الجمهوريـــة الجزائريـــة الديمقراطيـــة الشعبيـــة

وزارة التعليـــم العالـي والبحـــث العلمـــي

BADJI-MOKHTAR– ANNABA – UNIVERSITY
UNIVERSITE BADJI –MOKHTAR – ANNABA

جامعة باجي مختار- عنابة

FACULTÉ DES SCIENCES DE L'INGÉNIEUR

كلية علوم المهندس

DÉPARTEMENT D'INFORMATIQUE

قسم الإعلام الآلي

ANNEE UNIVERSITAIRE 2006

سنة جامعية 2006

MEMOIRE

Présenté en vue de l'obtention du diplôme de **MAGISTER** en Informatique

THEME

Une Evaluation Qualitative et Quantitative des approches de Séparation des préoccupations basée sur les patrons de conception et leur utilisation pour l'Intelligence Artificielle et les systèmes Multi-Agents

Option

Intelligence Artificielle Distribuée (IAD)

Par : Abdel Hakim Hannousse

DEVANT LE JURY

ENCADREUR	**:**	Hayet Farida Merouani	Dr.	U. Annaba
PRESIDENT	:			
EXAMINATEURS	:			

Une Evaluation Qualitative et Quantitative des Approches de Séparation de Préoccupations Basée sur les Patrons de Conception et leur Utilisation pour Intelligence Artificielle et les Systèmes Multi-Agents

Un Mémoire en vue de l'obtention du diplôme de Magister en informatique Option Intelligence Artificielle Distribuée
Université Badji Mokhtar de Annaba

Abdel Hakim Hannousse

2006

Devant le Jury:

مـــــــــلخـــــــــص

إن الدارس للذكاء الاصطناعي يجد أمامه مجموعة من الأسئلة المبهمة التي تستوجب الرد عليها لحل المشاكل المتعلقة بالبحث وإنجاز البرامج المعتمدة في المجال، من بين هده المشاكل نذكر منها على سبيل المثال لا الحصر إيجاد كيفية خاصة لتمثيل البيانات المستخلصة من المتخصصين، وإيجاد برنامج ذكي لاستخلاص وتعميم المعلومات،وتوزيع النشاط على مجموعة من البرامج لربح الوقت وإضافة لمسة ذكية للبرامج المعتمدة.

الا شكال المطروح هنا، هو أن جل الطرق الثقليدية المعتمدة في مجال الذكاء الاصطناعي، ليست كافية وحدها حل المشاكل المطروحة، لهدا نرى معظم الباحثين يتجهون إلى المزج بين مجموعة من هده الطرق قصد إيجاد حل افضل. في هدا النطاق،دراستنا تعتمد أولا على إدخال مفهوم جديد لهدا المجال وهو *البترونات* وفي الحقيقة هو ليس بجديد بقدر ما هو ولأول مرة يستخدم كطريقة حل في مجال الذكاء الاصطناعي.

الجزء الأول من هده الرسالة يبين كيفية إدماج *البترونات* في مجال الذكاء الاصطناعي لحل بعض المشاكل المطروحة، فيما الجزء الثاني منها يمثل اعتماد هده الأخيرة كوسيلة للمقارنة بين احدث الطرق المعتمدة في هندسة البرامج ، حيث أنها تمثل طريقة هامة في هدا المجال، أضف إلى دلك أن برمجتها اعتمادا على هده الطرق الحديثة تسمح لنا بتحسين البرمجة في مجال الذكاء الاصطناعي. هدا البحث يعتبر الأول من نوعه في المجال أد يدمج مجالين في آن واحد الذكاء الاصطناعي وهندسة البرامج.

Résumé

Les problèmes les plus fréquemment capturés dans la conception orientée objets sont : l'enchevêtrement et la dispersion du code source des programmes. Le premier peut apparaître quand plus d'une préoccupation sont utilisées dans une même application et ça influe négativement sur la réutilisabilité de ses codes source. Le deuxième peut apparaître dans le cas où le code d'une préoccupation d'une application a été totalement dispersé et on ne peut pas l'encapsuler dans une seule entité, de ce fait, son code devient non réutilisable.

Ces deux problèmes : enchevêtrement et dispersion du code influent sur le processus de développement des logicielles de différentes manières : mauvaise traceabilité, manque de productivité, peu de réutilisation, et de qualité ainsi qu'une difficulté de l'évolution. Afin de résoudre ces problèmes, plusieurs techniques sont actuellement considérées, elles tentent d'augmenter le taux d'expression de l'approche orientée objets. Ces techniques sont connues sous l'appellation : Approches Avancées de Séparation des Préoccupations. Parmi ces approches on peut citer : la Programmation Orientée Aspects, les Filtres de Composition et la séparation multidimensionnelle des préoccupations.

D'un autre coté, les patrons de conception représentent des descriptions abstraites de solutions de problèmes sous certaines contraintes. Les patrons de conception rassemblent les connaissances des experts dans la conception des logiciels suite à de multiples expériences aboutis. Les patrons de conception peuvent nous aider à développer des logiciels plus flexibles et plus faciles à maintenir. En plus, ils permettent à des débutants de maîtriser rapidement les bonnes pratiques de programmation et comprendre plus aisément les systèmes logiciels existants.

Le but principal de ce travail consiste à démontrer l'efficacité de l'utilisation des patrons de conception comme benchmark pour l'évaluation des approches avancées de séparation des préoccupations. Notre idée consiste à implémenter les patrons de conceptions en utilisant à la fois l'approche orientée objets et les approches avancées de séparation de préoccupations et à comparer le code résultant on utilisant des métriques qualitatives et quantitatives.

Notre étude prend comme domaine d'application les systèmes multi-agents et particulièrement ceux qualifiés de réactifs, qui englobent les systèmes orientés objets. Une brève discussion sur l'utilisation des patrons de conception dans le domaine de l'I.A est apportée afin de tirer profit de l'application de notre approche dans un cadre réel prometteur.

ABSTRACT

The recurrent problems that occur in object oriented software design are code tangling and code scattering. While, the first one occurs when we use more than one concern in the same application which complicates the source code, and decreases the reusability of it, the second problem occurs when the concern is not centralized and can't be encapsulated in a single entity. In the two cases, the concern code can't be reused efficiently in other application.

Both the code tangling and the code scattering problems affect the development process of the application in different manners: bad traceability, lack of productivity, weak code reusability and quality and difficult application evolution. To solve those problems, several techniques are being researched. Most of them attempt to increase the expressiveness of the Object Oriented paradigm. Such techniques are known as Advanced Separation Of Concerns.

On the other hand, design patterns are abstract descriptions of solutions to problems under certain constraints. Design patterns capture expert knowledge's and is abstracted from many successful designs. Design patterns can help develop more flexible and maintainable software. In addition, design patterns can help novices learn good programming faster and understand existing systems better.

The main purpose of this research is to demonstrate the efficiency of the applicability of patterns as a benchmark to evaluate ASOC approaches. Our idea consists of implementing design patterns using both OO paradigm and ASOC approaches and comparing resulting codes using both a qualitative and a quantitative metrics.

Our study is conducted within the multi-agents systems and particularly reactive ones that include object oriented systems. A brief discussion about using patterns in AI area is provided in order to benefit from the applicability of our work in a promising real framework.

Dedicated to My Family
My Wife &
My Best Friends

ACKNOWLEDGEMENTS

The work presented in this thesis could not have been possible without the support of many people. Many thanks to my supervisors: Djamel Meslati and Hayet Merouani for their timely advice, consultations, encouragement, and critiques throughout the development of this work. Meslati inspired me by first introducing me to design patterns, then convincing me of their values. He has been an invaluable source of support and guidance all along my graduate program.

Many thanks to my best friends: Chemseddine Dridi, Ibrahim Boutheldja, Riad Nedjah and Faouzi Dehdouh.

I also would like to thank my Mom and Dad for their encouragement, understanding and caring. They have always been a source of motivation. Without them, I could have never accomplished what I have done today.

Last and not least, I am very grateful to my wife who also has been a source of guidance and goal.

LISTE DES TABLEAUX

LISTE DES FIGURES

TABLES DES MATIÈRES

Chapitre 1

INTRODUCTION

1. Objectifs

Différents problèmes se produisent de manière récurrente pendant le développement des logiciels selon le paradigme orienté objets. Parmi ces problèmes on note l'enchevêtrement et la dispersion du code source. Alors que le premier se produit quand plus d'une préoccupation ont été utilisés dans la même application, le deuxième a lieu quand le code d'une préoccupation n'est pas centralisé et encapsulé dans une seule entité. Dans les deux cas, la réutilisation du code résultat diminue et la complexité de la compréhension du ce code augmente.

L'enchevêtrement et la dispersion du code source affectent le processus du développement des logiciels de différentes manières: mauvaise traçabilité, manque de productivité, peu de réutilisation, qualité réduite ainsi qu'une difficulté de l'évolution et de la maintenance. Afin de résoudre ces problèmes, plusieurs techniques sont actuellement considérées, elles tentent d'augmenter la puissance descriptive de l'approche orientée objets. Ces techniques sont connues sous l'appellation : Approches Avancées de Séparation des Préoccupations (ASOC en abrégé pour Advanced Separation of Concerns). Parmi ces approches on peut citer : la Programmation Orientée Aspects, les Filtres de Composition et la séparation multidimensionnelle des préoccupations.

D'un autre coté, les patrons de conception représentent des descriptions abstraites de solutions de problèmes sous certaines contraintes. Les patrons de conception rassemblent les connaissances des experts dans la conception des logiciels suite à de multiples expériences abouties. Les patrons de conception peuvent nous aider à développer des logiciels plus flexibles et plus faciles à maintenir. En plus, ils permettent à des débutants de maîtriser rapidement les bonnes pratiques de programmation et de comprendre plus aisément les systèmes logiciels existants.

Le but principal de cette recherche est de démontrer l'efficacité de l'application des patrons de conception comme un standard pour l'évaluation des approches ASOC. Notre idée consiste à implémenter les patrons de conception en utilisant les différentes approches ASOC et de comparer les codes résultats en utilisant une gamme de métriques qualitatives et quantitatives, l'application des résultats obtenus dans le contexte de l'intelligence artificielle et dans les systèmes multi-agents étant un sous objectif de ce travail.

2. Motivation

De nos jours, la majorité des recherches a été consacrée à trois approches ASOC: les Filtres de Composition (CF) [2, 21], la Programmation Orientée Aspects (AOP pour Aspect Oriented Programming) [28, 31, 35] et la séparation multidimensionnelle des préoccupations (hyperespace) [40]. Ces approches

visent à fournir de meilleurs concepts et mécanismes de séparation des préoccupations. Malheureusement, la philosophie de chaque approche est différente et les concepts utilisés ne sont pas semblables. Devant cette diversité, la communauté du génie logiciel souhaite l'émergence d'une unification des approches qui puisse avoir un impact sur la standardisation et l'usage de la séparation de préoccupations. Pour atteindre cet objectif, beaucoup reste à faire, aussi bien au niveau théorique que pratique. Alors qu'au niveau pratique, il est question de tentatives d'estimation des approches à travers des applications pratiques, le niveau théorique englobe les tentatives qui consistent à faire ressortir et unifier les concepts, rehausser le pouvoir descriptif et imaginer des modèles formels de séparation.

La motivation principale de notre travail est de contribuer à l'émergence d'une approche unifiée de modélisation des préoccupations à travers une évaluation des approches ASOC en utilisant les patrons de conception comme benchmarks.

3. Contexte

Ce travail ne peut être considéré comme une étude comparative complète des approches ASOC. Dans notre esprit, cela ne peut pas être accompli sans considérer une comparaison a plusieurs niveaux, comme l'efficacité de la mise en oeuvre, l'utilité et, plus généralement, la contribution au développement, au processus d'évolution et de maintenance. Cependant, le travail dans sa forme actuelle se base sur une idée originale qui consiste à utiliser les patrons de conception comme benchmarks. Dans un autre travail conduit séparément, nous avons aussi réalisé un mapping des concepts entre les différentes approches ASOC. Nous croyons que la conjonction de ces études comparatives nous mènera, à la fin, vers une unification des concepts des différentes approches.

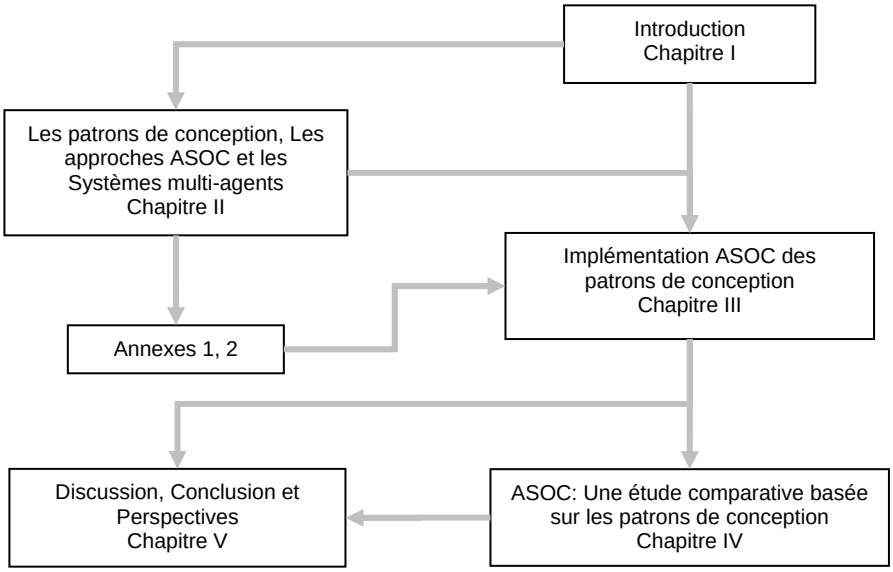
Lors de notre étude, nous avons été confrontés au choix des différentes approches ASOC à considérer et les patrons de conception à utiliser. Nous avons limité notre étude aux trois approches ASOC (CF, AOP et Hyperespaces), qui sont considérées comme des approches principales de séparation des préoccupations (voir la revue spéciale de CACM [45]). A propos des patrons de conception, nous avons choisi un patron de chaque catégorie des patrons suivant la classification proposée par les quatre auteurs (R. Gamma, R. Helm, R. Johnson, et J. Vlissides) qui forme le célèbre groupe nommé GOF (Gang Of Four) [80]. Cette classification propose trois catégories: patrons créationnistes, patrons structurels et patrons comportementaux. La première catégorie traite des patrons qui exigent une génération de nouveaux objets pour la résolution de problèmes. Les patrons dans la deuxième catégorie exigent la restructuration du système existant. La troisième regroupe des patrons qui changent le comportement des objets pour résoudre un problème.

Les patrons qui ont été sélectionnés pour notre travail sont: Singleton de la catégorie créationniste, ChainOfResponsibility de la catégorie structurelle et Adapter de la catégorie comportementale. Notre choix de ces trois patrons est lié à notre domaine d'application (i.e. systèmes multi-agents). En effet, pendant que le ChainOfResponsibility est un bon exemple d'interactions au sein d'un système basé agents, le Singleton traite le cas où seulement un agent peut

exister à la fois (cas des agents superviseurs), et l'Adapter traite de l'adaptation qui est une des caractéristiques majeures des agents. Notons que ce travail traite principalement des systèmes multi-agents réactifs qui englobent le paradigme orienté objets.

4. Structure de la thèse

Cette thèse se divise en deux parties. Une partie d'introduction des paradigmes et des concepts et une partie décrivant le résultat de notre recherche. La première partie introduit le lecteur aux patrons de conception et aux nouveaux paradigmes de conceptions des logiciels. La seconde partie, présente notre idée originale pour évaluer les paradigmes ASOC et les résultats de l'application de cette idée aux trois approches ASOC retenues. La figure suivante présente les dépendances entre les différents chapitres de cette thèse.



Le chapitre 1 a introduit notre travail ainsi que nos objectifs de recherche.

Le chapitre 2 introduit les patrons de conception, les approches ASOC ainsi que les systèmes multi-agents réactifs.

Le chapitre 3 décrit comment on peut implémenter les patrons de conception en utilisant les approches ASOC. Il considère trois patrons: l'Adaptateur (Adapter), Chaîne de responsabilité (ChainOfResponsibility) et le Singleton (Singleton).

Le chapitre 4 évalue les approches ASOC en comparant les codes source résultat moyennant quelques métriques utilisées en génie logiciel.

Le chapitre 5 donne une aperçue et quelques idées générales sur la résolution des problèmes classiques de l'intelligence artificielle (IA) en utilisant le concept de patron, et conclut la thèse en donnant les directions futures du travail.

La partie des annexes est une partie complémentaire. Dans l'annexe 1, on

présente une description détaillée des patrons de conception utilisés dans notre étude. Dans l'annexe 2, on décrit d'une manière détaillée les implémentations pratiques des approches ASOC, il s'agit d'AspectJ pour la programmation orientée aspects, ConcernJ pour les Filtres de Composition et Hyper/J pour la séparation multidimensionnelle des préoccupations.

Chapitre 2

LES PATRONS DE CONCEPTION, LA SEPARATION DES PREOCCUPATIONS ET LES SYSTEMES MULTI-AGENTS

Dans le domaine du développement des logiciels, nombreux sont les problèmes qui se produisent et se reproduisent de manière répétée. Pour chacun, il peut y exister plusieurs solutions et à chaque occurrence de ce problème, il faut encore chercher une solution. Cette solution peut être identique à l'une des solutions déjà utilisée précédemment mais malheureusement non sauvegardées. Pour pallier à ce problème, les compétences et les connaissances qui résolvent un problème doivent être capturées dans une nouvelle entité appelée *patron* et peuvent être documentées dans une forme (i.e. langage) spéciale qui devient un outil de communication entre les concepteurs. Les patrons sont considérés comme une approche prometteuse dans le développement des systèmes et dans d'autres domaines. L'idée principale derrière les patrons est d'enregistrer les expériences d'un domaine spécifique afin de permettre aux concepteurs d'avoir un meilleur rendement et de communiquer plus efficacement.

D'un autre côté, la séparation des préoccupations est une approche où un système logiciel est vu comme une composition de préoccupations indépendantes telles que la synchronisation, la sécurité, la persistance, etc. Il est actuellement bien admis qu'une séparation de préoccupations appropriée a une influence sur le processus de développement : elle réduit la complexité du logiciel, facilite leur réutilisation, améliore leur compréhension, etc.

Ce chapitre traite plusieurs paradigmes. Il explique le concept de patrons, leur philosophie, leur structures ainsi que leur applicabilité dans le développement des logiciels et décrit leur l'implémentation dans le paradigme orienté objet. Il introduit aussi les principales approches ASOC et donne un aperçu sur les systèmes multi-agents.

1. La philosophie des patrons

Les patrons ont été introduits par C. Alexander en 1977 [17-19]. Ce dernier fût le premier à définir des patrons pour résoudre des problèmes récurrents dans la conception architecturale urbaine. La communauté du génie logiciel a aussi observé des problèmes répétitifs pour lesquels beaucoup de temps et d'efforts peuvent être économisés si leurs solutions été enregistrées et réutilisées à chaque besoin. Cette idée est derrière la naissance de la technologie des patrons.

L'expérience professionnelle a montré que les patrons ont plusieurs propriétés importantes. C'est aussi ces valeurs qui motivent beaucoup de praticiens dans différents domaines à d'écrire et à apprendre le concept de patrons.

Les patrons capturent les connaissances des experts et les rendrent explicitent. Ils fournissent un vocabulaire commun de haut niveau d'abstraction qui peut

être utilisé facilement par des patriciens d'un domaine spécifique. L'usage des patrons dans le développement des systèmes permet de réutiliser des architectures des logiciels et aussi de comprendre des systèmes existants plus facilement [1, 5, 14, 20, 23, 37, 39, 48, 48, 52, 55].

Les paragraphes qui suivent discutent les propriétés clés des patrons. Pour plus de détail voir [36, 66, 72-75, 78, 91].

Capturer les connaissances des experts: Par définition même, les patrons capturent des expériences riches. Un patron capture dans une forme spéciale l'essence d'une expérience faite face à un problème et la rend accessible pour une future utilisation. Il peut capturer aussi une structure importante, une expérience, une compétence dans un domaine donné, mais qui n'été pas cependant largement connue.

Le partage de la connaissance: Les développeurs tentent de réutiliser des expériences abouties. Tout au long de leurs carrières, ils réutilisent leurs expériences et deviennent plus compétents. Malheureusement, cette forme de réutilisation est limitée à l'expérience individuelle, le partage des connaissances entre les développeurs est ainsi limité ou absent. L'utilisation des patrons fournit un mécanisme pour partager plus facilement les connaissances entre les développeurs et améliore significativement leurs compétences mutuelles.

Un vocabulaire commun: Les noms des patrons fournissent un vocabulaire commun pour les développeurs. En documentant et explorant des alternatives possibles, les patrons facilitent la communication et l'échange.

Permettre la réutilisation d'architecture des logiciels: Dans un environnement volatile, réutiliser des patrons est souvent le seul moyen adéquat pour réutiliser les compétences antérieures du développement. En effet, bien que la plateforme matérielle des systèmes d'exploitations change, les patrons eux-mêmes peuvent être réutilisés tels quels en grande partie. Seuls des portions limitées des patrons doivent être réimplémentées pour les rendre conformes aux caractéristiques d'une éventuelle nouvelle plateforme. Par conséquent, les risques et les efforts du développement peuvent être grandement réduits.

Aide à l'apprentissage et à la formation: Beaucoup de systèmes utilisent les patrons. Lorsque les patrons sont maîtrisés, ils permettent de suivre, plus facilement, le flux de contrôle d'un système existant et rende sa compréhension aisée. Apprendre des patrons aide à comprendre des systèmes existants plus rapidement. Ils fournissent des solutions aux problèmes communs et décrivent des structures et des mécanismes des systèmes plus profondément. Souvent, ils encouragent et aide les novices à concevoir des systèmes en commettant moins d'erreurs tout en leur donnant l'opportunité d'apprendre rapidement des systèmes existants.

1.1. La forme d'un patron

Les patrons peuvent être présentés sous une forme narrative ou bien sous une forme structurée. Cela peut améliorer la lisibilité des patrons. Il existe différentes formes structurées. Les éléments suivants sont nécessaires pour décrire un patron sous une forme structurée.

