RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université LARBI Ben M'HIDI
-Oum el bouaghi-

Faculté des sciences exactes et des Sciences de la nature
et de la vie
Département de Mathématiques et Informatique

N ° d'ordre : ……..
Série : ……

Evaluation de l'Impact du Refactoring Aspect dans un Contexte Multi-Agents :

Une analyse dynamique

Mémoire présenté par :

CHEBOUT Mohamed Sedik

Pour l'obtention du magister en informatique

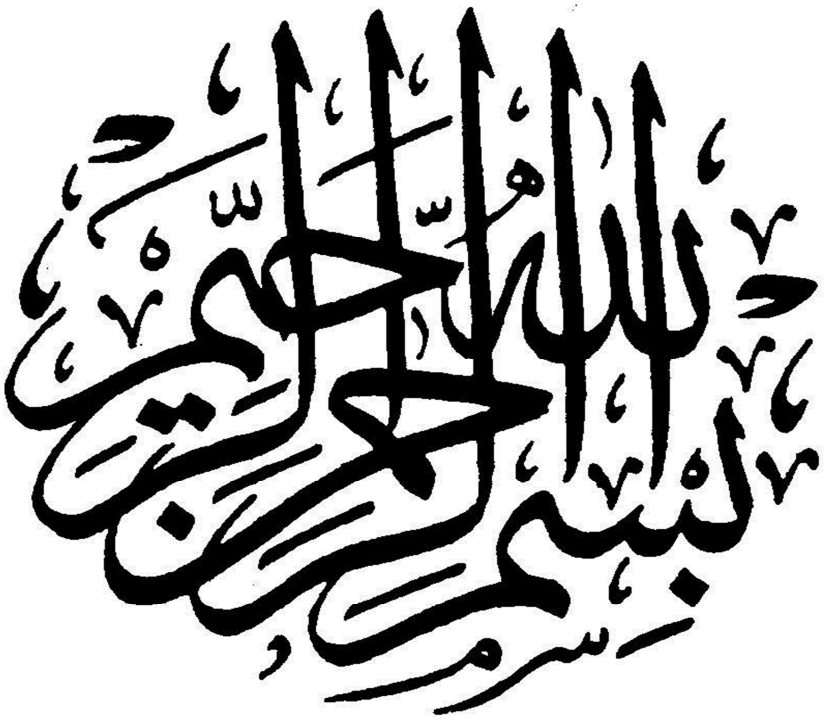
(Option : **Intelligence artificielle et imagerie**)

Soutenu le : **16/06/2014**

Devant le jury composé de :

NINI Brahim	MCA	Université d'Oum El Bouaghi	*Président*
MOKHATI Farid	MCA	Université d'Oum El Bouaghi	*Rapporteur*
BOUTEKKOUK Fateh	MCA	Université d'Oum El Bouaghi	*Examinateur*
LAOUAR Mohamed Ridda	MCA	Université de Tébessa	*Examinateur*

2013-2014


بسم الله الرحمن الرحيم

Remerciements

Louange à dieu. Bienfaiteur miséricordieux.
Paix et bénédiction sur son prophète, Mohamed, ultime envoyé.

Je tiens tout d'abord à exprimer ma profonde gratitude à Mr MOKHATI Farid maitre de conférences à l'université Larbi ben m'hidi d'oum el bouaghi, d'avoir accepté d'encadrer ce travail. J'apprécie son aide, ses commentaires judicieux ainsi que sa capacité de lire et de relire les différentes parties de ce mémoire.

Je tiens également à remercier Mr. BADRI Mourad, Professeur à l'université du Québec à trois rivières, Co-encadrant de ce mémoire, pour sa disponibilité, son dynamisme et son efficacité.

Je voudrais bien remercier les membres du jury qui m'ont fait l'honneur de bien vouloir juger ce travail et d'y apporter leurs critiques constructives et leurs valeureuses remarques.

Je remercie tous mes collègues, particulièrement Ayyoub KALACHE, pour sa qualité humaine remarquable et son encouragement.

Je remercie très chaleureusement ma cousine Meriem BELGUIDOUM de m'avoir conseillé, et encouragé pour terminer ce travail.

Je remercie infiniment mr.KHADRAOUI Samir Akid, secrétaire générale de la commune de Meskiana, pour sa patience, sa compréhension, son encouragement, et de me permettre à finaliser mes travaux dans ce mémoire.

Mes derniers remerciements qui ne sont pas les moindres sont pour les personnes les plus chères dans ma vie, mes parents qui m'ont toujours soutenu et encouragé dans tout ce que j'entreprenais durant toute ma vie. Ce travail est le fruit de leurs patiences et sacrifices. Je ne pourrais pas oublier ma femme qui a subi tous les désagréments durant ces deux années et m'a énormément soutenu et motivé.

Enfin, je remercie tous les gens que j'aurais involontairement oubliés de remercier.

Mohamed sedik CHEBOUT

Dédicaces

Je dédie ce travail à :

Mes très chers parents
Ma femme Noussaiba
Mes frères
Ainsi qu'à tous ceux qui m'aiment.

Résumé

Les applications multi-agents existantes dans la littérature sont développées sans tenir compte de la séparation entre les préoccupations fonctionnelles de celles non fonctionnelles, ce qui affecte leurs qualités. Une des techniques utilisées pour améliorer la qualité du logiciel est le Refactoring. Cette technique sert à améliorer l'extensibilité, la modularité, la réutilisabilité, la complexité et la maintenance du logiciel. Une piste relativement nouvelle pour implémenter le Refactoring est l'utilisation de la programmation orientée aspect. Par conséquent, il en résulte l'apparition d'une nouvelle technique connue sous le nom de Refactoring Orienté Aspect. Dans ce mémoire, nous proposons une nouvelle approche basée sur l'analyse dynamique pour évaluer l'impact du Refactoring Aspect sur la qualité des applications multi-agents, plus spécifiquement, l'approche proposée a pour but de savoir si le Refactoring Orienté Aspect nous permet d'apporter une amélioration sur le comportement des agents en terme de communication et échange de messages pour atteindre et compléter leurs tâches. Notre approche est sanctionnée par un outil d'analyse de performances qui supporte les spécificités des agents implémenté sous la plateforme AgentFactory.

Mots clés : Multi-Agents, Refactoring orienté Aspect, Analyse Dynamique, Impact.

Abstract

Existing multi-Agent Application in the literature are developed without considering the separation between functional concerns of those non-functional, which affect their quality. Among techniques used to improve the quality of the software we cite the Refactoring. This technique is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. Aspect-oriented programming approach is supposed to enhance a system's features, such as its modularity, readability and simplicity. Due to a better modularization of crosscutting concerns, the combination of the two techniques results in the appearance of a new technique known as Aspect Oriented Refactoring. In this manuscript we present a novel approach based on dynamic analysis for assessing the quality of multi-agent application. The purpose of the proposed approach is to assess the impact of Refactoring Oriented Aspect on agent behavior in terms of communication to achieve their goals. Our approach is supported on profiling tool working under AgentFactory platform.

Keywords: MAS (Multi Agent System), Aspect Oriented Refactoring, Dynamic Analysis, Impact.

ملخص

يقوم إنشاء التطبيقات المتعددة الأعوان الحالية في الدراسات السابقة على التطوير بدون الأخذ بعين الاعتبار الفصل بين الوظائف الأساسية والفرعية. وهذا يؤثر على نوعيتها. تعتبر إعادة الصياغة من بين التقنيات المستخدمة لتحسين نوعية البرمجيات. هذه التقنية تقوم على تحسين إمداد البرمجية بوظائف أخرى. وزيادة أقسام البرنامج، كما تسهل عملية الصيانة وإعادة الاستخدام في المستقبل. من بين الطرق الحديثة لتطوير الصياغة في البرمجيات نجد البرمجة الموجهة المنحى مما ينتج عنه تقنية جديدة تعرف بـ الصياغة الموجهة المنحى.

سنقوم في هذه المذكرة باقتراح نمط حديث يعتمد على التحليل الديناميكي من أجل تقييم التأثير الذي تقوم به إعادة صياغة البرمجيات باستخدام البرمجية موجهة المنحى على نوعية التطبيقات المتعددة الأعوان. بمعنى أدق : النمط المقترح يهدف إلى معرفة ما تقدمه لنا إعادة الصياغة الموجهة المنحى من تحسين على هيئة الأعوان من حيثُ الاتصالات وتبادلُ المعلومات من أجل تحقيق الأهداف المرجوة. تستخدم منهجيتنا وسيلة تعمل تحت نظام : مصنع الأعوان.

الكلمات المفتاحية : نظام متعدد الأعوان، الصياغة الموجهة المنحى، التحليل الديناميكي، التأثير.

Table des matières

Table des Figures

Chapitre I

Chapitre II

Chapitre III

Chapitre IV

Liste des tables

Introduction générale

1. Contexte de recherche

Les Systèmes Multi-Agents (SMA) représentent un axe de recherche important issue de l'Intelligence Artificielle Distribuée. Essentiellement, ces systèmes cherchent à étudier la manière de répartir un problème sur un certain nombre d'entités autonomes et coopérantes appelées agents. Ils étudient également la manière de coordonner les comportements intelligents de ces agents selon des lois sociales [Dem95; Bon88; Fer 95].

La performance est un critère très important qu'il faut prendre en considération lors de la conception et du développement de tout système informatique. En effet, les concepteurs et les développeurs visent toujours à construire le système qui permet d'atteindre les meilleurs résultats à moindre coût. Reed [Ree94] remarque que les programmeurs admettent des performances "acceptables" et ne cherchent pas à optimiser leurs programmes pour atteindre des meilleures performances.

Le refactoring est une activité d'ingénierie logicielle qui consiste à modifier le code source d'une application de manière à améliorer sa qualité tout en préservant son comportement extérieur vis-à-vis de ses utilisateurs. La plupart des environnements de développement contemporains supportent l'application automatique de refactoring au niveau du code source. La première apparition de refactoring est due à [Opd92]. Fowler [Fow99] a repris certains de ces refactorings et créé un catalogue de 72 refactorings du code Java. Par ailleurs, Kerievsky [Ker04] a proposé un catalogue de refactorings qui permet la restructuration du code en introduisant des patrons de conception.

La programmation orientée aspect (POA) est un nouveau paradigme de programmation qui trouve ses racines en 1996, suite aux travaux de Gregor Kiczales et de son équipe au Centre de Recherche Xerox à Palo Alto. La POA est donc une technologie relativement jeune, puisque les premiers outils destinés à son utilisation ne sont apparus qu'à la fin des années 90. La POA est indépendante de la programmation par objets; il est possible d'utiliser la programmation par aspects conjointement à d'autres paradigmes de programmation (fonctionnelle, impérative, etc) [Ter04].

L'analyse dynamique de programmes est une famille de techniques d'analyse d'exécution qui mesure le comportement d'un programme, en particulier la fréquence et la durée des appels de fonction, quand il fonctionne [Bal99]**.** La sortie est une suite des événements enregistrés (*une trace*) ou d'un résumé statistique des événements observés (*un profil*). L'analyse statique de programmes est une famille de techniques permettant de dériver des résultats sur l'exécution de programmes sans exécuter ces derniers [Nie04]. Elle se distingue ainsi de l'analyse dynamique ou test, qui revient à essayer le programme

sur différentes entrées jugées représentatives, afin de vérifier s'il produit les résultats attendus sur ces entrées.

2. Problématique

Actuellement, il y a beaucoup de travaux de recherche qui se focalisent sur l'impact du refactoring orientée aspect en utilisant l'analyse statique, cependant il y a très peu de travaux qui explorent l'espace de l'analyse dynamique dans le domaine du refactoring orientée aspect et surtout dans un contexte orienté agent.

Pour traiter ce point; on suppose que l'on dispose de :

- Langage de pattern qui supporte les fondements de la programmation orientée aspect et nous permet de traiter les préoccupations transversales comme des modules supplémentaires représentés sous forme d'aspects, et nous offrir aussi un compilateur orienté aspect et un mécanisme d'intégration qui sert à insérer les aspects dans des endroits bien déterminés dans le programme principal.
- Un outil de refactoring orienté aspect (automatique ou semi-automatique) qui prend en entrée un programme orienté agent non réfactorisé et qui produit en sortie sa version réfactorisée, l'outil en question doit prendre en considération les préoccupations transversales spécifiques à l'agent. Il doit aussi interagir avec la plate-forme agent sur laquelle le système orienté agent est exécuté.
- Un profileur spécifique qui nous permet, non seulement de capturer et collecter les données sur le temps CPU, l'utilisation de la mémoire et des ressources, mais aussi de collecter des informations sur le comportement des agents; quantifier les messages échangés, le poids des messages échangés, l'impact des messages échangés du côté du récepteur, la typologie des messages échangés, etc.

3. Objectifs

L'objectif de notre travail est de proposer une nouvelle approche d'évaluation de l'impact de la technique du refactoring aspect sur les applications multi-agents. Notre approche consiste à identifier et proposer des critères d'évaluation dynamique spécifique à l'agent qui prend en considération la communication entre les agents comme attribut principal qui fait partie de l'ensemble des propriétés de l'agent telles que la collaboration, la coopération, etc. Les résultats de notre évaluation sont représentés par un graph d'appel orienté agent généré par un profileur dédié, après la terminaison de la session d'exécution.

Pour valider notre approche d'évaluation, une application orientée agent a été développée sous la plateforme AgentFactory et réfactorisée de différentes manières selon le constructeur orienté aspect choisi, le processus de refactoring se déroule manuellement.

4. Organisation du mémoire

Le reste de ce mémoire est organisé en 4 chapitres :

Le chapitre 1 est une introduction au monde agent avec ses différents types tout en mettant l'accent sur la communication dans les systèmes multi agents comme problématique de recherche. Il présente également un aperçu sur les plateformes de développement et de déploiement des systèmes multi agent.

Le deuxième chapitre présente dans sa première partie la programmation orientée aspect et la problématique adressée par ce nouveau paradigme et, dans sa deuxième partie le langage AspectJ avec ses différents constructeurs qui permettent de manipuler les préoccupations transversales.

L'analyse dynamique et le profilage font l'objet du troisième chapitre. Nous abordons dans ce chapitre l'analyse dynamique, et décrivons les concepts liés au profilage, et les différentes techniques d'instrumentation de code, et nous donnons à la fin du chapitre quelques exemples d'outils de profilage.

Le dernier chapitre présente notre approche que nous proposons pour l'évaluation de l'impact du refactoring aspect sur la qualité des applications SMA. Une discussion détaillée résume les résultats obtenus après la validation de l'approche.

Finalement, nous donnons quelques conclusions et perspectives importantes à moyen termes.

Chapitre I

Agents et systèmes multi agents

1. Introduction

Nous allons aborder dans ce chapitre des définitions et des concepts clés relatifs au domaine des SMA, qui nous paraissent les plus appropriés pour le besoin de l'approche d'évaluation que nous avons l'intention de proposer et décrire.

Notre objectif n'est pas d'établir un état de l'art complet et rigoureux de ces notions mais plutôt d'introduire certains concepts qui nous seront utiles par la suite. Il s'articulera essentiellement autour de cinq sections : on commence par définir le concept d'agent ainsi qu'une typologie d'agents, nous présenterons ensuite une des architectures les plus utilisée, à savoir BDI (Belief, Desire, Intention).

Le concept de système multi-agent et ensuite présenté avant de passer en revue la communication comme problématique de recherche. Une liste des principales plateformes de développement et de déploiement des SMAs est traitée ainsi que la plateforme AgentFactory que nous avons opté dans le cadre de notre recherche.

Enfin un bilan du chapitre est dressé afin de montrer l'apport de la communication dans le monde d'agents dans notre cas.

2. Notion d'agent

La notion d'agent est issue de différents domaines comme la Psychologie Sociale, l'Intelligence Artificielle, les Sciences de l'Homme et de la Vie, etc. Le concept assez riche se spécialise en fonction des applications pour lesquelles l'agent a été conçu.

2.1. Définitions

« ***Un agent est défini comme une entité physique ou virtuelle, autonome, capable d'agir dans un environnement, de communiquer avec d'autres agents, de percevoir son environnement et de se reproduire (éventuellement)***» (Figue 1.1) [Fer95].

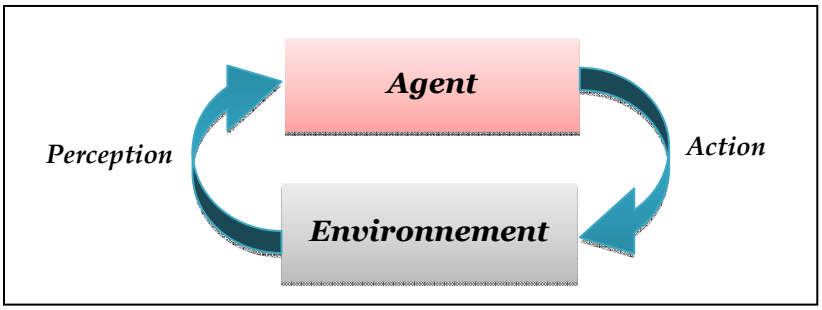


Figure 1.1 **Relation Agent-Environnement**

De plus, un agent possède usuellement un objectif individuel (fonction de satisfaction), des ressources, une représentation partielle de l'environnement, des compétences et il offre ses services. Son comportement est fonction de ses observations, de ses connaissances, de ses croyances, de ses compétences et de ses interactions [Gle04].

Le concept de l'agent trouve donc tout son intérêt dans la notion de comportement, de réaction et de perception. Ces notions contribuent à définir l'agent comme une entité active coopérant dans l'univers des agents. Elles lui permettent de tenir compte de son environnement, d'interagir avec celui-ci, afin d'atteindre un but commun [Gho13].

Une autre définition proposée par [Woo00a] : Un agent est un système informatique encapsulé situé dans un environnement dans lequel il est capable d'effectuer une action flexible et autonome, compatible aux objectifs de la conception.

Les agents sont :

- Des entités clairement identifiables de résolution de problèmes avec des bornes et des interfaces bien définies;
- Situés dans un environnement particulier; ils reçoivent des entrées liées aux états de cet environnement par des capteurs et agissent sur cet environnement par des émetteurs;
- Destinés à atteindre un objectif spécifique;
- Autonomes et responsables de leur comportement;
- Capables d'adopter un comportement flexible pour résoudre des problèmes selon les objectifs de la conception;
- Capables dans un univers multi-agents, de communiquer, coopérer, se coordonner, négocier les uns avec les autres.

La figure 1.2 donne, de façon générale, l'architecture interne d'un agent.

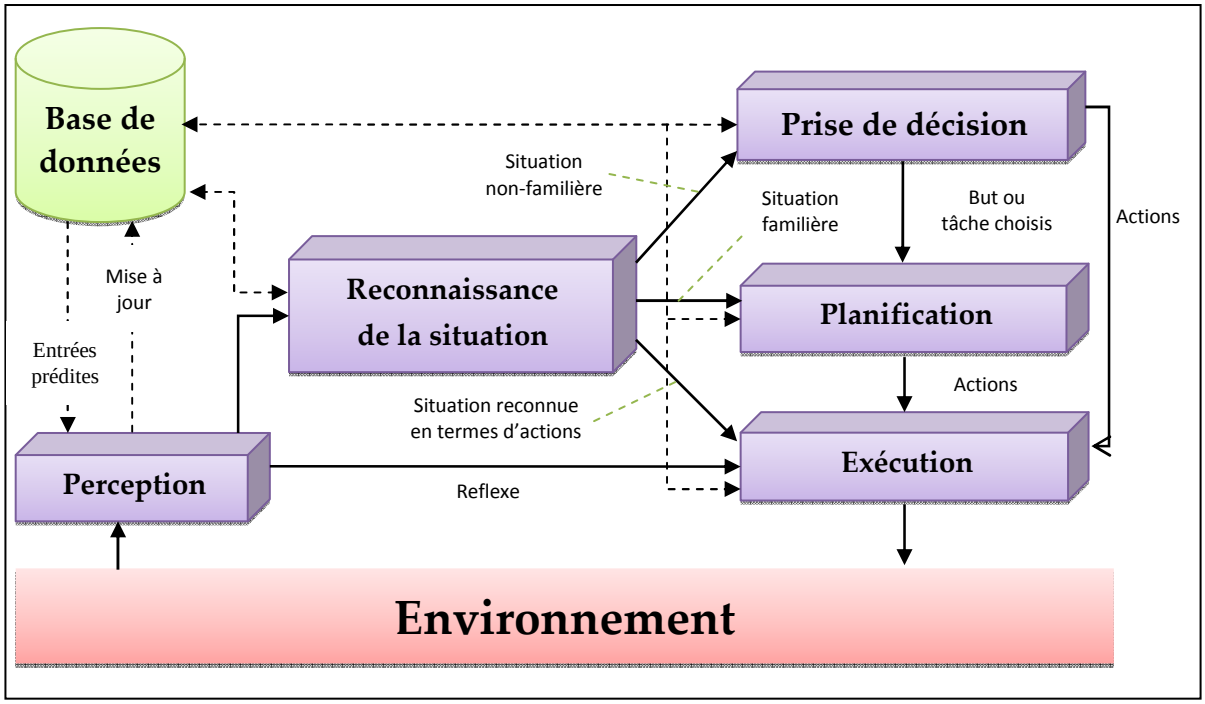


Figure 1.2 : **Architecture d'agent [Cot99]**

Dans la littérature, d'autres définitions [Jen98] mettent en évidence d'autres propriétés clés :

- Les aptitudes sociales : désignent la capacité d'un agent à interagir avec les autres agents de façon coopérative ou compétitive pour atteindre ses objectifs ;
- La pro-activité : désigne l'aptitude d'un agent à se fixer des buts pour atteindre ses objectifs sur sa propre initiative.

2.2. Types d'agents

Deux types d'agents ont longtemps été distingués dans les SMA [Cha96] :

- *les agents réactifs* au comportement basé sur les stimulus/réponses.
- *les agents cognitifs* au comportement plus « réfléchi », c'est-à-dire résultant d'un choix parmi un ensemble d'actions possibles, ce choix étant le résultat d'un raisonnement.

D'autres types d'agents, qualifiés d'hybrides, utilisant donc ces deux types de comportement, sont ensuite apparus.

Un type d'agents appelé *agent hybride* apportant une réponse aux imperfections des deux types précédents a été proposé [Mül96b].

Nous présentons brièvement ces différents types ainsi que l'architecture BDI relative aux agents cognitifs.

2.2.1 Agents réactifs

Les agents réactifs sont des agents ayant la capacité de répondre uniquement à la loi stimulus/réponse. Pour ce faire, ils n'ont pas besoin ni d'expliciter leurs buts, ni d'établir des mécanismes de planification [Fer88]. De ce fait, un agent réactif est extrêmement simple. Il dispose d'un processus de raisonnement procédural, d'un protocole et d'un langage de communication réduit [Mor94].

Les architectures *réactives*, dites *aussi comportementales*, se caractérisent par des agents qui ont la capacité de réagir rapidement à des problèmes simples, qui ne nécessitent pas un haut niveau de raisonnement. En effet, leurs décisions sont essentiellement basées sur un nombre très limité d'informations et sur des règles simples de type *situation – action* [Mül96b]. Par ailleurs, ce type d'architectures ne nécessite aucune représentation symbolique de l'environnement réel.

Dans un système à base d'agents réactifs, l'intelligence émerge de l'interaction des agents avec leur environnement. L'inconvénient majeur de ce type d'architectures est l'absence de formalisme, ce qui ne facilite pas la compréhension et la prédiction du comportement des agents, et la quasi-impossibilité de vérifier leur cours d'action [Gho13].

2.2.2 Agents cognitifs

Les agents cognitifs, quant à eux, sont des agents à base de connaissance se caractérisant par leur capacité de raisonnement et par la complexité de leur structure. Chaque agent dispose d'une base de connaissances et diverses informations liées à son domaine d'expertise et à la gestion des interactions avec son environnement. Ils sont généralement intentionnels, c'est-à-dire qu'ils possèdent des buts et des plans explicites leur permettant d'atteindre leurs objectifs. Ils sont parfois amenés à négocier pour résoudre leurs conflits [Fer88].

Ce qui distingue principalement les agents cognitifs des agents réactifs est leur faculté d'anticipation. Leur capacité à raisonner sur des représentations du monde leur permet de mémoriser des situations, de les analyser, de prévoir les changements induits par leurs actions et finalement de décider du comportement à adopter dans le futur et ainsi sur des critères de granularité (tâches, données, paquet de communication, etc.). Un point crucial dans la conception d'agents cognitifs est donc la planification de leurs actions. Une revue détaillée des approches de planification d'action est proposée dans [Seg01; Bal02].

La table ci-dessous établit la différence entre les agents cognitifs et les agents réactifs. La liste des caractéristiques n'est pas exhaustive [Mor94].

Caractéristiques	Agent Cognitif	Agent Réactif
Capacité de raisonnement	Elevé	Faible
Représentation explicite de l'environnement	Oui	Non
Nombre d'agents	Petit	Grand
Granularité	Forte	Faible
Structure	Complexe	Simple
Temps de réponse	Lent	Rapide
Modularité de l'architecture	Oui	Non

Table 1.1 **Comparatif entre agent cognitif et agent réactif [Sea90]**

2.2.3 Agents hybrides

Les architectures précédentes présentent certaines faiblesses. En effet, les architectures purement réactives ont un comportement assez simpliste alors que les architectures délibératives utilisent des mécanismes de raisonnement qui ne sont pas faciles à manipuler et qui ne sont pas suffisamment réactifs.

Afin d'apporter une réponse à ces imperfections, des architectures hybrides en couches ont été proposées [Mül96b]. L'idée principale est de structurer les fonctionnalités d'un agent en deux ou plusieurs couches hiérarchiques qui interagissent entre elles afin d'atteindre un état cohérent de l'agent.

Une approche hybride structurée en couches présente plusieurs avantages :

- Elle permet de modulariser un agent; ainsi les différentes fonctionnalités sont clairement séparées et reliées par des interfaces bien définies ;
- Elle permet une conception de l'agent plus compacte, ce qui augmente sa robustesse et facilite son « débogage » ;
- Elle accroît les capacités de l'agent car les différentes couches peuvent s'exécuter en parallèle ;
- Elle augmente la réactivité de l'agent car il peut raisonner dans un monde symbolique tout en surveillant son environnement et en réagissant en conséquence;
- Elle réduit les connaissances nécessaires à une couche individuelle pour prendre ses décisions.

2.2.4. Architecture BDI : une architecture d'agents cognitifs

L'architecture d'agents délibératifs la plus importante est l'architecture BDI (Belief, Desire, Intention). L'idée de base de l'approche est de décrire l'état interne d'un agent en termes d'*attitudes mentales* et de définir une architecture de contrôle grâce à laquelle l'agent peut sélectionner le cours d'action de ses attitudes mentales :

- Croyances (Beliefs) : ce que l'agent connaît de son environnement ;

- Désirs (Desires) : les états possibles vers lesquels l'agent peut vouloir s'engager ;
- Intentions (Intentions) : les états vers lesquels l'agent s'est engagé ou a engagé des ressources.

Il existe plusieurs architectures internes possibles pour mettre en œuvre un agent (BDI [Geo88], hybride, réactif, etc). Nous choisissons de présenter uniquement l'architecture BDI car elle permet d'illustrer de nombreuses propriétés chez les décideurs. Une présentation plus détaillée des autres architectures est dans [Boi01].

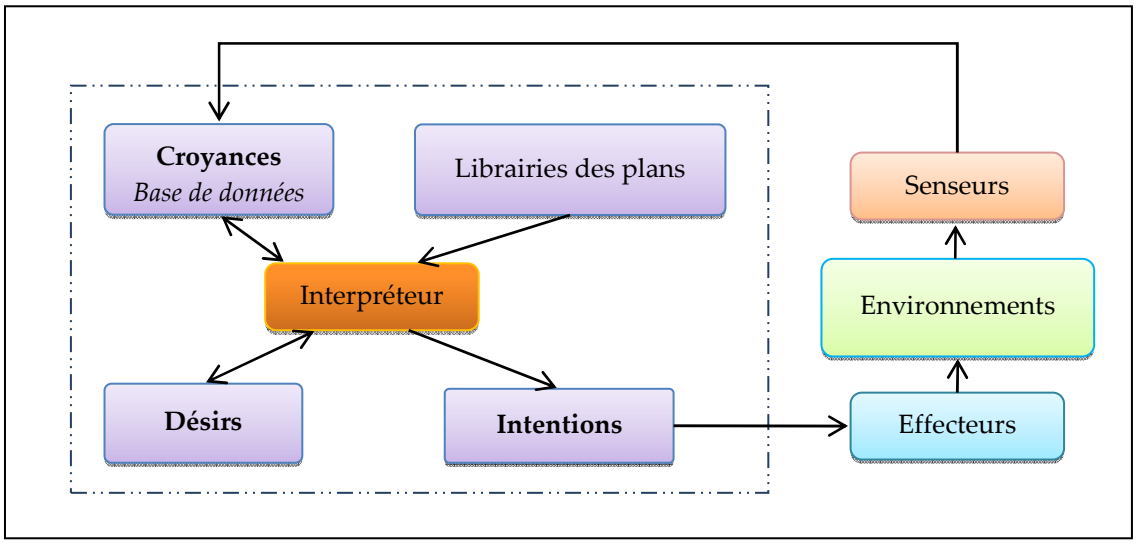


Figure 1.3. **Architecture BDI [Kin96]**

Dans des approches BDI plus pratiques, il a été montré que ces trois attitudes mentales n'étaient pas suffisantes, une extension a alors été proposée en rajoutant la notion de *buts* (goals) et de *plans* (plans) [Gho13].

- *Croyances* **:** décrivent l'état de l'environnement du point de vue d'un agent. Elles expriment ce que l'agent croît sur l'état courant de son environnement [Mül96b]. Les croyances peuvent évoluer dans le temps en fonction de nouvelles perceptions ou interactions avec d'autres agents.
- *Désirs* **:** représentent les états de l'environnement ou de lui-même, que l'agent aimerait voir atteints, si possible. Un désir est une étape dans le processus de création d'un *but*. Si un désir d'un agent est poursuivi de manière consistante, il devient l'un des *buts* qui indiquent les options de gestion qu'à un agent. Tous les désirs d'un agent ne sont pas nécessairement consistants ni réalisables simultanément ;
- *Intentions* **:** les intentions d'un agent sont « les désirs que l'agent a décidé d'accomplir ou les actions qu'il a décidé de faire pour accomplir ses désirs ;
- *Plans* **:** sont des connaissances qui concernent l'ensemble des actions qu'il faut entreprendre pour atteindre un but (désir). Le plan effectivement choisi et mis en œuvre correspond à une intention.

3. Système multi-agents

«***Un Système Multi-Agent est un macro-système composé d'agents autonomes qui interagissent dans un environnement commun pour réaliser une activité collective cohérente, bien qu'ils puissent chercher à atteindre des objectifs individuels parfois contradictoires*** » [Gle04].

Agréger des agents dans un ensemble ne signifie pas forcément que le système est un système multi-agent. Parler de systèmes multi-agents implique la prise en compte de problématiques organisationnelles et collectives portant sur les entités, la dynamique et la structure [Arl04].

Un agent au sein d'un système peut avoir un *rôle* ou une certaine *tâche* à remplir vis-à-vis des autres agents. La dynamique induite par le regroupement d'agents doit être prise en compte à la fois au niveau des agents (Quels sont leurs plans ? Quels services offrent-ils aux autres ? Quelles capacités collectives possèdent-ils ?), et au niveau du collectif (Quelles sont les interactions possibles ? Quelles collaborations/ coopérations/formations peuvent-ils adopter ?).

Enfin, la façon dont sont agencés les agents au sein de la société et dans quel contexte est chose primordiale. Les notions d'organisation, avec des modèles comme AGR (Agent/Groupe/Rôle) [Fer07], ou bien d'infrastructures collaboratives, sont des sujets de recherche importants dans le domaine multi-agent.

Un système multi-agent est donc un système composé d'agents autonomes ayant pour but de fournir une fonction collective. Les mécanismes permettant d'obtenir cette fonction prennent en compte les enjeux sociaux entre agents, comme la coopération.

