Université de Caen
UFR de Sciences
Département d'Informatique
équipe MAD

École des Mines de Douai
Centre de Recherche
GIP
équipe CSL

Mise en œuvre de la programmation par aspects dans le cadre des systèmes multi-agents

---

Stage de DEA réalisé au sein des équipes MAD (GREYC) et CSL
Encadrants : **Serge Stinckwich** et **Noury Bouraqadi**
Période du stage : Avril - Septembre 2003

Mémoire de DEA rédigé par
Romain ROBBES
Le 26 septembre 2003

Remerciements

- Je tiens à remercier Anne Nicolle et Bernard Morand pour s'être intéressés à ce travail et avoir bien voulu faire partie du jury.
- Je remercie aussi Noury Bouraqadi et Serge Stinckwich, mes deux maîtres de stage, pour l'aide et l'encadrement qu'ils m'ont fournit.
- Plus généralement, je remercie le département informatique du GREYC et celui de l'École des Mines de Douai pour avoir bien voulu m'accœuillir et me fournir chacun un endroit pour travailler.
- Je remercie une nouvelle fois mes deux encadrants, ainsi que l'ESUG, pour m'avoir permit d'assister à la conférence de l'ESUG cet été, pendant une semaine en Slovénie, qui reste un souvenir inoubliable.
- Enfin je remercie tous mes camarades pour leur soutien, ils sont trop nombreux pour être cités ici.
- Et je remercie tout lecteur éventuel de ce mémoire qui ne se serait pas reconnu, pour l'intérêt qu'il y manifeste ...

Table des matières

1 Introduction **5**

2 Présentation des concepts **7**
2.1 Programmation par Aspects 7
2.1.1 Les limites des technologies actuelles 7
2.1.2 La solution proposée par l'AOP 11
2.2 Systèmes Multi-Agents 15
2.2.1 Définition d'un agent 15
2.2.2 Comportement d'un agent 15
2.2.3 Systèmes et organisations 16
2.2.4 Architecture Groupe/Agent/Rôle 16
2.2.5 Raisons de ce choix 19

3 Trois facettes de la relation AOP/SMA **20**
3.1 État de l'art 20
3.1.1 Sur les liens entre AOP et SMAs 20
3.1.2 De l'importance des rôles 21
3.1.3 Relation entre rôles et aspects 22
3.2 Trois facettes de la relation AOP/SMA 23
3.3 AOP dans l'infrastructure SMA 23
3.3.1 Bénéfices 23
3.3.2 Recherche d'accointances 23
3.3.3 Construction de messages 25
3.3.4 Communications distantes 27
3.3.5 Autres aspects de communication 27
3.3.6 Contrôle du flot d'exécution 28
3.3.7 Protocole des rôles 28
3.3.8 Gestion des ressources 29
3.4 AOP pour le développement SMA 29
3.4.1 Groupes vs Aspects : Similitudes 30
3.4.2 Différences entre groupes et aspects 31

3.4.3 Différences entre programmation objets et programmation agents pour l'intégration de l'AOP 33
3.4.4 Conclusions sur l'analogie Groupe/Aspect 34
3.4.5 Agentification des aspects 34
3.4.6 Conclusion : complémentarité des groupes et des aspects 36
3.5 SMA pour l'AOP 38

4 Notre adaptation du modèle Aalaadin 39
4.1 Limitations du modèle Aalaadin 39
4.1.1 Un modèle «trop abstrait» 39
4.1.2 Incapacité à décrire certains aspects sous forme de groupes 40
4.2 Extensions apportées au modèle 41
4.2.1 Concrétisation par la réification 41
4.2.2 Instauration de méta-rôles 43
4.3 État de l'implémentation 44
4.3.1 Démarche de développement 44
4.3.2 Présentation de l'implémentation 45
4.3.3 Utilisation du framework MetaclassTalk 45
4.3.4 Aspect de communication à distance 46
4.3.5 Comparatif avec MADKit 46

5 Conclusion et travaux futurs 47
5.1 Résumé des travaux 47
5.2 Travaux concernant la suite du stage 48
5.3 Autres pistes de recherches 48
5.3.1 Continuer l'exploration et l'implémentation d'aspects d'infrastructures 48
5.3.2 Améliorer le support des aspects agents 50
5.3.3 Expliciter les apports des SMAs pour l'AOP 50
5.4 Sur le stage 51

Table des figures

2.1 "Vue d'artiste" d'une application de commerce électronique . . 8
2.2 Mélange des préoccupations dans le code source 9
2.3 Dispersion des préoccupations au sein de l'application 10
2.4 Développement et tissage des aspects 11
2.5 Exemple de structure organisationnelle 18
2.6 Exemple de système multi-agents 19

Chapitre 1

Introduction

Les systèmes multi-agents constituent à ce jour une des disciplines de l'informatique où les applications sont les plus difficiles à développer. Les causes de ce fait sont nombreuses. En effet, l'intelligence artificielle distribuée est une discipline dont les conditions de travail sont complexes :

- Chaque agent est autonome
- Chaque agent s'exécute dans un environnement distinct. Les programmes sont donc distribués
- Les agents communiquent par envoi de messages qui peuvent être complexes, allant parfois jusqu'à employer le langage naturel
- Les agents ont des connaissances sur le monde incertaines ...
- ... mais raisonnent dessus pour accomplir leurs buts

Les caractéristiques énumérées ci-dessus ne sont qu'une partie de celles pouvant figurer dans un système multi-agents, mais elles laissent entr'apercevoir les difficultés qui peuvent survenir pendant le développement d'une telle application. Le principal problème est qu'il faut le plus souvent prendre en compte toutes ces propriétés en même temps, sans disposer d'outils appropriés.

Parallèlement, le paradigme de la programmation par aspects [KLM+97](abrégé par AOP, pour Aspect Oriented Programming) s'est imposé comme une façon efficace de parvenir à la séparation des préoccupations (separation of concerns en anglais [Par72]), chère au génie logiciel. Ce domaine de recherche est à la fois récent, et peu connu, nous le présenterons. L'AOP n'a donc été que peu appliqué aux autres branches de l'informatique, en particulier l'IAD.

C'est donc dans ce contexte, à l'interface à la fois de l'intelligence artificielle distribuée et du génie logiciel, que s'inscrit le sujet de ce stage. Il nous a en effet semblé que l'utilisation de l'AOP dans les applications multi-agents pourrait aider à la résolution d'une partie des problèmes dont souffre le développement en IAD. Cependant cette affirmation a besoin de soutien pour être

prise en compte, et c'est ainsi que le sujet de ce stage à vu le jour : il s'agit donc d'étudier en détail les relations possibles entre programmation orientée aspects et systèmes multi-agents. Ceci se faisant en prenant en compte l'étude de cette relation sous plusieurs angle, afin d'en avoir la vision la plus large possible, notamment en envisageant aussi la relation inverse à celle de départ, à savoir les apports que peuvent avoir les SMAs pour l'AOP.

Le compte-rendu de ce stage est donc organisé de la manière suivante : Je vais dans un premier temps présenter de façon plus complète les concepts sur lesquels se basent notre réflexion, qui sont principalement l'AOP et les SMAs. Je continuerais ensuite en exposant les résultats de mon stage, qui se présentent sous la forme de l'isolation et l'étude de trois facettes de la relation entre AOP et SMAs, et aussi de la définition d'un modèle de système multi-agents original, dérivé du modèle Aalaadin, proposé par Jacques Ferber [FG98]. Enfin je conclurai et présenterai les pistes de recherche envisageables lors d'une poursuite en thèse.

Chapitre 2

Présentation des concepts

Le sujet de mon stage se situant à l'interface de deux domaines de recherche distincts, il est nécessaire de les présenter tous les deux pour ne pas désorienter les spécialistes de l'une ou de l'autre discipline. C'est donc l'objectif de cette partie, qui va présenter la programmation orientée aspects dans un premier temps, puis les systèmes multi-agents, et plus particulièrement l'architecture groupe/agent/rôle, qui sera utilisée par la suite.

2.1 Programmation par Aspects

La programmation par aspects (AOP) est un paradigme de programmation visant à aider le plus possible à accomplir la séparation des préoccupations dans une application. Ce principe stipule que les différentes fonctionnalités d'une application doivent être le plus séparée possible, afin de pouvoir évoluer indépendamment les une des autres [Par72]. L'AOP a été introduite en 1996 par Gregor Kiczales [KLM+97], qui travaillait auparavant dans le domaine de la réflexivité, ayant été notamment un des concepteurs de CLOS [KAJ+] [GKB91].

2.1.1 Les limites des technologies actuelles

Les technologies utilisées actuellement dans le développement des applications présentes des limitations quant à la prise en compte des diverses préoccupations des applications. Ces limitations vont être exposées maintenant, par le biais d'un exemple support, et consistent principalement dans les deux problèmes que sont le mélange du code des diverses préoccupations, et sa dispersion dans celui de toute l'application.

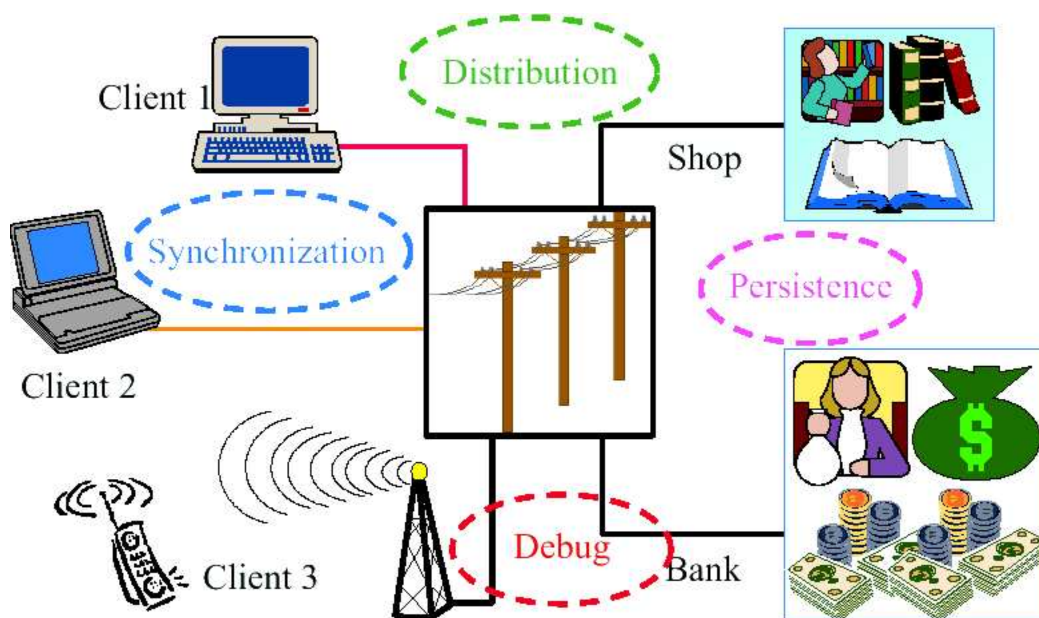

FIG. 2.1 – "Vue d'artiste" d'une application de commerce électronique

Example-Support

La figure 2.1 représente de façon imagée une application de commerce électronique, ainsi que quelques préoccupations auxquelles elle doit faire face : il y a bien sur les préoccupations au niveau métier (ce que fait réellement l'application), mais aussi d'autres préoccupations dues au contexte dans lequel l'application est déployée, qui sont les préoccupations de persistance, distribution, synchronisation et de débogage.

En effet, cette application doit faire face aux préoccupations suivantes :

Préoccupation métier : l'application doit remplir correctement la tâche pour laquelle elle a été conçue, à savoir permettre à des clients d'acheter des livres sur Internet, voire à partir de leur téléphone.

Distribution : cette application étant une application de commerce électronique sur Internet, elle s'exécute naturellement sur plusieurs machines, qui sont les serveurs de l'entreprise l'utilisant, et les postes des clients.

Synchronisation : comme les clients peuvent accéder à certains articles simultanément, ils ne doivent pouvoir subir que des modifications ato-

```
price
   | currentPrice |
   self halt.
   lock critical:
  [currentPrice := price].
    ^ currentPrice

price: new
    lock critical:
        [price := new].
    database
        setPrice: new
        for: self .
    Transcript show:
        'price is : ', new
```

FIG. 2.2 – Mélange des préoccupations dans le code source

miques par exemple (pour éviter que deux clients commandent en même temps le dernier exemplaire en stock d'un livre)

Persistance : l'application doit maintenir une base de données pour survenir à des défaillances. Celle-ci doit être mise à jour à chaque modification d'une des données sur lesquelles l'application travaille.

Débogage : pendant le développement de l'application, les développeurs peuvent avoir besoin d'inclure du code de trace, ou autre (points d'arrêts), pour détecter les éventuels défauts dont pourrait souffrir l'application.

Mélange du code

La superposition des préoccupations dans l'application se traduit au niveau du code source par des passages incessants d'une préoccupation à une autre, comme le montre la figure 2.2. Dans celles-ci les diverses préoccupations de l'application sont symbolisées par des lignes de code de différentes couleurs, le noir symbolisant le code métier de l'application. Le code métier se trouve donc délayé dans le code des autres préoccupations, et est donc à la fois difficilement lisible, compréhensible, et maintenable. Cette considération s'applique d'ailleurs à toutes les préoccupations recensées dans l'application. Ce qui s'applique à la lecture et à la maintenance du code s'applique encore plus à son écriture : le développement d'une fonctionnalité associé à la prise en compte d'autres fonctionnalités est difficile, long et sujet à erreurs.