- **Nom.** C'est un vocable ou une expression significative pour décrire un patron. C'est un élément extrêmement important parce qu'il facilite la communication entre les patriciens d'un domaine et les aide dans la sélection du patron dont ils ont besoin quand ils cherchent d'une solution à un problème particulier.
- **L'intention.** C'est une expression ou une phrase qui répond aux questions de ce type : Que peut faire ce patron ? Que résout-il comme problème ?
- **Aussi connu comme.** Autres noms célèbres pour le patron, s'il en existe.
- **Motivation.** Un scénario qui illustre le problème adressé et comment ce problème peut être résolu en utilisant le patron.
- **Applicabilité.** Quelles sont les situations dans lesquelles le patron peut être appliqué? Comment peut-on reconnaître ces situations?
- **Structure.** Une représentation graphique des différents composants du patron qui interviennent dans la résolution d'un problème.
- **Participants.** Une description détaillée des différents composants du patron ainsi que la responsabilité de chacun.
- **Collaborations.** Il donne une réponse à propos de la collaboration des participants pour répartir les responsabilités impliquées.
- **Conséquences.** Comment le patron supporte il ses objectifs? Quels sont les compromis et les résultats qui découlent de l'application du patron? Quels aspects du système permet-il de varier indépendamment?
- **Implémentation.** Quels pièges, illusions, ou techniques doit-on connaître lors de l'implémentation du patron?
- **Echantillon de Code.** Fragments de code qui illustrent comment le patron est implémenté en utilisant certains langages de programmation comme Java, C++ ou Smalltalk.
- **Les usages connus.** Des exemples sur le domaine d'application du patron. Au moins deux exemples de domaines différents doivent être fournis.
- **Les patrons apparentés.** Quels sont les patrons qui sont en rapport avec celui-ci? Quelles sont les différences importantes? Avec quels autres patrons doit-on l'utiliser?

1.2. La classification des patrons de conception

Les patrons de conception décrits par GOF [80, 81] sont organisés en trois catégories: créationniste, structurelle et comportementale.

Patrons Créationnistes : Ils nécessitent la génération de nouveaux objets dans leurs solutions, par exemple le patron Singleton.

Patrons structurels : Ceux qui nécessitent une restructuration du système existant pour implémenter la solution, tel que le patron Adapter.

Patrons comportementaux : Ceux qui nécessitent des interactions spécifiques entre les participants du système pour mettre en œuvre la solution,

tel que le patron ChaînOfResponsibility.

1.3. Evaluation des implémentations des patrons de conception

Les patrons de conception sont considérés très importants et utiles pour la réutilisation et l'évolution des logiciels. Malheureusement, leur implémentation sous le paradigme orienté objets n'est pas toujours convenable, cela est dû à la nature et aux limites de ce paradigme. Dans la suite, on cite quelques problèmes.

Problème de Confusion

La représentation d'un patron sous forme de classes rend la partie relative au patron et les autres parties de l'application fortement liées. Dans ce cas, il est difficile de distinguer le comportement relatif au patron des comportements relatifs aux autres classes sur lequel il a été appliqué. Cela réduit la traçabilité, la réutilisation du patron et rend difficile la maintien des patrons une fois le système conçu.

Problème de redirection

Différents patrons de conception utilisent le mécanisme de la redirection explicite afin d'activer leurs comportements spécifiés dans d'autres classes du système. Par conséquent, cela minimise la compréhension du code, sa flexibilité et augmente le taux de communication entre les objets.

Problème de rupture d'encapsulation

L'implémentation des patrons de conception à plusieurs effets habituellement indésirables, cela est dû à la possibilité offerte aux patrons de modifier la structure des systèmes, surtout quand le comportement du patron nécessite d'accéder à certains attributs dans d'autres classes. Ceci implique une violation des droits d'accès à cet attribut, donc, violation du mécanisme de l'encapsulation.

Problème lié a l'héritage

L'intégration d'un patron dans la hiérarchie d'un système orienté objet, nécessite, dans la plupart des cas, l'utilisation du mécanisme de l'héritage ce qui implique une forte liaison entre le patron et le reste du système. Par conséquent, le taux de la réutilisation est réduit. Dans certains cas, l'intégration du patron devient impossible. Par exemple, lors de l'utilisation des modèles d'implémentation qui ne supportent pas le concept d'héritage multiple (comme Java), l'intégration est problématique.

Problème de composition des patrons

Si des plusieurs patrons sont utilisés dans le même système, il devient difficile de garder la trace de l'exécution d'un patron particulier. Cela est particulièrement le cas si les mêmes classes sont impliquées dans plus d'un patron.

Problème de violation des concepts de l'orienté objets

Quelques patrons de conception préconisent de changer le comportement de certaines classes ce qui viole certains principes du paradigme orienté objets

comme l'héritage et les modificateurs d'accès.

2. La séparation avancée des préoccupations

Les applications d'aujourd'hui contiennent plusieurs préoccupations comme la sécurité, la persistance et la synchronisation. Les problèmes les plus importants qui se produisent dans le développement des logiciels sous le paradigme orienté objets sont : l'enchevêtrement et la dispersion de code source des préoccupations. Le premier se produit quand plus d'une préoccupation sont utilisées dans la même application, ce qui complique la compréhensibilité du code source résultat et diminue le taux de son réutilisation. Alors que, le deuxième se produit quand le code de la préoccupation n'est pas centralisé et qu'il est impossible de l'encapsuler dans une seule entité.

L'enchevêtrement et la dispersion du code source affectent le processus du développement des logiciels de différentes manières: mauvaise traçabilité, manque de productivité, peu de réutilisation, difficulté de l'évolution et de la maintenance, etc. Afin de résoudre ces problèmes, plusieurs techniques sont actuellement considérées, elles tentent d'augmenter le taux d'expression de l'approche orientée objets. Ces techniques sont connues sous l'appellation : Approches Avancées de Séparation des Préoccupations. Parmi ces approches on peut citer : la programmation orientée aspects, les filtres de composition et la séparation multidimensionnelle des préoccupations.

La séparation des préoccupations est un concept clé en génie logiciel. Il fait référence à la capacité d'identifier, d'encapsuler, et de manipuler, indépendamment, des parties d'un logiciel relatives à une préoccupation particulière. L'utilisation des préoccupations a comme motivation fondamentale l'organisation et la décomposition des logiciels en parties maniables et compréhensibles.

Les préoccupations comme la persistance et la synchronisation sont commun. Une séparation des préoccupations appropriée est nécessaire afin de réduire la complexité du logiciel et d'améliorer la compréhensibilité de son code, augmenter la traçabilité, faciliter la réutilisation, l'adaptation et la maintenance et simplifier l'intégration d'autres composants dans les logiciels.

Cette partie du chapitre présente les concepts principaux de chaque approche ASOC: AOP, CF et Hyperespaces, qui seront utilisés dans le chapitre suivants. Le détail de ces approches est donné en annexe 2 sous forme d'une présentation des aspects syntaxiques et sémantiques des langages d'implémentation de ces approches, un pour chaque approche: AspectJ pour AOP, ConcernJ pour CF et Hyper/J pour Hyperespaces.

2.1. La programmation orientée aspects (AOP)

La programmation orientée aspects ou AOP, apporte une solution élégante et simple pour le problème de séparation des préoccupations. C'est une technique de programmation qui permet, dans un premier lieu, d'implémenter chaque préoccupation sous forme d'un module appelé aspect. L'ensemble des aspects

présente une caractéristique importante : celle de leur indépendance mutuelle, ce qui les rend hautement réutilisables. Dans un second lieu, les différents aspects sont combinés avec les classes, représentant les fonctionnalités de base de l'application, selon des règles bien définies. Cette combinaison est faite par un outil spécifique dit tisseur (i.e. Weaver en anglais).

Ce nouveau paradigme propose une restructuration des applications sur la base du concept d'aspect. Un aspect est une particule de l'application qui représente totalement et exclusivement une propriété donnée de l'application. Un aspect se définit comme une préoccupation qui est classiquement dispersé parmi les autres composantes d'un logiciel.

La programmation par aspects est une technique qui permet d'exprimer clairement les programmes qui sont liées à de tels aspects, les encapsule et les isole. En tant que technique de séparation de préoccupations, le but principal de l'AOP est de :

1. Séparer précisément les aspects du code de base de l'application
2. Indiquer comment l'aspect est intégré à l'application

Une fois que le programmeur aura écrit le code source de son application et spécifié ses aspects, il utilise le tisseur (i.e. Weaver) pour les fusionner. Il est à noter que la programmation orientée aspect est indépendante de la programmation orientée objets ; en effet, il est possible d'utiliser la programmation par aspects conjointement à d'autres paradigmes de programmation.

Le but de l'AOP est de rendre le code d'une application plus modulaire, en séparant le code d'une application en deux parties, une partie fonctionnelle qui représente le corps de l'application elle-même et une partie non fonctionnelle qui représente le code des préoccupations. Cela signifie que les préoccupations sont totalement localisées plutôt qu'éparpillées [35, 65, 67, 71, 85, 89, 90, 92].

Depuis la présentation de la technique d'AOP en 1997, l'essentiel des travaux conduits dans ce domaine consiste à développer des tisseurs pour des préoccupations diverses. En particulier, en 1998 AspectJ est mis à la disposition des utilisateurs par les chercheurs du centre de recherche PARC de Xerox.

AspectJ est une extension générique compatible de Java qui étend celui-ci par des concepts afin d'obtenir un langage orienté aspect [34, 83, 84].

AOP étend le modèle objet ordinaire par les cinq nouveaux concepts suivants:

2.1.1. Les aspects

L'aspect est une entité qui ressemble à une classe mais modélise une préoccupation qui entrecoupe plusieurs classes. Les aspects sont définis séparément et plusieurs aspects peuvent exister dans un même système.

2.1.2. Les points de jointure

Une application contient plusieurs points pertinents où plusieurs événements peuvent se produire. Ces points sont appelées points de jointure, parmi ces points on peut cité: l'instanciation d'un objet, l'invocation et exécution d'une méthode, l'accès en lecture ou en écriture à une variable d'instance, etc.

2.1.3. Les points de coupure

Ils sont des formes particulières de prédicats qui utilisent des opérateurs booléens et des primitives spécifiques pour capturer un ou plusieurs points de jointure.

2.1.4. Consignes (Advices)

Des événements spécifiques peuvent être capturés et des actions peuvent être appliquées dès que ces points sont atteints. Ces actions sont appelées consignes. Elles sont semblables aux méthodes. Le code d'une consigne est exécuté quand un point de jointure d'un point de coupure spécifique est capturé.

AspectJ supporte trois genres de consignes. Le genre de consigne détermine comment l'aspect réagit faces aux points de jointures. AspectJ divise les consignes en trois catégories selon qu'elles s'exécutent avant, après ou autour des points de jointure.

Les consignes avant. Une consigne *avant* (before advice) doit être exécutée avant le code qui a produit l'événement. Par exemple quand une exécution d'une méthode est capturée, le code de la consigne sera exécuté puis la méthode peut continuer son exécution normalement.

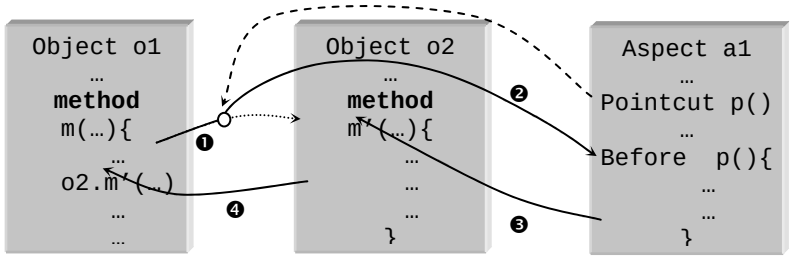


Fig. 2.1. Scénario d'exécution d'un consigne avant dans une application

Les consignes après. La consigne après (After advice) doit être exécutée après le code qui a produit l'événement. Par exemple quand une exécution d'une méthode est capturée, le code de la consigne sera exécuté juste après la fin de l'exécution de cette méthode.

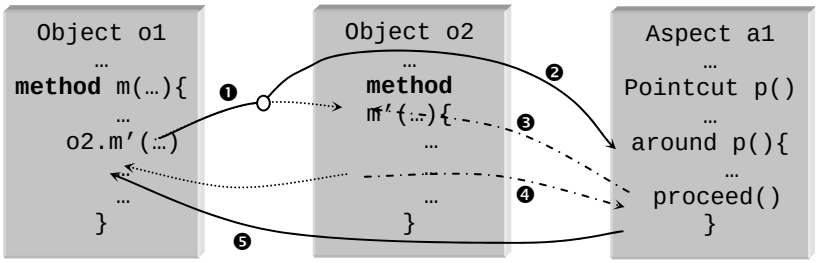


Fig. 2.2. Scénario d'exécution d'une consigne après dans une application

Les consignes autour. La consigne autour (Around advice) doit être exécuté autour du code qui a produit l'événement. Par exemple quand une exécution d'une méthode est capturée, le code de la consigne sera exécuté à la place de

cette méthode, et le code original de la méthode peut être activé explicitement en utilisant une primitive spéciale dite *proceed()*.

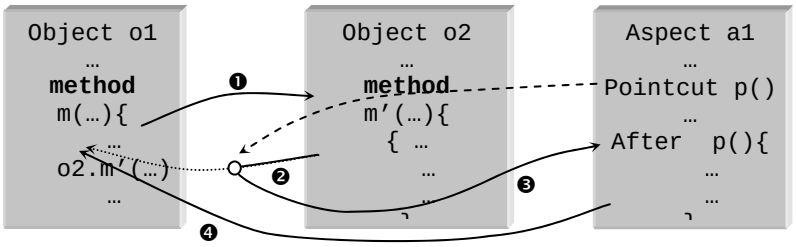


Fig. 2.3. Exemple de scénario d'exécution d'une consigne autour

De multiples consignes peuvent être appliquées pour le même point de jointure. Dans un tel cas, l'ordre d'exécution des consignes doit être spécifié pour la résolution des conflits qui peuvent apparaîtrent.

2.1.5. Les introductions

Le concept d'introduction nous permet de modifier la structure de l'application en introduisant de nouvelles variables d'instances ou en modifiant la structure d'héritage des classes.

2.2. CF: La Composition de Filtres

La composition de filtres est une technique de séparation des préoccupations qui a été introduite par M. Aksit et L. Bergmans [58-63]. Son principe de base consiste à ressortir hors de l'objet, tout comportement de manipulation des messages. Ceci va à l'encontre du modèle objet conventionnel où la manipulation des messages se réalise par le noyau de l'objet lui-même.

Afin d'assurer la séparation de la manipulation des message du comportement de l'objet, le noyau de l'objet est entouré par une couche qui l'englobe appelé interface.

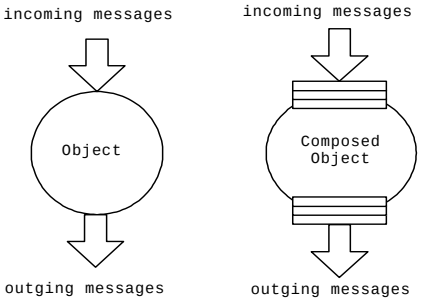


Fig. 2.4. L'objet ordinaire et l'objet de CF

Ce nouveau modèle vise le développement de l'abstraction de l'objet conventionnel par la modularité et l'orthogonalité :

La modularité : Signifie qu'un ou plusieurs filtres peuvent être attachés aux objets sans aucune modification de leurs définitions dans les langages. Elle assure aussi l'indépendance, entre la spécification des filtres et leurs

implémentation.

L'orthogonalité : Signifie que la sémantique de chaque filtre est indépendante aux autres, ce qui facilite leurs composition.

Ces deux propriétés distinguent l'approche CF des autres approches de séparation de préoccupations.

Un modèle CF se compose de deux partie : une partie noyau, et une partie interface.

2.2.1. La partie noyau

Cette partie est identique au modèle objet conventionnel. Cela signifie que la partie noyau englobe des méthodes et des variables d'instances comme la montre la figure 2.5.

Les méthodes apparaissent à l'extrémité du noyau pour permettre leur accès par les autres objets du système; en revanche, les variables d'instances sont totalement encapsulées dans le noyau, elles ne sont pas visibles aux autres objets de système.

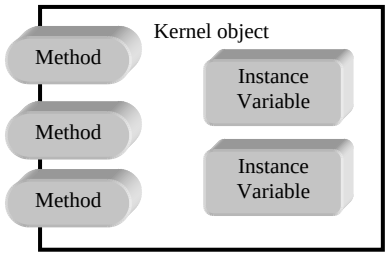


Fig. 2.5. La partie noyau du modèle CF

La partie noyau peut être implémenter en utilisant n'importe quel langage orienté objets conventionnel comme: *Java*, *C++*, *Smalltalk* ou C#.

2.2.2. La partie interface

La partie interface c'est la deuxième partie de modèle de CF, ses composants sont présentés dans la figure 2.6.

Les composants suivants peuvent être distinguer dans la partie interface de ce modèle : les objets internes, les objets externes, les filtres d'entées et les filtres de sorties.

Les objets internes. Sont totalement encapsulés dans l'interface du modèle et sont utilisés pour composer ses comportements avec le comportement de la partie noyau. A la réception d'un message par la partie interface de l'objet, il sera éventuellement redirigé vers un des objets internes. Ces objets sont automatiquement créés et encapsulés dès que l'objet CF est crée.

Les objets externes. Le objets externes sont situés à l'extérieur de le l'objet CF comme des objets globaux dont les références sont disponibles dans la partie interface. Ils sont utilisés généralement pour partager la responsabilité entre plusieurs entités. Des références aux objets externes doivent être

transmises à l'objet CF au moment de sa création.

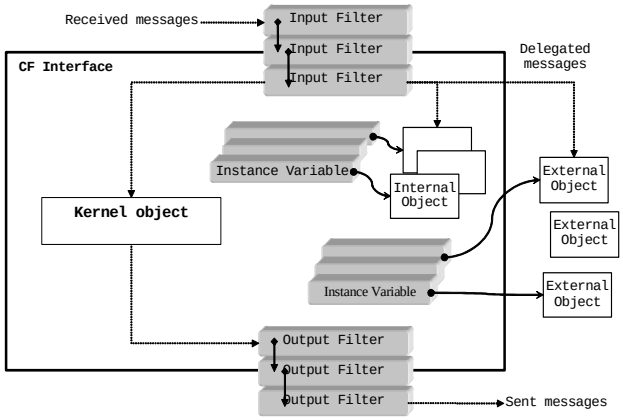


Fig. 2.6. La partie interface de modèle CF

Les filtres d'entrée et les filtres de sortie. Les filtres sont des structures de données définissant un comportement observable d'objets où chaque filtre spécifie un comportement particulier de manipulation des messages. Les filtres d'entrée (respectivement les filtres de sortie) se chargent de manipuler les messages reçus (respectivement émis) par l'objet CF.

2.2.3. L'aspect sémantique des filtres

Il existe différents types de filtres et à chaque type il est associé une spécification propre qui lui permet de déterminer les actions qui doivent être effectuées pour un message particulier. En général il existe cinq types de filtres:

- **Dispatch.** Quand le message est accepté, le filtre décide vers quel objet doit être redirigé ce message, par contre, si le message est rejeté le message passe au filtre suivant. Ce type de filtre fournit au modèle CF des techniques d'abstractions des données comme l'héritage simple, multiple et dynamique ainsi que la délégation.
- **Error.** Quand le message est accepté par un filtre, il lui permet de continuer vers le filtre suivant, sinon, le message doit être bloqué ; ce type de filtres fournit au modèle CF les techniques de contrôle multiples et des pré-conditions.
- **Wait.** Quand le message est accepté il passe au filtre suivant, sinon, l'objet CF le mette dans la queue d'une file d'attente jusqu'à la satisfaction des conditions qui le bloquent.
- **Meta.** Quand un message est rejeté par un filtre, il passe au filtre suivant, sinon, il le doit être réifié d'abord et envoyé à un autre objet pour le manipuler.
- **Substitute.** Quand un message est accepté, sa cible ou/et son sélecteur doit être substitué par d'autre valeurs, dans le cas contraire le message est passé au filtre suivant.

La Figure 2.7 présente clairement la sémantique de délégation et le mécanisme de substitution des filtres.

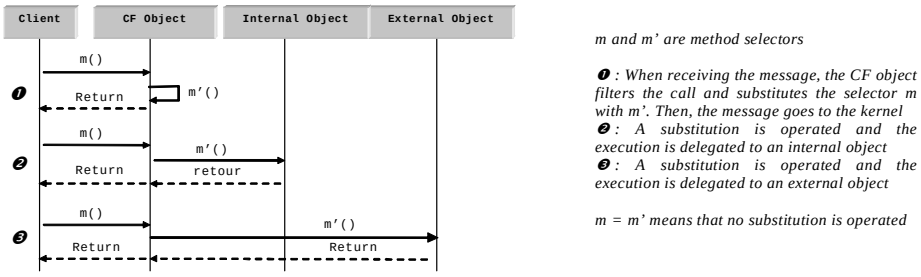


Fig. 2.7. Diagramme de séquence incluant les mécanismes de délégation et de substitution

2.2.4. Le mécanisme de la superimposition

La superimposition est une technique qui consiste à adjoindre systématiquement de nouveaux comportements à un objet à des endroits spécifiques [22].

Quatre concepts clés qui doivent être pris en considération pour satisfaire la superimposition :

- **Qui.** Le module du système qui sera superimposé
- **Quoi.** Le nouveau comportement à superimposé
- **Où.** Les endroits où le nouveau comportement sera superimposé
- **Quand.** Le moment dans lequel la superimposition doit être accomplie.

2.3 La séparation multidimensionnelle des préoccupations

Bien que l'appellation « séparation des préoccupations », soit peu utilisée, la notion sous-jacente a toujours été au coeur du génie logiciel, et tous les développeurs l'utilisent, parfois sans le savoir. En fait les développeurs procèdent souvent à cette séparation lorsqu'ils identifient, encapsulent et manipulent des parties pertinentes d'un logiciel. Les préoccupations permettent d'organiser et de décomposer le logiciel en des parties qui sont manipulables et compréhensibles.

Plusieurs types de préoccupations peuvent être identifiés, ceci dépend du développeur, leur rôle, ou une étape du cycle de vie du logiciel. Par exemple, dans le cas de la programmation orientée objet, le type de préoccupation utilisé est la classe, chaque préoccupation de ce type est un type de données défini et encapsulé par une classe.

On peut, bien entendu, identifier d'autres types de préoccupations, comme par exemple : des caractéristiques d'impression, de gestion de bases de données, des règles du marketing, de distribution et beaucoup d'autres selon le domaine d'application. On appelle un type de préoccupations comme les classes, ou tout autre type, une « dimension de préoccupation », et la séparation des

préoccupations consiste à la décomposition du logiciel en une ou plusieurs dimensions de préoccupations.

Une bonne séparation de préoccupations peut aider à :

- réduire la complexité du logiciel et le rendre plus compréhensible.
- limiter l'impact des changements, facilitant ainsi l'évolution et une adaptation non envahissante
- faciliter la réutilisation.
- simplifier l'intégration de nouveaux composants.

Ces buts, malgré leur importance, n'ont toujours pas été atteints en pratique. Ceci est dû principalement au fait que l'ensemble des préoccupations pertinentes change avec le temps et peut être influencé par plusieurs facteurs tels que : la diversité des activités de développement ou les étapes du cycle de vie du logiciel.

La décomposition en une seule dimension peut promouvoir quelques ambitions, mais elle peut aussi être un obstacle pour d'autres; ainsi, chaque critère de décomposition ou d'intégration sera approprié pour quelques spécifications mais pas pour la totalité. Par exemple la décomposition par classe dans le cas de l'orientée objet facilite beaucoup l'évolutions des détails de la structure de données parce qu'ils seront encapsulés dans des classes, mais rend difficile l'addition et l'évolution des variables d'instance et des méthodes, puisque celles-ci peuvent exister dans plusieurs classes et peuvent être liées les unes aux autres.

