mauvais résultats. Seulement trois patrons de conception donnent des bons résultats. Ces patrons sont : Adaptateur, Décorateur et Fabrique abstraite. Par contre, si on néglige ce genre de couplage on trouve que CaesarJ donne des bons résultats avec six patrons de conception, que sont : Adaptateur, Décorateur, Pont, Fabrique abstraite, Constructeur, et Fabrication. Notons que la plupart de ces partons sont des patrons de structure aussi.

Couplage entre les classes internes

Inner cclass

Cclass

	1- ObserverProtocol	1-1 ObserverProtocol-Observer	1-2 ObserverProtocol-Subject	2- ObserverImp	2-1- ObserverImp§Subject	3- ObserverBinding	3-1 ObserverBinding-FlowerOpening	3-2 ObserverBinding-FlowerClosing	3-3 ObserverBinding-BeeIsOpenObserver	3-4 ObserverBinding-BeeIsCloseObserver	3-5 ObserverBinding-HumIsOpenObserver	3-6 ObserverBinding-HumIsCloseObserver	4- FlowerObserverDeploy	5- Flower	6- Bee	7- Hummingbird	8-Main	CBCi inner classes		CBCi Cclass
1- ObserverProtocol																		0	2	0
1-1 ObserverProtocol-Observer			1															1		
1-2 ObserverProtocol-Subject		1																1		
2- ObserverImp	1																	1	3	1
2-1- ObserverImp§Subject		1	1															2		
3- ObserverBinding	1													1				2	18	4
3-1 ObserverBinding-FlowerOpening			1											1				2		
3-2 ObserverBinding-FlowerClosing			1											1				2		
3-3 ObserverBinding-BeeIsOpenObserver		1	1												1			3		
3-4 ObserverBinding-BeeIsCloseObserver		1	1												1			3		
3-5 ObserverBinding-HumIsOpenObserver		1	1													1		3		
3-6 ObserverBinding-HumIsCloseObserver		1	1													1		3		
4- FlowerObserverDeploy				1		1												2	2	2
5- Flower																		0	0	0
6- Bee																		0	0	0
7- Hummingbird																		0	0	0
8-Main							1	1	1	1	1	1	1	1	1	1		10	10	5
CBCe inner classes	2	6	8	1	0	1	1	1	1	1	1	1	1	4	3	3	0		35	12
	16			1		7							1	4	3	3	0	35		
CBCe Cclass	2			1		2							1	2	2	2	0	12		

Figure 31 Résultats de couplage du patron de conception Observateur (Avec CaesarJ)[1][Debboub, 2014]

Décorateur donne de bons résultats avec JBoss AOP et CaesarJ, le déploiement dynamique des aspects offert par ces deux approches était très utile dans l'implémentation du patron de conception Décorateur, où l'intention de ce dernier exige d'attacher des responsabilités supplémentaires à un objet dynamiquement.

- **Résultats de cohésion**

La figure 34 illustre graphiquement les résultats de cohésion pour : AspectJ, JBoss AOP et CaesarJ (une cohésion forte est préférable)

Dans l'attribut de cohésion les trois approches ont donné de bons résultats, cela est dû à l'efficacité des nouveaux concepts introduit par la programmation orientée aspect et qui améliore la modularisation. Cependant, dans cet attribut, AspectJ a donné l'occasion à CaesarJ pour prouver sa force. Quelques patrons de conception ont montré un manque de cohésion avec AspectJ, comme Commande, Composite, Observateur et Interpréteur.

L'approche CaesarJ est basée essentiellement sur le principe qui insiste sur le rassemblement de composants qui travaillent ensemble, ce qui explique les bons résultats de CaesarJ par rapport aux autres approches dans l'attribut de cohésion.

[1] CBC_I : Mesure le couplage Intérieur / CBC_E : Mesure le couplage extérieur.

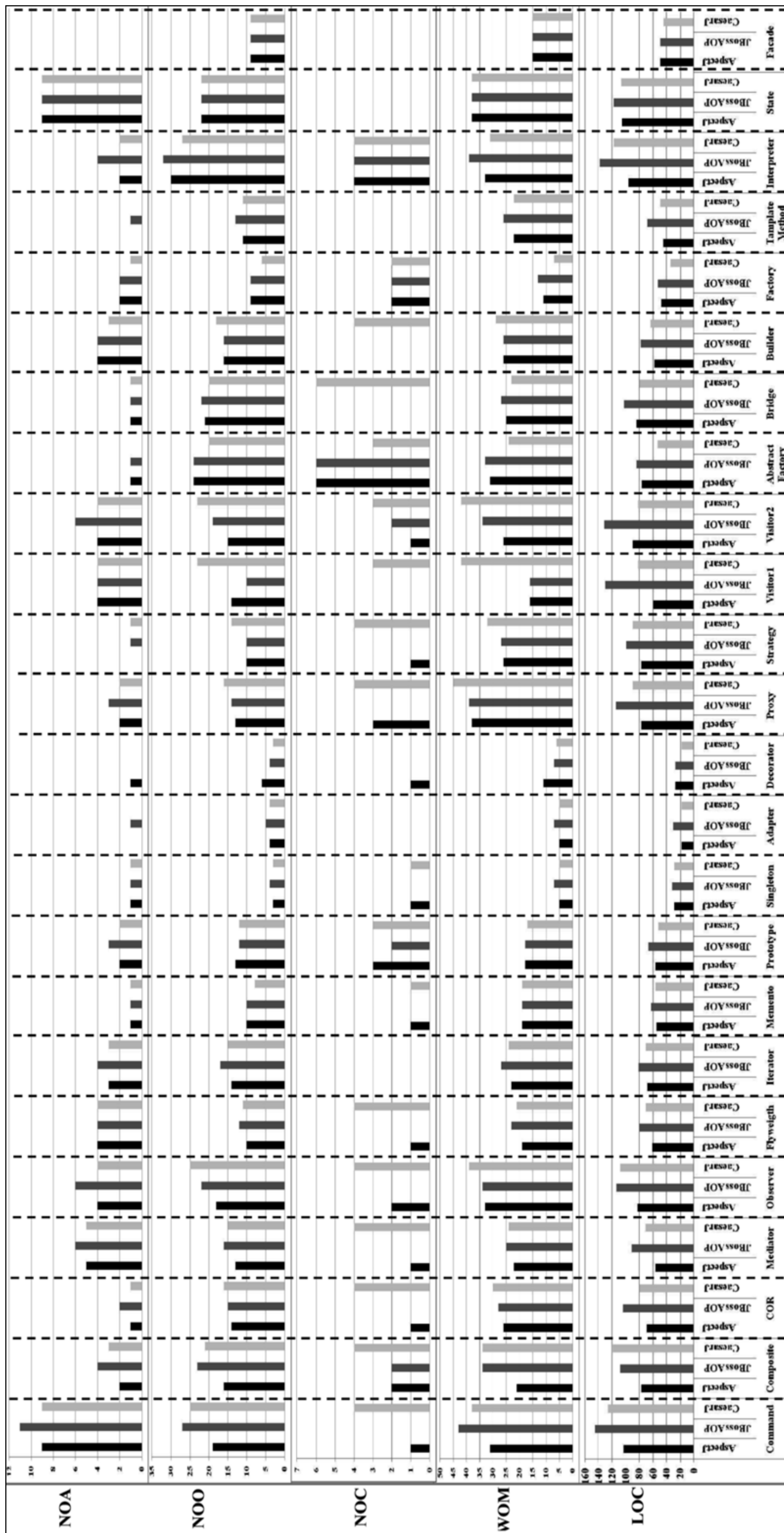

Figure 32 Résultats des métriques de taille

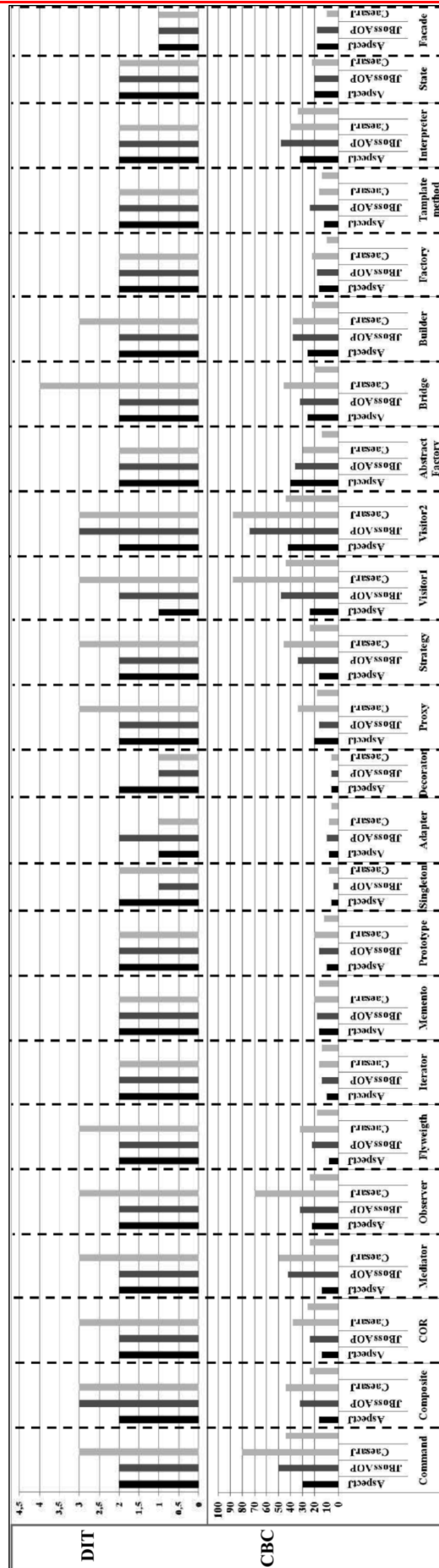

Figure 33 Résultats des métriques de couplage

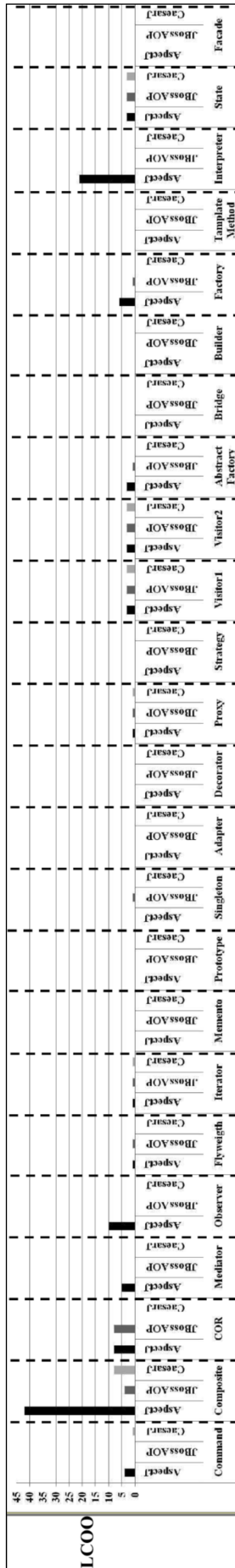

Figure 34 Résultats de la métrique de cohésion

2.1.2.2 *Résultats des métriques spécifiques pour la programmation orientée aspect*

La figure 35 illustre graphiquement les résultats obtenus en utilisant les métriques spécifiques à la programmation orientée aspect. Ces métriques reflètent l'utilisation des nouveaux concepts de la programmation orientée aspect par les trois approches : AspectJ, JBoss AOP et CaesarJ.

Figure 35 Résultats des métriques spécifiques à la programmation orientée aspect.

Les résultats montrent que l'utilisation des points de coupure avec AspectJ dépasse celle de JBoss AOP et CaesarJ, cela a donné une valeur élevé pour l'attribut de CIM dans la plupart des patrons de conception d'AspectJ. CaesarJ utilise moins de point de coupure dans l'implémentation des différents patrons de conception.

Par contre, dans l'utilisation de la métrique NCI on trouve qu'avec un minimum de composants additionnels aux classes de base AspectJ peut réaliser les différents patrons de conception, cela simplifie la compréhension en réduisant la taille de l'application.