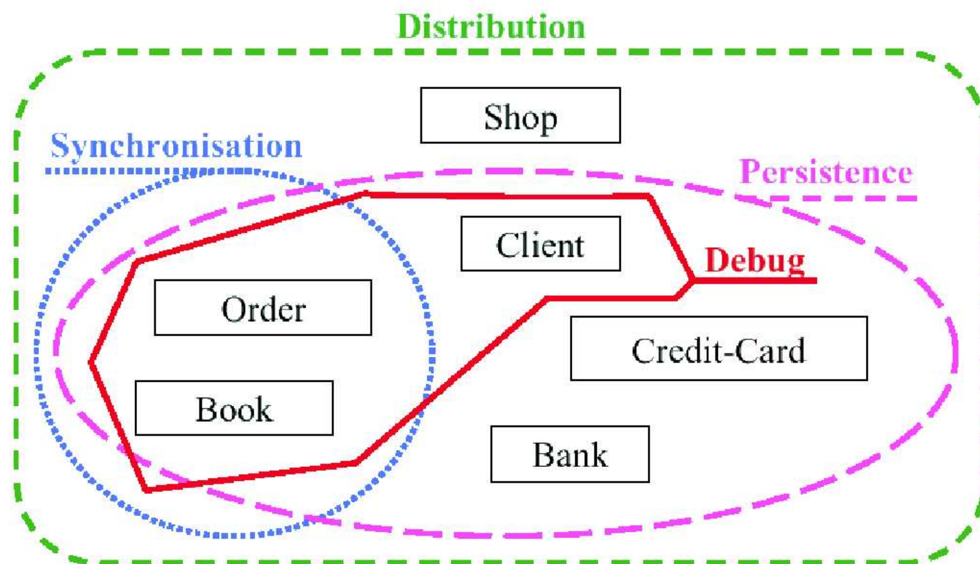

FIG. 2.3 – Dispersion des préoccupations au sein de l'application

Dispersion et duplication du code

En prenant l'exemple de la préoccupation de synchronisation (voir figure 2.3), on s'aperçoit que la plupart des préoccupations de l'application sont :

-répétées un grand nombre de fois dans le code : à chaque modification d'une variable d'instance d'un objet pouvant subir des accès concurrents, ces contraintes doivent être présentes, ce qui peut donc être très fréquent.

-dispersées dans toute l'application : Les objets nécessitant ce type de contrôle peuvent appartenir à plusieurs classes (dans notre exemple : Livre, Client ...), et le code de la préoccupation de synchronisation peut donc se trouver dispersé dans plusieurs endroits distincts.

Ces deux faits compliquent donc singulièrement la compréhension de chacune des préoccupations, car le lecteur du code ne peut pas avoir l'ensemble du code de la préoccupation regroupé comme il le désirerait. De plus comme une partie de ce code peut être dupliquée un grand nombre de fois, il est très difficile à modifier. Comment changer facilement la politique de persistance d'une application, si il faut manuellement changer des centaines d'accès ?

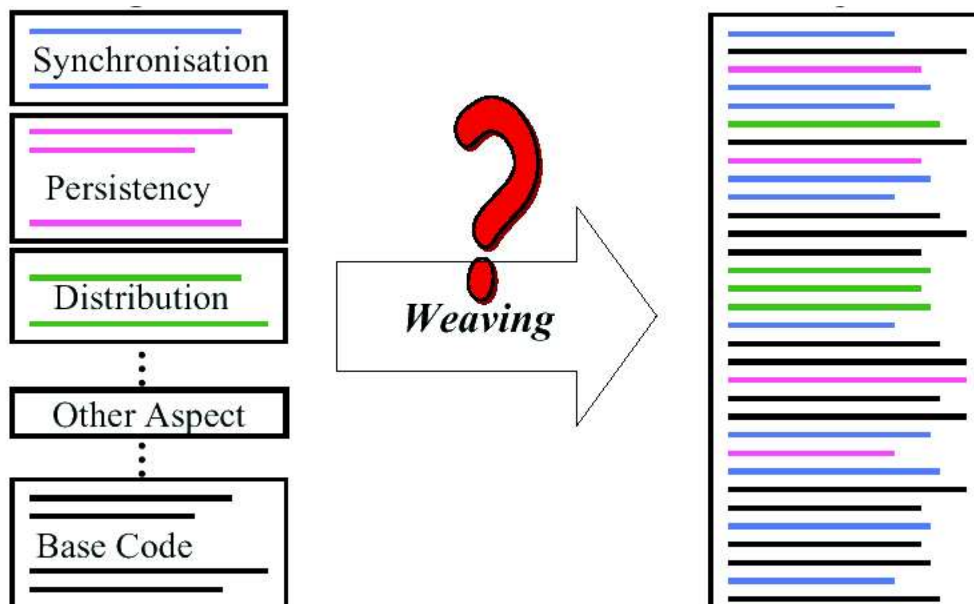

FIG. 2.4 – Développement et tissage des aspects

2.1.2 La solution proposée par l'AOP

Ces deux problèmes majeurs rendent difficile la construction d'une application ayant d'importants besoins non-fonctionels. Examinons donc la solution proposée par l'AOP.

Définition d'un aspect

Au cœur de la programmation orientée aspect intervient donc la notion d'aspect, qui est le terme employé pour désigner une des préoccupations de l'application. La définition exacte d'un aspect est donc la suivante :

> Un aspect est une propriété transverse à l'application [1].

Autrement dit, un aspect est une unité de code accomplissant une tâche bien définie, mais qui touche différentes parties de l'application. L'AOP essaie de résoudre ces problèmes en isolant justement ces aspects dans des unités séparées, et en se chargeant de leur intégration par la suite, selon les directives du développeur.

[1]Définition employée par Noury Bouraqadi.

Le processus est plus facilement visualisé par la figure 2.4 : cet exemple reprend l'hypothétique application de commerce électronique utilisée plus haut, et montre son développement en utilisant des technologies AOP. Ici, l'application est découpée selon les diverses préoccupations identifiées plus haut qui sont la synchronisation, la persistance, la distribution, ainsi que le code de base, qui représente le code métier de l'application. Chacun des aspects code spécifiquement sa fonctionnalité, et rien d'autre. Ainsi un aspect de synchronisation contiendra un code permettant de gérer les accès concurrents de façon entièrement générique, par exemple. Parallèlement, le développeur implémente de même le code de base de l'application, qui consiste donc en l'ensemble des classes nécessaires à la faire fonctionner, comme ici les classes Livre, Commande, Compte ...

Intégration des aspects au code de base

Il reste ensuite au développeur à spécifier les points où les aspects se greffent dans le code de base, où ils vont prendre le pas ou modifier le code de base. Par exemple, l'aspect de persistance doit être déployé de sorte qu'il intervienne à chaque accès à une variable d'instance d'un objet nécessitant ce mécanisme. Cette étape de déploiement se fait généralement dans la partie de configuration de l'aspect (partie de l'aspect spécifiant comment le tisser sur l'application), sous la forme de «points de jonctions», qui sont une manière de répertorier facilement diverses localisations dans le code. Une fois la configuration spécifiée, un programme se charge de produire le code exécutable approprié. Ce programme est appelé *weaver*, ou *tisseur* en français. On retrouve donc, comme indiqué sur la figure 4 un code où toutes les préoccupations sont prises en compte en même temps.

Les points de jonction choisis les plus couramment sont les appels de méthode (avant ou après une méthode, voire en remplaçant totalement l'appel d'une méthode [2]), ainsi que les accès à une variable d'instance [3]. Ils offrent la flexibilité nécessaire à l'intégration de tout les aspects.

Processus de tissage

Le processus de tissage est celui qui va intégrer réellement le code des aspects afin de produire le code exécutable, selon les directives formulées par

[2] c'est utile par exemple pour implémenter un système de cache évitant d'appeler plusieurs fois une méthode coûteuse.

[3] on peut aussi noter que l'AOP n'est pas lié spécifiquement à la programmation orientée objets. L'AOP est aussi applicable à la programmation procédurale [CKFS01], voire fonctionnelle (le concept de monade peut alors être utilisé pour implémenter des aspects [Meu97]).

le développeur au moyen des points de jonction. Il peut prendre plusieurs formes, basée sur deux approches distinctes, une plutôt statique, et l'autre plus dynamique.

L'approche statique est celle employée par le prototype de Gregor Kiczales, nommé AspectJ [KHH+01]. C'est un outil permettant de faire de l'AOP en Java. Il fonctionne de la manière suivante : Le développeur fournit en entrée le code de base de l'application, celui des aspects ainsi que la spécification de l'intégration, puis AspectJ génère du code Java comprenant le code de base étendu avec celui des aspects, avant d'appeler le compilateur Java classique.

L'approche dynamique est le plus souvent basée sur la réflexion et les MOP (Meta Object Protocol, Protocole de Méta Objet [GKB91]), qui ajoutent des capacités de réflexion à l'application, permettant ainsi de capturer les envois de messages et les accès aux variables d'instance. Parmi ces systèmes figure MetaclassTalk ([Bou99],[BL02],[Bou03]), le système employé pour l'implémentation de la plate-forme SMA introduite par la suite dans ce rapport. Les aspects sont implémentés sous la forme de méta-objets, agissant comme des interpréteurs des objets du niveau de base. Ils permettent par exemple de tisser ou détisser des aspects au cours de l'exécution de l'application, aidant ainsi grandement à l'implémentation d'applications auto-adaptatives. Cette caractéristique est leur principal avantage.

Intérêt de l'AOP

L'utilisation de l'AOP offre de nombreux avantages et contribue grandement à l'augmentation de la qualité générale du code de l'application, en isolant le code de base de celui des aspects. Examinons ces conséquences une par une.

Le code est plus clair En effet, il devient beaucoup plus facile de comprendre le code si les fonctionnalités sont nettement séparées. Le développeur peut ainsi se focaliser plus facilement sur la partie de code qui l'intéresse, sans avoir à ce soucier des autres. Cela représente déjà une amélioration majeure sur la situation présentée sur la figure 2, illustrant le problème du mélange du code des diverses préoccupations.

Le code est plus facilement modifiable Le développeur peut plus facilement modifier le code de base et celui des aspects, car il n'a pas à remplacer toutes les occurrences d'un code. Il peut simplement modifier le code qui est contenu dans un aspect, le problème de la dispersion étant résolu.

L'application est déployable avec plus de flexibilité On peut choisir beaucoup plus facilement quels aspects sont a déployer selon le contexte demandé. Si en cours de déploiement l'aspect de débogage est approprié, il vaut mieux l'enlever pour livrer le produit à l'utilisateur. De même, l'utilisation d'un aspect de communications distantes n'est utile que si l'application s'exécute réellement sur plusieurs machines, et doit être enlevé le cas échéant. L'AOP permet donc une plus grande flexibilité dans la configuration des applications. On peut aussi choisir entre plusieurs politiques concernant un aspect particulier (comme la distribution plus haut), en choisissant l'aspect avec lequel l'application sera composée.

Les aspects sont réutilisables Les aspects comme la communication distante, la persistance, s'ils sont écrits de façon assez générique, peuvent être facilement réutilisés dans d'autres applications, pourvu que les parties de code et de déploiement de l'aspect soient clairement séparées.

Conclusion sur l'AOP L'utilisation de l'AOP permet d'écrire un code plus clair et de meilleure qualité, qui est de plus beaucoup plus facilement réutilisable, tant au niveau des aspects qu'à celui du code de base, qui est lui plus facilement configurable. L'AOP est donc une bonne solution aux problèmes de mélange et de dispersion du code habituellement rencontrés quand on essaye de faire cohabiter plusieurs préoccupations dans une application.

Par contre, le code est parfois un peu plus difficile à concevoir, car la rédaction d'aspects nécessite des capacités d'abstraction pour comprendre les concepts impliqués. De plus, il faut savoir quels aspects sont liés au code de base pour savoir ce qui se passe réellement (il peut y avoir beaucoup d'effets de bords). Enfin, les problèmes de composition d'aspects utilisant les même points de jonction sont parfois non triviaux à résoudre.

2.2 Systèmes Multi-Agents

Les système multi-agents sont un domaine de recherche assez vaste, qu'il est difficile de décrire succinctement. Nous essayerons cependant de nous limiter au strict minimum nécessaire à notre étude.

2.2.1 Définition d'un agent

Dans la littérature, un agent est généralement une entité possédant les propriétés suivantes :

autonomie : un agent est capable de vivre sans se référer constamment aux autres, de prendre des décisions seuls.

adaptation : un agent peut réagir à des changements dans son environnement.

apprentissage : certains agents peuvent apprendre de leurs expériences passées, et ainsi modifier ou accélérer leur prise de décision.

interaction : un agent a la possibilité de percevoir et d'agir sur son environnement par le biais de capteurs et d'effecteurs. Il peut ainsi communiquer avec d'autres agents par ce biais.

collaboration : dans le cadre d'un système multi-agents, les agents peuvent élaborer des stratégies involvant plusieurs agents dans des rôles distincts.

