République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université L'Arbi Ben M'Hidi-Oum El Bouaghi

Faculté des Sciences Exactes et Sciences de la Nature et de la Vie

Département de Mathématiques et d'Informatique

N° D'ordre :……

Série :……..

MÉMOIRE

Pour l'obtention du diplôme de Magister en Informatique

OPTION

Intelligence Artificielle et Imagerie

Présenté par

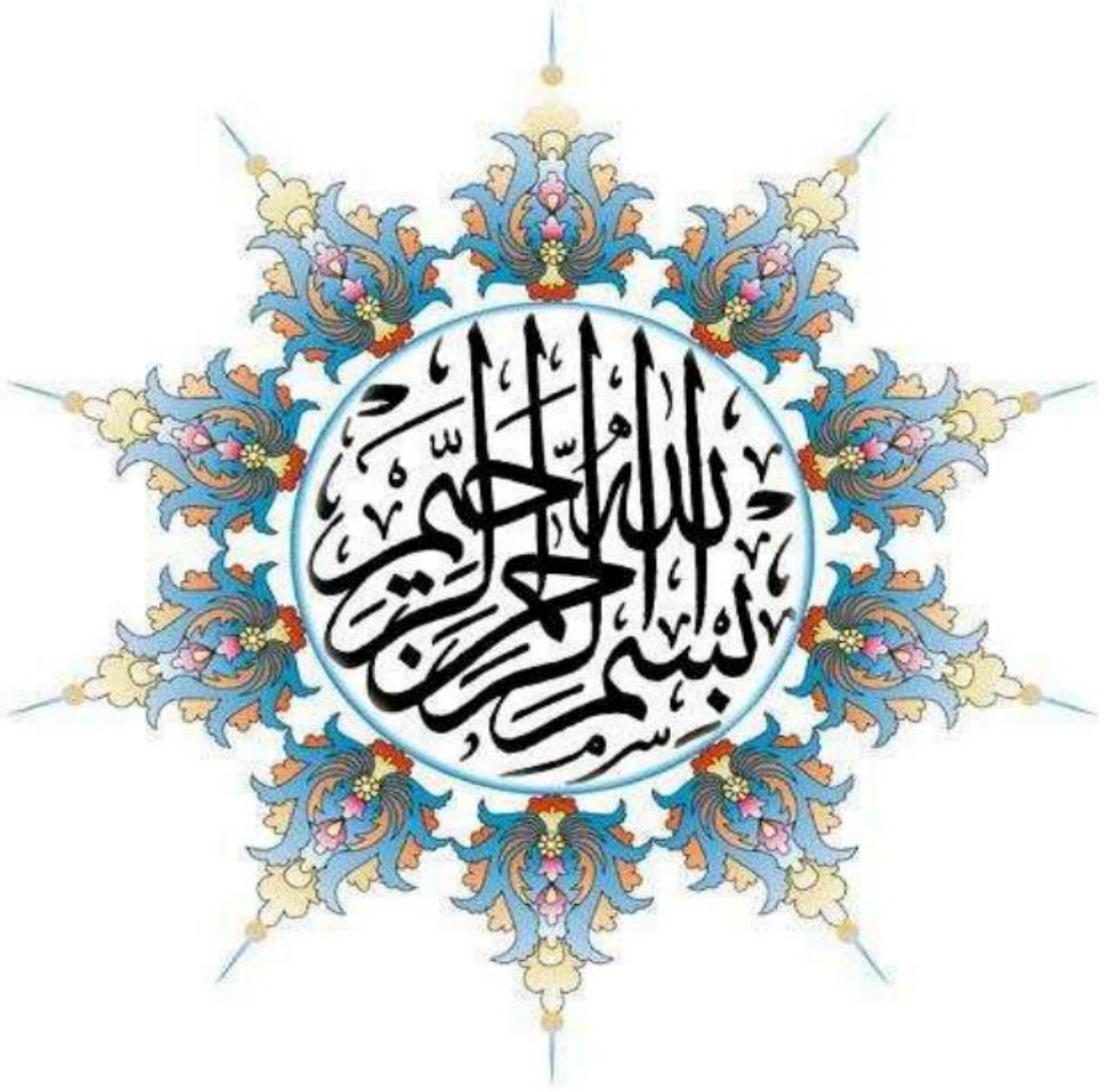
Zerrougui Salim

Une approche pour l'extraction d'Aspects dans les Applications Multi-Agents

DEVANT LE JURY COMPOSÉ DE

Mr. Nini Brahim	Maitre de conférences	Univ. d'O.E.B.	**Président**
Mr. Mokhati Farid	Maitre de conférences	Univ. d'O.E.B.	**Rapporteur**
Mr. Amirat Abdelkarim	Maitre de conférences	Univ. de Souk Ahras.	**Examinateur**
Mr. Kazar Okba	Professeur	Univ. de Biskra	**Examinateur**
Mr. Boutekkouk Fateh	Maitre de conférences	Univ. d' O.E.B.	**Examinateur**

2013/2014


بسم الله الرحمن الرحیم

Résumé. *L'extraction d'aspect « Aspect Mining » est une étape préalable et importante pour la restructuration d'aspect « Aspect Refactoring »* et joue un rôle crucial dans la compréhension et la maintenance de programme. *Plusieurs techniques d'extraction d'aspect ont été proposées pour les programmes orientés objet. Cependant, l'utilisation de l'extraction d'aspect dans les systèmes multi agents (programmes multi agents) est un domaine de recherche inexploré. Les spécificités inhérentes aux systèmes multi agents (autonomie, pro activité, adaptabilité, etc.) rendent leur réutilisabilité et maintenabilité des tâches difficiles à réaliser. Nous proposons, dans ce mémoire, une approche hybride (basée sur l'analyse statique et l'analyse dynamique) et semi-automatique d'extraction d'aspects pour les systèmes orientés agent. Les principales motivations de l'approche proposée sont: (1) l'identification des préoccupations transverses dans les codes orientés agent existants, et (2) l'explicitation de ces préoccupations aux ingénieurs logiciels impliqués dans l'évolution de code orienté agent afin de faciliter son restructuration et, par conséquent, d'améliorer ça réutilisabilité et sa maintenabilité. L'approche proposée est supportée par un outil logiciel qui nous avons développé nommé MAMIT (Mas Aspect-MIning Tool). L'approche et l'outil associés sont illustrés à l'aide d'un cas d'étude concrète.*

Mot clés : Systèmes multi-agents, Extraction d'aspect, les préoccupations transversales, la programmation orientée aspect, Analyse statique, Analyse dynamique.

Abstract. *Aspect mining is an important activity that helps in aspect refactoring and plays a crucial role in program comprehension and maintenance. Several aspect mining techniques are being proposed dealing with object-oriented programs. However, using aspect mining for agent-based programs is an unexplored research domain. The specificities inherent in multi-agent systems (autonomy, proactivity, adaptability, etc) make their reusability and maintainability difficult tasks to achieve .We propose, in this thesis, a hybrid semi-automatic approach that is based on static and dynamic analysis for aspect mining from agent oriented code. The main motivations of the proposed approach are, on one hand, to identifying cross-cutting concerns in existing agent-oriented code and, on the other hand, to make them explicitly available to software engineers involved in the evolution of that code. The proposed approach is supported by a software tool we developed called MAMIT (Mas Aspect-MIning Tool) and, it is illustrated by a concrete case study.*

Key words: Aspect Mining, Multi-Agent Systems, Aspect-Oriented Programming, Static Analysis, Dynamic Analysis, cross-cutting concerns.

ملخص. استخراج الجوانب هو نشاط هام يساعد في إعادة هيكلية الجوانب ، ويلعب دوراً حاسما في فهم وصيانة البرامج ، اقترحت العديد من تقنيات استخراج الجوانب و التي تتعامل مع البرامج الكائنية التوجه . ومع ذلك، فلستخدام استخراج الجوانب للبرامج القائمة على الوكلاء هو مجال بحث غير مستكشف. الخصوصيات الكامنة في الأنظمة المتعددة الوكلاء (الحكم الذاتي، المبادرة والقدرة على التكيف ،إلخ) جعل إعادة استخدامها وصيانتها مهام يصعب تحقيقها ، نقترح في هذه الرسالة، نهجا هجين و شبه تلقائي يستند على التحليل الثابت و الديناميكي لاستخراج الجوانب من الأنظمة المتعدد الوكلاء. الدوافع الرئيسية للنهج المقترح هي ، من جهة، لتحديد الشواغل القاطعة في البرامج المتعدد الوكلاء الموجودة و من ناحية أخرى، لجعلها متاحة بشكل واضح لمهندسي البرمجيات المعنيين بتطوير هته البرامج. المنهج المقترح مدعم بأداة مبرمجة تسمى MAMIT (Mas Aspect-MIning Tool) الموضحة من خلال دراسة لحالة ملموسة .

الكلمات المفاتيح: استخراج الجوانب، الأنظمة المتعددة الوكلاء، البرمجة الجانبية التوجه ، التحليل الثابت ، التحليل الديناميكي ، الشواغل القاطعة.

Dédicaces

Je dédie ce modeste travail comme un témoignage d'affectation, de respect et d'admiration :

A mes parents, mes sœurs qui m'ont aidé moralement et matériellement.

A mes amis pour leurs fidélités.

A mes enseignants, pour leurs efforts afin de nous assurer une formation solide.

A toute personne qui de près ou de loin m'a fait part de renseignements nécessaires à mon projet.

Salim.

Remerciements

C'est avec un grand plaisir que je réserve ces lignes en signe de gratitude et de reconnaissance.

Mon premier remerciement va au DIEU Tout-Puissant.

Je profite de cette occasion pour adresser particulièrement
Mes vifs remerciements à mon encadreur Docteur MOKHATI Farid, pour sa disponibilité, sa patience, ses encouragements, ainsi que la pertinence des ses conseils et toute l'attention dont il nous a fait tout au long de ce projet, à mon Co-encadreur Professeur BADRI Mourad, pour toutes ces orientations, conseils, efforts, encouragements

Je tenue à remercier tous les membres du Jury, pour le grand honneur qu'ils m'ont fait en acceptant d'examiner ce travail et d'y apporter leurs critiques constructives et leurs valeureuses remarques.

Enfin je ne saurais terminer sans exprimer mes reconnaissons à tous mes enseignants qui ont contribué à ma bonne formation.

Salim.

Table des matières

Introduction Générale

Chapitre 01: Les Systèmes Multi-Agents et la plateforme JADE

Chapitre 02: La programmation orientée aspect et le tisseur ASPECTJ

Chapitre 03: Les techniques d'extraction d'Aspects

Liste des figures

Chapitre 01: Les Systèmes Multi-Agents et la plateforme JADE

Chapitre 02: La programmation orientée aspect et le tisseur ASPECTJ

Chapitre 03: Les techniques d'extraction d'Aspects

Chapitre 04: L'approche proposée

Chapitre 05: Validation de l'approche

Liste des tables

Chapitre 01: Les Systèmes Multi-Agents et la plateforme JADE

Chapitre 02: La programmation orientée aspect et le tisseur ASPECTJ

Chapitre 03: Les techniques d'extraction d'Aspects

Chapitre 04: L'approche proposée

Chapitre 05: Validation de l'approche

Introduction Générale

Introduction Générale

Le paradigme agent a été utilisé, pendant de nombreuses années, pour le développement de systèmes complexes et distribués pour de nombreuses raisons bien connues telles que l'autonomie, la réactivité, la robustesse et la pro-activité [Seg04, Mok08]. Afin de faciliter le processus d'élaboration de tels systèmes, les développeurs utilisent souvent des plateformes multi-agents pour mettre en œuvre ces systèmes complexes. Bien que ces plates-formes soient importantes, elles omettent la qualité du code source. Par conséquent, les processus de compréhension et de maintenance de ces systèmes deviennent des tâches difficiles à accomplir. Parmi les principales causes de cette difficulté, on peut citer la dispersion et l'enchevêtrement du code où les préoccupations fonctionnelles et non fonctionnelles (préoccupations transversales) sont fusionnées.

L'approche la plus connue pour implémenter les préoccupations non fonctionnelles séparément de celles fonctionnelles est la programmation orientée aspect. Elle promet une meilleure modularité, une meilleure réutilisation du code et une meilleure adaptation du code aux modifications [Kic97].Ces dernières années, seulement quelques approches [Sil06, Gar05, Gar04b, Sil09, Kul04] ont été proposées dans la littérature afin de régler le problème de séparation des préoccupations transversales dans les systèmes multi-agents aux étapes préliminaires de développement (i.e. étapes de spécification et de conception). Ces travaux ont considérablement enrichi le domaine en proposant de nouvelles stratégies et outils pour améliorer le développement de systèmes multi-agents. Toutefois, ils ne traitent pas la séparation des préoccupations non fonctionnelles de celles fonctionnelles pour les applications multi agents existantes.

La séparation des préoccupations fonctionnelles et non fonctionnelles dans les SMA existants nécessite un passage par deux phases principales : l'extraction et la restructuration des aspects. Malgré l'évolution rapide des systèmes multi agents, l'utilisation de l'extraction des aspects pour les systèmes orientés agent est un domaine de recherche qui n'est pas encore exploré.

Dans la littérature, plusieurs travaux [Ran12, Abb12, Zha12, Par12] ont été réalisés pour l'extraction des aspects à partir des applications orientées objet. Par contre, aucun travail réalisé (à notre connaissance) pour l'extraction des aspects à partir des applications multi-agents. Dans tels types d'applications, plusieurs spécificités peuvent être considérées comme

des préoccupations non fonctionnelles, nous citons entre autres, la mobilité, l'interaction, l'apprentissage, l'autonomie et la collaboration [Gar04a]. Les modèles sociaux sont également considérés comme préoccupations transversales comme le processus de « MatchMaking », associé au « MatchMaker », un agent spécial qui fournit un service de Pages jaunes au moyen de laquelle un agent peut trouver d'autres agents de prestation de services dont il a besoin pour atteindre ses objectifs [Sil06].

Dans ce mémoire, nous proposons une approche hybride (i.e. combine l'analyse statique et celle dynamique) semi-automatique qui permet aux utilisateurs de chercher et d'identifier les préoccupations transversales dispersées dans le code d'un système orienté agent existant et qui peuvent être transformés en aspects. Bien que notre approche puisse être applicable aux différentes préoccupations transversales citées ci-dessus, nous nous sommes intéressés, dans un premier temps par l'identification des préoccupations transversales relatives au processus de Matchmaking. L'approche proposée fonctionne au niveau de granularité des fragments (Expressions) et des méthodes (Déclarations), en utilisant un catalogue des graines des préoccupations générées à partir d'une classification de l'API de la Plateforme multi agent utilisée pendant le développement du code source à analyser.