Par exemple nous présentons dans le tableau 10 les résultats de la métrique NCI dans l'exemple illustratif Observateur (Présenté dans le chapitre précédent). Notons que les composants introduits sont ombrés.

Tableau 10 Résultats de la métrique NCI dans le patron de conception Observateur

AspectJ	JBoss AOP	CaesarJ
ObserverProtocol	Observable	ObserverProtocol
HummingbirdObserver	ObservableMixIn	ObserverImp
BeeObserver	FlowerObserver	ObserverBinding
Flower	FlowerObserverMixIn	FlowerObsDeploy
Bee	ObserverAspect	Flower
Hummingbird	XMLFile	Bee
Main	Flower	Hummingbird
	Bee	Main
	Hummingbird	
	Main	
3	**6**	**4**

Le tableau 11 résume les différents résultats des métriques spécifiques à la programmation orientée aspect.

Tableau 11 Résultats des métriques spécifiques à la programmation orientée aspect

METRIQUES AOP	CIM	CDA	NCI	NAP	NCP	NP
AspectJ	24	66	43	5	20	25
JBoss AOP	14	56	94	0	17	17
CaesarJ	11	50	69	1	15	16
MEILLEUR RSULTAT	CaesarJ	CaesarJ	AspectJ	JBoss AOP	CaesarJ	CaesarJ

2.1.2.3 *Résultats de performance*

Le tableau 12 présente les résultats de la métrique de la performance (en millisecondes) pour AspectJ, JBoss AOP et CaesarJ (un résultat faible est préférable).

Cette métrique fournit une estimation sur l'IDE et l'outil support d'AspectJ, JBoss AOP et CaesarJ.

Tableau 12 Résultats de performance

	AspectJ	JBoss AOP	CaesarJ
COMMANDE	577	608,5	530,5
COMPOSITE	7019	6075	7385
COR	1639	1893	1803
MEDIATEUR	339,5	764	300
OBSERVATEUR	1795,5	2016	1873
POIDS MOUCHE	1428,5	1403	1420,5
ITERATEUR	1038	1297,5	1116,5
MEMENTO	1530	1711	1560,5
PROTOTYPE	858	868,5	522,5
SINGLETON	231	430	203
ADAPTATEUR	2954,5	2751,5	3037
DECORATEUR	483,5	850,5	329,5

PROXY	5816,5	4435	5887
STRATEGIE	1033,5	934	999
VISITEUR1	2630,5	2818	2568,5
VISITEUR2	3029	2835	/
FABRIQUE ABSTRAITE	866,5	1455	725,5
PONT	4652,5	4440	4614
MONTEUR	2443	2188	2443,5
FABRICATION	1015	1317,5	999,5
PATRON DE METHODE	1670	1843	1522
INTERPRETEUR	273,5	604	195,5
ETAT	4323,5	3599,5	4363,5
FACADE	2682	2682	2682

2.2 Interprétation des résultats

2.2.1.1 Interprétation des résultats des métriques étendues de la programmation objet

- **Interprétation des résultats de taille**

Comme présenté précédemment, pour l'attribut de taille, AspectJ prend la première place, cela revient principalement à la simplicité de son mécanisme d'introduction qui est la base d'implémentation de la plupart des patrons de conception.

En effet, le mécanisme d'introduction d'AspectJ donne aux aspects le pouvoir de changer le comportement ou la hiérarchie des classes de base avec des simples commandes, ce qui réduit le nombre des composants utilisés. Par exemple, l'une des solutions proposée pour le patron de conception Visiteur repose principalement sur ce mécanisme, ce qui nous a permit d'implémenter ce patron avec un seul aspect concret.

JBoss AOP prend la troisième place dans l'attribut de taille, cela est expliqué par le fait qu'il travaille avec les composants Java, ce qui rend son mécanisme d'introduction moins souple que celui d'AspectJ et demande trois éléments différents pour son implémentation. Ces éléments sont : les interfaces, les classes Mix-In et le fichier XML.

L'utilisation de différents éléments pour assurer l'introduction augmente les valeurs de toutes les métriques de taille, comme le nombre de lignes de code, ou de méthodes.

Aussi les relations entre les classes de base et les classes Mix-In nécessitent souvent l'utilisation des attributs, ce qui explique la valeur élevée de ces derniers dans les résultats obtenus.

CaesarJ repose sur le raffinement des hiérarchies des composants, pour la réalisation des différentes préoccupations, ce raffinement augmente les valeurs des différentes métriques de taille et spécialement la métriques NOC qui indique les nombre d'enfants de chaque composant et représente le double du résultat obtenu avec AspectJ et JBoss AOP.

- **Interprétation des résultats de couplage**

En générale, le couplage d'AspectJ réside dans les aspects concrets, car c'est là ou on réalise les multiples relations entre les classes de base et les rôles assignés par les patrons et représentés par les interfaces internes déclarées dans les aspects abstraits.

Le couplage dans AspectJ donne un résultat remarquable comparé à celui de JBoss AOP ou CaesarJ surtout avec le premier groupe ou il ya beaucoup d'interactions entre les classes de base et leurs rôles assignés.

Dans JBoss AOP, le couplage entre les classes de base et les aspects réside dans le fichier XML, puisque c'est ici ou on trouve les relations créant les différentes préoccupations.
Les interactions entre les composants introduits dans l'implémentation des patrons de conception comme les interfaces et les classes Mix-In augmentent aussi la valeur de cet attribut. Cependant ce problème n'est pas soulevé avec les patrons de conception qui utilisent des intercepteurs dans leurs réalisations, comme le patron de conception Singleton où on a utilisé un seul intercepteur pour gérer le mécanisme d'instanciation, ce qui a réduit la valeur de couplage dans ce patron.

CaesarJ ne donne pas de bons résultats pour cet attribut, cela est peut être justifié par le fait que les différents éléments constituants le composant CaesarJ forment une sorte de couplage, en plus du couplage produit entre les classes internes de ces composants.

- **Interprétation des résultats de cohésion**

La cohésion était bonne dans les différentes implémentations, cela revient principalement aux nouveaux concepts introduits par la programmation orientée aspect.

CaesarJ a prit la première place dans la cohésion, le mécanisme supporté par cette approche localise les différentes préoccupations dans des unités distinctes à plusieurs niveaux ce qui donne une meilleure modularisation.

AspectJ a montré quelques faiblesses dans cet attribut, surtout avec les patrons de comportement, comme l'Observateur et la chaine de responsabilités. Dans [Mezini, 2003], Mezini et Ostermann ont critiqué AspectJ à travers l'analyse du patron de conception Observateur, parmi ces critiques, ils notent un manque de support pour la multi-abstraction. Ils trouvent que le fait d'implémenter les deux méthodes *addObserver()* et *removeObserver()* dans l'aspect de haut niveau constitue une mauvaise séparation des préoccupations à l'intérieur de l'aspect lui-même.
Ce problème peut expliquer les mauvais résultats de la cohésion pour certains patrons de conception dans notre étude.

2.2.1.2 *Interprétation des résultats des métriques spécifiques pour la programmation orientée aspect*

Les résultats des métriques spécifiques pour la programmation orientée aspect, montrent qu'AspectJ utilisent plus de points de coupure que JBoss AOP et CaesarJ. Suivant les remarques de Sousa et al [Sousa, 2008], l'utilisation des points de coupure dans CaesarJ, doit être laissé aux situations dont lesquelles les conceptions structurelles ne peuvent pas résoudre le problème affronté, ce qui explique l'utilisation faible des points de coupure par ce dernier.

Cette métrique indique à son tour un fort couplage entre les classes de l'application et les points de coupure des aspects, ce qui limite leur réutilisation tels quels sont et nécessite la redéfinition de leurs points de coupure.

Dans la définition des deux métriques CIM et CDA nous avons dit qu'il est souhaitable d'avoir un nombre réduit de CIM et élevé de CDA, dans le tableau des

résultats, et si on calcule les relations en pourcentage entre ces deux métriques on trouve : dans AspectJ 2 ,75%, JBoss AOP 4% et CaesarJ 4,54 %. Ce qui donne plus d'avantage à CaesarJ puisque avec un nombre réduit des modules explicitement cités dans les points de coupure nous avons pu modifier plus de modules implicitement cités.

Par contre, pour la métrique que nous avons proposée, NCI, et qui calcule le nombre de modules introduits par l'approche dans l'implémentation des préoccupations, on trouve qu'AspectJ utilise moins de modules, cela revient principalement à la simplicité de son mécanisme d'introduction et l'approche qu'il suit. Le composant CaesarJ nécessite dans la plus part du temps l'utilisation de plusieurs composants, ces derniers ont des avantages au niveau de la cohésion mais influencent négativement la taille des programmes.

2.2.1.3 Interprétation des résultats de performance

Dans la plupart des patrons de conception, JBoss AOP a donné les mauvais résultats pour le temps d'exécution, ce problème est plus évident avec les patrons qui utilisent les intercepteurs comme Singleton et Décorateur.

L'étape de tissage de JBoss AOP peut être la cause de ces résultats, puisque son tisseur devient très lent au moment du tissage, car il fait des activités supplémentaires avant le chargement des classes. Par la suite, et une fois que les classes sont chargées, le tissage n'aura plus d'effets sur la performance de l'application [JBoss AOP, Document2].

2.3 Comparaison qualitative

La comparaison qualitative touche les concepts qui ne peuvent pas être mesurés quantitativement, alors que ces concepts ne manquent pas d'importance pour l'évaluation des grands projets informatiques.

2.3.1 Méthode de travail

Après l'implémentation des 23 patrons de conception avec les trois approches, nous avons essayé d'analyser la façon suivie par chaque approche pour la résolution des différents problèmes posés par les patrons de conception.

Par la suite les étudiants ont été sollicités à répondre à quelques questions concernant quatre catégories, que sont : la compréhension de l'approche, les concepts de l'approche, la maturité des outils support et enfin la difficulté trouvée lors de l'implémentation des 23 patrons de conception.

Pour cela, les étudiants ont été invités à remplir un formulaire de questions, chaque question nécessite une note et propose un champ de saisi si l'étudiant souhaite donner un commentaire. Le formulaire comporte les questions suivantes pour chaque approche.

- **Questions sur la compréhension de l'approche**
  - Est-il facile d'apprendre cette approche?
  - Est-ce que les nouveaux termes de cette approche sont faciles à maitriser ?
  - Y a-t-il des documents suffisants pour apprendre cette approche ?
- **Question sur les concepts de l'approche**
  - Est-ce que les nouveaux concepts de cette approche sont faciles à comprendre et à utiliser?
  - Pensez-vous que les concepts de cette approche sont appropriés pour l'implémentation des patrons de conception ?

– Est-ce que ces concepts sont suffisants pour la création d'abstractions réutilisables ?

- **Questions sur l'IDE et les outils support**

  – Pensez-vous que l'IDE utilisé est approprié et facile à utiliser?
  – Est-il facile de trouvé des outils adéquats avec cette approche ?
  – Pensez-vous que les outils support de cette approche sont mâtures?

- **Questions sur les patrons de conception**

  – Pensez-vous que cette approche aide les développeurs dans l'implémentation des patrons de conception?
  – Pensez-vous que les implémentations obtenues sont faciles à comprendre et à faire évoluer ?

2.3.2 Résultats obtenus

2.3.2.1 Analyse des implémentations

La majorité des solutions aspect proposées pour les patrons de conception facilitent la réutilisation des codes en offrant des composants qui prennent en charge les différentes préoccupations causées par les rôles des patrons. En effet, chaque approche aspect utilise sa puissance pour satisfaire le but de la séparation des préoccupations et rendre les classes de base simples et légères.

AspectJ utilise des aspects abstraits pour capturer les préoccupations additionnels et repose sur l'héritage de ces aspects en utilisant les aspects concrets pour relier ces différentes préoccupations avec les classes de base.

JBoss AOP utilise les interfaces et le concept Mix-in pour capturer et projeter les différentes préoccupations sur une application particulière.