Ces caractéristiques sont parmi les plus générales permettant de décrire un agent, et pourtant toutes ne concernent pas les agents les plus simples comme les agents réactifs. D'autres sont encore plus spécifiques, comme la mobilité de certains agents logiciels, ou les ressources limités des agents physiques.

2.2.2 Comportement d'un agent

Le comportement le plus général d'un agent peut être une boucle rétroactive de la forme :

1. Recevoir des informations de l'environnement ou des autres agents par ses capteurs.
2. Raisonner sur les informations en sa possession et déterminer la meilleure action à faire.
3. Traduire et transmettre cette information aux effecteurs pour modifier l'environnement.

Nous allons maintenant présenter les deux écoles les plus courantes pour développer un agent.

Agents réactifs

Un agent réactif est un agent au comportement simple, défini par des règles du type condition -> action. Si une des conditions est remplie, la partie action de la règle concernée est exécutée. Ce comportement est donc simple à implémenter, et rapide à exécuter.

Agents délibératifs

Ces agents ont un comportement plus complexe que les précédents. Si ils utilisent eux aussi des capteurs et des effecteurs, leur étape de raisonnement est beaucoup plus approfondie, car ils adoptent souvent une architecture de type BDI [4], et maintiennent donc des croyances sur leur environnement, eux-mêmes, et les autres agents, accompagnés de buts à réaliser, le tout pouvant être remis en question d'un instant à l'autre. Ils sont donc plus difficiles à implémenter et plus lent à prendre des décisions.

2.2.3 Systèmes et organisations

Nous avons jusqu'alors mentionné seulement les agents isolés, ce qui n'est pas suffisant pour parler d'intelligence artificielle distribuée. On parle donc de système multi-agents dès qu'il y a plusieurs agents. Les interactions entre les divers agents ont dès lors besoin d'être structurés par une organisation. celle-ci peut être des plus simples, comme dans le cas des fourmis artificielle, qui ne communiquent que par leur environnement, soit une organisation plus complexe, comme le modèle Groupe/Agent/Rôle, qui a retenu ici notre attention.

2.2.4 Architecture Groupe/Agent/Rôle

L'architecture Groupe/Agent/Rôle a été développée principalement par Jacques Ferber et porte le nom de modèle Aalaadin. Elle vise principalement à faciliter la conception de systèmes multi-agents en introduisant les concepts de groupes et de rôles pour mieux la structurer. Voici donc les définitions des principaux termes de la terminologie d'Aalaadin.

Groupe

Un ensemble d'agents. Un agent peut joindre plusieurs groupes, mais son admission n'est pas automatique. Un groupe définit la cardinalité des rôles

[4] Belief, Desire, Intentions.

(nombre d'agents pouvant remplir le même rôle simultanément) qui le composent. L'admission d'un agent dans un groupe n'est pas automatique, car le groupe définit les pré-requis à l'obtention d'un rôle, qui sont par exemple les compétences que doit posséder l'agent. Ces pré-requis peuvent aussi être des rôles que l'agent doit avoir déjà obtenu, ou encore être liés à des droits d'accès ...

Les groupes sont définis par une structure de groupe, qui énumère tout les états possibles d'un groupe (combinaisons valides de rôles pourvus). Cette structure est ensuite instanciée dynamiquement sous forme de groupes, pouvant être incomplets (tous les rôles ne sont pas pourvus).

Agent

Entité active et communicante, jouant des rôles dans des groupes. Il n'y a pas dans Aalaadin de contraintes supplémentaires sur les agents. Un agent peut donc assumer autant de rôles qu'il le désire, et ce dans un nombre de groupe tout aussi arbitraire.

Rôle

Une représentation abstraite de la fonction d'un agent ou d'un service, qui peut aussi servir à identifier un agent dans un groupe. Un agent peut endosser plusieurs rôles, ils seront locaux au groupe dans lequel l'agent endosse le rôle. L'admission n'est pas automatique. Ces rôles ont les caractéristiques suivantes [5] :

Unicité : Un rôle peut être unique ou multiple, i.e. peut être occupé par un ou plusieurs agents selon son type.

Compétences : Ce sont les conditions qu'un agent doit satisfaire pour endosser le rôle en question.

Capacités : Propriétés qu'un agent acquiert quand il joue un rôle particulier.

Les rôles n'existent pas toujours dans l'implémentation des agents. Par exemple dans MADKit (L'implémentation d'Aalaadin réalisée par Ferber), leur implémentation est laissée au développeur, qui peut donc soit les implémenter «à la main», soit utiliser un mécanisme plus raffiné.

Structure Organisationnelle

Une structure organisationnelle définit un ensemble de groupe formant un système multi-agents en ayant une relation analogue à celle reliant groupes

[5] En reprenant les définitions de Jacques Ferber pour Aalaadin.

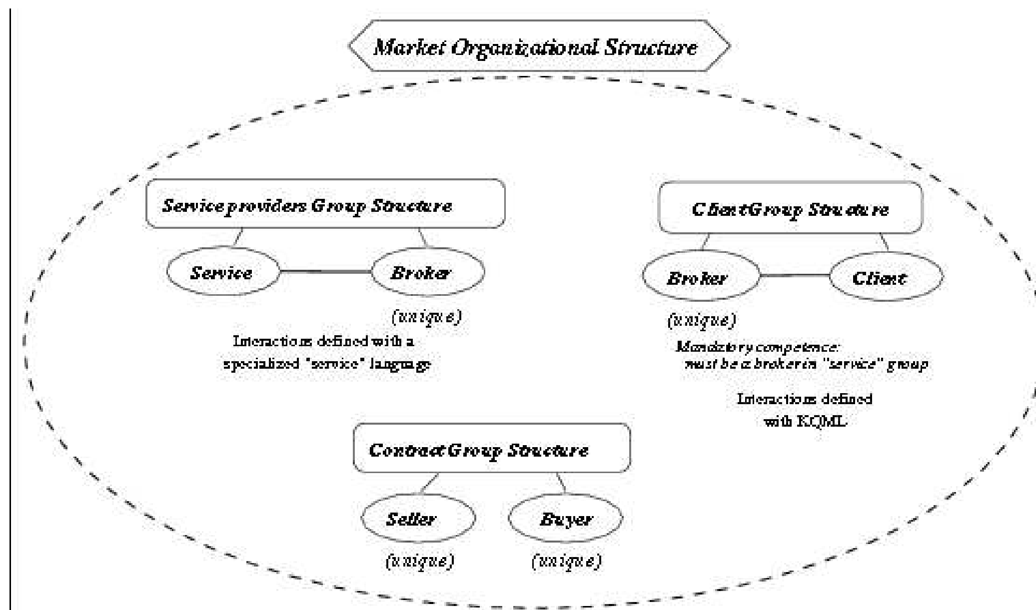

FIG. 2.5 – Exemple de structure organisationnelle

et structures de groupes. Elle est instanciée sous la forme d'une organisation. C'est donc la spécification du problème initial. De la même façon que pour les groupes, les instanciations d'une structure organisationnelle ne sont pas forcément complètes (elles ne peuvent contenir qu'une partie des groupes définis)

Exemples

Toutes ces définitions sont plus compréhensibles avec un exemple, comme celui montré par la figure 2.5. Cet exemple extrait du rapport sur Aalaadin [FG98].illustre l'organisation d'une structure de marché, organisée selon l'algorithme de Contract-Net : les fournisseurs et les clients adressent leur demandes à un Broker, qui se charge de les mettre en relation par la suite. La structure organisationnelle spécifie les groupes possibles (en l'occurrence un groupe d'acheteurs, un de vendeurs, et plusieurs groupes acheteurs-vendeurs instanciables dynamiquement). Chacune des structures de groupes spécifie ensuite les cardinalités (nombres de rôles possible par groupe) pour chacun des rôles.

Ceci constitue la description statique d'une structure organisationnelle, voyons comment cela se traduit lors de l'exécution du système. La figure 2.6

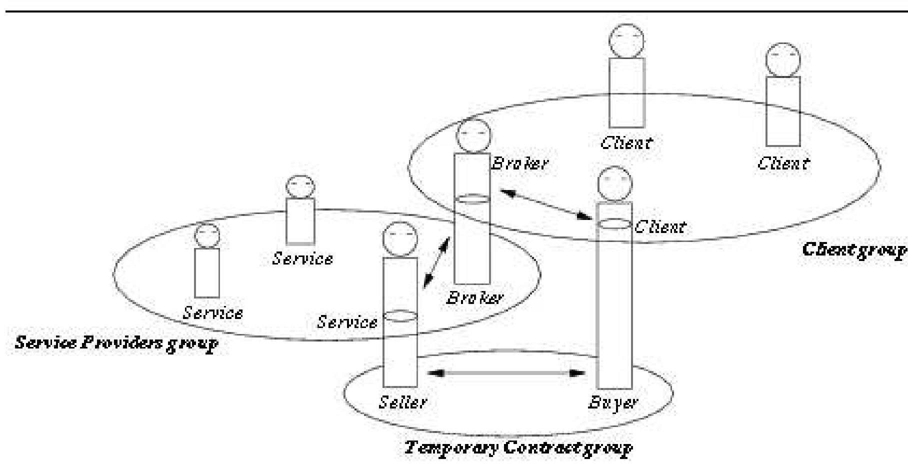

FIG. 2.6 – Exemple de système multi-agents

est un exemple d'instantiation de cette structure organisationnelle sous la forme d'une organisation. Sur cette figure, les ellipses représentent les groupes actuellement instanciés : un groupe d'acheteurs, un groupe de vendeurs, et un groupe ou deux agents (un client et un fournisseur) on été mis en relation pour échanger des marchandises. La multiplicité des rôles est illustrée par le fait que trois agents occupent deux rôles : l'un occupe les deux rôles de Broker, et les deux autres sont à la fois membre de leur groupe en contact avec le Broker, et membres du groupe client-fournisseur.

2.2.5 Raisons de ce choix

Comme notre but est de faciliter le développement d'application multi-agent en utilisant des technologies basées sur l'AOP, et plus généralement en utilisant des techniques favorisant une nette séparation des fonctionnalités de l'application, il nous est assez vite apparu que la conception encouragée par le modèle Aalaadin s'inscrivait dans la même ligne de pensée. En effet, le découpage d'une application multi-agent sous forme de groupes et de rôles permet de séparer les fonctions de l'application. De plus l'utilisation de groupes et de rôles permet de mieux spécifier les interaction entre les agents, en limitant les interactions au contexte en cours, celui du groupe Le choix d'un modèle multi-agent étant nécessaire, nous avons donc choisi de nous baser sur ce modèle, tout en nous réservant le droit de l'étendre, d'une façon montrée dans la partie consacrée à cette contribution.

Chapitre 3

Trois facettes de la relation AOP/SMA

Cette partie de mon rapport présente une des deux parties du travail accompli (hormis le travail bibliographique), l'autre étant la conception et l'implémentation du modèle multi-agent lié aux résultats exposés dans celle-ci. L'objet de cette partie est l'étude des liens entre AOP et SMAs en elle-même. Elle est organisée de la manière suivante : après une revue bibliographique du domaine (forcément restreinte du fait de sa nouveauté), la démarche de travail est exposée, puis ses résultats le sont.

3.1 État de l'art

Le travail bibliographique s'est scindé en l'étude de plusieurs domaines, qui sont les suivants :

- L'étude de l'AOP en général, présentée plus haut,
- L'étude d'une architecture SMA particulière, Aalaadin, elle aussi déjà évoquée,
- Celle des liens entre ces deux domaines, assez réduite du fait de la rareté des articles concernés, qui suit,
- Et enfin un travail plus poussé sur la notion de rôles, nécessité par l'importance que cette notion a prise au cours du temps dans nos travaux.

3.1.1 Sur les liens entre AOP et SMAs

Comme nous l'avons mentionné plus haut, peu de travaux ont eu lieu jusqu'à présent sur la relation ou l'utilisation de l'AOP dans le domaine des SMAs, et de plus avec une optique assez différente de la notre. Ces tra-

vaux représentent cependant une base de travail assez intéressante, bien que restreinte, et sont donc présentés ici. Ils viennent principalement de deux sources, qui sont une équipe Brésilienne travaillant sur l'utilisation de l'AOP principalement pour implémenter les caractéristiques des agents, et dans une moindre mesure d'une équipe Argentine utilisant des MOP dans l'implémentation de leurs agents.

Travaux de A. Zunino et A. Amandi

Les travaux de l'équipe argentine définissent un agent (sur le plan implémentatoire) comme un objet pourvu de un ou plusieurs méta-objets lui fournissant les propriétés nécessaires à en faire un agent [ZA00]. Cette approche permet une bonne séparation d'une partie des fonctionnalités de l'application, et facilite donc en partie le développement d'une application, comme cherchent à le montrer les auteurs.

Travaux de Garcia, da Silva, Lucena, Milidiú

L'équipe brésilienne a une approche plus intéressante à nos yeux, car elle emploie l'AOP pour arriver aux mêmes objectifs que Zunino et Amandi. Elle parvient donc à séparer des propriétés comme l'autonomie, l'apprentissage ..., du code métier de l'agent. Comme l'équipe argentine, elle se contente de l'implémentation de l'agent uniquement [Chr] [GdSLM]. Elle utilise pour cela plusieurs méta-objets sur plusieurs niveaux, ce qui donne une structure assez complexe.