Hyperespace (Hyperespaces en anglais) est une approche qui permet une identification explicite de chaque dimension et chaque préoccupation, à n'importe quel étape du développement, l'encapsulation de ces préoccupations, la gestion des relations entre ces préoccupations et leur intégration.

Dans cette approche, l'accent est mis sur 4 concepts principaux:

- préoccupation, (Cocnern)
- hyperespace, (Hyperspace)
- Hyper-module, (Hypermodule)
- Hyper-tranche, (Hyperslice)

Cette méthode est le fruit du travail du laboratoire *alphaworks* d'*IBM*, qui ont sorti leur produit *Hyper/J* qui implémente l'approche *hyperespace* pour le langage *Java*, et entre aussi dans le cadre de la suite des travaux commencés sur les méthodes orientées sujet (Subject Oriented Programming) du même laboratoire.

La figure 2.8 illustre l'approche hyperespace, nous expliquons les différents concepts juste après.

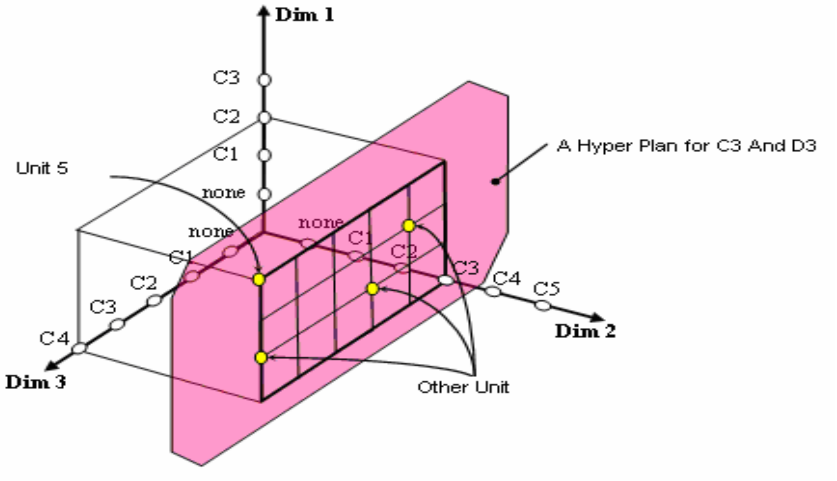


Fig. 2.8. Exemple d'un hyperespace à trios dimensions

Cette partie donne une définition intuitive de l'approche, ainsi que quelques détails.

2.3.1 L'espace des préoccupations

Le logiciel est composé de plusieurs parties qui sont formulées dans un certain langage. Une unité est une construction syntaxique dérivée d'un tel langage. Elle peut être, par exemple, une méthode, une variable d'instance, une procédure, une fonction, une règle, une classe, une interface, etc. Cependant, on distingue des unités simples qui sont considérées comme étant atomiques, et des unités composées qui sont des regroupements d'unités simples. Ainsi les méthodes, les variables d'instance, les fonctions sont toutes des unités simples tandis que les classes, et les paquets sont des unités composées.

L'espace des préoccupations renferme toutes les unités dans une partie du logiciel, et son rôle est d'organiser les unités de manière à séparer les préoccupations cruciales, décrire plusieurs types de relations, et indiquer comment les systèmes et les composants du logiciel peuvent être construits et intégrés à partir des unités contenues dans ces préoccupations.

Le processus de la séparation multidimensionnelle de préoccupations comprend trois étapes importantes :

- **Identification.** C'est le processus de sélection de préoccupations ainsi que les unités que renferment ces préoccupations.
- **Encapsulation.** L'identification est utile, mais pour réaliser entièrement les avantages de la séparation des préoccupations, celles-ci doivent également être encapsulés de sorte à ce qu'elles puissent être manipulées en tant qu'entités de première classe. Une classe *Java* est un exemple d'une préoccupation encapsulée.
- **Intégration.** Une fois les préoccupations encapsulées, on doit pouvoir les intégrer pour créer un logiciel capable de manipuler plusieurs préoccupations. Dans *Java,* les classes sont intégrées en les chargeant. Les

préoccupations autres que les classes et les interfaces ne peuvent être intégrées dans *Java*.

2.3.2 Identification de préoccupations (La matrice des préoccupations)

Un hyperespace est un espace de préoccupations spécialement conçu pour supporter la séparation multidimensionnelle de préoccupations. Ses unités sont organisées dans une matrice multidimensionnelle, chacun de ses axes représente une dimension de préoccupations, et chaque point dans un axe, une préoccupation dans cette dimension. Ceci rend explicite l'identification de toutes les dimensions d'intérêt, les préoccupations qui appartiennent à chaque dimension, et quelles préoccupations sont affectées par quelles unités. Les coordonnées d'une unité indiquent toutes les préoccupations qu'elle affecte; et cette structure montre bien que chaque unités affecte au plus une seule préoccupation pour une dimension donnée.

Chaque dimension, peut être aperçue ainsi comme une partition de l'ensemble de toutes les unités (i.e. l'espace des préoccupations) : une décomposition particulière du logiciel. Chaque préoccupation dans une dimension définit un « hyperplan » qui contient toutes les unités qui affectent cette préoccupation. La structure de la matrice permet un traitement uniforme de tous les types de préoccupations, et permet aux développeurs de manipuler et de trancher à travers la matrice selon les préoccupations désirées.

Il peut arriver qu'une ou plusieurs unités n'appartiennent à aucune préoccupation, l'approche hyperespace a prévue pour cela une préoccupation appelée *none* pour chaque dimension, qui renferme toutes les unités qui appartiennent à l'espace des unités, mais qui n'affectent aucune préoccupation de cette dimension, donc la préoccupation *none* contient généralement les unités qui ne présentent aucun intérêt dans une dimension donnée.

2.3.3 Encapsulation des préoccupations (Les Hyper-tranches)

La matrice des préoccupations identifie les préoccupations et organise les unités selon les dimensions et les préoccupations, mais elle ne permet pas l'encapsulation de ces dernières, c'est à dire que les ensembles d'unités résultants ne peuvent pas être traités comme des modules, sans un mécanisme additionnel. Dans l'approche hyperespace, ce mécanisme est appelé hyper tranche (*hyperslice*) [40] qui est un ensemble de préoccupations « complets en déclaration», ce qui veut dire qu'elles déclarent tout ce à quoi elles font référence.

Les unités sont souvent reliées entre elles de différentes manières, une méthode peut invoquer une autre ou peut utiliser une variable d'instance, et quand ce genre de relations existe un fort couplage en résulte. La complétude déclarative est une propriété des hyper-tranches qui a pour but d'éliminer ce fort couplage qui peut exister entre différentes hyper-tranches [40]. La complétude déclarative inclut que chaque hyper-tranche doit au minimum déclarer chaque méthode que ces unités membres invoquent, et chaque variable que ces membres utilisent, une définition complète n'est pas requise pour ces déclarations, par exemple, on peut déclarer des méthode sans les implémenter, ainsi cette déclaration peut être abstraite.

La complétude déclarative est très importante puisqu'elle élimine le couplage entre hyper-tranches, donc au lieu d'avoir une hyper-tranche qui fait référence à une autre, chacune d'elles définira ce dont elle a besoin, au moyen de la déclaration abstraite des méthodes et des variables que ses unités invoquent ou utilisent.

L'implémentation des unités qui ont été déclarées abstraites se fera plus tard dans l'intégration.

Donc, chaque ensemble d'unités complet en déclaration est une hyper-tranche ce qui implique qu'une hyper-tranche est simplement une préoccupation plus, éventuellement les déclarations abstraites d'unités que ses propres unités utilisent et qui appartiennent à d'autres hyper-tranches.

2.3.4 Intégration des préoccupations (Les Hyper-Modules)

Les hyper-tranches construisent des blocs, qui peuvent être intégrés pour former d'autres blocs. Ceci est fait par des relations d'intégration. Il arrive parfois qu'on soit ramené à utiliser une hyper-tranche qui a quelques déclarations abstraites, et qu'on utilise une autre hyper-tranche qui contient l'implémentation complète de ces déclarations, et en profiter pour implémenter tout ce qui a été déclaré abstrait. Ce genre de relations s'appelle la correspondance. Dans un hyperespace, ce contexte d'intégration s'appelle hyper-module.

Un hyper-module est donc un ensemble de hyper-tranches qui vont être intégrées, et des relations d'intégration qui spécifient la manière dont les hyper-tranches doivent être intégrées. La correspondance est une relation d'intégration très importante qui indique quelles unités des différentes hyper-tranches vont être intégrées. D'autres détails peuvent être précisés, comme par exemple, si deux méthodes correspondent, est ce que l'une doit couvrir l'autre, ou est ce qu'elles doivent être exécutées ensembles, et si c'est le cas dans quel ordre ? Comment la valeur de retour sera t-elle calculée?

Il existe en pratique, des outils qui permettent de réaliser l'intégration selon les relations d'intégration précédemment décrites, donc on peut procéder à la production d'un ensemble d'unités intégrées. Cet ensemble sera bien entendu complet en déclaration et pourra être considéré comme une hyper-tranche composée, obtenu par l'intégration de nombreuses hyper-tranches subordonnées, ce qui permet à de grands systèmes, complets, d'être construits en partant d'un ensemble de plusieurs hyper-tranches et de relations d'intégrations successives.

La complétude déclarative, la correspondance et les relations d'intégration sont des caractéristiques qui peuvent promouvoir l'évolution d'un système logiciel [40], et puisqu'une hyper-tranche ne dépend pas d'une autre directement, les différents artifices sont sujet à une contrainte de complétude dans laquelle chaque définition d'une unité doit correspondre à, soit une ou plusieurs définition(s), éventuellement abstraite(s), soit à une ou plusieurs implémentation(s) appartenant à une ou plusieurs hyper-tranche(s). Changer une définition ou une implémentation devient alors non envahissant, la correspondance nous procure une certaine flexibilité et supporte directement la substitution des unités au sein d'une ou plusieurs hyper-tranche(s).

2.4 Les Systèmes Multi-Agents

Un système Multi-Agents peut être vu comme un environnement artificiel qui comporte un ensemble des entités actives qui coopèrent entre elles pour atteindre un objectif donnée, et un ensemble d'objets inactifs qui seront considérés comme étant des outils manipulables.

Ferber [44] a définie un agent comme étant « une entité physique ou virtuelle capable de:

- être active dans un environnement.
- communiquer avec d'autres agents.
- conduit par un ensemble de tendances (dans la forme d'objectifs individuels ou d'une fonction de satisfaction/survie qu'elle essaie d'optimiser).
- posséder ses propres ressources.
- percevoir son environnement (mais à une échelle limitée).
- seulement une représentation partielle de son environnement qui a été disponible pour un agent (peut-être aucune).
- posséder des compétences et offrir des services.
- se reproduire.

Le comportement d'un agent essaye de satisfaire ses objectifs, en tenant compte des ressources, de sa compétence, de sa représentation ainsi que des communications qu'il reçoit. » [44]

Certaines propriétés communes rendent les agents différents des objets conventionnels:

- les agents sont autonomes. Donc, l'agent doit être capable de faire des actions préventives ou indépendantes qui bénéficieront l'utilisateur.
- les agents possèdent un certain niveau d'intelligence.
- les agents n'agissent pas uniquement d'une manière réactive, mais parfois ils agissent proactivement.
- les agents ont une capacité sociale qui leurs permettent de communiquer avec l'utilisateur, le système, et d'autres agents.
- les agents peuvent aussi coopérer avec les autres agents pour exécuter des tâches plus complexes.
- les agents peuvent se déplacer d'un système à un autre pour accéder à des ressources éloignées ou même rencontrer d'autres agents.

Il y a plusieurs architecture d'agents: BDI, Intelligents, Cognitives, et Réactivés, on s'intéresse dans ce travail aux agents réactifs qui se comportent comme un système orienté objets classiques.

L'idée derrière les agents réactifs est qu'aucun agent ne perçoit parfaitement son environnement. Des événements aléatoires ou une indisponibilité peut

handicaper l'agent. Donc, un agent réactif peut être vue comme étant un objet ordinaire qui réagit seulement avec les événements de perception.

3. Conclusion

Dans ce chapitre nous avons introduire toutes les techniques et les approches utilisés dans notre recherche: Les patrons de conception, les approches avancées de séparation des préoccupations ainsi que les systèmes multi-agents réactifs.

Les patrons de conceptions et les approches ASOC sont des paradigmes prometteurs qui visent à résoudre les problèmes récurrents dans la conception orienté objets. Cependant, même si beaucoup de littérature a été consacrée à ces paradigmes, peu est fait pour leur comparaison, évaluation et intégration

Dans le prochain chapitre on s'intéresse à la représentation des patrons de conception sous les différentes approches ASOC.

Chapitre 3

L'IMPLEMENTATION DES PATRONS DE CONCEPTION PAR LES APPROCHES ASOC

Dans ce chapitre, nous présentons les mises en oeuvre des trois patrons de conception: Adapter, Singleton et Chaîne de Responsabilité sous les différentes approches ASOC. Une description détaillée de ces patrons est donnée en annexe 1.

1. Mise en Oeuvre des Patrons de Conception en AOP

Aspect/J est une extension du langage Java qui lui permet de supporter les concepts de l'approche orientée aspect. Ce nouveau modèle propose deux niveaux d'abstractions: le niveau de classes et le niveau d'aspects, le niveau de classes implémente la partie fonctionnelle de l'application, et le niveau d'aspects implémente les différentes préoccupations requises. AspectJ propose aussi un mécanisme de tissage qui lui permet de composer les deux niveaux ensemble pour obtenir le code final de l'application.

Une des techniques d'implémentation des patrons sous AOP consiste à déclarer un aspect abstrait pour chaque patron qui définit les différents rôles de leurs participants.

Dans cette section nous présentons la mise en oeuvre des trois patrons de conception choisie en utilisant AspectJ. Nous utilisons pour cela, l'extension UML qui permet de supporter la représentation les aspects.

1.1. Le Patron Adapter

Une implémentation possible du patron Adapter en AspectJ consiste a spécifier une interface appelée adaptee, qui inclut une déclaration de tout le contexte d'adaptation, spécifier la classe AdaptedContext qui implémente cette interface, et rendre le nouveau contexte accessible aux autres objets de système.

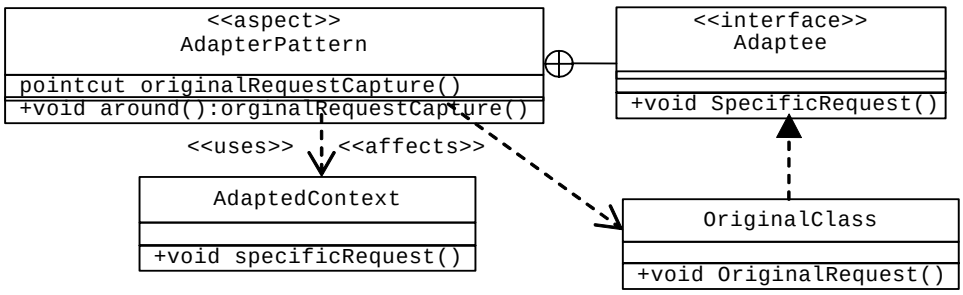


Fig 3.1. Une mise en oeuvre du patron Adapter en AOP

L'aspect AdapterPattern rend la classe originalClass une implémentation de l'interface Adaptee. Dans ce cas, chaque appel aux méthodes, originalRequest(), sera accepté par la classe et capturé par le pointcut originalRequestCapture de l'aspect AdapterPattern. Ensuite, il sera redirigé vers la classe AdaptedContext pour être exécuté. La classe AdaptedContext inclut une spécification du nouveau contexte. Le listing suivant donne le code complet de l'aspect AdapterPattern.

```
public aspect AdapterPattern {
     public interface Adaptee {
        public void specificRequest();
     }
     Declare parents:originalClass implements Adaptee;
     public void oiginalClass.specificRequest() {
            AdaptedContext.specificRequest();
     }
 }
```

Listing 3.1. Le patron Adapter en AspectJ

1.2. Le Patron ChaînOfResponsibility

Le patron chaîne de Responsabilité peut être implémenté en AspectJ en spécifiant un aspect abstrait qui spécifie une interface Handler commune qui doit être implémentée par tous les objets membres de la chaîne. L'interface Handler inclut une déclaration des deux principales méthodes: acceptRequest () et handleRequest () qui donne respectivement quand la requête doit être acceptée et comment doit être traitée.

Le vecteur successeurs présente la structure de données représentant la chaîne, et les deux méthodes: setSuccessors et getSuccessors sont utilisés respectivement pour ajouter un membre dans la chaîne et de retourner un membre de cette celle-ci.

Le point de coupure eventTrigger est utilisé pour capturer chaque requête passée à la chaîne. Une consigne après est utilisé pour laisser la requête passer au membre suivant dans la chaîne si le membre courant n'est pas capable de répondre à la requête.

Le patron ici est implémenté par un aspect concret appelé Chain qui affecte toutes les classes membres de la chaîne.

Le Boolean alreadyHandledRequest est utilisé pour spécifier si la requête a été bien reçue par le membre ou pas.

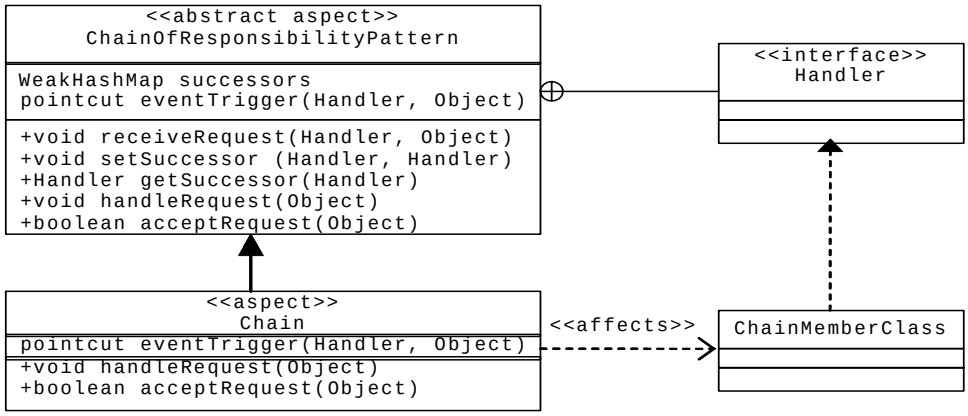


Fig 3.2. Une mise en oeuvre du patron Chaîne de responsabilité en AOP

Le code suivant montre comment les aspects ChainOfResponsibility et Chain peuvent être implémentés en AspectJ.

```
import java.util.WeakHashMap;
public abstract aspect ChainOfResponsibilityPattern {
 protected interface Handler {}
 private WeakHashMap successors = new WeakHashMap();
 Protected void receiveRequest(Handler handler, Object request){
      if (handler.acceptRequest(request)) {
              handler.handleRequest(request);
      } else {
                  // The handler will not accept the request
                   Handler successor = getSuccessor(handler);
        if (successor == null) {

      // The last handler in the chain must deal with the request
      // This is a rudimentary implementation and more complex
      // logic could be applied here or perhaps in the concrete
      // aspects that extend this abstract one

                  handler.handleRequest(request);

        } else {
       // Handle the request on to the next successor in the chain
                  receiveRequest(successor, request);
         }
        }
   }
   public boolean Handler.acceptRequest(Object request) {

      // The default as defined here is to reject the request
      // This is implemented by the application specific
      // concrete aspects

        return false;
   }
   public void Handler.handleRequest(Object request) {

     // A default empty implementation that is overridden
     // if required by the application specific concrete aspects
```

```
   }
    protected abstract pointcut eventTrigger(Handler handler, Object
    request);

    after(Handler handler, Object request):eventTrigger(handler, request) {
                receiveRequest(handler, request);
   }
 public void setSuccessor(Handler handler, Handler successor) {
             successors.put(handler, successor);
 }
  public Handler getSuccessor(Handler handler) {
            return ((Handler) successors.get(handler));
 }
}

public aspect Chain extends ChainOfResponsibilityPattern {

    declare parents : Handler1 implements Handler;
    declare parents : Handler2 implements Handler;
    declare parents : HandlerN implements Handler;

 Protected pointcut eventTrigger(Handler handler, Object event):
   // an implementation code of the pointcut
   Private boolean Handler.alreadyHandledRequest = false;
   Public boolean Handler.acceptRequest(Object event) {
       return !this.alreadyHandledRequest;
   }
   public void Handler1.handleRequest(Object event) {
       if (!this.acceptRequest(event)) {
           // Correspond code to the accepting event by Handler1
       }

       this.alreadyHandledRequest = true;
   }
      public void HandlerN.handleRequest(Object event) {
       if (!this.acceptRequest(event)) {
          // Correspond code to the accepting event by HandlerN
       }
       this.alreadyHandledRequest = true;
  }
 }
```

Listing 3.2. Le patron Chaîne de Responsabilité en AspectJ

1.3. Le patron Singleton

Le patron singleton peut être implémenté en AOP par une déclaration d'un aspect qui prend en charge le lancement de l'exception dès que une nouvelle demande d'instanciation de la classe du singleton est effectuée. Le diagramme de classe et le code suivant montrent comment cette idée d'implémentation peut être mise en oeuvre en AspectJ.

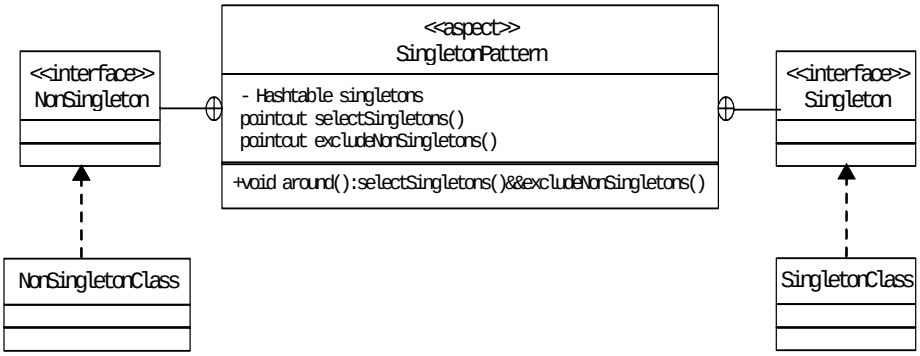


Fig 3.3. Une mise en oeuvre du patron singleton en AOP

```
import java.util.Hashtable;

public abstract aspect SingletonPattern issingleton() {
   protected Hashtable singletons = new Hashtable();
   Public interface Singleton {}
   Public interface NonSingleton {}

   // Pointcut to define specify an interest in all creations
   // of all Classes that extend Singleton

   pointcut selectSingletons() : call((Singleton +).new (..));

    /* Pointcut to ensure that any classes in the Singleton inheritance tree
    that are marked as Non Singletons are not included in the Singleton
    logic.*/

pointcut excludeNonSingletons():!call((NonSingleton+).new (..));

Object around() : selectSingletons() && excludeNonSingletons() {
     Class type = thisJoinPoint.getSignature().getDeclaringType();
         synchronized(singletons) {
                if (singletons.get(type) == null) {
                        singletons.put(type, proceed());
                }
         }
   return (Object) singletons.get(type);
}
}
public aspect Singleton extends SingletonPattern {

   declare parents: class1 implements Singleton;
   declare parents: class2 implements NonSingleton;
}
```

Listing 3.3. Le patron singleton en AspectJ

L'aspect abstrait *SingletonPattern* definie deux rôles: *Singleton* et *NonSingleton*. Ces deux rôles sont implémentés sous forme d'interfaces afin que l'aspect abstrait puisse travailler avec les singletons sans s'inquiéter du détails de la mise en oeuvre.