De plus, notre approche est supportée par MAMIT (Mas Aspect MIning Tool), un outil que nous avons développé comme un plug-in ECLIPSE. L'approche proposée a été validée sur un exemple concret. Il s'agissait d'analyser par MAMIT le code source d'une application Jade nommé *agentes09* [1] où la préoccupation transversale à identifier est la préoccupation liée au MatchMaker qui est appelé aussi le facilitateur d'annuaire (DF) en JADE [Sil06]. Les résultats obtenus sont très satisfaisants et montrent une très bonne précision des aspects candidats avec une intervention minime de l'utilisateur.

Le reste de ce mémoire est organisé en cinq chapitres :

- **Chapitre 1 : Les systèmes multi agents et la Plateforme JADE**

  Ce chapitre présente les notions des systèmes multi agent et le framework JADE.

- **Chapitre 2 : La programmation orientée aspect et le tisseur ASPECTJ**

  Ce chapitre présente le paradigme de la Programmation Orientée Aspect, ses concepts de base ainsi que le tisseur ASPECTJ.

- **Chapitre 3 : Les techniques d'extraction des aspects**

  Ce chapitre présente la notion de l'extraction des aspects et les travaux réalisés dans les systèmes orienté objets et orienté agent.

- **Chapitre 4 : L'approche proposée**

  Ce chapitre présente l'approche proposée pour l'extraction d'aspects à partir des applications multi agents.

- **Chapitre 5 : Validation de l'approche**

  Ce chapitre présent la validation de l'approche que nous avons proposée à travers la présentation de l'outil la supportant sur un exemple concret.

Chapitre 01

Les Systèmes Multi-Agents et la plateforme JADE

1. Introduction

Un système multi-agents (SMA) est un ensemble d'agents situés dans un environnement qui communiquent entre eux afin d'atteindre un but. Un agent est une entité programmée et donc définie par des caractéristiques afin de pouvoir évoluer de façon quasi autonome. Cette entité peut-être matérialisée par un programme, un robot, un processus, etc.

Les systèmes multi-agents sont le plus souvent implémentés à l'aide de plate-formes multi-agents comme celle que nous allons voir: la plate-forme JADE (Java Agent DEvelopment framework). Cette plate-forme doit être compatible avec la norme FIPA (1997) qui rassemble un ensemble de règles permettant à une société d'agents d'interagir entre eux.
Alors, le meilleur moyen pour construire un système multi-agent(SMA) est d'utiliser une plate-forme multi-agent. 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.

2. La notion d'agent

Différent domaines ont utilisé le concept d'agent comme l'objet d'étude pour une très long période (ex: l'informatique, robotique, psychologie, biologie, sociologie...etc.). Quand même il n'y a pas une définition adoptée par la communauté sur le terme "agent". Plusieurs définitions ont été proposé dans la littérature, nous citons entre autre, la définition de Ferber et celle de Wooldridge.

2.1. Définition de Ferber

D'après Jacques Ferber [Fer95], on appelle agent une entité physique ou virtuelle :

- capable d'agir dans un environnement,
- peut communiquer directement avec d'autres agents,
- mue par un ensemble de tendances (sous la forme d'objectifs individuels ou d'une fonction de satisfaction, voire de survie, qu'elle cherche à optimiser),
- possède des ressources propres,
- capable de percevoir (mais de manière limitée) son environnement,

- ne dispose qu'une représentation partielle de cet environnement (et éventuellement aucune),
- possède des compétences et offre des services,
- qui peut éventuellement se reproduire, mourir et changer d'état,
- dont le comportement tend à satisfaire ses objectifs, en tenant compte des ressources et des compétences dont il dispose, et en fonction de sa perception, et de ces représentations des communications qu'il reçoit.

2.2. Définition de Wooldrige

« *Un agent est un programme informatique qui est situé dans un environnement et qui est doté de comportements autonomes (actions) lui permettant d'atteindre, dans cet environnement, les objectifs qui lui ont été fixé à sa conception* » [Woo97].

En partant des définitions précédentes et de l'ouvrage de Wooldrige et Jennings, [Jen94], nous pouvons identifier les caractéristiques suivantes pour un agent

- *situé* – l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement.
- *autonome* – l'agent est capable d'agir sans l'intervention d'une autre entité (humain ou agent) et contrôle ses propres actions ainsi que son état interne;
- *proactif* – l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative au bon moment;
- *capable de répondre à temps* – l'agent doit être capable de percevoir son environnement et d'élaborer une réponse dans le temps requis;
- *social* – l'agent doit être capable d'interagir avec des autres agents (logiciels ou humains) afin d'accomplir des tâches ou aider ces agents à accomplir les leurs.

2.3. Les Préoccupations d'Agents

Un agent intelligent (Figure 1) peut évoluer dans un environnement. Il doit être capable de recevoir des informations par des récepteurs et d'agir sur cet environnement par des effecteurs suivant un comportement établi à partir des observations et du raisonnement de l'agent. Il peut se déplacer d'une Plateforme à une autre, il peut s'adapter et apprendre des nouvelles connaissances. L'interaction est essentielle lorsque l'environnement est constitué d'autres agents où ils peuvent se coopérer pour atteindre leurs objectifs, donc plusieurs préoccupations

peuvent être appliqués par l'agent au même temps, nous distinguons deux types les préoccupations *Agenthood* et les préoccupations additionnelles (Figure 2).

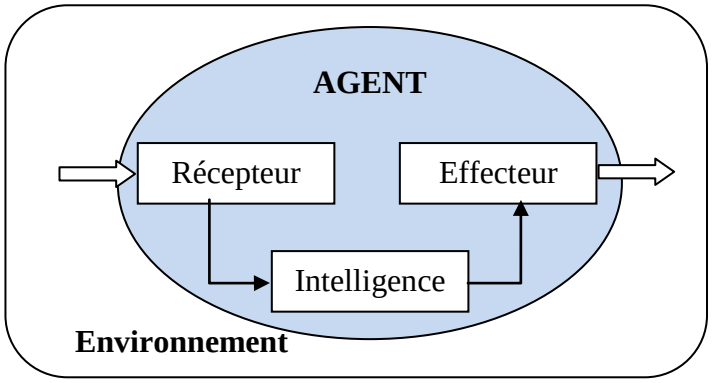


Figure 1. Structure simplifiée d'un agent intelligent [GB2002].

2.3.1. Les Préoccupations AgentHood

Ce sont les fonctionnalités incorporées par toutes les architectures d'agents indépendant de leurs types. Ils consistent généralement en un ensemble de préoccupation de base des agents « les services et les connaissances d'agents » et quelques préoccupations comportementale. Il n'existe pas de définition communément accepté de « AgentHood », l'autonomie, l'interaction et l'adaptation sont considérés comme des propriétés « agent Hood » d'agents, alors que la collaboration, les rôles, l'apprentissage et la mobilité ne sont ni nécessaire ni des conditions suffisantes par « agent Hood » [Gar04a].

2.3.1.1. L'interaction

La préoccupation de l'interaction est la propriété de l'agent qui implémente la communication avec l'environnement extérieur. Le comportement de l'interaction se compose essentiellement de la réception et de l'envoi des messages à d'autres agents par des capteurs et des effecteurs. Lorsqu'un message est reçu, il est stocké dans une boîte de réception d'agent. Lorsqu'un agent effectue des actions et des plans, il a besoin d'envoyer des messages aux autres agents. Un message est envoyé à partir d'une action simple ou d'un plan. Les messages envoyés sont organisés et stockés dans une boîte d'envoi. Les messages sont structurés selon un langage de communication d'agent, par exemple ACL (Agent Communication Language). Depuis que différents agents peuvent utiliser différentes ACLs, les messages sont translatés vers un style de message interne utilisé par l'agent. Le protocole d'interaction peut également

implémenter un comportement sensoriel, qui consiste à observer les événements dans l'environnement.

2.3.1.2. L'adaptation

La préoccupation d'adaptation est la propriété de l'agent qui le modifie selon les événements internes et externes. Il existe deux types d'adaptation : adaptation des connaissances et l'adaptation du comportement. Ils suivent le même protocole de base, qui consiste d'observer les événements environnementaux ou internes, et de rassembler les informations nécessaires, en sélectionnant et en invoquant les adaptateurs associés.

L'adaptation des connaissances se traduit par la modification de quelque morceau de la connaissance de l'agent. L'adaptation du comportement se traduit par l'annulation d'un plan ou la sélection de nouveaux plans qui doivent être exécutées ultérieurement.

2.3.1.3. L'autonomie

Généralement, elle signifie qu'un agent a le contrôle de ses propres actions et peut agir indépendamment des autres. Afin d'être autonome, l'agent doit. (i) créer ses propres objectifs sur la base des événements internes et externes, (ii) prendre des décisions sur l'instanciation de but, (iii) possède ses propres threads de contrôle (autonomie d'exécution) et (iv) créer des objectifs sans intervention extérieure directe (pro-activité). La réalisation des objectifs proactifs dépend de leur degré d'autonomie. Le degré d'autonomie augmente ou diminue selon les réussites et les échecs des actions de l'agent dans le passé. Ce sont les dimensions d'autonomie d'agent qui trouve couramment sur la littérature.

2.3.2. Les préoccupations additionnelles

En plus des préoccupations *AgentHood*, le développeur d'agent peut avoir à faire face à d'autre préoccupation, l'agent peut se déplacer d'un environnement à un autre, acquérir de nouvelles connaissances pour améliorer sa performance et de collaborer avec d'autres agents comme le déplacement des agents d'une plateforme à une autre, sans oublier l'apprentissage, et la collaboration des agents [Gar04a].

2.3.2.1. La mobilité

La préoccupation de mobilité englobe le comportement qui supporte les déplacements vers des environnements éloignés. Pendant l'exécution de ses plans, un agent mobile peut se déplacer d'un environnement à un autre afin d'atteindre ses objectifs. Multiples facettes de stratégie de mobilité nécessite à être pris en considération, y compris la spécification des

éléments mobiles, les descriptions de quand l'agent doit se déplacer, le départ vers des environnements distants, le retour à l'environnement d'origine et le contrôle de son itinéraire.

2.3.2.2. L'apprentissage

La propriété d'apprentissage implique le comportement de l'agent responsable de raffinage ou d'acquérir des connaissances. Les agents cognitifs apprennent en se basant sur l'expérience comme résultat de leurs propres actions, leurs erreurs, les interactions successives avec l'environnement extérieur et les collaborations avec d'autres agents. Les Agents peuvent utiliser différentes techniques d'apprentissage, mais le protocole d'apprentissage général est le suivant: (i) un événement est détecté comme pertinent, (ii) l'événement est pris et les informations sont recueillies à des fins d'apprentissage, (iii) l'algorithme d'apprentissage traite les informations recueillies, (iv) l'information est stockée et mène alternativement à de nouvelles conclusions, (v) si une nouvelle conclusion est atteinte, l'agent de la connaissance s'adapte.

2.3.2.3. La collaboration

La collaboration est considérée comme une forme plus sophistiquée de l'interaction, car elle implique des protocoles de collaboration et des rôles. Les protocoles de collaboration définissent les moyens pour que les agents logiciels puissent interagir avec d'autres agents dans une organisation multi-agent. Les agents jouent des rôles différents dans la poursuite de leurs objectifs individuels et de travailler ensemble avec d'autres agents dans des contextes multiples. La structure de rôle est similaire à la structure de l'agent. Comme un agent, un rôle a des croyances, des objectifs, des actions et des plans pour la réalisation des collaborations avec d'autres agents. Il peut avoir des comportements spécifiques pour interagir avec d'autres agents, des algorithmes de décision spécifiques, et des stratégies d'adaptation spécifiques. Elle peut aussi avoir des comportements spécialisés liés aux propriétés supplémentaires. Toutefois, un rôle ne peut exister sans un agent.

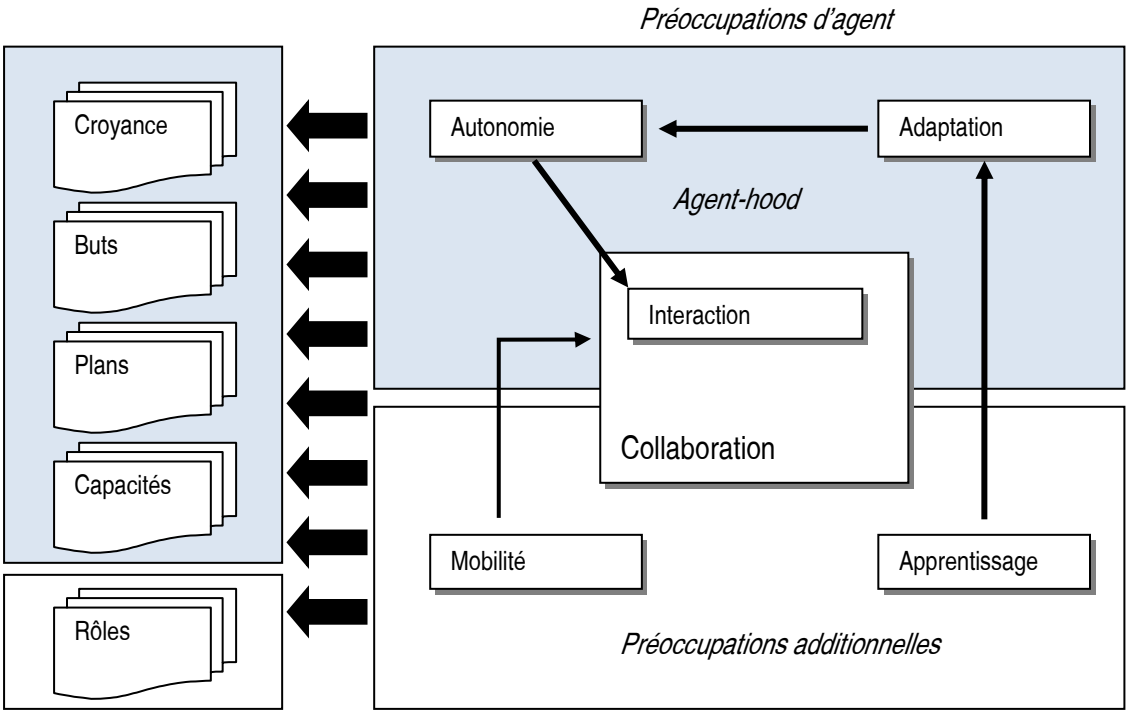


Figure 2. Les préoccupations d'agent [Gar02].

2.4. Modèles d'agents

Les modèles d'agents sont les suivants:

2.4.1. L'agent cognitif

Est un agent qui possède une représentation explicite de son environnement. Pour atteindre son objectif, il raisonne sur l'état de l'environnement et choisi les bonnes actions à exécuter. Généralement, un système cognitif comprend un petit nombre d'agents, chacun est assimilable à un système expert plus au moins complexe. Dans ce cas on parle d'agent de forte granularité (exécution de traitement et de code complexe) (Figure 3).