Il ressort de cette définition un élément essentiel des systèmes multi-agents, celui de l'interaction. Les interactions dans un SMA permettent aux agents de participer à la réalisation d'un but global. Selon [Cha01], les interactions sont assurées par un transfert d'informations entre agents ou entre l'environnement et les agents, à l'aide de deux mécanismes : perception et communication.

- Dans la perception, les agents ont connaissance d'un changement de comportement d'un tiers à travers l'environnement.
- Dans la communication, un agent fait un acte délibéré de transfert d'informations vers un ou plusieurs autres agents.

Dans la littérature, trois formes d'interaction entre agents sont distinguées [Bou06] :

- *la coopération* désigne une situation où un groupe d'agents travaillent ensemble pour résoudre un but commun. D'une manière générale, les agents coopèrent

parce qu'ils ont besoin de partager des ressources ou/et bien des compétences pour accomplir leurs objectifs ;

- *la coordination* désigne l'organisation du travail des agents afin d'éviter les interactions négatives et favoriser les interactions utiles. Certaines situations de coopération exigent que les agents se coordonnent pour réaliser leur but global commun ;
- *la communication* désigne les échanges d'informations entre agents. La communication rend la coopération ou la coordination possible.

4. Notion d'environnement

Ferber [Fer95] considère qu'un système multi-agent est composé d'agents, d'objets, d'un environnement et d'opérations. Nous choisirons de présenter l'environnement selon deux axes :

- *l'environnement d'un agent* est constitué de tout ce qui l'entoure : les objets ou les autres agents. On distingue deux types d'environnements, physique et social, pour un agent :
  - l'*environnement physique* correspond à tous les objets que l'agent peut percevoir ou sur lesquels il peut agir. Par exemple, si l'on considère une fourmi comme étant un agent, son environnement physique est composé de nourriture et d'obstacles. On parle d'environnement physique essentiellement dans le cas d'agents situés.
  - l'*environnement social* correspond aux autres agents avec lesquels un agent est en interaction par envoi de message par exemple. On parle d'environnement social surtout dans le cas d'agents communicants.
- *l'environnement du système* est composé de tout ce qui n'est pas dans le système : les objets ou d'autres agents faisant partie d'un autre système multi-agent. L'environnement du système est ce qui va le guider vers une adéquation fonctionnelle dans le cas de systèmes auto-organisateurs.

5. Communication dans les systèmes multi-Agents

Les communications, dans les systèmes multi-agents, sont à la base des interactions et de l'organisation. Si les agents peuvent coopérer, coordonner leurs actions, réaliser des tâches en commun, c'est parce qu'ils communiquent. Koning et Pesty [Kon01] distinguent deux modes de communication : la communication directe par passage de message et la communication indirecte via l'environnement ou par partage de ressources.

5.1. Communication par envoi de messages

La *communication par envoi de messages* consiste à échanger des messages explicites entre les agents [Cha87]. La communication peut être individuelle. Un agent désigne un seul

agent destinataire de son message. Elle peut être regroupée. Le message est alors diffusé entre l'ensemble des agents (diffusion totale) ou d'un sous-ensemble d'agents (diffusion partielle).

Les systèmes utilisant la communication par envoi d'informations sont caractérisés par une distribution totale des connaissances, des résultats partiels et des méthodes utilisées pour aboutir à un résultat. Un agent ne peut donc manipuler que sa base de connaissances locales. Il peut, cependant, envoyer des messages aux autres agents qu'il connaît et sont appelés ses accointances [Gho13]. C'est le mécanisme le plus connu et le plus répandu.

5.2. Communication par partage d'information

La *communication par partage d'information* est parfois désignée par le modèle de Blackboard ou Tableau noir [Hay85, Cha87, Nii86]. Dans ce mode de communication, la résolution du problème commence lorsque le problème et les données initiales sont sur le tableau (figure 1.4). Cette structure comporte en plus les résultats partiels fournis par les agents (souvent appelés sources de connaissances). Elle représente de ce fait le seul moyen d'échanger les informations.

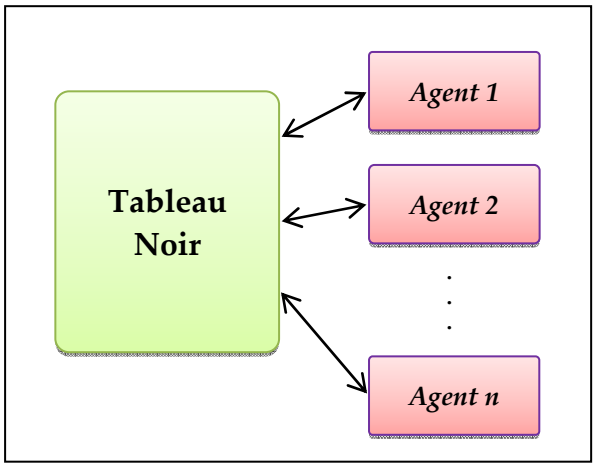


Figure 1.4 **Communication par partage d'informations [Hey85]**

L'avantage de cette architecture est surtout sa souplesse pour décrire les modules et articuler leur fonctionnement. Son principal inconvénient provient de sa relative inefficacité. De ce fait, elle s'avère particulièrement utile lors de la phase de prototypage ou lorsque les temps de réponses ne sont pas trop contraints.

5.3. Les langages de communication

D'après [Fin97], pour que les agents puissent interagir de manière efficace, ils doivent posséder un langage de communication commun, leur permettant de se comprendre ainsi que de s'échanger des informations et des connaissances.

Avant aborder les langages de communication existants, nous présentons brièvement la théorie des actes de langage, considérée en intelligence artificielle comme un modèle général de communication entre les agents.

5.3.1. Théorie des actes de langage

La théorie des actes de langage[1] ou « Speech Act Theory », constitue un fondement théorique de la communication, basée sur l'idée suivante : « Lorsqu'on parle, on effectue des actions ». Un acte de langage définit un message contenant l'affirmation positive ou négative, et provoquant des changements de l'environnement.

Dans la théorie des actes de langage, les intentions des émetteurs sont identifiées en utilisant les verbes performatifs. Ces derniers sont classifiés en plusieurs catégories : les affirmatifs (informer), les directifs (ordonner), les promissifs (promettre), les déclaratifs (déclarer) et les expressifs (exprimer).

D'après cette théorie, chaque acte de communication multi-agent peut être décrit sous la forme d'un message, dont le type est défini par le verbe performatif, tel que « demander » ou « informer ». Dans la littérature, deux langages sont fréquemment utilisés : KQML et FIPA-ACL.

5.3.2. KQML (Knowledge Query and Manipulation Language)

Le langage KQML (Knowledge Query and Manipulation Language) [Lab98] a été proposé pour supporter la communication inter-agents. Ce langage définit un ensemble de types de messages (performatifs) et des règles qui définissent les comportements suggérés pour les agents qui reçoivent ces messages. Chaque message comprend trois couches [Lab99] :

- ***La couche de contenu*** : qui spécifie le contenu réel du message d'agent ;
- ***La couche de communication*** : décrit tous les paramètres de communication de bas niveau, par exemple, l'identificateur de l'agent émetteur et celui de l'agent récepteur, l'identificateur de la communication, etc.
- ***La couche de message*** : considéré comme le noyau de KQML. Sa fonction principale s'agit d'identifier le protocole de réseau, utilisé pour envoyer le message et de déterminer le performative, indiquant le type de ce message (par ex. une affirmation, une requête, une commande, etc.).

Le message KQML est représenté sous la forme d'une liste, contenant le performative, qui correspond au type particulier d'acte de langage (par exemple, tell (transférer l'information aux autres agents), ask-one (demander la réponse à l'agent correspondant), etc.), et les arguments associés à ce performative (Figure 1.5). Il est à noter que l'ordre des arguments dans une liste n'est pas important.

---

[1] Théorie introduite en 1962, par le philosophe et linguiste anglais J.L.Austin dans son livre « How to do things with words ».

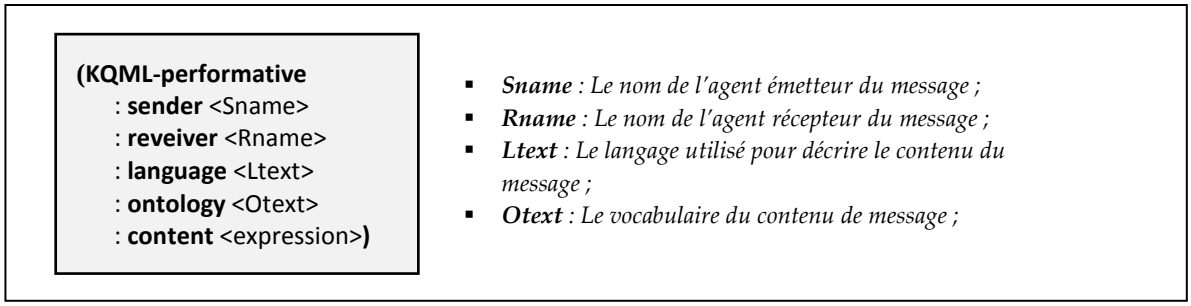


Figure 1.5 **la structure d'un message KQML [KQML97]**

KQML est muni d'une seule règle de conversation : *une conversation commence au moment où le premier message est envoyé par un agent à un autre agent et se termine quand ce dernier répond*. Pour chaque message, les actes de langage autorisés sont restreints aux performatives.

KQML fournit un ensemble de trente-six (36) performatifs (Table 1.2) divisé en trois catégories. Dix-huit performatives de discours permettent l'échange d'informations et de connaissances :

- Sept (07) performatives de régulation des conversations traitent les erreurs et les variantes à la règle de conversation (error, sorry, standby …)
- Onze (11) performatives d'interconnexions étendent les possibilités de communication à plus de deux agents : (register, unregister, broadcast …)
- Dix-huit (18) performatives de discours (ask-if, ask-all, tell, describe, stream-all …)

Le langage KQML permet d'utiliser les différents protocoles de communication, notamment, TCP/IP, SMTP (e-mail), COBRA, etc.

Bien qu'il présente un grand intérêt pour les utilisateurs, KQML montre quelques lacunes [Coh95] :

- La signification floue de certains performatifs (par exemple, le cas d'un performatif « dénier ») ;
- Le manque des performatifs promissifs, exprimant l'engagement, auprès d'un tiers, d'accomplir une action ;
- L'utilisation de ce langage que pour des communications isolées.

Performative	Type d'acte	Description
ask-if	Directif	S veut savoir si P est dans la base de connaissances de R.
ask-all	Directif	S veut connaître toutes les instances de P qui soient vraies pour R.
ask-one	Directif	S veut connaître une des instances de P qui soit vraie pour R.
stream-all	Directif	version multi-réponses de ask-all
eos	Déclaratif	marqueur de fin pour un stream-all
tell	Assertif	P est dans la base de connaissances de S.

untell	Assertif	P n'est pas dans la base de connaissances de S.
deny	Assertif	la négation de P est dans la base de connaissances de S.
insert	Directif	S veut que R ajoute P à sa base de connaissances
uninsert	Directif	S veut que R annule un précédent insert
delete-one	Directif	S veut que R supprime de sa base de connaissances une des instances qui correspond à P.
delete-all	Directif	S veut que R supprime de sa base de connaissances toutes les instances qui correspondent à P.
undelete	Directif	S veut que R annule un précédent delete-one ou un précédent delete-all.
achieve	Directif	S veut que R essaie de rendre P vrai.
unachieve	Directif	S veut que R annule un précédent achieve.
Advertise	Déclaratif	S veut que R sache que S peut envoyer un message dela forme de P.
unadvertise	Déclaratif	S veut que R sache que S ne peut plus envoyer de message de la forme de P.
subscribe	Directif	S veut que R le prévienne si des changements au(x)réponse(s) d'une de ses demandes interviennent.
error	Assertif	S considère que le précédent message de R est erroné.
sorry	Assertif	S comprend le précédent message de R mais ne peut pas le satisfaire.
standby	Assertif	S informe R qu'il doit attendre qu'il soit prêt avant de lui envoyer la réponse à P.
ready	Engageant	S indique à R qu'il a une réponse et qu'il est prêt à l'envoyer à R dès que celui-ci le préviendra.
next	Directif	S veut la réponse suivante à une précédente demande qu'il a envoyée à R.
rest	Directif	S veut l'ensemble des réponses manquantes à une précédente demande qu'il a envoyée à R.
discard	Directif	S ne veut pas de la fin des réponses à une précédente demande qu'il a envoyée à R.

Table 1.2 **Liste des performatifs KQML [KQML97]**

5.3.3. FIPA ACL (Agent Communication Language)

Ces dernières années, KQML semble perdre du terrain au profit d'un autre langage plus riche sémantiquement ACL (pour Agent Communication Language). Un langage mis de l'avant par la FIPA (pour Foundation for Intelligent Physical Agents) qui s'occupe de standardiser les communications entre agents. ACL est basé également sur la théorie du langage et a bénéficié grandement des résultats de recherche de KQML. Si toutefois, les deux langages se rapprochent au niveau des actes du langage, il n'en ait rien au niveau de la sémantique et il semble qu'un grand soin a été apporté au niveau de ACL tant au niveau de certains protocoles qu'au niveau de la sémantique des actes eux-mêmes [**Cha01**].

La sémantique formelle de FIPA ACL se compose en cinq niveaux [O'Bri98] :

- ***Le protocole***, qui décrit les règles sociales des dialogues entre les agents ;

- ***Les actes communicatifs (AC),*** qui définit le type de communication entre les agents (par exemple, la demande, la confirmation, etc.) ;
- ***La méta-information*** concernant le message (l'identification d'un agent émetteur et d'un agent récepteur, le contexte, etc.) ;
- ***Le langage du contenu***, qui décrit la grammaire et la sémantique associée, utilisées pour exprimer le contenu d'un message ;
- ***L'ontologie***, qui définit le vocabulaire et les significations des termes et des concepts, employées dans le contenu.

Elément	Signification
performatif	le type de l'acte communicatif
sender	l'émetteur du message
receiver	le destinataire du message
reply-to	le participant à l'acte de communication
content	le contenu du message (l'information transportée par le performatif)
language	le langage dans lequel le contenu est représenté
encoding	décrit le mode d'encodage du contenu du message
ontology	le nom de l'ontologie utilisé pour donner un sens aux termes utilisés dans le content
protocol	le nom du protocole d'interaction
conversation-id	l'identifiant de la conversation
reply-with	un identifiant du message, en vue d'une référence ultérieure
in-reply-to	il référence le message auquel l'agent est entrain de répondre (précisé par l'attribut reply-with dans le précédent message de l'émetteur)
reply-by	Un délai pour répondre au message

Table 1.3 **la structure d'un message FIPA-ACL [FIPA02]**

Voici un exemple de message *FIPA-ACL* qui est envoyé par l'agent *A* à l'agent *B* :

```
(inform
        : sender A
        : receiver B
        : reply-with : Dianrs
        : language : Prolog
        : ontology Ordinateur
        : content prix(Imprimante,1500 EUR))
        : conversation-id conv01
        : reply-by 10 min)
```

Figure 1.6 : **Exemple d'un message FIPA-ACL**

FIPA-ACL est superficiellement semblable à KQML. Sa syntaxe est identique à celle de KQML excepté différents noms pour quelques primitifs réservés. Le message FIPA ACL (figure ci-dessus) est représenté sous la forme d'une liste, contenant le type de l'acte communicatif (par exemple, INFORM, REQUEST), le nom de l'agent émetteur et celui de

l'agent récepteur, le contenu et le contexte du message, l'ontologie à utiliser pour interpréter ce contenu, et le protocole [FIPA00].

FIPA fournit également la description normative d'un ensemble de protocoles d'interaction de haut niveau, y compris la demande d'action (Table 1.4) [FIPA02].

FIPA ACL peut être considéré comme l'extension de KQML possédant deux langages différents. Le langage externe définit la signification intentionnelle du message. Le langage interne (ou le contenu) décrit l'expression à laquelle s'appliquent les croyances, les désirs et les intentions des agents, décrites dans le primitif de communication.

Différemment de KQML, FIPA ACL est basé sur la sémantique logique de la communication. Ceci facilite la description des formats de la communication. Cependant, les agents ne possèdent pas toujours les capacités logiques nécessaires. Il est à noter que la sémantique de FIPA ACL est basée en grande partie sur les croyances des agents, qui peuvent être inconnues pour les autres agents ou bien qui changent rapidement. Ceci rend beaucoup des conditions de communication préalables, difficiles à contrôler.

Une autre grande différence entre FIPA ACL et KQML, concerne les actes communicatifs. FIPA ACL contient un ensemble des actes communicatifs normatifs, qui peuvent être primitifs ou composés. Les nouveaux actes communicatifs ne peuvent être définis qu'en combinant les actes existants et en utilisant les opérateurs prédéfinis. Ceci permet de maintenir l'intégrité sémantique du langage.

Performatif	**Type d'action**	**Description**
accept-proposal	Directif	S accepte une proposition d'action envoyée précédemment
agree	Engageant	S accepte de réaliser une action demandée.
cancel	Engageant	S informe qu'il n'a plus besoin qu'un autre agent réalise une action demandée précédemment.
call-for-proposal	Directif	S fait un appel d'offre pour une action.
confirm	Directif	S informe qu'une proposition est vraie (ce type de message est envoyé quand S pense que son interlocuteur est incertain à propos de la valeur de la proposition).
disconfirm	Directif	S informe qu'une proposition est fausse.
failure	Assertif	S informe qu'il a essayé de réaliser un action mais que cette tentative a échoué.
inform	Assertif	S informe de la valeur d'un proposition.
inform-if	Assertif	S informe qu'une proposition est vraie.
inform-ref	Assertif	S informe R de la valeur d'un objet.
not-understood	Assertif	S informe R qu'il n'a pas compris une des actions (éventuellement communicative) de R.
propagate	Directif	S demande à R de propager un message à un ensemble d'agents défini par S.
propose	Engageant	S propose de réaliser une certaine action
Proxy	Directif	S demande à R de propager un message à un ensemble

		d'agents sélectionnés par R.
query-if	Directif	S demande si une proposition est vraie ou non.
query-ref	Directif	S demande la valeur d'un objet.
refuse	Engageant	S refuse de réaliser une action.
reject-proposal	Directif	S refuse une proposition d'action.
request	Directif	S demande à R de réaliser une action.
request-when	Directif	S demande à R de réaliser une action quand une certaine proposition deviendra vraie.
request-whenever	Directif	S demande à R de réaliser une action à chaque fois qu'une certaine proposition deviendra vraie.
suscribe	Directif	S demande à R de le prévenir des changements de valeur d'un objet.

Table 1.4 **Liste des performatives FIPA ACL [FIPA02]**

6. La communication comme problématique d'évaluation :

Selon [Hus10], La plupart des travaux d'évaluation dans le domaine des SMA traitent de la communication comme critère de comparaison entre les différents systèmes [Myl02a, Cer02, Dav02, Jur06a, Bin04]. La liste des travaux montre l'importance de la communication comme critère d'évaluation. Des caractéristiques plus complexes (coopération, coordination, etc) sont évaluées sur la base de sa mesure de la communication comme étant critère de base.

La communication peut être évaluée selon 2 niveaux d'abstractions [Hus10] :

- "Micro level": l'étude de la communication se fait au niveau des agents sans prendre compte ce qui se passe au niveau global (niveau SMA).
- "Macro level" : dans ce cas, l'étude de la communication se fait sur la globalité du système. L'étude est réalisée sur l'ensemble des agents et l'environnement.

7. Approches d'évaluation de la communication

Une nouvelle approche d'évaluation de la communication a été proposée par Hussein Djoumaa dans [Hus10]. Elle consiste à étudier la communication suivant trois approches :

7.1 Approche normale

Le but de l'approche normale est de comparer l'évolution des communications durant l'exécution, et de mesurer le nombre des messages échangés entre les agents. Cette approche est utilisée dans les évaluations réalisées dans les SMA. Elle est intéressante pour identifier les liens entre les différents agents et la comparaison entre l'utilisation des différents canaux de communications entre agents.

7.2 Approche basée type

Cette approche consiste à évaluer la communication selon le type de messages reçus par un agent, en traitant la quantité des informations portées par les messages, et de diviser

l'ensemble des messages reçus par un agent en des sous-ensembles ayant le même type, et donc le même effet sur l'agent. Ensuite, le poids est associé à un message selon son type. La distinction entre les types des messages dépend de l'application des SMA à évaluer.

Par exemple, Gaspar [Gas91] a proposé quatre types des messages échangés: *present*, *request*, *answer*, et *inform*. Ces quatre types sont distingués selon le changement des comportements de l'expéditeur ou du destinataire.

- "*Request*" implique un changement de coté de l'expéditeur, dans l'attente de la réponse.
- "*answer*" : implique un changement d'état du récepteur, pas plus d'attente pour la réponse.
- "*present*" inclut un éventuel changement de coté de l'expéditeur et/ou du récepteur. Typiquement, un "present" permet de s'introduire face aux autres agents au début de l'exécution.
- "*Inform*" ne comporte pas de changement à la fois pour l'émetteur et le récepteur. Il pourrait engendrer d'autres "inform", et éventuellement des réponses.

Cette solution proposée par Gaspar est qualifiée comme statique, dans la mesure où le poids d'un message est affecté en se basant sur la conception du SMA, toutefois, elle est idéale dans le cas où deux messages du même type ont des effets équivalents sur l'agent indépendamment du temps.

Le problème principal de cette approche est que la quantité des informations portées par une unité de communication (message) est considérée comme statique et fixe selon le type. Dans la majorité des systèmes multi-agents, un message peut avoir deux effets différents à deux instants différents.

7.3 Approche basée poids des messages

Avant d'entamer cette approche de communication, la notion de communication pertinente est définie. Une communication pertinente est un message qui change les connaissances d'un agent ou déclenche une action [Hus10].

Donc, la solution proposée par l'approche basée type (des messages reçues et s'avérant non pertinents pour l'agent) est de considérer plutôt les communications pertinentes que la quantité totale des communications, par conséquent, cette approche consiste à associer le poids au message reçu selon les résultats de son traitement.

Cette approche consiste à associer le poids au message reçu selon les résultats de son traitement. Le traitement d'un message est divisé en 2 fonctions : décision et mémorisation.

La fonction de mémorisation associe une valeur à la variation des connaissances causée par la réception d'un message, d'autre part la fonction de décision associe une valeur à l'action déclenchée (le résultat de l'étape de décision). Les types des actions possibles sont définis. Un type d'action a un poids. Ensuite, la valeur de la fonction de décision est considérée comme la somme des poids des actions déclenchées [Hus10].

8. Plates-formes de développement des SMAs

Le processus de développement d'un système multi agents a été débattu via la présentation des méthodologies de conception de ces systèmes et les plateformes permettant de les mettre en œuvre.

Une plate-forme multi-agent est un ensemble d'outils nécessaire à la construction et à la mise en service d'agents au sein d'un environnement spécifique. Ces outils peuvent servir également à l'analyse et au test du SMA ainsi créé. Ces outils peuvent être sous la forme d'environnement de programmation (API) et d'applications permettant d'aider le développeur, de concevoir et réaliser leurs applications sans perdre de temps à réaliser des fonctions de base pour la création et l'interaction entre agents.

Il existe un nombre important d'environnements de développement des applications orientées agents: il y a aussi bien des produits commerciaux que des logiciels dans le domaine public. Parmi les plates-formes fournies comme logiciels libres, nous citons: JADE [Bel99], ZEUS [Nwa99, Ter98**]**, MADKIT [Gut01] et AgentFactory [Rem08] pour les agents cognitifs, et SWARM [Min96] pour les agents réactifs.

8.1 JADE

JADE (Java Agent Development Framework) est une plate-forme multi-agent développée en Java par le CSELT (Groupe de recherche de Gruppo Telecom, Italie).

JADE permet le développement de systèmes multi-agents et d'applications conformes aux normes FIPA [FIPA97, FIPA00, FIPA02]. JADE comprend trois composantes de base : un environnement d'exécution dans lequel les agents peuvent « vivre » ; une bibliothèque de classes que les programmeurs peuvent utiliser pour développer leurs agents ; et une suite d'outils graphiques permettant la gestion et l'administration des agents actifs, De plus, JADE peut être distribuée sur plusieurs hôtes

JADE possède trois modules principaux (nécessaire aux normes FIPA).

- DF « Director Facilitor » fournit un service de « pages jaunes » à la plate-forme ;
- ACC « Agent Communication Channel » gère la communication entre les agents ;
- AMS « Agent Management System » supervise l'enregistrement des agents, leur authentification, leur accès et l'utilisation du système.

Ces trois modules sont activés à chaque démarrage de la plate-forme. Ainsi, le but de JADE est de simplifier le développement des systèmes multi-agents pour réaliser des SMA interopérables.

8.2 ZEUS

Zeus est une plate-forme multi-agent conçue et réalisée par British Telecom (Agent Research Programme of BT Intelligent Research Laboratory) qui utilise une méthodologie appelée « role modeling » pour le développement de systèmes collaboratifs [Nwa99].

Les agents possèdent trois couches :

- La première couche est celle de la définition où l'agent est vu comme une entité autonome capable de raisonner en termes de ses croyances, ses ressources et de ses préférences.
- La seconde couche est celle de l'organisation. Dans celle-ci, il faut déterminer les relations entre les agents.
- La dernière couche est celle de la coordination

ZEUS est écrit dans le langage Java et elle est fondée sur les travaux de la FIPA. L'architecture des agents ZEUS est similaire à la majorité des agents collaboratifs. Elle regroupe principalement les composantes suivantes :

- une boîte aux lettres et un gestionnaire de messages qui analyse les messages de la boîte aux lettres et les transmet aux composantes appropriées ;
- un moteur de coordination ;
- un planificateur qui planifie les tâches de l'agent en fonction des décisions du moteur de coordination, des ressources disponibles et des spécifications des tâches ;
- plusieurs bases de données représentant les plans connus par l'agent, les ressources et l'ontologie utilisée ;
- un contrôleur d'exécution qui gère l'horloge locale de l'agent et les tâches actives.

L'environnement comporte trois bibliothèques : une avec des agents utilitaires, une avec des outils pour la construction des agents, et une avec des composants agents. Les caractéristiques des domaines d'applications de ZEUS ont été définies par les concepteurs, parmi ces caractéristiques, on peut mentionner :

- Chaque agent crée un plan qui nécessite un raisonnement explicite pour atteindre son but ;
- La résolution de problèmes nécessite une coopération entre agents ;

- Le rôle de chaque agent consiste à contrôler un système externe qui réalise une tâche du domaine d'application, la résolution de problème est ainsi effectuée par ce système externe et contrôlée par les agents.

8.3 MADKIT

MadKit (Multi-Agent Development Kit) [Fer09] est une plateforme de développement de systèmes multi agents développée par le Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM) de l'Université Montpellier II, destinée au développement et à l'exécution de systèmes multi-agents et plus particulièrement à des systèmes multi agents fondés sur des critères organisationnels groupes et rôles [Mic05].

MadKit est écrit en Java et fonctionne en mode distribué de manière transparente à partir d'une architecture "peer to peer" sans nécessiter de serveur dédié. Il est ainsi possible de faire communiquer des agents à distance sans avoir à se préoccuper des problèmes de communication qui sont gérés par la plateforme. Il est libre pour l'utilisation dans l'éducation.

MADKIT est fondé sur le modèle organisationnel ALAADIN (Figure 1.7). Il utilise un moteur d'exécution où chaque agent est construit en partant d'un micronoyau. Chaque agent a un rôle et peut appartenir à un groupe. Il y a un environnement de développement graphique qui permet facilement la construction des applications.

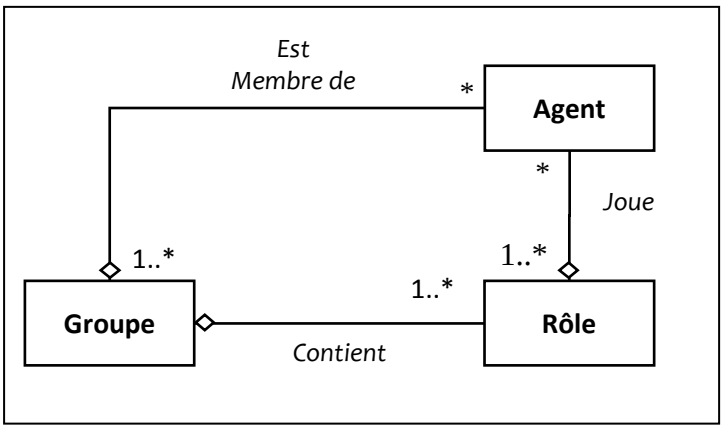


Figure 1.7 : **le modèle organisationnelle [Fer07]**

- **Agent :** L'agent est simplement décrit comme une entité autonome communicante qui joue des rôles au sein de différents groupes ;
- **Groupe :** Un groupe peut être vu comme un moyen d'identifier par regroupement un ensemble d'agents. Chaque agent peut être membre d'un ou plusieurs groupes ;
- **Rôle :** Le rôle est une représentation abstraite d'une fonction, d'un service ou d'une identification d'un agent au sein d'un groupe particulier. Chaque agent peut avoir plusieurs rôles, un même rôle peut être tenu par plusieurs agents, et les rôles sont locaux aux groupes.

MadKit présente les qualités suivantes [Fer09]:

- Simplicité de mise en œuvre et de prise en main ;
- Intégration très facile de la distribution au sein d'un réseau ;
- L'aspect pratique et l'utilisation efficace des concepts organisationnels pour créer différents types d'applications ;
- Hétérogénéité des applications et des types d'agents utilisables: on peut faire tourner sur MadKit aussi bien des applications utilisant des agents réactifs simples de type fourmis que des applications disposant d'agents cognitifs sophistiqués.

8.4 SWARM :

Swarm est un ensemble de bibliothèques facilitant l'implémentation des modèles basés agents, l'environnement offre un ensemble de bibliothèques qui permettent l'implémentation des systèmes multi-agent avec un grand nombre d'agents simples qui interagissent dans le même environnement [Min96].

Le simulateur Swarm est un ensemble de bibliothèques portables que l'on peut utiliser dans des environnements variés. Ce simulateur est gratuit et facilement téléchargeable sur : http://www.swarm.org

Dans Swarm, l'agent est vu comme une extension du processus. L'originalité de ce système provient de la notion d'activité associée à un swarm ou essaim qui permet un contrôle du déroulement des actions effectuées par les agents. Un swarm est le composant de base du système, L'inspiration du modèle d'agent utilisé vient de la vie artificielle, SWARM est l'outil privilégié de la communauté américaine et des chercheurs en vie artificielle.

Dans une application de ce type, l'utilisateur spécifie le comportement d'un ensemble d'agents sous la forme d'un ensemble d'actions déclenchées selon un calendrier. Il est possible de créer plusieurs swarm liés entre eux par une hiérarchie.

8.5 AgentFactory (AF):

Agent factory est une plateforme open source basée sur le langage java pour la création et le déploiement des applications orientées agent. Elle est maintenue par les chercheurs du laboratoire PRISM (**P**ractice and **R**esearch in **I**ntelligent **S**ystems and **M**edia) à l'École d'informatique à l'Université College à Dublin. Elle a été soutenu avec succès dans divers domaine de recherche : l'informatique mobile, la réalité augmenté, les réseaux des capteurs distribués, etc [Rem08].

Il est structuré autour d'un environnement d'exécution compatible avec les normes de la FIPA, qui offre des services de base (la gestion des agents, la messagerie et les pages jaunes) et un système basé sur les plug-ins pour le développement des agents. Le kit de

développement AFAPL2 fournit un langage et un moteur de raisonnement pour les agents BDI, mais les autres modèles d'agents peuvent être intégrés dans la plateforme comme le kit de développement RMA (Reactive Message Agent).

Le point fort d'AgentFactory est son intégration avec les environnements de développement intégrés (IDE) comme NetBeans et Eclipse. Nous avons opté, dans le cadre de notre travail, pour cette plate-forme multi-agent pour développer notre architecture. Ce choix est justifié par le fait que AgentFactory dispose d'un profileur orienté agent spécifique à la plateforme qui prend en charge les facteurs liés aux différentes spécificités des SMA(s), tels que : la quantification de la communication (nombre de messages échangé, l'impact du message sur le comportement des agents, etc), ce qui nous permet d'analyser les performances de n'importe quel SMA développé sous AgentFactory, ainsi que les facteurs ordinaires liés aux performances des programmes JAVA tel que : le temps CPU, identification des portions de code qui utilisent inutilement le temps CPU et des ressources.