Le composant CaesarJ gère bien les différentes situations, en s'appuyant sur le principe d'enveloppement. Il permet de capturer les préoccupations supplémentaires dans des composants qui enveloppent par la suite les différentes classes de base.

Le schéma de la figure 36 résume les différentes stratégies employées par les trois approches.

On remarque que la partie interface de collaboration de CaesarJ représente le niveau le plus abstrait par rapport aux autres parties. En effet, dans les différentes implémentations l'interface de collaboration n'était en aucun car reliée à une application ou même un type de donné spécifique.

2.3.2.2 Réponses au questionnaire

Tous les étudiants ont rendu les formulaires notés et dans la plupart des cas commentés, le tableau 13 résumé ces notes en donnant les moyennes obtenues pour les trois approches AspectJ, JBoss AOP et CaesarJ.

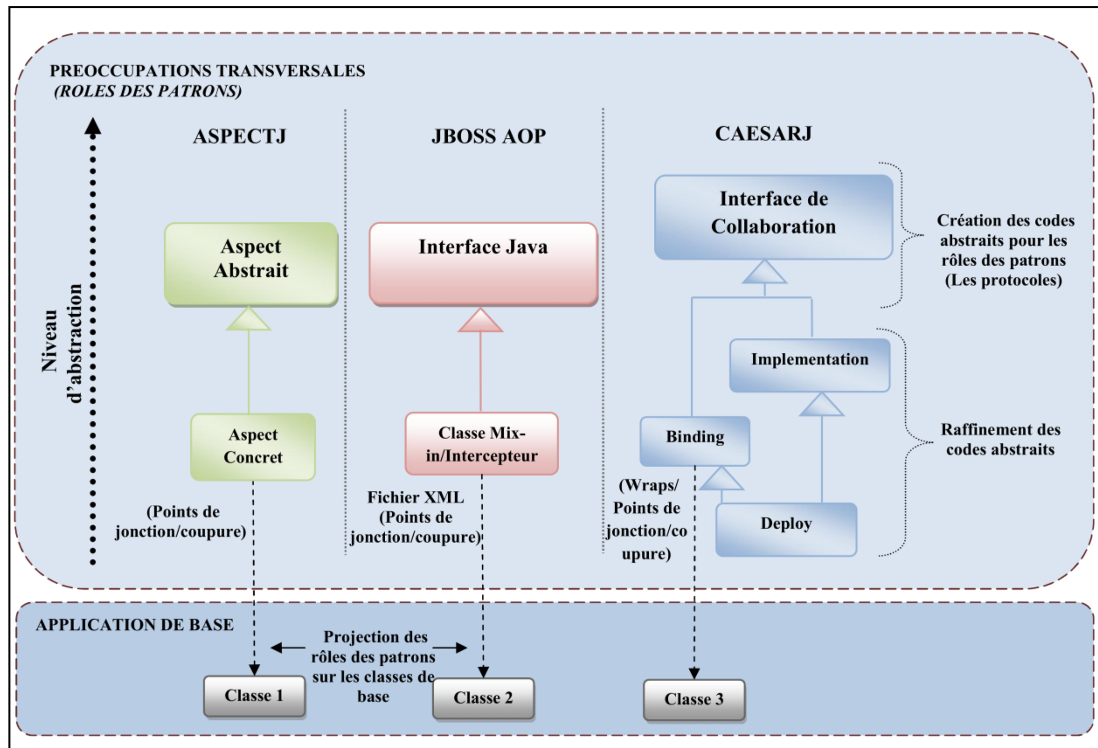

Figure 36 Les différentes stratégies employées par les trois approches

Tableau 13 Résultats de la comparaison qualitative

		AspectJ	JBoss AOP	CaesarJ
Compréhension de l'approche	Est-il facile d'apprendre cette approche?	9	7	7
	Est-ce que les nouveaux termes de cette approche sont faciles à maitriser ?	9	7	6
	Y a-t-il des documents suffisants pour apprendre cette approche ?	10	5	7
Les concepts de l'approche	Est-ce que les nouveaux concepts de cette approche sont faciles à comprendre et à utiliser?	8	7	7
	Pensez-vous que les concepts de cette approche sont appropriés pour l'implémentation des patrons de conception ?	9	5	9
	Est-ce que ces concepts sont suffisants pour la création des abstractions réutilisables ?	6	4	9
IDE et outil support	Pensez-vous que l'IDE utilisé est approprié et facile à utiliser?	9	9	9
	Est-il facile de trouvé des outils adéquats avec cette approche ?	8	5	7
	Pensez-vous que les outils support de cette approche sont mâtures?	9	5	6
Patrons de conception	Pensez-vous que cette approche aide les développeurs dans l'implémentation des patrons de conception?	9	5	9
	Pensez-vous que les implémentations obtenues sont facile à comprendre et à évoluer ?	7	6	8

2.4 Interprétations des résultats

2.4.1 Interprétation des résultats de l'analyse des implémentations

L'analyse des implémentations, peut donner une idée sur la démarche suivie par chaque approche, ainsi, elle peut être utilisée pour déduire des lignes directives pour chacune elle.

- **Pour AspectJ**

Les programmes avec AspectJ sont construits généralement comme suit :

- Tout d'abord, il faut mettre la partie la plus abstraite de l'application (dans nos exemples, ces parties sont les rôles attribués par les patrons de conception) dans les aspects abstraits. Les méthodes de cette partie sont souvent abstraites.
  On peut déclarer aussi des interfaces internes suivant les rôles des patrons, ou même des points de coupures abstraits afin de les identifier plus tard.
- Ensuite, utiliser des aspects concrets qui héritent des aspects abstraits et qui implémentent les différents méthodes et points de coupure déclarés abstraits dans la première partie.
  Afin de lier ce code avec les classes de base, on peut rajouter des méthodes, des points de coupure, des advices, des déclarations inter-type selon le besoin.

- **Pour JBoss AOP**

JBoss AOP, suit la même démarche que celle d'AspectJ, en revanche et puisque JBoss AOP n'étend pas le langage Java, on se trouve obligé d'utiliser les éléments de ce langage, surtout en ce qui concerne les partie réutilisables et qui se résument en interfaces et en classes abstraites. En générale un programme avec JBoss AOP se construit comme suit :

- Tout d'abord, il faut mettre la partie la plus abstraite de l'application sous forme de signatures dans des interfaces (dans nos exemples, ces parties sont les rôles attribués par les patrons de conception).
- Ensuite, on implémente les méthodes déclarées dans les interfaces dans des classes Mix-In.
- On crée les relations entre les classes de base, et les différentes préoccupations implémentées dans les classes Mix-in dans des fichiers XML (ou des annotations). On ajoute aussi les points de coupure, les déclarations des classes aspect ainsi que l'identification des codes advice.

- **Pour CaesarJ**

La structure de composant CaesarJ fournit un guide précis de l'endroit où chaque morceau de code doit être placé et propose le procédé suivant:

- Tout d'abord, il faut mettre la partie la plus abstraite de l'application (dans nos exemples, ces parties sont les rôles attribués par les patrons de conception) dans l'interface de collaboration. Cette partie ne doit en aucun cas être reliée a une application ou un type de donné spécifique.

- Ensuite, placer tout le code qui ne dépend pas des types spécifiques dans la partie implémentation CaesarJ. A la différence de l'interface de collaboration les méthodes des classes de cette partie peuvent être implémentée et reliées à un grand nombre d'applications.
- Placer le code spécifique à une application dans la partie Binding, et établir les liaisons entre ce code et les classes de base en utilisant les points de coupure ou les wraps.
- Créer le composant complet à travers une classe vide de CaesarJ (i.e. weavelet) qui hérite des deux parties implémentation et Binding.

2.4.2 Interprétation des résultats des réponses au questionnaire

Nous allons dans ce qui suit analyser les différents résultats des réponses au questionnaire.

- **Compréhension de l'approche**

Suivant les résultats obtenus dans le tableau précédent (Tableau 13), AspectJ était le plus facile à comprendre. Contre notre attente, qui est allée plutôt vers JBoss AOP puisque il utilise du Java pur et il n'introduit pas de nouveau mots clé, les réponses des étudiants montrent que les nouveaux termes d'AspectJ sont faciles à comprendre et à maitriser. CaesarJ est un peu plus difficile à comprendre par rapport à AspectJ, surtout au début, comme le dit un étudiant en laissant un commentaire sur la première question *« j'ai trouvé la compréhension de CaesarJ plus difficile qu'AspectJ, mais une fois que les concepts de base de CaesarJ sont bien compris il peut être très utile* ». La difficulté de JBoss AOP réside dans les relations entre les classes java et les fichiers XML.

La majorité des étudiants ont trouvé des difficultés pour trouver des documents sur JBoss AOP et CaesarJ, comparé à AspectJ, cela est un facteur important et il peut même expliquer la mauvaise note obtenue par ces deux approches dans cette première catégorie.

- **Les concepts de l'approche**

D'après les réponses des étudiants, les nouveaux concepts d'AspectJ sont plus faciles à comprendre et à utiliser. Pour CaesarJ ils trouvent que pour le maitriser il faut d'abord comprendre ses composants et les relations entre leurs classes internes. Comme suggère un étudiant « *vous devez seulement comprendre l'objectif de chaque partie de composant CaesarJ* »

Concernant l'adaptabilité des nouveaux concepts de ces approches avec les patrons de conceptions, AspectJ et CaesarJ semblent avoir la même appréciation et il sont bien vu par rapport à JBoss AOP. Voici deux réponses de deux étudiants sur la cinquième question : « *Oui, puisque AspectJ gère la séparation des préoccupations d'une façon très efficace* » et « «*Absolument, CaesarJ est très utile, il simplifie les patrons de conception, et rend le patron bien organisé dans les différentes parties du composant CaesarJ* »

Concernant la possibilité de créer des éléments réutilisables, les aspects abstraits d'AspectJ semblent insuffisants. Par contre CaesarJ offre plusieurs niveaux de réutilisation (i.e. Interface de collaboration, implémentation). Les éléments de réutilisation dans JBoss AOP sont ceux de java ce qui explique

le mauvais résultat pour cette approche dans la note et les commentaires obtenus.

- **IDE et outil support**

L'IDE utilisé été Eclipse qui été adéquat avec les trois approches, les extensions plug-in pour AspectJ et CaesarJ sont faciles. Par contre JBoss AOP ne possède pas de plug-in récent, ce qui nécessite de faire quelques modification dans la class path avant l'utilisation de ce dernier.

L'éditeur et le programme de visualisation avec AspectJ sont plus faciles à utiliser et très sophistiqués, les relations entre les aspects et le reste du code sont très claires. Cela été très apprécié par les étudiants qui le trouvent mature et complet. Pour le plug-in CaesarJ, les étudiants ne le trouvent pas complet comme celui d'AspectJ, mais il est facile à utiliser aussi. Peu d'entre eux ont signalé quelques problèmes comme cet étudiant qui dit « «*Il y a quelques problèmes lors du refactoring des projets CaesarJ* »

- **Les patrons de conception**

Les nouveaux concepts introduits avec les trois approches simplifient la réalisation des différents patrons de conception, et cela été bien vu dans différents travaux comme [Hannemann, 2002] [Garcia, 2006] [Hachani, 2005].

Ces concepts sont capables de réduire radicalement l'implémentation de quelques patrons, cela peut être le cas pour AspectJ où l'utilisation de son mécanisme d'introduction réduit l'implémentation des deux patrons Adaptateur et Visiteur.