3.1.2 De l'importance des rôles

Aalaadin utilise la notion de rôle pour représenter la fonctionnalité de ses agents. Cette notion de rôle a cependant plusieurs sens, au niveau de la conception objet et à celui de la conception agent. Alors que nous avions besoin de modifier la définition des rôles dans Aalaadin, il était plus que raisonnable de savoir ce qui a déjà été fait auparavant.

La notion de rôle à donc été introduite en programmation objet par Bent Bruun Kristensen [Kri96]. Selon lui, un rôle modélise une perspective particulière sur un objet, dans les interactions qu'il peut avoir avec les autres objets. Cette perspective lui permet d'acquérir de nouvelles méthodes et variables d'instance, ce qui lui permet d'acquérir un certain comportement en accord avec le rôle endossé. Les rôles peuvent se greffer sur des objets ou d'autres rôles, et peuvent aussi hériter les uns des autres. Ils permettent donc une bonne réutilisation du code. Kristensen a développé plus avant ses

idées dans d'autres articles [Kri97] [KO96]. Il s'est aussi interessé auparavant sur les activités transverses, qui se basent plus sur les interactions entre les objets pour accomplir une tâche que sur les objets eux-mêmes [Kri93]. Les définitions des rôles en multi-agents varient un peu, et sont pour l'instant représentées principalement dans Aalaadin [FG98]. La différence principale est qu'elles s'interessent plus à la conception qu'à l'implémentation. Ces définitions ont été rappelées plus haut (section 2.2.4), et ne nous conviennent pas tout à fait. Notons aussi une implémentation d'un système de rôles pour agents utilisant l'AOP pour définir les rôles [1]. Ces travaux sont ceux d'Elizabeth Kendall [Ken]. La solution proposée est à notre sens peu satisfaisante, les aspects servant surtout d'extensions de classes.

3.1.3 Relation entre rôles et aspects

Si Kendall utilise l'AOP pour implémenter un système de rôles, d'autres se sont également intéressés à la relation qu'il y avait entre les rôles au niveau objet et l'AOP. Ainsi D. Bardou [Bar], compare l'AOP à diverses techniques ayant recours à la notion de point de vue, comme les rôles et la programmation par sujets (Subject Oriented Programming, SOP). Il arrive à des conclusions assez intéressantes, en montrant que toutes ces technologies ont un certain nombre de points communs, en reliant certains concepts-clés de ces technologies aux points de jonctions des aspects, et en essayant de délimiter ce qui fait un code de base et un code «d'aspect» pour chacune de ces technologies.

[1] c'est donc l'approche opposée à la nôtre...

3.2 Trois facettes de la relation AOP/SMA

Comme nous l'avons mentionné plus haut, les travaux existant sur le sujet se portent surtout sur l'utilisation de l'AOP pour supporter l'implémentation des agents. Notre sujet était plus vaste, car son objet était une étude plus globale des liens entre l'AOP et les SMAs. Une partie du temps a donc été occupée à caractériser la relation entre ces deux domaines, et qui a abouti à la détection de trois aspects ou facettes de cette relation, qui sont exposées dans les parties suivantes.

Les trois facettes isolées sont les suivantes, présentées par ordre de difficulté croissante :

- L'utilisation de l'AOP dans l'implémentation d'une plate-forme SMA et de ses agents.
- En quoi l'AOP peut être utile dans le développement d'applications SMAs en elles-mêmes
- Et enfin, en considérant la relation comme bidirectionnelle, que peuvent apporter certaines théories et modèles SMAs qui soit de nature à enrichir les théories de l'AOP.

3.3 AOP dans l'infrastructure SMA

La première facette isolée est la plus évidente. C'est sur cette facette que des travaux ont été publiés, même si ils n'en couvrent pas l'intégralité. Son objectif est donc de trouver en quoi les techniques AOP seraient utiles dans l'implémentation d'une plate-forme SMA et de ses agents.

3.3.1 Bénéfices

Il va de soi que tous les bénéfices usuels de l'AOP se retrouvent dans cette étude. Une plate-forme SMA développée de la sorte sera donc d'autant plus flexible, maintenable, évolutive que l'on aura réussi à isoler un nombre important d'aspects de cette plate-forme. C'est donc l'objet principal de cette facette. Ces aspects sont valides à la fois au niveau des agents, et au niveau de la plate-forme en elle-même. Les possibilitées sont multiples, et se trouvent sous la forme des aspects suivants, qui ont été isolés au cours de ce stage.

3.3.2 Recherche d'accointances

L'intelligence artificielle distribuée est principalement caractérisée par le fait que les agents interagissent entre eux. La communication est donc une

composante très importante d'une application SMA, qui gagne à être la plus nettement séparée possible. Mais toute communication est précédée de la recherche du ou des agents avec lesquels cette communication doit se faire. Cette recherche a donc souvent lieu, et peut se reproduire si les références sur un agent deviennent invalide. Cette recherche peut envisager un certain nombre de cas, et employer plusieurs stratégies. C'est donc un bon candidat pour l'extraction sous la forme d'un aspect, car elle est à la fois dispersée dans le code de tous les agents, peut se trouver mélangée aux autres fonctionnalités (elle peut intervenir avant toute communication), et peut nécessiter une grande configurabilité.

Il reste donc a déterminer la meilleure façon de mettre en place cet aspect, c'est-à-dire à déterminer où se greffe le code de cet aspect sur le code de base. Cela revient à définir les points de jonction adéquats pour l'aspect. L'idéal serait pour l'agent de voir les références sur les autres agents de la même façon que des références sur des objets classiques, et donc d'abstraire complètement les stratégies de recherche et de vérification de ces références. Les points de jonction sur lesquels tisser cet aspect au code de base sont donc les accès aux variables d'instances qui sont des agents.

C'est à ce point précis qu'entre en jeu pour la première fois l'utilisation de l'architecture Groupe/Agent/Rôle . En effet, les communications dans cet architecture se font plutôt entre rôles qu'entre agents. Un agent aura tendance à référencer les services dont il aura besoin sous la forme de leur dénomination abstraite (le rôle endossé) plutôt que par un agent particulier. Cette manière de voir les choses se conjugue bien avec l'aspect décrit plus haut, car il suffira ainsi de nommer les références sur les agents par le rôle attendu pour disposer d'assez d'information pour trouver un agent satisfaisant. L'utilisation d'un nom de rôle est par ailleurs assez descriptive du point de vue du code de l'agent.

Voici un exemple de pseudo-code illustrant une stratégie de recherche d'accointances :

```
accesVariable(R).
1. si R est une référence sur un agent alors:
2.      [G := groupe auquel appartient R.
3.      si R est initialisée alors:
4.          [si R est valide alors:
5.              [retourner R.]
6.          sinon:
7.              [retourner rechercher(R,G).]]
8.      sinon:
```

```
9.         [retourner rechercher(R,G).]]
10. sinon: [retourner R]

rechercher(R,G)
1. R' := G.recherche(R).
2. si R' est valide alors:
3.     [retourner R'.]
4. sinon:
5.     [faire:
6.           [R' := trouver un agent capable d'endosser R.]
7.     tant que: [R' n'est pas valide.]
8.     retourner R.]
```

Les bénéfices de l'isolement de cet aspect sont sensibles : l'extraction du code de recherche et de vérification d'accointances hors du code de base permet de simplifier ce dernier, et de changer beaucoup plus facilement si besoin est ce dernier. L'apport majeur est de pouvoir traiter les références sur les agents de la même façon que celle sur les objets, en rendant implicites des traitements auparavant explicites et répétitifs.

3.3.3 Construction de messages

Venons en maintenant à un aspect lié à la communication, dont il est inutile de rappeler l'importance en IAD. Les aspects liés à la communication sont aussi de bons candidats à l'extraction du code de base, car ils sont, comme la recherche d'accointances, particulièrement répétitifs, ayant lieu à chaque interaction avec un autre agent. La communication étant un domaine vaste en elle-même, un seul aspect de communication serait probablement trop gros pour être utilisable, aussi est-il nécessaire de scinder cet aspect en plusieurs parties.

Le premier de ces aspects est celui de la construction de messages. En effet, les agents communiquent entre eux en utilisant des messages complexes, dans des formats variables. Il serait donc appréciable de pouvoir abstraire et paramétrer ce processus de construction de messages. Cet aspect est aussi en partie lié au processus de dispatch des messages au sein de l'agent, et à celui de la réception des messages.

Mise en œuvre

La démarche suit le même objectif que précédemment, qui est d'alléger la syntaxe des interactions entre les agents. Ainsi après avoir allégé la recherche

et la vérification des références sur les agents, nous cherchons à faire de même pour leurs envois de messages. L'idée est de calquer les envois de messages entre agents sur ceux entre les objets, l'aspect de construction de messages se chargeant de construire les messages effectifs, de remplir les informations implicites, puis d'envoyer le message à bon port.

Cet aspect prend en compte les processus d'envoi et de réception de messages (dans l'implémentation, au moyen de simples boîtes aux lettres, mais on peut penser à des communications utilisant l'environnement, voire des espaces de tuples ou un tableau noir). Après réception par le destinataire, l'aspect se chargera aussi du dispatch du message au sein de l'agent. Les méthodes pour effectuer ce dispatch relèvent malheureusement parfois de la simple énumération de possibilités (au moyen de mécanisme comme des «switchs» dans les langages à la C). L'aspect aura ici une toute autre façon de procéder, aidé en cela par le dynamisme du langage utilisé pour l'implémentation, qui est Smalltalk. Il se chargera en fait d'invoquer une méthode de l'agent receveur, correspondant au message envoyé au niveau objet par le premier agent. Les points de jonction avec le code de base sont eux aussi relativement faciles, puisqu'ils se réduisent à tous les messages envoyés à des objets qui sont des références sur les agents [2].

Bénéfices

L'utilisation des références directes par l'aspect précédent, alliée à cet aspect de construction abstraite de messages, permet de faciliter le travail du développeur, qui peut appliquer des traitements objets sur les agents, ce qui lui permet d'alléger considérablement son code, en évitant pour cet aspect un important mélange des préoccupations, les communications étant omniprésentes dans une application multi-agents . Le problème de la (elle aussi très importante) dispersion de ce code est aussi résolu, et permet donc, en évitant une multiplication de ces occurrences, une plus grande configurabilité de cette communication. Par exemple, il devient beaucoup plus simple de changer le médium de communication utilisé si cela est nécessaire, ou de lui ajouter des propriétés, comme le fait de tracer tout les messages envoyés [3].

Un autre bénéfice est que la recherche des messages à partir des méthodes implémentées par les agents nous permet d'utiliser beaucoup plus naturellement l'héritage comme moyen de spécialiser le comportement des agents. Un exemple consisterait en une application de gestion d'agenda, où

[2] puisque ceux-ci sont des agents, l'agent envoyeur n'a de toute façon pas moyen de communiquer directement avec les objets qui les représentent.

[3] cela est vu plus en détail par la suite, notamment dans la seconde facette.

un agent s'occuperait de centraliser les requêtes. Il s'occuperait de trouver un créneau pour des réunions nécessitant la présence de plusieurs agents. Cet agent pouvant éventuellement être impliqué dans des réunions, il pourra plus simplement hériter le comportement de base (celui d'un agent membre de la communauté) que le réimplémenter. Il en va de même pour les autres agents, qui seront vraisemblablement spécialisés.

3.3.4 Communications distantes

Un autre aspect de communication est celui prenant en charge les communications passant par le réseau, ou toutes les communications qui ne se font pas dans le même programme (par des pipes ou autres sous UNIX). Le but est de fournir une communication à distance totalement transparente du point de vue de l'utilisateur, que les messages soit transmis implicitement à travers le réseau si cela est nécessaire, mais que ce ne soit pas un souci pour l'utilisateur.

Une fois les deux autres aspects (recherche d'accointances et construction de messages) tissés, l'intégration de celui-ci devient assez facile, car les deux aspects précédents permettent de localiser beaucoup plus facilement le travail à faire :

- l'aspect «recherche d'accointances» permet de localiser la recherche des agents distants, au même endroit que la recherche des agents locaux (vraisemblablement en second recours).
- l'aspect «construction des messages» permet de localiser l'écriture du message dans le socket correspondant, et de localiser sa lecture sur la machine distante.

3.3.5 Autres aspects de communication

D'autres aspects peuvent aussi venir se greffer sur les aspects de communication listés plus haut. Ils sont pour l'instant moins développés que les autres, mais sont tout de même envisageables.

Ainsi on peut concevoir des aspects prenant en charge des langages de communication plus complexes, comme ACL ou KQML. Ces derniers sont des langages utilisés par les agents pour communiquer en fournissant des indications sur le sens des messages par le biais de performatifs, qui sont des mots-clés comme : affirmer, interroger, demander ... qui permettent de moduler le sens des messages échangés. On pourrait imaginer utiliser ces modes de communication sur des agents délibératifs, afin d'automatiser le choix de ce performatif en fonction des connaissances de l'agent sur le contenu du message à envoyer.

D'autres possibilités existent dans l'ajout de propriétés aux aspects existants, et ce par le biais d'aspects complémentaires, comme mentionné plus haut, à l'instar de la trace des messages. D'autres propriétés peuvent être utiles, comme le cryptage des messages échangés, permettant de sécuriser les conversations entre les agents.