Le profileur AgentSpotter est un outil conçu spécifiquement pour la collecte et la représentation des informations de profilage sur les SMAs [Din08a] et [Din08b] développés sous la plateforme agentFactory.

9. Conclusion

Nous avons présenté dans ce chapitre, un aperçu sur le paradigme multi-agent, tout en mettant l'accent sur le rôle déterminant de l'interaction dans les SMA, plus précisément, nous avons concentré sur la communication, qui présente un critère d'évaluation important dans le domaine d'évaluation des SMAs. Dans ce chapitre, nous avons également présenté les principaux concepts utilisés tout au long de ce mémoire. Ces définitions vont permettre au lecteur de comprendre le sens précis dans lequel nous les avons utilisés. Un bref aperçu a été aussi présenté sur les plateformes de développement des systèmes multi agent dans le but de bien positionner la plateforme sélectionnée dans notre travail.

Chapitre II

Refactoring Orienté Aspect et AspectJ

1. Introduction

De nos jours, les développeurs cherchent à réduire le coût et les efforts nécessaires au maintien des logiciels qui acquièrent une complexité de plus en plus croissante. L'utilisation de différents paradigmes a réduit ce coût et a permis l'évolution des systèmes impliquant l'application d'un minimum d'effort [Lou10]. Cependant, l'activité de maintenance pour les systèmes logiciels complexes n'est, en général, pas encore maîtrisée. Dans la littérature, plusieurs techniques sont utilisées dans le domaine de maintenance de logiciel dans le but d'améliorer sa qualité. Parmi ces techniques nous citons le *Refactoring*, une technique relativement nouvelle qui sert à restructurer le code source de logiciel pour améliorer sa qualité notamment sur les plans extensibilité, modularité, réutilisabilité, complexité et maintenabilité.

L'un des paradigmes utilisés pour faciliter le développement des systèmes et réduire leurs coûts est le paradigme de développement orienté aspect de logiciels (*Aspect Oriented Software Developement, AOSD*) [Rob05]. Ce paradigme apporte une simplicité au processus d'évolution des logiciels. En effet, l'AOSD fournit des mécanismes pour séparer les différentes préoccupations du système durant le développement du logiciel. De plus, le paradigme AOSD peut être considéré comme une approche qui permet d'adresser d'une manière différente la résolution de certains problèmes liés à l'évolution des logiciels.

La programmation orientée aspect (POA) est un nouveau paradigme de programmation qui trouve ses racines en 1996, suite aux travaux de Gregor Kiczales et de son équipe au Centre de Recherche Xerox à Palo Alto [Lop98b, XER00]. La POA est donc une technologie relativement jeune, puisque les premiers outils destinés à son utilisation ne sont apparus qu'à la fin des années 90.

Il est à noter que la programmation par aspects est indépendante de la programmation par objets. En effet, il est possible d'utiliser la programmation par aspects

conjointement à d'autres paradigmes de programmation (fonctionnelle, impérative, etc) [Ter04].

Dans ce chapitre nous rappellerons d'abord les problèmes résolus par la programmation orientée aspect. Puis nous introduisons les concepts de la programmation orientée aspect, nous présentons aussi le langage AspectJ ainsi que ses concepts. Nous introduirons aussi quelques exemples choisis de programmation par aspects écrits principalement avec le langage AspectJ vu comme une extension du langage Java. Ces exemples seront l'occasion de caractériser le modèle d'aspects sous-jacent en précisant les notions de points de jonction, de coupes, d'introductions.

2. Le Refactoring

Selon Fowler [Fow99], la définition du refactoring est : "***Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure***". Le refactoring est automatisé par l'implémentation de certains outils de refactoring. Il existe des outils de refactoring pour les langages objet et aussi pour les langages fonctionnels.

Lors de l'application du refactoring sur les programmes orienté-aspect, nous sommes intéressés par les questions suivantes :

1. Est-ce que les techniques de refactoring orientés objets existants (tels que ceux qui ont été proposées par Fowler [Fow9]), peuvent être adoptées par les programmes orientés aspects, si non, comment devons-nous faire pour appliquer ces refactorings aux programmes orientés aspect?
2. Y a-t-il de nouvelles techniques de refactoring qui sont propres à la programmation par aspect mais différentes des techniques de refactoring orienté objet existantes.
3. Comment soutenir le refactoring automatique des programmes orienté-aspect.

2.1. Préservation du comportement d'un programme

Par définition, un refactoring ne devrait pas modifier le comportement ou les fonctionnalités d'un logiciel. La première définition de la conservation du comportement a été donnée par [Opd92]. Il stipule que, pour les mêmes valeurs d'entrée et l'ensemble des valeurs de sortie résultant doivent être le même avant et après l'application du refactoring. Mais ce n'est pas une contrainte assez forte dans certains domaines d'application. Par exemple, elle ne tient pas compte d'autres critères comportementaux importants tels que les contraintes de temps, de mémoire, de consommation d'énergie, de synchronisation entre processus distribués, etc. Dans les applications où le traitement de données joue un rôle prépondérant, une autre contrainte essentielle est la préservation de la cohérence de ces données.

Au niveau de code source, la programmation par contrat [Mey92] est souvent proposée pour préserver certains aspects du comportement, en spécifiant des pré-conditions, post-conditions et invariants sur les transformations.

2.2. Refactoring et Maintenance :

Le refactoring est une activité de maintenance d'un genre particulier. Il s'agit non pas d'une tâche de correction des anomalies ou de réalisation de petites évolutions pour les utilisateurs mais d'une activité plus proche de la maintenance préventive et adaptative dans le sens où elle n'est pas directement visible de l'utilisateur.

Le refactoring consiste à modifier le code source de manière à améliorer sa qualité sans altérer son comportement du point de vue de ses utilisateurs. Son rôle concerne essentiellement la pérennisation de l'existant, la réduction des coûts de maintenance et l'amélioration de la qualité de service au sens technique du terme (performance et fiabilité). Le refactoring est, en ce sens, différent de la maintenance corrective et évolutive, [Jea05] qui modifie directement le comportement du logiciel, respectivement pour corriger un bogue ou ajouter ou améliorer des fonctionnalités.

Par ailleurs, la maintenance a généralement une vision à court terme de l'évolution du logiciel puisqu'il s'agit de répondre à l'urgence et d'être réactif. Le refactoring est une démarche qui vise à pallier activement les problèmes de l'évolution logicielle, notamment celui de l'érosion. Il s'agit donc d'une activité au long court, devant être dotée d'une feuille de route continuellement mise à jour au gré des changements [Jea05].

Idéalement, le refactoring doit être envisagé comme un processus continu plutôt que comme un chantier devant être mené ponctuellement. Il peut être effectué en parallèle de la maintenance dès lors qu'il est compatible avec ses contraintes de réactivité. À l'instar de la correction d'erreur, plus un refactoring est effectué tôt moins il coûte cher.

2.3. Processus de refactoring :

À l'instar de tout projet logiciel, le refactoring [Jea05] s'inscrit dans un processus comportant les étapes fondamentales suivantes :

1. La préparation, qui consiste à mettre en place, si ce n'est déjà fait, les outils permettant de gérer les changements et les valider.
2. L'analyse du logiciel, qui consiste à identifier les éléments du logiciel nécessitant un refactoring et à sélectionner ceux qui sont pertinents.
3. La réalisation des opérations de refactoring.
4. La validation du refactoring, qui consiste à vérifier la non-régression du logiciel du point de vue des utilisateurs, tant pour les aspects fonctionnels que pour la qualité de service.

5. Si le logiciel en cours de refactoring subit des maintenances correctives en parallèle, une fusion de la version de maintenance et de la version de refactoring est nécessaire.

Le processus que nous venons de décrire considère implicitement que le refactoring est un projet en lui-même. Cependant, du fait de sa nature quelque peu ésotérique pour les utilisateurs, puisqu'il s'agit d'une série d'opérations techniques, le refactoring est rarement appliqué en dehors des projets d'évolution ou de maintenance des logiciels [Jea 05].

2.4. Outils de refactoring disponibles :

Parmi les premiers outils de refactoring dédiés pour les langages orientés objet, on trouve le *REFACTORING BROWSER* [Don97b] qui est utilisé pour effectuer des opérations de refactoring pour le langage SMALTALK. Un tel outil comprend les opérations de refactoring nécessaires pour manipuler la structure des programmes.

L'environnement de développement NetBeans contient aussi un outil de refactoring riche en opérations de refactoring et doté d'une API pour ces opérations que nous pouvons les utiliser pour appeler ces opérations directement à partir d'un programme et qui nous permet aussi de les composer dans des séquences de refactoring. Un tel outil gratuit offre l'opportunité de construire des transformations complexes comme l'introduction des patrons de conception, mais reste moins rapide en matière d'évolution par rapport à *IntelliJ IDEA* [IDEA] avec ses deux versions payantes et gratuites. IntelliJ IDEA intègre un outil de refactoring riche en catalogue d'opérations de refactoring avec une interaction dynamique avec les utilisateurs pour corriger les bogues ou bien prendre en compte des évolutions de certaines opérations.

D'autres outils de refactoring comportent une base théorique et formelle qui prouve que certaines opérations parmi les opérations qu'ils proposent sont correctes, comme le cas de *HaRe* l'outil de refactoring du langage Haskell [Hui05]. Un tel outil est efficace pour faire des compositions de refactoring pour changer l'architecture des programmes comme la transformation réversible proposée par Julien Cohen [Jul11] et qui permet de transformer un programme Haskell à structure types de données vers son équivalent à structure fonctions.

3. Problèmes adressés par le paradigme aspect :

Le but principal [Ass07] de l'ingénierie logicielle est d'obtenir la qualité logicielle. Les approches traditionnelles telles que les approches objet fournissent un support important de décomposition permettant d'offrir la réutilisation, et pour améliorer la qualité logicielle, grâce à l'encapsulation, le polymorphisme, l'héritage, etc. Cependant, quelques préoccupations, appelées *préoccupations transverses* dans la terminologie orientée aspect, entravent la réutilisation des modules.

3.1. Préoccupations transversales

Les préoccupations transversales (en anglais *crosscutting concerns*) sont les fonctionnalités dites non métiers. Un développeur est souvent confronté à ce genre de fonctionnalités lorsqu'il développe une application de grande envergure. Il existe en fait deux principaux symptômes liés aux préoccupations transversales: l'enchevêtrement du code et l'éparpillement du code [Keb10].

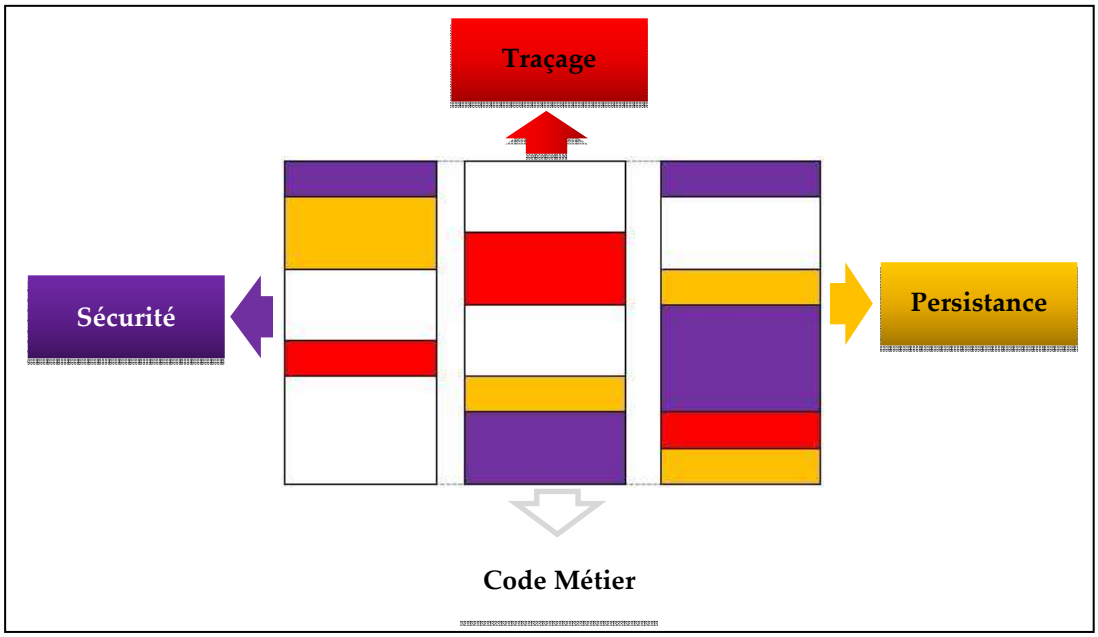


Figure 2.1 **Les exigences non-fonctionnelles traversent la modularisation fonctionnelle du système [Ram03]**

L'objectif de la POA est la séparation des préoccupations. La contrainte principale est la dispersion des préoccupations transversales à travers le code métier.

Dans cette section nous expliquons les problèmes résolus par la programmation orientée aspect :

- **Entrelacement de code**: L'enchevêtrement du code est provoqué quand un module est implémenté pour traiter plusieurs préoccupations en même temps. Un développeur a souvent affaire, pendant qu'il développe un module, à des préoccupations (Figure 2.2) telle que la logique métier, la gestion transactionnelle de la persistance, le logging, la sécurité, etc. Cela conduit à la présence simultanée d'éléments issus de chaque préoccupation et il en résulte un enchevêtrement du code

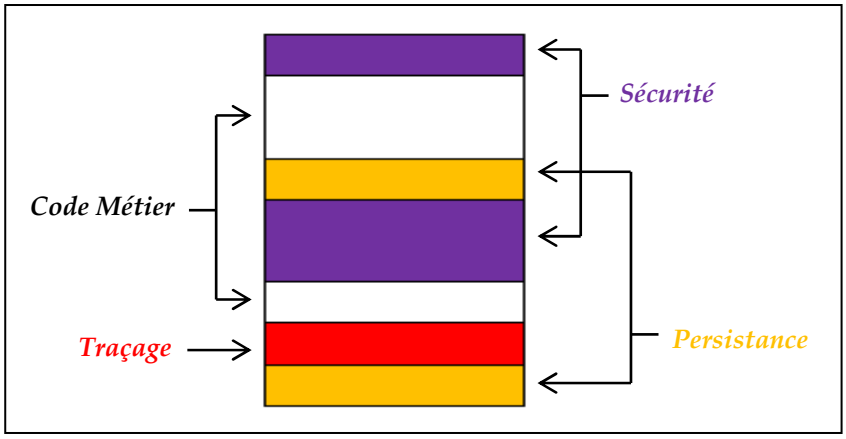


Figure 2.2 **Entrelacement de code [Ram03]**

- **Eparpillement du code**: L'éparpillement du code survient quand une préoccupation est implémentée dans plusieurs modules. Les préoccupations transversales sont, par définition, dispersées à travers **plusieurs** modules (Figure 2.3).

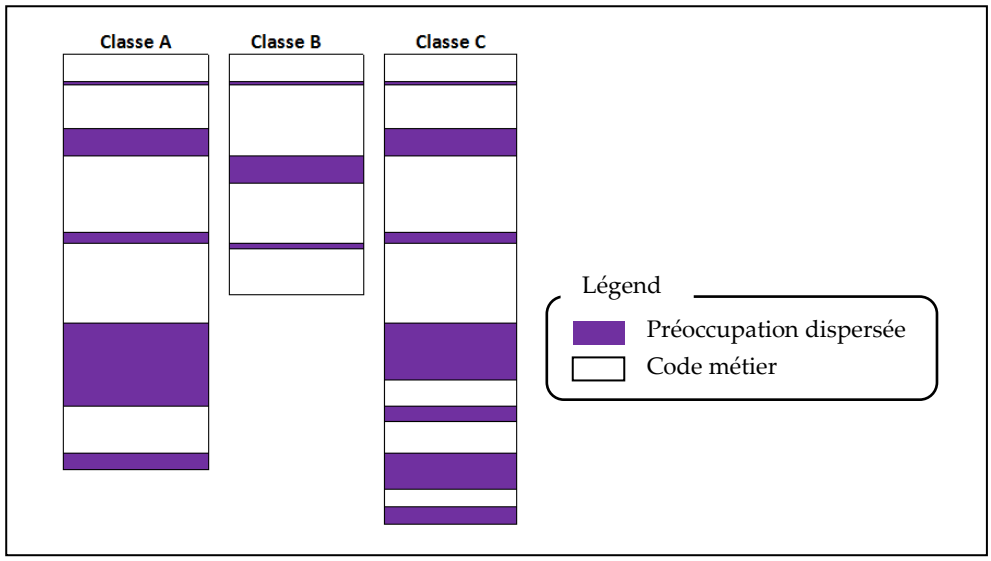


Figure 2.3 **Eparpillement du code d'une préoccupation [Mar 06]**

Les problèmes d'entrelacement et d'éparpillement engendrent, généralement, un code de pauvre qualité et difficile à réutiliser et à maintenir, ce qui fait de l'évolution du système logiciel une tâche fastidieuse.

C'est là que la programmation orientée aspect intervient en apportant des mécanismes à la fois simples à appréhender et puissants qui permettent de capturer des préoccupations transversales. En effet, l'orientée aspect procure une solution élégante aux problèmes d'enchevêtrement et d'éparpillement du code. Aujourd'hui, cette technique d'ingénierie logicielle s'affirme comme étant la prochaine étape pour le découpage des systèmes en offrant une nouvelle dimension pour la modularisation notamment avec la notion **d'aspect**.

En effet, parallèlement aux classes qui sont un support idéal pour modulariser les préoccupations métiers du système, les aspects sont un support pour capturer les préoccupations transversales. Dans une démarche orientée aspect, les préoccupations transversales peuvent évoluer indépendamment des préoccupations métier et vice-versa. Et afin que l'application finale prenne en compte toutes les préoccupations, le système passe par une étape dite de **tissage d'aspects**. Durant cette étape, les préoccupations transversales encapsulées dans les aspects vont être tissées ou intégrées dans les préoccupations métiers.

4. Etapes de Développement d'une Application Orientée Aspect

Le développement de logiciels en utilisant l'approche orientée aspect est similaire au développement de logiciels avec d'autres méthodologies (figure 2.4): identification des préoccupations, leur implémentation, et leur combinaison pour former le système final. La communauté des chercheurs de l'AOP définit ces trois étapes comme suit :

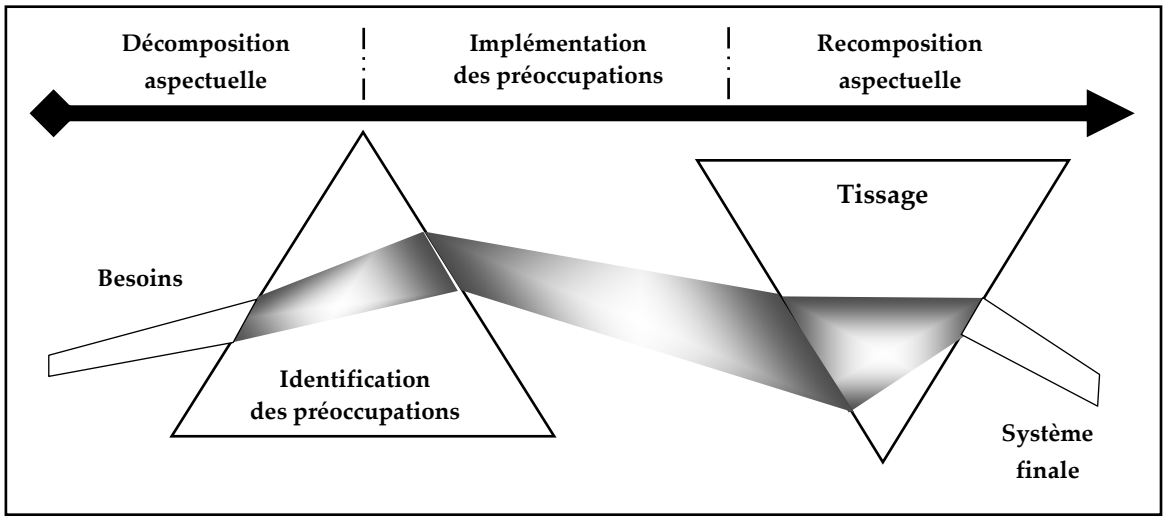


Figure 2.4 **Etapes de développement dans une méthodologie AOP [Ram 03]**

1) **Décomposition aspectuelle** : la décomposition des éléments du système. Les besoins sont ici décomposés pour identifier les préoccupations fonctionnelles et transversales,
2) **Implémentation des préoccupations** L'implémentation de chaque préoccupation. Chaque problématique sera codée séparément dans un composant ou un aspect. Dans un aspect, le programmeur définit aussi les règles d'intégration de l'aspect avec les composants concernés,
3) **Recomposition aspectuelle** Des règles de recompositions sont spécifiées en créant des unités appelées aspects. Le processus de recomposition, aussi connu sous le nom de tissage ou d'intégration, utilise ces informations pour composer le système final.

5. Concepts de la programmation orientée aspect

5.1. Aspect

Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une application [Ren04].

Les trois éléments principaux définis dans un aspect sont les coupes (*pointcuts*), les codes *advices* et le mécanisme d'introduction. Les coupes définissent *où* l'aspect doit être intégré dans une application et les codes *advices* définissent ce que fait l'aspect (le *quoi*). Le mécanisme d'introduction permet d'ajouter du contenu structurel dans une application. Nous allons définir ces concepts avec plus de détails dans les paragraphes suivants.

5.2. Point de jonction

Un point de jonction est un point dans le flot de contrôle d'un programme dans lequel un ou plusieurs aspects peuvent être appliqués. [Ren04].

Bien que la notion de point de jonction soit générale (potentiellement, chaque instruction d'un programme peut être un point de jonction), tous les points dans le flot de contrôle ne sont pas considérés comme utiles pour la POA. Les points de jonction sont groupés en fonction de leur type, et seulement un sous-ensemble de tous les types possibles de points de jonction est supporté par les langages orientés aspects. Les catégories suivantes décrivent les types de points de jonction communément rencontrés et qui sont indépendants de toute implémentation :

- **Les méthodes** : dans les langages orientés objet, l'exécution d'un programme peut être considérée comme une séquence d'appels et d'exécution de méthodes. Les appels et les exécutions de méthodes sont donc deux types de points de jonction couramment utilisés.
- **Les constructeurs** : les constructeurs sont principalement utilisés pour créer les instances des classes d'une application. Comme pour les méthodes, Les appels et les exécutions d'un constructeur correspondent à des types de points de jonction.
- **Les attributs** : les langages orientés aspect considèrent les opérations de lecture et d'écriture sur les attributs comme des types de points de jonction.
- **Les exceptions** : les exceptions sont lancées pour signaler une situation d'exécution anormale et elles sont capturées pour exécuter un traitement particulier. Ils sont tous les deux considérés par la plupart des langages orientés aspect comme des types de points de jonction.

En conclusion, tous les programmes, mêmes les plus simples, contiennent plusieurs points de jonction différents. La tâche du développeur est de sélectionner les points de jonction qui sont utiles pour implémenter un aspect donné. Cette sélection est réalisée grâce à la notion de coupe (pointcut), qui est présentée dans le paragraphe suivant.

5.2.1. Limites des points de jonction

Nous venons de voir certains types d'évènements qui peuvent constituer des points de jonction. Mais la POA peut connaitre des limites quant à la granularité correspondante aux points de jonction. En effet, les instructions (*if, while, for, switch*, ...) sont jugées trop fines par certains outils de la POA et ne sont donc pas interceptées. C'est par exemple le cas pour le langage AspectJ que nous verrons à la deuxième partie de ce chapitre.

5.3. Coupe

La notion de point de jonction n'est pas suffisante à elle seule pour définir quels points de jonction sont pertinents pour un aspect donné. On a besoin d'une autre notion pour décrire les points de jonction. Cette notion est la coupe.

> ***Une coupe sélectionne un ensemble de points de jonction*** [Ren04].

Un langage de programmation orienté aspect doit fournir au développeur une structure syntaxique permettant de déclarer une coupe. Cependant, chaque langage définit sa propre syntaxe.

5.4. Code advice

Un aspect définit une fonctionnalité transversale. Comme nous l'avons vu, il spécifie le caractère transversal grâce aux coupes. La fonctionnalité est, quant à elle, spécifiée par des codes advices [2].

> ***Un code advice est un bloc de code définissant le comportement d'un aspect*** [Ren04].

Un code advice définit donc un bloc de code qui va venir se greffer sur les points de jonction définis par la coupe à laquelle il est lié. Un aspect nécessite souvent plusieurs codes advices pour caractériser la fonctionnalité transversale.

Un code advice peut être exécuté selon trois modes : avant, après, ou autour d'un point de jonction. Lorsqu'il est exécuté autour du point de jonction, il peut carrément remplacer l'exécution de ce dernier, ou bien lui redonner le contrôle.

5.5. Mécanisme d'introduction

> ***Le mécanisme d'introduction est un mécanisme d'extension permettant d'introduire de nouveaux éléments structuraux au code d'une application*** [Ren04].

Le mécanisme d'introduction permet d'étendre la structure d'une application et non pas le comportement de cette dernière. En effet, le mécanisme d'introduction ne s'appuie

[2] *Le terme advice ne connait pas de traduction reconnue par la communauté POA francophone*

pas sur la notion de coupe mais va opérer sur des emplacements bien définis dans le programme.

On peut dire que le mécanisme d'introduction est pour l'orientée aspect ce que l'héritage est pour l'orientée objet, à la différence de l'héritage en POO, l'introduction ne peut étendre les classes qu'en rajoutant de nouveaux éléments, il n'est donc pas possible de redéfinir une méthode, par exemple. Il est aussi important de remarquer que tous les éléments introduits ne peuvent être utilisés que par des aspects. En effet, l'application ne peut pas savoir à l'avance qu'un élément sera ajouté à une classe et donc en faire usage.

5.6. Tissage

> ***Le tissage (weaving) est le processus qui prend en entrée un ensemble d'aspects et une application de base et fournit en sortie une application dont le comportement et la structure sont étendus par les aspects*** [Ren04].

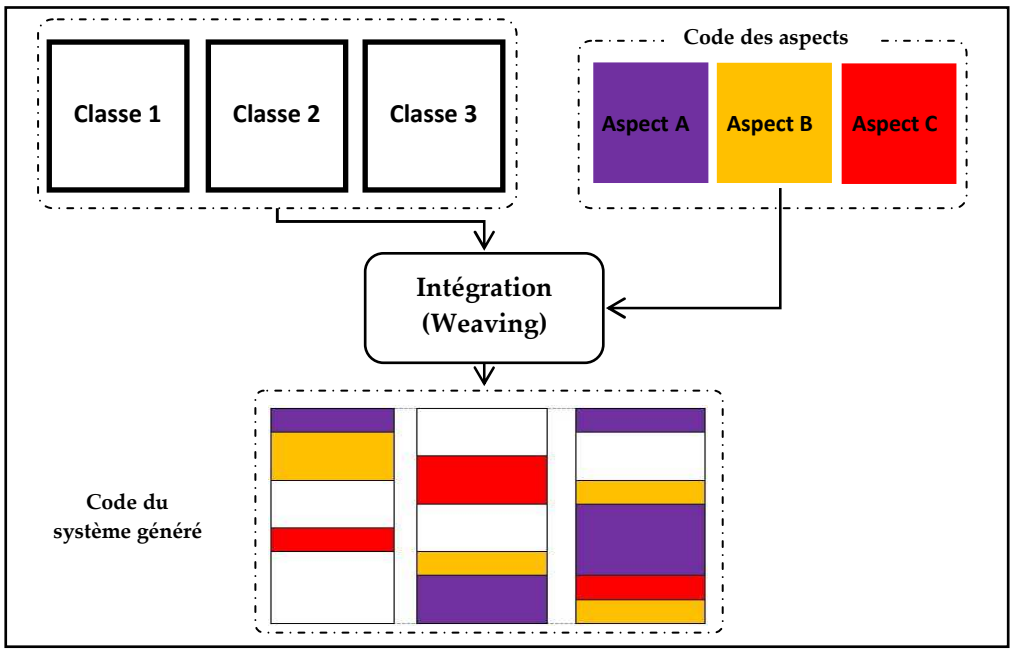


Figure 2.5 **Tissage des aspects [Bal 02]**

Une application orientée aspect contient des classes et des aspects. L'opération qui prends en entrée les classes et les aspects et produit une application qui intègre les fonctionnalités des classes et des aspects est connue sous le nom de tissage d'aspect. Le programme qui réalise cette opération est appelé tisseur d'aspects ou bien tisseur tout court (Figure 2.5).

6. Ordonnancement d'aspects

Nous avons vu qu'un aspect greffe des codes advices sur les points de jonction définis par sa coupe. Dès lors, il se peut que plusieurs aspects aient des points de jonction en

communs dans leur coupe. Or dans certains cas, il est nécessaire qu'un code advice soit exécuté avant un autre (s'il y a des dépendances entre aspects) [Bal02].

Les outils de la POA fournissent à cet effet des techniques permettant de spécifier l'ordre dans lequel doivent être tissés les aspects. Si le programmeur ne spécifie pas d'ordre, aucune garantie n'est donnée en général quant à l'ordre résultant après le tissage. Cependant, certains outils tels qu'AspectJ proposent des règles implicites d'ordonnancement d'aspects.

7. Type de refactoring orienté aspect :

Le refactoring orienté aspect diffère de refactoring classique en ce qui concerne les constructeurs de la POA impliqués, il peut être divisé en trois groupes [Jan06]:

1. *Refactoring orienté objet des aspects courants* (aspect-aware OO refactorings) : qui représente une extension du refactoring OO traditionnel. Son but est de ne pas briser les constructeurs de la programmation orientée aspect, par exemple l'application du refactoring *renameMethod* à une méthode nécessite une mise à jour à toutes les références de cette méthode non seulement dans le code OO, mais également dans le code aspect.
2. *Refactoring des constructeurs de la programmation orientée aspect*: est un refactoring impliquant explicitement les éléments du programme de l'AOP, exemple : extractPointCut, extractAdvice, etc.
3. *Refactoring des préoccupations transversales*: ce type de refactoring vise à transformer et migrer les préoccupations transversales non modularisées vers des aspects.

Nous allons, dans le cadre de ce mémoire, adopter le troisième type de refactoring.

8. AspectJ

8.1. Historique et origine

L'histoire d'AspectJ est étroitement liée à celle de la programmation orientée aspect. Un premier prototype d'AspectJ a été réalisé en 1998 [Lop98, XER00]. Et depuis, plusieurs versions d'AspectJ ont vu le jour, et chacune apportait de nouvelles fonctionnalités et corrigeait les bugs de la précédente. La première version officielle d'AspectJ, désignée AspectJ 1.0 a été réalisée en novembre 2001. Durant cette année, l'POA a été complètement reconnue par la communauté informatique mondiale, et une édition spéciale du journal *Communications of the ACM* a été dédiée à l'AOP.

En décembre 2002, le projet AspectJ a quitté XEROX PARC et a rejoint la communauté open-source Eclipse, et depuis, le plugin Eclipse *AspectJ Developpement Tools (AJDT)* est développé. Ce plugin intègre AspectJ et permet d'écrire, de compiler et

d'exécuter des programmes orientés aspects dans l'environnement de développement Eclipse.

8.2. Présentation générale

AspectJ est un langage orienté aspect pour Java qui est tissé à la compilation. Les aspects sont tissés directement dans le bytecode de l'application. On obtient donc un ensemble de fichiers *.class* contenant l'application aspectisée et compatibles avec la machine virtuelle Java. Sa plus grande force réside dans le fait qu'il est issu des travaux de la même équipe à l'origine de l'orientée aspect [Ter04].

AspectJ permet de déclarer des aspects, des coupes, des codes advices et des introductions. Il bénéficie d'une multitude d'outils de débogage, d'environnement de développements et de visualisateurs d'aspects. AspectJ permet de définir deux types de transversalités avec les classes de base : transversalité statique (static crosscutting) qui consiste à augmenter la **structure** des classes avec le mécanisme d'introduction fournit par aspectJ et aussi d'ajouter des liens entre des classes comme l'héritage et l'implémentation d'interfaces.