L'interface singleton est affectée à toute classe limitée à un seul objet (singleton) par un sous aspect de l'aspect abstrait *SingletonPattern.* De même, l'interface *NonSingleton* est affectée a toute classe qu'on souhaite maintenir normale (c à d classe non singleton).

Deux points de coupure *selectSingletons()* et *excludeNonSingletons()* sont déclarés afin de capturer respectivement tout appel aux constructeurs des classes qui implémentent l'interface singleton et celle qui implémentent l'interface NonSingleton.

Une consigne autour a été utilisée, cette dernière contrôle l'accès aux constructeurs et lance l'exécution d'une routine d'exception si le constructeur a déjà été appelé.

2. Mise en Oeuvre des Patrons de Conception en CF

L'approche de composition de filtres est une extension modulaire aux langages objets tel que Java [51], C++ [68] et Smalltalk [94]. Vu que toutes les interactions du système objets sont faites en envoyant des messages, la manipulation des messages reçus sont à la charge de l'objet ce qui complique extrêmement le comportement de ce dernier. Pour séparer la manipulation des messages reçus du comportement de l'objet, une couche extérieure appelée interface, qui enveloppe l'objet, a été mise en oeuvre.

Dans cette section, nous entreprenons la description de nos trois patrons de conception choisis en utilisant le modèle CF qui est considéré l'une des meilleures stratégies proposées par la communauté ASOC. Cette description de la mise en oeuvre des patrons a été présentée par une extension au diagramme de classe d'UML, cette extension est basée sur les règles suivantes:

Le rectangle jaune avec le stéréotype "Concern" représente une spécification de la préoccupation.

La partie noyau sera présentée par un rectangle blanc.

Une Boîte bleue avec flèches entrant et sortant corresponde un ensemble de filtres d'entrée.

La croix blanche dans la préoccupation symbolise la spécification de la superposition qui dénote un ensemble d'objets où l'interface des filtres est insérée.

Les lignes rouges interrompues indiquent le processus de superimposition.

2.1. Le patron Adapter

Pour décrire le patron Adapter en utilisant le modèle CF, nous proposons de considérer l'Adapter comme une préoccupation particulière qui inclut une déclaration d'une interface de filtres qui se charge de déléguer tous les messages reçus, concernant un contexte qui n'est pas pris en charge par l'objet noyau, à un objet externe de la classe AdaptedContext, qui comporte le contexte d'adaptation. Les filtres de types substitution et dispatch prennent en charge cette redirection de messages. Voici le diagramme de classe et le code

du patron Adapter.

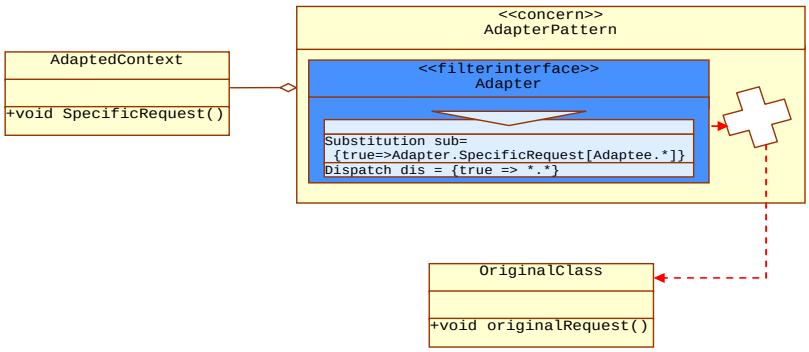


Fig 3.4. . Une mise en oeuvre du patron Adapter en CF

```
concern AdapterPattern begin
     filterinterface Adapter begin
          Externals
            Adaptee adaptee;
          Inputfilters
              Substitution sub =
             {true => Adapter.SpecificRequest [Adaptee.*]}
             Dispatch dis = {true => *.*}
     end filterinterface Adapter;

   superimposition begin
         Selectors
             AdapterObject = { *= classAdapter;}
         Filterinterface
            AdapterObject <- self::Adapter;
   end superimposition;
 end concern AdapterPattern;
```

Listing 3.4. Le patron Adapter en ConcernJ

2.2. Le patron ChaînOfResponsibility

Le patron chaîne de responsabilité peut être mis en oeuvre par le modèle CF comme étant un concern en ConecrnJ. Comme montré dans la figure 3.5, ce concern se compose de deux parties: une partie partagée qui implémente la chaîne et une autre entrecoupante qui assurent que les objets membres de la chaîne utiliseront l'objet Chain pour sélectionner le prochain candidat membre de la chaîne si le membre courant n'est pas capable de manipuler la requête reçue.

L'interface de filtres définie un filtre *Meta* qui se charge d'intercepter les requêtes et les envoyer sous une forme réifiée a l'objet Chain qui est chargé de trouver le prochain membre de la chaîne candidat pour la manipulation de la requête reçue. Le filtre *Dispatch* accepte tous les messages rejetés par le *Meta*

filtre.

L'interface de filtres *ChainEngine* et la partie noyau constituent ensemble notre chaîne. La partie de *superimposition* spécifie le concern entrecoupant les autres parties du système.

Le problème de cette implémentation réside dans la spécification de la partie noyau. Selon cette idée d'implémentation, en doit spécifier dans la partie noyau les deux méthodse *AcceptHandler* et *HandleRequest* pour chaque membre de la chaîne dans la même classe, ce qui implique une sorte de redondance et de conflit entre les différentes déclarations. Une solution à ce problème consiste à partager la partie noyau entre plusieurs classes, malheureusement, cette solution a été décrite par les concepteurs du modèle mais elle n'est pas encore implémentée dans la version actuelle de ConcernJ.

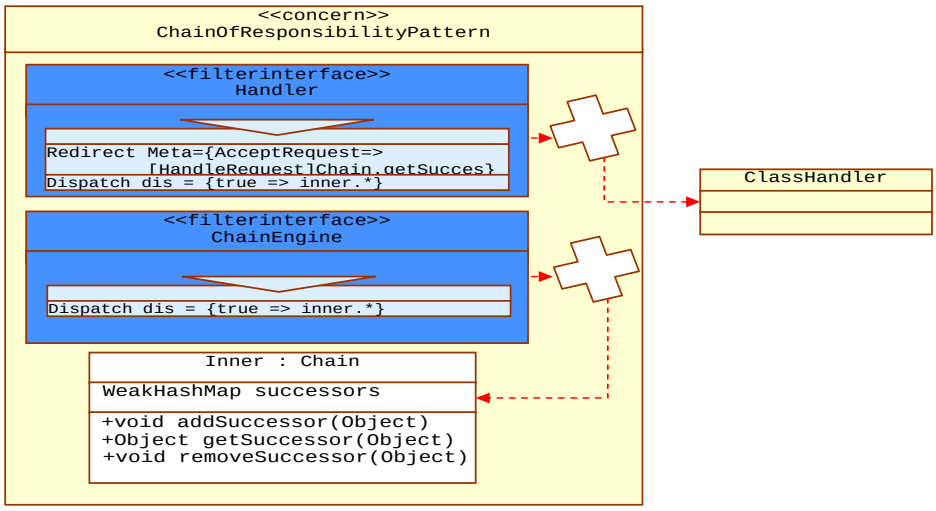


Fig 3.5. Une mise en oeuvre du patron Chaîne de Responsabilité en CF

```
concern ChainOfResponsibilityPattern begin

   filterinterface Handler begin
   // this part declares the crosscutting code
      Externals
        Chain : ChainOfResponsibility;
        // declare a shared instance of this concern
      Inputfilters
          redirect : Meta = {
          AcceptRequest =>[HandleRequest]Chain.getSuccessor };
        disp : Dispatch = { inner.* };
          /* dis filter accept all methods implemented by myself*/
   end filterinterface Handler;

   filterinterface ChainEngine begin
   //defines the interface of the Chain object
      Methods
        getSuccessor(Object);
        addSuccessor(Object);
        removeSuccessor(Object);
```

```
      Inputfilters
        Disp : Dispatch = { inner.* };
        // accept all methods implemented by the kernel object
    end filterinterface CahinEngine;

    superimposition begin
      Selectors
        allHandler = { *=ClassHandler1,...,*=ClassHandlerN};
      Filterinterfaces
         self <- self::ChainEngine;
         allHandler <- self::Handler;
    end superimposition;

    implementation in Java;
      class Chain {
         private WeakHashMap successors = new WeakHashMap();
         void addSuccessor(Object o) { … };
         Object getSuccessor(Object o) { … };
         void removeSuccessor(Object o) { … };
      }
    end implementation;

end concern ChainOfResponsibilityPattrn;
```

Listing 3.5. Le patron Chaîne de Responsabilité en ConcernJ

2.3. Le patron Singleton

Le patron singleton peut être implémenté en CF en spécifiant un concern qui comporte une déclaration d'un interface de filtres composée d'un filtre de type Error qui rejette tout appel au constructeur de l'objet CF si la condition instanceFlag est satisfaite. La condition instanceFlag est évalue à vrai si le constructeur de l'objet à déjà été exécuté.

La partie de superimposition à été utilisée afin mettre ce comportement à toute les classes singleton.

Le diagramme de classe et le code suivants mettent en évidence la mise en œuvre du patron singleton en CF en utilisant le modèle ConcernJ.

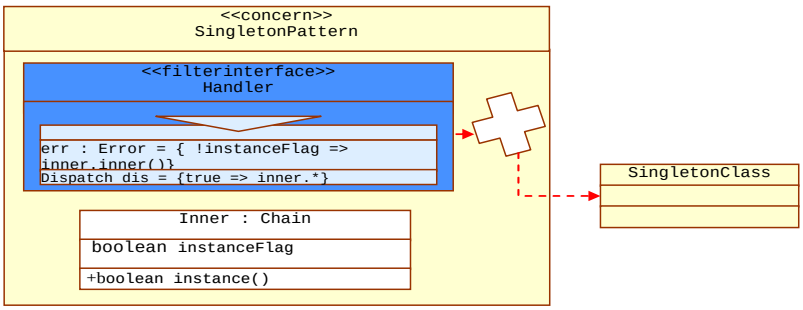


Fig 3.6. Une mise en oeuvre du patron singleton en CF

```
concern SingletonPattern begin
```

```
filterinterface Singleton begin
   Conditions
     instanceFlag;
   Methods
     instance();
   Inputfilters
     err : Error = { !instanceFlag => inner.constructor()};
     disp : Dispatch = { inner.* };
     // accept all methods implemented by myself
   end filterinterface Singleton;

   superimposition begin
      Selectors
          allSingleton = {
          *=ClassSingleton1,...,*=ClassSingletonN};

     Conditions
        allSingleton <- self::instanceFlag;
     Methods
       allSingleton <- self::instance();
     Filterinterfaces
       allSingleton <- self::Singleton;
   end superimposition;

   implementation in Java;
       class Singleton {
            public boolean instanceFlag = instance();
            private boolean instance() {
                if !instanceFlag {
                        instanceFlag = true;
                        return !instanceFlag;
                } else return instanceFlag;
            }
       }
   end implementation;

end concern SingletonPattern;
```

Listing 3.6. Le patron singleton en ConcernJ

3. Les patrons de conception dans l'approche hyperespace

Hyper/J est un environnement qui supporte l'approche des hyperespaces dans au niveau du langage standard Java. Lors de l'utilisation d'hyperJ, les développeurs fournissent trios entrées: un fichier des hyperespace qui décrit les fichiers des classes Java à composer, un mapping des préoccupations qui décrit des éléments dans ces fichiers Java (e.g. packages, classes, opération) qui seront projetés sur les préoccupations, et un fichier hypermodule qui décrit comment l'intégration entre préoccupations devrait être faite. Dans cette section nous décrirons les patrons sélectionnés qui utilisent *Hyper/J.*

3.1. Le patron Adapter

En Hyper/J, nous pouvons décrire le patron Adapter sous forme de deux classes: classe adapter qui décrit le contexte d'adaptation, et une classe

Adaptee qui décrit le contexte initial, à ce moment là aucune spécification entre ces deux classes n'a encore été spécifiée. Le fichier de l'hypermodule spécifie les rapports entre ces deux classes. Le type de relation utilisé ici est mergeByName qui désigne que toutes les méthodes qui ont le même nom sont fusionnées en une seule. Par conséquent, faire correspondre les deux méthodes de notre patron request et specificRequest doit être signalé explicitement. Pour cette raison, le rapport equate a été utilisé. Le code suivant décrit comment le patron Adapter peut être mise en oeuvre en Hyper/J.

```
class Adapter {
   public void request() {
      // the request specification code
   }
}

class Adaptee {
   public void specificRequest() {
      // the specificRequest specification code
   }
}
// hyperspace file

Adapter
Adaptee

// concern mapping file

class Adapter: Feature.kernel
class Adaptee: Feature.adaptation

// hypermodule file

hypermodule AdapterPattern
   hyperspaces:
      Feature.kernel;
      Feature.adaptation;

   relationships:
      mergeByName;
      equate class Feature.kernel.Adapter,
                   Feature.adaptation.Adaptee;

end hypermodule
```

Listing 3.7. Le patron Adapter en Hyper/J

3.2. Le patron ChaînOfReponsibility

Le patron Chaîne de Responsabilité peut être mise en oeuvre en utilisant Hyper/J en fusionnant toutes les classes membres de la chaîne avec la classe Handler. La classe Handler contient une méthode handleRequest qui implémente le comportement de traitement des requêtes reçues ainsi que la méthode handlerBehavior. Cette dernière méthode est une méthode vide utilisée pour forcer la correspondance avec la méthode handlerBehaviorMethod_handler de la classe handlerBehavior. Cet assortiment doit être spécifié un par un pour chaque membre de la chaîne. Le code suivant

montre clairement comment cette implémentation peut être faite en Hyper/J.

```
class Handler {
    public void handleRequest(){}
}

class handleBehavior {
   public void handlerBehaviorMethod_Handler(){
   // the handleRequest behavior of the specific handler Handler
   }
}

// hyperspace file

hyperspace ChainOfResonsibility

packageHandler.*;
Handler;
handlerBehavior;

// concern mapping file

Package Handler.*: Feature.kernel
class Handler: Feature.ChainOfResponsibility
class handlerBehavior: Feature.ChainOfResponsibility

// hypermodule file

hypermodule ChainOfResponsibility

   hyperspaces:
      Feature.kernel;
      Feature.ChainOfResponsibility;

   relationships:
      mergeByName;
      equate class Feature.kernel.*,
                   Feature.ChainOfResponsibility.Handler;
      equate operation Feature.kernel.Handler.handleRequest,
                           Feature.CahinOfResponsibility.handerBehaviorMetho
                           d_handler;
end hypermodule;
```

Listing 3.8. Le patron Chaîne de Responsabilité en Hyper/J

3.3. The Singleton Pattern

Le patron singleton peut être mis en oeuvre utilisant Hyper/J en étendant le constructeur de la classe singleton par le booléen instanceFlag qui prend la valeur vrai si cette classe n'a pas été instantiée auparavant. A l'exécution du constructeur la valeur de l'instanceFlag doit être d'abord évaluée, si la valeur de cette instance correspond à la valeur vrai le constructeur doit continuer son exécution, sinon il doit terminer son exécution par le lancement d'une routine d'interruption.

Cette implémentation peut être spécifiée en Hyper/J par l'utilisation du rapport mergeByName entre la classe singleton et une classe intermédiaire qui spécifie

le contexte singleton. Cette nouvelle classe contient la déclaration de l'instanceFlag ainsi qu'une déclaration du nouveau contexte du constructeur. Le code suivant décrit cette mise en oeuvre.

```
class OriginalClass {
     // the SingletonClass specification code
}

class Singleton {
     Private Boolean instanceFlag = false;
     public void OriginalClass() {
           If (!instanceFlag) {
                 Throws new Exception("This class cannot be instantiated
                 more than once");
           } else instanceFlag = true;
     }
}

// hyperspace file

OrginalClass
Singleton

// concern mapping file

class OriginalClass: Feature.kernel
class Singleton: Feature.singleton
operation OriginalClass.OriginalClass: Feature.kernel;
operation Singleton.OriginalClass: Feature.singleton;
// hypermodule file
hypermodule AdapterPattern

   hyperspaces:
      Feature.kernel;
      Feature.singleton;

   relationships:
      mergeByName;
      equate class Feature.kernel.OriginalClass,
                   Feature.singleton.Singleton;
      order operation Feature.singleton.OriginalClass, before
                      Feature.kernel.OriginalClass;

end hypermodule;
```

Listing 3.9. Le patron Singleton en Hyper/J

4. Conclusion

Dans ce chapitre nous avons décrit les trois patrons de conception choisis sous différentes approches ASOC, ce qui montre que les patrons ne sont pas liés à un paradigme particulier du génie logiciel mais ils peuvent être mis en oeuvre en utilisant les différents paradigmes. Notre but est de comparer ces différentes mises en oeuvre en utilisant quelques métriques qualitatives et quantitatives qui est l'objectif du prochain chapitre.

Chapitre 4

ASOC: UNE ETUDE COMPARATIVE BASEE SUR LES PATRONS DE CONCEPTION

Dans le contexte des systèmes multi-agents réactifs, le paradigme orienté objets offre des solutions flexibles aux problèmes communs dans le développement des systèmes. Malheureusement, le paradigme orienté objet est souvent incapable de modulariser certaines préoccupations entrecoupantes, ce qui diminue le taux de réutilisation, de maintenance et de l'évolution de ces systèmes. De là, vérifier si les approches ASOC sont capables d'améliorer la modularisation de ces préoccupations, revêt une importance capitale et représente, de ce fait, le but principal de ce chapitre.

Notre première idée a consisté à effectuer une comparaison conceptuelle des approches ASOC basée sur un mapping des concepts des différentes approches [10, 13]. Malheureusement, cette approche ne peut pas satisfaire pleinement notre objectif à cause de la complexité de certaines approches ASOC et en particulier l'approche de séparation multidimensionnelle des préoccupations. En effet, pour celle-ci, les concepts sont très divers et le mapping ne sera pas suffisant pour une comparaison efficace. Pour cette raison, nous avons opté pour une approche complémentaire, qui consiste en une étude qualitative et quantitative. De plus, pour rendre cette comparaison plus révélatrice nous avons choisi les patrons de conception comme benchmarks à implémenter par les différentes approches ASOC.

1. Le Mapping des Concepts

L'idée derrière le mapping est de répondre à la question: Pour chaque spécification particulière d'une approche, quel est la spécification correspondante dans les autres? Dans [10, 13], nous avons essayé d'identifier la correspondance entre les deux approches AOP et CF, dans ce dernier travail, nous avons trouvé qu'un système CF correspond à un ensemble d'unités où chacune représente une classe étendue par l'ajout d'une interface particulière. Pour faciliter l'explication de cette première idée, on suppose que CFU est une unité de CF:

CFU = <K, I> où:

K: la partie noyau.
I: la partie interface correspondante.

K = <M, IV> où:

M: l'ensemble de méthodes représentant le comportement du noyau.
IV: l'ensemble des variables d'instance.

I = < F, Externals, Internals, MC, SP > où:

F: l'ensemble de filtres
Externals: l'ensemble des objets externes
Internals: l'ensemble des objets internes
MC: l'ensemble de conditions et de méthodes
SP: la spécification de la superimposition

L'équivalent de CFU en AOP est < K, A> où:

K: est identique au K de CFU, il représente la partie fonctionnelle d'un système en AOP. Il contient tous les composants de K.
A: un ensemble d'aspects représentant la partie non fonctionnelle de système (i.e. les préoccupations).

Puisque I de CFU matérialise plusieurs parties d'une préoccupation, sa traduction en AOP sera vue comme étant plusieurs aspects « A » qui coopèrent. A chaque filtre correspond un aspect. Un aspect interface est utilisé pour introduire les objets Externals et Internals. Chaque méthode publique dans I possède une méthode abstraite dans l'aspect interface, de tel sorte que tous les appels a ces méthodes seront capturés par les aspects et seront délégués à l'un des objets internes ou externes. Les classes des objets internes et externes eux-mêmes restent inchangées.

Par conséquent, nous pouvons représenté l'ensemble des aspects « A » formellement comme suit:

A = <IA, FA> où:

IA: représente un aspect qui contient toutes les déclarations des objets internes, externes, ainsi que toutes les variables d'instance. En plus, il introduit des méthodes abstraites correspondant aux méthodes publiques contenues dans les parties Internals et Externals.

FA=<AA, CA> où:

AA: est un ensemble d'aspects abstraits où chacun contient une spécification sémantique d'un type particulier d'un filtre, donc, leur nombre est égal au nombre de type de filtres définis en CF.

CA: est un ensemble d'aspects concrets où chacun doit hériter d'un aspect de AA, de ce fait, chaque aspect de CA correspond a un filtre dans F.

Chaque aspect abstrait implémente un aspect sémantique d'un filtre particulier de

CF, de ce fait, chaque aspect doit contenir les deux méthodes: accepter() et rejeter() qui implémentent respectivement le comportement d'acceptation et de rejet d'un filtre.

```
abstract aspect Filter_filterType {
    public void accept() {
            // accepting action
    }
    public void reject() {
            // rejection action
    }
}
aspect idFilteri extends Filter_filterType {
    pointcut idFilteri_accept():if(ConditionPart) && if(MatchingPart);
    before() : idFilteri_accept() {
      accept();
    }
    before() : !idFilteri_accept(){
      reject();
    }
}
```

Selon notre modèle de mapping, un point de coupure appelé filter_accept est associé à chaque élément du filtre FE. Ce pointcut se divise en deux parties : une partie condition et une partie de correspondance. La partie condition sera utilisée pour évaluer la possibilité d'acceptation de message par le filtre. Elle est traduite en AOP en une primitive conditionnelle qui contient la même expression logique de la partie de condition en CF. La partie correspondance contient la partie matching de CF, qui est traduite en AOP par l'utilisation de la technique de réflexion. Le tableau suivant présente un récapitulatif de notre modèle de mapping entre CF et AOP.

Modèle de Filtres de Composition	Modèle Aspects
La partie noyau de modèle CF	Partie fonctionnelle
La partie interface de modèle CF	Partie non fonctionnelle
Filtre	Aspect
Nom de filtre	Nom d'aspect
Type de filtre	Aspect abstrait
Un élément de filtre	Point de coupure
La patie condition d'un filtre	Primitive conditionnelle
La partie matching	Point de coupure execution
Ordre des filters	Precedence entre les aspects
Sémantique d'un filtre	Consigne autour

La partie supérimposition	Weaving
Encapsulation des objects	Members privés

Table 4.1. Modèle de mapping CF-AOP

En [93] un mapping entre AOP et Hyperespace a été effectué, malheureusement, ce mapping étant complexe, il ne sera pas présenté ici. Comme conclusion à cette première expérience, le mapping de concepts entre les différentes approches ASOC s'avère complexe est parfois non révélateur des différences/ressemblances entre les concepts des approches ASOC.