Nous avons noté quelques observations sur ces implémentations :

- Plusieurs castings dans le code final des patrons réalisés avec JBoss AOP, cela réduit la compréhension générale des implémentations.
- Les solutions proposées en AspectJ réduisent de manière significative le nombre de participants impliqués dans un patron. En effet, le mécanisme d'introduction offert par AspectJ nécessite de définir un seul aspect, alors qu'on doit définir trois différents éléments avec le mécanisme mix-in de JBoss AOP, qui sont l'interface contenant les éléments à introduire, la classe implémentant ces derniers, et enfin le fichier XML qui assure la liaison.
- le déploiement dynamique offert par JBoss AOP et CaesarJ, donne plus d'avantages pour certains patrons de conceptions. A titre d'exemple, le patron de conception Décorateur a bénéficié de cette façon de tisser pour ajouter ou supprimer des fonctionnalités supplémentaires dynamiquement.
- L'analyse de résultats de patron de conception Observateur montre que l'utilisation des aspects abstraits comme les seuls éléments de réutilisation n'est pas toujours suffisante. En effet, la gestion des différents observateurs dans l'aspect abstrait n'est pas toujours la même dans toutes les applications et nécessite d'être changée pour s'adapter à une application spécifique.
- Les implémentations CaesarJ des deux patrons : Fabrication abstraite et Pont sont essentiellement basées sur les concepts de CaesarJ, ce qui place le code entier de ces deux patrons dans le composant CaesarJ y compris les classes de base.

3 Synthèse générale

Comme nous l'avons exposé dans les sections précédentes, l'implémentation des 23 patrons de conception de GoF avec les trois approches : AspectJ, JBoss AOP et CaesarJ a pu soulever quelques avantages et inconvénients des trois approches. Par exemple la simplicité du mécanisme d'introduction d'AspectJ lui a permis d'occuper la première place dans l'attribut de taille, ce qui selon le modèle de Sant'Anna et al [Sant'Anna, 2003] améliore la compréhension des différentes implémentations de cette approche.

JBoss AOP a le privilège d'utiliser les instructions de Java, cependant l'utilisation des fichiers XML peut influencer le temps d'exécution des différentes implémentations surtout lors de l'utilisation des intercepteurs.

CaesarJ assure une bonne cohésion, cela été bien clair dans l'implémentation des 22 patrons de conception et peut être expliqué par le fait que l'idée de base de cette approche consiste à assembler les classes travaillant ensemble dans des unités spécifiques. Par contre dans l'attribut de couplage, les implémentations de CaesarJ étaient plus couplées que celles d'AspectJ, surtout si on prend on compte le couplage produit entre les classes internes. Ce genre de couplage peut être vu comme un inconvénient pour CaesarJ, puisque il n'est pas clair et il nécessite plus d'attention de la part des développeurs.

Il est aussi important de noter qu'AspectJ utilise moins de composants dans la réalisation des différents patrons de conception comparé à JBoss AOP et CaesarJ. Cependant, et contrairement à CaesarJ qui se focalise sur l'utilisation de la structure, AspectJ se base sur les événements dynamiques, et utilise plus de points de coupure. Cela peut expliquer les bons résultats de couplage d'AspectJ avec les patrons de comportement et CaesarJ avec les patrons de structure.

Notre comparaison quantitative a pu valider quelques remarques observées dans des études qualitatives. Par exemple le manque de cohésion dans quelques solutions AspectJ, a pu confirmer quantitativement quelques observations soulevées par Mezini et Ostermann dans [Mezini, 2003], comme le manque de soutien pour la multi-abstraction avec AspectJ comparé à CaesarJ. Ou l'utilisation exagérée des points de coupure par AspectJ comme signalé par Sousa et Monteiro dans [Sousa, 2008].

La comparaison qualitative, nous a permis de prendre des remarques sur la compréhension générale, les concepts, les outils supports de chaque approche comme le manque de documentations pour CaesarJ et JBoss AOP, l'absence de plug-in pour JBoss AOP, la difficulté du composant CaesarJ, l'absence d'éléments suffisants pour la réutilisation pour AspectJ et JBoss AOP.

De même, l'implémentation des 23 patrons de conception nous a permis de donner des lignes directrices aidant les nouveaux utilisateurs de la programmation orientée aspect.

Le tableau 14 résume quelques remarques observées dans notre étude comparative.

Tableau 14 Synthèse générale.

			AspectJ	JBoss AOP	CaesarJ
Comparaison quantitative	Métriques étendues de la POO	Couplage	Points forts : - Occupe la première place. - Couplage clair, réside dans les aspects concrets.	Points faibles : -Couplage réside dans les fichiers XML, ce qui demande plus d'attention de la part des développeurs.	Point faible : - Couplage produit entre les classes internes.
		Cohésion	Point faible : -Manque de soutien pour la multi-abstraction.	Point faible : - Pas de nouveaux éléments pour la réutilisation.	Points forts : - Supporte la muli-abstraction.
		Taille	Points forts : - Mécanisme d'introduction simple à mettre en œuvre.	Points faibles : -Mécanisme d'introduction moins souple et demande trois éléments pour son implémentation ainsi que des attributs supplémentaires. - Utilisation des castings.	Points faibles : - Composant CaesarJ utilise différents éléments dans sa réalisation.
	Métriques spécifique pour la POA		Points forts : -Utilise moins de composants. Point faible : -Utilise plus de point de coupure.	Points forts : - Utilisation moyenne des points de coupure.	Points forts : -Utilise moins de point de coupure.
	Métrique de performance		Points forts : -Temps d'exécution moyen.	Points faibles : -Temps d'exécution élevé, surtout avec les patrons qui utilisent les intercepteurs.	Points forts : - Temps d'exécution moyen.

<table>
<tr><td rowspan="4">Comparaison quantitative</td><td rowspan="2">Remarques générales sur l'approche</td><td rowspan="2">Points forts :
- Documentation suffisante.
- Plug-in complet même pour la visualisation</td><td>Point forts :
-Reste en java pur.</td><td>Point fort :
-Plug-in simple et facile à maitrisé.
- Utilise les mêmes points de coupure d'AspectJ (Facilite la migration)</td></tr>
<tr><td>Points faibles :
-Manque de documentation
-Manque d'outils support.
- Syntaxe différente pour les points de coupure.</td><td>Points faibles :
- Documentation insuffisante.
- Compréhension difficile des nouveaux concepts introduit par CaesarJ.</td></tr>
<tr><td rowspan="2">Remarques sur l'implémentation des patrons de conception</td><td>Points forts :
- Implémentation simple et compréhensible.</td><td>Points forts :
- Déploiement dynamique aide dans l'implémentation de quelques patrons.</td><td>Points forts :
- Déploiement dynamique aide dans l'implémentation de quelques patrons.
- Implémentation bien organisée dans le composant CaesarJ.</td></tr>
<tr><td>Points faible :
- Déploiement statique</td><td>Points faibles :
- Relations entre les classes aspect et les classes de base nécessite plus d'attention.</td><td>Points faible :
- Code entier de quelques patrons dans le composant CaesarJ y compris les classes de base.</td></tr>
</table>

4 Vers un outil générique d'extraction des métriques

Comme souligné précédemment, les métriques logicielles sont très importantes pour quantifier l'effort et le coût de développement et de maintenance d'un logiciel. Elles permettent aussi de comparer diverses versions d'une même application et nous renseignent sur des paramètres clés qui peuvent nous orienter vers les points faibles d'un logiciel. Cependant, avec l'évolution constante des paradigmes, les outils d'évaluation des métriques deviennent rapidement obsolètes et nécessitent d'être mis à jour.

Dans notre étude comparative, nous avons eu des difficultés pour trouver ou travailler avec un outil mature et complet pour l'extraction de différentes métriques.

Pour faire face à cette situation, nous proposons dans cette partie une démarche préliminaire pour la construction d'un évaluateur générique capable d'évaluer des

métriques logicielles adaptées à n'importe quel langage orienté aspect en utilisant des représentations universelles des programmes sous forme de fichier XML et ensuite l'expression des métriques sous forme de requêtes XQuery.

4.1 Pourquoi utiliser XML ?

XML (eXtensible Markup Language) un langage à balise étendu, est en quelque sorte un langage HTML amélioré permettant de définir de nouvelles balises. La force de XML réside dans sa capacité à pouvoir décrire n'importe quel domaine de données grâce à son extensibilité..

La simplicité, la lisibilité, et l'universalité sont les principaux facteurs qui rendent XML un standard pour générer une représentation universelle de code source.

Un document XML ne nécessite pas la pré-analyse pour obtenir des informations sur la structure d'un programme. XML permet de créer une structure similaire à un arbre de syntaxe abstraite. De même une représentation XML du code source offre la possibilité d'effectuer des requêtes qui interagissent avec la structure complète du programme. Pour cette tâche XML offre des outils utiles comme XQuery qui facilitent la construction des requêtes et ainsi d'exprimer des métriques très complexes.

4.2 Aspect conceptuel de l'approche

Nous proposons d'utiliser le langage XML pour l'évaluation des programmes orientés aspects, où la représentation universelle offerte par ce langage facilite l'extraction des différentes métriques. L'architecture générale de l'approche est basée sur les deux étapes suivantes:

- **Etape 1**:construction des fichiers XML (du code source vers XML)
- **Etape 2**: extraction des métriques

La figure 37 présente l'architecture générale de l'approche :

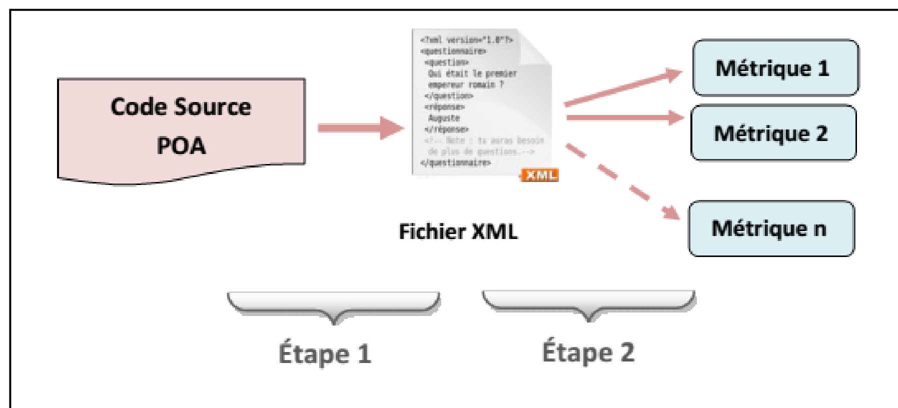

Figure 37 L'architecture générale de l'approche

Dans ce qui suit nous allons détailler les deux étapes :

4.2.1 Construction des fichiers XML (du code source vers XML)

Notre approche est basée essentiellement sur la création de représentations uniformes des programmes à analyser en utilisant les fichiers XML. Dans cette création nous avons pris en compte tous les éléments nécessaires dans la définition des métriques de couplage de cohésion et de taille.

Les formes proposées pour les classes est les aspects ne sont pas identiques, vu que chaque composant a des concepts différents des autres, les deux listings suivants présentent les formes XML proposées respectivement pour les classes et les aspects.

4.2.1.1 *Forme XML proposée pour les classes*

En utilisant un analyseur syntaxique basé sur le patron de conception Visiteur, et à l'aide de l'utilisation d'API JDOM nous avons réussi à réaliser un plug-in qui prend en entré les classes java qu'on souhaite analyser, ce plug-in balise tous les éléments internes ainsi que les relations entre les classes dans une application donnée, le listing suivant montre la forme XML proposée pour les classes.

```
7    <class name="classe1">
8
9    <implements_interfaces>❶
10   <implements_interface name="interface1"></implements_interface>
11   </implements_interfaces>
12
13   < inheritance_classes>❷
14   <inheritance_class name="classe2"></inheritance_class>
15   <inheritance_class name="classe4"></inheritance_class>
16   </inheritance_classes>
17
18   <classes_call>❸
19   <class_call name="classe5"></class_call>
20   </classes_call>
21
22   <classes_children>❹
23   < childname="classe6"></child>
24   </classes_children>
25
26   <attributs>❺
27   <attribut name="i"/>
28   <attribut name="k"/>
29   <attribut name="m"/>
30   </attributs>
31
32   <methods>❻
33
34   <method name="methode1">
35   <parameters>
36   <parametre name ="j"/>
37   <parametre name ="k"/>
38   </parameters>
39   <classes_call>
40   <class_call name ="classe5"></class_call>
41   </classes_call>
42   <attributs_use>
43   <attribut name ="i"/>
44   </attributs_use>
45   </method>
46
47   <method name ="methode2">
48   <parameters></parameters>
49   </classes_call>
50   <class_call name ="classe10"></class_call>
51   </classes_call>
52   <attributs_use >
53   <attribut name ="i"/>
54   </attributs_use>
55   </method>
56   </methods>
57
58   </class>
```

Chaque balise a une influence sur une ou plusieurs métriques :

- **La balise <implements_interfaces>**
  Cette balise contient l'ensemble des interfaces implémentées par la classe, l'attribut de couplage ***DIT*** utilise cette balise pour atteindre la racine de la classe.