Quant à elle, la transversalité dynamique (dynamic crosscutting) consiste à augmenter le *comportement* des classes. Les coupes servent à sélectionner des points précis dans les classes. Et les advices iront se greffer avant, après ou autour de ces points afin d'étendre leur comportement.

8.3. Point de jonction

Un point de jonction est n'importe quel point d'exécution dans un système. Dans cette section, nous verrons les mots-clés disponibles dans AspectJ pour chacun des types de points de jonction et nous détaillerons ensuite la manière de définir les expressions passées en paramètre à ceux-ci.

8.3.1. Type de point de jonction AspectJ

Nous allons voir dans la table ci-après les types de points de jonction proposés par AspectJ pour les identifier.

Point de jonction	Description
Method call	Quand une méthode est appelée
Method execution	Quand le corps d'une méthode est exécuté
Constructor call	Quand un constructeur est appelé
Constructor execution	Quand le corps d'un constructeur est executé
Static initializer execution	Quand l'initialisation statique d'une classe est exécutée
Object pre-initialization	Avant l'initialisation de l'objet
Object initialization	Quand l'initialisation d'un objet est exécutée
Field reference	Quand un attribut non-constant d'une classe est référencé

Field set	Quand un attribut d'une classe est modifié
Handler execution	Quand un traitement d'une exception est exécuté
Advice execution	Quand le code d'un advice est exécuté

Table 2.1 **Points de jonctions disponibles dans AspectJ [Ram03]**

Dans AspectJ, tous les points de jonction ont un contexte associé à eux. Par exemple, le contexte d'un point de jonction correspondant à un appel de méthode contient l'objet appelant, l'objet appelé, et les arguments de la méthode. De la même manière, le contexte d'un point de jonction correspondant au traitement d'une exception contient l'objet courant, et l'exception lancée.

8.4. Définition des profils

À l'exception d'*adviceexecution*, tous les mots-clés que nous avons vus requièrent un paramètre. Ce paramètre est une expression qui permet, en spécifiant un profil, de filtrer l'ensemble de points de jonction donné par le mot-clé. Par exemple, parmi tous les points de jonction de type appel de méthode, nous ne souhaitons garder que les appels vers une méthode dont le nom est ***x***.

L'expression précisant le profil des éléments qui nous intéresse peut faire usage de quantificateurs (appelés wildcards) afin d'introduire de la généricité dans les profils sélectionnés. Ces wildcards sont : *****, **..** et + (Table 2.2.).

Wildcards	Description
*	Le symbole : * peut être utilisé pour remplacer des noms de classes, de méthodes et d'attributs. Il peut soit remplacer l'entièreté du nom, soit une partie de celui-ci. **Exemples** : *public void edu.ulb.Class.*(String)* représente toutes les méthodes publiques de la classe *Class* (dans le package *edu.ulb*) prenant un String en paramètre et ne retournant rien.
..	Le symbole : .. permet de prendre en compte le polymorphisme des méthodes. En effet, les paramètres d'une méthode peuvent être omis à l'aide de ce symbole. **Exemples :** *public void edu.ulb.Class.*(..)* représente toutes les méthodes publiques de la classe *Class* (dans le package edu.ulb) ne retournant rien. Remarque : Le wildcard .. peut également être utilisé pour introduire de la généricité dans la hiérarchie des packages. Exemples : * *edu..*.*(..)* représente toutes les méthodes de toutes les classes de n'importe quel package de la hiérarchie *edu*.
+	Le symbole + est utilisé en tant qu'opérateur de sous-typage. En effet, mis en suffixe d'un nom de classe, il permet d'identifier l'ensemble contenant cette classe et toutes ses sous-classes. S'il est mis à la suite d'un

	nom d'interface, il identifie toutes les classes implémentant l'interface. Exemple : *public void edu.ulb.Class+.*(..)* représente toutes les méthodes publiques de la classe *Class* (dans le package edu.ulb) et de toutes ses sous-classes, ne retournant rien.

Table 2.2 **Les wildcards dans AspectJ [Mar06]**

8.5. La Coupe

Dans AspectJ, les coupes correspondent à plusieurs points de jonctions dans le flot d'un programme. Par exemple, la coupe :

```
call(void Point.setX(int))
```

capture chaque point de jonction correspondant à un appel à la méthode *setX()* de la classe *Point* qui ne retourne aucune valeur et qui a comme paramètre un entier.

Une coupe peut être construite à partir d'autre coupes en utilisant les opérateurs and, or et not (respectivement &&, || et !). Par exemple, la coupe :

```
call(void Point.setX(int)) || call(void Point.setY(int))
```

Désigne les points de jonction correspondant à un appel à la méthode *Point.setX()* ou un appel à la méthode *Point.setY()*.

Les coupes peuvent identifier des points de jonction de différentes classes, en d'autres termes, elles peuvent être transverses aux classes. Par exemple, la coupe suivante:

```
call(void FigureElement.incrXY(int,int)) ||
call(void Point.setX(int))               ||
call(void Point.setY(int))               ||
call(void Line.setP1(Point))             ||
call(void Line.setP2(Point))
```

capture chaque point de jonction qui est un appel à une des cinq méthodes (la première méthode est une méthode d'interface).

Dans le dernier exemple, la coupe capture tous les points de jonction correspondant au mouvement d'un objet de type *FigureElement*. AspectJ permet de déclarer des coupes nommées avec le mot-clé *pointcut* afin qu'elles puissent être réutilisées sans avoir à les redéfinir. Les instructions suivantes déclarent une coupe nommée :

```
pointcut move():
call(void FigureElement.incrXY(int,int)) ||
call(void Point.setX(int))               ||
call(void Point.setY(int))               ||
call(void Line.setP1(Point))             ||
call(void Line.setP2(Point));
```

Ainsi, on peut appeler à n'importe quel moment la coupe nommée *move()*.

AspectJ offre aussi un mécanisme qui permet de spécifier des coupes en termes de propriétés de méthodes autres que leur nom exact. La façon la plus simple de le faire est d'utiliser les expressions régulières pour exprimer les champs de la signature des méthodes. Par exemple, la coupe suivante :

```
call(void Point.set*(..))
```

capture chaque point de jonction correspondant à un appel d'une méthode ne retournant aucun résultat et appartenant à la classe *Point* et commençant par la chaine *set*, quel que soit le type et le nombre de ses paramètres.

La coupe :

```
call(public * Line.* (..))
```

capture tous les appels aux méthodes publiques (*public*) de la classe *Line*.

Les exemples précédents ne font appel qu'à un seul type de coupe qui est l'appel de méthode (i.e. call). Il existe d'autres types de coupes dans AspectJ comme : l'exécution de méthode (ex : *execution(void Point.setX(..))*), ou l'accès aux attributs (ex : *get(Point.x)*), etc.

8.6. Advice

Les coupes capturent les points de jonction, mais elles ne font rien de plus. Pour implémenter un comportement transversal, on utilise les advices. En effet, un advice fait correspondre une coupe (i.e. un ensemble de points de jonction) à un bout de code exécuté à chaque point de jonction de cette coupe.

Un code advice est un bloc d'instruction associé à une coupe. Il est exécuté avant, après ou autour des points de jonction sélectionnés par la coupe qui lui est associée. AspectJ offre 3 types de codes advices : *before*, *after* et *around*.

Les codes advices de type *before* (respectivement *after*) permettent d'introduire un comportement avant (respectivement après) un point de jonction, l'exemple suivant écrit le string "*Avant un appel de méthode*" avant chaque appel de méthode de l'application.

```
/* définition de la coupe */
pointcut ex_coupe(): call(* *..*(..));

/* définition du code advice */
before(): ex_coupe() { System.out.println("Avant un appel de méthode"); }
```

Cependant les codes advices de type *after* se déclinent en 2 variantes : *after returning* et *after throwing* qui signifient respectivement après le retour d'une méthode sans exception et avec exception.

```
/* définition de la coupe */
pointcut ex_coupe(): call(* *..*(..));

/* définition du code advice */
after(): ex_coupe() { System.out.println("Après un appel de méthode"); }
```

Les codes advices de type *after returning* s'exécutent à chaque terminaison normale d'un des points de jonction de la coupe. Il est possible de récupérer la valeur retournée (si elle existe) en paramétrant le code advice. C'est ce qu'illustre le code suivant :

```
after() returning (double d): call(double Class.method(..)) {
        System.out.println("method(..); a retournée : " + d); }
```

Les codes advices de type *after throwing* s'exécutent à chaque terminaison anormale d'un des points de jonction de la coupe. Il est possible de récupérer l'exception lancée en paramétrant le code advice. C'est ce qu'illustre le code suivant :

```
after() throwing (Exception e): call(double Class.method(..)) {
        System.out.println("method(..) a lancée l'exception : " + e); }
```

Un advice de type *around,* quant à lui, définit un bloc d'instructions qui s'exécute autour d'un point de jonction. Il permet éventuellement de remplacer carrément l'exécution d'une méthode. AspectJ fournit la méthode *proceed()* qui permet de rendre le contrôle de l'exécution au point de jonction dans un code advice de type *around*. Le code suivant montre un exemple d'advice de type around:

```
Object around(): {
          System.out.println("avant le point de jonction");
         Object ret=proceed();
         System.out.println("après le point de jonction");
return ret; }
```

8.7. Filtrage

AspectJ propose des mots-clés identifiant des ensembles de points de jonction indépendants d'un type quelconque. Ils sont, la plupart du temps, utilisés à des fins de filtrage grâce à l'opération ensembliste d'intersection. Ils permettent ainsi d'affiner l'ensemble obtenu. Ces mots-clés sont détaillés dans la table suivante :

Mots clés	Description
withincode(methexpr)	Identifie l'ensemble des points de jonction se trouvant dans une méthode dont le profil vérifie *methexpr*.
within(typeexpr)	Identifie l'ensemble des points de jonction se trouvant dans une classe ou une interface dont le profil vérifie *typeexpr*.
this(typeexpr)	Identifie l'ensemble des points de jonction dont le profil de l'objet **source** vérifie *typeexpr*. Exemple : l'objet réalisant l'appel d'une méthode dans un point de jonction de type call.
target(typeexpr)	Identifie l'ensemble des points de jonction dont le profil de l'objet **destination** vérifie *typeexpr*. Exemple : l'objet sur lequel est appelée une méthode dans un point de jonction de type call.

Table 2.3 **Les mots-clés identifiant des ensembles de points de jonction [Mar06]**

Il existe encore deux mots-clés qui méritent plus d'attention. Il s'agit des mots-clés *cflow* et *cflowbelow*. Ces mots-clés permettent d'introduire des filtrages basés sur le flot de contrôle. Les opérateurs que nous avons vus jusqu'ici (withincode, within, this et target) peuvent être qualifiés d'opérateurs statiques. En effet, ils ne dépendent pas de la façon dont s'exécute le programme.

Les deux opérateurs *cflow* et *cflowbelow* prennent en compte la dynamique du programme. Nous allons, dans un premier temps, nous intéresser uniquement à cflow car cflowbelow a un comportement presque identique au précédent.

L'opérateur *cflow* prend une coupe en paramètre. Il identifie tous les points de jonction situés entre le moment où l'application passe par un des points de jonction de la coupe et le moment où l'application sort de ce point de jonction. L'ensemble de points de jonction fourni par *cflow* contient également les points de jonction de la coupe.

8.8. Déclaration inter-type

Les déclarations inter-types, dans AspectJ, correspondent au mécanisme d'introduction vu précédemment. Elles permettent de déclarer des membres dans des classes, ou de changer la relation d'héritages entre classes. Le code suivant montre quelques exemples

de déclaration inter-types. Il s'agit d'ajouter un attribut *name* de type *String* et deux méthodes *setName()* et *getName()* à la classe *Point* :

```
public String Point.name ;
public void Point.setName ( String name ) { this.name = name ; }
public String Point.getName ( ) { return name ; }
```

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

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

8.9. Aspect

Dans AspectJ un aspect contient tous les ingrédients nécessaires pour la définition d'une préoccupation transversale à savoir : les définitions de coupes, les codes advices et les déclarations inter-types. Il peut aussi éventuellement contenir des attributs et des méthodes qui lui sont propres. Le code suivant montre un exemple d'aspect réalisant la préoccupation de mise à jour d'affichage :

```
aspect UpdateDisplay {
        pointcut move(FigureElement elem) : target (elem) &&
( call ( void Line.setP1 (Point) ) ||
  call ( void Line.setP2 (Point) ) ||
  call ( void Point.setX (int) )    ||
  call ( void Point.setY (int) )    ||
  call ( void FigureElement.incrXY(int, int) ) ) ;
        after(FigureElement elem) returning : move ( elem ) {
                Display.update(elem); } }
```

8.10. Ordonnancement d'aspects

Il se peut qu'un point de jonction soit associé à plusieurs aspects. Dans certains cas, l'ordre dans lequel ces aspects sont exécutés est important. AspectJ fournit à cet effet le mot-clé *declare precedence* :

```
public aspect OrdreAspect {
    declare precedence: Aspect1, Aspect2;
}
```

Ce code indique que l'aspect *Aspect1* doit être appliqué avant l'aspect *Aspect2*. Il est également possible de faire usage de *wildcards* dans les noms des aspects.

Si aucune précision n'est fournie quant à l'ordre dans lequel tisser les aspects, AspectJ applique les règles implicites suivantes :

1. Les codes *advices* définis dans un sous-aspect ont la priorité sur ceux hérites.
2. Pour tout couple de codes *advices* défini dans le même aspect :
    a. Si l'un des deux codes *advices* est de type *after*, celui qui est défini en deuxième est prioritaire.
    b. Sinon, le code *advice* défini en premier est prioritaire.
2. L'ordre de tissage n'est pas spécifié pour deux codes *advices* définis dans deux aspects non liés par une relation d'héritage.

9. Conclusion

Dans ce chapitre, nous avons fait un survol sur la programmation orientée aspect. Après avoir expliqué les problèmes auxquels l'POA apporte des solutions efficaces, nous l'avons considérée uniquement en tant qu'extension de la programmation orientée objet et nous avons vu qu'elle permettait de combler certains manques de celle-ci. Les fonctionnalités transversales, et la duplication de code qui les accompagnent sont généralement bien gérées par la POA. Nous avons également donné, dans ce chapitre, un aperçu sur l'un des langages supportant la POA, à savoir, AspectJ. Ce langage est l'outil le plus utilisé actuellement dans le développement en POA.

Chapitre III

Analyse dynamique de programmes et profilage

1. Introduction

L'analyse dynamique de programmes est une activité très importante dans le domaine de l'ingénierie de logiciel qui sert à analyser le programme en exécution et d'observer son comportement.

L'analyse dynamique étudie une exécution concrète d'un programme, au contraire de l'analyse statique, par conséquent l'analyse dynamique est précise parce qu'aucune approximation ou abstraction n'est faite. Les inconvénients de l'analyse dynamique sont que ses résultats peuvent ne pas être valides pour les exécutions futures, qu'il n'y a pas d'abstractions et que la taille des données générées devient très difficile à gérer [Rac05].

2. Analyse statique

L'analyse statique de programmes est un champ de recherche dont le but est, selon Nielson [Nie04], d'offrir des techniques pour *prédire* des approximations sûres et calculables à l'ensemble des valeurs ou des comportements découlant dynamiquement lors de l'exécution d'un programme. Autrement dit, il s'agit de raisonner sur un programme sans l'exécuter.

L'analyse statique est habituellement construite de façon sûre, c'est-à-dire que ses résultats s'étendent à toutes les exécutions possibles du programme (pas de faux négatifs), ce qui se fait souvent au détriment de la précision, l'absence de faux positifs. Pour ce faire, elle fait preuve de conservatisme, c'est-à-dire qu'elle démontre des propriétés faibles, mais qui sont effectivement toujours vraies [Ern03].

3. Le profilage

3.1. Définition et généralités

Le profilage [Raj02, Bal99] des programmes est une technique d'analyse permettant de déterminer, à l'exécution (analyse dynamique), les causes de mauvaises performances de programmes et donc les zones du code où l'optimisation serait le plus rentable.

L'analyse dynamique permet [Duf04] :

- D'obtenir des résultats plus précis que par une analyse statiques pour des exécutions concrètes ;
- D'obtenir de l'information de nature temporelle à propos de l'exécution : Compréhension de programmes, détection de phases, etc.
- D'obtenir de l'information sur la fréquence de certains évènements : Optimisation JIT (Just In Time), débogage de performances, etc.
- Analyse de l'utilisation des ressources (utilisation de mémoire, CPU, etc)

De nombreuses caractéristiques peuvent faire l'objet d'une analyse dynamique, parmi lesquelles :

- le temps CPU,
- les accès mémoire,
- le graphe des appels,
- les blocs de base,
- les entrées/sorties,
- et le nombre de cycles CPU pour chaque ligne d'instructions dans un ou plusieurs sous programmes.

4. Types d'analyse dynamique

Selon Dufour Bruno [Duf04], on distingue deux types d'analyse dynamique :

1. L'analyse dynamique en ligne:
   - Le programme est évalué au cours de son exécution ;
   - Des calculs complexes peuvent perturber l'exécution ;
   - Seulement l'information pertinente est enregistrée.
2. L'analyse dynamique hors-ligne:
   - Le programme est évalué après l'exécution à l'aide des traces d'exécution ;
   - L'impact sur l'exécution est diminué ;
   - La quantité d'information enregistrée est énorme et proportionnelle au temps d'exécution.

Le profilage indique aux programmeurs et aux compilateurs les lieux les plus propices à des optimisations. Il s'agit donc par cette analyse de générer une vue globale montrant les relations entre les blocs de base des programmes, puis plus tard, lors d'un autre profilage ou d'une compilation, de se concentrer sur les parties nécessitant un travail supplémentaire, dans une perspective optimisante [ken02].

L'analyse dynamique présente un processus itératif, qui comporte plusieurs phases généralement regroupées en trois : l'instrumentation du code, la compilation et l'exécution du code instrumenté.

5. Phases du profilage

5.1. L'instrumentation du code

L'instrumentation consiste à ajouter des fragments de code (e.g, sondes) à un programme de façon à ***ajouter*** la génération des résultats d'analyse à son comportement initial, au cours de l'exécution [Bin07, Duf04].

Il est question ici *d'équiper* les codes sources pour le profilage, i.e. y ajouter des informations particulières devant guider le profilage en lui-même.

Une application instrumentée paraît inchangée pour l'utilisateur, et après son exécution, des données sont recueillies. La version instrumentée du programme doit donc être exécutée sur une ou plusieurs entrées "significatives" du programme, afin de collecter en sortie des informations de profilage.

L'instrumentation de programmes peut se faire à trois niveaux :

- Le code source,
- Le bytecode et
- la machine virtuelle.

L'instrumentation du code source relève bien sûr de l'approche statique, et l'instrumentation de la machine virtuelle de l'approche dynamique. Mais l'instrumentation du bytecode peut être réalisée avant l'exécution, pendant le chargement des classes ou bien même à l'exécution.

5.1.1. Instrumentation statique :

L'instrumentation statique [Del07] peut avoir lieu au niveau du code source. La principale contrainte de l'instrumentation statique est de disposer de tout le code à instrumenter avant son exécution.

L'instrumentation du code source commence par une analyse du code source, à l'aide d'analyseur lexical et syntaxique, puis place des sondes dans le code source. Ainsi,

Cenqua Clover Code Coverage for Java [CENQUA], par exemple, analyse le code source Java, à l'aide d'analyseurs générés par ANTLR (ANother Tool for Language Recognition). Cette technique lui permet de découvrir naturellement les structures de contrôles de flot pour placer les sondes permettant de découvrir les branches mortes.

5.1.2. Instrumentation dynamique

L'instrumentation dynamique [Del07] d'un programme repose principalement sur deux techniques.

- La première consiste à instrumenter le *bytecode* des classes Java au cours de leur chargement.
- La seconde, en revanche, utilise la machine virtuelle pour intercepter des événements qui nous intéressent, grâce à des *hooks* [3].

Mais il existe une troisième voie d'instrumentation utilisant à la fois l'instrumentation du bytecode et de la machine virtuelle.

5.1.2.1. Instrumentation du *bytecode* au chargement

Pour instrumenter le bytecode au chargement d'une classe, il est assez évident pour un développeur Java de surcharger la classe *ClassLoader* [Del07]. Un agent Java est déployé via fichier JAR. C'est un service qui est disponible depuis la version 5 de Java et qui permet d'exécuter un deuxième programme en même temps que le programme principal grâce à l'utilisation de l'option ***-javaagent*** de la commande Java. Cette commande fonctionne sur toutes les JVMs qui implémentent le service de l'agent Java.

Le framework **ASM** [Bru02, Bru07, Kul07] qui fournit une bibliothèque pour parcourir les classes Java, ajouter des bytecodes aux classes et réaliser des transformations complexes du bytecode. En effet, ASM peut entièrement modifier le bytecode d'une classe existante ou peut créer dynamiquement de nouvelles classes [Cas12]. Selon l'étude [Kul07], ASM est beaucoup plus rapide que les autres bibliothèques d'instrumentation telles que BCEL [Dah02], SERP [Whi02] et Javassist [Chi04] et le coût mémoire d'ASM est très faible.

5.1.2.2. Gestion d'événements

La gestion d'événements [Del07] requiert que la machine virtuelle dispose d'une interface pour intercepter des événements comme le chargement d'une classe ou l'exécution d'une fonction. Cette technique est limitée par le nombre d'événements observés, par leurs précisions.

[3] *Un hook (littéralement « crochet » ou « hameçon ») permet à l'utilisateur d'un logiciel de personnaliser le fonctionnement de ce dernier, en lui faisant réaliser des actions supplémentaires à des moments déterminé. (http://fr.wikipedia.org/wiki/Hook_(informatique)).*

Il existe des interfaces natives (accessibles en C/C++) pour surveiller l'exécution de programmes Java. Avant Java 5, le standard (supporté à la fois par les machines virtuelles de Sun Microsystems, BEA Systems et IBM) s'appelait JVMPI (Java Virtual Machine Profiler Interface). Il permet de placer des hooks notamment sur les événements suivants :

- le chargement de classe ;
- le démarrage et fermeture de thread ;
- l'entrée et la sortie de méthode ;
- le chargement et le déchargement de méthode compilée (JIT).

Selon [Del07], la limitation de JVMPI réside donc dans le nombre limité d'événements observés. Mais, cette interface permet de consulter directement un certain nombre d'informations sur la machine virtuelle (trace d'appels notamment), et permet d'interagir avec l'objet source de l'événement grâce à l'interface native Java (JNI).

La surveillance de la machine virtuelle paraît une bonne solution pour réaliser des mesures de performances. Malheureusement, comme indiqué dans les articles [Har02, Dim04], lorsque JVMPI ou un de ses équivalents place un hook sur l'entrée et la sortie de méthodes, ce n'est pas pour une méthode en particulier mais pour toutes les méthodes, ce qui oblige à maintenir des tables de hachage, entrainant un surcoût à l'exécution.

Avec l'arrivé de Java 5, JVM TI (JVM Tool Interface) prend le relais en tenant compte de ce problème. Comme indiqué dans [O'ha04], il offre moins d'événements. Cette interface apporte d'avantages sur les solutions hybrides à base d'injection de bytecode.

5.2. La compilation et l'exécution du code instrumenté

L'exécution s'effectue sur une ou plusieurs données d'entrées correctement choisies, et il en résulte la génération de plusieurs fichiers de statistiques [ken 02].

5.3. La visualisation et l'analyse des statistiques

L'arbre des appels des fonctions est analysé, le pourcentage de temps CPU utilisé est calculé, le nombre d'appels à chaque fonction est quantifié, également, etc [ken02]. C'est ainsi qu'on obtient les données de profilage.

5.4. Recompilation du programme

Grâce aux données de profilage précédemment collectées, on procède à une nouvelle compilation du programme pour produire du code optimisé.

Il est à noter que l'on peut être amené à reprendre toutes les étapes ci-dessus avant de pouvoir obtenir un code optimisé satisfaisant.

La figure ci-après résume d'une manière graphique les différentes étapes de profilage :

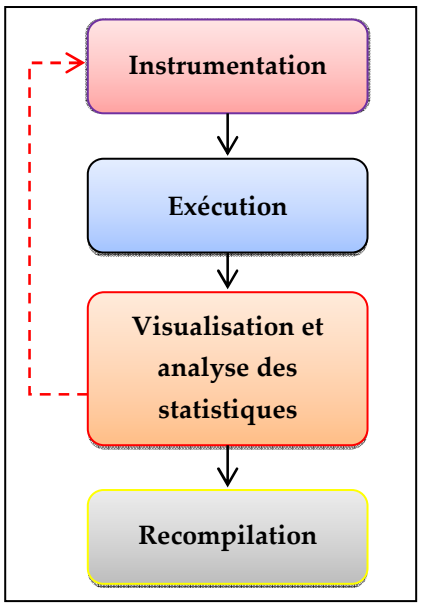


Figure 3.1 : **Etapes de profilage**

6. Techniques d'instrumentation et données de profilage

Il existe plusieurs types de données de profilage, à savoir les données issues du profilage du graphe de contrôle de flots, les données issues du profilage des valeurs et les données issues du profilage des adresses mémoires [Ken02]. Chacune de ces données peut être recueillie par l'une des deux techniques que sont l'échantillonnage et le traçage.

6.1. L'échantillonnage

L'analyse par échantillonnage consiste à enregistrer un échantillon de l'exécution périodiquement, autrement dit, le programme (non modifié) est interrompu fréquemment, et le compteur de programme (PC) ou la pile d'appels sont enregistrés lors de ces interruptions, avec pour but de déterminer dans quelles procédures, dans quelles lignes de code le programme passe le plus de temps, et les régions du code qui sont exécutées le plus fréquemment.

On distingue donc deux stratégies de base [Duf04] :

- Par interruption / minuteur : un échantillon pour chaque période (ex: 100ms) ;
- Par compteurs : un échantillon à chaque n occurrences (ex: 1 échantillon à chaque 10 appels)

6.2. Le traçage

Tracer un système et exploiter ses traces d'exécution est un moyen classique de mise au point [Tou11]. Tracer un système consiste à enregistrer des historiques d'exécution reflétant les événements qui se sont produits dans le système pendant son exécution. Les traces capturées (acquises) varient selon le type des événements considérés (logiciels,

matériels), le niveau de détail de traçage, l'organisation des informations (structures et formats) et leur stockage.

Exploiter les traces d'exécution consiste à analyser les informations enregistrées afin de répondre à des questions sur le fonctionnement du système (Comment marche-t-il ?), l'évaluation des performances (Le système est-il performant ?), la validation (Respecte-t-il les spécifications ?), le débogage (Où est l'erreur ?) et l'optimisation (Comment pourrait-il marcher mieux ?).

Les évènements de la trace contiennent les informations suivantes [Rac05] :

Identificateur : identificateur d'évènements;
Type : type d'évènement :

- *mEntry* : l'entrée de la méthode;
- *mExit* : la sortie de la méthode;
- cEntry : le début de l'instanciation;
- *cExit* : la fin de l'instanciation.

Origine : classe qui a fait appel aux constructeurs ou aux méthodes d'une autre classe;
Commentaire : classe dont une instance reçoit l'appel de méthode;
Nom : nom de l'évènement (si le Type est une méthode alors *Nom* sera le nom de la méthode et si c'est un constructeur se sera le nom de la classe dont une instance est construite);
Valeur retournée : la valeur retournée par la méthode appelée s'il y a lieu.

6.2.1. Exemple :

Prenons un programme (figure 3.2) simple de 6 classes A, B, C, D, E, F où :

```
package MagisterInformatique_OEB2014;
public class TestMain {
    public static void main(final String[] args) {
        A a = new A();
        D d = new D();
        C c = new C(d);
        E e = new E();
    } }
class A {
    B b = new B();
    public A(){
        operation1(b);
    }
    public void operation1(B b){
        b.operation2();
    } }
```

```
class B {
    public void operation2(){
    } }
class C {
 public D d;
 public C(D d){
this.d = d;
    operation3();
 }
 public void operation3(){
    this.d.operation4();
 } }
```

```
class D {
    public void operation4(){
    }}
class E {
    public F f = new F();
    public E() {
        operation5();
    }
    public void operation5(){
        this.f.operation6();
    } }
class F {
public voidoperation6(){
    }
}
```

Figure 3.2 **Exemple d'un programme implémentant 6 classes**

La trace dynamique générée par une exécution de la méthode *main()* de la classe *Testmain* qui n'inclut que les appels des méthodes et des constructeurs est donnée par la figure 3.3 :

mEntry(JVM,TestMain,main) cEntry(TestMain,A) cEntry(A,B) cExit(A,B) mEntry(A,A,operation1) mEntry(A,B,operation2) mExit(A,B,operation2) mExit(A,A,operation1) cExit(TestMain,A) cEntry(TestMain,D) cExit(TestMain,D) cEntry(TestMain,C) mEntry(C,C,operation3) mEntry(C,D,operation4) mExit(C,D,operation4) mExit(C,C,operation3) cExit(TestMain,C) cEntry(TestMain,E)	cEntry(E,F) cExit(E,F) mEntry(E,E,operation5) mEntry(E,F,operation6) mExit(E,F,operation6) mExit(E,E,operation5) cExit(TestMain,E) mExit(JVM,TestMain,main)

Figure 3.3 **Traces d'exécution du programme de la figure précédente**

6.3. Données issues du profilage

Grâce à l'échantillonnage on peut profiler des valeurs et des adresses mémoires **[Ken02]** :

- ***Données issues du profilage des valeurs :*** Il est question ici d'identifier les valeurs spécifiques rencontrées comme opérandes d'instructions, ainsi que les fréquences auxquelles ces valeurs sont rencontrées. Grâce à ces informations, on peut :
  - Déterminer les opérandes dont la valeur est toujours constante,
  - Identifier les codes invariants dans les boucles, ce qui permet des optimisations telles que la propagation de constantes, la réduction forte, etc.
- ***Données issues du profilage des adresses mémoires :*** Ces données sont sous forme d'ensembles d'adresses mémoires référencées par un programme. Elles sont utiles pour effectuer la disposition des données en mémoire et les transformations par placement de code, toutes choses qui améliorent de manière sensible l'exploitation de la hiérarchie des mémoires.

7. Optimisations déduites du profilage

7.1. Identification des parties coûteuses d'un programme

Pour améliorer les performances d'un programme, il faut en déterminer la consommation en ressources machine: la vitesse du processeur et sa disponibilité, la taille mémoire et sa capacité, etc [ken02] :

- Une exécution limitée par le CPU passe le plus clair de son temps dans le CPU et en est limitée par la vitesse ou la capacité; les améliorations possibles consisteront alors à modifier l'algorithme, réordonner les codes et éviter les blocages, supprimer les boucles superflues, appliquer le blocking pour conserver les données en cache ou dans les registres ou même changer d'algorithme.
- Une exécution limitée par les entrées/sorties est telle que le programme attend des entrées/sorties pour se terminer et peut être limitée par la vitesse des accès disques ou les mémoires cache; les améliorations possibles sont : optimisation de l'usage des données pour minimiser les accès disques, compression de données, etc.
- Une exécution limitée par la mémoire est telle que le programme fait de fréquents swap out des pages mémoires; on peut améliorer les accès mémoire en améliorant la localité des références. On peut aussi diminuer, le cas échéant, la taille mémoire utilisée par le programme.
- Une exécution peut être limitée par des bugs : le programme lit incessamment la même valeur dans le même fichier ou alors il y a un "floating point exception" qui ralentit le programme. Il peut également subsister un ancien code non enlevé complètement (mise à jour inachevée) ou encore un manque de mémoire pour le programme causé par des malloc() non suivis de free().

8. Quelques outils de profilage

8.1 JRat (**J**ava **R**untime **A**nalysis **T**oolkit**)**

JRat [jrat] rentre dans la catégorie des applications tierces qui instrumentent le bytecode Java. Ce profiler, développé depuis 2001, a connu plusieurs évolutions majeures au sein de la version **1-alpha2** dont le passage de BCEL [BCEL] à ASM [Bru02, Bru07, Kul07] pour ce qui est de la technologie de manipulation/instrumentation du bytecode. Le temps d'injection s'en est trouvé sensiblement amélioré.