2.4.1.1. Le modèle BDI

Pour réaliser des agents rationnels, de nombreuses architectures agent délibératif existent (BDI introduit par Bratman 1987, AOP introduit par Shoham 1993, 3APL introduit par Hindriksetal 1999 et SOAR introduit par Lehman et al 1996). Mais les plus intéressants est le modèle BDI qui est considéré comme un modèle philosophique pour décrire des agents rationnels. Il se compose des concepts de la croyance, le désir et l'intention.

Les croyances capturent des attitudes informationnelles, les désirs capturent des attitudes de motivation, et les intentions capturent des attitudes des agents délibératifs.

Rao et Georgeff en 1995 ont adopté ce modèle et l'ont transformé en une théorie formelle et un modèle d'exécution pour les agents logiciels, basée sur la notion de croyances, objectifs et plans [Tha08].

2.4.2. L'agent réactif

Est uniquement capable de percevoir et agir sur l'environnement. Ils sont des agents qui n'ont pas de représentation explicite de l'environnement et n'ont pas de mémoire de son historique. Ils fonctionnent suivant le mécanisme stimulus/action. Le stimulus étant un élément de l'environnement (action, message, situation, etc.). Les systèmes de ce type généralement contiennent un grand nombre d'agents, qui se caractérisent par une faible granularité (peu de code, traitements simples). Leur communication est généralement simple au moyen des traces ou signaux (modifications locales de l'environnement perceptible par les agents cibles).

2.4.3. L'agent Hybride

Ce type d'agent a des capacités réactives et d'autres cognitives. Cela permet d'obtenir simultanément les avantages des architectures cognitives et réactives, tout en éliminant leurs limitations par la combinaison du comportement proactif de l'agent, dirigé par les buts, avec un comportement réactif aux changements de l'environnement (figure 3). L'organisation de cette combinaison est faite par une architecture en couche où chacune soit une composante cognitive avec représentation symbolique des connaissances et capacités de raisonnement, soit une composante réactive .

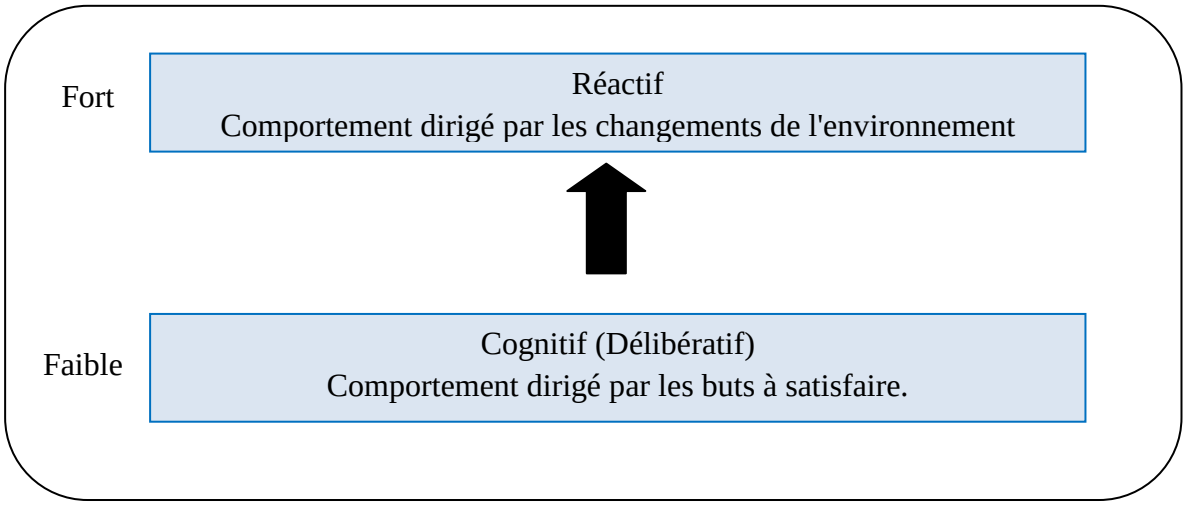


Figure 3. Couplage à l'environnement.

3. Système Multi-Agents

Les systèmes multi-agents (SMA) mettent en œuvre un ensemble de concepts et de techniques permettant à des logiciels hétérogènes, ou à des parties de logiciels, appelés "agents" de coopérer suivant des modes complexes d'interactions.

D'après ***Ferber*** [Fer95] un SMA est composé :

- D'un environnement E;
- D'un ensemble d'objets situés O (une position dans E leur est associée; ils sont passifs);
- D'un ensemble d'agents A (A inclus dans O) ;
- D'un ensemble de relations R unissant des objets ;
- D'un ensemble d'opérations O_p permettant aux agents A de percevoir et manipuler les objets de O;
- D'opérateurs chargés de représenter l'application de ces opérations et la réaction du monde à cette tentative de modification (que l'on appelle les lois de l'univers).

3.1. Caractéristiques des SMAs

Il existe des caractéristiques propres aux SMA, par rapport aux autres systèmes informatiques. Nous en fournissons une liste issue de la littérature. Un SMA possède la plupart des caractéristiques suivantes [Boi04] :

- **Distribution** : le système est modulaire, l'élément de base étant l'agent.
- **Autonomie** : un agent est en activité permanente et prend ses propres décisions en fonction de ses objectifs et de ses connaissances.
- **Décentralisation** : les agents sont indépendants, il n'y a pas de décisions centrales valables pour tout le système.
- **Échange de connaissances** : les agents sont capables de communiquer entre eux, selon des langages plus ou moins élaborés.
- **Interaction** : les agents ont une influence localement sur le comportement des autres agents, généralement sur un pied d'égalité (il n'y a pas d'ordres, seulement des requêtes).
- **Organisation** : les interactions créent des relations entre les agents, et le réseau de ces relations forme une organisation qui peut évoluer au cours du temps.
- **Situation dans un environnement** : les agents sont ancrés dans un environnement, source de données, de contraintes et d'incertitude, lieu d'actions et d'influences entre

agents. L'évolution du SMA est la combinaison des évolutions des agents et de l'environnement.

- **Ouverture** : le système échange des informations avec l'extérieur, des agents peuvent entrer et sortir du SMA ou encore être modifiés en cours d'évolution.
- **Émergence** : Dans tous les SMA, une fonction globale est attendue à partir d'un ensemble de spécifications au niveau local de chacune des entités. Cette propriété du niveau global n'est pas programmée dans les agents et n'existe que par leurs interactions conduisant à des processus permanents de réorganisation.
- **Adaptation** : il est impossible de spécifier le but global et d'organiser les agents pour l'atteindre, ou même de prouver que le SMA réalise effectivement une fonction globale adéquate. Mais le système adapte son comportement à l'environnement en cours de fonctionnement, et offre une robustesse de ce comportement, à défaut d'une optimisation.
- **Délégation** : l'utilisateur accepte de ne pas maîtriser le comportement de l'application globale, à défaut de pouvoir supporter la complexité liée à l'ensemble des décisions prises par les agents dans le système. Il délègue une partie du contrôle de l'application globale aux agents.
- **Personnalisation** : lorsqu'un agent représente un utilisateur, typiquement dans un SMA appartenant à la famille des systèmes intégrés dans un contexte plus large, il s'adapte à lui.
- **Intelligibilité** : les SMA proposent une manière naturelle de modéliser d'autres systèmes ou de mettre en œuvre des applications, ce qui les rendent simples à appréhender pour un utilisateur extérieur.

3.2. Les modèles de conception dans le paradigme orienté agent

Les modèles de conception (Design Pattern) sont des formulations explicites de bonnes expériences déjà passés dans le développement de logiciel qui consistent en des solutions bien testées des problèmes récurrents qui se posent dans certains systèmes.

Un patron de conception (design pattern) est un arrangement caractéristique de modules, reconnu comme bonne pratique en réponse à un problème de conception d'un logiciel [Kol05]. Il décrit une solution standard, utilisable dans la conception de différents logiciels.

Un patron de conception est issu de l'expérience des concepteurs de logiciels. Il décrit sous forme de diagrammes un arrangement récurrent de rôles et d'actions joués par des modules

d'un logiciel, et le nom du patron sert de vocabulaire commun entre le concepteur et le programmeur.

Les modèles fournissent une réutilisation facile d'une bonne conception de logiciels. Ce concept est également devenu indispensable au développement de logiciels multi-agents à grande échelle dans un faible coût en favorisant la réutilisation [Tah99]. Dans la dernière décennie, plusieurs modèles de conception ont été émergés, nous présentons ci-après quelques uns:

3.2.1 Les modèles sociaux (Social Pattern)

Ce sont des modèles de conception qui offrent une perspective sociale sur les systèmes multi-agents [Kol05].Les auteurs de [Kol02] ont proposé des modèles sur les aspects sociaux et intentionnels récurrentes dans les systèmes multi agent, ils sont classifiés en deux catégories [Kol05] : les patrons pairs et de médiation. Ces derniers disposent d'agents intermédiaires (Moniteur, Broker, Médiateur, Ambassade, Matchmaker et Enveloppeur) qui aident les autres agents d'arriver à un accord sur un échange de services. Les patrons pair (réservation, de souscription, d'appels d'offres et des patrons d'enchères) décrivent les interactions directes entre les agents de négociation.

3.2.2 Le modèle de réservation (Booking Pattern)

Implique un client et un certain nombre de fournisseurs de services. Le client émet une demande pour réserver une ressource d'un fournisseur de service. Le fournisseur peut accepter la demande, le nier, ou proposer de placer le client sur une liste d'attente, jusqu'à ce que la ressource demandée soit disponible lorsqu'un autre client annule sa réservation.

3.2.3 Le modèle d'abonnement (Subscription Pattern)

Implique un agent page-jaune et un certain nombre de fournisseurs de services. Les fournisseurs annoncent leurs services en abonnant aux pages jaunes. Un fournisseur qui ne souhaite plus être annoncé peut demander à être désinscrit.

3.2.4 Le modèle d'appel d'offres (Call-For-Proposals Pattern)

Implique un initiateur et un certain nombre de participants. L'initiateur lance un appel à propositions pour un service à tous les participants et accepte les propositions qui offrent le service pour un coût spécifié. L'initiateur sélectionne un participant à fournir le service.

3.2.5 Le modèle d'enchère (The Bidding pattern)

Implique un initiateur et un certain nombre de participants. L'initiateur organise et dirige le processus d'appel d'offres, et reçoit des propositions. À chaque interaction, l'initiateur publie l'offre actuelle, il peut accepter une commande, augmenter l'offre, ou annuler le processus

3.2.6 Le modèle de surveillance (The Monitor pattern)

Dans ce modèle les abonnés s'inscrivent à un agent de surveillance pour recevoir les notifications de changement d'état dans certains sujets de leur intérêt. L'agent accepte des souscriptions, demander des notifications de sujets d'intérêt, recevoir ces notifications d'événements et alertes les abonnés sur des événements pertinents. Le sujet fournit des notifications de changement d'état comme il a été demandé. L'abonné s'inscrit à la notification de changement d'état pour les sujets distribués, recevoir des notifications et informations sur l'état actuel et mettre à jour ses informations d'état local.

3.2.7 Le modèle de courtier (Broker pattern)

L'agent courtier est l'intermédiaire de l'accès aux services d'un agent (fournisseur) pour satisfaire la demande d'un consommateur. Les agents **courtier** localisent les fournisseurs correspondant à une demande de service d'un client. Il demande et obtient le service des fournisseurs de services et le transmet ensuite au client.

3.2.8 Le modèle du Médiateur (Mediator pattern)

Dans ce modèle un agent médiateur intervient dans les interactions entre les différents agents. Un initiateur adresse le médiateur au lieu de demander directement un autre collègue (performeur), Le Médiateur possède des modèles de connaissances des collègues et coordonne la coopération entre eux. À l'inverse, chaque performeur a un modèle de connaissance du médiateur. Pendant que les courtiers intermédiaires sont simplement des fournisseurs avec les consommateurs, un médiateur encapsule les interactions et tient à jour des modèles de comportement des initiateurs et des performeurs au cours du temps

3.2.9 Le modèle de l'ambassade (Embassy Pattern)

Une ambassade achemine un service demandé par un agent étranger à un agent local et s'occupe de la réponse. Si l'accès à l'agent local est accordé, l'agent étranger peut envoyer des messages à l'ambassade pour la traduction. Le contenu est traduit conformément à une ontologie standard. Les messages traduits sont transmise aux agents locaux. Les résultats de la requête sont retransmis vers l'agent étranger, traduits en sens inverse.

3.2.10 Le Modèle Enveloppeur (Wrapper Pattern)

Intègre un système existant dans un système multi-agents. Cet agent interface les clients au système existant en agissant comme un traducteur entre eux. Cela garantit que les protocoles de communication sont respectés et le système existant reste découplé du reste du système multi agent.

3.2.11 Le modèle MatchMaker

Localise un fournisseur correspondant à une demande des consommateurs pour un service, puis il lui donne la main au consommateur pour gérer directement le fournisseur choisi, contrairement au courtier qui gère directement toutes les interactions entre le consommateur et le fournisseur, la négociation pour le service et la disposition réels des services sont deux phases distinctes.

La figure 4 présente un scénario pour le modèle de Matchmaker représenté par un diagramme de séquence UML. Le client envoie une demande de service contenant les caractéristiques du service qu'il souhaite obtenir d'un fournisseur de service. Le match-maker peut aussi répondre par un refus (refuser la demande) ou une acceptation (envoyer l'emplacement du FS). Le fournisseur de service (ServiceProvider) envoie un abonnement dans les Pages Jaunes de Matchmaker. Ce dernier peut aussi répondre par un refus (refus abonnement) ou une acceptation (accepter abonnement). Le fournisseur de service envoie un désabonnement dans les pages jaunes de Matchmaker, où il peut aussi répondre par un refus (refus de résiliation), ou une acceptation (accepter désabonnement).