- **La balise < inheritance_classes>**
  Cette balise contient l'ensemble des classes parents de la classe étudiée, les deux métriques de couplage ***DIT*** et ***CBC*** utilisent cette balise dans leurs comptes.

- **La balise <classes_call>**
  Cette balise contient les classes appelées par la classe étudiée, l'attribut de couplage ***CBC*** utilise cette balise pour compter le couplage de la classe étudiée avec les autres classes de l'application.

- **La balise <classes_children>**
  Cette balise contient les classes filles de la classe étudiée, l'attribut de taille ***NOC*** utilise cette balise pour compter l'ensemble des classes filles.

- **La balise <attributs>**
  Cette balise contient l'ensemble d'attributs de la classe étudiée, les deux attributs de taille et de cohésion respectivement : ***NOA*** et ***LCOO*** utilisent cette balise dans leurs comptes.

- **La balise <methods>**
  Cette balise contient l'ensemble des méthodes ainsi que les informations de chaque méthode de la classe étudiée, les attributs de couplage, de taille et de cohésion respectivement : ***CBC***, ***WMC***, ***NOM*** et ***LCOO*** utilisent cette balise dans leurs comptes.

4.2.1.2 Forme XML proposée pour les Aspects

Afin d'adapter le plug-in crée précédemment pour les aspects nous avons ajouté ***aspectjtools-1.5.4*** au *Librairies* du plug-in. Malheureusement, par manque de temps, nous n'avons pas pu faire marcher l'analyseur syntaxique des aspects ce qui nous a obligé de créer les fichiers XML des aspects manuellement.

Après avoir étudié en détails les langages AspectJ et JBoss AOP, et les comportements possibles d'un aspect dans ces deux langages, nous avons balisé manuellement toutes les éléments nécessaires dans le compte des métriques.

Ces éléments concernent :

- L'introduction de nouveaux composants sur les classes de l'application : comme l'ajout de méthodes ou d'attributs.
- L'altération de l'arbre d'héritage des classes d'application

L'influence de ces éléments sur les attributs de mesure est présentée dans la figure 38.

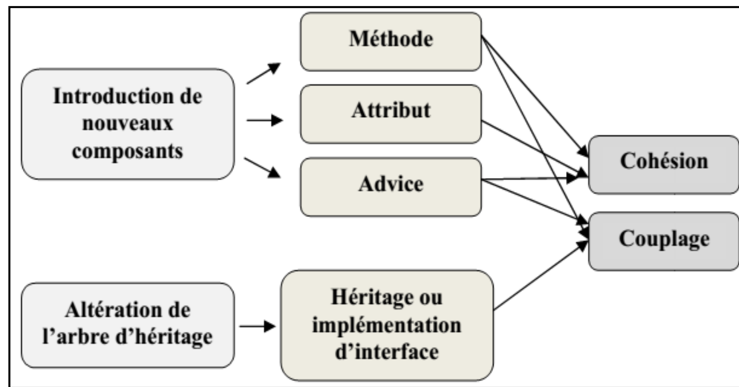

Figure 38 L'influence d'éléments aspect sur les attributs de mesure

Le listing suivant montre la forme XML proposée pour les aspects :

```
59   <aspect name="aspect1">
60
61   < inheritance_aspects>❶
62   <inheritance_aspect name="introduction"/></inheritance_aspect>
63   </inheritance_aspects>
64
65   <aspect_attributs/>❷
66
67   <declare_parents>❸
68   <declare_parent      class_name="classe4"      interface_name="interface1"
     type="implements"/>
69   <declare_parent class_name="classe7" class_name="class9" type="extends"/>
70
71   </declare_parents>
72
73   <add_attributs>❹
74   <add_attribut class_name="classe1" attribute_name="m"/>
75   </add_attributs>
76
77   <add_methods>❺
78   <add_method class_name="classe1" method_name="intro">
79   <parametres/>
80   <classes_call>
81   <class_call name ="classe10"></class_call>
82   </classes_call>
83   <attributs_use >
84   <attribut name ="i"/>
85   </attributs_use>
86   </add_method>
87   </add_methods>
88
89
90   <advices>❻
91   <advice type="before" list_classes_and_methods="classe4.methode5">
92   <parametres/>
93   <classes_call/>
94   <attributs_use/>
95   </advice>
96
97   <advice type="after" list_classes_and_methods="classe5.methode6">
98   <parametres/>
99   <classes_call/>
100  <attributs_use/>
101  </advice>
102  < /advices>
103
104  </aspect>
```

Chaque balise a une influence sur une ou plusieurs métriques :

- **La balise < inheritance_aspects >**
  Cette balise contient l'ensemble des aspects parents de l'aspect étudié, l'attribut de couplage utilise cette balise dans le compte de la métrique ***DIT***.

- **La balise < aspect_attributs >**
  Cette balise contient l'ensemble des attributs de l'aspect étudié, cette balise est utilisée par l'attribut de taille dans le compte de la métrique ***NOA***.

- **La balise < declare_parents >**
  Cette balise contient les informations nécessaires pour le mécanisme d'introduction, plus précisément dans l'altération de l'arbre d'héritage d'une ou plusieurs classes, on trouve alors dans cette balise la classe visée par l'introduction, l'interface à implémenter ou la classe d'héritage, ainsi que le type d'altération (i.e. implements ou extends).

- **La balise < add_attributs >**
  Cette balise contient l'ensemble des attributs à introduire. On trouve dans cette balise la classe visée, ainsi que les attributs introduits.

- **La balise < add_methods >**
  Cette balise contient l'ensemble des méthodes à introduire, on trouve dans cette balise la classe visée, ainsi que les méthodes introduites, avec les informations nécessaires pour chaque méthode.

- **La balise < advices>**
  Cette balise contient l'ensemble des advices de chaque aspect étudié, la forme des advices ressemble à celle des méthodes.

4.2.2 Tissage des fichiers XML

Comme nous l'avons déjà expliqué, les aspects peuvent modifier de façon transparente les programmes orientés objet. Ils peuvent injecter du code, introduire de nouvelles variables ou de nouvelles méthodes, ou encore modifier la hiérarchie des classes et des interfaces du programme. Cette modification à une sémantique particulière et un impact bien précis sur le calcul des métriques. C'est pourquoi nous estimons qu'il est nécessaire de prendre en compte la phase de tissage dans le calcul des métriques.

L'étape tissage des fichiers XML désigne alors la réalisation d'un fichier XML général de toute l'application à partir des fichiers XML créés précédemment, le mot tissage indique l'intégration et l'influence des balises des aspects sur les balises des classes.

Dans notre travail nous avons pris en compte l'influence des aspects sur les attributs de couplage et de cohésion, Par exemple l'introduction d'une méthode sur une classe donnée peut altérer l'attribut de couplage de cette dernière, car si la méthode introduite contient dans la balise**<classes_call>**des nouvelles classes, cela rend la classe visée par l'introduction couplée avec l'ensemble de ces classes.

En revanche, nous n'avons pas pris en compte l'influence de ces introductions sur l'attribut de taille, car on trouve que même si cette influence touche l'occupation de l'espace mémoire et le temps d'exécution elle ne donne pas une image juste sur l'effort de développement, et de la maintenance de ces programmes, et c'est ce qui nous intéresse plus dans ce genre d'évaluation.

Le schéma de la figure 39 résume la partie de construction et tissage des fichiers XML.

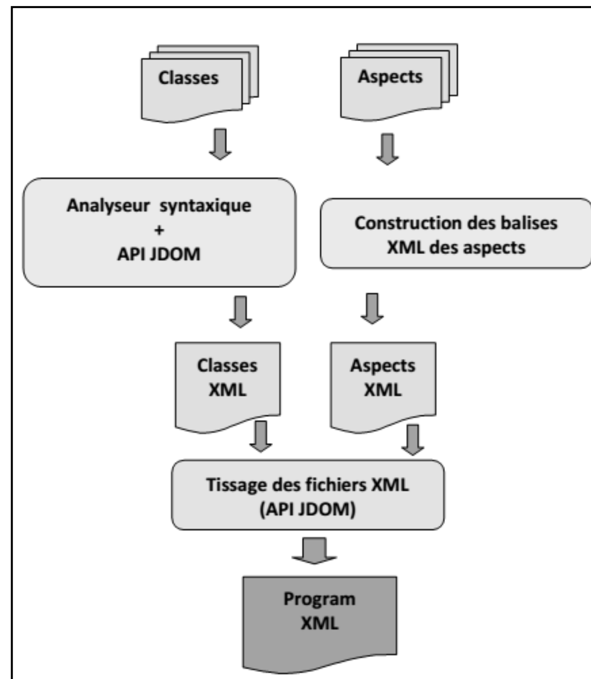

Figure 39 Tissage des fichiers XML

4.3 Extraction des métriques

La stratégie d'extraction des métriques que nous avons suivie est basée sur l'utilisation du langage de requête XQuery qui permet d'interroger le fichier XML général créé dans la première étape.

Exemple de requête sur la métrique NOA :

RFC.xq | DIT.xq | NAD.xq | NMD.xq | NOC.xq

```
for $c in doc('project_tissage.xml')/project/class
let $attribut := $c/attributs/attribut
let $n := count($attribut)
return
<classe>  {$c/@nom}  possede  {$n}  attributs</classe>
```

Figure 40 Requête sur la métrique NOA

5 Conclusion

Dans ce chapitre nous avons donné un aperçu sur les métriques logicielles et l'influence de la programmation orientée aspect sur ces métriques.

Par la suite nous avons présenté la démarche et les résultats de notre étude comparative qui inclue deux sortes de comparaisons, la première est quantitative et se base sur des métriques structurelles et de performances. Les résultats de cette comparaison ont dévoilé quelques avantages et inconvénients de trois approches, comme la puissance du mécanisme d'introduction d'AspectJ, le fort couplage entre les classes internes de CaesarJ, ou même le temps d'exécution des implémentations réalisées avec JBoss AOP.

L'autre comparaison est qualitative, elle est basée sur des observations faites lors de l'implémentation des patrons de conception, et pour lesquelles ils n'existent pas des mesures quantitatives.

La fin de ce chapitre présente une démarche préliminaire pour la réalisation d'un outil générique d'extraction des métriques logicielles, il se base sur l'utilisation des deux langages XML et XQuery. Cette démarche ne représente qu'un début de travail pour des perspectives futures.

CONCLUSION ET PERSPECTIVES …

La programmation orientée aspect a prouvé son efficacité dans plusieurs études expérimentales, avec un nouveau lot de concepts et mécanismes, la POA a pu remédier à certains problèmes observés avec la programmation orientée objet. Cependant l'apparition de plusieurs approches supportant ce nouveau paradigme dans un temps aussi court a dépassé la cadence des études comparatives nécessaires de ces approches. En effet, on trouve que la majorité de ces études prennent AspectJ comme le représentant majeur de la programmation orientée aspect. Ce qui a engendré une ambigüité au niveau de la différence entre les langages orientés aspect et a diminué l'utilisation et l'exploitation des capacités des autres langages.

Dans la présente thèse, nous avons implémenté les 23 patrons de conception de Gang of Four avec les trois langages orientés aspect que sont : AspectJ, JBoss AOP et CaesarJ. Afin d'être juste avec les trois langages nous avons demandé l'aide d'un groupe de 12 étudiants Master II, Option : Ingénierie des Logiciels Complexes. L'utilisation de ces implémentations comme des benchmarks hypothétiques nous a permis par la suite de faire deux sortes de comparaisons : quantitative et qualitative.