Il existe deux méthodes pour utiliser JRat :

1. **L'ancienne** : depuis le **JRat Desktop**, choisir le .class, le package ou bien le JAR, WAR ou EAR à instrumenter (ce qui permet de gérer la portée du profilage) puis lancer l'application sans oublier de placer le JAR de JRat dans le classpath.

2. **La nouvelle** apparue depuis la version **1-alpha2** : instrumenter le code à la volée grâce à l'option -javaagent apparue dans le JDK 1.5.

Ainsi instrumentée, chaque méthode consigne dans un fichier le delta de temps entre l'entrée et la sortie. A l'issue de l'exécution, le fichier peut être ouvert dans le JRat Desktop, il présente les informations suivantes :

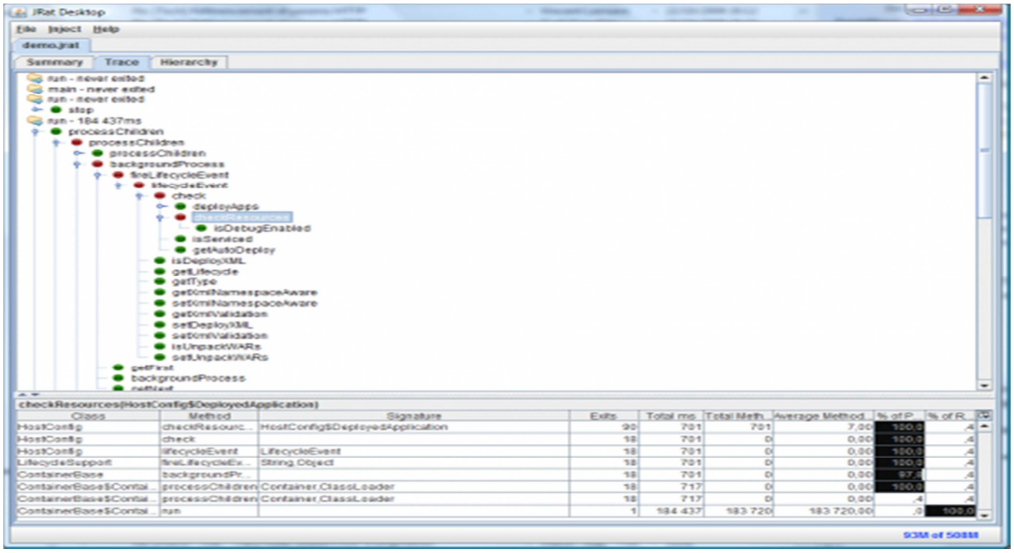


Figure 3.4 : **JRat Desktop**

Les points forts de JRat sont clairement :

- Sa facilité de mise en œuvre qui le rend parfait pour les utilisations ponctuelles.
- Sa gratuité.
- Une journalisation des performances si le code reste instrumentée.

8.2 VisualVM [VisualVM]

Un autre outil intégré à la JVM est **VisualVM**, que ses créateurs décrivent comme "*un outil visuel intégrant diverses lignes de commande du JDK et offrant des capacités de profilage légères*", et permet d'analyser graphiquement la mémoire, VisualVM fournit également un *sampler* et un *profiler* léger.

Le sampler de VisualVM permet de mesurer périodiquement l'usage du CPU et de la mémoire, avec en plus une mesure de l'usage du CPU par méthode. Ceci permet d'obtenir un aperçu rapide des temps d'exécution des méthodes, échantillonné à intervalles réguliers:

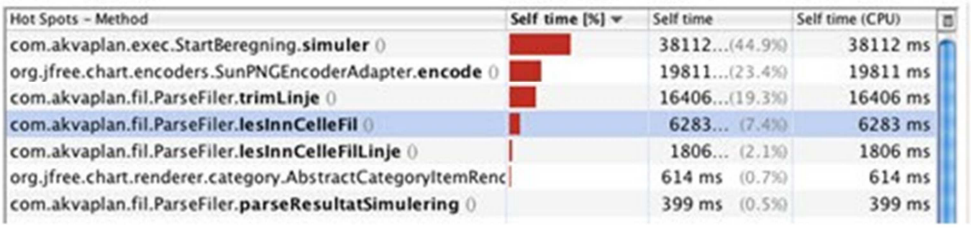


Figure 3.5 **: Tableau temporel de VisualVM pour l'exécution des méthodes**

Le profiler de VisualVM nous apporte la même information que le Sampler, mais plutôt qu'une mesure périodique, il permet de collecter des statistiques dans le cadre normal de l'exécution de l'application (grâce à l'instrumentation du bytecode). Les statistiques obtenues sont plus précises, et mises à jour plus fréquemment, que les données collectées par le Sampler.

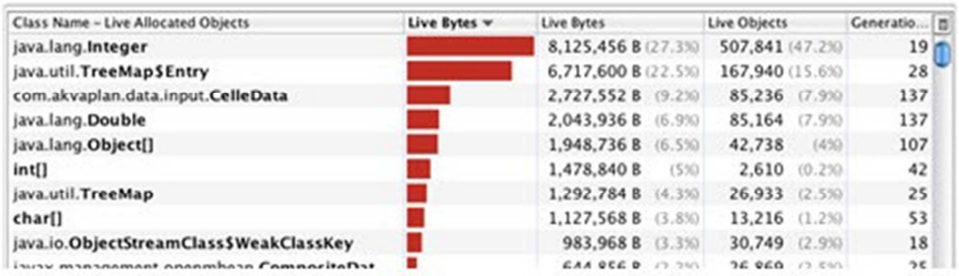


Figure 3.6 : **sortie écran du Profiler VisualVM**

L'utilisation du profiler a quand même un inconvénient: L'instrumentation mise en œuvre va essentiellement redéfinir la plupart des classes et méthodes de l'application, ce qui ralentira considérablement l'exécution de l'application.

Il est important de noter que VisualVM n'est pas un Profiler complet, car il n'est pas capable de tourner en permanence sur une JVM en production. Ses données ne sont pas sauvegardées, et aucun système de définition/notification d'alertes n'est disponible.

9. Travaux Similaires :

Dans l'inexistence totale des travaux portant sur l'évaluation de l'impact de séparation des préoccupations transversales sur les applications orientées agent, nous préférons, au moins, dresser un bref aperçu sur certains travaux sur l'impact (analyse statique et dynamique) de la séparation des préoccupations transversales sur les applications orientées objets existantes :

Hannemann et Kiczales [Han02] ont entrepris une étude dans laquelle ils ont développé et ont comparé des implémentations Java et AspectJ des 23 patrons de conception GoF [Bau97]. Ils revendiquent que le langage de programmation utilisé affecte la mise en œuvre des patrons de conception. De là, il est naturel d'explorer l'effet des

techniques de la programmation orientée aspect sur l'implémentation des modèles GoF. Pour chacun des 23 patrons GoF, ils ont développé un exemple représentatif qui se sert du patron et qui implémente un exemple Java et un autre AspectJ. Le but de cette étude étant de modulariser les rôles des patrons. Les auteurs ont conclu que la modularité est améliorée dans 17 des 23 cas et 12 implémentations de patron orientées aspect donnent une meilleure réutilisation. Le degré d'amélioration avec la POA varie selon chaque implémentation de patron.

Alessandro Garcia et al. [Gar03] ont repris le travail de Hannemann et Kiczales [Han02] et ont présenté une étude quantitative qui compare des solutions Java et leurs équivalents AspectJ pour les 23 patrons GoF. Ils ont utilisé des attributs logiciels comme critères d'évaluation, entre autres, la séparation de préoccupations, le couplage, la cohésion et la taille. Alessandro Garcia et al. [Gar06] ont tiré quelques conclusions de leur travail. Ils ont constaté que :

- L'utilisation des aspects améliore le couplage et la cohésion de quelques implémentations de patrons de conception.
- La plupart des solutions orientées aspect améliorent la séparation des préoccupations transverses reliées aux patrons de conception, même si seulement quatre implémentations orientées aspect ont montré une amélioration significative au niveau de la réutilisation des classes.
- L'aspectualisation des patrons de conception réduit le nombre d'attributs de 10 patrons, et diminue le nombre d'opérations et des paramètres respectifs de 12 patrons de conception.
- La relation entre les rôles des patrons et l'application spécifique des préoccupations est parfois si intense qu'il ne semble pas trivial de séparer ces rôles dans des aspects.
- L'utilisation des mesures de couplage, de cohésion et de la taille a été utile pour aider à la détection d'opportunités pour le refactoring orientée aspect des patrons de conception.

En plus, ils ont discuté l'adaptabilité des solutions analysées en ce qui concerne la séparation des préoccupations et la détermination d'un modèle qui peut, soi-disant, prédire la modularisation des patrons de conception avec les aspects.

Ceccato et al. [Cec04] ont mené une étude quantitative pour mesurer l'impact de l'aspectualisation d'un programme. Les auteurs se sont intéressés surtout à ce qu'ils s'appellent couplage implicite introduit par les aspects pour essayer de déterminer les avantages et désavantages de la POA. Ils ont proposé une approche basée sur quelques métriques (taille, complexité, héritage, cohésion et couplage) qu'ils ont calculées sur de petits programmes OO et leurs équivalents AspectJ [AspectJ]. Les auteurs ont conclu qu'il y a eu une amélioration générale des valeurs de certaines métriques (donc des attributs relatifs à ces métriques). Le coût à payer de ces améliorations est, cependant, une

augmentation des valeurs de la métrique CIM (Coupling on Intercepted Modules) et de la métrique CDA (Crosscutting Degree of an Aspect) en raison de l'interception de l'aspect à l'exécution des méthodes (couplage POA). Les résultats indiquent que les propriétés telles que la proportion du système touchée par un aspect et le degré de connaissance d'un aspect des modules dans lesquels il s'injecte sont capturées par les métriques de couplages (CDA et CIM). Les auteurs envisageaient, alors, la définition d'un ensemble d'indicateurs POA communs, pour qu'il soit adopté par la communauté POA, afin de simplifier la comparaison des résultats obtenus par différentes équipes de recherche et d'avoir une méthode d'évaluation standard.

Diaz et Compo [Dia01] ont implémenté un système de contrôle de température (TCS, temperature control system), de quatre différentes manières : a) version orientée objet , b) version aspectJ, c) Système de POA basé sur la réflexion, nommé : TaxonomyAOP et d) Système basé sur les évènements, nommé Bubble. Pour l'évaluation, des métriques ont été appliquées sur le code source résultant, et le temps de réponse des différentes mises en oeuvre ont été mesurées. Ce travail contient en fait une étude comparative des performances de AspectJ pour les versions : taxonomie AOP et Bulle, mais pour les deux derniers aucune autre information n'est disponible.

Zhang et Jacobsen [Zha03] ont réfactorisé une entière implémentation d'ORB (Object Request Broker), réalisant la production de plusieurs aspects dans aspectJ qui ont été identifiée en utilisant les techniques d'aspect Mining. La performance est évaluée par la collecte des mesures statiques : mesurer le temps de réponse pour des appels à distance, et la collecte de données de performance en insérant des points de mesure dans la pile d'exécution de l'ORB. La performance de l'ORB initiale a été comparée à celle de la version AspectJ.

10. Conclusion

Nous avons présenté, dans ce chapitre, l'analyse dynamique de programmes et les concepts relatifs à la notion de profilage. Nous avons également introduit les différentes techniques d'instrumentation de code avec quelques exemples d'outils de profilage. Ces deux notions importantes représentent les fondements de bases pour notre contribution qui va être présentée dans le chapitre suivant. Elles sont utilisées pour évaluer dynamiquement l'impact des constructeurs apportés par la programmation orienté aspect sur les performances des SMAs en termes de communication entre agents.

Chapitre IV

Approche proposée

1. Introduction

Les profileurs emploient une grande variété de techniques pour rassembler des données, y compris des *interruptions*, *l'instrumentation* de code, des *hooks* de système d'exploitation, et des *compteurs* d'exécution.

Actuellement, il existe un bon nombre de logiciels qui permettent de faire l'analyse dynamique de programmes. Le facteur commun entre ces outils c'est le profilage des programmes JAVA sans prendre en considération le type de l'application en cours d'exécution, par contre, il y a un nombre très restreint d'outils qui font l'analyse dynamique orientée-agent, nous citons : COUGAAR [Aar03] et AgentSpotter [Din08a, Din08b]. Dans le cadre de notre approche, nous optons pour l'outil AgentSpotter.

Le point fort de agentSpotter c'est son aspect visuel qui permet de visualiser le comportement des agents en interaction après la session de profilage, cette visualisation est illustrer par le graph d'appel orienté agent [Din08a], et ainsi le diagramme espace-temps [Din08b] qui nous permet de contrôler où les agents passent la plupart de leurs temps de traitement des différentes tâches. Un autre point fort de AgentSpotter par rapport à COUGAAR [Aar03], c'est qu'il supporte l'analyse dynamique hors-ligne et par conséquent, l'impact sur l'exécution est diminué et ainsi la quantité d'information enregistrée est énorme et proportionnelle au temps d'exécution.

Nous proposons dans ce chapitre une nouvelle approche qui permet d'étudier l'intérêt de la séparation des préoccupations transversales et son impact sur les applications orientées agent. Plus précisément, notre approche a pour but de savoir si le refactoring orienté aspect nous permet d'apporter une amélioration sur le comportement des agents en terme de communication et échange de messages pour leur permettre d'atteindre et de compléter leurs tâches. Par conséquent, on peut décider que le refactoring OA a un impact positif sur les applications orientées-agent réfactorisées, ou bien d'apporter une surcharge de communication et dans ce cas la séparation des préoccupations transversales ou bien les mécanismes de la programmation orientée aspect influe négativement sur la communication entre les agents.

Pour valider notre approche d'évaluation nous avons procédés dans la deuxième partie de ce chapitre à la réfactorisation d'une application développée précisément pour démontrer l'efficacité du graph d'appel orienté agent comme étant un outil de profilage.

2. Préoccupations transversales orientées agent :

Le comportement de l'agent est composé de l'ensemble de ses propriétés. La Table 4.1 résume les définitions des principales propriétés de l'agent. Plusieurs chercheurs [Gar02; Pac03; Gar03] considèrent la mobilité [Uba01], l'interaction [Gar04; Gar03], l'apprentissage [Gar04], l'autonomie [Gue99; Ama98], et la collaboration [Ken99; Uba01] comme étant des propriétés transverses.

Préoccupation de l'agent	Définition
Interaction	L'agent communique avec son environnement et les autres agents à travers ses capteurs et ses effecteurs
Adaptation	Un agent adapte son état selon les messages reçus à partir de l'environnement
Autonomie	Un agent est capable d'agir sans intervention extérieure, il a son propre contrôle et peut accepter ou refuser un ordre.
Apprentissage	Un agent peut apprendre selon son expérience précédente en régissant et en interagissant avec son environnement
Mobilité	Un agent est capable de se déplacer d'un environnement dans un réseau à un autre
Collaboration	Un agent peut coopérer avec d'autres agents pour atteindre ses buts et réaliser les objectifs du système

Table 4.1 **: Propriétés transversales de l'agent [Gar02]**

On va présenter dans cette section quelques préoccupations transversales dans le développement des SMAs. Une préoccupation est une partie du système qui peut être traitée comme une unité conceptuelle séparée.

On peut noter que la préoccupation **interaction** transverse les classes qui implémentent d'autres préoccupations de l'agent, on remarque qu'il a un impact énorme sur la structure de base de l'agent. Le code spécifique à la préoccupation *interaction* est dupliqué et dispersé à travers plusieurs modules dans la hiérarchie du système.

Noter que même si nous essayons de proposer une solution orientée objet de la situation présentée dans la figure 4.1, nous ne pouvons pas trouver une solution plus modulaire. Ce problème se produit parce que l'interaction est une préoccupation transversale indépendante de la décomposition orientée objet utilisée, cette situation est similaire pour les autres préoccupations orientées agent telles que l'apprentissage qui est, généralement, transversale.

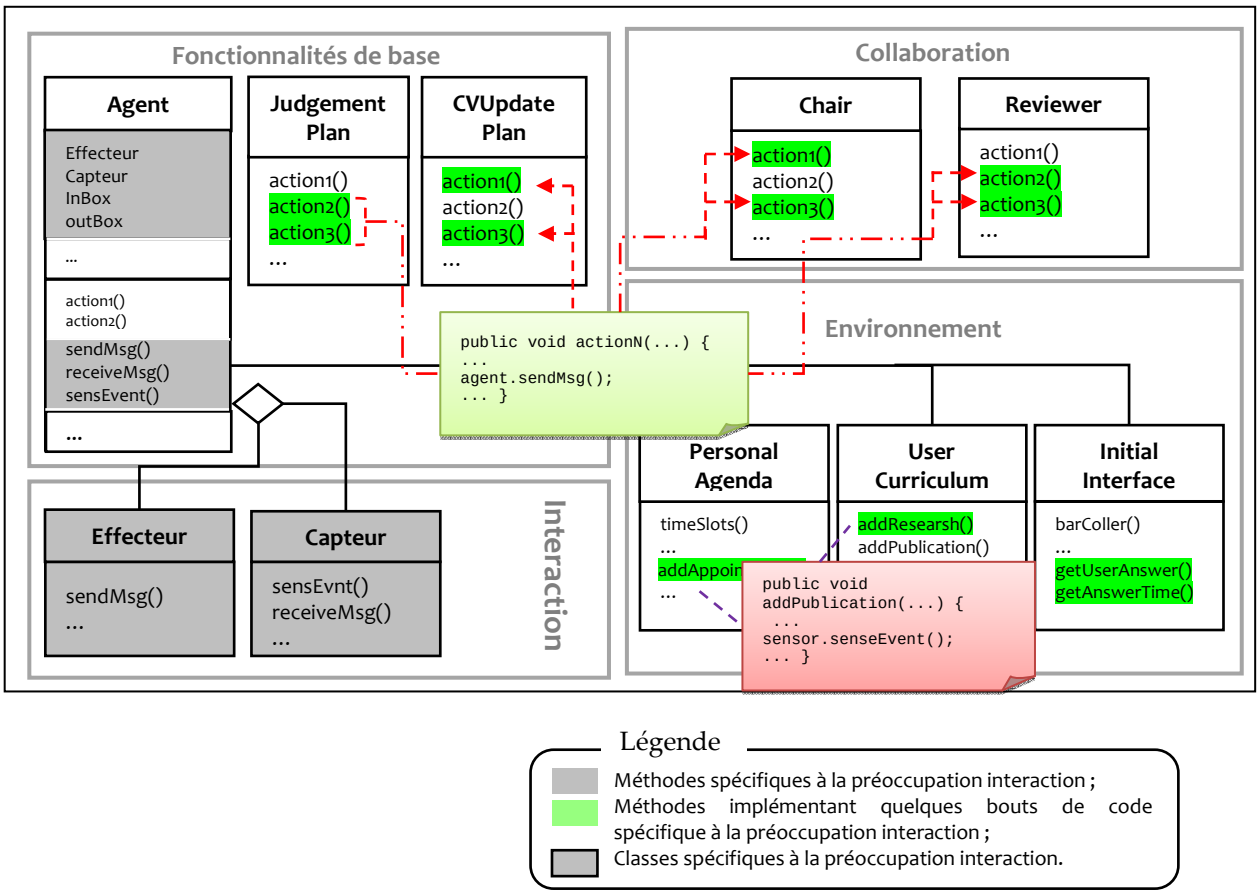


Figure 4.1 : **Préoccupations transversales de l'agent [Gar04]**

Une interface transversale [Gar02], spécifie où et quand un aspect affecte les autres modules, par exemple, la figure 4.2 montre l'interface *Iinformation Gathering* dans le composants (aspect) *learning* qui transverse généralement les autres préoccupations. Un aspect comporte un ensemble d'interfaces transversales.

Une architecture d'agent orientée aspect fournit des composants pour modéliser les aspects des préoccupations transversales. Les préoccupations de l'agent sont modularisées [Gar02] dans des aspects individuels, une architecture orientée aspect est composée de deux types de modules : les composants noyaux qui modélisent les préoccupations de base de l'agent, et les aspects qui modélisent les préoccupations transversales de l'agent dans des modules séparés des autres composants noyaux. La Figure 4.2 montre une représentation partielle d'une architecture orientée aspect de l'agent qui comporte un module noyau et deux aspects, avec des interfaces simples et transversales.

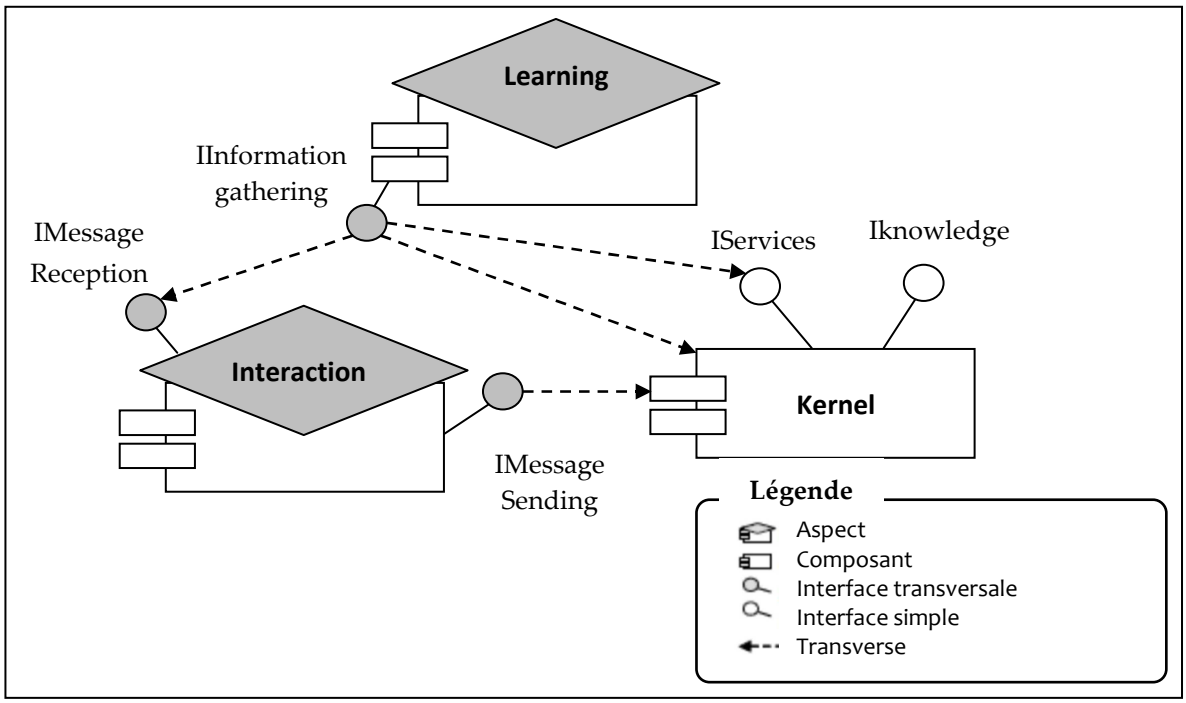


Figure 4.2 : **Architecture orienté aspect [Gar02]**

3. Fondements de base de notre approche :

Nous allons détailler dans cette partie les différents outils et plateformes, que nous avons utilisés dans le cadre de notre approche d'évaluation orientée agent, en se basant sur les techniques de l'analyse dynamique.

3.1. Profileur spécifique pour les SMAs

3.1.1 AgentSpotter :

AgentSpotter (Figure 4.3) est un outil de profilage conçu spécifiquement pour la collecte et la représentation des informations de profilage sur les SMAs [Din08a] et [Din08b]. Conçu pour supporter le maximum de compatibilité avec n'importe quel type de plate-forme agent.

Le Service AgentSpotter (agentSpotter service) fonctionne dans l'environnement d'exécution de la plate-forme agent, il permet la collecte de données sur les agents eux-mêmes (les actions réalisées, les messages échangés, etc), ainsi que les données du système, telles que le temps processeur et l'utilisation de la mémoire, c'est la seule portion du système qui doit être portée de façon à permettre AgentSpotter à être intégrer sur différentes plates-formes agent.

AgentSpotter Station est une application indépendante attachée au profileur AgentSpotter. Elle fournit un certain nombre de visualisation sur divers aspects liés à la

performance du système en question afin de permettre aux programmeurs d'identifier les endroits dans le code qui utilisent beaucoup plus de temps de traitement que prévu.

Une description plus détaillée de cette mise en œuvre de : AgentSpotter station et les données recueillies par AgentSpotter peuvent être trouvés dans [Din08a].

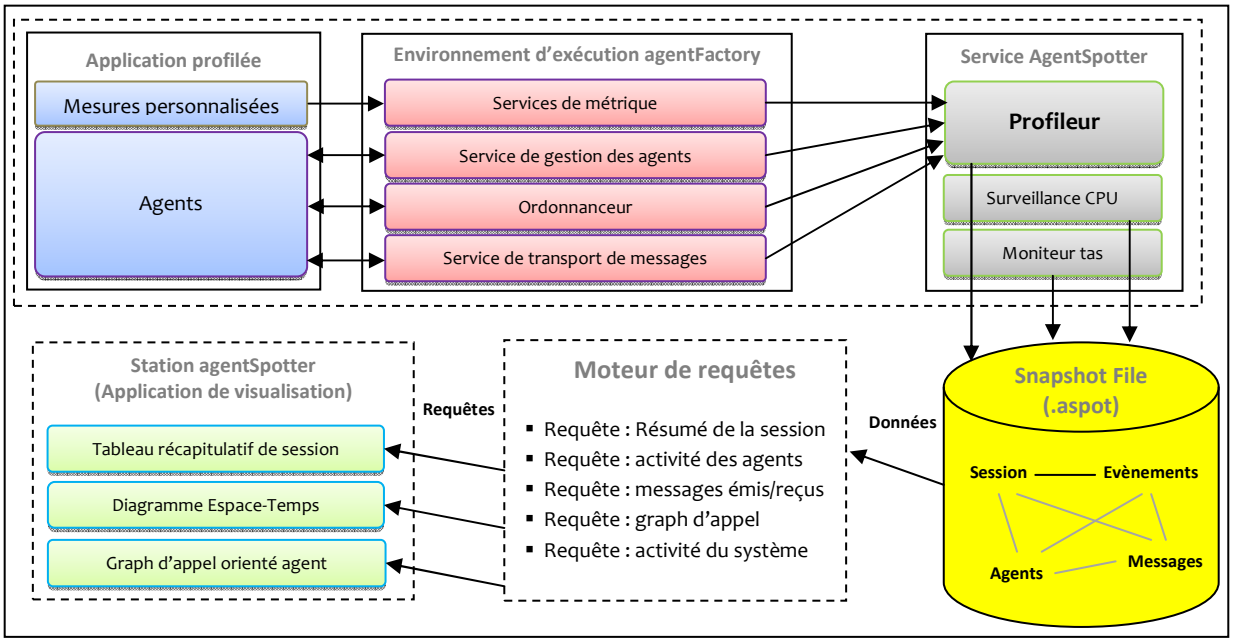


Figure 4.3 **Architecture de AgentSpotter** [Din08a]

En outre, l'application AgentSpotter Station fournit un ensemble des statistiques concernant la session d'exécution qui comporte les éléments suivants :

- **Durée totale :** temps exécution de la session.
- **L'activité totale** : quantité de temps de traitement enregistré au cours de la session.
- **Nombre total de message**: Nombre de messages envoyés ou reçus dans la plateforme agent, les chiffres pourraient éventuellement ne pas correspondre en cas de communication inter-plate-formes.
- **Nombre moyen d'agents actifs par seconde** : ce nombre donne une idée du niveau de concurrence de l'application.

3.2. Graph d'appel orienté agent

Le modèle conceptuel présenté dans la détermination des attributs dynamiques est lié à trois niveaux : le niveau de la session, le niveau de l'agent émetteur et le niveau de l'agent récepteur. Une transformation graphique du modèle décrit dans la figure 4.4 devrait être une arborescence représentant les niveaux que nous avons déjà énumérés, plus un niveau supplémentaire pour le contenu du message FIPA ACL [Din08a].

Le contenu du message est définit comme suit : type plus contenue de message par exemple : « request : doSomeThing(123)»

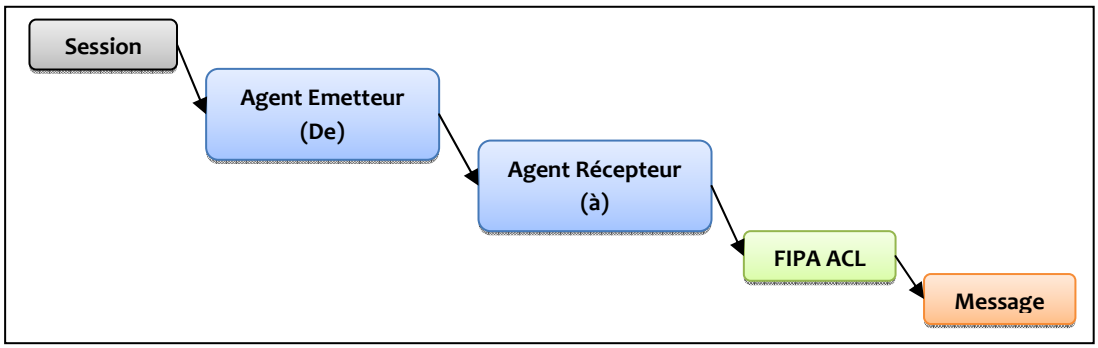


Figure 4.4 **Les niveaux de l'arborescence dans le graph d'appel orienté-agent [Din08a]**

La session à la racine de l'arbre doit être égale à 100% de l'impact de tous les agents émetteurs tel que défini par l'équation (5). Ensuite, à chaque niveau, chaque nœud doit totalisera récursivement l'impact de ses nœuds enfants jusqu'aux feuilles qui présentent le contenu des messages. Ces feuilles rapportent simplement leur impact tel que défini par l'équation (5). Plus précisément, à chaque niveau de l'arborescence, les valeurs suivantes doivent être affichées :

- **Etiquette** : texte informatif associé au nœud. La structure de l'étiquette dépend du niveau de la façon suivante:
  - **Session :** capture la date, l'heure, et la durée de la session
  - **Agent Emetteur :** Identificateur de l'agent (AgentID) emetteur,
  - **Agent Récepteur :** Identificateur de l'agent (AgentID) récepteur,
  - **FIPA ACL :** Type de Message (performative) et le contenu,
  - **Message :** envoyé dans : *temps d'envoie écoulé* et reçu dans : *temps de réception écoulé*
- **Temps de l'impact total** : la somme des temps de l'impact de tous les nœuds enfant du nœud courant,
- **Pourcentage de temps du parent** : pourcentage de temps total de l'impact pour le nœud courant divisé par le temps total de l'impact de son parent dans l'arborescence,
- **Pourcentage de temps de la session** : pourcentage de temps total de l'impact pour le nœud courant divisé par le temps total de la session.

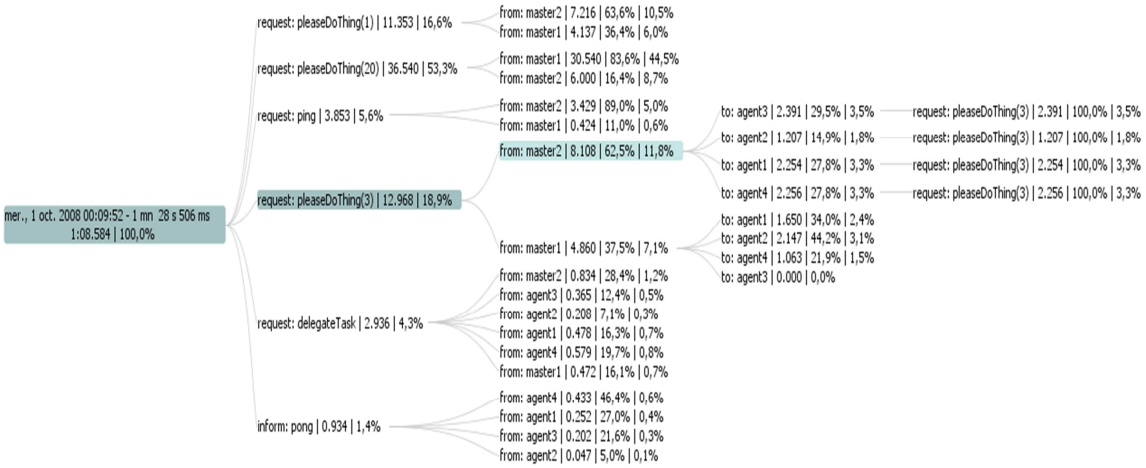


Figure 4.5 **Exemple d'un graph d'appel orienté agent**

3.3. Diagramme espace-temps

L'objectif de diagramme espace-temps (Figure 4.6) est de montrer par rapport au graph d'appel orienté agent plus de détails et de contexte d'exécution sur les performances du système multi-agents.