3.3.6 Contrôle du flot d'exécution

La politique d'exécution de la plate-forme SMA est aussi une fonctionnalité qui est amenée à changer au cours du développement d'une application SMA. Ainsi l'application pourra être déployée au final sur plusieurs machines, avoir été développée sur un seul poste, et passer dans des phases de débogage, dans lesquelles la capacité de pauser l'exécution pour savoir ce qu'il se passe est primordiale. Cette variabilité nécessite de une bonne modularisation de cette propriété, ce qui fait d'elle un bon candidat pour un aspect.

On peut donc recenser plusieurs politiques d'exécution d'un SMA, dont les suivantes :

exécution sur plusieurs machines : tous les agents s'exécutent dans un parallélisme total, soit le plus équitable possible si plusieurs agents s'exécutent sur une machine.

exécution sur une machine : tous les agents s'exécutent sur la même machine, avec diverses politiques de répartition du temps machine. Cela peut aller d'une thread par agent, laissant ainsi au système la tâche de répartir le temps, à des politiques allouant strictement le même temps aux agents, ou d'autre leur faisant exécuter une action tour à tour.

exécution avec ressources limitées : des politiques particulières peuvent être employées pour des systèmes ayant un nombre de processus/threads limités, comme assigner plusieurs agents à un même fil de contrôle.

exécution en mode débogage : si le besoin s'en fait sentir, on doit être capable d'interrompre tout ou partie des agents du système afin d'en observer l'état, de le faire fonctionner en pas-a-pas, d'ajouter des points d'arrêts sous certaines conditions ...

Tout ces cas d'utilisation rendent le contrôle du flot d'exécution au sein de la plate-forme un élément sujet à variation, qu'il serait donc appréciable de pouvoir isoler d'une façon ou d'une autre.

3.3.7 Protocole des rôles

Cet aspect entre aussi dans le domaine de la communication, mais est spécifique aux architectures Groupe/Agent/Rôle. Dans cette architecture,

les conversations ne se font pas (conceptuellement) entre des agents, mais plutôt entre les rôles que portent ces agents dans un même groupe. Cela entraîne souvent une dispersion de la gestion du dialogue entre les agents endossant des rôles d'un même groupe, car c'est à eux de se préoccuper de l'enchaînement des «répliques» de la conversation. La gestion du dialogue est donc une fonctionnalité transverse aux rôles, qui pourrait être centralisée dans un aspect spécifique au groupe.

Reste donc à savoir comment modéliser cet aspect d'une façon satisfaisante au niveau du groupe. La manière la plus simple semble être d'employer un automate capable de décrire tout les états possibles de la conversation, et les transitions possibles entre ces états. Ainsi chaque envoi de message serait confronté à cet automate avant d'être validé, et chaque réception serait accompagnée des transitions possibles. Le rôle se baserait sur les connaissances de l'agent pour choisir la meilleure transition aux yeux de l'agent.

Il faut noter que nous sommes encore à l'ébauche de cet aspect, qu'il n'est pas encore, et loin de là, fixé à nos yeux. Cet aspect est de plus à la frontière de cette facette et de la suivante, concernant l'utilisation de l'AOP pour le développement SMA. L'infrastructure de cet aspect appartient à cette facette, mais son emploi, sous la forme d'une spécification des états de la conversation, est à définir par le développeur lui-même.

3.3.8 Gestion des ressources

Une autre fonctionnalité à la fois transverse aux diverses parties de l'agent, et possiblement paramétrable selon les conditions de déploiement de l'agent, est celle de la gestion des ressources. Plusieurs cas peuvent se présenter au développeur :

ressources illimitées : le cas le moins contraignant, l'agent dispose d'un temps «infini» et d'une mémoire «infinie» pour répondre à ses requêtes.

ressources limitées : dans le cadre de l'informatique embarquée, un agent peut disposer de ressources limitées, et donc avoir besoin de monitorer ses ressources, afin d'en éviter les dépenses abusives.

contraintes de temps réel : l'agent doit être capable de fournir une réponse dans un laps de temps donné. C'est une variante du cas précédent. On peut dans ce cas envisager de doter l'application d'algorithme anytime [4] plutôt que d'algorithmes classiques.

[4] algorithmes permettant de fournir une réponse à n'importe quel instant. La qualité de cette réponse croît en fonction du temps accordé à l'agent. Celui-ci occupe donc au mieux le temps qui lui est alloué.

3.4 AOP pour le développement SMA

Jusqu'alors, nous nous sommes surtout penchés sur l'utilisation de l'AOP pour implémenter une plate-forme SMA. Les aspects énumérés à présent ont déjà un impact sur le développement d'applications, mais ils ne concernent pas directement celui-ci. Il serait bon de pouvoir appliquer les mêmes techniques à l'application, en nous attendant aux mêmes bénéfices.

Ce sont des réflexions similaires qui nous ont amenés à employer l'architecture Groupe/Agent/Rôle, car l'effort de modularisation qu'elle encourage de part sa modélisation à base de rôles et groupes nous sont apparues comme une bonne base à la séparation des fonctionnalités, principe fondamental de l'AOP et plus généralement du génie logiciel.

Afin de mieux cerner ces possibilités, nous allons dans un premier temps comparer les groupes (ensemble de rôles) et les aspects afin de mettre à l'épreuve cette similarité. Dans un second temps, nous allons proposer et évaluer une façon d'«agentifier» les aspects non-fonctionels afin de les promouvoir comme «citoyen de première classe» d'un SMA.

3.4.1 Groupes vs Aspects : Similitudes

Nous nous intéressons dans cette partie aux points communs et aux divergences entre un aspect et un groupe, et ce dans les deux parties suivantes. Nous conclurons ensuite sur le bien-fondé de cette analogie, en ajoutant quelque remarques.

Transversalité

La notion la plus caractéristique que l'on peut dégager de la définition d'un aspect est la transversalité. La définition d'un aspect se ramène donc à toute portion de code accomplissant une fonctionnalité ou propriété donnée, mais qui est distribuée en plusieurs endroits dans le code global (objets, fonctions, ou méthodes).

Un groupe peut-il remplir cette définition ? Nous avons défini plus haut un groupe comme un ensemble d'agents occupant des rôles dans celui-ci. La responsabilité du groupe est donc de s'assurer que des agents compétents occupent les rôles qu'il définit, afin que les interactions entre les agents soient correctes. Ces interactions sont donc régies par les rôles qu'occupent les agents dans cette structure. Un groupe est donc composé d'un ensemble de rôles en interaction, visant à accomplir une tâche prédéfinie par cette coopération. On voit donc facilement que le code de cette tâche est distribué entre

tous les agents participant aux groupes. Cette tâche est donc une propriété transversale aux agents constituant ce groupe.

Weaving

Comme indiqué plus haut, le weaving ou tissage est le moyen de lier le code de base avec celui de l'application en AOP.

Comment se déroule ce processus de weaving dans le cadre d'une architecture groupe/agents/rôles ? Tout simplement lorsqu'un agent se joint à un groupe en endossant un rôle particulier. L'agent acquiert dans ce cas le rôle en question, et participe donc à la tâche à réaliser.

Configuration

Les aspects sont décrit la plupart du temps en deux parties : La première partie est générique et réutilisable, c'est le code de l'aspect à proprement parler, implémentant donc la propriété transverse désirée. La seconde partie est un "script de configuration", qui permet de définir comment l'aspect générique s'adapte à l'application spécifique que l'on est en train de coder. C'est cette partie qui définit les points de jonction avec lesquels l'aspect va se lier sur le code de base.

On peut utiliser les groupes de la même façon : l'ensemble des rôles possible dans un groupe représente la tâche à faire effectuer aux agents impliqués, et consiste donc en l'ensemble des interactions génériques nécessaires à sa réalisation. Le groupe en lui-même se charge de l'adaptation de la tâche à l'application, de sa configuration, en implémentant les stratégies de contrôle d'accès, de vérification, afin de permettre aux seuls agents correctement qualifiés de joindre le groupe .

3.4.2 Différences entre groupes et aspects

Weaving et Modification de code

Le weaving est sensiblement différent entre un rôle pour un agent, et un *advice* [5] sur un objet. Un advice a un effet sur tout les appels de la méthode, alors que le rôle ne modifie le comportement que dans un contexte particulier, c'est-à-dire seulement dans le groupe ou l'agent endosse ce rôle.

Il s'ensuit que les problèmes de cohabitation, d'ordonnancement des aspects sont moins présents dans une organisation, qui est un ensemble de

[5] code introduit par l'aspect avant, après ou à la place d'une méthode.

groupes, que dans un ensemble d'aspects. Dans le cas d'un ensemble d'aspects, le cas de l'ordonnancement des aspects «cryptage» et «logging» est souvent mentionné : on préfère que l'aspect «logging» soit exécuté avant l'aspect «cryptage», car on a généralement envie de disposer de traces lisibles... Soient deux groupes, dont l'un est loggé, et l'autre crypté qui cohabitent. Si un agent participe aux deux groupes, alors les messages reçus ou envoyés dans un de ces groupes seront cryptés, et ceux reçus/envoyés dans l'autre seront loggés. Il n'y a pas d'ambiguïté possible [6].

Dynamicité

Les agents peuvent venir dans un groupe pendant l'exécution du SMA, ce qui est moins courant pour les aspects (tout les systèmes d'AOP ne permettent pas le weaving dynamique par exemple). C'est de plus des opérations que les agents peuvent faire de leur propre initiative.

Aspects fonctionnels ou non

Il existe encore peu de littérature sur les aspects fonctionnels ou métiers. Pour l'instant, les recherches en AOP se sont plutôt orientées vers les aspects non-fonctionels, comme : logging, synchronisation, distribution, persistance, cryptage. Les aspects les mieux connus sont donc de ce type.

Si l'on considère les exemples d'architecture Groupe/Agent/Rôle les plus courants, on voit que les groupes semblent modéliser des aspects plus fonctionnels. Dans l'exemple «canonique» d'Aalaadin, qui est une organisation modélisant un réseau de contrat, on énumère trois type de groupes :

Groupe clients : constitué d'autant de clients que nécessaire, et d'un agent jouant le rôle de «Broker», se chargeant de mettre en relation les clients et les fournisseurs.

Groupe fournisseurs : ce groupe comprend tout les fournisseurs du SMA, ainsi qu'un agent jouant le rôle de Broker, qui doit jouer également ce rôle dans le groupe des clients.

Groupe client-fournisseur : ce groupe est instancié à chaque fois qu'un client fait une requête au Broker que celui-ci complète. Le fournisseur le plus avantageux est mis ainsi en relation avec le client par le biais de ce groupe restreint ne comprenant que deux rôles.

On voit que l'on n'a pas affaire au même type d'aspect que précédemment. Aalaadin se focalise en effet sur des fonctionnalités plus «haut niveau»

[6] Il reste à voir quoi faire lorsque cette ambiguïté est désirée. Cela est abordé plus loin avec l'utilisation des aspects sur les groupes, et leur agentification.

que les aspects utilisés jusqu'à présent. Cependant, il existe aussi d'autres aspects non fonctionnels pour les groupes, comme l'autre exemple canonique d'Aalaadin, qui est la migration automatique des agents d'un site vers un autre[7].

3.4.3 Différences entre programmation objets et programmation agents pour l'intégration de l'AOP

Cette partie se concentre sur les dis similarités entre les applications de l'AOP aux objets et aux agents, qui est aussi une source de différences entre groupes et aspects.

Configuration des aspects pour les agents

La différence principale entre un objet et un agent est qu'un objet est passif, alors qu'un agent est actif : il prendra ses décisions de façon autonome, en fonction de son environnement et de ses capteurs.

Ainsi c'est la plupart du temps de sa propre initiative, qu'un agent cherchera à entrer dans un groupe, et donc à se «weaver» avec un aspect particulier, si l'on considère les groupes comme des aspects. Le rôle de configuration du groupe n'est donc pas de désigner tel ou tel agent comme faisant partie de lui-même, mais d'autoriser ou non un agent à jouer un rôle en son sein. Il «réagit» donc aux requêtes qui lui sont faites [8].

Introspection des agents

Les agents savent dans quels groupes/aspects ils sont et sous quel rôle, ils peuvent donc raisonner sur ces faits si ils en ont la possibilité. De plus, il peuvent demander à quitter ou joindre des groupes, ou demander à d'autres agents de quitter ou joindre des rôles. L'action de joindre un groupe est même une action assez courante pour un agent. Il me semble que peu d'objets peuvent décider de se faire appliquer des aspects, bien que l'utilisation de procédés de monitoring puisse remplacer cela (seulement dans les frameworks comme MetaclassTalk ou le tissage peut être dynamique).

[7] on peut aussi le voir comme un aspect fonctionnel de l'agent responsable de la migration, qui fait partie de la partie non-fonctionnelle de la plate-forme MADKit.

[8] on verra plus tard que l'on peut donc en faire un rôle réactif pour un agent.

Fréquence d'emploi

Les agents semblent communiquer moins souvent que les objets. Un agent est une entité assez complexe, qui peut donc faire des traitements complexes avant ou entre ses communications externes (par exemple pour un agent délibératif planifiant ses actions). Une grande partie des messages seront donc internes à l'agent, et donc pas forcément interceptés par des aspects ou des méta-objets.