2. Comparaison basée sur les patrons de conception

L'utilisation des patrons de conception est une méthode importante pour la réutilisation des solutions aux problèmes récurrents dans le développement des logiciels. Cependant, leur mise en oeuvre nécessite une certaine séparation afin de les maintenir aussi réutilisables que possible. Leur considération comme un outil de comparaison entre les différents paradigmes ASOC est une stratégie prometteuse. Dans ce contexte, nous comparons les mises en œuvres de nos trois patrons présentées au chapitre 3 en se basant sur quelques métriques utilisées en génie logiciel.

2.1. Etude Qualitative

Une étude qualitative doit être basée sur des métriques qualitatives. Dans ce contexte, on constate l'existence de diverses métriques. Notre choix s'est porté sur les métriques relatives aux notions importantes suivantes :

Meilleure localisation du code. Cela mesure le niveau de la séparation du code des patrons des autres parties de l'application.

Réutilisation. Signifie la possibilité de déclarer les patrons d'une manière abstraite, permettant aux utilisateurs de les réutiliser dans d'autres applications.

Composition. C'est la capacité d'utiliser plus d'un patron dans la même application sans enchevêtrement de leurs codes.

(Un)pluggability. La capacité d'activation/désactivation de l'effet d'un patron à un moment donné (i.e. au moment de la compilation ou au moment de l'exécution).

Compréhensibilité. Elle se base sur la nature de la description d'un patron moyennant les constructions du langage utilisé. Si la description est directe, le code devient facile à comprendre. Dans le cas où une description utilise des artifices spécifiques, la compréhension diminue.

2.2. Discussion

Le tableau suivant décrit les résultats obtenus de l'évaluation qualitative des mises en oeuvres des patrons de conception dans les approches ASOC. Les +/- dans le tableau désigne le degré d'évaluation de la métrique qualitative. Par exemple, «---» dans la colonne « Meilleure Localisation du Code » pour Java, indique que ce dernier ne permet pas une description compacte et facile à localiser dans le code global. Six niveaux d'évaluation ont été proposés Très bien (+++),Bien (++), Assez bien (+),faible (-),très faible (--), insignifiant (---).

	Meilleure Localisation du Code	Réutilisation	Composition	(Un)pluggability	Comprehensibilité
Java	---	---	-	---	+++
AspectJ	+++	+++	++	+++	+
ConcernJ	+++	+++	+++	+++	---
Hyper/J	--	+	++	+++	+++

Table 4.2. Evaluation qualitative des mises en oeuvre des patrons de conception par les approches ASOC

De cette table, nous pouvons conclure que la localisation des patrons est meilleure en AspectJ et ConcernJ, où le code des patrons est écrit sous une forme particulière (i.e. aspect en AspectJ et concern en ConcernJ) ce qui facilite leur localisation. Au contraire, la mise en oeuvre des patrons en Java et Hyper/J ne satisfait pas les contraintes de localisation puisque le code de patron est totalement écrit sous forme de classes Java ordinaires ce qui complique la distinction du code des patrons du reste du code de l'application.

La réutilisation des patrons est possible en AspectJ ainsi qu'en ConcernJ, car, dans ces deux modèles pratiques, on peut déclarer les patrons par des entités abstraites réutilisables (i.e. des aspects et des concerns abstraits). Ce n'est pas le cas pour les classes abstraites utilisées en Java, qui doivent être héritées par d'autres classes concrètes pour les rendre utilisables. Ceci rend les patrons dépendant d'autres parties de l'application et, par conséquent, réduit le taux de réutilisation.

Vu que le code du patron en AspectJ et ConcernJ est totalement indépendant d'autres parties d'application, son effet peut être activé ou désactivé dans l'application à tout moment. AspectJ offre un mécanisme simple pour spécifier les patrons qui doivent être intégré dans l'application. Il permet, pour cela, de spécifier les identificateurs des patrons aspects dans un fichier .lst au moment de la compilation de l'application. La désactivation sera possible aussi en retirant les identificateurs des patrons aspects du fichier .lst et en recompilant le code de l'application. De la même façon, ConcernJ offre la possibilité de spécifier les patrons concerns qui doivent être utilisés dans une application en les déclarant dans la partie de superimposition. La (UN)pluggability dans les objets ordinaires nécessite

certaines modifications dans la structure de l'application (i.e. modifier l'héritage entre les classes).

La mise en oeuvre des patrons en AspectJ et ConcernJ complique la compréhension du code des patrons parce que elle utilise des primitives particulières pour décrire un patron ce qui nécessite un effort de la part des développeurs pour comprendre et se familiariser avec ces modèle pratiques.

Composer des patrons dans une même application peut être mis en œuvre en AspectJ et ConcernJ sans aucun enchevêtrement de ces codes. AspectJ propose un modèle d'abstraction de haut niveau ce que nous permet de spécifier un ordre de dominance entre les différents aspects représentant les patrons. De même, ConcernJ utilise un ordre explicite d'exécution des filtres sur un objet CF en spécifiant cet ordre dans la partie superimposition.

2.3. Etude Quantitative

Notre étude quantitative se base sur diverses métriques quantitatives. Le choix de ces dernières dépend du but quand cherche à atteindre par la mesure effectuée. Dans cette thèse, le but est de trouver des mesures pour la qualité de conception des modèles orientés objets aussi bien que des modèles ASOC. Cela suggère un grand nombre de métrique, où chaque métrique donne une mesure spécifique. Le résultat d'une métrique doit être proportionnel à ce qu'elle mesure. Dans ce qui suit, nous considérons certains points pour comparer quantitativement les différentes mises en œuvre des patrons. Nos métriques peuvent être divisés en deux catégories: celles intervenant au moment de la compilation et celles intervenant au moment de l'exécution.

2.3.1. Les métriques intervenant au moment de la compilation

Nous avons considéré le nombre des étapes nécessaire pour composer le code des préoccupations avec les autres parties de l'application. Cet attribut dépend étroitement de la mise en oeuvre de chaque approche. Les figures suivantes donnent une aperçue sur chaque approche.

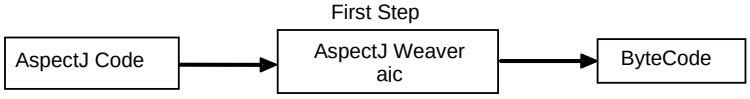


Fig. 4.1. Le processus de compilation en AspectJ

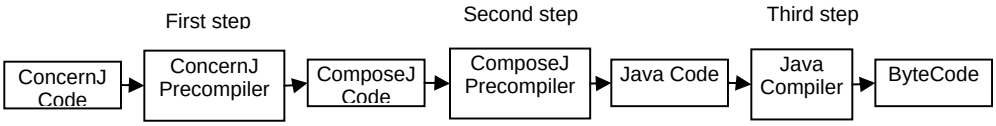


Fig. 4.2. Le processus de compilation en ConcernJ

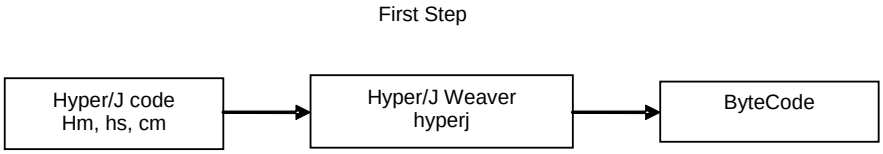


Fig. 4.3. Le processus de compilation en Hyper/J

D'après ces figures, nous remarquons que AspectJ et Hyper/J nécessitent seulement deux étapes pour accomplir le processus de compilation, alors que ConcernJ nécessite trois étapes dans leur processus de compilation. La première consiste à transformer le code de ConcernJ en ComposeJ ; la deuxième consiste à transformer le code de ComposeJ en Java finalement, le code Java sera converti en bytecode ; de se fait, AspectJ et Hyper/J satisfont cette métrique plus que ConcernJ.

Comparé aux autres métriques, que nous allons décrire par la suite, le nombre des étapes de compilation dépend de l'environnement de développement considéré.

Les métriques qui vont suivrent, sont utilisées fréquemment en génie logiciel et particulièrement pour le paradigme orienté objets. Dans le contexte de notre étude, nous avons été amené à ajouter quelques considérations relatives aux approches ASOC.

Nombre de Ligne de Code d'un Module (LOMC): *LOMC* compte le nombre de lignes de code parfaites d'un module (i.e. classe, aspect, concern, hypermodule etc.). La métrique LMOC nous aide à définir les efforts exigés des utilisateurs pour décrire un patron. Nous devons mentionner ici que le résultat de cette métrique doit être indépendant de tout style de programmation, pour cette raison nous avons considéré les points suivants:

1. chaque instruction sera considérée comme une seule ligne.
2. les en-têtes de classe, d'aspect, de concern, de hyperspace, de hypermodule, de méthode, de filterinterface, et de consigne, …, sont comptées comme une seule ligne.
3. les importations, les commentaires, les javadocs et les lignes vides ne sont pas comptés.

4. une ligne est comptée pour une accolade ouvrante mais pas pour une accolade fermente.
5. la définition d'une propriété ou d'une constante est comptée comme une seule ligne.

Nombre d'Opérations dans un Module (WOM): WOM compte le nombre d'opérations définies dans un module donné. WOM détermine la complexité interne d'un module en terme de nombre de fonctions définie dans ce dernier, de ce fait, il nous donne une idée sur combien de temps et d'effort sont nécessaires pour développer et maintenir un module. Une grande valeur de WOM influe sur la réutilisation du code quand les opérations seront hérités par les sous modules. Dans notre étude nous avons considérés les points suivants dans le calcul de WOM:

1. les constructeurs et les méthodes sont comptés comme des opérations.
2. les consignes, les introductions de méthodes ou de constructeurs sont comptées comme des opérations.
3. chaque spécification du filtre dans un filterinterface est comptée comme une opération.
4. chaque spécification du rapport dans la partie rapports dans un hypermodule est comptée comme une opération.

Profondeur d'arbre d'héritage (DIT): DIT c'est la profondeur d'un module donné dans l'arbre d'héritage. La profond d'un module dans la hiérarchie influe sur la compréhensibilité, la complexité et la réutilisation de ce module. Dès qu'un module est plus profond dans l'arbre d'hiérarchie, le nombre des opérations héritées par ce module devient plus grand et donc sa compréhension nécessite la compréhension des classes dont-il hérite. Les points suivants ont été pris en considération durant le calcul de cette métrique:

1. la classe Object n'est pas comptée.
2. toutes les modifications qui affectent la structure d'un système par la déclaration des introductions en AspectJ doivent être prises en considération.
3. le DIT d'une interface est toujours considéré nul.

Nombre de fils d'un Module (NOC): NOC est le nombre des sous modules d'un module donné. NOC indique l'influence potentielle d'un module dans la conception à travers ses dépendances. Une valeur maximale est souhaitable pour cette métrique. Les points suivants sont pris en considération durant le calcul de NOC:

1. toutes les modifications qui affectent la structure d'un système par la déclaration des introductions en AspectJ.
2. NOC est toujours nul pour les entités interface.

Degré d'enchevêtrement de Module (CDM): CDM est le nombre de modules affectés par le module en question. Cela donne une idée de l'impact total du module donné sur les autres modules. Une grande valeur de CDM est habituellement souhaitable.

Couplage par appels aux méthodes (CMC): CMC est le nombre de modules déclarant des opérations qui peuvent être appelées par un module donné. L'utilisation, par un module, d'un nombre important de méthodes, déclarées dans d'autres modules, indique l'importance du module en question et si sa séparation de l'application sera facile (dans un but de réutilisation). Une petite valeur de CMC est habituellement désirable.

Les points qui doivent être considérées dans le calcul de CMC sont :

1. les appels aux constructeurs doivent être comptés.
2. les appels aux méthodes introduitent par un module sont comptés comme un appel au module qui introduit ces méthodes.

Couplage par accès aux propriétés (CFA): CFA est le nombre de modules déclarant des propriétés qui sont accessibles au module donné. Comme CMC, CFA mesure la dépendance entre un module donné et les autres, mais en terme d'accès aux propriétés. Une petite valeur de CFA est souhaitable. Les points suivants doivent être pris en considération:

1. les accès aux propriétés introduites par un module sont comptés comme un accès au module qui fait l'introduction de ces propriétés.

Couplage entre Modules (CBM): CBM est le nombre de modules déclarant des méthodes ou des propriétés qui peuvent être appelés ou accédées par un module donné. C'est une combinaison de CMC et CFA. Une petite valeur de CBM est souhaitable.

Nombre de Types de Modules (NOT): NOT est le nombre de types de modules utilisés dans la conception d'un package. NOT indique la complexité du package. Toute classe (normale ou interne), interface, aspect, concern, hypermodule, hyperespace sont comptés.

Abstraction (A): A est le rapport du nombre de modules abstraits sur le nombre total de modules dans un package. La valeur de A appartient à l'intervalle [0, 1]. La valeur zéro indique que la package est complètement concret et une valeur 1 indique que le package est totalement abstrait ce qui est parfait.

Couplage Afférent (CA): CA est le nombre de modules à l'extérieur d'un package qui dépendent des modules de ce package. C'est un indicateur de la responsabilité du package. Une haute valeur de cette métrique est souhaitable.

Couplage Efférent (CE): CE est le nombre de modules à l'intérieur d'un package qui dépendent des modules à l'extérieur du package. C'est un indicateur de l'indépendance des packages. Une valeur réduite de cette métrique est habituellement souhaitable.

Instabilité (I): I est le rapport entre CE et le total de couplage (CE + CA). Tel que

I = CE / (CE + CA). Cette métrique est un indicateur de la stabilité d'un package. L'instabilité est comprise entre 0 et 1. Une valeur 0 indique que le package en question est complètement stable. De ce fait, une petite valeur de I est souhaitable.

Distance normalisée de séquence (Dn): Dn est la distance perpendiculaire normalisée de la ligne idéalisée (A + I = 1) d'un package. Tel que, Dn = 1-(A+I). Cette métrique est un indicateur de balance d'un package entre l'abstraction et la stabilité. La valeur de Dn est comprise entre 0 est 1. Une valeur 0 indique une conception parfaite du package. Un package qui se met sur la séquence principale (la ligne A + I = 1) n'est pas trop abstrait pour sa stabilité, ni trop instable pour son abstraction. De ce faite, les valeurs désirées sont 0 ou 1.

2.3.2. Les métriques intervenant aux moment de l'exécution

Nous avons limité notre étude aux trois métriques suivantes:

1. le taux de réponse d'un module (RFM)
2. le nombre d'objets dont on a besoin pour l'exécution d'un système (NBO)
3. le nombre d'appels externes (NFC)

Taux de Réponse d'un Module (RFM): RFM mesure la communication potentielle entre un module donné et les autres modules du système. L'exécution d'un nombre important d'opérations (i.e. méthodes, consignes, etc.) pour répondre à un message reçu rend les tâches des modules plus complexes. Les point qui doivent être pris en considération durant le calcul de RFM sont :

1. RFM = nombre de méthodes dans le module + le nombre de méthodes à invoquer + le nombre d'opérations implicitement exécutées
2. Une méthode introduite par un module sera considérée comme une opération de ce module

Nombre d'objets nécessaire pour l'exécution d'un système (NBO): NBO indique l'espace mémoire libre nécessaire pour exécuter un système. Les points suivants doivent être considérés dans le calcul de NBO:

1. un objet pour chaque classe concrète doit être compté
2. un objet pour chaque aspect concret doit être compté
3. un objet spécial pour un hyperspace doit être considéré
4. un objet doit être considéré pour chaque filtre en CF

Nombre d'appels externes (NFO): NFO indique le taux de communication entre modules. Durant le calcul de cette métrique, nous devons considérer chaque appel d'une opération qui réside à l'extérieur du package du module en question comme un appel externe.

2.4. Les outils d'expérimentation

Pour décrire les patrons suivant les différentes approches ASOC nous avons choisi

les environnements suivants avec leurs versions les plus récentes :

1. Java pour le description des patrons en orienté objets. Le JDK1.5 à été utilisé.
2. AspectJ 1.5 pour la description en AOP. [108]
3. Eclips 3.0 comme une plateforme d'exécution des programmes Java et AspectJ. [108]
4. Hyper/J™2.0 (produit d'IBM) pour la description des patrons en hyperspace [40]
5. ComposeJ 1.3.4 (produit du groupe Trese de l'université Twente, Pays bas) pour la description des patrons CF [51]

Pour les outils d'évaluation des métriques nous avons examiné plusieurs outils, parmi les plus important, on cite :

1. JDepend 2.9.1: Il fournit une interface graphique pour évaluer certains métriques pour un package Java, cependant, JDepend n'examine que les métrique de Martin (i.e. CE, CA, A, I et Dn). [111]

2. JHawk 3.5: Il fournit une interface utilisateur graphique pour examiner des classes en Java et mesurer un nombre intéressant de métriques, malheureusement, il nous autorise à n'examiner qu' une seule classe à la fois, ce qui rend impossible d'examiner la dépendance entre classes [110]

3. Compuware OptimalAdvisor: Il examine seulement une partie de nos métriques et ne supporte que des classes Java [112]

Durant l'examen des outils de mesure des métriques, nous avons trouvé que Aopmetrics est un outil idéal pour l'évaluation des classes Java et des aspects en AspectJ; cet outil a été développé sous un projet d'une thèse de master en informatique dans l'université Wroclaw des technologies en Pologne. La version courante disponible est 3.0. [104]

Ant Task est la seule méthode disponible pour lancer Aopmetrics. Ant est une nouvelle façon d'exécuter des programmes indépendamment de tout système d'exploitation en utilisant la spécification en XML; avec cette technologie en doit constuire un fichier qui décrit toutes les tâches qui doivent être exécutées par le système en spécifiant les chemins des outils nécessaires [109]. Dans notre étude nous avons utilisé la version Ant apache 1.6.5. Le processus d'exécution d'aopmetrics sous la technologie Ant est présenté par la figure 4.4.

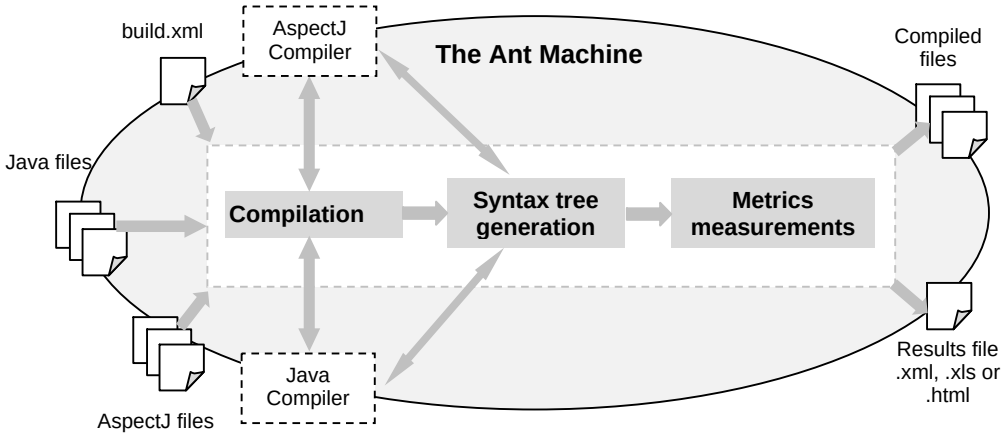


Fig. 4.4. Le processus d'exécution de Aopmetrics sous la technologie Ant Task

L'Aopmetrics Ant Task consiste à utiliser les compilateurs Java et AspectJ pour compiler le code sources des programmes en entrée et obtenir des bytecodes. Le code objet est ensuite sauvegardé dans un emplacement spécifié au départ, puis l'arbre syntaxique des deux modèles est généré et les métriques mesurées. Enfin, l'Aopmetrics exporte les résultats dans un fichier xml, xls ou html qui est spécifié à la machine Ant au départ.