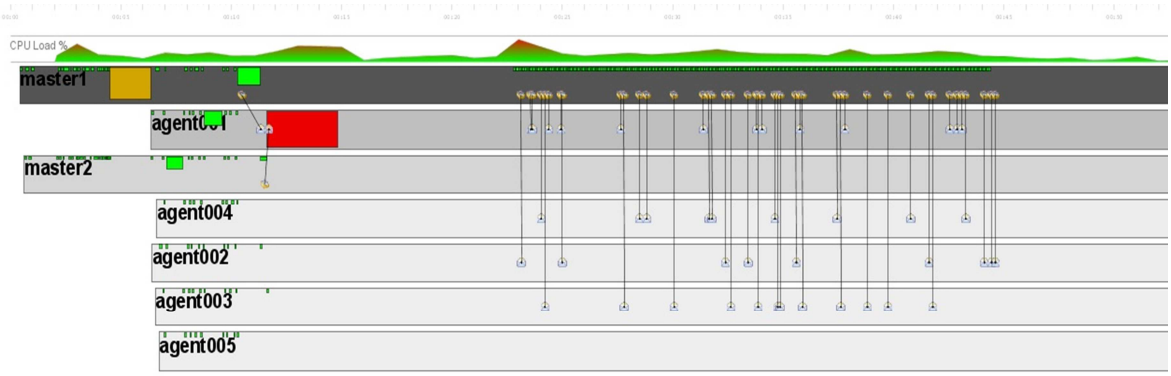


Figure 4.6 **Diagramme espace-temps [Din08b]**

- **La chronologie de la session (Session time line) :** représente le temps d'exécution de l'application en cours de profilage, elle fournit un contexte temporel de la section en cours de visualisation, et aussi de permettre aux programmeurs d'accéder aux divers point de la session.
- **La ligne CPU** : est une représentation graphique de la charge du processeur pour la session courante, le gradient vertical allant de vert (faible utilisation du CPU) au rouge (utilisation élevée du processeur) fournit un sens graphique de la charge du système. Une fenêtre (popup) d'information révèle les statistiques d'utilisation exactes une fois la souris sera au-dessus la ligne.
- **chronologie des agents** : La plus importante caractéristique du diagramme espace-temps c'est la chronologie des agents (agent timeline). Elle présente tous les

évènements liés à la performance et la communication qui se produisent au cours d'une session de profilage. La chronologie de l'agent ne commence qu'au moment où l'agent est créé. En outre, la chronologie de l'agent montre des couleurs différentes qui reflètent la charge associée à l'agent en cours par rapport aux autres agents. [Din08b] décrit en détail les différentes composantes de ce diagramme.

3.4. Détermination des attributs dynamiques orientés agent

Il s'agit dans cette phase de déterminer les attributs dynamiques qui nous permettent d'évaluer la communication dans les applications orientées agents réfactorisées.

Les agents ont tendance à effectuer des actions en réagissant à la réception des messages provenant d'autres agents dans le système. L'impact d'un message peut être conduit au traitement supplémentaire qui doit être pris en considération afin de réagir à l'information qui y est contenue, formuler une réponse ou effectuer une tâche demandée.

Nous introduisons *la mesure de l'impact d'un message* de l'agent (proposé par [Din08a]) d'être utilisée comme un équivalent à la durée de traitement d'une fonction utilisée dans les profileurs traditionnels. Il est important de noter que les évènements provenant de l'environnement de l'agent ne sont pas représentés par cette mesure.

Le profileur AgentSpotter [Din08a] implémente concrètement cette mesure dans la version adaptée spécifiquement avec la plateforme Agent Factory 2.0 [Rem08].

Une autre mesure importante adoptée est celle de la quantité d'information transportée dans le système, nous avons déjà détaillé cette approche d'évaluation de la communication dans le premier chapitre (section 7.1).

3.5. Formalisation des attributs :

Un agent est un processus comportant trois phases successives: réception d'un message, traitement et action.

Une mesure potentielle [Din08a] de $TM\alpha,B$ *:* l'impact du message $M\alpha$ envoyé par l'agent A vers l'agent B, et reçu dans un temps écoulé α , est d'utiliser la quantité totale de temps de calcul utilisée par l'agent B jusqu'à ce que l'agent B reçoive un message $M\Omega$ à partir d'un autre agent X dans un temps écoulé Ω tel que : $\Omega \geq \alpha$.

Soit b une activité de l'agent B déroulé dans un temps t tel que : $\alpha \leq t \leq \Omega$, l'impact du message $M\alpha$ sur l'agent B, $TM\alpha,B$ est donné par l'équation récurrente :

$$T_{M\alpha}B = \sum_{t=\alpha}^{\Omega} bt \qquad (1)$$

La figure 4.7 montre un éclaircissement des concepts présentés dans la section précédente d'une manière graphique :

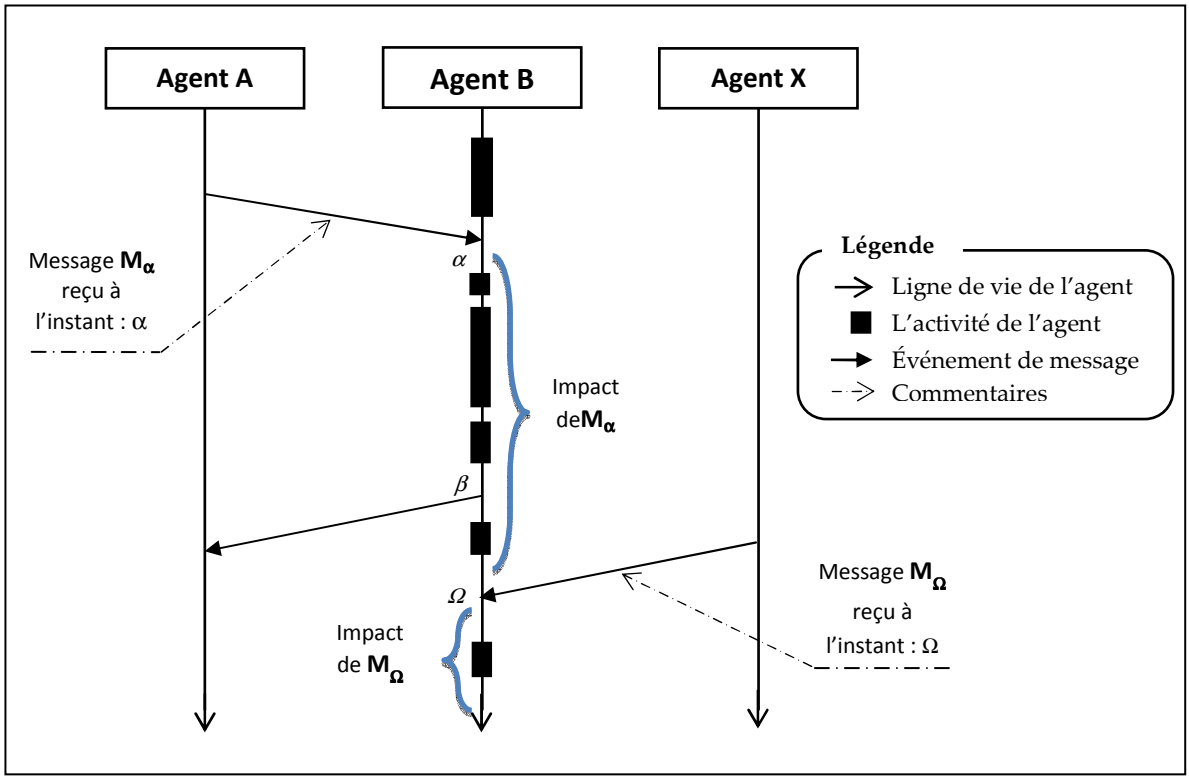


Figure 4.7 **Diagramme : impact des messages échangés entre agents [Din08a]**

Maintenant, on va déterminer l'impact total : Tx,y de tous les messages envoyés par un agent donné x à un autre agent y. Soit M le nombre total de messages envoyés $1 \leq m \leq M$, tel que m l'identifiant unique de l'impact du message, α_m : le temps de réception du message m de x à y et Ω_m la prochaine durée de réception du message provenant juste après m de toute autre source, tel que : $\alpha_m \leq \Omega_m$. L'impact total Tx,y est alors donné par l'équation :

$$T_{x,y} = \sum_{m=1}^{M} \sum_{t=\propto m}^{\Omega m} bt \qquad (2)$$

En appliquant les équations de manière récursive, on peut calculer l'impact total Tx d'un agent x sur N autres agents numérotés $1 \leq a \leq N$ comme suit:

$$T_x = \sum_{\alpha=1}^{N} \sum_{m=1}^{M\propto} \sum_{t=\propto m}^{\Omega m} bt \qquad (3)$$

Enfin, l'impact total TS de tous les K agents numéro $1 \leq k \leq K$ d'une session S, est donné par l'équation:

$$T_s = \sum_{k=1}^{k} \sum_{\alpha=1}^{Nk} \sum_{m=1}^{M\propto} \sum_{t=\propto m}^{\Omega m} bt \qquad (4)$$

Il faut noter que la durée totale de l'activité A_S de la session *S*, est donnée par l'équation:

$$As = Ts + \sum_{k=1}^{k} \sum_{t=\propto s}^{\propto k0-1} bt \qquad (5)$$

Cette méthode proposée par [Din08a] pour calculer l'impact des messages échangés entre agents est imparfaites, des métriques abstraites sont susceptibles d'être développées dans le futur, toutefois elle fournit des informations utiles pour le développement et le refactoring des applications orientées-agents.

4. Présentation de l'approche :

L'approche que nous proposons (Figure 4.8) pour l'évaluation de l'impact du refactroing sur les applications multi-agents est accomplie en deux phases principales: phase de refactoring et phase d'évaluation. Cette dernière est composée de trois niveaux : Conceptuel, analyse dynamique et Evaluation.

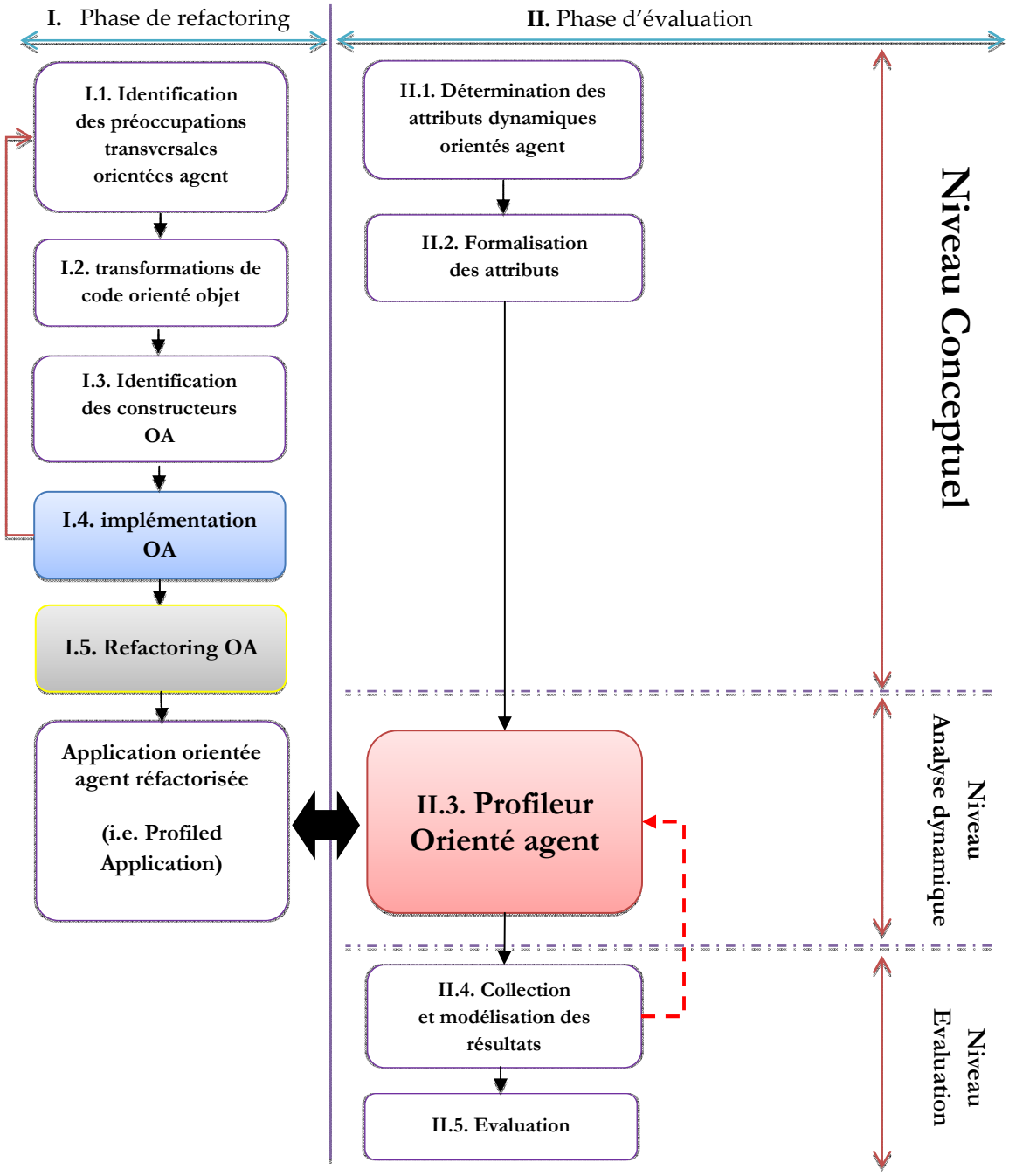


Figure 4.8 **Méthodologie de l'approche proposée**

4.1. Phase de refactoring orienté aspect :

Cette phase est très importante et la qualité de résultats qui en découlent contribue à la détermination celle de la phase d'évaluation. Dans la littérature, aucun outil disponible permettant la réalisation de cette activité. Dans le cadre de notre approche, nous procédons manuellement à la réalisation de cette phase en quatre étapes :

4.1.1. Identification des préoccupations transversales orientées agent

Un système multi-agent est composé d'un ensemble d'agents autonomes qui ont la capacité d'interagir entre eux et avec son environnement dans le but d'accomplir des tâches. Considérant la communication comme étant une propriété de l'agent qui doit être séparée comme un aspect, elle comporte les rôles joués par l'agent en répondant sur un message reçu par un autre agent ou bien d'un évènement provenant de l'environnement. Chaque rôle représente une activité de communication dans un contexte spécifique.

Notre travail dans cette section de la phase de refactoring consiste à séparer la propriété d'interaction (communication) dans des aspects selon le modèle présenté dans [Gar02]. La Figure 4.9 décrit le modèle utilisé dans la phase de refactoring.

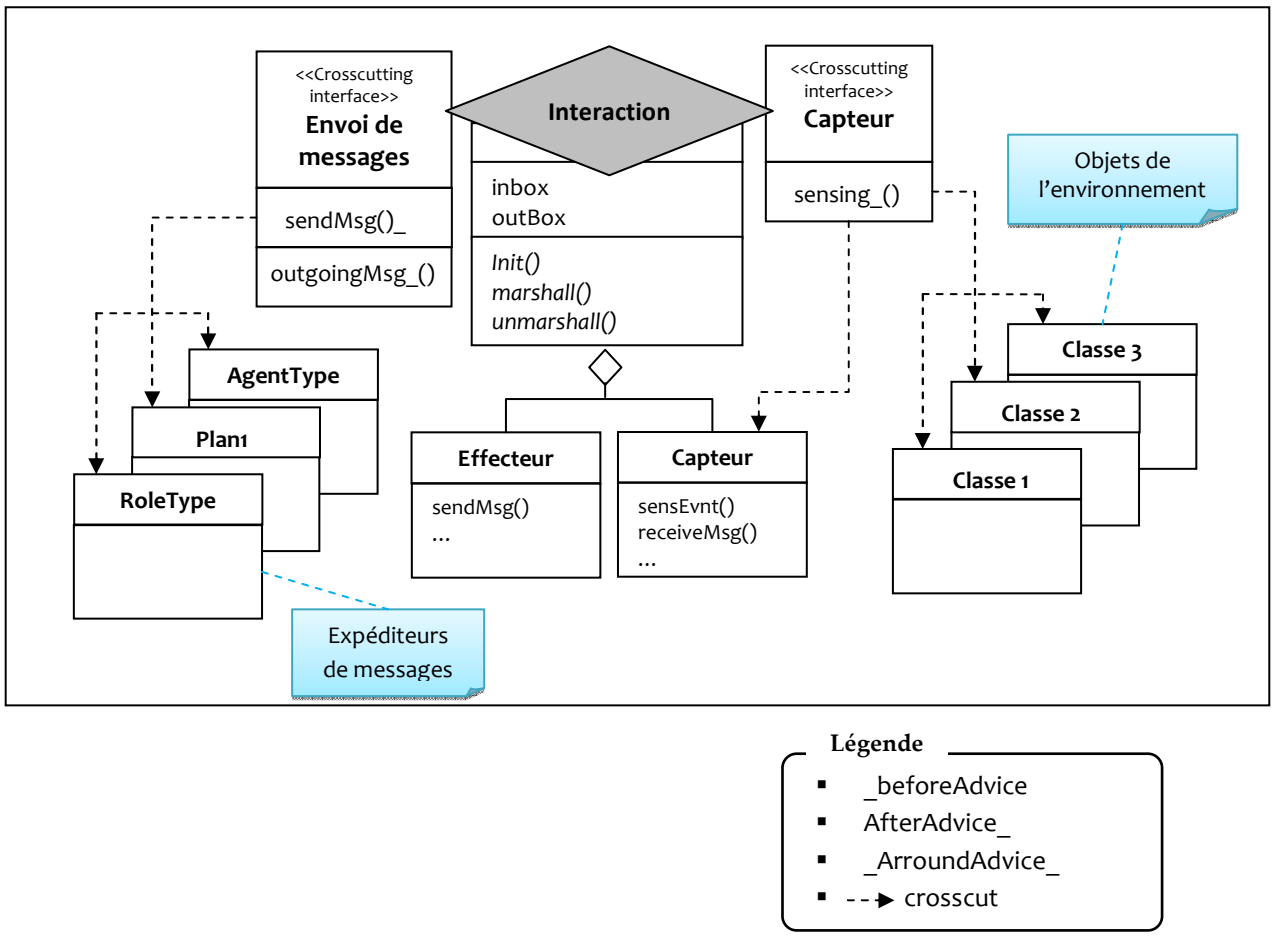


Figure 4.9 **Séparation des propriétés alternatives de l'agent [Gar02]**

4.1.2. Transformations de code orienté objet

Cette étape consiste à la séparation proprement dite de tout code implémentant la préoccupation transversale relative à la communication, tout en éliminant [Fow99] si c'est possible les portions de code : méthodes, classes, instructions, etc. du code source original de notre application. Ces transformations sont applicables dans le but de préparer le code

original pour accepter le refactoring orienté aspect. Elles consistent à l'identification des endroits où il est possible d'intégrer le code aspect en utilisant des marqueurs.

4.1.3. Identification des constructeurs orientés aspect :

Comme nous avons cité dans le chapitre II, Un code advice est un bloc d'instruction associé à une coupe. Il est exécuté avant, après ou autour des points de jonction sélectionnés par la coupe qui lui est associée.

Il s'agit d'identifier les codes advices AspectJ qui doivent être appliqués dans le processus de refactoring. Nous avons basé notre étude sur trois variantes de codes advices :

- ***Around()*** qui permet de définir un bloc d'instructions qui s'exécute autour d'un point de jonction. Il permet éventuellement de remplacer carrément l'exécution d'une méthode. AspectJ fournit la méthode ***proceed()*** qui permet de rendre le contrôle de l'exécution au point de jonction dans un code advice de type ***around*** [Iva02].
- ***after() returning()*** qui signifient après le retour d'une méthode sans exception [Iva02].
- ***Around()*** et ***cflow() :*** définissent le même mécanisme que Around() avec un flot de control sur la coupe sélectionnée [Iva02].

4.1.4. Processus de refactoring

Après avoir sélectionné les différents constructeurs impliqués dans la phase de refactoring, nous allons procéder à la reconstruction et la reformulation de toutes portions de code qui implémentent la communication (Figure 4.10) en utilisant les mécanismes fournit par aspectJ, autrement dit, on va produire une version orientée aspect de la propriété *communication* de l'agent.

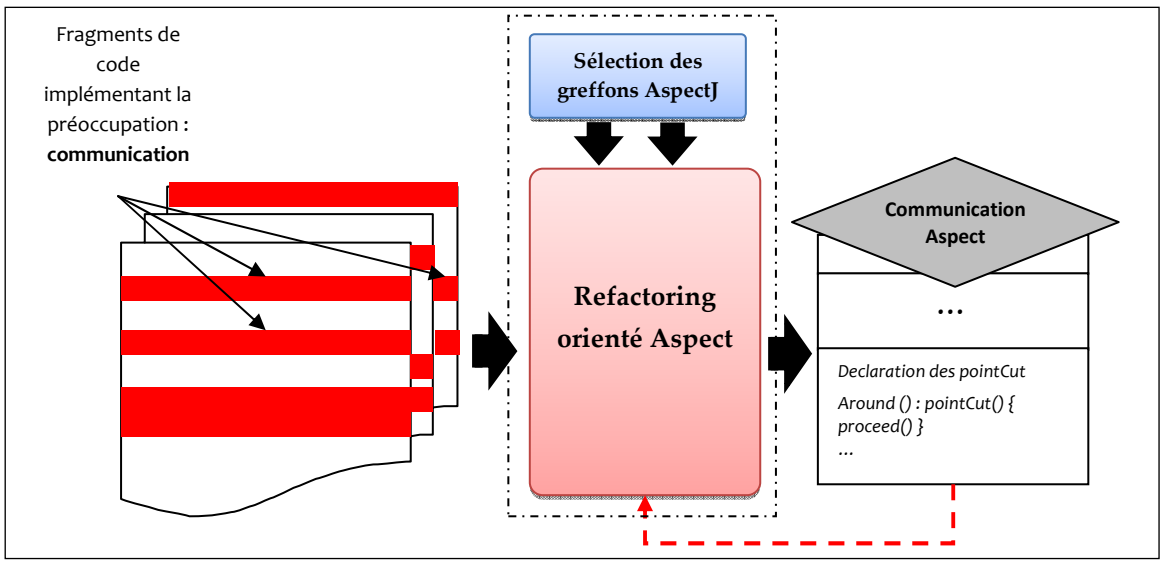


Figure 4.10 **Transformation de la préoccupation *communication* en Aspect.**

Notre objectif dans cette étape est de développer différentes versions d'une même application orientée agent, chacune est réfactorisée d'une manière différente à savoir : version réfactorisée en utilisant des aspects implémentant le code advice *around()*, version réfactorinsée avec *after() returning()*, et la dernière version est réfactorisée avec le code advice *around* combiné avec le control *cflow*.

4.2. Phase d'évaluation :

Après avoir présenté les différents outils impliqués dans notre contribution, nous allons proposer dans ce qui suit les différentes formules mathématiques que nous avons appliqués pour calculer l'impact des différents constructeurs aspectJ utilisés.

4.2.1. Modélisation de l'impact

Pour évaluer l'impact de chaque constructeur apsectJ, nous avons définis des formules de calculs mathématiques qui nous permettent de mesurer l'impact des codes advice impliqué dans la phase de refactoring.

Soit TM_{cAd} le temps de l'impact du message *M* envoyé par un agent quelconque X aux autres agents du système de la version réfactorisée avec des aspects implémentant le code advice *cAd*. Soit TM_{or} le temps de l'impact du message *M* de la version originale (sans refactoring), envoyé par un agent quelconque X vers tous les autres agents. Il est à noter que TM_{cAd} et $TMor$ sont calculés à base de l'équation (4).

Pour calculer l'impact des *nbrAd* (nombre de codes advices impliqués) codes advices appliqué par notre refactoring (dans notre cas *nbrAd* = 3) sur le temps des messages envoyés par un agent X aux autres agents *nbrAg* (Nombre d'agents dans le système), pour une durée de session *S*, nous introduisons la mesure θ définit par l'équation : (1')

$$\theta = \frac{\sum_{i=1}^{nbrAg}\sum_{j=1}^{nbrAd}(\mathrm{T}Mor\ i - \mathrm{T}McAd\ ij)}{S} \quad \text{Tel que S} > 0 \qquad (1')$$

On pose: $\partial = \sum_{i=1}^{nbrAg}\sum_{j=1}^{nbrAd}(\mathrm{T}Mor\ i - \mathrm{T}McAd\ ij)$

La moyenne MOY*cAd* de l'impact des constructeurs aspectJ par agent est donnée par (2') :

$$\mathrm{MOY}cAd = \frac{\partial}{nbrAg} \quad \text{Tel que } nbrAg > 0 \qquad (2')$$

Le taux de dépassement d'un code advice donnée *cAd* sur le temps de l'impact des messages envoyés par l'agent X aux autres agents par une durée de session S, est donné par l'équation (3')

$$\delta = \frac{\sum_{j=1}^{nbrAg}(\mathrm{T}Mor\ j - \mathrm{T}McAd\ \ cAd\ j)}{S} \qquad (3')$$

Pour calculer le taux de surcharge imposé par les codes advices sur chaque agent récepteur du message *M*, l'équation (4') définit le taux de surcharge pour un constructeur aspectJ *cAd* donné :

$$\lambda = \frac{\sum_{j=1}^{nbrAg}(\mathrm{T}Mor\ j - \mathrm{T}McAd\ \ cAd\ j)}{\mathrm{T}Mor} \quad \text{Tel que } \mathrm{T}Mor > 0 \qquad (4')$$

5. Implémentation

Pour valider notre approche d'évaluation, nous avons développé une application multi agent sous la plateforme agent AgentFactory. Cette application est réfactorisée de différentes manières dans le but d'évaluer l'apport des constructeurs de la programmation orientée aspect mentionnée dans la première partie de ce chapitre, sur le comportement des agents après l'exécution du système en question et surtout l'impact de ces constructeurs sur l'interaction des agents.

Notre étude de cas présenté dans cette section est une implémentation d'un scénario simple entre deux types d'agents : ***master*** et ***worker.*** Les agents *masters* demandent aux agents *workers* de réaliser des petites, moyennes et longues tâches. Si un agent worker a récemment été surchargé, il peut refuser d'exécuter la tâche requise, de temps en temps les agents *masters* vont déléguer l'attribution des tâches aux agents *workers*, auquel cas l'agent worker devient un agent master pour un bref délai.

L'application référence présentée est exécutée avec des entrées bien déterminées, le nombre total des agents dans le système égale *sept*, dont *deux* agents jouent le rôle d'un ***master*** (*master1* et *master2*) et le reste des agents jouent le rôle d'un ***worker*** (*worker1*, *worker2*, …, *worker5*). Au début tous les agents *workers* sont libres et dans ce cas l'un des deux agents *masters* vont choisir aléatoirement un agent *worker* pour lui affecter une tâche à réaliser. Ensuite, le deuxième agent *master* va chercher un autre agent pour lui affecter une autre tâche dans ce cas si l'agent *worker* choisi par le deuxième agent *master* est déjà en cours d'effectuer une tache (surchargé), ce dernier refuse la demande de l'agent *master*, et la tâche en cours est affectée à un autre agent.

5.1. Modélisation des protocoles d'interactions entre agents

La figure 4.11 montre un template générique du contract net protocol de la FIPA (Foundation for Intelligent Physical Agents), dans cette figure, le cadre avec les traits interrompus dans le coin haut droit indique que le package est un template. Les paramètres du template sont divisés en trois catégories :

- Les paramètres de rôle,

- Les contraintes et,
- Les actes de communications.

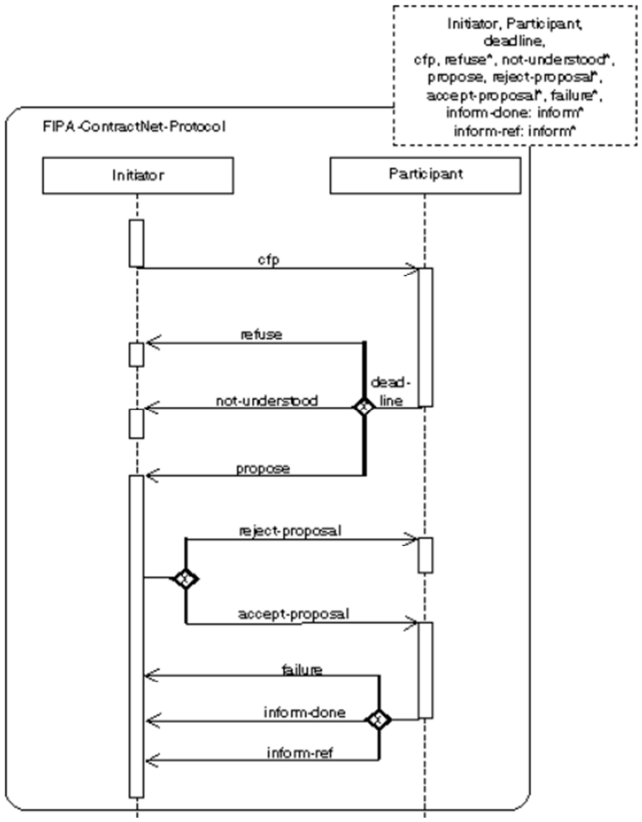


Figure 4.11 **protocole d'interaction contract-Net de la FIPA [FIPA01]**

La figure 4.12 applique le contract net protocole de la FIPA à un scénario possible sur les sorties de notre étude de cas. Notons que les agents *Initiator* et *Participant* sont instanciés respectivement par les agents *Master* et *Worker* et que le *call-for-proposal* est devenu *request(length),* tel que *length* représente la quantité de travail que l'agent *master* a demandé d'effectuer, comme nous l'avons mentionné dans le paragraphe précédent. Notons aussi que dans ce scénario, deux formes de refus ont été envoyés par l'agent *worker* : *refuse1* et *refuse2*.

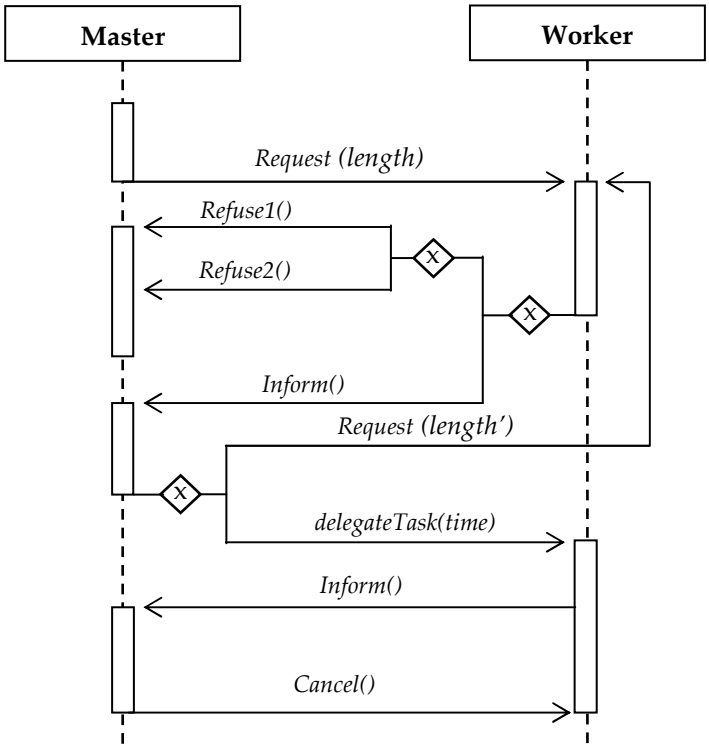


Figure 4.12 **Instanciation du protocole contract-Net pour notre étude de cas**

6. Evaluation

6.1 Temps de l'impact des messages reçus :

Comme nous avons cité précédemment le processus d'évaluation se déroule en trois étapes : exécution, profilage et collection et interprétation des données. Dans notre cas d'étude nous avons exécuté l'application référence avec ses quatre versions (originale, réfactorisée avec l'advice : ***around(),*** réfactorisée avec l'advice ***after() Returning()*** et la dernière version réfactorisée avec ***around()*** combiné avec ***cflow(),*** pendant dix minutes.