3.4.4 Conclusions sur l'analogie Groupe/Aspect

Les groupes possèdent une bonne partie des propriétés des aspects, dont la plus importante, qui est la transversalité. Ils respectent aussi la dualité entre partie générique de l'aspect (l'ensemble des rôles que peut contenir le groupe), et partie configurable (que l'on peut ajouter dans l'implémentation du groupe, pour ajouter des propriétés supplémentaires). Ils ont aussi une notion de "weaving", qui est accomplie quand un agent endosse un rôle dans un groupe. Toutes ces propriétés font donc des groupes multi-agents de bons candidats pour remplir le rôle d'aspect dans le développement d'une application SMA.

Il existe cependant des nuances entre ces deux notions, qui sont dues à deux choses, à savoir le fait d'utiliser des groupes en eux-mêmes, et celui d'utiliser des agents en lieu et place des objets.

Ainsi, le tissage peut s'effectuer à l'initiative de l'agent, et n'est donc pas obligatoirement passif comme pour un objet. De plus, la modification de comportement n'est pas globale, mais la plupart du temps locale au groupe dans lequel l'agent joue un rôle. Il n'est non plus pas rare de voir certain agents joindre ou quitter des rôles, ce qui est moins courant pour les objets, et qui est de plus renforcé par la proactivité des agents.

Enfin, les domaines d'application des aspects et des groupes ne sont à première vue pas exactement les mêmes : les aspects modélisent plus rarement (à l'heure actuelle) des propriétés fonctionnelles, et certains aspects ne sont pas évidents à modéliser dans le domaine des groupes, surtout ceux dont la portée est globale (dépassant un simple groupe), comme le logging.

Néanmoins, si aspects et groupes diffèrent sur certains points, ils se recoupent sur les points les plus importants, et notamment la transversalité, ce qui permet à mon sens de qualifier les groupes d'aspects.

3.4.5 Agentification des aspects

Bien-fondé et utilité

Nous avons évoqué dans la section «Weaving et modification de code», que les groupes et les rôles étaient la plupart du temps utilisés d'une façon qui permettait de diminuer les risques de conflits entre les divers groupes, comparativement aux aspects.

L'exemple canonique du problème de composition des aspects est le suivant : dans quel ordre tisser les aspects de «logging» et de «cryptage» sur une application ? En effet, leur ordre modifie leurs actions : dans un des cas, les sorties loggées seront cryptées, dans l'autre non. Si les deux aspects interviennent sur les mêmes méthodes, il faut choisir lequel s'applique en premier : si le cryptage s'applique en premier, le log passant après affichera des informations cryptées. Si à l'inverse le log est le premier aspect à s'exécuter, alors les traces produites seront en clair.

L'utilisation de groupe prévient ce problème, car les groupes marquent le contexte dans lesquels ils sont employés. Un agent dans un groupe crypté avec un rôle A, et dans un groupe loggé avec un rôle B, n'aura pas cette ambiguïté : les messages adressé aux rôle A seront cryptés, et ceux du rôle B seront loggés dans un fichier.

Cependant, que se passe-t-il si l'on souhaite tout de même utiliser ces deux propriétés sur un seul rôle, voire tout simplement ajouter une propriété à un groupe déjà existant ?

Dans ce cas, l'utilisation des groupes ne nous permet pas de répondre à cette question. Il nous est donc nécessaire d'ajouter les deux propriétés suivantes à notre modèle (décrites dans les deux parties suivantes) :

- ajout de méta-rôles
- utilisation des groupes comme support des aspects

Ajout de méta-rôles

Un méta-rôle est un rôle prenant place au-dessus d'un autre rôle, afin d'en paramétrer le comportement, de la même manière qu'un méta-objet peut contrôler un objet du niveau de base. Plus précisément, un méta-rôle est un rôle que peut endosser un agent pour joindre un groupe. Ce rôle est cependant paramétré par un ensemble de rôles de l'agent sur lesquels il s'applique, et sur lequel il a une action. A l'instar d'un méta-objet, son comportement est déterminé par un protocole particulier, comportant principalement des méthodes activées à chaque envoi ou réception de message par un rôle

des rôles de base[9]points d'ancrage, permettant d'exécuter du code arbitraire quand on les accède[10].

Par exemple, un méta-rôle de log pourrait avoir comme implémentation la suivante :

```
receive: aMessage from: aBaseRole sender: aRole
    log write: 'received message : ', aMessage, ' from : ', aBaseRole.

send: aMessage from: aBaseRole to: aRole
    log write: aBaseRole, ' sends : ' , aMessage, ' to : ', aRole.
```

Il reste à continuer l'implémentation de l'aspect de logging au sein de la plate-forme SMA, ce qui pourrait se faire par la création d'un groupe de log ayant les rôles suivant :

- ayant autant de rôles «Logged» que souhaitable, transmettant la trace des messages qu'ils reçoivent sur leur rôles de base.
- un unique rôle «Log», se chargeant du stockage ou de l'affichage de la trace.

De cette façon, nous avons réussi à créer un groupe représentant un aspect non-fonctionnel, qui est la fonctionnalité de logging des messages échangés dans certains groupes de l'application. Nous pourrions faire de même pour un aspect de cryptage par exemple. Il reste donc à voir comment effectuer la jonction avec le code, ce qui est l'objet de la partie suivante.

Utilisation des groupes comme support des aspects

Le moyen le plus simple de tisser un aspect sous forme de groupe est d'instaurer des «propriétés de groupe». Celles-ci sont des propriétés non-fonctionnelles comme la trace ou le cryptage, qui sont automatiquement liées aux rôles endossés par les agents. Un agent entrant dans un tel groupe entrera en fait automatiquement dans deux groupes, avec deux rôles, l'un faisant office de méta-rôle pour l'autre.

Ceci permet donc d'ajouter des propriété non-fonctionnelles aux groupes, ce qui permet de mieux partitionner les propriétés ajoutées à l'application. On peut ainsi plus facilement appliquer le cryptage sélectivement à une partie de l'application, et même à des parties seulement d'un agent (seulement les rôles

[9]On peut éventuellement enrichir ce protocole par l'ajout de *hooks* .

[10]sur ces accès de variable d'instance, ... si cela est nécessaire. On peut aussi enrichir la définition des points de jonctions, qui sont pour l'instant limités à l'intégralité des méthodes, mais on peut aussi vouloir exercer un contrôle de grain plus fin.

concernés, par exemple ceux du groupe client/fournisseur dans l'exemple du Broker).

Les agents peuvent aussi être ajoutés individuellement à ces groupes, voire de leur propre chef, si ils sont capables de raisonner sur ces faits.

3.4.6 Conclusion : complémentarité des groupes et des aspects

À notre sens, les deux concepts de groupes et d'aspects sont complémentaires. Comme nous l'avons vu plus haut, les aspects permettent de faciliter l'implémentation d'une plate-forme SMA, en allégeant le travail de développement de certaines tâches, et en permettant à l'application d'être plus paramétrable. Parallèlement à cela, les aspects sont remplacés au niveau applicatif par les groupes, qui sont mieux à même de modéliser une application SMA, en utilisant les possibilités offertes par Aalaadin en termes de groupes et de rôles. Les groupes ne sont par contre pas capable d'exprimer les mêmes choses que les aspects, ils semblent en effet mieux adaptés à l'expression de préoccupations fonctionnelles. Cependant ils peuvent, en enrichissant le modèle Groupe/Agent/Rôle par le moyen de méta-rôles, retrouver la capacité d'exprimer des besoins non-fonctionnels, comme le log ou le cryptage. L'introduction de ce concept permet par ailleurs d'homogénéiser le développement de l'application multi-agents, car elle permet d'exprimer les aspects non-fonctionnels sous la forme de groupes et de rôles. Nous avons donc fourni un moyen «d'agentifier» les aspects.

Il faut noter que ces raisonnements sont aussi en cours de transposition au niveau des organisations, c'est-à-dire au niveau supérieur à celui des groupes. On trouve en effet à ce niveau des éléments transversaux aux groupes et aux rôles. Un exemple est celui de certaines contraintes sur les rôles transverses aux groupes (dans l'exemple cité plus haut du Broker, il y a une contrainte de ce type qui stipule que l'agent portant le rôle de Broker dans le groupe Client doit porter ce même rôle dans le groupe Fournisseur). Ces réflexions sont en cours d'élaboration et de validation.

3.5 SMA pour l'AOP

Pour l'instant, nous avons surtout abordé notre étude de la relation entre AOP et SMAs dans un seul sens : celui des bénéfices que peut apporter l'utilisation de l'AOP dans le développement des plates-formes et des applications SMA. Cependant, à l'issue de l'étude des facettes précédentes, nous nous sommes aperçus que les modèles de groupes et de rôles pouvaient modéliser avec plus d'aisance certaines fonctionnalités que les aspects classiques. Aussi nous est-il apparu opportun d'essayer de faire profiter les théories AOP de ces bénéfices, ce qui aurait donc pour conséquence un enrichissement mutuel de ces deux théories.

Il conviendrait donc d'étudier les causes de la capacité du modèle Groupe/Agent/Rôle, à modéliser les préoccupations fonctionnelles des applications. Il semble que ce soit surtout les notions de groupes et de rôles qui sont au coeur de ce fait, celle d'agent n'étant que secondaire. Pour appuyer ce postulat, on peut remarquer que la modélisation à base de rôles existe en-dehors de la communauté SMA, étant au départ une méthodologie mise au point dans la communauté objet.

La notion de groupe est elle aussi importante, car elle forme le «liant» entre les rôles, permettant de fait la communication entre les agents. Il faut donc trouver un moyen de passer du concept d'agent à celui d'objet, pour se ramener à un contexte plus familier, et plus applicable en général.

Cependant je n'ai pas pu me pencher plus sur ces concepts. L'exploration de cette facette de la relation AOP / SMA tient donc plus du domaine des pistes de recherche que des résultats effectifs. La marche à suivre serait vraisemblablement de partir de l'exploration de cette facette, de généraliser les résultats obtenus, pour enfin aboutir à une caractérisation des aspects fonctionnels, encore mal connus dans le domaine de l'AOP.

Chapitre 4

Notre adaptation du modèle Aalaadin

Durant l'exploration des trois facettes, nous nous sommes aperçus que le modèle SMA sur lequel nous nous sommes basés, Aalaadin, n'était pas exactement ce dont nous avions besoin. Il constituait certes un bon point de départ, mais ne répondait pas à toutes nos attentes. Nous avons donc défini un modèle plus proche de ces dernières, et donc plus à même de nous aider dans l'exploration des trois facettes. L'objectif de cette partie étant de décrire ce modèle, nous allons donc commencer par expliquer ce qui constitue à notre sens les faiblesses d'Aalaadin, puis y apporter nos solutions, pour enfin parler de l'état de son implémentation, et la confronter à celle de référence, MADKit [1], qui est l'implémentation d'Aalaadin de Ferber et al [GF97].

4.1 Limitations du modèle Aalaadin

Les principales limitations de ce modèle auxquelles nous avons été confronté sont les suivantes :
- Aalaadin est un modèle parfois trop abstrait, notamment dans la description de ses concepts de rôles.
- L'utilisation des groupes comme aspects ne permet pas tel quel d'exprimer tout ce qui est désirable.

4.1.1 Un modèle «trop abstrait»

Le modèle Aalaadin est trop abstrait à notre sens, du fait qu'il ne soit que conceptuel. Dans Aalaadin, la seule information que nous avons sur les rôles

[1] disponible sur `http://www.madkit.org/`.

est qu'un agent peut en endosser autant qu'il veut, et ce, dans n'importe quel nombre de groupes. On sait aussi que les agents communiquent au travers de leurs rôles, et se basent uniquement sur leur rôles pour trouver les services qu'ils désirent. Cependant, notre travail de recherche nécessite de se baser un tant soit peu sur une implémentation, la description d'aspect purement formels ne suffisant parfois pas.

Nous avons donc aussi jeté un oeil sur l'implémentation de référence d'Aalaadin, appelée MADKit [2]. Elle-même ne fournit pas vraiment de moyen d'implémenter les rôles. Aalaadin et son implémentation préfèrent «laisser le champ libre au développeur» en la matière, le laissant implémenter ses agents à sa guise. Cela a ses avantages (plus grande liberté pour le développeur, possibilité d'utiliser différents langages), mais aussi ses inconvénients : le développeur ne dispose pas de moyen d'implémenter efficacement et proprement ses rôles, ce qui débouche souvent sur des agents monolithiques et difficilement réutilisable.

L'approche que nous envisageons, et qui est décrite plus loin, est l'introduction d'un système de rôles qui aura aussi cours pendant le développement, et permettra de cette manière aux aspects de se baser sur un support concret.

4.1.2 Incapacité à décrire certains aspects sous forme de groupes

Comme nous l'avons décrit dans la facette 2, l'utilisation de groupes comme des aspects au niveau de l'application fonctionne bien au niveau des aspects dits «fonctionnels», mais ne fonctionne pas dans le cas des aspects «non-fonctionnels», qui sont en général plus intrusifs que les aspects fonctionnels. En effet, les aspects comme le log ou le cryptage, sont vraiment des aspects qui doivent interférer avec le code de base pour y intégrer un comportement particulier. Au contraire, les groupes possèdent un système de contexte qui limite les possibilités de conflits ou d'interférences entre les groupes ou au sein d'un agent : un message envoyé à un agent ne sera interprété que par le rôle qui l'a reçu, qui est le rôle que l'agent endosse dans le groupe d'où le message est originaire. Ce comportement est souhaitable dans la plupart des cas, mais ne l'est pas dans les cas précédemment mentionnés. Il faut bien entendu noter que ces cas n'ont pas été bien sûr prévus par Aalaadin au départ, ce qui est compréhensible.