Le fichier de configuration build.xml est un fichier qui contient des informations sur les chemins d'accès aux compilateurs Java et AspectJ, il doit contenir aussi le chemin des fichiers entrants et le chemin pour sauvegarder les fichiers objets ainsi que la nature et le chemin du fichier résultat des métriques mesurées; le code suivant décrit la structure principale du fichier build.xml.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="aop-metrics" default="all">
  <property name="dir.build" value="build" />
  <property name="dir.testworkdir" value="${dir.build}/testworkdir" />
  <property name="dir.libraries" value="lib" />
  <property name="version" value="0.2" />

<path id="classpath.common">
<fileset dir="${dir.libraries}">
<include name="*.jar" />
</fileset>
</path>

<target name="all" depends="run" description="Compiles, run." />
<target name="run" description="Computes metrics on the project.">

<taskdef name="aopmetrics" classname="org.tigris.aopmetrics.AopMetricsTask">
```

```
  <classpath refid="classpath.common" />
  <classpath location="${dir.build}/${ant.project.name}.jar" />
</taskdef>

<aopmetrics workdir="${dir.testworkdir}" sourcelevel="1.5" export="resultfilekind"
   resultsfile="resultfileName">
  <fileset dir="concernedpackage" includes="*.java" />
   <fileset dir="concernedpackage" includes="*.aj" />
  <classpath refid="classpath.common" />
</aopmetrics>