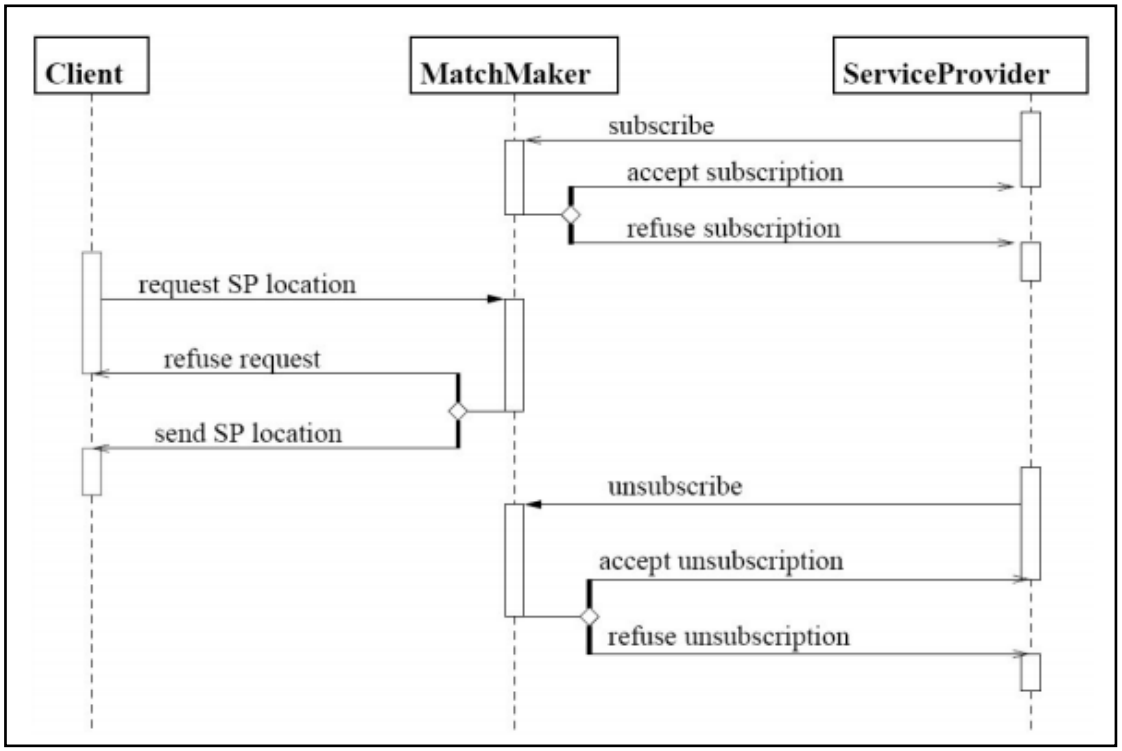


Figure 4. Diagramme de communication de modèle MatchMaker [Kol05].

3.3. Les plateformes Multi-Agents

Afin de faciliter le développement des applications multi-agents, plusieurs plateformes ont été proposées dans la littérature. Nous donnons dans ce qui suit un aperçu sur les plateformes les plus utilisées tout en mettant l'accent sur la plateforme JADE sur laquelle est basée notre travail.

3.3.1 JACK

Est un langage de programmation et un environnement de développement pour agents cognitifs, développé par la société Agent Oriented Software comme une extension orientée agent du langage Java.

3.3.2 JADE

(Java Agent DEvelopment) est un framework de développement de systèmes multi-agents, open-source et basé sur le langage Java. Il offre en particulier un support avancé de la norme FIPA-ACL, ainsi que des outils de validation syntaxique des messages entre agents basé sur les ontologies.

3.3.3 Jadex

Est une plate-forme agent développée en JAVA par l'université de Hambourg qui se veut modulaire, compatible avec de nombreux standards et capable de développer des agents suivant le modèle BDI.

3.3.4 Jason

Est un environnement *open source* de développement d'agents dans le formalisme AgentSpeak, et développé en Java par Jomi Fred Hübner et Rafael H. Bordini.

3.3.5 MadKit

Est une plate-forme multi-agents modulaire écrite en Java et construite autour du modèle organisationnel Agent/Groupe/Rôle. C'est une plate-forme libre basée sur la licence GPL/LGPL développée au sein du LIRMM.

3.3.6 Semantic Agent

Est basé sur JADE et permet le développement d'agents dont le comportement est représenté en SWRL. SemanticAgent est développé au sein du LIRIS, il est open-source et sous licence GPL V3.

4. La plateforme JADE

JADE (Java Agent DEvelopement framework) est une plate-forme multi-agent créée par le laboratoire TILAB et décrite par Bellifemine et al. Dans [Bell99, Bell00]. JADE permet le développement de systèmes multi-agents et d'applications conformes aux normes FIPA. Elle est implémentée en JAVA et fournit des classes qui implémentent les règles « JESS » pour la définition du comportement des agents. JADE possède trois modules principaux (nécessaire aux normes FIPA).

- **Le facilitateur d'Annuaire :** DF «Directory Facilitator » fournit un service de « pages jaunes» à la plate-forme;
- **Le Canal de communication :** ACC «Agent Communication Channel » gère la communication entre les agents ;
- **Le Système de gestion d'Agent :** 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.

4.1. La norme FIPA

La FIPA (Foundation for Intelligent Physical Agents) est une organisation à but non lucratif fondée en 1996 dont l'objectif est de produire des standards pour l'interopération d'agents logiciels hétérogènes. Par la combinaison d'actes de langages, de logique des prédicats et d'ontologies publiques, la FIPA cherche à offrir des moyens standardisés permettant d'interpréter les communications entre agents de manière à respecter leur sens initial, ce qui est bien plus ambitieux que XML, qui ne standardise que la structure syntaxique des documents. Afin d'atteindre ce but, le FIPA émet des standards couvrant :

- Les applications (applications nomades, agent de voyage personnel, applications de diffusion audiovisuelles, gestion de réseaux, assistant personnel…) ;
- Les architectures abstraites, définissant d'une manière générale les architectures d'agents ;
- Les langages d'interaction (ACL), les langages de contenu (comme SL, CCL, KIF ou RDF) et les protocoles d'interaction ;
- La gestion des agents (nommage, cycle de vie, description, mobilité, configuration);
- Le transport des messages : représentation (textuelle, binaire ou XML) des messages ACL, transport (par IIOP, WAP ou HTTP) de ces messages.

Ces standards évoluent, et sont régulièrement mis à jour, ainsi que de nouveaux standards qui sont nouvellement proposés. Les standards qu'édicte la FIPA ne constituent pas vraiment une plate-forme de construction multi-agents. Ce n'est pas non plus l'objectif que s'est fixé la FIPA. Tout au plus, la FIPA normalise une plate-forme d'exécution standardisée dans un but d'interopérabilité. Ces normes s'appliquent donc pour la plupart en phase de déploiement. Elles n'abordent pas les phases d'analyse ni de conception. Elles peuvent cependant guider certains choix d'implémentation.

4.2. Architecture logicielle de la plate-forme JADE

JADE reprend donc l'architecture de l'Agent *Management Reference Model* proposé par FIPA. Les différents modules présentés dans la Figure 5, sont présentés sous forme de services. Les services de base proposés sont le facilitateur d'annuaire (DF) et le système de

gestion d'agent (AMS). Il est possible de lui demander de tenir en plus le service de Message Transport Service (MTS) pour communiquer entre plusieurs plates-formes. Mais ce service sera chargé à la demande pour ne conserver par défaut que les fonctionnalités utiles à tout type d'utilisation. L'agent est l'acteur fondamental de la plate-forme, un Agent Identifier (AID) identifie un agent de manière unique. Le DF est un composant qui fait le service d'annuaire. C'est un service de « pages jaunes » qui permet de mettre en relation les agents avec leurs compétences.

Un agent peut enregistrer ses compétences dans le DF ou interroger le DF pour connaître les compétences proposées par les autres agents. L'AMS est un autre composant important car il contrôle l'accès et l'utilisation de la plate-forme et maintient un répertoire contenant les adresses de transport des agents de la plateforme. Ce service est plus un service de type « pages blanches » qui effectuent la correspondance entre l'agent et l'AID.

Chaque agent doit s'enregistrer à un AMS pour avoir un AID. Il n'y a qu'un AMS par plate-forme. Le MTS est une méthode par défaut de communication entre agents de différentes plates-formes. Cela permet l'interconnexion entre systèmes hétérogènes ou tout au moins de système ne communiquant pas de la même façon. *L'Agent Platform* (AP) constitue l'infrastructure physique sur laquelle se déploient les agents. Il contient le DF, l'AMS et le MTS. Lorsqu'on parle d'AP, on inclut souvent le matériel électronique, l'OS, le software et les composants cités ci-dessus avec les agents. Enfin, l'Agent Identifier (AID) est un identifiant précis d'un agent. On lui donne plusieurs paramètres tels que l'adresse de transport, l'adresse de service de résolution de nom, … Un exemple est : name@HAP (Home Agent Platform).

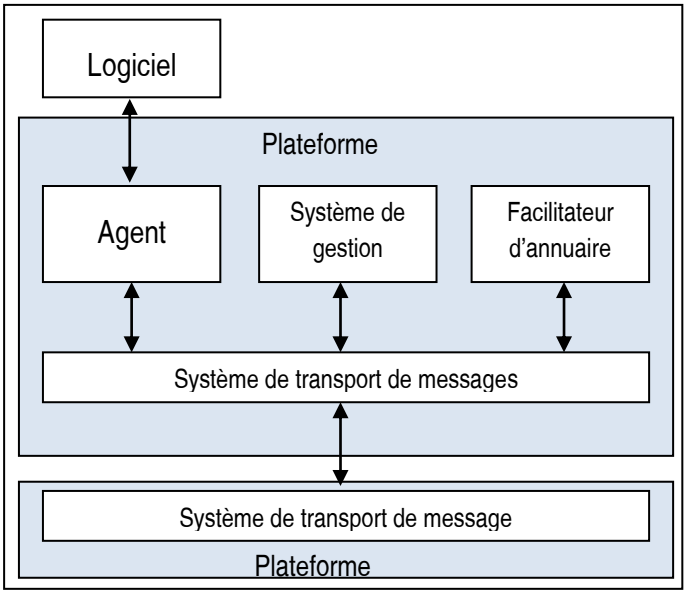


Figure 5. Architecture logiciel de La plate-forme JADE [Fer05].

Dans la plate-forme JADE, deux méthodes sont fournies par la classe *Agent* afin d'obtenir l'identifiant de l'agent DF par défaut et celui de l'agent AMS: *getDefaultDF()* et getAMS() respectivement. Ces deux agents permettent de maintenir une liste des services et des adresses de tous les autres agents de la plate-forme. Le service DF propose quatre méthodes afin de pouvoir :

- Enregistrer un agent dans les pages jaunes (register).
- Supprimer un agent des pages jaunes (deregister).
- Modifier le nom d'un service fourni par un agent (modify).
- Rechercher un service (search).

Le service AMS s'utilise généralement de manière transparente (chaque agent créé est automatiquement enregistré auprès de l'AMS et se voit attribué une adresse unique). Ces deux services fournissent donc les annuaires qui permettent à n'importe quel agent de trouver un service ou un autre agent de la plate-forme.

4.3. Les paquetages de JADE

Les sources de la plate-forme JADE sont organisées en une hiérarchie de paquetages et sous-paquetages Java, où chaque paquetage, en principe, contient l'ensemble des classes et des interfaces qui implémentent une fonctionnalité spécifique. Les principaux paquetages sont les suivants[Bel07]:

- ***jade.core*** implémente le noyau de JADE, l'environnement d'exécution distribué qui prend en charge la plate-forme entière et de ses outils. Il comprend la classe fondamentale *jade.core.Agent* ainsi que toutes les classes élémentaires d'exécution nécessaires à la mise en œuvre des conteneurs d'agent. Il comprend également un ensemble de sous-paquetages de mise en œuvre de chacun un service spécifique au niveau du noyau. Ce sont :
  - ***Jade.core.event*** qui implémente le service de notification d'événement distribué. Cela permet aux abonnés d'être informés des événements système générés par les différents composants distribués d'une plate-forme;
  - ***Jade.core.management*** qui implémente l'agent de service de gestion du cycle de vie distribué;
  - ***Jade.core.messaging*** qui implémente le service de distribution de message;
  - ***Jade.core.mobility*** qui implémente la mobilité des agents et le service de clonage, y compris le transfert de l'état et le code d'un agent.

- ***Jade.core.nodeMonitoring*** qui permet aux conteneurs de surveiller les uns les autres et de découvrir des conteneurs inaccessibles ou morts ;
- ***Jade.core.replication*** qui permet la réplication du conteneur principal en tant qu'option de basculement en cas de pannes graves dans le conteneur principal d'origine ;
- ***Jade.core.behaviours*** est un sous-paquetage de ***jade.core*** qui contient une hiérarchie des comportements fondamentaux indépendants des applications. Un comportement JADE représente une tâche qui peut être effectuée par un agent.

- ***jade.content*** et ses sous-paquetages contiennent la collection de classes qui permet aux programmeurs de créer et de manipuler le contenu des expressions complexes selon un langage et ontologie de contenu. Cela comprend tous les mécanismes nécessaires codés pour la conversion automatique entre le format de représentation interne de Jade et le format de transmission de contenu du message conforme au FIPA.
- ***jade.domain*** contient la mise en œuvre de l'AMS et agents DF, comme ils ont été spécifiés par les normes FIPA, plus les extensions spécifiques de JADE. Chaque sous-paquetage contient les classes représentant les différentes entités d'une ontologie prédéfinie (Table 1) de JADE.
- ***jade.gui*** contient des composants à propos général de java et des icônes qui peuvent être utilés pour construire des GUIs pour les agents JADE. Le paquetage fournit plusieurs composants graphiques prêts à l'emploi pour l'affichage des abstractions JADE typiques, par exemple l'AID, le ACLMessage , et le AgentDescription .
- ***jade.imtp*** contient les implémentations de l'IMTP de JADE (Internal Message Transport Protocol). En particulier, le sous-paquetage jade.imtp.rmi est l' IMTP par défaut de JADE qui est basé sur Java RMI .
- ***jade.lang.acl*** contient le support pour le FIPA ACL (Agent Communication Language), y compris la classe ACLMessage, l'analyseur, l'encodeur, et une classe d'assistance pour représenter des modèles de messages ACL.
- ***jade.mtp*** contient l'ensemble des interfaces Java qui devraient être implémentés par une MTP JADE. Il contient également deux sous-paquetages avec une implémentation basée sur le protocole HTTP (qui est l'implémentation par défaut) et une autre reposant sur le protocole IIOP.

Ontologie	Paquetage	Description
FIPA-Agent-Management	jade.domain.FIPAAgent Management	Les entités, les exceptions et les actions nécessaires pour interagir avec l'AMS et le DF, selon les spécifications du FIPA
JADE-Agent-Management	jade.domain.JADEAgent Management	Les extensions JADE à l'ontologie de gestion d'agent-FIPA
JADE-Introspection	jade.domain. introspection	Les extensions JADE liées à la surveillance des événements de plate-forme
JADE-Mobility	jade.domain.mobility	Les extensions JADE liées à la mobilité des agents
JADE-Persistence	jade.domain.persistence	Les extensions JADE liées à la persistance de l'agent
DFApplet-Management	jade.domain.DFGUI Management	Ontologie utilisé par l'interface graphique DF (GUI) pour interagir avec le DF. Il permet plusieurs interfaces graphiques de la même DF, y compris des interfaces graphiques implémentées sous forme d'applets

Table 1. Les ontologies prédéfinies dans JADE [Bel07].

- ***jade.proto*** contient les implémentations de certains protocoles d'interaction à usage général, y compris certains de ceux spécifiés par le FIPA.
- ***jade.tools*** contient l'implémentation de tous les outils graphiques de Jade.
- ***jade.util*** contient plusieurs classes d'utilitaires divers.
- ***jade.wrapper*** avec ***jade.core.Profile*** et les classes de ***jade.core.Runtime*** fournissent un support pour l'interface de JADE qui permet aux applications Java externes à utiliser JADE comme une bibliothèque.
- ***FIPA*** est un paquetage qui comprend le module IDL (Interface Definition Language) spécifié par le FIPA pour la MTP basée sur IIOP.