Dans la comparaison quantitative nous avons travaillé avec des métriques structurelles et de performance. Suivant la nature de ces métriques nous avons effectué trois sous-comparaisons :

- Une sous-comparaison basée sur des métriques objets influencées par la programmation orientée aspect : les résultats de cette comparaison reflète trois attributs majeurs de la qualité logicielle que sont : le couplage, la cohésion et la taille,
  - **Couplage** : Dans cet attribut, nous avons défini deux sortes de couplage avec CaesarJ. Le premier concerne juste le couplage produit entre les éléments du composant CaesarJ (Interface de collaboration, partie implémentation, partie binding, etc.), le deuxième prend en charge le couplage produit entre les cclasses internes formant ces éléments. Ce dernier a donné une valeur très élevée, ce qui a relégué CaesarJ à la dernière place.
  - **Cohésion** : contrairement au couplage CaesarJ a donné de bons résultats dans l'attribut de cohésion, en effet chaque élément du composant CaesarJ est responsable d'une tache spécifique. AspectJ a montré quelques faiblesses dans la cohésion, et la séparation des préoccupations en plusieurs niveaux ce qui a donné plus d'avantage à CaesarJ.
  - **Taille** : La simplicité du mécanisme d'introduction d'AspectJ, était le principal facteur pour qu'il prenne la première place dans cet attribut. En effet, avec une simple déclaration inter-type AspectJ peut ajouter, modifier, supprimer une méthode ou altérer une hiérarchie, contrairement à CaesarJ ou JBoss AOP qui nécessitent trois différents éléments pour effectuer une telle tâche.
- Une sous-comparaison basée sur des métriques propre à la programmation orientée aspect : les résultats de cette comparaison concernent les nouveaux

concepts la programmation orientée aspect, comme les points de coupures, les composants ajoutés, etc. Dans cette comparaison nous avons remarqué l'appui d'AspectJ et JBoss AOP sur les points de coupures et CaesarJ sur la structure. De même nous avons observé qu'avec un nombre réduits de composants AspectJ a pu réaliser les différents patrons de conception.

- Une sous-comparaison basée sur une métrique de performance : les résultats de cette comparaison concernent le temps d'exécution des différentes implémentations, ce qui nous laisse prendre une idée sur les outils support des différents langages. Les résultats de cette métrique ont montré un point faible pour JBoss AOP est le fait qu'il prenne plus de temps dans le chargement des classes, surtout quand il s'agit de programmes qui utilisent des intercepteurs.

Dans la comparaison qualitative, les étudiants ont été invités à remplir des formulaires de rétroaction comprenant différentes questions divisées en quatre catégories: la compréhensibilité des approches, les concepts des approches, la maturité des outils soutenant les approches et les difficultés rencontrées lors de l'implémentation des différents modèles de conception.

Chaque question nécessite une note et offre un espace pour des remarques au cas où l'étudiant souhaiterait laisser un commentaire.

Le calcul des notes et l'analyse des observations laissées par les étudiants ont montrés quelques avantages et inconvénients pour chaque langages comme le manque de documentations pour CaesarJ et JBoss AOP, l'absence de plug-in pour JBoss AOP, la difficulté du composant CaesarJ, l'absence d'éléments suffisants pour la réutilisation pour AspectJ et JBoss AOP.

L'implémentation des patrons de conception avec les trois langages, et l'étude comparative nous ont permis de donner à la fin de cette thèse des lignes directrices aidant les nouveaux utilisateurs de ces langages.

Comme première perspective, ce travail peut être élargi pour contenir d'autres approches et métriques, et surtout des systèmes plus larges pour avoir des idées sur l'efficacité de ces approches dans le monde pratique.

La seconde perspective concerne la proposition de patrons spécifiques à l'approche orientée aspects. Cette perspective n'est pas nouvelle mais jusqu'à présent il n'existe pas un catalogue qui englobe des patrons destinées pour le monde orienté aspect.

Une autre perspective concerne la visualisation des programmes orientés aspect. En effet, après notre étude, nous pensons que l'amélioration des visualisations des programmes aspect est nécessaire. Cette visualisation aide à comprendre le fonctionnement des aspects, surtout quand il s'agit de logiciels de grande taille et de grande complexité.

La dernière perspective que nous proposons est la réalisation d'un outil générique complet et mature pour le calcul des métriques. Dans notre étude nous avons été amenés dans la plupart du temps à faire les calculs des métriques manuellement. Un outil générique permettra d'envisager le calcul de nombreuses métriques sur des logiciels volumineux.

REFERENCES

[Abiteboul, 1991]	S. Abiteboul S, A.J Bonner, “Objects and Views”, In Proceedings of the ACM SIGMOD International Conference on Management of Data, Vol: 20, pp: 238-247, New York, USA 1991.
[Aksit, 1998]	M. Aksit and B. Tekinerdogan, “Aspect-oriented programming using composition-filters”, In ECOOP Workshops, pp: 435, Brussels, Belgium, 1998.
[Amirat, 2007]	A. Amirat, “ Une Approche Hybride pour la Séparation des Préoccupations avec Résolution de Conflits durant l’Ingénierie des Besoins ”, Ph.D. Thesis, departement of computer science, University of Badji mokhtar-Annaba, Algeria, 2007.
[Aracic, 2006]	I. Aracic, V. Gasiunas, M. Mezini, K. Ostermann, “Overview of CaesarJ. Transactions on Aspect-Oriented Software Development I”, LNCS, Vol: 3880, pp:135-173. 2006.
[AspectJ, Web]	The official site of AspectJ : http://www.eclipse.org/aspectj/
[AspectWerkz, Web]	The official site of AspectWerkz : http://aspectwerkz.codehaus.org/
[Baltus, 2002]	J. Baltus, “La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué”, Mini-Workshop: Systèmes Coopératifs. Matière Approfondie, Institut d'informatique, Namur, 2002.
[Bardou, 1996]	D. Bardou, C. Cony, “Split Objects: A Disciplined Use of Delegation within Objects”, In Proceedings of ACM OOPSLA'96 OOPSLA 96, Vol 31, pp: 122-137, 1996.
[Bartolomei, 2006]	T.T. Bartolomei, A. Garcia, C. Sant’Anna, E. Figueiredo, “Towards a Unified Coupling Framework for measuring Aspect-Oriented Programs”, . In: 3rd International Workshop on Software Quality Assurance, pp: 46 -53, USA, 2006.
[Basili, 1994]	V. Basili, G. Caldiera, H. Rombach, H. “The Goal Question Metric Approach”, Chapter in Encyclopedia of Software Engineering, Wiley, 1994.
[Belhanafi, 2005]	N. Belhanafi, C. Taconet, G. Bernard, “ Mécanismes de réactivité au contexte dans un intergiciel orienté composant ”, In Proceedings of Nouvelles Techniques de la Répartition (NOTERE), Gatineau, Québec, Canada, 2005.
[Bergmans, 1994]	L.M.J. Bergmans, “Composing Concurrent Objects, Applying Composition filters for the Development and Reuse of Concurent Object-Oriented programs”, Ph.D. Thesis, departement of computer science, University of Twente, The Netherlands, 1994.
[Bertoa, 2002]	M. Bertoa, A. Vallecillo, “Quality Attributes for COTS

	Components", In the Proceedings of 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), pp: 128-144, Spain, 2002.
[Bertrand, 2004]	R. Bertrand, J.G. Blais, G. Raîche," Modèles de mesure: l'apport de la théorie des réponses aux items ", Addison illustrée, 2004.
[Bodmer, 2006]	F. Bodmer, T. Maret, "AOP Tools Comparison", In Mini-Proceedings of the Master Seminar - Advanced Software Engineering Topics: Aspect Oriented Programming, pp : 77-106, Switzerland, 2006.
[Braz, 2009]	S.A.E.M Braz, "A Qualitative Assessment of Modularity in CaesarJ components based on Implementations of Design Patterns", M.Sc. Thesis, Nova de lisboa University, 2009.
[CaesarJ, Web]	The official site of CaesarJ : http://www.caesarj.org/
[Chang, 2008]	C.W. Chang, C.R Wu, H.L. Lin, "Integrating Fuzzy Theory and Hierarchy Concepts to Evaluate Software Quality", Software Quality journal, Vol : 16(2), pp:263-276, 2008.
[Chen, 2010]	X. Chen, N. Ye, W. Ding, "A Formal Approach to Analyzing Interference Problems in Aspect-Oriented Designs", Unifying Theories of Programming, Vol: 6445, 2010, pp: 157-171, 2010.
[Chidamber, 1994]	S.R. Chidamber, C. Kemerer, "A metrics suite for object oriented design", IEEE Transactions on Software Engineering journal, Vol: 20(6), pp: 476-493, 1994.
[Christopher, 1977]	A. Christopher, S. Ishikawa, M. Silverstein, M. Jcobson, I. Fiksdkhal-King, S. Angel, " A Pattern language ", Oxford University Press, 1977.
[Debboub, 2013]	S. Debboub, D. Meslati, "Quantitative and qualitative evaluation of AspectJ, JBoss AOP and CaesarJ, using Gang-of-Four design patterns", International Journal of Software Engineering and Its Applications (IJSEIA), Vol: 7(6), pp: 157-174, 2013.
[Debboub, 2014]	S. Debboub, D. Meslati, "Study of advanced separation of concerns approaches using the GoF design patterns: A quantitative and qualitative comparison", Information and Software Technology journal, Vol: 56(10), pp: 1345-1359, 2014.
[Denier, 2007]	S. Denier, " Expression et composition des motifs de conception avec les aspects", Ph.D. Thesis, departement of computer science, University of Nantes, France, 2007.
[Edward, 1998]	V.B. Edward, "Metrics for object-oriented software engineering", Site: http://www.ipipan.gda.pl/~marek/objects/TOA/moose.html, 1998.
[Ettinger , 2004]	E. Ettinger, M. Verbaere, "Untangling: a slice extraction refactoring". In Proceeding of AOSD 04, pp :93-101, 2004.

[Fenton, 1997] N. Fenton, S. Pfleeger, "Software Metrics: A Rigorous and Practical Approach", Thomson Computer Press(2.ed), 1997.

[Figueiredo, 2006] E. Figueiredo, A. Garcia, C. Lucena, "AJATO : an AspectJ Assessment Tool", In : proceedings ECOOP.06, LNCS, Vol: 4067, 2006.

[Filman, 2005] R. Filman, T. Elrad, S. Clarke, M. Akşit, "Aspect-Oriented Software Development". Addison-Wesley, 2005.

[Filters, Web] The official site of composition filtres : http://trese.cs.utwente.nl/oldhtml/composition_filters/

[Florijn, 1997] G. Florijn,M. Meijers,P.V. Winsen, "Tool support for object-oriented patterns", In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97). Lecture Notes in Computer Science, Vol: 1241, pp: 472-495, 1997.

[Gamma, 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns, Elements of reusable Object-Oriented Software", Addison-Wesley Publishing Company, 1995.

[Garcia, 2003] A.F. Garcia, C.N Sant'Anna, C.F.G. Chavez, V.T. Silva, C.J.P Lucena, A. Staa,"Agents and Objects: An Empirical Study on Software Engineering", Technical Report 06-03, Computer Science Department, PUC-Rio, 2003.

[Garcia, 2006] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. Staa, "Modularizing Design Patterns with Aspects: A Quantitative Study", LNCS TAOSD I, Springer, Vol: 3880, pp: 36-74, 2006.

[Gottlob, 1996] G. Gottlob, M. Schrefl, "Extending Object-Oriented Systems with Roles", ACM Transactions on Information Systems (TOIS), Vol: 14(3), pp: 268-296 , 1996.

[Hachani, 2006] O. Hachani, "Patrons de conception à base d'aspects pour l'ingénierie des systèmes d'information par réutilisation", Ph.D. Thesis, departement of computer science, University of Joseph fourier, Grenoble I, France, 2006.

[Hachani, 2005] O. Hachani, D. Bardou, Organisation et catégorisation de patrons de conception par aspects, In Proceedings INFORSID, France, 2005.

[Hannemann, 2002] J. Hannemann, G. Kiczales, "Design Pattern Implementation in Java and AspectJ". In Proceedings of OOPSLA, ACM SIGPLAN Notices, Vol: 37(11), pp: 161-173, 2002.