Après avoir modélisé les données avec l'outil de visualisation AgentSpotter station intégrée avec notre profileur orienté agent, nous avons procédé à une reformulation des données collectées dans le but de les représenter graphiquement sous forme de graph avec MS Excel version 2010.

La figure 4.13 montre le graph d'appel orienté-agent de notre application référence après dix minutes d'exécution, pour les quatre versions : *originale, around, after returning* et *aroun&cFlow.*

Le graph d'appel orienté agent montre bien qu'il y a une différence remarquable entre les trois versions de l'application référence en terme de temps de l'impact des messages échangés entre agents (mentionnée avec le rectangle rouge dans la figure 4.13). L'agent *master1* a un impact total égal à 4.26 minutes sur les agents *workers* pour la version originale, par contre le temps de l'impact pour les trois autres versions réfactorisées est

supérieur strictement par rapport à la version originale (égal à 7.37 minutes, 7.35 minutes, et 7.38 minutes pour around()&cFlow(), Around() et after() returning() respectivement) pour une durée d'exécution égale à 10 minutes.

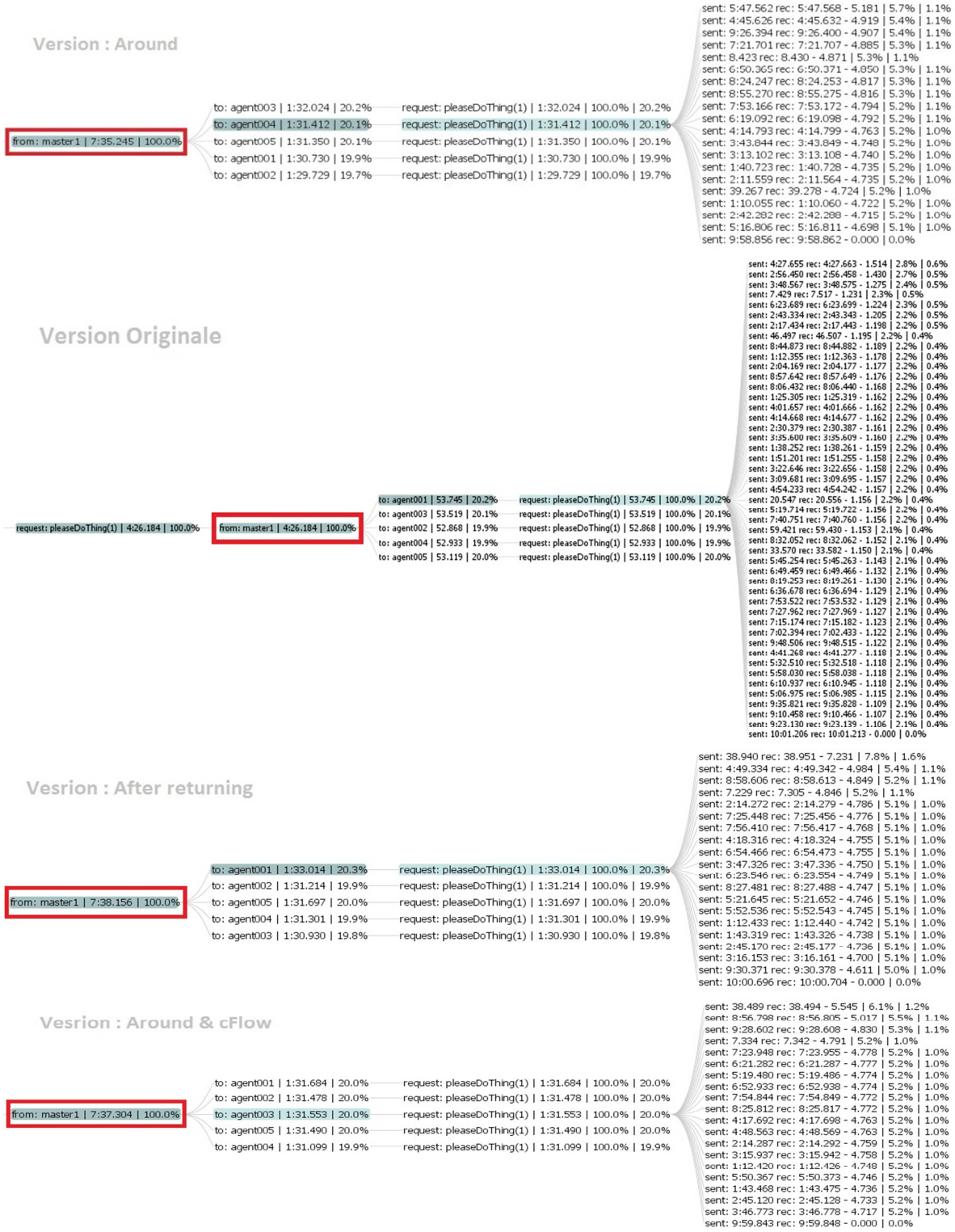


Figure 4.13 **graph d'appel orienté-agent pour une durée d'exécution égale à 10 minutes**

Dans tous les cas, le paramètre passé par la requête (pleaseDoThing) est liée à la quantité de travail demandé par l'agent *master* aux agents *workers* qui est identique pour tous les agents, ce qui implique que les durées d'activités des différents agents *workers* sont proches pour chaque version.

La figure 4.14 présente le temps de l'impact détaillé (en minites) des messages envoyés par l'agent *master1* aux agents *workers* pour les différentes versions de l'application référence. Le graphe montre bien que les codes advices aspectJ utilisés influent globalement et différemment d'une façon indésirable sur l'impact des messages envoyés par l'agent *master*. Initialement, on peut remarquer que les constructeurs aspectJ augmentent le temps de l'impact à cause de la surcharge (*overhead*) imposée par le compilateur aspectJ et son mécanisme d'introduction (weaning). Nous allons détailler cet impact négatif dans la section de discussion.

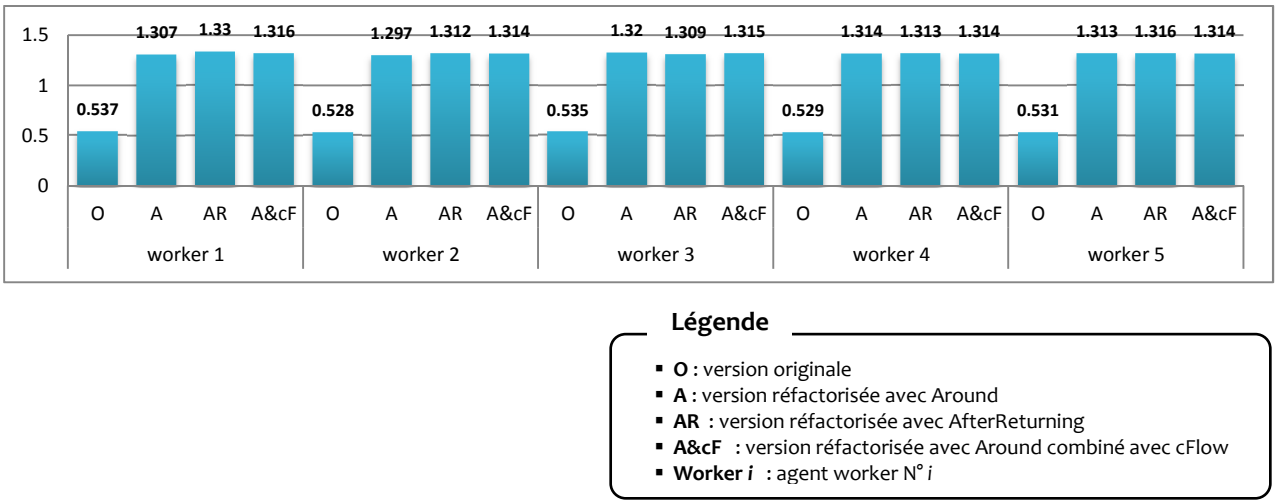


Figure 4.14 **Temps de l'impact des messages envoyés par l'agent : Master1 (Durée de session égale à 10 minutes)**

En outre, le temps de l'impact des messages envoyés par l'agent *master1* diffère d'un agent *worker* à un autre. L'impact de l'agent *master1* sur l'agent *worker1* pour la version originale égal à 0.537 secondes, qui représente le plus grand impact de l'agent master1 par rapport aux autres *worker(s)*, 0.528, 0.535s, 0.529s et 0.531 s pour les agent *worker2*, *worker3*, *worker4* et *worker5* respectivement. Cela est dû à deux raisons possibles.

1. La tâche désignée à l'agent *worker* inclut des portions de code qui utilisent le temps de traitement processeur ou bien les ressources offertes d'une manière inefficace, et par conséquent introduit des retards inutiles.
2. La deuxième raison sur laquelle l'impact de l'agent *master1* n'est pas identique sur l'agent *worker* est le déséquilibre du système à cause des évènements imprévus.

Ces deux raisons montrent bien l'utilité du diagramme espace-temps pour examiner le timing des messages en question.

La figure 4.15 montre le temps total de l'impact des messages envoyés par l'agent *master1* aux agents *worker* pendant une durée d'exécution égale à 10 minutes.

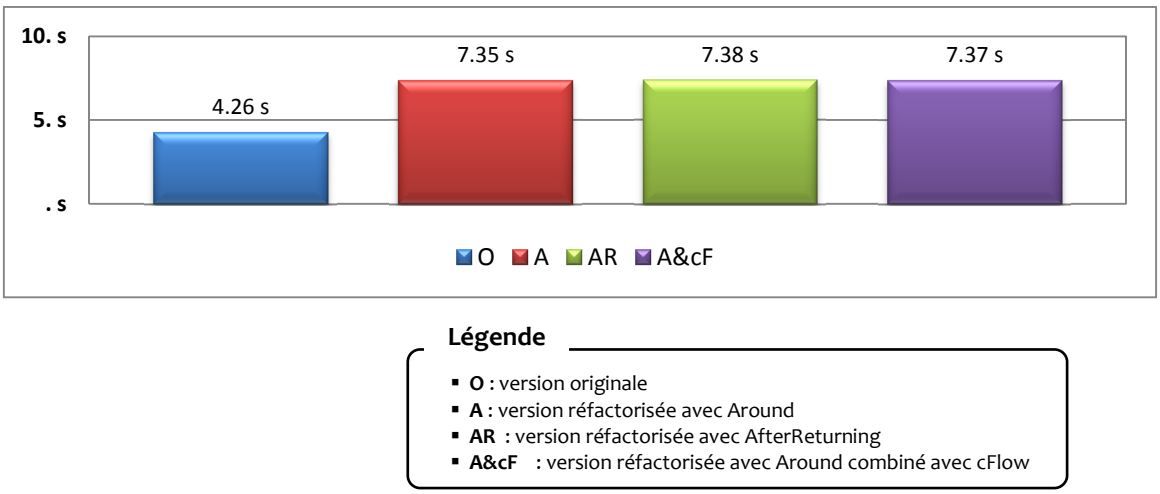


Figure 4.15 **Temps total de l'impact des messages envoyés par l'agent *master1***

Pour montrer l'impact de refactoring orienté aspect, nous avons calculé le taux de dépassement (positif ou négatif) selon l'équation (3') pour chaque constructeur aspectJ par rapport à la version originale (figure 4.16) en terme de temps de l'impact des messages envoyés par l'agent *master*.

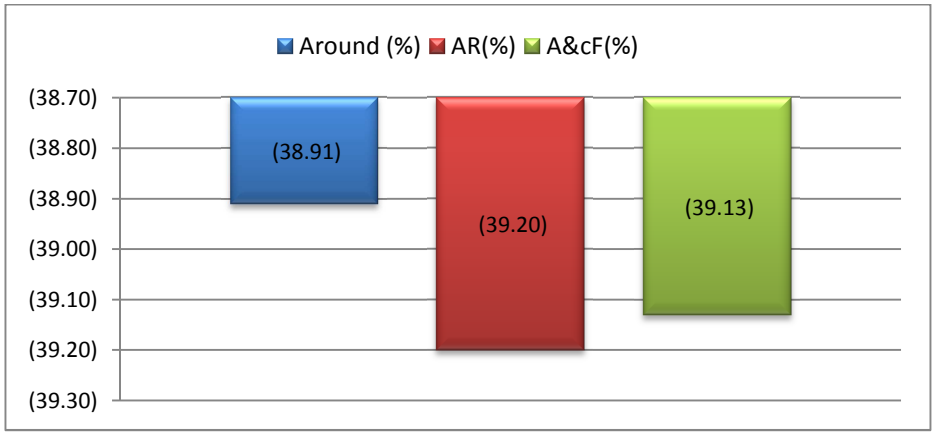


Figure 4.16 **Taux de dépassement pour chaque advice AspectJ par rapport à la version originale**

On peut remarquer que l'advice **afterReturning** nous a apporté une augmentation importante dans le sens négatif égale à 39.20% dans le temps de l'impact des messages envoyés par l'agent *master1* par rapport à la version non réfactorisée, et l'advice **around()** combiné avec **cFlow**() montre une augmentation moins que la précédente égale à 39.13%, par contre l'advice **around** montre un dépassement très petit de 38.91%. Par conséquent, un impact positif nous apporte une diminution dans le temps de l'impact des messages envoyés par l'agent *master1*.

Nous pouvons remarquer que les résultats présentés par notre évaluation explique clairement que le style du refactoring en terme de constructeurs aspectJ sélectionnés

durant la phase de séparation des préoccupations transversales orientés agent influe négativement sur le comportement des agents. Nous allons détailler cet impact prévu dans les sections suivantes.

Le graph suivant (figure 4.17) présente la moyenne de l'impact selon l'équation (2') de chaque advice aspectJ par rapport aux nombre total d'agent dans le système.

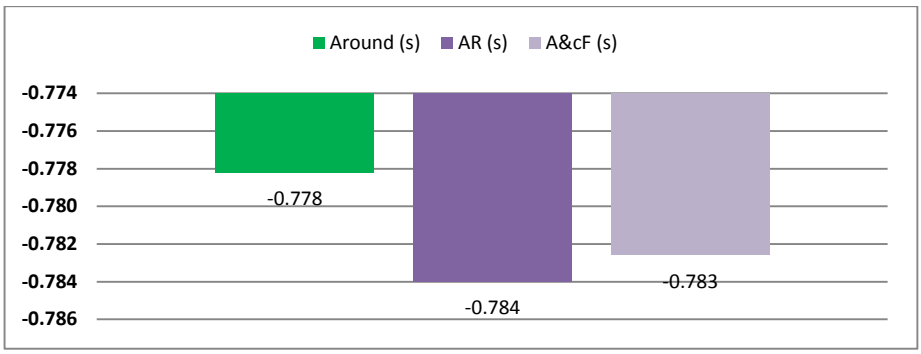


Figure 4.17 **la moyenne de l'impact de chaque constructeur AOP par rapport au nombre total d'agent**

Après avoir présenté l'impact des différents constructeurs aspectJ exprimé avec les trois codes advices choisis dans notre travail, maintenant nous allons montrer l'impact de ces constructeurs sur l'unité de base de notre profileur qui est l'agent.

La figure suivante montre l'impact (calculé selon l'équation (4')) de chaque Advice AspectJ (en %) pour chaque agent worker dans le système.

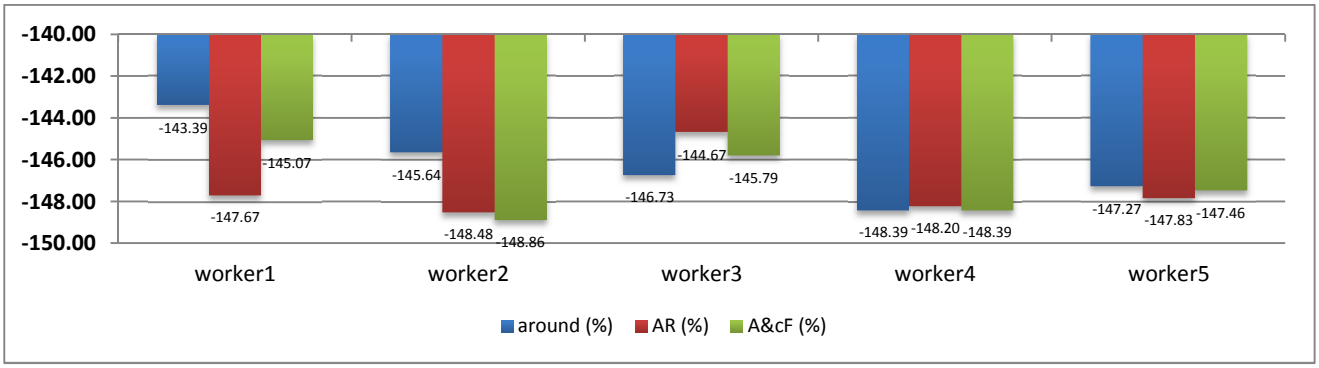


Figure 4.18 **Taux de surchage de chaque code advice sur chaque agent.**

6.2. Nombre de messages échangés :

Nous allons aborder dans cette section l'évaluation de la communication selon l'approche normale présentée dans le premier chapitre (proposé par [Hus10]), qui consiste à quantifier le nombre de messages échangés. Notre objectif derrière cette évaluation est de montrer l'impact des codes advices AspectJ utilisés sur la quantité d'informations échangées.

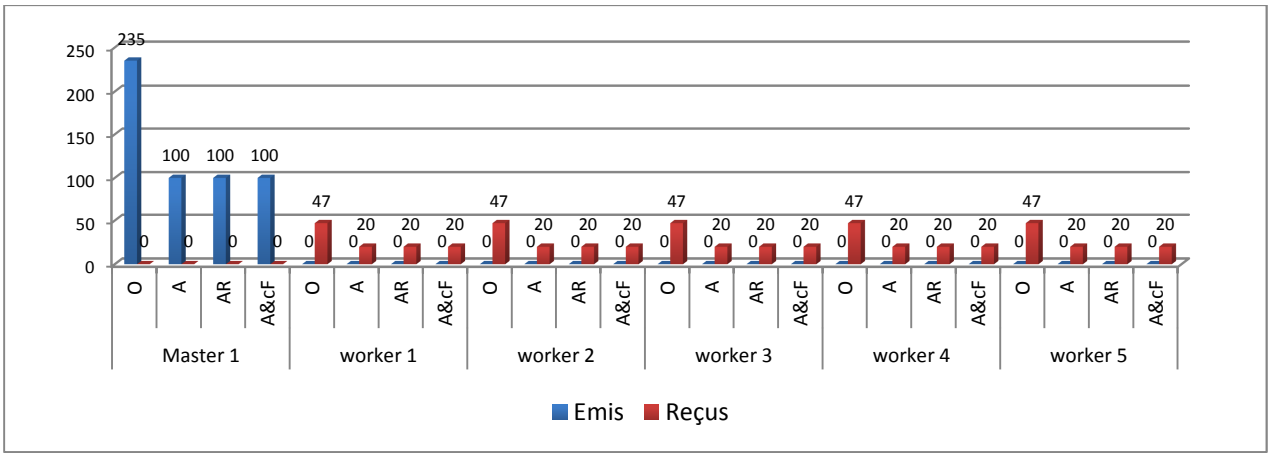


Figure 4.19 **Nombre de messages échangés par chaque agent dans le système pour chaque advice AspectJ**

Le graph de la figure précédente montre que les codes advices aspectJ n'ont pas un impact sur le nombre de messages échangés dans le sens où le nombre de messages émis par un agent est égal au nombre de messages reçus par l'autre. Idem pour les autres versions. L'évaluation de nombre de messages échangés a pour but de montrer que les constructeurs aspectJ n'apportent pas un déséquilibre dans le système en terme de communication pertinente. Autrement dit, il n'existe pas des messages émis par l'agent *master1* qui ne sont plus reçus par les agents *worker*.

D'un autre côté, l'agent *master1* a envoyé 235 messages pour la version originale qui est équivalent au nombre de messages reçus par les agents *worker* (47*5 =235). Par contre, le nombre de messages émis par l'agent *master1* pour les versions réfactorisées est inférieur à celui de la version originale, (égale à 100). Donc, nous avons une diminution égale à 57% (figure 4.20) de messages émis, due à la surcharge imposée par le compilateur aspectJ.

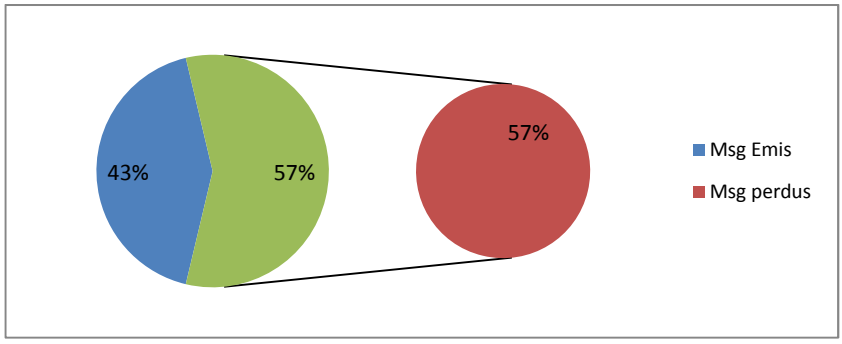


Figure 4.20 : **Pourcentage des messages non envoyés par rapport à ceux envoyés par l'agent master1 pour les différentes versions refactorisées**

Idem pour les messages reçus (figure 4.21), le refactoring orienté aspect impose une dégradation de nombre de messages reçus par les *workers* égale à 57%.

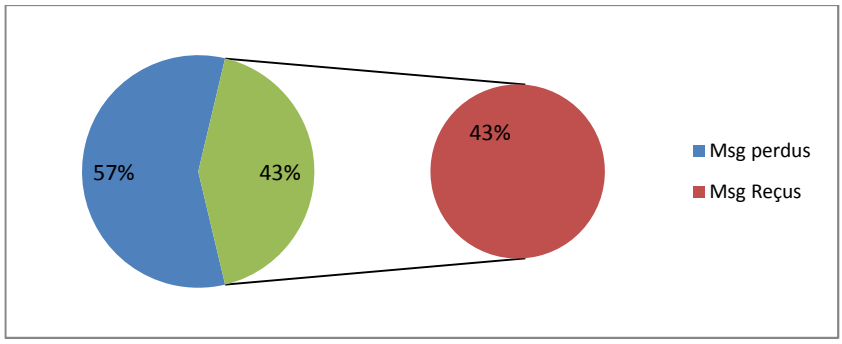


Figure 4.21 : **Pourcentage des messages non reçus par rapport à ceux reçus effectivement par les agents *workers* pour les différentes versions refactorisées**

7. Discussion :

La métrique proposée dans la section (**4.4**) pour mesurer le temps de l'impact des messages envoyés par l'agent a un nombre de lacunes. Elle fonctionne clairement sur l'hypothèse que les actions de l'agent sont directement liées aux messages reçus par lui. L'impact du message sur un agent est égal à la somme des temps d'exécution de toutes les actions effectuées par l'agent entre la réception de ce message et le message suivant. Cela signifie que la métrique proposée peut ne pas être adaptée à certains types de SMA, où l'action de l'agent n'est pas liée uniquement à la communication avec les autres agents.

Le principal inconvénient de telle approche est qu'il n'y a pas un lien de causalité prouvable entre la réception du message et l'exécution d'une action. Par exemple, un agent détecte avec ses capteurs un changement dans l'environnement qui peut exiger une réaction, ainsi lorsque les actions sont exécutées comme une conséquence directe de la réception d'un message ACL, il n'est pas garanti que l'ensemble des actions ont été effectuées avant la réception du message suivant.

Idéalement, la meilleure méthode pour mesurer l'impact des messages reçus est de contrôler le processus de raisonnement interne de l'agent, de manière à identifier les actions qui sont exécutées en tant que résultat direct de la réception d'un message et ne prendre que ceux qui sont comptés dans le calcul de l'impact des messages. Cependant, le processus de raisonnement dépend fortement de la plateforme agent et montre un travail important pour permettre à *agentspotter* de s'adapter avec telle plateforme agent autre qu'AgentFactory.

Concernant le processus de refactoring orienté aspect, l'application référence réfactorisée avec ses différentes versions montre bien l'existence d'une charge de traitement supplémentaire due à l'utilisation des constructeurs aspectJ. Les aspects qui utilisent le code advice *after Returning* augmente considérablement la mesure de l'impact des messages envoyés : surcharge égale à 147,37% (figure 4.22), ainsi la surcharge imposée par les aspects qui implémentent le code advice *around* combiné avec *cFlow* est inférieur à la précédente, égale à 147.11 %, mais supérieur à celle du code advice around

seul qui est égale à : 146.28%, ce qui implique que la primitive **cFlow** est la raison de cette surcharge.

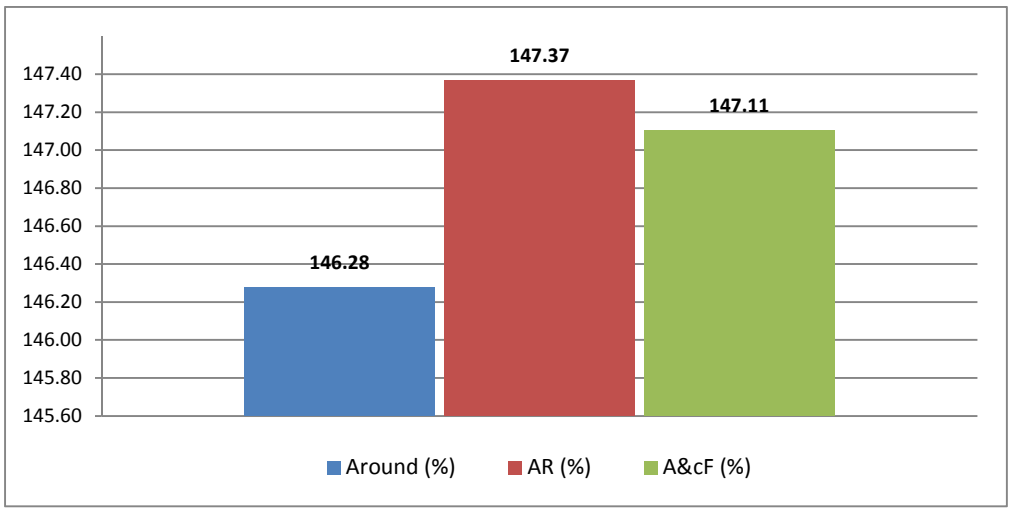


Figure 4.22 **Surcharge enregistrée sur le système**

La primitive *cFlow(pointCut)* consiste en une vérification dynamique de la coupe (pointcut), ce qui implique un enrichissement du code machine (ByteCode) par le compilateur aspectJ qui sert à contrôler tous mouvements du point de jonction mentionnée dans la coupe. Par exemple, le compilateur aspectJ va ajouter du code qui enregistre l'historique de l'empilement et le dépilement des appels à une méthode donnée.

Toutefois, les aspects qui utilisent le code advice *around* impose une surcharge inférieure que la précédente, malgré elle est valeureuse en terme de temps de l'impact des messages envoyés, cette surcharge explique bien l'utilisation de la primitive *proceed()* fournit avec aspectJ qui permet de rendre le contrôle de l'exécution au point de jonction dans un code advice de type *around.*

Nous avons ainsi procédé à une réfactorisation générale qui utilise un aspect global implémentant un code advice *around* qui capture tous les appels à toutes les méthodes de notre application référence dans le but d'implémenter un profileur simple (Figure 4.23) qui mesure le temps d'exécution de chaque méthode.

Notre objectif derrière cette implémentation se résume en :

- La programmation orientée aspect fournit un outil de qualité pour le profilage vis-à-vis les outils d'instrumentations offerts qui sont basés sur l'instrumentation dynamique au niveau byteCode,
- Le tissage d'aspect ne peut pas, par principe, vérifier le passage du contrôle de flot dans les structures de contrôle, ou suivre l'évolution de la valeur d'une variable.

```
public aspect profilingAspect {
        pointcut publicOperation() : execution(public * *.*(..));

        Object around() : publicOperation() {
                long debut = System.nanoTime();
                Object ret = proceed();
                long fin = System.nanoTime();
            System.out.println(thisJoinPointStaticPart.getSignature() + " a pris " + (fin-debut)
                + " seconds");
  return ret;       } }
```

Figure 4.23 **Mesure du temps d'exécution de méthode**

8. Limites des techniques de la POA pour les applications orienté agents

8.1 Préoccupations transversales orientées agent non séparables :

Le refactoring orienté aspect a montré son importance dans le processus de séparation des préoccupations transversales orientées agent, nous citons à titre d'exemple la mobilité, l'apprentissage, l'autonomie. La conception et l'implémentation ont montré des améliorations expressives en termes de séparation des préoccupations pour ces propriétés. Cette observation prouve l'efficacité de l'approche orientée aspect avec ces constructeurs pour séparer les préoccupations transversales spécifiques à l'agent.

Cependant, il existe des préoccupations transversales orienté agent qui souffre de telles séparations en utilisant l'approche aspect, par exemple, la séparation de la préoccupation *communication* (qui présente notre choix dans ce travail), dans des aspects génère un couplage implicite et donc une surcharge imprévue en terme de temps de communication à cause de traitement distribué entre les aspects, et les classe en même temps, et non sur l'un des deux (e.g. un rôle d'un agent dispersé sur plusieurs classes et aspects, le traitement d'un message reçu [Gar03] qui nécessite un passage à travers un grand nombre de classes et d'aspect en même temps).

Une solution initiale proposée à ce problème est de séparer complètement le traitement des messages reçus dans des aspects, par conséquent, il existe des cas où la séparation des préoccupations liées à l'agent conduit à des solutions plus complexes.

8.2 Chevauchement des préoccupations :

Il y avait certaines préoccupations qui se sont montrées comme se chevauchant. Par exemple, l'adaptation et l'apprentissage sont deux exemples classiques de préoccupations qui se chevauchent. L'implémentation des aspects d'apprentissage comprend les mêmes comportements déjà implémentés par les aspects d'adaptation. Afin d'éviter la duplication du code nous avons exposé ce comportement commun dans l'interface des aspects d'adaptation ainsi que les aspects de l'apprentissage peuvent y accéder, ou bien de séparer le comportement commun dans un super-aspect.

Il existe d'autre limites que présentent les techniques de la programmation orientée aspect sur le processus de refactoring orienté aspect des systèmes multi-agents. Nous avons focalisé uniquement sur les deux éléments précédents, Garcia et al [Gar03] présente d'autres limites concernant les conflits inter-aspect et le manque d'inconscience, etc.

8.3 Refactoring orienté aspect et autonomie :

Un des fondements du paradigme multi-agent est de considérer un agent comme *autonome*. L'autonomie signifie que l'agent peut agir sans intervention directe de la part d'un utilisateur ou d'autres agents et qu'il contrôle ses propres actions et son état interne. Cette autonomie constitue la différence principale entre un agent et un objet. Un objet est doté de méthodes qu'il suffit d'invoquer pour en obtenir l'exécution.

Notre étude de cas présente une sorte d'autonomie de la part des agents *workers*. Cette situation est modélisée par le fait que l'agent *worker* refuse une requête provenant des agents *masters*, dans le cas où il est en train de réaliser une tâche, et donc surchargé.

Le graph d'appel orienté agent (Figure 4.24) montre que certaines demandes de l'agent master1 ont un impact égal à 0.0, ce qui signifie en pratique qu'ils ont été ignorés (aucune action a eu lieu à la suite de réception de ces messages). Autrement dit, si l'agent *master1* envoi une requête à l'agent *worker* et tout de suite l'agent *master2* envoi la même requête, l'agent surchargé refuse tout simplement d'exécuter la demande.