4.4. Les Agents JADE

L'agent Jade est un agent réactif qui peut être créé par l'instanciation d'une classe qui hérite la classe Jade.core.Agent, il peut être trouvé dans plusieurs états (Figure 6):

- ✓ ***Initialisé***: l'agent est créé mais n'est pas encore enregistré auprès du service de nommage (AMS).
- ✓ ***Active***: l'agent est enregistré auprès du service de nommage (AMS), il possède une adresse unique et peut donc communiquer avec les autres agents.
- ✓ ***Suspendu***: l'exécution de l'agent est suspendue.
- ✓ ***En attente***: l'agent est bloqué et doit attendre un événement comme un message par exemple.
- ✓ ***Transit***: l'agent rentre dans cet état lorsqu'il migre dans un autre conteneur.
- ✓ ***Inexistant***: l'agent est détruit et supprimé du service de nommage (AMS).

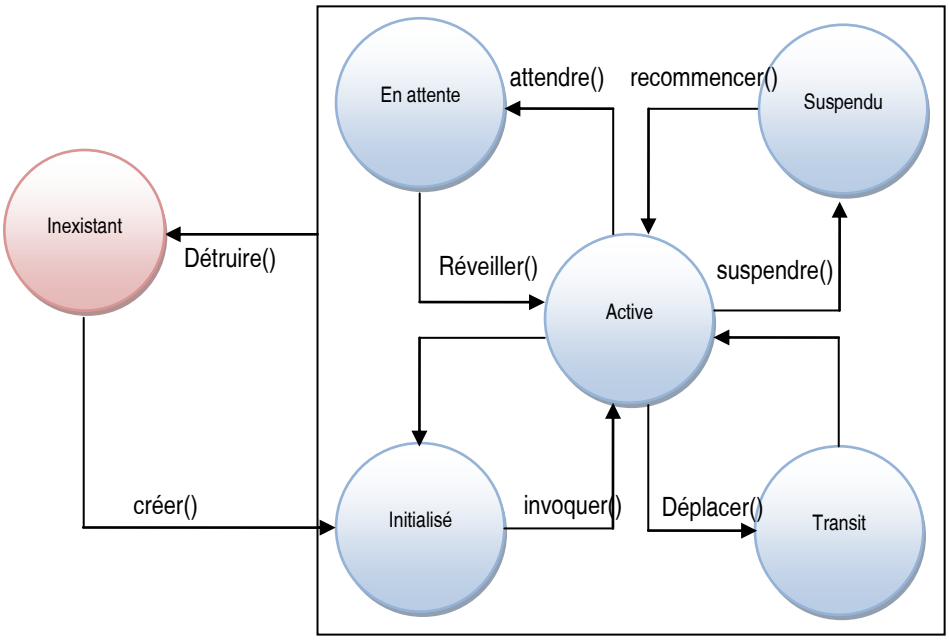


Figure 6. Le cycle de vie d'un agent [Noe07].

4.5. Comportements des agents dans la plate-forme JADE

Un agent doit être capable de gérer plusieurs tâches de manière concurrente en réponse à différents évènements extérieurs. Afin de rendre efficace cette gestion, chaque agent de JADE est composé d'un seul thread et chaque comportement qui le compose est en fait un objet de type *Behaviour*. Des agents multi-thread peuvent être créés mais il n'existe pour l'heure actuelle aucun support fourni par la plate-forme (excepter la synchronisation de la file

des messages ACL). Afin d'implémenter un comportement, le développeur doit définir un ou plusieurs objets de la classe *Behaviour*, les instancier et les ajouter à la file des tâches « ready » de l'agent. Il est à noter qu'il est possible d'ajouter des comportements et sous-comportements à un agent ailleurs que dans la méthode *setup().*Tout objet de type *Behaviour* dispose d'une méthode *action()* (qui constitue le traitement à effectuer par celui-ci) ainsi que d'une méthode *done()* (qui vérifie si le traitement est terminé). Dans les détails, l'ordonnanceur exécute la méthode *action()* de chaque objet *Behaviour* présent dans la file des tâches de l'agent. Une fois cette méthode terminée, la méthode *done()* est invoquée. Si la tâche a été complétée alors l'objet *Behaviour* est retiré de la file. L'ordonnanceur est non-préemptif et n'exécute qu'un seul comportement à la fois, on peut donc considérer la méthode *action()* comme étant atomique. Il est alors nécessaire de prendre certaines précautions lors de l'implémentation de cette dernière, à savoir éviter des boucles infinies ou des opérations trop longues. La façon la plus classique de programmer un comportement consiste à le décrire comme une machine à états finis. L'état courant de l'agent étant conservé dans des variables locales. Enfin, il existe également quelques méthodes supplémentaires afin de gérer les objets *Behaviour* :

- *reset()* qui permet de réinitialiser le comportement;
- *onStart()* qui définit des opérations à effectuer avant d'exécuter la méthode action();
- *onEnd()* qui finalise l'exécution de l'objet *Behaviour* avant qu'il ne soit retiré de la liste des comportements de l'agent;

La plate-forme JADE fournit sous forme de classes un ensemble de comportements (Figure 7) ainsi que des sous-comportements prêt à l'emploi. Elle peut les exécuter selon un schéma prédéfini, par exemple la classe *SequentialBehaviour* est supportée et exécute des sous-comportements de manière séquentielle. Toutes les classes prédéfinies dans JADE héritent de la classe Abstraite *Behaviour*.

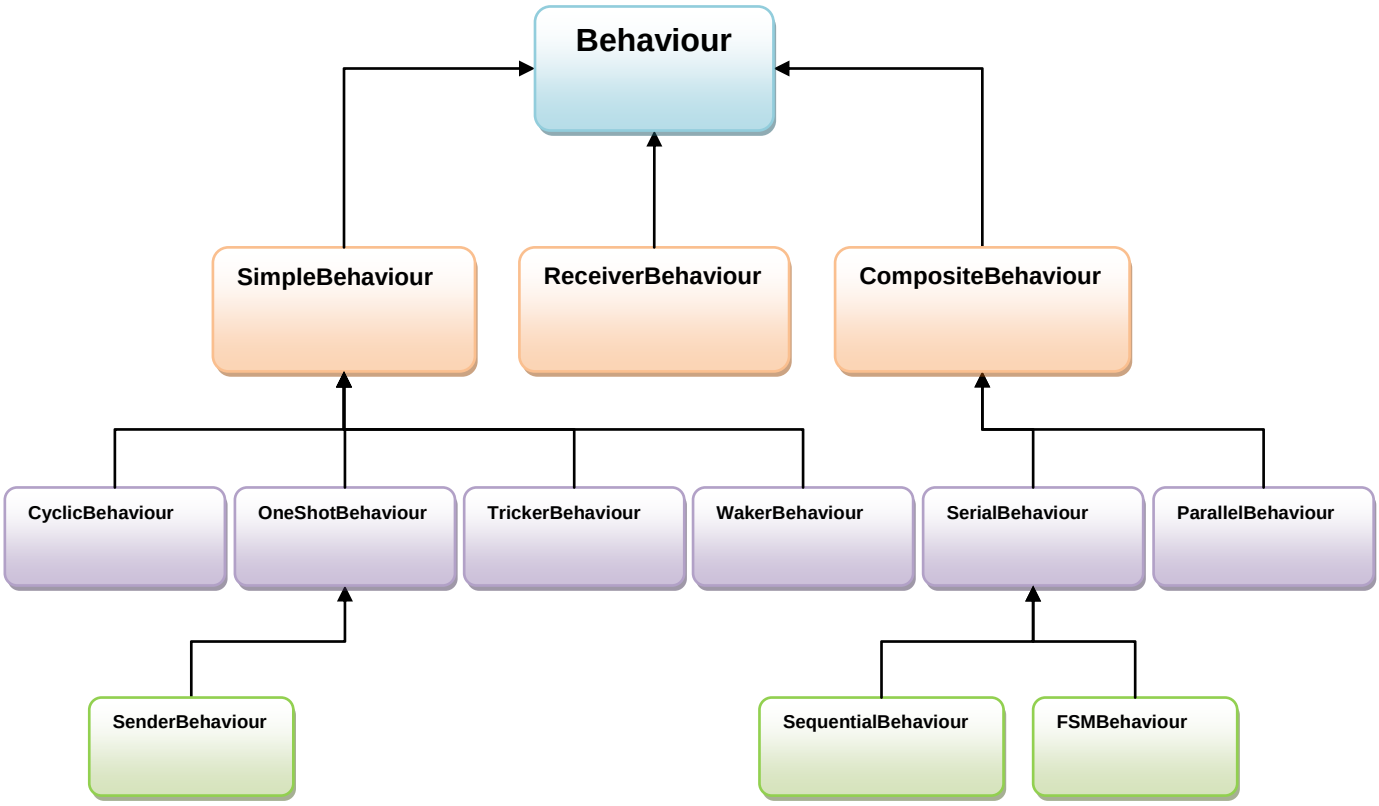


Figure 7. Le graphe d'héritage de la classe Behaviour [Pél02].

4.6. Communication entre Agents

Une fois l'agent activé, il lui faut pouvoir communiquer avec les autres agents. Pour cela, il y a le langage ACL (Acts Communication Language). Le format du message est défini dans la classe jade.lang.acl.ACLMessage. La communication se fait de manière asynchrone, c'est à dire que l'agent peut bloquer un comportement pour qu'il attende la réception d'un message, et pendant ce temps faire d'autres opérations. De cette façon un agent peut avoir plusieurs comportements responsables de plusieurs tâches distinctes en fonction de leur comportement comme vu précédemment lorsqu'un agent souhaite envoyer un message, il doit créer un nouvel objet ACLMessage, compléter ses champs avec des valeurs appropriées et enfin appeler la méthode *send()*. Lorsqu'un agent souhaite recevoir un message, il doit employer la méthode *receive()* ou la méthode *blockingReceive()*.

Un message ACL dispose obligatoirement des champs suivants :

Champs	Description
Performative	type de l'acte de communication
Sender	expéditeur du message
Receiver	destinataire du message
reply-to	participant de la communication
content	contenu du message
language	description du contenu
encoding	description du contenu
ontology	description du contenu
protocol	contrôle de la communication
conversation-id	contrôle de la communication
reply-with	contrôle de la communication
in-reply-to	contrôle de la communication
reply-by	contrôle de la communication

Table 2. Contenu d'un message ACL.

Tous les attributs de la classe ACLMessage peuvent être obtenus et modifiés par les méthodes *set/get.* Le contenu des messages peut être aussi bien du texte que des objets car la sérialisation Java est supportée.

4.7. Les pages jaunes

Le facilitateur d'annuaire (DF) fournit un service de *Pages Jaunes* qui permet à un agent de trouver d'autres agents offrants des services dont il a besoin pour atteindre ses objectifs. Un agent souhaitant publier un ou plusieurs services doit fournir au DF une description, y compris son AID et la liste de ses services. Un agent qui souhaite rechercher des services doit fournir au DF une description du modèle (Template).

Le résultat de la recherche est la liste de toutes les descriptions qui correspondent au modèle fourni. Une description correspond au modèle si tous les champs spécifiés dans le modèle sont présents dans la description avec les mêmes valeurs.

Les services fournis par l'agent DF sont généralement utilisé par tous les agents d'un SMA implémenté en JADE, ainsi que tous les agents qui ont besoin de publier leurs services dans les pages jeunes de l'agent DF vont présenter le code de la Figure 8.

```
DFAgentDescriptiondfd = new DFAgentDescription();
dfd.setName(genericAgent.getAID());
ServiceDescriptionsd = new ServiceDescription();
sd.setType(genericAgent.getServiceType());
sd.setName(genericAgent.getServiceName());
dfd.addServices(sd);
try {
DFService.register(genericAgent, dfd);
}
```

Figure 8. Le code pour enregistrer un service dans les pages jaunes.

Pour dés-enregistrer les services dans les pages jaunes un agent doit présenter le code de la figure 9.

```
try {
DFService.deregister(genericAgent); }
```

Figure 9. Le code pour dés-enregistrer un service des pages jaunes

Tous les agents qui ont besoin de trouver un agent fournisseur spécifique dans les pages jaunes vont présenter le code de la figure 10.

```
DFAgentDescription template = new DFAgentDescription();
ServiceDescriptionsd = new ServiceDescription();
sd.setType(genericAgent.getServiceType());
template.addServices(sd);
try {
DFAgentDescription[] result = DFService.search(genericAgent,
template);
for (int i = 0; i <result.length; ++i) {
genericAgent.getProviders()[i] = result[i].getName(); }
```

Figure 10. Le code pour rechercher dans les pages jaunes sur l'agent qui offre un service.

4.8. Outils de débogage de JADE

Pour supporter la tâche difficile du débogage des applications multi-agents, des outils ont été développés dans la plate-forme JADE. Chaque outil est empaqueté comme un agent, obéissant aux mêmes règles, aux mêmes possibilités de communication et aux mêmes cycles de vie d'un agent générique (Agentification de service) [Fer02].

4.8.1. L'agent *RMA* (Remote Management Agent)

L'agent RMA (figure 11) permet de contrôler le cycle de vie de la plate-forme et tous les agents la composant. L'architecture répartie de JADE permet le contrôle à distance d'une autre plate-forme. Plusieurs RMA peuvent être lancés sur la même plate-forme du moment qu'ils ont des noms distincts.

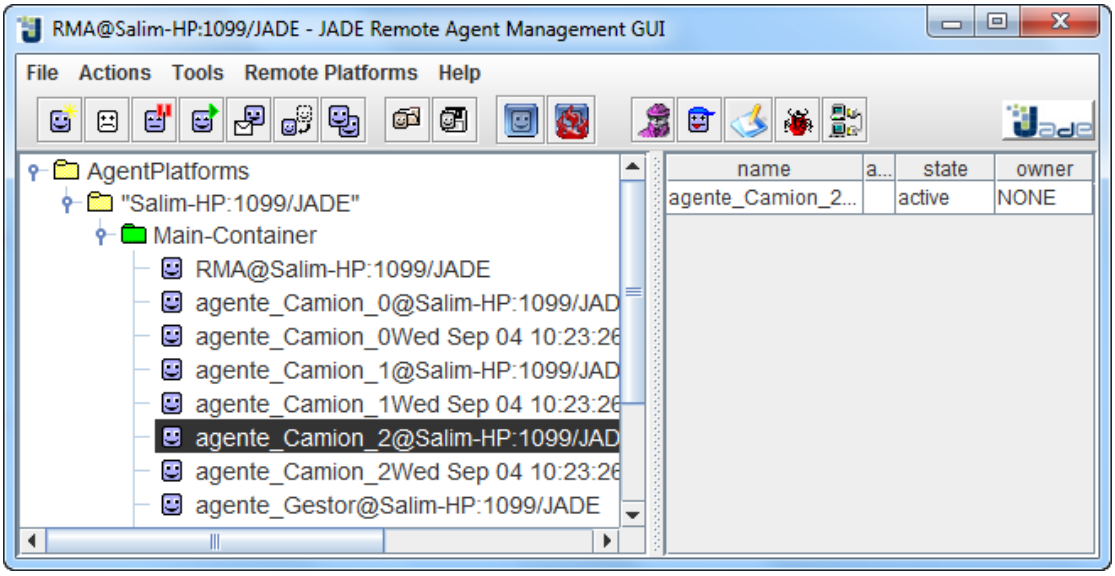


Figure 11. L'interface de l'agent RMA.