Là encore, ils nous faut étendre le modèle d'Aalaadin pour permettre ces comportements intrusifs quand ils sont désirés, et faciliter par ce biais leur mise en œuvre.

[2]Pour : Multi-Agent Development Kit.

4.2 Extensions apportées au modèle

Après avoir montré les limitations nous concernant dans Aalaadin, nous montrons comment modifier le modèle pour les outrepasser, en reprenant les points dans l'ordre précédent.

4.2.1 Concrétisation par la réification

Notre modèle s'efforce d'être plus concret, en fournissant des bases permettant d'implémenter plus facilement les concepts évoqués de groupes et de rôles.

Exemple de MADKit

Si l'on regarde les exemples fournis dans MADKit, comme les exemples de ping-pong ou de Contract-Net, on s'aperçoit très vite que ces exemples ne fournissent sur le plan de l'implémentation aucun support au développeur pour l'aider dans l'implémentation des groupes et des rôles. Les exemples codent les agents de sorte à ce qu'ils se conforment à l'interface spécifiée, en termes de messages échangés, mais ne permettent pas de spécifier cette interface au niveau de l'implémentation, ce qui peut entraîner des risques d'erreur. De plus, la réutilisabilité du code est compromise par la façon dont les agents sont implémentés : une méthode appelée `handleMessage:` fait tout le travail, pour tout les rôles. Cette méthode se base de plus sur le type du message pour choisir comment le traiter, ce dispatch manuel alourdissant le code. Concernant la manipulation de groupes et de rôles dans MADKit, elle est réduite à sa plus simple expression : les seules références sur ces données utilisables par le programmeur sont des chaînes de caractères, permettant seulement de rechercher des agents jouant les rôles demandés.

Réification des rôles dans l'implémentation

Quel est l'intérêt de réifier les rôles ? Nous avons dit plus haut qu'un groupe, qui est notamment constitué d'un ensemble de rôles, peut être considéré comme un aspect. Pour que cette affirmation ne soit pas vérifiée seulement au niveau conceptuel mais aussi au niveau de l'implémentation, il faut trouver un moyen de définir ces groupes comme des entités manipulables et réutilisable. Il faut donc réussir à séparer le code des groupes de celui des agents de base. Le fait de réifier les rôles nous permet d'arriver à une partie de ce résultat.

Le modèle retenu est donc le suivant : l'agent en lui-même n'a pas de comportement particulier. Son «rôle» est de fournir un support pour les rôles

réels. Il se chargera donc des infrastructures de communication, de l'exécution du comportement, et de fournir des fonctions de convenance. Le comportement de l'agent sera en fait contenu dans les divers rôles qu'il pourra endosser.

Les rôles comprendront donc les méthodes à appeler à la réception des messages, selon le format employé par l'aspect de construction et de dispatch des messages, ce qui permet de résoudre une fois pour toutes les problèmes de dispatch manuel. Les rôles ont aussi une référence sur le groupe dans lequel ils sont endossés, afin de s'y référer en cas de besoin. c'est le cas notamment pour l'aspect de recherche d'accointances, qui emploie cette référence pour chercher dans le groupe un agent ayant le rôle demandé. Concernant les accointances, celles-ci sont des simples variables d'instance du rôle, portant le nom des rôles demandés, comme Broker, ...

Les conséquences de ces choix sont les suivantes :

- Les rôles sont des objets réutilisables, endossable par les agents.
- L'agent en lui-même est une «coquille vide», capable d'avoir un comportement en endossant un ou plusieurs rôles dans un ou plusieurs groupes.
- Les rôles ont une hiérarchie de rôles, qui permet de spécialiser leur comportement.
- La distribution des messages au sein de l'agent est effectuée en fonction du groupe receveur.
- L'approche retenue permet qu'un agent puisse endosser le même rôle dans plusieurs groupes différents [3].
- Les rôles forment la base des aspects d'implémentation que sont la recherche d'accointances et la construction de messages, dont nous avons parlé en décrivant la première facette.
- Ils sont une brique dans le traitement des groupes comme des aspects, exposé dans la seconde facette, car ils permettent de regrouper le code d'un aspect sous la forme d'un ensemble de rôles.

Réification des groupes et des organisations

L'étape suivante, venant quasi-naturellement une fois les rôles réifiés, est l'étape de réification des groupes et des organisations dans le modèle Aaladin. Cette étape permet de manipuler réellement les groupes, et donc de les transformer en entités manipulables aussi facilement que des aspects. Par souci d'homogénéité, et comme nous subodorons que les structures organisationnelles peuvent être des aspects de niveau supérieur, ces dernières ont

[3] une autre approche envisagée précédemment, à base de mixins, ne le permettait pas.

subi le même traitement.

Concrètement, cela se traduit par la création de classes Group et Organisation, qui ont de plus les moyens de contrôler l'accès des agents aux rôles internes. Ceci est accompli par des spécification des rôles et de leur cardinalités possible au sein des groupes. On peut imaginer aussi étendre les organisations afin qu'elles implémentent réellement les contraintes du type de celle présentes dans l'exemple de l'organisation du Contract-Net.

Enfin, par souci d'homogénéité, des rôles ont été crées pour que ce soit les agent qui puissent gérer les groupes et les organisations, et non pas un noyau. Cela a donc débouché sur la création de rôles GroupManager et OrganisationManager, endossables par des agents si ils veulent créer un nouveau groupe. Ces agents peuvent donc décider d'admettre ou non d'autres agents dans le groupe, ou plus simplement déléguer ce travail à l'objet Group ou Organisation. Cette approche permet notamment d'accéder aux listes d'agents, de groupes, et d'organisations disponibles en visitant des groupes automatiquement maintenu, qui sont les groupes Groups et Organisations, contenant les agents responsables de la gestion de groupes et d'organisations. Lorsqu'un agent quitte ou endosse un de ces rôles, il quitte ou endosse automatiquement un rôle dans un de ces groupes.

Une autre différence avec le modèle Aalaadin, est que les cardinalités des rôles ne sont pas liées aux rôles en eux-mêmes, mais aux groupes. Cela permet une meilleure réutilisabilité des rôles en déportant une partie des contraintes sur les groupes.

Vers une réification des protocoles des rôles ?

Une adjonction en cours d'évaluation est celle qui permettrait de supporter proprement l'aspect de gestion des protocoles de conversation des rôles, mais cela implique des changements assez profonds du modèle, car cela déplace une partie importante des responsabilités des rôles vers les groupes. Cela a par contre l'intérêt de rendre les rôles encore plus réutilisables, en séparant les compétences de leur enchaînement, qui serait dévolu aux groupes, pourvus d'un automate listant les états possibles de la conversation entre plusieurs agents. Mais cet ajout est plus dans la rubrique «future works» que dans celle des travaux présents.

4.2.2 Instauration de méta-rôles

Un deuxième point important est celui de l'ajout de méta-rôles pour permettre, à la façon des méta-objets, de pouvoir modifier le comportement d'un

rôle existant. Ce concept est à la base de notre implémentation des aspects non-fonctionnels implémentés sous forme de groupes.

Les agents peuvent donc endosser des rôles normaux, et aussi des méta-rôles, interférant sur le comportement des rôles, et donc le comportement est spécifié par un protocole particulier et précis. Ce protocole prendrait en charge surtout l'envoi et la réception de messages par un rôle, et permettrait de placer du code avant, après ou à la place de la réception du message par le rôle.

Cela aurait donc l'avantage de permettre de spécifier les deux types d'aspects, intrusifs ou non, et de faire ceci de façon homogène (voir l'exemple de l'implémentation de l'aspect de log sous forme d'un groupe comprenant des rôles et des méta-rôles) dans la couche agent, c'est-à-dire sans avoir recours à des aspects normaux quand on pourrait s'en passer. De plus, les aspects seraient visibles pour les agents, et ceux-ci pourrait donc les joindre quand ils le veulent, si ils peuvent utiliser ces connaissances.

4.3 État de l'implémentation

L'implémentation de ce modèle est en cours. Nous décrivons tout d'abord son fonctionnement général, puis nous nous attarderons sur des détails d'implémentation de certains aspects, avant de faire un bref comparatif avec MADKit.

4.3.1 Démarche de développement

Une première implémentation de l'architecture Groupe/Agent/Rôle [Rob] a été faite au tout début du stage, pour bien visualiser les spécificités de cette architecture, et pour avoir une connaissance plus approfondie de son fonctionnement. Cette connaissance de l'implémentation nous permet d'être plus à même de détecter les aspects extractibles de cette plate-forme. Une implémentation basique, avec un exemple multi-agent de calcul de la factorielle, a pris environ une quinzaine de jours. Dans la même optique, un fois le travail bibliographique plus avancé et les directions plus éclaircies, je me suis attaqué à la conception d'un exemple d'utilisation plus complexe (l'exemple de gestion de calendrier par agents mentionné plus haut), afin de mieux cerner les enjeux de l'utilisation de l'AOP dans le développement d'une application multi-agent. Enfin, une fois les concepts en eux-mêmes plus développés, je me suis attaqué à l'implémentation du prototype «final», en le réécrivant du début pour partir sur des bases saines. C'est donc cette implémentation que je vais décrire à présent.

4.3.2 Présentation de l'implémentation

La première implémentation reprenait le modèle Aalaadin sans modification majeures, mais la plate-forme actuelle possède une bonne partie des caractéristiques à la fois d'Aalaadin et de son modèle dérivé présenté dans le chapitre 4. L'ordre d'implémentation des diverses fonctionnalités a été décidé en fonction de l'exemple implémenté (le calendrier). Les fonctionnalités jugées les plus handicapantes [4] ont été implémentées en premier. L'ordre retenu a donc été le suivant :

1. rôles réifiés et réutilisables, endossables dynamiquement
2. dispatch des méthodes des rôles en fonction du groupe au sein de l'agent
3. groupes réifiés
4. rôles de gestion des groupes
5. organisations
6. rôles de gestion des organisations
7. autres groupes et rôles «système» (rôle noyau, rôle Transcript, groupes Transcript, Groups, Organisations
8. aspect de recherche d'accointances
9. aspect de construction de message
10. aspect de politique d'exécution

Les autres aspects, ainsi que les méta-rôles sont en cours d'analyse ou d'implémentation. Le développement a été le plus suivi possible par le biais de tests unitaires. Les fonctionnalités numérotées ci-dessus sont soit terminées pour les premières, soit en phase de l'être pour les dernières.

4.3.3 Utilisation du framework MetaclassTalk

MetaclassTalk a été utilisé pour implémenter une partie des concepts de la plate-forme, qui sont les aspects d'implémentation de celle-ci. C'est un système très puissant, qui permet de modifier totalement la sémantique du langage d'origine (Smalltalk) [5]. Cette puissance en fait cependant un outil assez complexe à prendre en main, demandant de bonnes connaissances de la réflexivité en général, et de son application à Smalltalk en particulier. J'ai donc eu droit à quelques surprises par moment... De plus cet outil est

[4] ce sont les fonctionnalités qui ont le plus manqué lors de l'implémentation de l'exemple en utilisant la première version de la plate-forme.

[5] Il est possible en utilisant ce framework de définir de nouveaux mécanismes d'héritage par exemple.

encore à l'état de prototype, et donc ne comprend pas encore tous les outils qui pourraient faciliter au maximum son utilisation. Néanmoins, deux aspects importants ont pu être implémentés grâce à MetaclassTalk, ce sont les aspects de recherche d'accointances et de construction de messages.

4.3.4 Aspect de communication à distance

Les détails d'implémentation de mécanismes de sérialisation et de désérialisation utilisés dans les programmes de communication à distance sont assez complexes, et ne sont pas expliqués ici. L'implémentation étant bien entendu un prototype, elle ne comprend pas de code s'occupant des communications distantes. On peut cependant signaler que parallèlement à ce travail, un aspect de communication distante à été mis en place à l'École des Mines de Douai, par un autre stagiaire de DEA, Rabih Nassrallah [Nas03]. Il pourrait donc être utilisé pour servir dans cette implémentation.

4.3.5 Comparatif avec MADKit

Si la première implémentation était très proche dans sa réalisation de MADKit, utilisant la même démarche car dépourvue du moindre aspect, il en est tout autrement pour la seconde.

L'utilisation de rôles réifiés a une influence majeure sur le développement, forçant le développeur à séparer aux mieux les fonctionnalités de l'application, et à y réfléchir jusque dans l'implémentation, ce qui a des conséquences importantes sur son design.

De même, les aspects de recherche d'accointances et de construction de messages permettent d'alléger considérablement le code, et de séparer nettement les politiques de construction et d'acheminement de messages, celles-ci devenant dès lors vraiment paramétrables. La longueur et la lisibilité du code source s'en ressentent également.

Il reste que bien évidemment, MADKit est beaucoup plus élaboré [6] qu'un simple prototype en cours d'implémentation. Ne vous attendez donc pas à des graphismes élaborés (cela pourrait peut-être constituer un aspect futur ?), mais plutôt à des sorties texte assez brutes .

[6] MADKit est en développement depuis plusieurs années, et va bientôt passer en version 4...