[ISO/IEC 9126, 2001] ISO/IEC 9126, Institute of Electrical and Electronics Engineering, Part 1, 2, 3, 4: Quality Model, 2001

[JBoss AOP, Document1] JBoss AOP Reference Documentation:

Site : http://docs.jboss.org/jbossaop/docs/2.0.0.GA/docs/aspect-framework/userguide/en/html/index.html

[JBoss AOP, Document2] JBoss AOP Reference Documentation, Web Document : https://docs.jboss.org/jbossaop/docs/2.0.0.GA/docs/aspect-framework/reference/en/pdf/jbossaop_reference.pdf

[JBoss AOP, Web] The official site of JBoss AOP : http://www.jboss.org/jbossaop/

[Katz, 2003] 1. Shmuel Katz and Marcelo Sihman. "Aspect-validation using model-checking", Verification: Theory and Practice, Vol: 2772, pp: 373-394, 2003.

[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, LNCS, Vol: 1241, pp: 220-242, 1997.

[Kumar, 2010] A. Kumar, "Analysis and Design of Metrics for Aspect-Oriented Systems", Ph.D. Thesis, School of Mathematics and Computer Applications, Thapar University, India., 2010.

[Laddad, 2009] R. Laddad, "AspectJ in Action, second ed", Manning Publications, 2009

[Lieberherr 1996] K. Lieberherr, "Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns", PWS Publishing Company, 1996.

[Lieberherr, 1994] K.J. Lieberherr, "The Art of Growing Adaptive Object-Oriented Software". PWS Publishing Compagny, Boston, 1995.

[Lorenz, 1994] Mark Lorenz et Jeff Kidd. "Object-Oriented Software Metrics: A Practical Guide", Addison Prentice-Hall, 1994.

[Lorenz, 1998] D.H. Lorenz, "Visitor Beans: An Aspect-Oriented Pattern". In ECOOP'98 Workshop on Aspect Oriented Programming, 1998.

[Meslati, 2006] D. Meslati, "MAGE : Une Approche Ontogénétique de l'Evolution dans les Systèmes Logiciels Critiques et Embarquées", Ph.D. Thesis, departement of computer science, University of Badji mokhtar-Annaba, Algeria, 2006.

[Meyer, 1997] B. Meyer, "Object -Oriented Software Construction". Prentice Hall (2.ed), 1997.

[Mezini, 2003] M. Mezini, K. Ostermann, "Conquering Aspects with Caesar", In Proceeding of international conference on Aspect-oriented software development AOSD'03, pp: 90-99, USA, 2003.

[Mostefaoui, 2007] F. Mostefaoui, J. Vashon, "Design-level Detection of Interactions in Aspect UML models using Alloy", Journal of Object Technology, Vol: 6(7), pp: 137-16, 2007.

[Munson, 2003] J.C. Munson, "Software Engineering Measurement", Auerbach Publications, 2003.

[Noda, 2001] N. Noda, T. Kishi, "Implementing Design Patterns Using Advanced Separation of Concerns". In OOPSLA 2001 Workshop on ASoC in OOS, 2001

[Nordberg, 2001] M.E. Nordberg, "Aspect-Oriented Dependency Inversion". In OOPSLA 2001 Workshop on ASoC in OOS, 2001.

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

[Parnas, 1972] D. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules", Communications of the ACM, Vol :15 (12), pp: 1053-1058, 1972.

[Pawlak, 2001] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, "JAC: A flexible solution for aspect-oriented programming in Java", In Proceedings REFLECTION, LNCS, Vol: 2192, pp: 1–24, 2001.

[Pawlak, 2004] R. Pawlak, J.P. Retaillé, and L. Seinturier, " Programmation orientée aspect pour Java / J2EE ", Addison Eyrolles, 2004.

[Quintian, 2004] L. Quintian, "JADAPT : un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet", Ph.D. Thesis, departement of computer science, University of Nice Sophia-Antipolis, France, 2004.

[Rosenberg, 1998] L. Rosenberg, "Applying and interpreting object oriented metrics", Site :

https://www.cs.purdue.edu/homes/apm/courses/BITSC461-fall03/metrics-slides/nasa-rosenberg-study.html, 1998.

[Sant'Anna, 2003] C. Sant'Anna, A. Garcia, C. Chavez, C. Lucena, A. Staa, " On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework", In Proceedings of Brazilian Symposium on Software Engineering, Manaus, Brazil, 2003.

[Sharma, 2008] A. Sharma, R. Kumar, P. S. Grover, " Estimation of Quality for Software Components: an Empirical Approach", ACM SIGSOFT Software Engineering Notes, Vol: 33(6), pp:1-10, 2008

[Sommerville, 2001] L. Sommerville, "Software Engineering", Addison Wesley (9.ed), 2010.

[Sousa, 2008] E. Sousa, M. P Monteiro, An Exploratory Study of CaesarJ Based on Implementations of the Gang-of-Four patterns, Technical report FCT-UNL-DI-SWE-2008-01, New University of Lisbon, 2008.

[Tarr, 1999] P. Tarr, H. Ossher, S.M. Sutton, "N Degrees of Separation: Multidimensional Separation of Concerns", In Proceedings of the International conference on software engineering (ICSE'99), pp: 107-119, 1999.

ANNEXES

ANNEXE A

Dans cette annexe, nous allons présenter les différents résultats de notre comparaison quantitative avec les 23 patrons de conception de GoF (Tous ces résultats, ainsi que d'autres sont disponibles sur notre site web : http://www.debboub-soumeya.sitew.org/Curriculum_Vitae.B.htm#Research_Projects.C)

COMMANDE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	LOCC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	103	31	4	19	9	30	2	4	3	4	2	1	1	2
JBOSS AOP	145	43	0	27	11	50	2	0	1	4	7	0	1	1
CAESARJ	126	38	1	25	9	44/80	3	1	1	4	4	0	1	1
RESULTATS	A	A	B	A	AC	A	AB	B	BC	ABC	A	BC	ABC	BC
	A=4 /B= 1/C=1					A		B	A=3/B=5/C=5					

COMPOSITE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	LOCC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	77	21	42	16	2	16	2	42	0	3	3	0	0	0
JBOSS AOP	108	34	4	23	4	32	3	4	0	3	6	0	0	0
CAESARJ	119	34	8	21	3	24/44	3	8	0	3	4	0	0	0
RESULTATS	A	A	B	A	A	A	A	B	ABC	ABC	A	ABC	ABC	ABC
	A=5/B=1					A		B	A=6/B=5/C=5					

COR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	LOCC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	69	26	8	14	1	14	2	8	3	3	2	1	1	2
JBOSS AOP	104	28	8	15	2	24	2	8	1	3	4	0	1	1
CAESARJ	79	30	0	16	1	26/38	3	0	1	3	4	0	1	1
RESULTATS	A	A	C	A	AC	A	AB	C	BC	ABC	A	BC	ABC	BC
	A=4/B=1/C=1					A		C	A=3/B=5/C=5					

MEDIATEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	LOCC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	57	22	5	13	5	14	2	5	1	3	2	0	1	1
JBOSS AOP	92	25	0	16	6	42	2	0	1	3	6	0	1	1
CAESARJ	71	24	0	15	5	24/50	3	0	1	3	4	0	1	1
RESULTATS	A	A	BC	A	AC	A	AB	BC	ABC	ABC	A	ABC	ABC	ABC
	A=4/B=1/C =1					A		BC	A=6/B=5/C=5					

OBSERVATEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	83	33	2	18	4	22	2	10	5	4	3	1	2	3
JBOSS AOP	113	34	0	22	6	32	2	0	1	3	6	0	1	1
CAESARJ	108	39	4	25	4	24/70	3	0	1	3	4	0	1	1
RESULTATS	A	A	B	A	AC	A	AB	BC	BC	BC	A	BC	BC	BC
	A=4/B=1/C=1					A		BC	A=1/B=5/C=5					

POIDS MOUCHE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	61	19	1	10	4	8	2	1	1	2	2	0	1	1
JBOSS AOP	80	23	0	12	4	22	2	1	1	2	5	0	1	1
CAESARJ	70	21	4	11	4	18/32	3	0	1	2	4	0	1	1
RESULTATS	A	A	B	A	ABC	A	AB	C	ABC	ABC	A	ABC	ABC	ABC
	A=4/B=2/C=1					A		C	A=6/B=5/C=5					

ITERATEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	68	23	0	14	3	10	2	1	0	1	1	0	0	0
JBOSS AOP	81	27	0	17	4	14	2	1	0	1	3	0	0	0
CAESARJ	70	24	0	15	3	14/16	2	1	0	1	1	0	0	0
RESULTATS	A	A	ABC	A	AC	A	ABC	ABC	ABC	ABC	AC	ABC	ABC	ABC
	A=5/B=1/C=2					A		ABC	A=6/B=5/C=6					

MEMENTO	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	55	19	1	10	1	16	2	0	0	1	2	0	0	0
JBOSS AOP	63	19	0	10	1	18	2	0	0	1	3	0	0	0
CAESARJ	56	19	1	8	1	16/20	2	0	0	1	2	0	0	0
RESULTATS	A	ABC	B	C	ABC	A/AC	ABC	ABC	ABC	ABC	AC	ABC	ABC	ABC
	A=3/B=3/C=3					A		ABC	A=6/B=5/C=6					

PROTOTYPE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	56	18	3	13	2	10	2	0	0	2	2	0	0	0
JBOSS AOP	67	18	2	12	3	16	2	0	0	2	3	0	0	0
CAESARJ	52	17	3	12	2	12§20	2	0	0	2	2	0	0	0
RESULTATS	C	C	B	BC	AC	A	ABC	ABC	ABC	ABC	AC	ABC	ABC	ABC
	A=1/B=2/C=4					A		ABC	A=6/B=5/C=6					

SINGLETON	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	29	5	1	3	1	6	2	0	2	2	2	1	2	3
JBOSS AOP	32	7	0	4	1	4	1	1	1	2	3	0	1	1

CAESARJ	29	5	1	3	1	8	2	0	2	1	2	1	2	3
RESULTATS	AC	AC	B	AC	ABC	B	B	AC	B	C	AB	B	B	B
	A=4/B=2/C=4					B		AC	A=1/B=5/C=1					

ADAPTATEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	18	5	0	4	0	8	1	0	0	1	1	0	0	0
JBOSS AOP	30	7	0	5	1	10	2	0	0	1	2	0	0	0
CAESARJ	18	5	0	4	0	6§8	1	0	0	1	1	0	0	0
RESULTATS	AC	AC	ABC	AC	AC	AC/C	AC	ABC	ABC	ABC	AC	ABC	ABC	ABC
	A=5/B=1/C=5					AC/C		ABC	A=6/B=5/C=6					

DECORATEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	28	11	1	6	1	6	2	0	1	1	2	0	1	1
JBOSS AOP	28	7	0	4	0	6	1	0	1	1	2	0	1	1
CAESARJ	18	6	0	3	0	6	1	0	1	1	1	0	1	1
RESULTATS	C	C	BC	C	BC	ABC	BC	ABC	ABC	ABC	C	ABC	ABC	ABC
	B=2/C=5					BC		ABC	A=5/B=5/C=6					

PROXY	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	77	38	3	13	2	20	2	1	3	3	4	1	4	5
JBOSS AOP	114	39	0	14	3	16	2	1	1	1	4	0	3	3
CAESARJ	90	45	4	16	2	18/34	3	1	1	1	4	0	3	3
RESULTATS	A	A	B	A	AC	B	AB	ABC	BC	BC	ABC	BC	BC	BC
	A=4/B=1/C=1					B		ABC	A=1/B=6/C=6					

STRATEGIE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	77	26	1	10	0	16	2	0	1	3	2	0	1	1
JBOSS AOP	99	27	0	10	1	34	2	0	1	3	5	0	1	1
CAESARJ	90	32	4	14	1	24/46	3	0	1	3	4	0	1	1
RESULTATS	A	A	B	AB	A	A	AB	ABC	ABC	ABC	A	ABC	ABC	ABC
	A=4/B=2					A		ABC	A=6/B=5/C=5					