</target>
</project>
```

Bien qu'il y ait beaucoup d'outils utilisés pour l'évaluation des programmes orientés objets, un seul outil existe pour les approches ASOC, en occurrence, il s'agit de Aopmetrics. A notre connaissance, il n'y a aucun outil pour l'évaluation des programmes en hyperespace ou en CF. Par conséquent, l'évaluation de nos métriques a été faite par Aopmetrics pour Java et AspectJ, et manuellement pour CF et hyper/J.

2.4. Résultats et Discussion

Notre étude quantitative s'est basée sur les 13 métriques suivantes: LOCC, WOM, DIT, NOC, CDA, CMC, CFA, CBM, NOT, A, CE, CA, I et Dn, ces métriques ont été estimées par l'utilisation de Aopmetrics pour les implémentations Java et AspectJ, et calculées manuellement pour ComposeJ et Hyper/J.

Dans notre comparaison, seule 8 métriques parmi les 13 (i.e. LOCC, WOM, DIT, NOC, CDA, CBM NOT et Dn) qui ont été mises en considération, puisque le reste sont des métriques intermédiaires qui ont été utilisés pour calculer une ou plusieurs des précédentes. Par exemple, les métriques A et I sont utilisées pour calculer la métrique Dn selon la formule Dn = 1 - (A + I). De même les métriques CE et CA permettent le calcul de I par la formule I = Ce/(Ce+Ca). Les tables et les figures suivantes présentent clairement nos résultats.

La première série de tableaux donne les résultats des métriques intervenant au moment de la compilation pour les trois patrons utilisés dans les trois approches ASOC.

La deuxième série donne les valeurs des métriques intervenant au moment de l'exécution.

Evaluation Quantitative par Patron Adapter

	LOCC	WOM	DIT	NOC	CDA	CMC	CFA	CBM	NOT	A	CE	CA	I	Dn
OOP	14	3	0	0	0	1	0	1	3	0,333333	0	0	0	0,666667
AOP	10	2	1	1	1	0	0	0	3	0,5	0	0	0	0,5
CF	18	4	0	0	1	1	0	1	2	0	1	0	1	0
Hyperspaces	18	5	0	0	2	0	0	0	4	0	3	0	1	0

Table 4.3. Evaluation des approches ASOC par AdapterPattern (à la compilation)

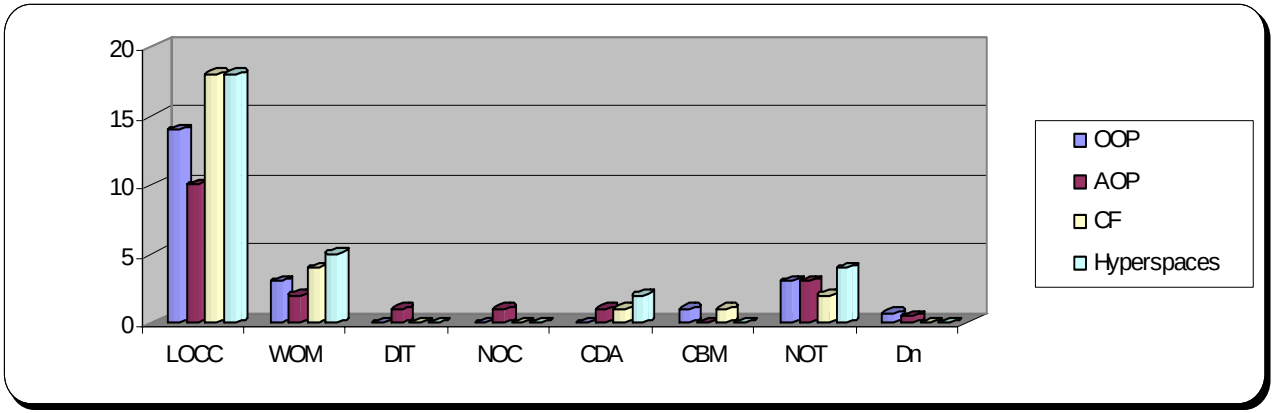


Fig. 4.5. Evaluation des approches ASOC par AdapterPattern (à la compilation)

Evaluation Quantitative par le patron Singleton

	LOCC	WOM	DIT	NOC	CDA	CMC	CFA	CBM	NOT	A	Ce	Ca	I	Dn
OOP	25	4	3	0	0	1	0	1	2	0	0	0	0	1
AOP	31	1	1	1	1	2	0	2	4	0,6	3	0	1	0,6
CF	32	6	0	0	1	0	0	0	1	0	1	0	1	0
Hyperspaces	26	8	0	0	2	0	1	1	4	0	3	0	1	0

Table 4.4. Evaluation des approches ASOC par SingletonPattern (à la compilation)

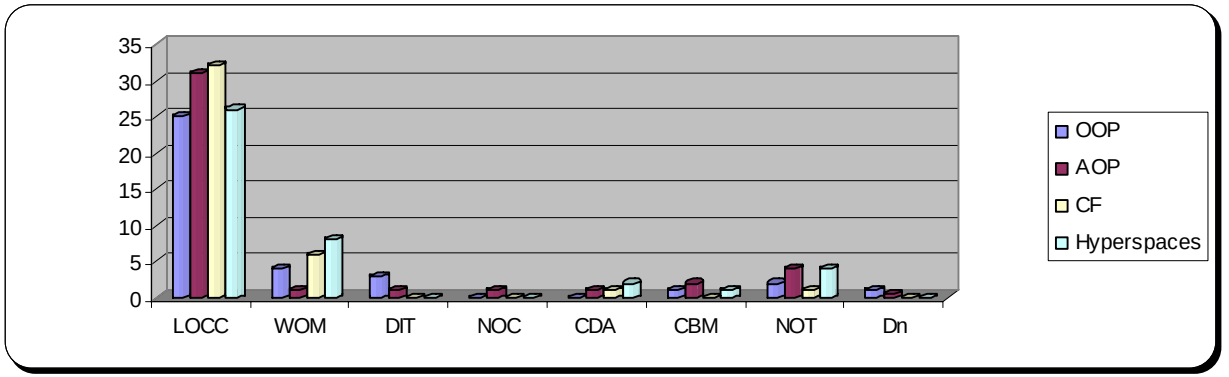


Fig. 4.6. Evaluation des approches ASOC par SingletonPattern (à la compilation)

Evaluation Quantitative par le patron ChainOfResponsibility

	LOCC	WOM	DIT	NOC	CDA	CMC	CFA	CBM	NOT	A	Ce	Ca	I	Dn
OOP	21	3	1	1	0	1	0	1	2	0,5	0	0	0	0,5
AOP	55	8	1	1	3	1	1	2	3	0,5	2	0	1	0,5
CF	30	7	0	0	2	0	0	0	1	0	1	0	1	0
Hyperspaces	28	6	0	0	2	1	0	1	5	0	3	0	1	0

Table 4.5. Evaluation des approches ASOC par ChainOfResponsibilityPattern (à la compilation)

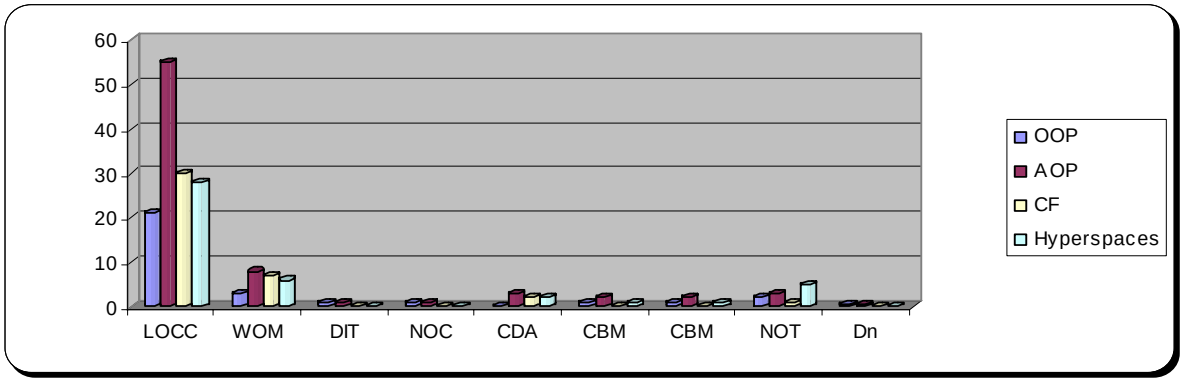


Fig. 4.7. Evaluation des approches ASOC par ChainOfResponsibilityPattern (à la compilation)

Pour avoir une décision concernant une approche ASOC dans son ensemble, il faut combiner les résultats des différentes métriques. Cependant, cette tâche est loin d'être évidente, elle est même dépendante des qualités qu'on souhaite privilégier.

Nous proposons un critère de décision propre à notre travail « P ». Le facteur proposé ici « P », calcule le taux de métriques satisfaites par chaque approche ASOC. De ce fait, « p » doit être calculé pour chaque approche « A » selon la formule suivante. W_i désigne un facteur d'importance associé à chaque métrique m_i (i.e. une pondération) :

$$P(A) = \frac{\sum_{i=1}^{metricsNumber} w_i \times V(m_{i,A})}{\sum_{j=1}^{metricsNumer} w_j} \times 100$$

La fonction V est une fonction logique quantifiable pour chaque métrique m_i selon la formule suivante :

$$V(m_{i,A}) = \begin{cases} 1 & if \quad m_{i,A} \equiv R(m_i) \\ 0 & otherwise \end{cases}$$

La fonction R dans V, est une fonction logique qui sera la fonction de maximisation ou de minimisation selon la nature de métrique m_i pour l'ensemble des approches intégrées dans l'évaluation. La description de R est donnée par la formule suivante :

$$R = \begin{cases} Max & for \quad CDM \; and \quad NOC \\ Min & Otherwise \end{cases}$$

Pour notre étude et pour faciliter le processus d'évaluation, nous avons assignée un facteur identique égal à 1 pour toutes les métriques choisies, puis nous avons calculé la valeur P pour chaque approche. Les tableaux et les figures suivantes montrent clairement nos résultats.

	V(LOCC)	V(WOM)	V(DIT)	V(NOC)	V(CDA)	V(CBM)	V(NOT)	V(Dn)	P (A)
OOP	0	0	1	0	0	0	0	0	12,50%
AOP	1	1	0	1	0	1	0	0	**50%**
CF	0	0	1	0	0	0	1	1	37,50%
Hyperspaces	0	0	1	0	1	1	0	1	**50%**

Table 4.6. Valeur de la fonction de comparaison P(A) pour AdapterPattern (à la compilation)

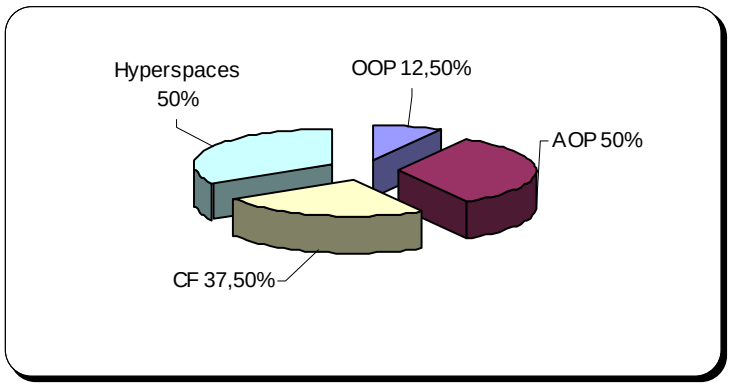


Fig 4.8. Valeur de la fonction de comparaison P(A) pour AdapterPattern (à la compilation)

	V(LOCC)	V(WOM)	V(DIT)	V(NOC)	V(CDA)	V(CBM)	V(NOT)	V(Dn)	P (A)
OOP	1	0	0	0	0	0	0	1	25%
AOP	0	1	0	1	0	0	0	0	25%
CF	0	0	1	0	0	1	1	1	**50%**
Hyperspaces	0	0	1	0	1	0	0	1	37,50%

Table 4.7. Valeur de la fonction de comparaison P(A) pour SingletonPattern (à la compilation)

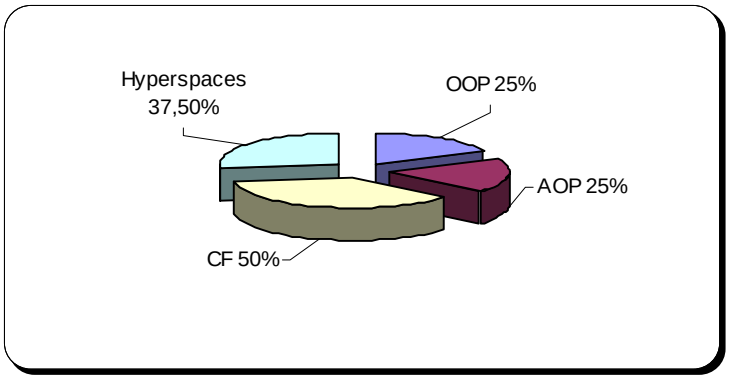


Fig 4.9. Valeur de la fonction de comparaison P(A) pour SingletonPattern (à la compilation)

	V(LOCC)	V(WOM)	V(DIT)	V(NOC)	V(CDA)	V(CBM)	V(NOT)	V(Dn)	P (A)
OOP	1	1	0	1	0	0	0	0	37,50%
AOP	0	0	0	1	1	0	0	0	25%
CF	0	0	1	0	0	1	1	1	**50%**
Hyperspaces	0	0	1	0	0	0	0	1	25%

Table. 4.8. Valeur de la fonction de comparaison P(A) pour ChainOfResponsibilityPattern (à la compilation)

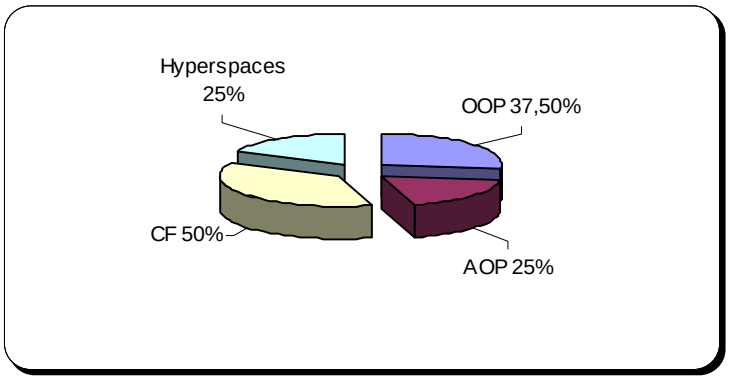


Fig. 4.10. Valeur de la fonction de comparaison P(A) pour ChainOfResponsibilityPattern (à la compilation)

D'après les tableaux et les figures précédentes nous pouvons remarquer que l'approche hyperespace montre une efficacité pour la description des patrons AdapterPattern et SingletonPattern et que l'approche CF est meilleure pour la description du patron ChainOfResponsibilityPattern.

Le même processus a été respecté pour l'évaluation des métriques quantitative intervenant au moment de l'exécution pour chacune des approches ASOC . Les tableaux et les figures suivants présentent clairement nos résultats :

	RFM	NBO	NFC
OOP	3	2	1
AOP	1	1	1
CF	4	1	2
Hyperspaces	1	3	0

Table. 4.9. Evaluation des approches ASOC par AdapterPattern (à l'exécution)

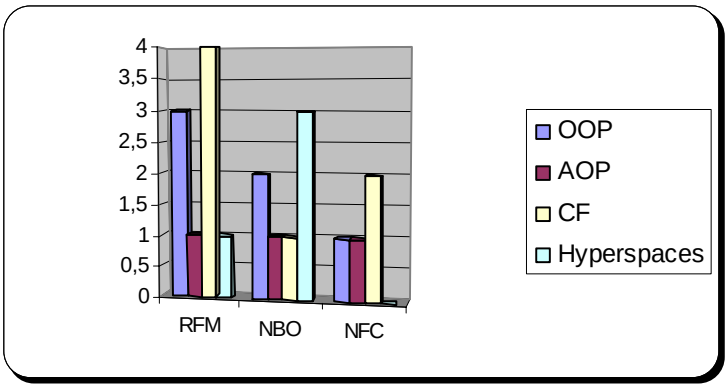


Fig. 4.11. Evaluation des approches ASOC par AdapterPattern (à l'exécution)

	RFM	NBO	NFC
OOP	1	2	1
AOP	0	3	1
CF	5	3	1
Hyperspaces	0	2	0

Table. 4.10. Evaluation des approches ASOC par SingletonPattern (à l'exécution)

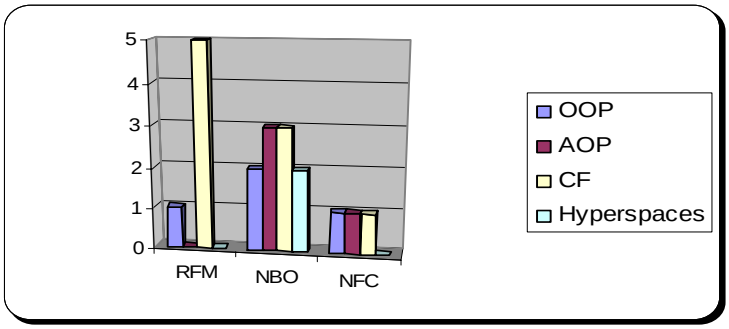


Fig. 4.12. Evaluation des approches ASOC par SingletonPattern (à l'exécution)

	RFM	NBO	NFC
OOP	4	1	1
AOP	11	2	1
CF	10	4	2
Hyperspaces	1	1	0

Table. 4.11. Evaluation des approches ASOC par ChainOfResponsibilityPattern (à l'exécution)

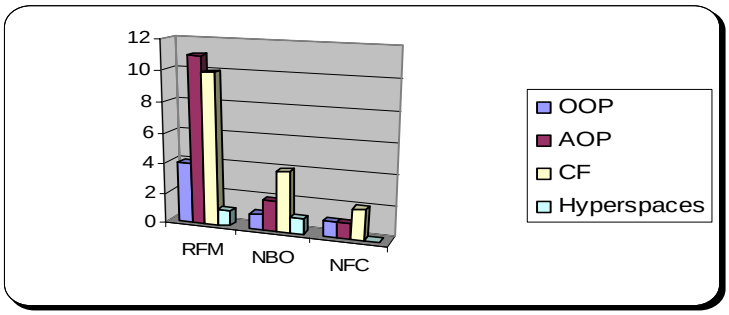


Fig. 4.13. Evaluation des approches ASOC par ChainOfResponsibilityPattern (à l'exécution)

Le même critère P avec les mêmes considérations précédentes est calculé pour chaque approche, les résultats obtenus sont présentés dans les tableaux et les figures suivants:

	V(RFM)	V(NBO)	V(NFC)	P (A)
OOP	0	0	0	0%
AOP	1	1	0	**66,67%**
CF	0	1	0	33,33%
Hyperspaces	1	0	1	**66,67%**

Table. 4.12. Valeur de la function de comparaison P(A) pour AdapterPattern(à l'exécution)

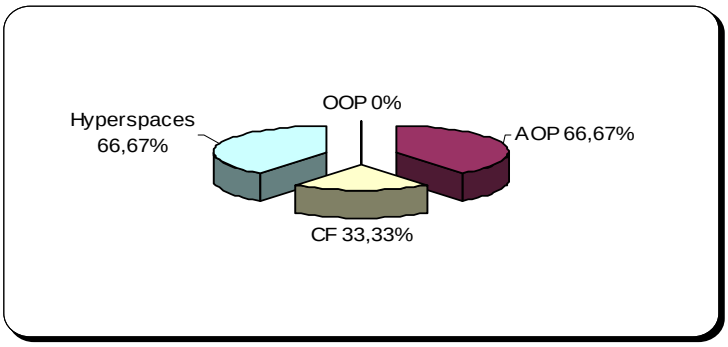


Fig. 4.14. Valeur de la function de comparaison P(A) pour AdapterPattern(à l'exécution)

	V(RFM)	V(NBO)	V(NFC)	P (A)
OOP	0	1	0	33,33%
AOP	1	0	0	33,33%
CF	0	0	0	0%
Hyperspaces	1	1	1	**100%**

Table. 4.13. Valeur de la function de comparaison P(A) pour SingletonPattern(à l'exécution)

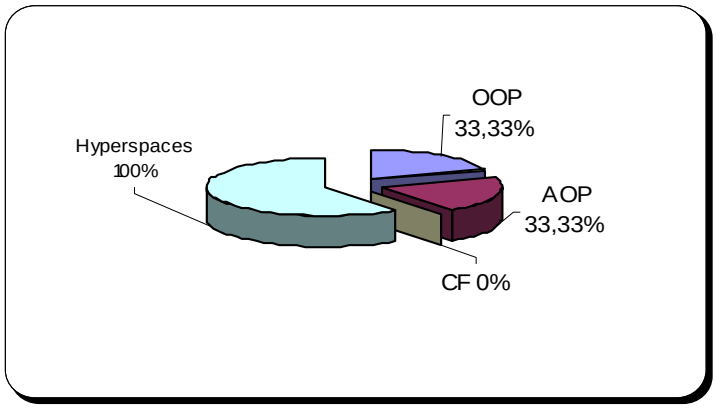


Fig. 4.15. Valeur de la function de comparaison P(A) pour SingletonPattern(à l'exécution)

	V(RFM)	V(NBO)	V(NFC)	P (A)
OOP	0	1	0	33,33%
AOP	0	0	0	0%
CF	0	0	0	0%
Hyperspaces	1	1	1	**100%**

Table. 4.14. Valeur de la function de comparaison P(A) pour ChainOfResponsibilityPattern (à l'exécution)

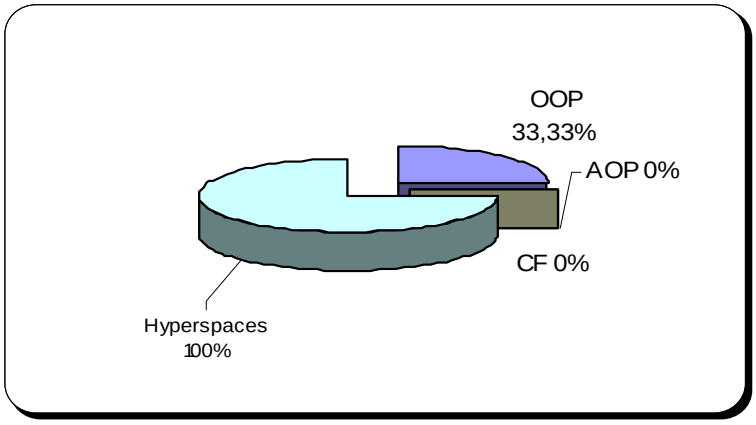


Fig. 4.16. Valeur de la function de comparaison P(A) pour ChainOfResponsibilityPattern (à l'exécution)

D'après les tableaux et les figures précédents nous remarquons que l'approche hyperespace montre, pour cette fois aussi, son efficacité pour les trois patrons : Adaptateur, Singleton et Chaîne de Responsabilité, et que l'approche AOP est efficace pour la description du patron Adaptateur.

3. Conclusion

Dans ce chapitre nous avons utilisé quelques métriques qualitatives et quantitatives dans le but d'évaluer les trois approches ASOC par le trois patterns retenus. Les métriques utilisées sont inspirées d'autres travaux pour l'approche orientée objets [6, 9, 15, 29, 30, 33, 41, 47, 50, 56, 64, 70, 82, 87, 99-107], et pour tenir compte de certaines spécificités des approches ASOC, nous les avons adapté. Nos résultats sont des résultats révélateurs qui montrent que l'utilisation des patrons de conception comme benchmarks pour l'évaluation des paradigmes ASOC est une approche prometteuse.

Chapitre 5

DISCUSSIONS & CONCLUSION

1. Application des patrons pour l'IA

Les patrons peuvent être utilisés dans beaucoup d'applications dans le contexte de l'intelligence artificielle tel que: la représentation de connaissances, les modes d'inférence ainsi que dans la modélisation des systèmes multi-agents.

Nous décrivons ci-après nos conclusion sur l'utilisation les patrons dans le contexte de l'IA.

1.1. Les Patrons et la représentation de connaissances

Tout d'abord, vu que la capture des expériences passées et leur documentation est l'objectif principal des patrons de conception, il est normal que les patrons puissent être utilisés dans la représentation des connaissances.

On peut découvrir une grande similarité entre un cas d'un système à base de cas et un patron. De tel sorte qu'un cas représente un problème avec une description de sa solution, le patron l'est aussi. De même la similarité entre les cas dans un CBR peut être assurée par le langage de patrons où les patrons sont classés et reliés entre eux selon le domaine d'application de chacun.

1.2. Les Patrons et les modes de raisonnement

Les patrons ont été déjà utilisés dans le but d'implémenter un moteur d'inférence par D. Pan [27] où le travail présenté consiste en l'implémentation d'un moteur d'inférence basé sur la logique de description. Les patrons ont été utilisés pour assurer quelques exigences dans le système; parmi ces exigences on peut citer:

1. Maintien de la base de la connaissance dans un état cohérent.
2. Support de l'ajout de nouveaux composants dans la base de connaissances.
3. Support de la suppression des composants de la base de connaissances.
4. Support du contrôle de la cohésion de concepts.
5. Support de plusieurs type de données.
6. Support de l'héritage entre les concepts.
7. Support de la classification de concepts.
8. Support du contrôle de la cohésion de règles.

Pan dans son travail démontre qu'il est possible d'utiliser les patrons de conceptions dans l'implémentation du moteur d'inférence [27].

Relativement à cette approche, nous pensons que l'implémentation du système et

la garantie des différentes exigences seront plus faciles à atteindre lorsqu'on utilise des patrons de conception avec les approches ASOC. Car cela rend le code source du moteur d'inférence plus réutilisable. Ce point a été conservé comme un point perspective dans notre travail.

1.3. Les Patrons & les systèmes multi-agents

Aujourd'hui, il y a peu de travaux qui tentent d'utiliser les patrons dans les applications des systèmes multi-agents, un des travaux intéressants est ce celui présenté par S. Sauvage dans [95, 96]. Dans ce travail, Sauvage démontre la capacité d'utiliser les patrons dans la conception des systèmes multi-agents. Pour plus de renseignements ce sujet, consulter [95, 96].

Notre contribution dans ce contexte consiste à utiliser les patrons comme un outil d'aide à la décision pour les architectures des systèmes multi-agents; où on documente les expériences dans le domaine de conception à base d'agents sous forme des patrons pour les rendre réutilisables. Cette même idée peut être utilisée pour les architectures neuronales.

2. Conclusion

Les patrons de conception sont importants dans la réutilisation des solutions à des problèmes récurrents. Cependant, leurs mises en oeuvre exige des techniques appropriées pour les rendent réutilisable.

Les approches ASOC présentent une excellente méthode de séparation de préoccupations dans le domaine du développement des logiciels, cependant, chaque approche à des limites de conception et d'abstraction. Dans le contexte de cette thèse, nous avons présenté une description de quelques patrons de conception sous AOP, CF et Hyperespace. Notre contribution principale est d'évaluer qualitativement et quantitativement les différentes implémentations obtenues en utilisant quelques métriques du génie logiciel.

Suite à nos expériences dans ce travail, nous pouvons formuler deux conclusions, d'un coté, l'implémentation des patrons sous les approches ASOC augmente le taux de réutilisabilité des logiciels, de l'autre, les patrons de conception peuvent être utilisés comme un standard d'évaluation des approches ASOC et plus généralement des paradigmes du génie logiciel.

3. Perspectives

Comme perspectives principales, nous proposons d'intégrer tous les patrons de GOF dans l'évaluation et l'implémentation de quelques applications de l'IA moyennant des patrons décrits selon les approches ASOC.

Pour ce qui est du travail de d'évaluation lui-même, nous pensons qu'une évaluation du temps d'exécution des programmes contenant divers patrons décrits selon les approches ASOC peut être un bon complément de ce travail. Il faut

cependant noter que ce temps est fortement dépendant de l'environnement d'exécution et de la manière dont le tissage est fait. Actuellement, les approches ASOC continuent d'évoluer et il en est de même pour les outils qui les supportent.

Bibliographie & Références

[1] A. Amiot, P. Cointe and Y.G. Guéhéneuc, Un méta-modèle pour coupler application et détection des design patterns, This paper has been accepted at LMO, French 2002.

[2] A. Borgida, R.J. Brachman, D.L. McGuinness, L.A. and Resnick, CLASSIC: A Structural Data Model for Objects, Proceedings of ACM SIGMOD International Conference on Management of Data, June 1989.

[3] A. Burggraaf, Solving Modelling Problems of CORBA using Composition Filters, M.Sc. thesis, Department of Computer Science, University of Twente, The Netherlands, 1997.

[4] A. Garcia, C.S. Anna and E. Figueiredo, Modularizing Design Patterns with Aspects: A Quantitative Study, Software Engineering Laboratory - Computer Science Department Pontifical Catholic University of Rio de Janeiro – PUC-Ri, 2005.

[5] A. Konar, Artificial Intelligence and Soft Computing : Behavioral and Cognitive Modeling of the Human Brain, Department of Electronics and Tele-communication Engineering, Jadavpur University, Calcutta, India, CRC Press, Boca Raion London, New York, Washington, D.C. 2000.

[6] A. Liso, Software Maintainability Metrics Model: An Improvement in the Coleman-Oman Model, http://www.stsc.hill.af.mil/crosstalk/2001/08/liso.html, 2004-05-17.

[7] A. Mendhekar, et al., RG: A Case Study for Aspect-Oriented Programming, Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February, 1997.

[8] A. Mitchell, J.F. Power, Toward a definition of run-time object-oriented metrics, 7th Ecoop Workshop On Quantitative Approaches In Object-Oriented Software Engineering 2003.

[9] A.H. Hannousse, D. Meslati, H.F. Merouani, ASOC: A Qualitative and A Quantitative Study Based Design Patterns, Séminaire National en Informatique de Biskra, SNIB'06, Biskra, Algérie, Mai 2006.

[10] A.H. Hannousse, D. Meslati, H.F. Merouani, Aspect Oriented Prgramming and Composition Filters: A Conceptuel Comparative Study, Aspect-Oriented Software Development, DSOA'05, Granada, Spain, September 2005.

[11] A.H. Hannousse, D. Meslati, H.F. Merouani, Aspect Oriented Prgramming and Composition Filters: A Design Patterns Based Comparative Study, Conférence International en Informatique Appliqué, CiiA'05, Bordj Borreridj, Algérie, Octobre 2005.

[12] A.H. Hannousse, D. Meslati, H.F. Merouani, State Of The Art Of Advanced Separation Of Concerns: A Design Patterns Based Comparative Study, Conference International en Productique, CIP'05,

Tlemcen, Algérie, December 2005.

[13] A.H. Hannousse, M.L. Kenouni, CoAspectJ: Vers un modèle Orienté Aspects Générique, implémentation de Composition de Filtres sous AspectJ, Mémoire de fin d'études, Département d'informatique, Université Badji Mokhtar, Annaba, Juin 2003.

[14] Appleton, Patterns and Software: Essential Concepts and Terminology, World Wide Web http://www.enteract.com/~bradapp/, last modified 14 February 2000.

[15] B. Henderson-Sellers, L. L. Constantine, I. M. Graham, Coupling and Cohesion (Towards a Valid Metrics Suite for Object Oriented Analysis and Design), Object Oriented Systems, Vol. 3, pp.

[16] B.R. Gaines, Class Library Implementation of an Open Architecture Knowledge Support System, International Journal of Human-Computer Studies, 41(1-2), 1995.

[17] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, and S. Angel, The Oregon Experiment, Oxford University Press, New York, 1975.

[18] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, A Pattern Language, Oxford University Press, New York, 1977.

[19] C. Alexander, The Timeless Way of Building, Oxford University Press, New York, 1979.

[20] C. Åsman, M. Engene, Software Patterns, Institutionen för Datavetenskap Chalmers Tekniska Högskola och Göteborgs Universitet, 1999.

[21] C. Stuurman, Techniques for defining Composition Filters Using Message Manipulators, Mcs. Thesis, University of Twente, Netherlands, August 1995.

[22] C. Vinkes, Superimposition in the Composition Filters, M. thesis, Software Engineering group, Electrical Engineering, Mathematics and Computer Science, University of Twente, October 2004.

[23] D. Gruijs, A Framework of Concepts for Representing Object-Oriented Design, M. Sc. thesis, University of Utrecht, Department of Computer Science, INF/SCR-97-28, August 3, 1998.

[24] D. Janaki Ram, P.Jithendra Kumar Reddy and M.S.Rajasree, An Approach to Estimate Design Attributes of Interacting Patterns, QAOOSE 2003.

[25] D. Meslati, A.H. Hannousse, M.A. Kenouni, La composition de filtres et AspectJ, une comparaison conceptuelle, accepted in Eighth Maghrebian Conference on Software Engineering and Aritificial Intelligence, MCSEAI, Sousse, Tunisie, Mai, 2004

[26] D. Meslati, M.T. Kimour, A.H. Hannousse, On Composition Filters And AspectJ: A PSM To PSM Transformation, The Seventh International Symposium On Programming and Systems, ISPS'2005, Algiers.

[27] D. pan, The Application of Design Patterns in Knowledge Inference Engine, Master thesis, Department of Computer Science, university of Calgary, July 1998.

[28] D.M. Wagelaar, A concepts based approach to Aspect oriented software design, University of Twente, M. Sc.Thesis, August 2002.

[29] D.N. Card, Khaled El Emam, Betsy Scalzo, Measurement of Object Oriented Software Development Projects, Software Productivity Consortium, Herndon, Virgina, 2001.

[30] D.P. Darcy, Chris F. Kemerer, Managerial use of Metrics For Object Oriented Software: An Exploratory Analysis, IEEE Transactions on Software Engineering, Vol. 24, Issue 8, pp. 629-639, 1998.

[31] E. Hilsdale, G. Kiczales, W.G. Griswold, J. Hugunin and Kersten, Aspect-Oriented Programming with AspectJ™, 2001.

[32] E. Hilsdale, J. Hugunin, Advice Weaving in AspectJ. Submitted to the 3rd International Conference on Aspect-Oriented Software Development (AOSD). April 2004. http://www.cs.indiana.edu/~ehilsdal/cv.2004-aosd-adviceweaving.pdf

[33] F.T. Sheldon, K. Jerath, H. Chung, Metrics for Maintainability of Class Inheritance Hierarchies, Journal of Software Maintenance and Evolution: Research and Practice, Vol. 14, Issue 3, pp. 147-160, 2002.

[34] G. Kiczales et al, An Overview of AspectJ, In Proc. Of ECOOP, Springer-Verlag(2001).

[35] G. Kiczales et al, Aspect Oriented Programming, In proc. Of European Conference on Object Oriented Programming (ECOOP), Lecture Notes in Computer Science Vol. 1241, pp. 220-242, 1997.

[36] G. Masuda, N. Sakamoto and K. Ushijima, Evaluation and Analysis of Applying Design Patterns, 2000.

[37] G. Sunyè, A. Guennec, and J. M. Jèzèquel, Design Patterns Application in UML, ECOOP 2000, LNCS 1850, pp. 44-62, 2000.

[38] G.K. Gill, Chris F. Kemerer, Cyclomatic Complexity Density and Software Maintenance Productivity, IEEE Transactions on Software Engineering, Vol. 17, Issue 12, pp. 1284-1288, 1991.

[39] H. Hamza, M. Faid, Towards A Pattern Language for Developing Stable Software Patterns, PLoP conference, 2003.

[40] H. Ossher, P. Tarr, Multi-Dimensional Separation of Concerns using Hyperspaces, IBM research Report 21452, April 1999.

[41] I. Alkadi, Application of a Revised DIT Metric to Redesign an OO Design, Journal of Object Technology, Vol. 2, Issue 3, pp. 127-134, 2003.

[42] J. Baltus, La Programmation Orientée Aspect et AspectJ: Présentation et application dans un Système Distribué, Facultés Uniuversitaires Notre-dame de la paix, Namur, Belgique, 2002.

[43] J. Bloch, Effective Java, Addison-Wesley Pub. Co, First Edition, 2001.

[44] J. Ferber, Les Système Multi-Agents : Vers une intelligence collective, Laforia, Université Pierre et Marie Curie,InterEditions, Paris, 1995.

[45] J. Gray, et al, Handling Crosscutting Constraints in Domain-Specific Modeling, CACM, vol. 44, no. 10, pp 87-93, October 2001

[46] J. Hannemann & G. Kiczales, Design Pattern Implementation in Java and AspectJ, to appear in Proceedings of OOPSLA 2002, 2002.

[47] J. Rosenberg, Some Misconceptions About Lines of Code, Fourth International Software Metrics Symposium, 5-7 Nov, pp. 137-142, 1997.

[48] J. Vlissides et al., Pattern Languages of Program Design, (Addison-Wesley) Vol. 2; 1996; ISBN 0-201-89527-7

[49] J. W. Cooper, Java™ Design Patterns: A Tutorial, Addition Wesley, ISBN: 0-0-201-48539-7, January 2000.

[50] J. Zhao, Towards a Metrics Suite for Aspect-Oriented Software, Technical Report SE-136-25, Information Processing Society of Japan, IPSJ 2003.

[51] J.C. Wichman, The Development of a Preprocessor to Facilitate Composition Filters in the Java Language, MSc. thesis, Dept. of Computer Science, University of Twente, 1999. http://trese.cs.utwente.nl/composition_filters

[52] J.O. Coplien, Software Patterns, Bell laboratories, The Hillside Group, Originally published by SIGS Books & Multimedia, 2000 at http://www.sigs.com.

[53] J.R. Wright, E.S. Weixelbaum, K. Brown, G.T. Vesonder, S.R. Palmer, J.I. Berman, and H.H. Moore, A knowledge-based configurator that supports sales, engineering, and manufacturing at AT&T network systems, Proceedings of the Innovative Applications of Artificial Intelligence Conference, pp.183--193, 1993.

[54] J.W. Cooper, The Design Patterns Java Companion, Addison Wesley Design patterns Series, October 2, 1998.

[55] K. Beck. Patterns and software development. Dr. Dobb's Journal, 19(2):18–23, 1994.

[56] K.D. Welker, Paul W. Oman, Software Maintainability Metrics Models in Practise, http://www.stsc.hill.af.mil/crosstalk/1995/11/Maintain.asp, 2004-06-04.

[57] L. Bergmans, Composing Concurrent Objects, Phd thesis, University of Twente, Netherlands, 1994.

[58] L. Bergmans, M. Aksit and B. Tekinerdogan, Aspect Composition Using Composition Filters, Chapter 12 of Software Architectures And Component Technolog, P 357-382.

[59] L. Bergmans, M. Aksit, Composing Crosscutting Concerns using

Composition Filters Supporting both intraclass and interclass crosscutting through model extension, ACM communication, Vol. 44, N°. 10, p. 51-57, October 2001.

[60] L. Bergmans, M. Aksit, Composing Multiple Concerns Using Composition Filters, TRESE Group, Department of Computer Science, University of Twente, 2000.

[61] L. Etzkorn, C. Davis, W. Li, A Statistical Comparison of Various Definitions of the LCOM Metric, The University of Alabama, Huntsville, TR-UAH-CS-1997-02, 1997.

[62] M. Aksit, K. Wakita, J. Bosch, L. Bergmans and A. Yonezawa, Abstracting Object Interactions Using Composition Filters, University of Twente, 2000.

[63] M. Aksit, L. Bergmans, Obstacles in Object Oriented Software Development, University of Twente, Netherlands, October 1992.

[64] M. Andersson, P. Vestergren, Object Oriented-Design Quality Metrics, Master thesis in Computer Science, Information technology, Computer Science Department, Uppsala University, Sweden, 2004.

[65] M. Lippert, C.V. Lopes, A study on Exception Detection and Handling Using Aspect-Oriented Programming, Proceedings of the 22nd international conference on Software engineering, Limerick, Ireland, ACM Press, pp. 418 – 427, 2000

[66] M. Meijers, Tool Support for Object-Oriented Design Patterns, Mcs. Thesis, Utrecht University, Netherlands, August 1996.

[67] M. Mickelsson, Aspect-Oriented Programming Compared To Object oriented Programming When Implementing A Distributed Web based Application, Mcs. Thesis, 2003.

[68] M.H.J. Glandrup, Extending C++ using the concepts of composition filters, Master thesis, University of Twente, Netherlands, November 20, 1995.

[69] N. Noda. T. Kishi, Implementing Design Patterns Using Advanced Separation of Concerns, OOPSLA 2001 Workshop on ASOC in OOS, 2001.

[70] N.E. Fenton, S.L. Pfleeger, Software Metrics A Rigorous & Practical Approach, Second Edition, PWS Publishing Company, 1997.

[71] N.M.N. Bouraqadi, T. Ledoux, Le point sur la programmation par aspects, Ecole de Mines de Nantes, Volume 20-n°4/2001, pp. 505-528, Juin 2000.

[72] O. Hachani & D. Bardou, Les aspects pour la réalisation de patrons de conception : Exemple de réalisation du patron Visiteur, Equipe SEGMA, LSR-IMAG, systèmes à composants adaptables et extensibles, 18 Octobre 2002.

[73] O. Hachani & D. Bardou, On Aspect-Oriented Technology and Object-

Oriented Design Patterns, Equipe SIGMA, LSR-IMAG 2003.

[74] O. Hachani & D. Bardou, Using Aspect-Oriented programming for Design Pattern Implementation, Equipe SIGMA, LSR-IMAG, 2002.

[75] O. Hachani, Apport de la programmation par aspects dans l'implémentation des patrons de conception par objets, Mémoire de DEA à l'université Grenoble, 25 Juin 2002.

[76] P.F. Perl-Schneider, M. Abrahams, L.A. Resnick, D.L. McGuinness, and A. Borgida, NeoClassic Reference Manual: Version 1.0, Artificial Intelligence Principles Research Department, AT&T Labs Research, 1996.

[77] P.S. Caro, Adding Systemic Crosscutting and Super-Imposition to Composition Filters, Master Thesis, Twente University, 2001.

[78] P.V. Van Winsen, (Re)engineering with Object Oriented Design Patterns, Master Thesis, Utrecht University, November 1996.

[79] R. Bosman, Automated Reasoning About Composition Filters, Ms. Thesis, University of Twente, Netherlands, 2004.

[80] R. Gamma et.al., Design Patterns - Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[81] R. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Massachusetts, 1995.

[82] R. Marinescu, Measurement and Quality in Object Oriented Design, University of Timisoara, 2002.

[83] R. Mulls, AspectJ Cookbook: Implementing Creational Object Oriented Design Patterns, Chapter 17, pp 175-187.O'Reilly edition 2003.

[84] R. Rajeev, et al, A distributed concurrent system with AspectJ, ACM SIGAPP Applied Computing Review, Vol. 9 Issue 2, July 2001

[85] R. Walker, et al, An Initial Assessment of Aspect-Oriented Programming. Proc. of the 21st International Conference on Software Engineering, pp. 120-130, 1999.

[86] R.C. Martin, Design Principles and Design Patterns, http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF, 2004-06-04.

[87] R.C. Martin, Object Oriented Design Quality Metrics an Analysis of Dependencies, http://www.objectmentor.com/resources/articles/oodmetrc.pdf, 2004-06-04.

[88] R.J. Brachman, P.G. Selfridge, L.G Terveen, B. Altman, A. Borgida, F. Halper, T. Kirk, A. Lazar, D.L. McGuinness, and L.A. Resnick, Integrated Support for Data Archaeology, International Journal of Intelligent and Cooperative Information Systems , 2:159-185, 1993.

[89] R.S. Pressman, Software Engineering A Practitioner's Approach, Fourth

Edition, McGraw-Hill, 1997.

[90] S. C. Soares, An Aspect Oriented Implementation Method, Msc. Thesis, Universidade Federal de Pernambuco Centro de Informatica, 2004.

[91] S. Chan, Design Pattern- Concept ands application, COMP414, Object-Oriented Technologies, Reading Project Report., The Hong Kong polytechnic university, 29 April 2002

[92] S. Clarke, R.J. Walker, Mapping Composition Pattern to AspectJ and Hyper/J, Department of Computer Science, Trinity Collage, University of British Columbia, 2001.

[93] S. Kébir et C. Araar, Hyper/J et Aspect/J : Une confrontation des concepts, Mémoire de fin d'études présenté pour l'obtention du diplôme d'ingénieur d'état en informatique, Option intelligence artificielle, Université de Annaba, Département d'informatique, Juin 2004.

[94] S. Koopmans, On the definition and the implementation of the Sina/st language, Mater thesis, university of Twente, Netherlands, August 25, 1995.

[95] S. Sauvage, Design Patterns for Multiagent Systems Design, university of Caen (France), 2004.

[96] S. Sauvage. Patterns orientés SMA : le pattern Marques. In Khaled Ghedira, editor, EcolIA–01, Ecole d'Intelligence Artificielle, pages 176–187, Hammamet (Tunisia), May 24–26 2001. URIASIS.

[97] S.G. Bruijin, Composable Objects with Multiple Views and Layering, M.Sc. thesis, University of Twente, Department of Computer Science, P.O. box 217, 7500 AE, Enschede, The Netherlands. March 1998.

[98] S.G. Bruijn, Composable Objects with Multiple Views and Layering, Mcs. Thesis, Unibversity of Twente, Netherlands, Mach 1998.

[99] S.R. Chidamber, C.F. Kemerer, A Metrics Suite For Object Oriented Design, M.I.T. Sloan School of Management, 1993.

[100] S.R. Chidamber, Chris F. Kemerer, A Metrics Suite For Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20, Issue 6, pp. 476-493, 1994.

[101] S.R. Chidamber, Chris F. Kemerer, Towards A Metrics Suite For Object Oriented Design, OOPSLA'91, pp. 197-211, 1991.

[102] T. Mens & A.H. Eden, On the evolution complexity of design Patterns, Electronic Notes in Theoretical Computer Science, SETra, 2004.

[103] T.J. McCabe, A Software Complexity Measure, IEEE Transactions on Software Engineering, Vol. 2, pp. 308-320, 1976.

[104] Tigris.org: Open Source Software Engineering Tools, aopmetrics project, Wroclaw University of Technology in Poland, http: www.tigris.org , 2005.

[105] W. Li, Another Metric Suite For Object Oriented Programming, The

Journal of Systems and Software, Vol. 44, Issue 2, pp. 155-162, 1998.

[106] W. Li, Sallie Henry, Maintenance Metrics for the Object Oriented Paradigm, Software Metrics Symposium, 21-22 May, pp. 52-60, 1993.

[107] W.B. McNatt and J. M. Bieman, Coupling of Design Patterns: Common Practices and Their Benefits, Computer Software & Applications Conf. (COMPSAC 2001), OCT. 2001.

[108] World Wide Web, url: http://www.eclips.com\AspectJ, Last visited December 2005.

[109] World Wide Web, url: http://ant.apache.org, Last visited December 2005.

[110] World Wide Web, url: http://www.virtualmachinery.com, Last visited October 2005.

[111] World Wide Web, url: http://www.clarkware.com/software, Last Visited, October 2005.

[112] World Wide Web, url: http://www.compuware.com, Last Visited, October 2005.

Annexe 1

DESCRIPTION DES PATRONS DE CONCEPTION

Cet annexe décrit, en détail, les trois patrons de conception utilisé dans notre étude: Adapter, Singleton et ChainOfResponsibility. Cette description est standard et inspirée de la description des patrons de GOF [80].

1. Patron Adapter

- **Nom:** Adapter
- **Objectif:** modifier le comportement d'une classe. Le patron Adapter garde les classes travaillent ensemble sans produire aucune incohérence dans le comportement de ces classes.
- **Autres Appellations:** Wrapper
- **Applicabilité:** Le patron Adapter est utilisable quand:
    1. on veut utiliser une classe existante, et son comportement n'égal pas a celui qui on est besoin.
    2. on veut créer une classe réutilisable qui coopère avec d'autres classes qui n'ont pas de comportement nécessairement compatibles.
    3. on est besoin d'utiliser plusieurs sous-classes existantes, mais c'est irréaliste d'adapter leur comportement en utilisant la technique d'héritage.
- **Structure:** Le patron Adapter utilise deux techniques pour adapter le comportement d'une classe: Il peut utiliser l'héritage multiple ce qui est monté par la figue A1.1 ou bien l'agrégation comme la montre la figure A1.2.

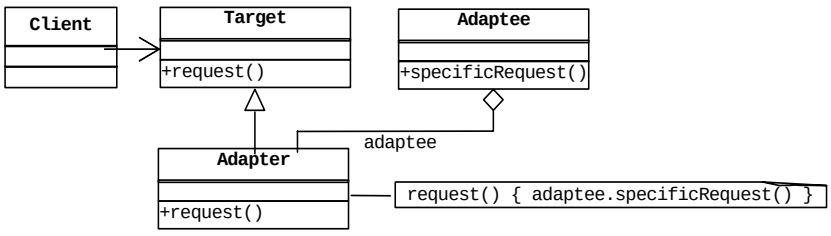


Fig. A1.1. Le patron Adapter a basa d'héritage

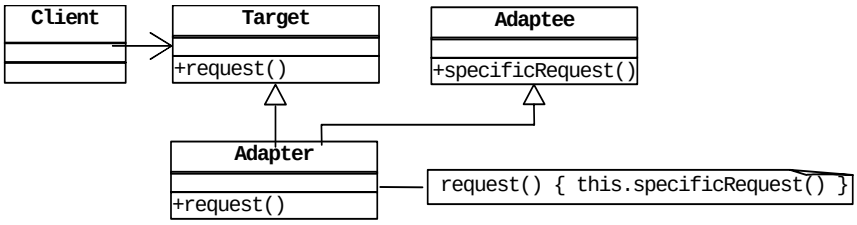


Fig. A1.2. Le patron Adapter a base d'agrégation

- **Participants:**

  1. Target: Définie le comportement désire par le Client.
  2. Adaptee: Définie le comportement actuel qui doit être adapté.
  3. Adapter: Adapte le comportement de la classe Adaptee avec le comportement de l'interface Target.

- **Collaborations:** La classe Client fait appel aux méthodes de l'instance de la classe Adapter qui redirige ces appels à une instance de la classe Adaptee.

- **Consequences:**

  1. Le client et les classes offrants les services sont indépendants.
  2. L'objet de la classe Adapter extend le programme par un niveau supplémentaire d'aggregation.
  3. La classe Adapter nous permet de cacher le comportement de la classe Adaptee quand elle est une sous-classe de la classe Adaptee.

- **Echantillon de Code en Java:** Le listing A1.1 au dessous présente une description du patron Adapter en langage Java.