4.8.2. L'agent *Dummy*

L'outil *Dummy Agent* (figure 12) permet aux utilisateurs d'interagir avec les agents JADE d'une façon particulière. L'interface permet la composition et l'envoi de messages ACL et maintient une liste de messages ACL envoyés et reçus. Cette liste peut être examinée par l'utilisateur et chaque message peut être vu en détail ou même édité. Plus encore, le message peut être sauvegardé sur le disque et renvoyé plus tard.

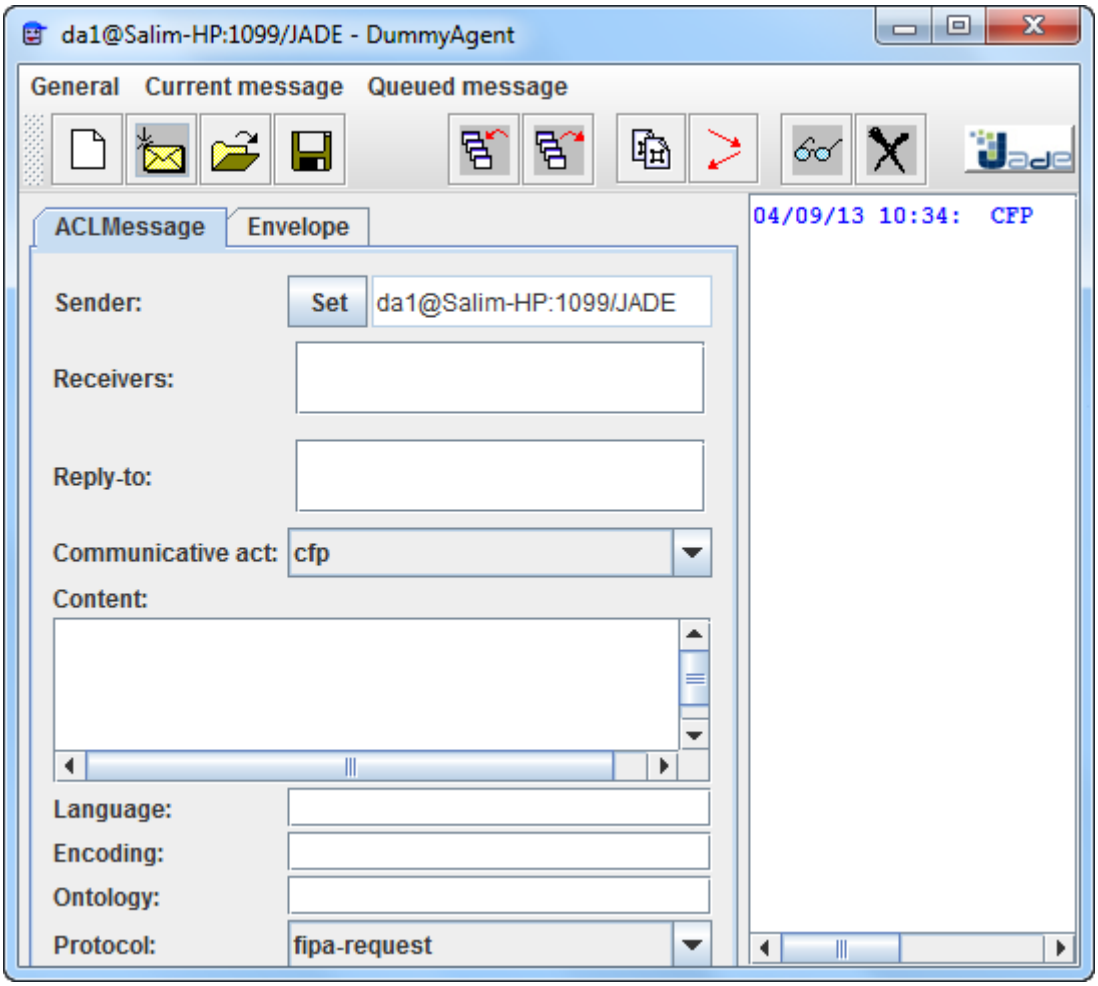


Figure 12. L'interface de l'agent dummy.

4.8.3. L'agent facilitateur d'annuaire (DF)

L'interface du DF (figure 13) peut être lancée à partir du menu du RMA. Cette action est en fait implantée par l'envoi d'un message ACL au DF lui demandant de charger son interface graphique. L'interface peut être juste vue sur l'hôte où la plate-forme est exécutée. En utilisant cette interface, l'utilisateur peut interagir avec le DF.

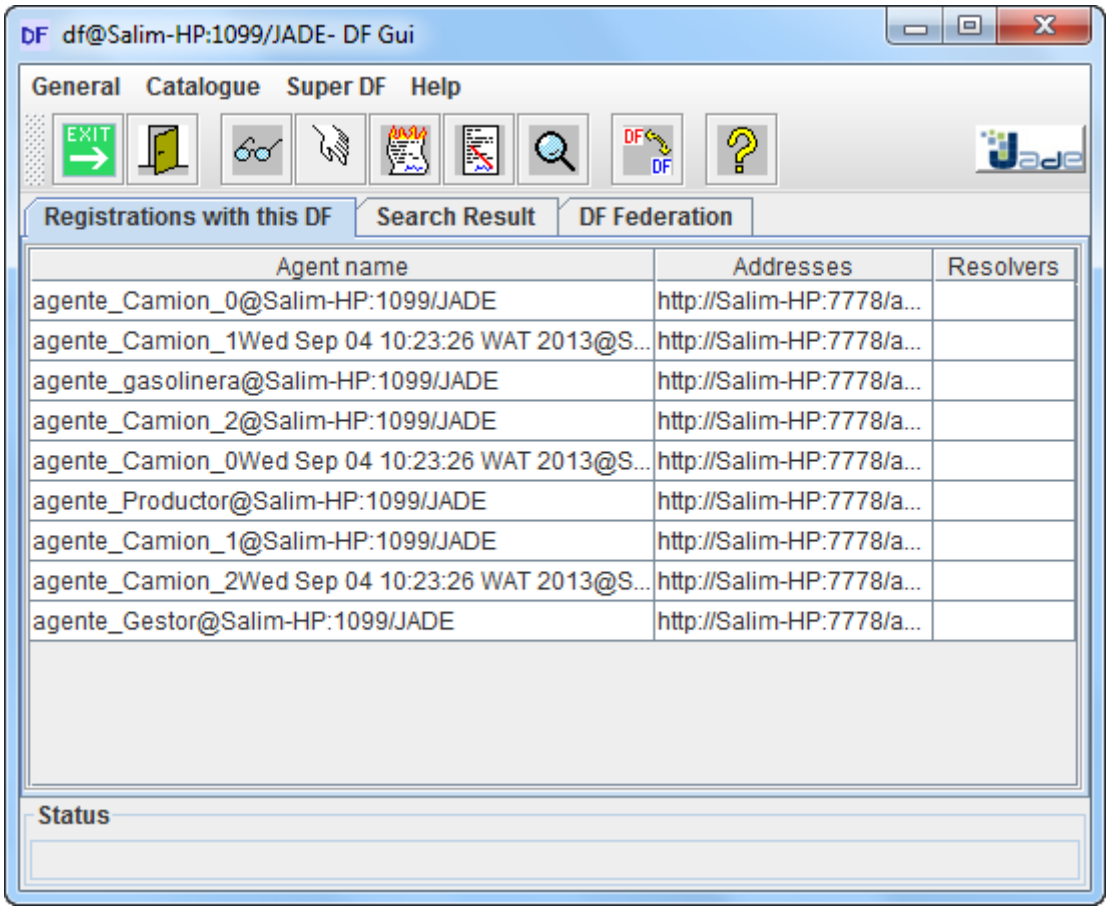


Figure 13. L'interface de l'agent DF.

4.8.4. L'agent *Sniffer*

Quand un utilisateur décide d'épier un agent ou un groupe d'agents, il utilise un agent *sniffer* (figure 14). Chaque message partant ou allant vers ce groupe est intercepté et affiché sur l'interface du sniffer. L'utilisateur peut voir et enregistrer tous les messages, pour les analyser plus tard. L'agent peut être lancé du menu du RMA ou de la ligne de commande suivante : *Java jade.Boot sniffer:jade.tools.sniffer.sniffer*

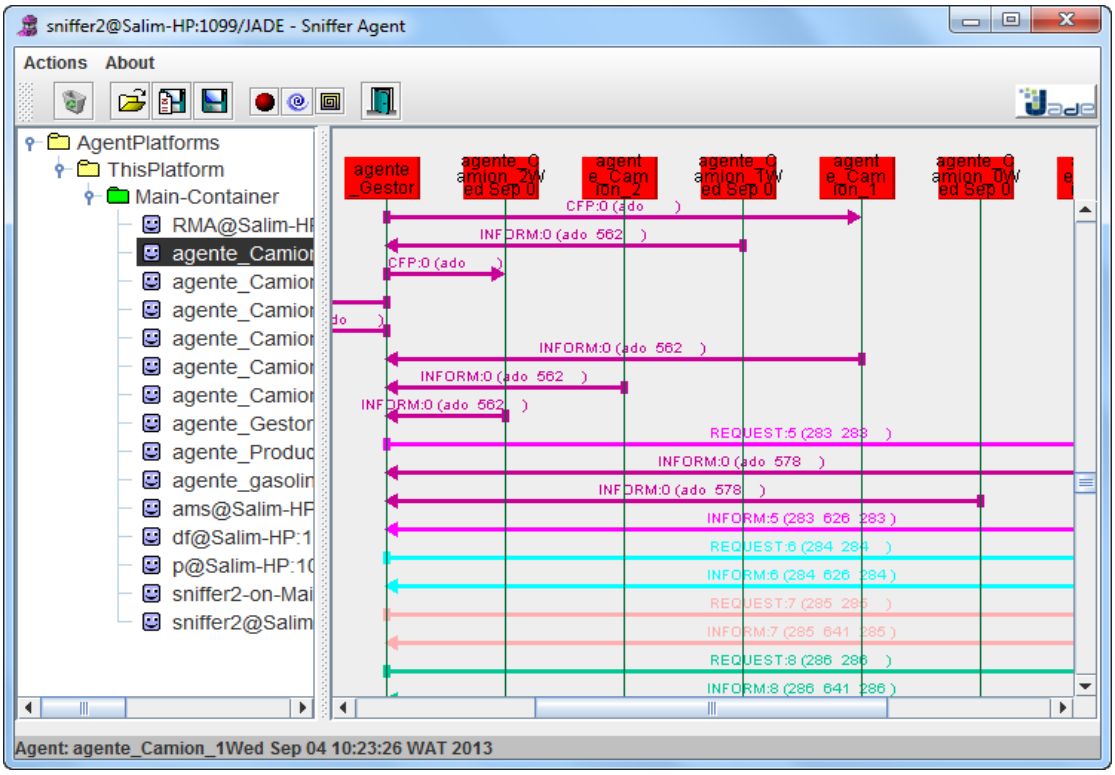


Figure 14. L'interface de l'agent sniffer.

4.8.5. L'agent Introspector

Cet agent (figure 15) permet de gérer et de contrôler le cycle de vie d'un agent s'exécutant et la file de ses messages envoyés et reçus.

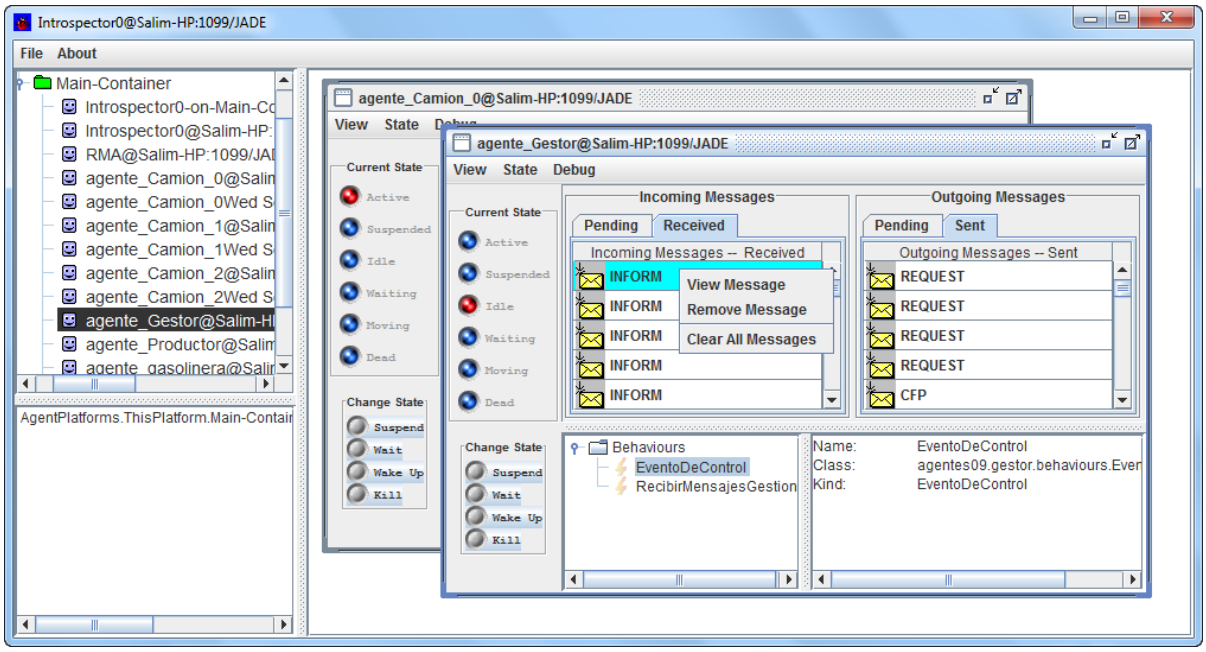


Figure 15. L'interface de l'agent introspector.