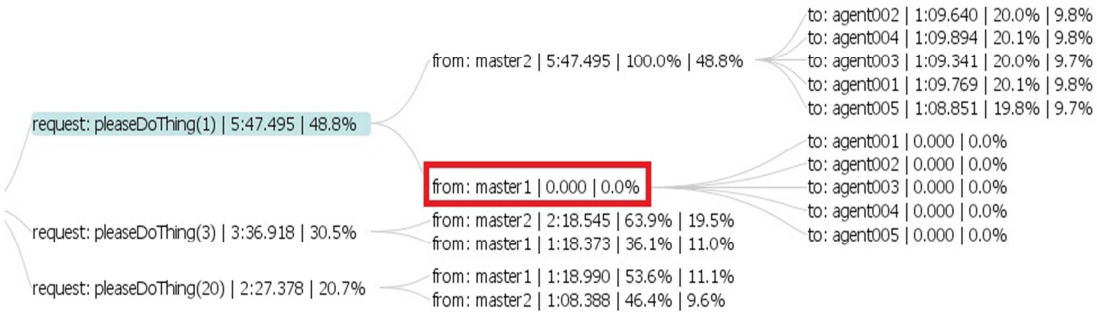


Figure 4.24 : **Requête avec un impact nul**

Dans ce cas, on ne peut pas mesurer l'effet de la communication refactorisée, sur les agents worker. Le refus résultant de l'état interne de l'agent surchargé nous empêche de savoir si le refactoring orienté aspect adopté par la séparation de tous fragments de code (méthodes ou classes) implémentant la communication, a un impact sur le comportement de l'agent après avoir reçu une requête de la part des agent master, tout en se basant sur le critère : temps de l'impact de message.

Par conséquent, nous constatons que les préoccupations transversales implémentant la communication qui ont été migrés vers des aspects dépendent fortement du degré de l'autonomie de l'agent. Si l'agent est purement autonome, l'impact du refactoring orienté

aspect est nul, par contre, si l'agent est purement réactif, l'impact du refactoring orienté aspect dépend de style de refcatoring impliqué par le développeur, et de constructeurs de la PAO choisis.

9. Conclusion

Nous avons présenté dans ce chapitre notre approche pour l'évaluation de l'impact des constructeurs aspectJ sur le comportement des agents tout en se basant sur les outils de l'analyse dynamique offerts comme le profilage. Il est important de noter le manque important des applications orientées agent réfactorisées. Pour cela, nous avons adopté une stratégie bien déterminée pour remédier au problème de la séparation des préoccupations transversales orientées agent mentionnée dans ce mémoire avec la propriété de la communication des agents.

La mesure présentée comme critère d'évaluation spécifique pour le contexte orienté agent souffre de lacunes qui concernent les évènements provenant de l'environnement. Elle se concentre uniquement sur l'impact des messages provenant d'autres agents.

Ce chapitre montre une évaluation qui s'applique uniquement sur les trois constructeurs aspectJ cité précédemment. Ce choix n'est pas fortuit puisque si on pense à la variété des constructeurs fournit par le langage AspectJ, c'est difficile de couvrir tous les constructeurs et notamment pour les appliquer dans un contexte orienté agent. Les résultats présentés dans ce mémoire subirent un taux d'erreur epsilon et ça dû à la nature de la manipulation des chiffres et des équations mathématiques. Notre objectif dans ce cas est d'optimiser les résultats dans le sens à minimiser epsilon avec toutes méthodes possibles.

Conclusion générale

Le refactoring est une technique utilisée pour modifier le code source d'une application dans l'objectif d'améliorer sa qualité sans altérer son comportement du point de vue de ses utilisateurs. Elle vise, essentiellement, la réduction des coûts de maintenance et l'amélioration de la qualité de service au sens technique du terme (performance et fiabilité). Une des dérivées de cette technique, on trouve, le Refactoring Orienté Aspect. Cette nouvelle technique diffère de celle classique par l'implication des constructeurs de la Programmation Orientée Aspect.

Le refactoring orienté aspect des systèmes multi agents est une tâche compliquée vis-à-vis le manque important des outils automatiques ou semi-automatiques qui permettent de migrer le code objet vers le code aspect tout en se basant sur les techniques de l'aspect mining par exemple. Pour cela, le refactoring appliqué dans ce mémoire est entièrement manuel.

Dans ce mémoire, nous avons développé une nouvelle approche, basée sur une analyse dynamique, pour l'évaluation de l'impact du Refactoring Orienté Aspect sur les performances des applications multi-agent. Cette approche consiste à identifier et proposer des critères d'évaluation dynamique spécifique à l'agent qui prend en considération la communication entre les agents comme attribut principal.

Il s'agit d'utiliser les profileurs dédiés comme un outil d'évaluation des performances d'un SMA durant son exécution. L'analyse dynamique permet d'obtenir des résultats plus précis que par une analyse statique pour des exécutions concrètes. Le type d'analyse dynamique adopté est l'analyse dynamique hors ligne (offline), qui sert à évaluer le programme après son exécution, il permet ainsi de diminuer l'impact sur l'exécution et enfin d'enregistrer une quantité d'information énorme (proportionnelle aux temps d'exécution).

Les profileurs disponibles ne prennent pas en considération les spécificités inhérentes aux systèmes multi agents (e.g. autonomie, coopération, mobilité, etc). En outre, le facteur commun entre ces profileurs est l'évaluation des performances (temps CPU, espace mémoire, nombre de classes chargées, etc) des programmes java sans mettre en cause le type du programme en question.

Dans le cadre de notre travail, nous avons opté pour l'outil de profilage AgentSpotter, pour les avantages qu'il procure. Il est adéquat pour notre analyse dynamique avec son système de visualisation qui permet de modéliser les interactions entre les agents par le graph d'appel orienté agent.

Les résultats obtenus par notre évaluation montrent clairement la surcharge imposée par le compilateur ajc, et les différentes implémentations des codes advices proposées par le langage AspectJ, ce qui prouve nos prévisions après le balayage des travaux élaborés dans la mesure des systèmes implémentés avec AspectJ, et justifier même avec le site officiel de aspectj [AspectJ] qui déclare que l'implémentation actuelle de AspectJ affecte la performance du système : *" There is currently no benchmark suite for AOP languages in general or for AspectJ in particular. It is probably too early to develop such a suite because AspectJ needs more maturation of the language and the coding styles first. Coding styles really drive the development of the benchmark suites since they suggest what is important to measure…"*

Comme perspectives à moyen termes, nous envisageons de:

- Proposer un nouveau type de refactoring orienté aspect qui a un impact positif sur le comportement du système multi agent pendant l'exécution en terme d'amélioration des propriétés de l'agent comme : le temps de réponse, la communication pertinente, etc.
- Proposer un Framework générique qui permet la réfactorisation orientée aspect automatique ou semi-automatique des préoccupations transversales orientées agent.
- Intégrer le processus du refactoring orienté aspect dans la phase de modélisation et de conception dans le cycle de développement des SMAs.
- Proposer d'autre critères dynamiques orientés agent qui mesurent les autres attributs de l'agent tels que : la mobilité, l'apprentissage, … etc; et aussi des métriques pour évaluer l'interaction entre l'agent et son environnement non seulement entre les agents seuls.

Bibliographie

[Aar03] Aaron Helsinger, Richard Lazarus, William Wright, and John Zinky. "**Tools and techniques for performance measurement of large distributed multiagent systems"**. In AAMAS '03: Proceedings of the second interna-tional joint conference on Autonomous agents 2003.

[Ama98] Amandi A, Price A: "**Building Object-Agents from a Software Meta-Architecture. In: Advances in Artificial Intelligence"**, LNAI, vol. 1515, Springer-Verlag, 1998.

[Arl04] ARLABOSSE F, GLEIZES M P et OCCELLO M : « **Méthodes de Conception. dans les Systèmes Multi-Agents** », volume 29 de Arago, pages 137–171. Editions Tech & Doc, 2004.

[AspectJ] http://www.eclipse.org/aspectj

[Ass07] Assia AIT ALI SLIMANE, Muhammad Usman BHATTI, " **Utilisation des services et des aspects pour la réutilisabilité du logiciel d'un automate pour l'analyse de plasma**" 2007.

[Bal02] Baltus Jean : « **La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué** » ; Facultés Universitaires Notre-dame de la Paix, Namur, Belgique ; 2002

[Bal99] Ball T: "**The concept of dynamic analysis"**, in: Software Engineering ESEC/FSE99, Springer, 1999.

[Bau97] Baumer D, Riehle D, Siberski W et Wulf M : **"Role Object Pattern"**. Proceedings of PLoP '97. Technical Report WUCS-97-34. Washington University Dept. of Computer Science, 1997.

[BCEL] Apache Jakarta BCEL. http://jakarta.apache.org/bcel/

[Bel99] Bellifemine, F., Rimassa, G. & Poggi, A. JADE – "**A FIPA-Compliant Agent Framework"**. Proceedings of the Forth International Conference and Exhibition on the Practical Application of Intelligent Agents and Multi-Agents (PAAM '99), 1999.

[Bin04] Bincheng; Jiming L; Xiaolong J: "**From local behaviors to global performance in a multi-agent system".** In Intelligent Agent Technology, p. 0-1, 2004.

[Bin07] Binder W, Hulaas J, et Moret P: "**Advanced Java bytecode instrumentation".** In Proceedings of the 5th international symposium on Principles and practice of programming in Java, pages 135–144. ACM, 2007.

[Boi01] Boissier.O, «**Modèles et architectures d'agents. Principes et architecture des systèmes multi-agents** », Chapitre 2, J. P. Briot, Y. Demazeau (dir), Hermès Science Publications, 2001.

[Bon88] BOND A.H. et GASSER. L : "**Reading in distributed artificial intelligence"** Morgan Kaufmann publishers, Inc, 1988.

[Bou06] Bouzguenda. L : « **Coordination multi-agents pour le Workflow interorganisationnel lâche** ». Thèse de doctorat, IRIT, 2006.

[Bru02] Bruneton E, Lenglet R, and Coupaye T : « **ASM : a code manipulation tool to implement adaptable systems ».** Adaptable and extensible component systems, 2002.

[Bru07] Bruneton E : "**Asm 3.0 a java bytecode engineering library"**. URL: http://download. forge. objectweb. org/asm/asmguide. pdf, 2007.

[Cas12] Caserta Pierre : « **Analyse statique et dynamique de code et visualisation des logiciels via la métaphore de la ville : contribution à l'aide à la compréhension des programmes** » ; thèse de Doctorat de l'université de Lorraine ; 2012

[Cec04] Ceccato M, Tonella P: « **Measuring the Effects of Software Aspectization** ». In WCRE Workshop on Aspect Reverse Engineering (W ARE).Published at the workshop Web site: http://homepages.cwi.nIFtourwe/ware/submissions.html, 2004.

[Cec05] Ceccato M, Tonella P, Ricca F : **"Is AOP code easier or harder to test than OOP code? "**, In On-fine Proceedings of the First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005), Chicago, Illinois, 2005.

[CENQUA] **Cenqua Clover for Java**. http://www.cenqua.com/clover/.

[Cer02] Cernuzzi L, Rossi G : "**On the evaluation of agent oriented modeling methods"**. In Agent Oriented Methodology Workshop, p. 0-1, 2002

[Cha01] Chaib-Draa B, Jarras I., Moulin B., "**Systèmes multi-agents : principes généraux et applications**". Principes et architecture des systèmes multi-agents, Chapitre 1, J. P. Briot, Y. Demazeau (dir), Paris, Hermès Science Publications, 2001.

[Cha87] Chaib-Draa B, Millot P : « A**rchitecture pour les systèmes d'intelligence artificielle** », IEEE Compint'87, Montréal 1987.

[Cha96] Chaib-draa B, "**Interaction between agents in routine, familiar and unfamiliar situations**". International Journal of Intelligent and Cooperative Information Systems, Vol. 1, n° 5, p. 7-20, 1996.

[Chi98] Chilimbi T, Mark D. Hill, James R. Larus. **"Improving pointer-based Codes Through Cache-Conscious Data Placement"**. Technical Report CS-TR-98-1365, University of Wisconsin—Madison, Mar., 1998.

[Chi00] Chilimbi T, Mark D, Hill, James R. Larus.**"Making Pointer-Based Data Structures Cache Conscious "**. IEEE Computer, Vol. 33, Num. 12, pp. 67-74, Décembre 2000.

[Chi04] Chiba S : Javassist: "**Java bytecode engineering made simple"**. Java Developer's Journal, 9(1), 2004.

[Chr03] Christensen A, Moller A, Schwartzbach M : "**Precise analysis of string expressions"**, Static Analysis (2003) 1076–1076.

[Coh95] Cohen P.R. et Levesque H.J. "**Communicative actions for artificial agents"**. In Proceedings of the International Conference on multi-agents Systems, AAAI Press, Juin 1995.

[Cor09] Cornelissen B, Zaidman A, van Deursen A, Moonen L, Koschke R : "**A systematic survey of program comprehension through dynamic analysis"**, Transactions on Software Engineering 35 (5) (2009) 684–702.

[Cot99] Côté Marc : « **Une architecture multiagent et son application aux services financiers** ». Mémoire présenté à la faculté des études supérieures de l'Université Laval. Faculté des sciences et de génie. Département d'informatique. Avril 1999.

[Dah02] Dahm M, J. van Zyl, and E. Haase. "**The bytecode engineering library (BCEL)".** 2002.

[Dav02] Davidsson P, Johansson S: "**Evaluating multi-agent system architectures: A case study concerning dynamic resource allocation"**. In Third International Workshop on Engineering Societies in the Agents, p. 0-1, 2002.

[Del07] DELAHAYE Mickaël : « **Instrumentation de code Java ; Étude bibliographique** » , Master 2 Recherche Informatique ; 2007

[Dem95] Demazeau. Y: "**From interactions to collective behaviour in agent-based systems".** In First European conference on cognitive science, p. 0_1, 1995.

[Dia01] Diaz J A Pace et Campo M R : **"Analyzing the Role of Aspects in Software Design"**. Communications of the ACM, 44(10):66–73, 2001

[Din08a] Dinh Doan Van Bien, David Lillis and Rem W. Collier : "**Call Graph Profiling for Multi Agent Systems"**, School of Computer Science and Informatics University College Dublin, 2008

[Din08b] Dinh Doan Van Bien, David Lillis and Rem W. Collier : "**Space-Time Diagram Generation for Profiling Multi Agent Systems"**, School of Computer Science and Informatics University College Dublin, 2008

[Dmi02] Dmitriev M : "**Application of the HotSwap technology to advanced profiling"** . In First International Workshop on Unanticipated Software Evolution (USE2002)., 2002.

[Dmi04] Dmitriev M: "**Profiling Java applications using code hotswapping and dynamic call graph revelation".** In WOSP'04: Proceedings of the 4th international workshop on Software and performance, pages 139–150. ACM Press, 2004.

[Don97b] Don Roberts, John Brant et Ralph Johnson: **"A refactoring tool for smalltalk"**. Theory and Practice of Object Systems (Issue 4) 1997.

[Dou03] Douglas J Brear, Thibaut Weise, Tim Wiffen, Kwok Cheung Yeung, Sarah A M Bennett et Paul H J Kelly : **Search strategies for Java bottleneck location by dynamic instrumentation**, 2003.

[Duf04] Dufour B, C. Goard, L. Hendren, C. Verbrugge, O. de Moor, and G. Sittampalam. "**Measuring the Dynamic Behaviour of AspectJ Programs"**. In Proc. OOPSLA 2004, 2004. to appear.

[Erc91] Erceau J., Ferber J., «**L'intelligence artificielle distribuée** », La Recherche, Volume 22(233), 1991.

[Ern03] Ernst, M. D. 2003. "**Static and dynamic analysis: Synergy and duality"**. Proceedings of the 1st ICSE Workshop on Dynamic Analysis (WODA). 2003

[Fer88] Ferber J, Ghallab M : **«Problématiques des univers multi-agent intelligents** », PRC-GRECO I.A, 1988.

[Fer89] Ferber J : « **Objets et agents : une étude des structures de représentation et de communication en intelligence artificielle »,** Thèse de Doctorat d'Etat, Université de Pierre et Marie Curie, Paris VI, 1989.

[Fer95] Ferber J : «**Les systèmes multi-agents : vers une intelligence collective** », InterEditions, 1995.

[Fer98] Ferber J, GUTKNECHT O : **"Aalaadin : a meta-model for the analysis and design of organizations in multi-agent systems"**. DEMAZEAU Y, éditeur : 3rd International Conference on Multi-Agent Systems, pages 128–135, Paris, 1998. IEEE.

[Fer07] Ferber J, Mansour S. « **Un modèle organisationnel pour les systèmes ouverts déployés à grande échelle** ». Acte des Journées Francophones sur les Systèmes Multi-Agents. Carcassonne, 2007.

[Fer09] Ferber J : « **MADKIT PAS À PAS Démarrage et prise en main du logiciel MadKit** », v1.1 ; LIRMM, Université de Montpellier II ; Mars 2009.

[Fin97] Finin T, Labrou Y, Mayfield J : «**KQML as an Agent Communication Language** », Software Agents, Bradshaw Jeffrey (Eds.), AAAI/MIT Press, 1997

[FIPA00] Dutch auction Interaction Protocol Specification. www.FIPA.org/specifications/ index.html

[FIPA01] **" FIPA ACL Message Structure Specification "**. FIPA, 2001.

[FIPA02] **"FIPA request interaction protocol specification"**, Rapport technique, Foundation for Intelligent Physical Agents, 2002. Available at www.fipa.org

[Fow99] Fowler M : "**Refactoring : Improving the Design of Existing Code"**, Addison-Wesley, 1999.

[Gar02] Garcia A, Silva V, Chavez C, Lucena C : **"Engineering Multi-Agent Systems with Aspects and Patterns"**. Journal of the Brazilian Computer Society, July 2002, v. 8, no. 1, pp. 57-72.

[Gar03] Garcia A, SantAnna F, Chavez C, Silva C, Lucena V. Tda, C J Pde, Staa Avon: **"Separation of Concerns in Multi-agent Systems: An Empirical Study"**, Proc. Int. Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003), vol 2940.

[Gar04a] GARCIA A : "**From Objects to Agents: An Aspect-Oriented Approach"**. PhD Thesis, Computer Science Department, PUC-Rio, Brazil, April 2004.

[Gar04b] GARCIA Alessandro, Uirá KULESZA, Christina CHAVEZ, Carlos LUCENA : « **The Interaction Aspect Pattern** », 2004

[Gar04c] Garcia A, Kulesza U, Lucena C : **"Aspectizing Multi-Agent Systems: From Architecture to Implementation";** "Software Engineering for Multi-Agent Systems III". Springer-Verlag, LNCS 3390, December 2004, pp. 121-143.

[Gas87] Gasser Les, Carl Braganza, and Nava Herman: "**MACE: A Flexible Testbed for Distributed AI Research**" in Michael N. Huhns, ed. Distributed Artificial Intelligence Pitman Publishers, 1987.

[Gas91] GASPAR G: **"Communication and belief changes in a society of agents: Towards a formal model of autonomous agent"**. In D.A.I. 2, p. 0_1, 1991.

[Geo88] Georgeff M.P. : «**A theory of actor for multi-agent planing**", in Readings in Distributed Artificial Intelligence, A.H. Bond and L. Gasser (Eds.), Morgan Kanfimann Publischers, 1988.

[Gho13] GHOMARI Abdessamed réda : « **approche méthodologique d'acquisition de connaissances agrégées à base d'agents cognitifs coopérants pour les systèmes d'aide à la décision stratégiques** », thèse de doctorat d'état en informatique ; 2013

[Gle04] Gleizes M P, «**Vers la résolution de Problèmes par émergence**", Habilitation à Diriger des Recherches, Université Paul Sabatier, Toulouse, 2004.

[Gri01] Grislin-Le Strugeon E, Adam E, Kolski C : « **Agents intelligents en interaction Homme-Machine dans les Systèmes d'information** », Chapitre 7, dans Environnements évolué et évaluation de l'I.H.M., Interaction Homme-Machine pour les S.I. 2, C. Kolski (Ed.),

[Gue99] Guessoum Z, Briot J: **"From Active Objects to Autonomous Agents"**. IEEE Concurrency, Special Series on Actors and Agents, Vol. 7, N. 3, 1999, pp. 68-76.

[Gut01] Gutknecht O, Ferber J, Michel F: **"Integrating tools and infrastructures for generic multi-agent systems"** In: Proceedings of the fifth international conference on Autonomous agents, AA 2001, ACM Press (2001) 441–448

[Han02] Hannemann J, Kiczales G : **"Design Pattern Implementation in Java and AspectJ"**. Proc. OOPSIA '02, pp. 161-173, Nov 2002.

[Har02] Harkema M, D. Quartel, B. M. M. Gijsen et R. D. van der Mei: **"Performance monitoring of Java applications"**. In WOSP'02: Proceedings of the 3rd international workshop on Software and performance, pages 114–127. ACM Press, 2002.

[Hey85] Hayes-Roth, B. (1985). "**A blackboard architecture for control**". Artificial Intelligence 26 (3): 251–321

[Hir04] Hirzel M, Diwan A, Hind M : **"Pointer analysis in the presence of dynamic class loading"**, in: ECOOP 2004–Object-Oriented Programming, 2004, pp. 96–122.

[Hui05] Huiqing Li et Simon Thompson: **"Formalisation of Haskell Refactorings"**. In Marko van Eekelen and Kevin Hammond, editors, Trends in Functional Programming, Sept. 2005. 20, 74, 160, 165

[Hus10] Hussein J : « **Analyse des performances d'un système multi-agents par visualisation** », Thèse de doctorat à l'Université de Grenoble, 2010.

[IDEA] http://www.jetbrains.com/idea/

[Iva02] Ivan Kiselev : **"Aspect-Oriented Programming with AspectJ"**, SAMS 2002.

[Jah97] Jahnke J H et Zündorf A : « **Rewriting poor design patterns by good design patterns** », in: S. Demeyer and H. Gall, editors, Proc. of ESEC/FSE '97 Workshop on Object-Oriented Reengineering, Technical University of Vienna, 1997.

[Jea05] Jean Philippe Retaillé : « **Refactoring des applications Java/J2EE** », Editions Eyrolles, 2005.

[Jen98] Jennings N R, Wooldridge M, Sycara K : **"A roadmap of agent research and development"**, In International journal of Autonomous Agents and Multi-Agents Systems, vol.1, n°1, p. 7-38, 1998.

[jrat] http://jrat.sourceforge.net/

[Jul11] Julien Cohen, Rémi Douence. "**Views, Program Transformations, and the Evolutivity Problem in a Functional Language"**. Research Report hal-00481941, http://hal. archives-ouvertes.fr/hal-00481941/en/, 19 pages, 2011. 20, 158

[Jur06a] Jurasovic, Jezic G, Kusek M : **"A performance analysis of multiagent systems"**. In International Transactions on Systems Science and Applications, p. 0-1, 2006a.

[Keb10] KEBIR Selim: « **Programmation orientée aspect en Java avec AspectJ** », 2010.

[Ken02] KENMEI YOUTA B R: « **Vers un profiling adaptatif** » : Etat de l'art des méthodes actuelles d'analyse de programmes et perspectives, UNIVERSITY OF YAOUNDE I, 2001/2002.

[Ken99] Kendall E : **"Role Model Designs and Implementations with Aspect-oriented Programming"**. OOPSLA 1999, pp. 353-369.

[Ker04] Kerievsky J : **"Refactoring to patterns",** Addison-Wesley, 2004.

[Kin96] Kinny D, Georgeff M, Rao A : **"A Methodology and Modelling Technique for Systems of BDI Agents**". In the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Eindhoven, The Netherlands, 1996.

[KQML97] **"KQML as an agent communication language"** Tim Finin, Yannis Labrou, and James Mayfield, in Jeff Bradshaw (Ed.), "Software Agents", MIT Press, Cambridge, to appear, (1997).

[Kul07] Kuleshov E : "**Using ASM framework to implement common bytecode transformation**

patterns". Proc. of the 6th AOSD, ACM Press, 2007.

[Lab98] Labrou Y et Finin T: **"Semantics for an Agent Communication Language"**. Agent Theories, Architectures and Languages IV. M. Wooldridge, J. P Muller and M. Tambe, Springer-Verlag.1998

[Lab99] Labrou Y, Finin T, Peng Y : «**Agent Communication Languages : the Current Landscape** », IEEE Intelligent Systems & Their Applications, Vol.14, N°2, 1999, pp. 45-52.

[Lak98] Lakhotia A et Deprez J C : « **Restructuring programs by tucking statements into functions** », in : M. Harman and K. Gallagher, editors, Special Issue on Program Slicing, Information and Software Technology 40, Elsevier, 1998.

[Lop98] LOPES C V, KICZALES G: **"Recent Developments in AspectJ"**; ECOOP'98 Workshop Reader, Chapitre, No 1543 LNCS, Springer-Verlag, 1998.

[Lou10] Loukil Sihem: « **Extension d'un langage de description d'architecture pour la programmation orientée aspect** », master de L'école Nationale d'Ingénieurs de Sfax, 2010, page 9.

[Mad07] Madeyski L; Szala L : **"Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study"**, IET Software Journal, vol. 1, no. 5, pp. 180–187, 2007. http://dx.doi.org/10.1049/iet-sen20060071

[Mar06] MAROT Antoine : « **L'animation d'algorithmes en programmation orientée aspect** », Université libre de Bruxelles ; 2006.

[Mey92] Meyer B : « **Applying "Design by Contract"** », Computer, vol. 25, n° 10, p. 40-51, 1992.

[Mic05] Michel F, G Daniel, Ferber J et Phan D: « **Projet integration Moduleco/Madkit : premiers resultats** ». Joint Conference on Multi-Agent Modelling for Environmental Management, SMAGET 05, Bourg-Saint-Maurice, Les Arcs, FRANCE, mars, 2005.

[Min96] Minar N, Burkhart R, Langton C, Askenazi M : **"The Swarm Simulation System: A Toolkit for Building Multi-Agent Simulation"**. SFI Working Paper 96-06- 042; 1996.

[Mor94] Moraïtis P : "**Paradigme multi-agent et prise de décision distribuée**", Thèse de Doctorat, Université de Paris IX, 1994.

[Mül96b] Müller J P: **"The Design of Intelligent Agents – A layered Approach"**, Lecture Notes in artificial Intelligence, 1996.

[Muy96] Mayfield J, Labrou Y, Finin T : « **Evaluating KQML as an Agent Communication Language** », In : Wooldridge Michael, Müller Jörg P, Tambe Milind (Eds.), Intelligent Agents II : Agent Theories, Architectures, Languages – IJCAI'95 Workshop (ATAL), 1996.

[Myl02a] Mylopoulos J; Kolp M; Giorgini P: **"Agent-oriented software development"**. In SETN, p. 0-1, 2002a.

[Nie04] NIELSON F, NIELSON H R et HANKIN C : **"Principles of program analysis"**. Springer, 2004

[Nii86] Nii H P: "**Blackboard systems: the blackboard model of problem-solving and the evolution of blackboard architectures**", AI Magazine, Vol. 7(3), 1986.

[Nwa99] Nwama H et al : **"Zeus: A toolkit for building distributed multi-agent systems"**. Applied Artificial Intelligence Journal, vol. 13(1), pp: 129-186, 1999.

[O'Bri98] O'Brien P D, Nicol R C, « **FIPA – towards a standard for software agents** », BT Technology Journal, Vol. 16 (3), pp 51-59, July 1998.

[O'ha04] O'Hair K: **"The JVMPI transition to JVMTI"**. Sun Developper Network, 2004.

[Ode00] Odell J, Paranuk H v D et Bauer B: **"Representing Agent Interaction Protocols in UML, submitted for Autonomous Agents"** 2000.

[Opd92] Opdyke W F, Refactoring : **"A Program Restructuring Aid in Designing Object-Oriented Application Frameworks"**, PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Pac03] Pace A, Trilnik F, Campo M : **"Assisting the Development of Aspect-based MAS using the SmartWeaver Approach"**. In: Garcia A et al "Software Engineering for Large-Scale Multi-Agent Systems". Springer-Verlag, LNCS 2603, April 2003.

[Pic04] Picard G : « **Méthodologie de développement de systèmes multi-agents adaptatifs et conception de logiciels à fonctionnalité émergente** », Thèse de Doctorat, Université Toulouse

III, 2004.

[Rac05] Samah Rached : « **Analyse du comportement des programmes à l'aide des matrices d'adjacence** », Université de Montréal, page 45 ;2005

[Raj02] Rajiv Gupta, Eduard Mehofer, Youtao Zhang : **"Profile Guided Compiler Optimizations"**, to appear, The Compiler Design Handbook : Optimizations and Machine Code Generation, 2002

[Ram03] RAMNIVAS Laddad: **"AspectJ in Action: Practical Aspect-Oriented Programming"** by Manning Publications, 2003.

[Ree94] Reed A : **"Experimental performance analysis of parallel systems: Techniques and open problems"**. In Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, p. 25_51, 1994.

[Rem08] Rem W Collier et al. **Web site for Agent Factory**, 2008. http://www.agentfactory.com/ (accessed October, 2008).

[Ren04] Renaud P, Jean-philippe R, Lionel S : « **Programmation orientée aspect pour Java/J2EE** », 2004.

[Rob05] Robert E, Filman Tzilla, Elrad Siobhan Clarke et Mehmet Aksit : **"Aspect-Oriented Software Development"** Addison-Wesley, Boston, 2005.

[Sch04] Schwarz D : **"Peeking inside the box : Attribute-oriented programming with Java 1.5"**, ONJava.com, 2004.

[Sea90] Searl J R ; **"Intention in communication"** ; chapter 19 ; collection intentions and actions, pp 401-415 ; MIT press London 1990.

[Seg01] Seghrouchni A El Fallah : « **Principes et architecture des systèmes multi-agents** », chapitre Les modèles de coordination d'agents cognitifs. Hermès, 2001.

[Sne98] Snelting G et F Tip : « **Reengineering class hierarchies using concept analysis** », in: Proc. Foundations of Software Engineering (FSE-6), SIGSOFT Software Engineering Notes 1998.

[Sre00] Sreedhar V, Burke M, Choi J: **"A framework for inter procedural optimization in the presence of dynamic class loading"**, in: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, ACM, 2000, pp. 196–207.

[Ter04] TERENCE Ferut, Leroy Sébastien **: "La programmation Orientée Aspects"**, Juin 2004.

[Ter98] Terna P : "**Simulation Tools for Social Scientists: Building Agent Based Models with SWARM**" Journal of Artificial Societies and Social Simulation, Vol. 1, 9 Pp.1998

[Tou11] Toupin D : **"Using tracing to diagnose or monitor systems. Software, IEEE"**, p : 87– 91, 2011.

[Uba01] Ubayashi N, Tamai T : **"Separation of Concerns in Mobile Agent Applications"**. Proc. of the 3rd Conference Reflection 2001, LNCS 2192, Kyoto, pp. 89-109, September 2001

[Van97] Van De Vijver G : « **Emergence et explication", Intellectica: Emergence and explanation** », 1997/2 n°25, ISSN N°0984-0028, pp. 185-194, 1997.

[VisualVM] http://visualvm.java.net/

[Whi02] White A, SERP : **"an Open Source framework for manipulating Java bytecode"**. http://serp.sourceforge.net/, 2002.

[Woo99] Wooldridge M, Jennings N. R et Kinny D : «**A methodology for agent-oriented analysis and design**", Proceedings of the Third International Conference on Autonomous Agents, pp. 69-76, Seattle, USA, 1999.

[Woo00a] Wooldridge M et Jeennings N R : «**Agent-Oriented Software Engineering**» in Handbook of Technology (ed. J. Bradshaw) AAAI/MIT Press. 2000

[Woo00b] Wooldridge M, Jennings N R et Kinny D: **"The gaia methodology for agent-oriented analysis and design"**. Journal of Autonomous Agents and Multi Agent Systems, p. 0_1, 2000.

[XER00] XEROX, http://www.aspectj.org, 2000

[Zha03] Zhang C, Jacobsen H-A : **"Quantifying Aspects in Middleware Platforms"**. In Proc. AOSD 2003, pages 130–139. ACM Press, 2003