```
// Adapter pattern
// "Adapter"
class Adapter extends Target {
  // Fields
  private Adapter adaptee = new Adaptee();
  // Methods
  public void request() {
    adaptee.specificRequest();
  }
}
// "Adaptee"
class Adaptee {
  //Methods
  public void specificRequest() {
    // specificRequest code
  }
}
```

Listing A1.1. Le patron Adapter en Java

- **Usages connus:** Le Modèle de la valeur qui fournit une interface value/value pour obtenir et mettre une valeur commune dans autres objets.

- **Patrons Similaires:**

  1. Bridge

2. Décorateur
3. Proxy

2. Patron Singleton

- **Nom:** Singleton
- **Objectif:** Assurez qu'une classe n'a qu'une seule instance dans le système.
- **Autres Appelations:** Aucune
- **Applicabilité:** Le patron Singleton peut être utilisé quand

    1. Il doit y avoir exactement une seule instance d'une classe, et ce doit être accessible aux clients d'un point d'accès célèbre.
    2. Quand l'instance devrait être extensible par héritage, et les clients devraient être capables d'utiliser une instance étendue sans modifier leur code.

- **Structure:**

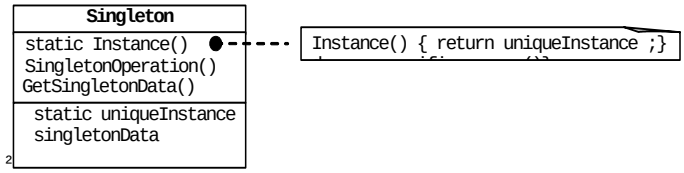


Fig. A1.3. Le patron Singleton

- **Participants:** Les classes et les objets participants au patron Singleton sont: la classe singleton qui doit être instancier une seule fois par le système.
- **Collaborations:** Les clients accèdent à une instance de la classe singleton uniquement via l'opération instance de singleton.
- **Conséquences:** Le patron Singleton a plusieurs avantages:

    1. Contrôler les accès à une instance
    2. Permet un nombre défini d'instance à une classe

- **Echantillon de Code en Java:** Le listing A1.2 décrit le patron singleton en langage Java.

```
// Singleton pattern --
// "Singleton"

class Singleton {

  // Fields
  private static boolean instance = false;
  // Constructor
  protected Singleton() throws SingletonException {
      if (instance) {
          throw new SingletonException ("One object is allowed");
      } else {
            instance = true;
```

```
            }
        }
        // Methods
        public void finalize() {
            instance = false;
        }
    }

    // "SingletonException"

    class SingletonException extends RuntimeException {

      //new exception type for singleton classes
      public SingletonException() {
          super();
      }
      public SingletonException(String s) {
          super(s);
      }
    }
```

Listing A1.2. Le patron Singleton en Java

- **Usages connus:** Un exemple d'utilisation du patron Singleton consiste à gérer le rapport entre les classes et leurs métaclasses. Une métaclasse est la classe d'une classe, où chaque métaclasse a une seule instance.

- **Patron similaires:**

    1. Abstract Factory
    2. Builder
    3. Prototype

3. Le patron ChaînOfResponsibility

- **Nom:** Chaîn Of Responsibility.
- **Objectif:** Évitez le couplage entre l'émetteur d'une requête et son récepteur en donnant une chance de traiter la requête par plus d'un objet. Enchaînez la requête reçue et leur faire passer à travers les objets membres de la chaîne afin d'être traité par un de ces membres.
- **Autres Appelations:** Aucune
- **Applicabilité:** On utilise le patron ChainOfResponsibility quand:

    1. Plus qu'un objet peut traiter une requête, et l'objet serveur n'est pas connu a priori. Le serveur devrait être constaté automatiquement.
    2. Vous voulez émet une requête à un ou plusieurs objets sans spécifier le serveur explicitement.
    3. l'ensemble d'objets qui peuvent traiter une requête devrait être spécifié dynamiquement.

- **Structure:**

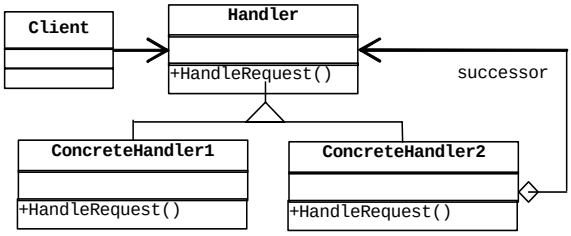


Fig. A1.4. Le patron ChainOfResponsibility

- **Participants:** Les classes et les objets participants dans cet patron sont:

  1. Handler: Définit une interface pour traiter les requêtes et spécifier les liens entre les successeurs
  2. ConcreteHandler: Traite les requêtes
  3. client: émetteur de la requête

- **Collaboration:** Quand un client diffuse une requête, la requête se propage à travers la chaîne jusqu'à un objet ConcreteHandler prend la responsabilité pour la traiter.

- **Consequences:**

  1. Réduire la dépendance entre l'émetteur et le récepteur - l'émetteur et le récepteur n'a aucune connaissance explicite sur l'autre.
  2. Le reçu de la requête n'est pas garanti - un émetteur pourrait parcourir toute la chaîne sans être traité
  3. La chaîne des serveurs peut être modifiée dynamiquement.

- **Echantillon de Code en Java:**

```
// Chain of Responsibility pattern
    // "Handler"
    abstract class Handler {
     // Fields
      protected Handler successor;
       // Methods
      public void SetSuccessor( Handler successor ) {
        this.successor = successor;
       }
      abstract public void HandleRequest( Object request );
    }
    // "ConcreteHandleri"
    class ConcreteHandleri extends Handler {
      // Methods
      public void HandleRequest( Object request )  {
        if ( // Can I handle request? ) {


                // handle request code
        } else {
            if (successor != null ) successor.HandleRequest(request);
          }
    }
```

Listing A1.3. Le patron ChainOfResponsibility en Java

- **Usages connues:** Plusieurs classes de la bibliothèque utilisent le patron ChainOfResponsibility pour traiter les événements des utilisateurs. Ils utilisent des noms différents pour la classe Handler, mais l'idée est la même: Quand l'utilisateur clique par la souris ou presse un bouton du clavier, un événement est produit et passé pour propager une chaîne de Handler.
- **Patrons similaires:** Le patron ChainOfResponsibility est souvent appliqué conjointement avec le patron Composite. Là, le parent d'un composant peut agir comme son successeur.

Annexe 2

DESCRIPTION DES MODELES PRATIQUES POUR LES APPROCHES ASOC

Cette annexe présente le détail des modèles pratiques utilisés dans notre étude pour les approches ASOC: AspectJ, ConcernJ et Hyper/J.

1. Le modèle AspectJ

La programmation sous AsectJ est très simple, dans le sens où le programmeur doit construire d'abord la partie fonctionnelle ordinaire en Java, puis il doit spécifier la ou les préoccupations nécessaires par une déclaration d'un ou plusieurs aspects sous AspectJ.

```
1    // declarations
2     aspect aspectName {
3
4      // pointcut definitions
5
6       pointcut pointcutName1: pointcutDefinition;
7       pointcut pointcutName1: pointcutDefinition;
8
9      // advice definition
10
11      before/after/around: pointcutExpression {
12
13     // code of advice to be added where corresponds
14    }
15 }
```

Listing A2.1. La définition d'un aspect en AspectJ

Le tableau suivant donne une description détaillée de syntaxe d'AspectJ.

Exigences	Primitives
Déclaration ordinaire d'un aspect	Modificateur aspect aspectName {}
Pour dire queaspectName1 hérite de aspectName2	Modificateur aspect aspectName1 extends aspectName2 {}
pour dire que aspectName1 implemente les l'interface de aspectName2	Modificateur aspect aspectName implements aspectName2 {}
Pour signaler que aspectName est aspect abstrait qui ne contient que des déclarations abstraites des points de jointure, pointcuts et des consignes.	Modificateur abstract aspect aspectName {}
pour donner à aspectName le droit d'accéder aux propriétés privates d'autres classes ou d'autres aspects.	Modificateur privileged aspect aspectName {}

spécifier que aspectName1 est prioritaire de aspectName2	Modificateur aspect aspectName1 dominates aspectName2 {}
Pour ne pas autoriser d'instancier aspectName plus d'une fois.	Modificateur aspect aspectName issingleton {}

Table A2.1. Description des primitives utilisées en AspectJ

Différents types de membres peuvent être introduit aux classes de système par l'utilisation des primitives d'introduction, le tableau suivant donne une récape de ces primitives.

Primitives	L'aspect sémantique
Modificateur Type ClassName.MethodName(paramaters){}	Introduire la méthode MethodeName à la classe className
Abstract Modificateur Type ClassName.MethodName(paramaters){}	Introduire la méthode abstraite MethodeName à la classe className
Modificateur ClassName.new(paramaters){}	Introduire un construceur à la classe className
Modificateur Type ClassName.field	Introduire la propriété field à la classe classeName

Table A2.2. Description des primitives d'introduction en AspectJ

AspectJ fournit un ensemble de primitives afin de spécifie les points de coupures, ces primitives sont résumer dans le tableau suivant.

Primitives	Signification	Remarques
Call(signature)	Appel d'une méthode ou d'un constructeur	Signature = type MethodName(parameters)
Exécution(signature)	Exécution d'une méthode ou d'un constructeur	Signature = type MethodName(parameters)
Get(signature)	Accées en lecture à une propriété	Signature = type className.field
set(signature)	Accées en écriture à une propriété	Signature = type className.field
Handler(signature)	Exécution d'une routine d'interruption	Signature = routine d'exception
If(signature)	Point de coupure logique	Signature = expression logique
Initialization(signature)	Instanciation d'un objet	Signature = identificateur d'un objet

Table A2.3. Description des primitives de points de coupure

La déclaration des consignes est possible sous AspectJ en utilisant les primitives suivantes.

primitives	sémantique
Before (parameters) : pointcuts {}	Déclaration d'un consigne avant
after (parameters) : pointcuts {}	Déclaration d'un consigne après
around (parameters) : pointcuts {}	Déclaration d'un consigne autour
Before/after/around (paramaters) throws exception: pointcuts {}	Générer une exception

Table A2.4. Description des primitives de consignes

2. Le modèle ConcernJ

Dans ce qui suit nous donnons la spécification de la syntaxe de concernJ. La spécification de la partie d'interface d'une className est présentée dans le listing suivant :

```
1   concern concernName {
2   filtermodule interfacePartName {
3   internals
4   // internals objects declaration
5   externals
6           // externals objects declaration
7   methods
8           // methods declaration
9   conditions
10          // conditions declaration
11  inputfilters
12          filterType : filterName = { FE*}
13  inputfilters
14          filterType : filterName = { FE*}
15  }
16  implementation begin in " Java " ;
17  // class which specifies the kernel object
18  }
```

Listing A2.2. ConcernJ specification of the CF object model

Le tableau suivant décrit la syntaxe générale pour la déclaration des différentes parties d'interface.

Les parties d'interfaces	Déclarations
internals	className objectName = new className();
externals	className objectName;
conditions	private boolean conditionIdentifier ();
Methods	Modificateur typeMethodName ;
inputfilters	FilterName: FilterKind = {FilterElements}
outputfilters	FilterName: FilterKind = {FilterElements}
superimposition	classNames

Table A2.5. Les primitives du modèle ConcernJ

La partie superimposition peur être spécifiée selon le listing suivant :

```
1   concern concernName {
2   filtermodule interfacePartName {
3   internals
4           // internals objects declaration
5   externals
6           // externals objects declaration
7   methods
8           // methods declaration
9   conditions
10          // conditions declaration
11  inputfilters
12          filterType : filterName = { FE*}
13  outputfilters
14          filterType : filterName = { FE*}
15  }
16  superimposition {
17  selectors
18          selectorName = {objects}
19  filtermodules
20          selectorName <- interfacePartName;
21  }
22  implementation begin in " Java " ;
23  // class which specifies the kernel object
24  }
```

Listing A2.3. Spécification de la superimposition en ConcernJ

3. Le modèle Hyper/J

Hyper/J est une extension du langage de programmation Java, avec une approche de séparation de préoccupations.

Pour bien comprendre le concept de l'hyperespace dans le contexte du langage Java, examinons l'exemple suivant :

hyperspace monHyperEspace
class A.Foo;
class A.Bar;
class B.Foo;

Cet exemple, est en fait une spécification d'un hyperespace complet. Il construit un hyperespace avec toutes les méthodes, les variables d'instance et les classes qui sont dans les paquets A et B. Les unités simples dans monHyperEspace sont : A.Foo.foo(), A.Bar.bar() et B.Foo.foo(). Et notons qu'on a deux classes nommées Foo, une dans le paquet A et l'autre dans B, et ceci pour une bonne raison qu'on verra ultérieurement.

Un hyperespace peut être aperçu comme une matrice multidimensionnelle, comme nous l'avons vu dans l'approche Hyperespace, et chaque axe représente une dimension dans cette matrice. Dans le paradigme de l'orientée objet, on a une seule dimension de préoccupations : L'Objet, et chaque classe dans l'application représente une préoccupation dans cette dimension.

Les coordonnées d'une unité dans la matrice, sont les préoccupations qu'affecte l'unité dans toutes les dimensions de l'hyperespace, et chaque unité doit affecter au plus une seule préoccupation pour une dimension donnée.

Dans Hyper/J, pour spécifier quelle unité est affectée à quelle préoccupation et dans quelle dimension on utilise le concern mapping, et Hyper/J organisera l'hyperespace selon le concern mapping, tout en s'assurant de mettre les unités qui appartiennent à l'espace des préoccupations, et qui ne sont affectées à aucune préoccupation dans une dimension donnée, dans la préoccupation none de cette dimension.

Prenons l'exemple précédant, de l'hyperespace monHyperEspace et supposons qu'on a le concern mapping suivant:

class A.Foo :Base.kernel
class A.Bar:Base.kernel
class B.Foo:Déboguage.logging

Les déclarations faites en haut, indiquent que les unités composées, A.Bar et A.Foo sont affectées à la préoccupation kernel de la dimension Base, et que l'unité B.Foo est affectée à la préoccupation logging dans la dimension Déboguage.

La réaction de Hyper/J face à la spécification de l'hyperespace et le concern mapping, est la création automatique d'une dimension appelée classFileDimension, et qui pour chaque classe déclarée dans la spécification de l'hyperespace, créera une préoccupation qui a le même nom que cette classe et qui renferme les différentes unités déclarées dans cette classe.

Après avoir parcouru le concern mapping, Hyper/J créera deux autres dimensions, Base et Déboguage et qui répartiront les préoccupations et les unités tel qu'il est indiqué dans le concern mapping.

Comme, seule l'unité B.Foo a été affectée à une préoccupation dans la dimension Déboguage, les autres unités seront automatiquement affectées à la préoccupation none de cette dimension.

La préoccupation kernel de la dimension Base, contiendra, ainsi, toutes les unités déclarées dans les classes A.Foo et A.Bar, à savoir, les deux méthodes A.Foo.foo() et A.Bar.bar().

Une des caractéristiques principales des déclarations du concern mapping, est qu'elles sont exclusives mutuellement, c'est à dire qu'une unité quelconque ne peut pas figurer dans deux préoccupations à la fois. Par exemple dans le concern mapping suivant :

package myPackage:Base.kernel
class myPackage.myClass : Base.classConcern
operation myPackage.myClass.myMethod:Base.methodConcern

Le contenu des différentes préoccupations est le suivant :

1. kernel, contiendra toutes les unités déclarées dans le paquet myPackage sauf la classe myClass et les différentes unités déclarées dans celle-ci.

2. classConcern contiendra toutes les unités déclarées dans la classe myClass sauf la méthode myMethod.
3. et enfin, methodConcern contiendra uniquement la méthode myMethod déclarée dans la classe myClass.

Une fois qu'on est identifié l'ensemble des préoccupations par la spécification du concern mapping, celles-ci (les préoccupations) peuvent être combinées dans un hyper-module aux moyens de la composition.

Prenons l'exemple précédant, et supposant qu'on veuille combiner la préoccupation logging avec la préoccupation kernel, ceci peut être réalisé en spécifiant l'hyper-module suivant :

hypermodule kernel&logging ;
hyperslices:
Base.kernel,
Déboguage.logging ;
relationships:
mergebyname;
end hypermodule ;

On remarque qu'on utilise les mêmes préoccupations que dans le concern mapping, mais on les note par le terme hyperslices, car il ne faut pas confondre hyperslice et concern (resp. hyper-tranche et préoccupation).

ABSTRACT

The recurrent problems that occur in object oriented software design are code tangling and code scattering. While, the first one occurs when we use more than one concern in the same application which complicates the source code, and decreases the reusability of it, the second problem occurs when the concern is not centralized and can't be encapsulated in a single entity. In the two cases, the concern code can't be reused efficiently in other application.

Both the code tangling and the code scattering problems affect the development process of the application in different manners: bad traceability, lack of productivity, weak code reusability and quality and difficult application evolution. To solve those problems, several techniques are being researched. Most of them attempt to increase the expressiveness of the Object Oriented paradigm. Such techniques are known as Advanced Separation Of Concerns.

On the other hand, design patterns are abstract descriptions of solutions to problems under certain constraints. Design patterns capture expert knowledge's and is abstracted from many successful designs. Design patterns can help develop more flexible and maintainable software. In addition, design patterns can help novices learn good programming faster and understand existing systems better.

The main purpose of this research is to demonstrate the efficiency of the applicability of patterns as a benchmark to evaluate ASOC approaches. Our idea consists of implementing design patterns using both OO paradigm and ASOC approaches and comparing resulting codes using both a qualitative and a quantitative metrics.

Our study is conducted within the multi-agents systems and particularly reactive ones that include object oriented systems. A brief discussion about using patterns in AI area is provided in order to benefit from the applicability of our work in a promising real framework.

RÉSUMÉ

Les problèmes les plus fréquemment capturés dans la conception orientée objets sont : l'enchevêtrement et la dispersion du code source des programmes. Le premier peut apparaître quand plus d'une préoccupation sont utilisées dans une même application et ça influe négativement sur la réutilisabilité de ses codes source. Le deuxième peut apparaître dans le cas où le code d'une préoccupation d'une application a été totalement dispersé et on ne peut pas l'encapsuler dans une seule entité, de ce fait, son code devient non réutilisable.

Ces deux problèmes : enchevêtrement et dispersion du code influent sur le processus de développement des logicielles de différentes manières : mauvaise traçabilité, manque de productivité, peu de réutilisation, et de qualité ainsi qu'une difficulté de l'évolution. Afin de résoudre ces problèmes, plusieurs techniques sont actuellement considérées, elles tentent d'augmenter le taux d'expression de l'approche orientée objets. Ces techniques sont connues sous l'appellation : Approches Avancées de Séparation des Préoccupations. Parmi ces approches on peut citer : la Programmation Orientée Aspects, les Filtres de Composition et la séparation multidimensionnelle des préoccupations.

D'un autre coté, les patrons de conception représentent des descriptions abstraites de solutions de problèmes sous certaines contraintes. Les patrons de conception rassemblent les connaissances des experts dans la conception des logiciels suite à de multiples expériences aboutis. Les patrons de conception peuvent nous aider à développer des logiciels plus flexibles et plus faciles à maintenir. En plus, ils permettent à des débutants de maîtriser rapidement les bonnes pratiques de programmation et comprendre plus aisément les systèmes logiciels existants.

Le but principal de ce travail consiste à démontrer l'efficacité de l'utilisation des patrons de conception comme benchmark pour l'évaluation des approches avancées de séparation des préoccupations. Notre idée consiste à implémenter les patrons de conceptions en utilisant à la fois l'approche orientée objets et les approches avancées de séparation de préoccupations et à comparer le code résultant on utilisant des métriques qualitatives et quantitatives.

Notre étude prend comme domaine d'application les systèmes multi-agents et particulièrement ceux qualifiés de réactifs, qui englobent les systèmes orientés objets. Une brève discussion sur l'utilisation des patrons de conception dans le domaine de l'I.A est apportée afin de tirer profit de l'application de notre approche dans un cadre réel prometteur.