VISITEUR 1	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	59	16	0	14	4	24	1	3	0	8	2	0	0	0
JBOSS AOP	130	16	0	10	4	48	2	3	0	4	5	0	0	0
CAESARJ	82	42	3	23	4	44§88	3	3	0	4	4	0	0	0
RESULTATS	A	AB	AB	B	ABC	A	A	ABC	ABC	BC	A	ABC	ABC	ABC
	A=4/B=4/C=1					A		ABC	A=5/B=5/C=5					

VISITEUR 2	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	90	26	1	15	4	42	2	3	0	4	4	0	0	0
JBOSS AOP	131	34	2	19	6	74	3	3	0	4	10	0	0	0
CAESARJ	82	42	3	23	4	44§88	3	3	0	4	4	0	0	0
RESULTATS	C	A	A	A	AC	A	A	ABC	ABC	ABC	AC	ABC	ABC	ABC
	A=4/C=2					A		ABC	A=6/B=5/C=6					

FABRIQUE ABSTRAITE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	76	31	6	24	1	40	2	3	1	1	1	0	2	2
JBOSS AOP	84	33	6	24	1	36	2	1	1	1	2	0	2	2
CAESARJ	54	24	3	20	0	14/30	2	0	0	0	4	0	0	0
RESULTATS	C	C	C	C	C	C	ABC	C	C	C	A	ABC	C	C
	C=5					C		C	A=2/B=1/C=5					

PONT	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	84	25	0	21	1	26	2	0	0	3	1	0	0	0
JBOSS AOP	102	27	0	22	1	32	2	0	0	2	3	0	0	0
CAESARJ	80	23	6	20	1	20/46	4	0	0	0	6	0	0	0
RESULTATS	C	C	AB	C	ABC	A/C	AB	ABC	ABC	C	A	ABC	ABC	ABC
	A=3/B=3/C=4					A/ABC		ABC	A=5/B=4/C=5					

MONTEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	58	26	0	16	4	26	2	0	0	3	1	0	0	0
JBOSS AOP	78	26	0	16	4	38	2	0	0	2	3	0	0	0
CAESARJ	64	29	4	18	3	22/38	3	0	0	0	4	0	0	0
RESULTATS	A	AB	AB	AB	C	A/C	AB	ABC	ABC	C	A	ABC	ABC	ABC
	A=4/B=3/C=1					A/ABC		ABC	A=5/B=4/C=5					

FABRICATION	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	48	11	2	9	2	16	2	6	2	1	1	0	1	1
JBOSS AOP	53	13	2	9	2	18	2	1	2	1	2	0	1	1
CAESARJ	34	7	2	6	1	10§22	2	0	0	0	3	0	0	0
RESULTATS	C	C	ABC	C	C	A/C	ABC	C	C	C	A	ABC	C	C
	A=1/B=1/C=5					A/C		C	A=2/B=1/C=5					

PATRON DE METHODE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	45	22	0	11	0	12	2	0	0	3	1	0	0	0
JBOSS AOP	68	26	0	13	1	24	2	0	0	2	3	0	0	0

CAESARJ	49	22	0	11	0	14§16	2	0	0	3	1	0	0	0
RESULTATS	A	AC	ABC	AC	AC	A	ABC	ABC	ABC	B	AC	ABC	ABC	ABC
	A=5/B=1/C=4					A		ABC	A=5/B=5/C=5					

INTERPRETEUR	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	96	33	4	30	2	32	2	21	0	6	1	0	0	0
JBOSS AOP	138	39	4	32	4	48	2	0	0	6	5	0	0	0
CAESARJ	118	31	4	27	2	34/40	2	0	0	6	1	0	0	0
RESULTATS	A	C	ABC	C	AC	A	ABC	BC	ABC	ABC	AC	ABC	ABC	ABC
	A=3/B=1/C=4					A		BC	A=6/B=5/C=6					

ETAT	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	105	38	0	22	9	20	2	3	1	4	1	0	3	3
JBOSS AOP	118	38	0	22	9	20	2	3	2	4	2	0	3	3
CAESARJ	107	38	0	22	9	22	2	3	1	4	1	0	3	3
RESULTATS	A	ABC	ABC	ABC	ABC	AB	ABC	ABC	AC	ABC	AC	ABC	ABC	ABC
	A=5/B=4/C=4					AB		ABC	A=6/B=4/C=6					

FAÇADE	TAILLE					COUPLAGE		COHESION	METRIQUES DE POA					
LANGAGE	LOC	WOM	NOC	NOO	NOA	CBC	DIT	LOCC	CIM	CDA	NMA	NPCA	NPCC	NP
ASPECTJ	49	15	0	9	0	18	1	0	4	0	1	0	1	1
JBOSS AOP	49	15	0	9	0	18	1	0	4	0	1	0	1	1
CAESARJ	44	15	0	9	0	10	1	0	\	\	\	\	\	\
RESULTATS	C	ABC	ABC	ABC	ABC	C	ABC	ABC	AB	AB	AB	AB	AB	AB
	A=4/B=4/C=5					C		ABC	A=6/B=6					

Le tableau suivant résume les différents résultats présentés :

	TAILLE	COUPLAGE	COHESION	AOP
COMMANDE	A=4	A	B	B=5/C=5
COMPOSITE	A=5	A	B	A=6
COR	A=4	A	C	B=5/C=5
MEDIATEUR	A=4	A	BC	A=6
OBSERVATEUR	A=4	A	BC	B=5/C=5
POIDS MOUCHE	A=4	A	C	A=6
ITERATEUR	A=5	A	ABC	A=6/C=6
MEMENTO	A=3/B=3/C=3	A	ABC	A=6/C=6
PROTOTYPE	C=4	A	ABC	A=6C=6
SINGLETON	A=4C=4	B	AC	B=5
ADAPTATEUR	A=5/C=5	AC/C	ABC	A=6/C=6
DECORATEUR	/C=5	BC	ABC	C=6
PROXY	A=4	B	ABC	B=6/C=6

STRATEGIE	A=4	A	ABC	A=6
VISITEUR1	A=4/B=4	A	ABC	A=5/B=5/C=5
VISITEUR2	A=4	A	ABC	A=6/C=6
AFABRIQUE ABSTRAITE	C=5	C	C	C=5
PONT	C=4	A/ABC	ABC	A=5/C=5
MONTEUR	A=4/	A/ABC	ABC	A=5/C=5
FABRICATION	C=5	A/C	C	C=5
PATRON DE METHODE	A=5	A	ABC	A=5/B=5/C=5
INTERPRETEUR	C=4	A	BC	A=6/C=6
ETAT	A=5	AB	ABC	A=6/C=6
FACADE	C=5	C	ABC	A=6/B=6
ASPECTJ	17	19§17	15	16
JBOSS AOP	2	4§6	19	8
CAESARJ	10	4§7	22	18
	ASPECTJ	ASPECTJ	CAESARJ	CAESARJ

Voici les résultats de performance :

	AspectJ			JBoss AOP			CaesarJ		
	Max	Min	Moyenne	Max	Min	Moyenne	Max	Min	Moyenne
COMMANDE	624	530	577	630	587	608,5	577	484	530,5
COMPOSITE	7885	6153	7019	6407	5743	6075	7900	6870	7385
COR	1826	1452	1639	1996	1790	1893	2030	1576	1803
MEDIATEUR	422	257	339,5	798	730	764	397	203	300
OBSERVATEUR	1937	1654	1795,5	2077	1955	2016	1967	1779	1873
POIDS MOUCHE	1452	1405	1428,5	1416	1390	1403	1452	1389	1420,5
ITERATEUR	1155	921	1038	1312	1283	1297,5	1187	1046	1116,5
MEMENTO	1624	1436	1530	1778	1644	1711	1700	1421	1560,5
PROTOTYPE	874	842	858	904	833	868,5	530	515	522,5
SINGLETON	259	203	231	438	422	430	234	172	203
ADAPTATEUR	3224	2685	2954,5	2810	2693	2751,5	3201	2873	3037
DECORATEUR	499	468	483,5	924	777	850,5	363	296	329,5
PROXY	6387	5246	5816,5	4819	4051	4435	6449	5325	5887
STRATEGIE	1193	874	1033,5	1004	864	934	1109	889	999
VISITEUR1	2701	2560	2630,5	2963	2673	2818	2826	2311	2568,5
VISITEUR2	3123	2935	3029	2836	2834	2835			
AFABRIQUE ABSTRAITE	922	811	866,5	1657	1253	1455	749	702	725,5
PONT	4840	4465	4652,5	4547	4333	4440	4887	4341	4614
MONTEUR	2778	2108	2443	2220	2156	2188	2810	2077	2443,5
FABRICATION	1046	984	1015	1325	1310	1317,5	1046	953	999,5
PATRON DE METHODE	1764	1576	1670	1884	1802	1843	1795	1249	1522
INTERPRETEUR	297	250	273,5	623	585	604	204	187	195,5
ETAT	4528	4119	4323,5	3643	3556	3599,5	4574	4153	4363,5
FACADE	2682	2682	2682	2682	2682	2682	2682	2682	2682

ANNEXE B

Notre comparaison qualitative, se base sur des formulaires qui comportent des questions, des notes et parfois des commentaires. (Tous les formulaires sont disponibles sur notre site web : http://www.debboub-soumeya.sitew.org/Curriculum_Vitae.B.htm#Research_Projects.C)

A titre d'exemple, voici une réponse de l'un des étudiants participants dans notre étude :

		AspectJ		JBoss AOP		CaesarJ	
		Note	**Commentaire**	**Note**	**Commentaire**	**Note**	**Commentaire**
Compréhension	Est-il facile d'apprendre cette approche?	8	Oui	7	Oui	8	Oui, pas trop difficile. Les cclasses sont surchargées de la même manière que les méthodes
	Est-ce que les nouveaux termes de cette approche sont faciles à maitriser ?	8	No, pas trop compliqué	6	Oui, mais je ne ai pas aimé le fait que la syntaxe "Les coupes" sont différentes de celle d'AspectJ et CaesarJ	8	Pas trop compliqué
	Y a-t-il des documents suffisants pour apprendre cette approche ?	10	Trop	5	Peu de travaux sur ce langage	8	Oui
Concepts de l'approche	Est-ce que les nouveaux concepts de cette approche sont faciles à comprendre et à utiliser?	7	Oui	6	Oui	8	Nouvelles fonctionnalités pour la modularité et la réutilisation par le regroupement des modules
	Pensez-vous que les concepts de cette approche sont appropriés pour l'implémentation des patrons de conception ?	8	Oui, surtout son mécanisme d'introduction	4	Non	10	Parfaitement, CaesarJ est très utile, il simplifie la programmation des modèles de conception, et rend le modèle de conception bien modularisé dans les différentes parties du composante CaesarJ
	Est-ce que ces concepts sont suffisants pour la création des abstractions réutilisables ?	5	A besoin d'autres composants pour la réutilisation	3	Ceux de Java	10	À mon avis CaesarJ supporte bien la réutilisation
IDE et les outils support	Pensez-vous que l'IDE utilisé est approprié et facile à utiliser?	8	Oui	9	Oui	9	Oui
	Est-il facile de trouvé des outils adéquats avec cette approche ?	8	Oui	5	Ceux de Java (la plupart du temps ils ne prennent pas en charge les nouveaux concepts de la POA)	8	Oui
	Pensez-vous que les	8	Offre un puissant	5	Pas de visualisation	7	Assez bien pour aider les

	outils support de cette approche sont matures?		IDE et des nouvelles fonctionnalités pour la visualisation des nouveaux concepts.				programmeurs dans la programmation
Les patrons de conception	Pensez-vous que cette approche aide les développeurs dans l'implémentation des patrons de conception?	8	Oui	5	Utiliser les classes java pour coder les aspects nécessite plus d'attention!	10	Il offre une meilleure séparation des préoccupations
	Pensez-vous que les implémentations obtenues sont faciles à comprendre et à faire évoluer ?	6	Surtout pour la compréhension des codes, ils sont très clairs.	5	Un peu	9	Oui, les codes sont simples et les modules réutilisables sont très clairs