4.8.6. L' agent Log Manager

L'agent de gestion du journal *Log Manager* (figure 16) est un outil qui simplifie la gestion dynamique et distribuée du système de traçage en fournissant une interface graphique qui permet aux niveaux de journalisation de chaque composant de la plate-forme JADE d'être changés au moment de l'exécution. Cela inclut tous les composants étant exécutées au niveau des nœuds distants, y compris les messages de journalisation spécifiques à l'application. Le gestionnaire du journal exploite les capacités de l'APIs *"java.util.logging"* sur lequel est basée la journalisation de JADE

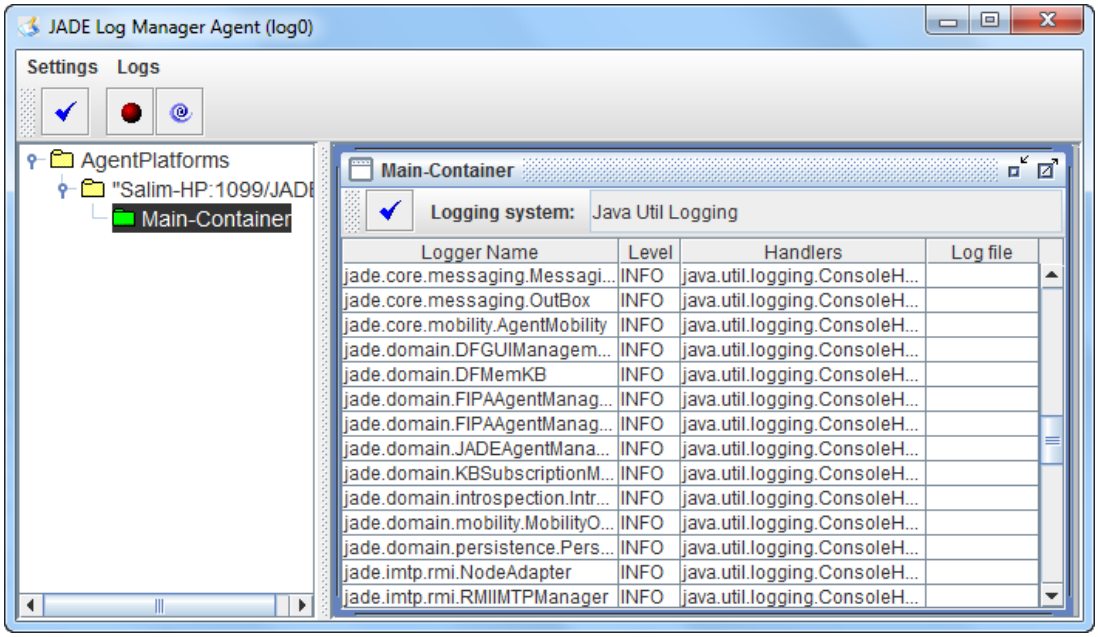


Figure 16. L'interface de l'agent Log Manager.

5. Conclusion

Les systèmes multi-agents présentent un domaine vaste et important et sa description dépasse le contenu d'un chapitre. Pour cela, nous nous sommes limités dans ce chapitre à présenter la plupart des concepts relatifs à notre travail. Par ailleurs, nous avons donné un bref aperçu sur les plateformes les plus utilisées dans la littérature tout en mettant l'accent sur la plateforme JADE sur laquelle est basée notre approche d'extraction d'aspects.

Chapitre 02

La Programmation Orientée Aspect et le Tisseur ASPECTJ

1. Introduction

La plupart des logiciels actuels répondent à diverses préoccupations (en anglais *concerns*), nous pouvons distinguer des préoccupations fonctionnelles (en anglais *core concerns*) et des préoccupations non-fonctionnelles dites transversales (en anglais *crosscutting concerns*) [Bal02].

- Les exigences fonctionnelles décrivent le comportement du système et définissent les fonctions ou les services que le système doit remplir.
- Les exigences non-fonctionnelles expriment les qualités ou contraintes imposées sur la manière de satisfaire les exigences fonctionnelles. Il s'agit donc principalement des exigences de performance, sécurité, sûreté, convivialité, concurrence, etc.

Par exemple :

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

Besoin fonctionnel (*core concerns*) Besoin non-fonctionnel (*crosscutting concerns*)

Puisqu'il est difficile d'implémenter les préoccupations non-fonctionnelles dans des modules propres à cause des limites des méthodes de programmation orientée objet, ils sont fusionnées avec les préoccupations fonctionnelles situées dans les modules métiers (traverse le code du comportement principal).

La programmation orientée aspect apporte une solution élégante et simple à ce problème. Cette nouvelle méthode de programmation permet d'implémenter chaque problématique indépendamment des autres, puis, de les assembler selon des règles bien définies. La programmation orientée aspect promet donc une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements [*Bal02*].

1.1. Où sommes-nous avec la POO ?

L'utilisation de la POO au lieu des techniques de la décomposition fonctionnelle a considérablement amélioré l'état des logiciels. Les avantages de l'utilisation des technologies orientées objet à toutes les phases du processus de développement des logiciels sont variées :

- La réutilisation des composants.

- Modularité.
- Moins complexe dans l'implémentation.
- Réduction des coûts de maintenance.
- Sûreté (ou bien la sécurité qui a été fournie par la notion de l'encapsulation (modes d'accès - privé, publique, protégé))
- Extensibilité.

Chacun de ces avantages aura une importance variée pour les développeurs. L'un d'eux, la modularité, est un progrès universel sur la programmation structurée qui mène à des logiciels plus propres mieux plus compréhensible [Gra03].

1.2. Les limites de la POO

L'intérêt de la POO pour le développement des logiciels complexes est indéniable. Néanmoins, il a été montré que, dans au moins deux cas, la POO ne fournit pas de solution satisfaisante pour aboutir à des programmes clairs et élégants. Ces cas concernent l'enchevêtrement du code et le phénomène de dispersion du code (éparpillement du code).

1.2.1. Enchevêtrement du code

L'enchevêtrement du Code (figure 1) est causé quand un module est implémenté pour traiter des multiples préoccupations simultanément.

Les développeurs souvent mettent en considération des préoccupations telles que le code métier, les performances, la synchronisation, gestion de Trace, la sécurité … etc lors de l'implémentation d'un module (figure 2).

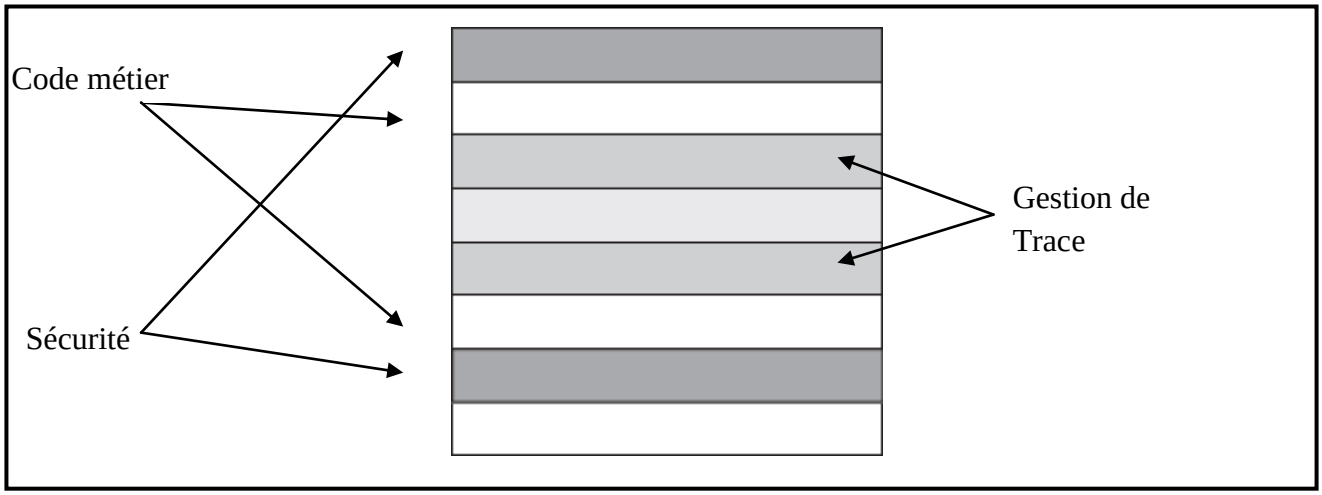


Figure 1. Enchevêtrement du code causé par plusieurs implémentations simultanées des diverses préoccupations.

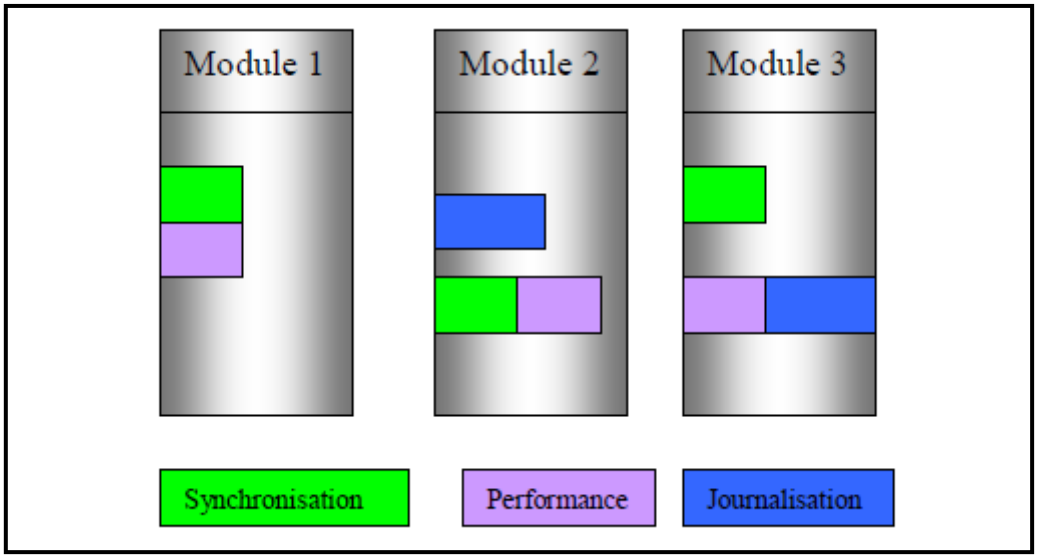


Figure 2. Les exigences non-fonctionnelles traversent la modularisation fonctionnelle d'un système [Bal02].

Un autre exemple pour illustrer l'enchevêtrement du code est la notion d'espace multi-dimensionnel de préoccupations (figure 3). Imaginons que les besoins sont projetés sur un espace multi dimensionnel où chaque préoccupation représente une dimension. Dans cet espace, chaque préoccupation est indépendante et peut évoluer sans affecter les autres préoccupations. Par exemple, le changement du schéma de sécurité n'affecte pas le code métier. Cependant, un espace multidimensionnel de préoccupation est réduit dans un espace uni dimensionnel d'implémentation [Lad09].

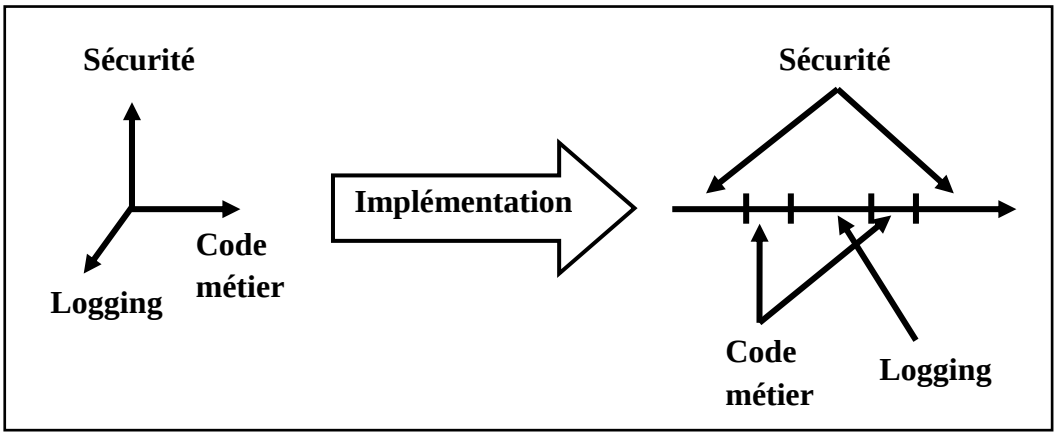


*Figure 3. Espace des préoccupations et espace d'implémentation [*Keb10*].*

Tant que l'espace d'implémentation est unidimensionnel, on se concentre souvent sur l'implémentation de la préoccupation qui a un rôle dominant ; les autres préoccupations sont alors enchevêtrées avec la préoccupation dominante [Keb10]. Donc, il vaut mieux que chaque préoccupation transversale doit être implémentée dans un module propre. Ce qui est très difficile avec la POO.

1.2.2. Dispersion du code

En POO le mécanisme principal d'interaction entre objets est l'invocation de méthode. Un objet qui souhaite effectuer un traitement invoque une méthode d'un autre objet. Un objet peut aussi invoquer une de ses propres méthodes. Dans tous les cas, il existe un rôle d'invocateur et un rôle d'invoqué.

L'invocation de méthode permet de s'abstraire de la façon dont un service est implémenté en se reposant complètement sur l'interface de l'objet invoqué. Il suffit que les paramètres transmis par l'invocateur soient conformes à la signature de la méthode pour que l'invoqué prenne en charge la demande.

L'implémentation d'une méthode en POO est clairement localisée puisqu'elle se situe dans une classe. La modification d'une méthode est une opération simple: il suffit de modifier le seul fichier qui contient la classe dans laquelle est définie la méthode. Lorsque la modification porte sur le code de la méthode, la modification est transparente pour tous les objets qui invoquent la méthode.

La modification de la signature de la méthode a davantage de conséquences, puisqu'il est nécessaire de modifier toutes les classes qui invoquent cette méthode.

Ces modifications sont d'autant plus coûteuses que la méthode est d'usage courant. En POO, l'implémentation est localisée dans une classe, tandis que son invocation, ou utilisation, est dispersé. Ce phénomène de dispersion du code (figure 4) est un frein à la maintenance et à l'évolution des applications orientées objet. Toute modification dans la manière d'utiliser un service entraine des modifications nombreuses, coûteuses et sujettes à une erreur.