Chapitre 5

Conclusion et travaux futurs

Arrivé au terme de ce stage, je vais conclure en mentionnant les travaux futurs, qui sont surtout les pistes de recherche exploitables par la suite. Puis j'évoquerais les spécificités de ce stage dues à la collaboration entre deux équipes de recherche.

5.1 Résumé des travaux

Ce stage avait au départ un sujet que l'on peut qualifier de vaste. Il consistait à étudier les diverses relations qu'il pouvait y avoir entre la programmation par aspects et les systèmes multi-agents. Partant de là, nous avons tout d'abord caractérisé trois aspects de cette relation, qui sont sont :

- L'utilisation des technologies AOP pour l'implémentation de SMAs.
- Comment adapter l'AOP afin qu'il soit utilisable pour le développement d'application SMAs.
- En quoi les modèles SMAs peuvent contribuer à enrichir l'AOP.

Nous nous sommes ensuite attaqués à l'exploration de ces trois facettes, en insistant surtout sur les deux premières, par manque de temps. Les deux facettes explorées peuvent d'ailleurs faire l'objet de publications.

Parallèlement à ce travail, nous avons modifié le modèle Aalaadin pour qu'il puissent supporter nos théories, ce qui a abouti à un modèle SMA original. Ses principales différences sont une concrétisation des concepts de groupe, de rôle, d'organisation sous une forme réifiée, permettant de supporter plus facilement la seconde facette, ainsi que l'instauration d'un système de méta-rôles permettant de mettre en place des comportements plus intrusifs pour certains rôles si nécessaire. L'implémentation de ce modèle est en cours, utilisant le framework MetaclassTalk pour la prise en charge de l'AOP et des méta-objets.

5.2 Travaux concernant la suite du stage

Un stage est toujours trop court, il reste donc un certain nombre de choses à faire. Ainsi l'implémentation n'est pas encore achevée, il reste donc un peu de travail de ce côté. Plus importants sont les projets de publications envisagés sur le travail effectué. Ils sont au nombre de deux, et sont basées sur les deux facettes qui ont été les plus explorées :

- Une publication sur l'implémentation de la plate-forme en général, et sur les aspects développés en particulier, montrant les différences avec une implémentation «classique», comme MADKit, sur un exemple.
- Une publication sur l'analogie qui a été faite entre les groupes et les aspects, afin de diffuser le développement par aspect dans la communauté SMA. Celle-ci contiendrait les comparaisons faites, ainsi que les modifications à apporter au modèle Groupe/Agent/Rôle pour qu'il puisse soutenir la comparaison : ajout de méta-rôles, réification des rôles ...

5.3 Autres pistes de recherches

Ce stage a permis de délimiter 3 facettes de la relation entre l'AOP et les SMAs . Étant un sujet de stage assez vaste et ouvert, il a naturellement donné naissance à plusieurs de pistes de recherche. Outre les projets de publications sur le travail accompli, les pistes de recherche suivent plus ou moins les trois facettes :

5.3.1 Continuer l'exploration et l'implémentation d'aspects d'infrastructures

Les aspects exposés sont les premières briques vers une implémentation plus modulaire et robuste des plates-formes SMA. Il reste donc d'autre briques à isoler, puis à définir. Celles me venant à l'esprit actuellement sont les suivantes :

Aspect embarqué

Un aspect embarqué prend en compte les diverses nécessités entrant en jeu dans la conception des agents embarqués. Ceux-ci on en effet des caractéristiques souvent particulières :

- Ils ont des ressources limitées, dues à la faible taille et à la faible puissance de l'électronique embarquée (moins de 1 Mégaoctet de mémoire souvent).

- Ils ont souvent des contraintes de temps réel.
- Ils opèrent souvent dans un environnement physique, donc complexe.
- Ils doivent donc se comporter le mieux possible avec ces ressources.

Un exemple typique d'informatique embarquée sont les robots métamorphes comme ceux développé par le projet CNRS/Robea MAAM [1]. Ces robots consistent en un assemblage reconfigurable de robots de taille inférieure.

Le développement d'un aspect de la sorte est cependant une tâche non négligeable, et demandera donc un certain travail.

Aspects de langages de haut niveau

Un aspect intéressant et déjà mentionné plus haut, est celui de la gestion des langages de haut niveau, tels que ACL et KQML. C'est donc un aspect complexe, qui est de plus à la frontière avec les aspect fonctionnels. Cet aspect n'a été que peu envisagé cependant, et est donc assez vague. Il pourrait vraisemblablement constituer un sujet de thèse à lui seul...

Aspect de protocole des rôles

Cet aspect évoqué lors de l'exploration de la première facette est encore inabouti. Cependant des pistes de recherche existent, en se basant notamment sur les activités transverses [Kri93], notion développée par Kristensen dans la conception objet. Ces théories ne sont qu'une base, mais si cet aspect arrive à voir le jour, il pourra servir de support à l'aspect suivant, qui est une piste de recherche encore prometteuse, et grande ouverte ...

Aspect de stratégie

Un autre aspect en projet, qui serait aussi vraisemblablement volumineux, est celui d'un aspect regroupant les notions de stratégie que peut employer un agent. L'utilisation d'un tel aspect permettrait de réutiliser encore plus de code, car il permettrait de séparer deux choses actuellement liées dans notre implémentation : les tâches qu'un agent doit accomplir, et la façon de les faire. Actuellement ces deux responsabilités sont toutes les deux situées dans les rôles que l'agent emploie. L'idéal serait donc de laisser les tâches dans le rôle, et mettre la stratégie qu'emploie l'agent dans un aspect à part (peut-être un méta-rôle). Ce changement est relativement important, puisqu'il nécessiterait de séparer les actions à faire de la stratégie pour les faire, et nécessiterait donc de réifier ces deux concepts en profondeur (trouver un moyen de réifier une

[1] `http://www.iutc3.unicaen.fr/serge/ProjetMAAM`.

suite de tâches, avec des alternatives ... pourrait revenir à créer un mini-langage de programmation, dont les «stratégies» serait les interpréteurs !).

Cet aspect est donc assez ambitieux, mais si il est implémenté, permettra de dissocier deux concepts importants, et ouvre donc la porte à un important gain de réutilisabilité.

5.3.2 Améliorer le support des aspects agents

La seconde facette est celle de l'intégration de l'AOP dans le domaine de la conception SMA. Au-delà de la comparaison apparemment fructueuse entre groupe et aspect, il reste à définir tous les concepts manquants pour permettre aux groupes multi-agents de se comporter comme des aspects.

Par exemple nous devons détailler les spécificités des groupes, et les adapter à l'AOP. Une première étape serait de définir complètement le mécanisme de «tissage» employé par les groupes, et de préciser les possibilités de configuration des groupes. Cela pourrait revenir à spécifier les points de jonctions spécifiques aux groupes et aux rôles, parmi lesquels ont peut dénombrer :

- l'admission d'un agent dans un groupe
- l'endossage d'un rôle par un agent
- la création d'un groupe
- les envois et réception de messages dont nous avons déjà parlé
- etc...

Ce travail est donc analogue à celui qui a été nécessaire pour spécifier les points de jonctions pour les classes. Cette facette possède donc un potentiel très intéressant aussi.

5.3.3 Expliciter les apports des SMAs pour l'AOP

Enfin la dernière facette, qui a été la moins explorée, pourrait bien être la plus intéressante : en effet, les spécificités des groupes, dont certaines restent encore à découvrir, offrent des facilités pour modeler des aspects qui sont parfois difficilement définissables en AOP classique : les aspects fonctionnels. Ceux-ci sont de plus très intéressant, car une bonne méthodologie pour les extraire augmenterait considérablement les facultés de réutilisabilité du code, en isolant les fonctionnalités un maximum. Là encore, beaucoup de travail reste à faire, car nous n'avons fait que défricher le terrain.

5.4 Sur le stage

Outre les résultats de recherche, ce stage fut particulièrement intéressant par les collaborations qu'il a entraîné, entre deux équipes distinctes (sur deux lieux eux aussi distincts, qui sont l'université de Caen et l'École des Mines de Douai), et entre deux thématiques de recherche qui semblaient peu liées entre elles. Le fait d'être à cheval sur deux domaines de recherche n'est cependant pas facile, car cela implique de maîtriser d'autant plus de concepts, de se documenter deux fois plus... La confrontation de ces deux courants fut tout de même fructueuse, comme le montrent les résultats de ce stage. Cette collaboration a d'ailleurs des chances de se poursuivre sous la forme d'une thèse, ce qui permettra d'explorer quelques-unes des nombreuses voies que ce stage a ouverte ....

Bibliographie

[Bar] Daniel Bardou. Roles, subjects and aspects : How do they relate ? Avail : http://citeseer.nj.nec.com/413435.html.

[BL02] Noury Bouraqadi and Thomas Ledoux. Aspect-oriented programming using reflection. 2002. Avail : http://csl.ensm-douai.fr/MetaclassTalk/uploads/1/aopUsingReflection.bou%raqadiLedoux.techReport2002.pdf.

[Bou99] Noury Bouraqadi. *A Smalltalk MOP for the Study of Metaclass Composition and Compatibility - Application to Aspect-Oriented Programming*. PhD thesis, 1999. Avail : http://csl.ensm-douai.fr/MetaclassTalk/uploads/1/phdNoury.ps.gz.

[Bou03] Noury Bouraqadi. Aop using reflection in smalltalk - the metaclasstalk experiment, 2003. Avail : http://csl.ensm-douai.fr/MetaclassTalk/uploads/1/aopUsingMetaclassTalk.%NouryBouraqadi.vub6jun2003.pdf.

[Chr] Alessandro Garcia Christina. Promoting advanced separation of concerns in intra-agent and inter-agent software engineering. Avail : http://citeseer.nj.nec.com/460915.html.

[CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proc. of ESEC/FSE*, 2001. Avail : http://www.cs.ubc.ça/~gregor/coady-FSE2001-aspectc-os.pdf.

[FG98] J. Ferber and O. Gutknecht. Aalaadin : a méta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS'98*, July 1998. Avail : http://citeseer.nj.nec.com/ferber97aalaadin.html.

[GdSLM] Alessandro F. Garcia, Viviane T. da Silva, Carlos J. P. Lucena, and Ruy L. Milidiú.

[GF97] O. Gutknecht and J. Ferber. MadKit : Organizing heterogeneity with groups in a platform for multiple multi-agent systems.

Technical Report 97188, LIRMM, 161, rue Ada - Montpellier - France, December 1997. Avail : `http://citeseer.nj.nec.com/gutknecht97madkit.html`.

[GKB91] Jim de Rivières Gregor Kiczales and Daniel G. Bobrow. *The Art of Meta Object Protocol*. MIT Press, 1991.

[KAJ+] G. Kiczales, J. Ashley, L. Jr, A. Vahdat, and D. Bobrow. *Object-Oriented Programming : The CLOS Perspective*. MIT Press.

[Ken] Elizabeth A. Kendall. Role models for agent system analysis, design, and implementation. Avail : `http://citeseer.nj.nec.com/393288.html`.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proc. of ECOOP*, 2001. Avail : `http://www.cs.ubc.ça/~gregor/kiczales-ECOOP2001-AspectJ.pdf`.

[KLM+97] G. Kickzales, J. Lamping, A. Mendhekar, C. Mæda, C. Videira Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of ECOOP*, 1997. Avail : `http://www.cs.ubc.ça/~gregor/kiczales-ECOOP1997-AOP.pdf`.

[KO96] Bent Bruun Kristensen and Kasper Osterbye. Roles : Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3) :143–160, 1996. Avail : `http://citeseer.nj.nec.com/kristensen96roles.html`.

[Kri93] Bent Buun Kristensen. Transverse activities : Abstractions in object-oriented programming,. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742, pages 279–296. Springer-Verlag, 1993. Avail : `http://citeseer.nj.nec.com/kristensen93transverse.html`.

[Kri96] Bent Bruun Kristensen. Object-oriented modeling with roles. In John Murphy and Brian Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1996. Avail : `http://citeseer.nj.nec.com/kristensen95objectoriented.html`.

[Kri97] Bent Bruun Kristensen. Subject composition by roles. In *Object Oriented Information Systems*, pages 181–196, 1997. Avail : `http://citeseer.nj.nec.com/kristensen97subject.html`.

[Meu97] W. De Meuter. Monads as a theoretical foundation for aop, 1997. Avail : `http://citeseer.nj.nec.com/demeuter97monads.html`.

[Nas03] Rabih Nassrallah. Programmation par aspects et services web, 2003. Avail : `http://csl.ensm-douai.fr/research/uploads/rabihNassrallah.dea2003EcoleD%esMinesDeDouai.pdf`.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12) :1053–1058, 1972. Avail : `http://www.cs.virginia.edu/~cs340/materials/papers/parnas.12.72.pdf`.

[Rob] Romain Robbes. Implémentation d'une architecture groupe/agent/rôle et application au «toy problème» de la factorielle. Avail : `http://www.iutc3.unicaen.fr:8000/romain/uploads/55/implementation.pdf`.

[ZA00] Alejandro Zunino and Analía Amandi. Brainstorm/J : a Java framework for intelligent agents. In *Proc. of the* 2nd *Argentinian Symposium on Artificial Intelligence (ASAI 2000 -* 29th *JAIIO)*, Tandil, Buenos Aires, Argentina, September 2000. SADIO. Avail : `http://www.exa.unicen.edu.ar/~azunino/asai2000.ps.gz`.