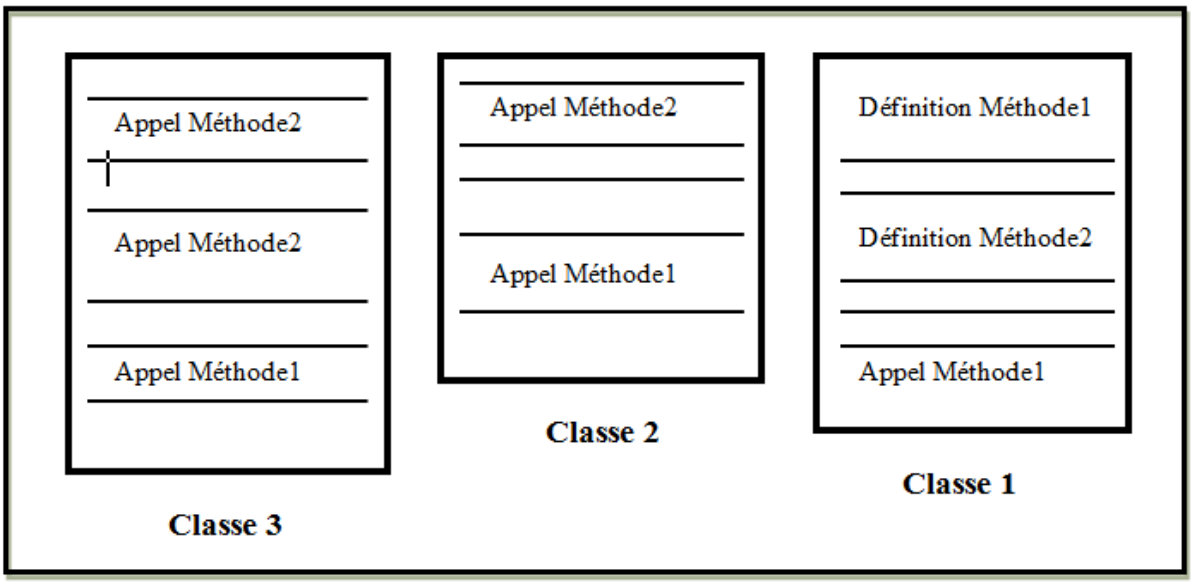


Figure 4. Dispersion de code dans un programme.

À cause de ces deux problèmes le processus de développement de logiciel est très touché et parmi ses mauvaises conséquences nous pouvons citer:

- ***Traçage difficile*****:** les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les exigences et leurs implémentations.
- ***Diminution de la productivité*****:** la prise en considération de plusieurs exigences au sein d'un même module empêche le programmeur de se focaliser uniquement sur son but premier. Le danger d'accorder trop ou pas assez d'importance aux aspects accessoires d'un module en découle directement.
- ***Diminution de la réutilisation du code*****:** un module peut implémente de multiples exigences. D'autres systèmes nécessitant des fonctionnalités similaires pourraient ne pas

pouvoir réutiliser le module tel qu'il est, entraînant de nouveau une diminution de la productivité.

- ***Diminution de la qualité du code*****:** les programmeurs ne peuvent pas se concentrer sur toutes les contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entraîner des effets de bords non-désirés (i.e. des bugs).

- ***Maintenance et évolutivité du code difficile*****:** lorsque l'on veut faire évoluer le système, on doit modifier de nombreux modules. Modifier chaque sous-système pour répercuter les modifications souhaitées peut conduire à des incohérences.

2. Définition de la Programmation Orienté aspect (POA)

La programmation Orientée Aspect [Paw04] est un paradigme de programmation qui a été défini par la société Xerox et qui permet de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel. Le principe est donc de coder chaque problématique séparément et de définir leurs règles d'intégration pour les combiner en vue de former le système final.

Par rapport à l'orienté objet, cette technique permet aux programmeurs d'encapsuler des comportements qui affectaient de multiples classes dans des modules réutilisables. La programmation OA permet donc d'encapsuler dans un module les préoccupations qui se recoupent avec d'autres.

La POA est une extension de langage de programmation. Elle peut être appliqué au Java, C, C++, C#. Il est alors possible de l'appliquer sur les langages de programmation procédurale ou les langages orientés Object [2]. Sa méthodologie fournit une séparation des préoccupations transversales, en introduisant une nouvelle unité de modularisation qui est l'*aspect* (figure 5).

Chaque aspect se concentre sur une fonctionnalité transversale spécifique. Les classes de base (fonctionnelles) ne sont plus accablées par les préoccupations transversales (non fonctionnelles).

Un *aspect tisseur* compose le système final en combinant les classes de base (fonctionnelles) et les aspects transversaux à travers un processus appelé *Tissage.* Ainsi, la POA permet de créer des applications qui sont plus faciles à concevoir, implémenter et à maintenir.

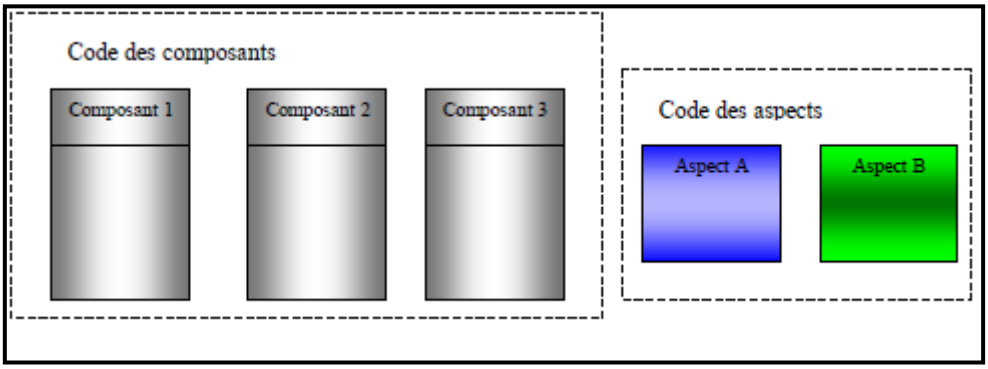


Figure 5. Un système réalisé par la POO et la POA [Bal02].

3. Avantages de la POA

Le couplage entre les modules gérant des aspects techniques peut être réduit de façon très importante, en utilisant ce principe, ce qui présente de nombreux avantages :

- **Maintenance aisée :** les modules techniques, sous forme d'aspect, peuvent être maintenus plus facilement du fait de son détachement de son utilisation.
- **Meilleure réutilisation :** tout module peut être réutilisé sans se préoccuper de son environnement et indépendamment du métier ou du domaine d'application. Chaque module implémente une fonctionnalité technique précise. On n'a pas besoin de se préoccuper des évolutions futures : de nouvelles fonctionnalités pourront être implémentées dans de nouveaux modules qui interagiront avec le système au travers des *aspects*.
- **Gain de productivité :** le programmeur ne se préoccupe que de l'**aspect** de l'application qui le concerne, ce qui simplifie son travail, et permet d'augmenter la parallélisation du développement.
- **Amélioration de la qualité du code :** la simplification du code qu'entraine la programmation par aspect permet de le rendre plus lisible et donc de meilleure qualité [3].

4. Inconvénients de la POA

Le tissage d'aspect qui n'est finalement que de la génération automatique de code inséré à certains points d'exécution du système développé, produit un code qui peut être difficile à analyser (parce qu'il est généré automatiquement) lors des phases de mise au point des

logiciels (débogage, test). Mais en fait, cette difficulté est du même ordre que celle apportée par toute décomposition non linéaire (par exemple fonctionnelle ou objet).

Une implémentation comme AJDT, basée sur AspectJ, offre des outils sophistiqués qui permettent de passer de façon transparente, en mode débogage, du code d'une classe à celui d'un aspect [3].

5. Les apports de la programmation orientée aspect

La programmation orientée aspect complémente la POO en apportant des solutions aux deux défis que sont l'enchevêtrement du code et le phénomène de dispersion de code. La programmation procédurale met en avant un découpage des fonctionnalités de l'application. La POO insiste davantage sur un regroupement de données et des traitements associés sous forme d'entités cohérentes. La POA rétablit quant à elle un certain équilibre en permettant de superposer au découpage orienté par les données de la POO des découpages qui correspondent à des fonctionnalités supplémentaires à intégrer aux applications orientées objet [Paw04].

Une application orientée aspect fondée sur la POO est composée de deux parties, les classes et les aspects :

- Les classes constituent le socle de l'application. Il s'agit des données et des traitements qui sont au cœur de la problématique de l'application et qui répondent aux besoins premiers de celle-ci.
- Les aspects intègrent aux classes des éléments (classes, méthodes, données) supplémentaires, qui correspondent à des fonctionnalités transversales ou à des fonctionnalités dont l'utilisation est dispersé.

Les langages orientés objet se présentent comme des extensions de langages existants par exemple C++ est une extension de C, soit comme de nouveaux langages, tels Java, Dans tous les cas, il a fallu construire de nouveaux compilateurs pour ces langages.

La situation est légèrement différente avec la POA, car ses outils peuvent se présenter sous forme d'extensions de langages existants ou de Frameworks :

- Dans la catégorie des extensions de langages existants, le langage AspectJ (nous le verrons plus tard), ajoute de nouveaux mots-clés au langage java afin que le développeur puisse manipuler tous les concepts (aspect, coupe, point de jonction, etc.) propres à la POA.

- Dans la catégorie des frameworks, JAC (Java Aspect Component), JBoss AOP et AspectWerkz, utilisent la syntaxe d'un langage de programmation existant, en l'occurrence java. Les concepts de la POA sont manipulés via des classes et méthodes appartenant au framework. Par exemple, définir un aspect avec JAC revient à étendre une classe d'un framework.

6. Réalisation de la POA

Pour réaliser la méthodologie de la POA, nous avons besoin de :

- Langage de mise en œuvre des aspects (fonctionnels et Non-fonctionnels) : Java, C, C++,…
- Langage de spécification des règles de composition des aspects : AspectC, AspectJ, etc.
- Outil d'intégration des aspects (weaving) : aspect Weaver, Compilateur AOP (AspectC, AspectJ, …).

6.1. Le Processus de développement en POA

En général, le cycle de développement en POA se fait en trois étapes (figure 6) :

• **La décomposition aspectuelle:** consiste à décomposer les besoins afin d'identifier et séparer les problématiques transverses et métiers. Cette phase est souvent comparée au passage d'un rayon de lumière à travers un prisme afin de séparer ses différentes composantes chromatiques.

• **Implantation des préoccupations:** consiste à implanter chaque problématique séparément. Les problématiques métiers sont implantés moyennant les techniques conventionnelles de la POO alors que les problématiques transverses sont implantées moyennant les techniques de la POA.

• **Recomposition aspectuelle:** consiste à construire le système final en intégrant ou recoupant les problématiques métiers avec les problématiques transverses. Cette phase est appelée Tissage (weaving en anglais) (figure 7) . Un tisseur (weaver) utilise des règles spécifiées par le concepteur de l'application afin de recouper correctement les problématiques entre-elles. Par analogie, nous pouvons comparer cette étape à un nouveau passage des composantes

chromatiques dans un prisme qui les combine pour faire sortir un rayon de lumière unique. La figure 6 illustre ce processus :

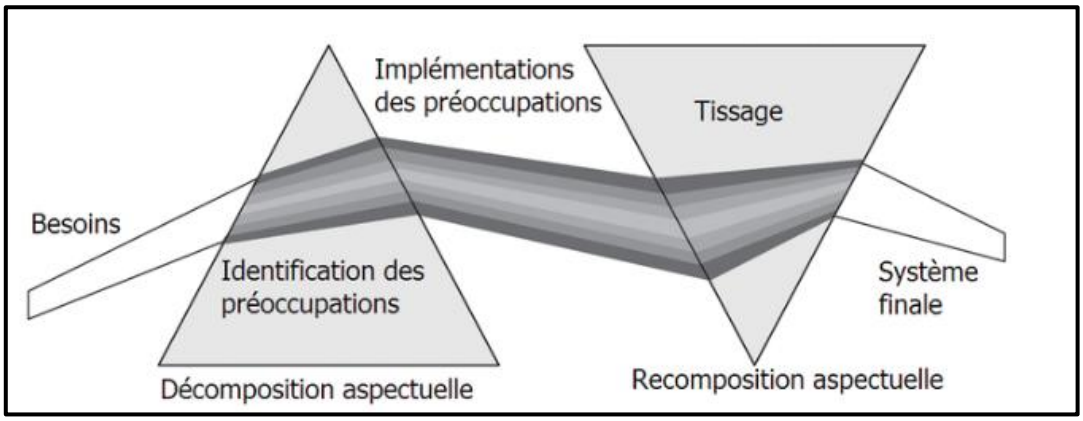


Figure 6. Cycle de développement en POA [Keb10].

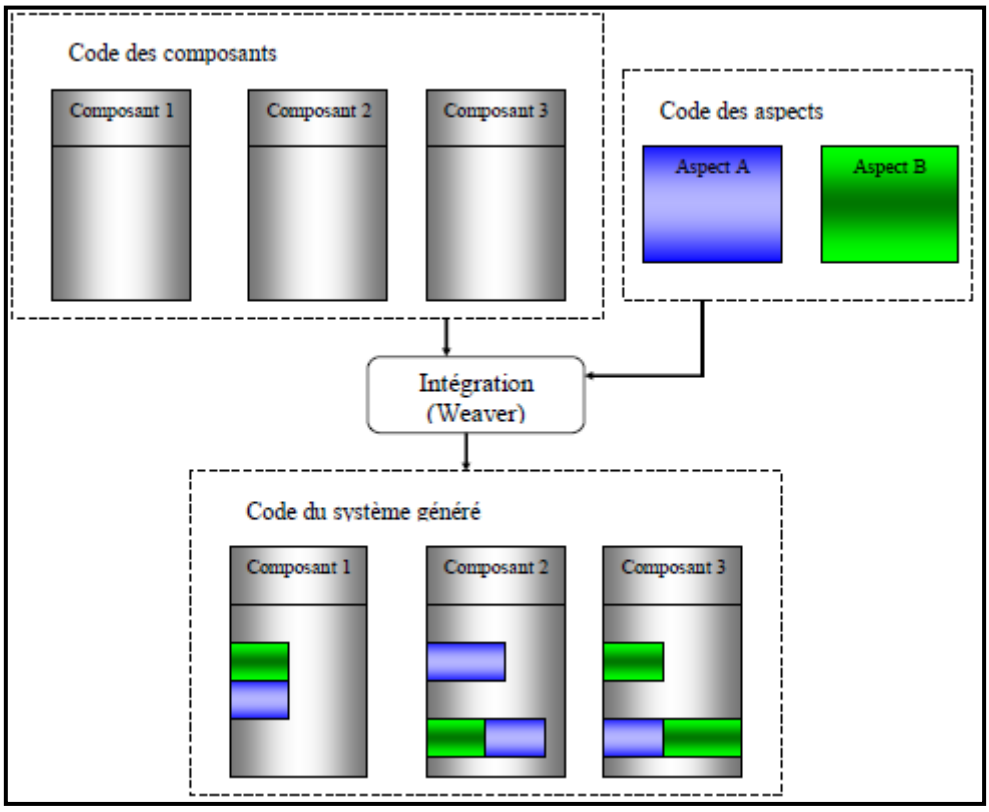


Figure 7. Intégration du code des composants et des aspects pour former le système final [Bal02].

7. Concepts et terminologie POA

De nouveaux concepts sont introduits avec la POA afin de permettre aux développeurs de spécifier et implanter les préoccupations transversales.

7.1. Aspect

Un aspect est une entité logicielle qui capture une fonctionnalité transversale à une Application [Paw04].

Par comparaison avec la programmation orientée objet, si une classe est une responsabilité (la clientèle, les commandes, les fournisseurs…), nous pouvons dire qu'un aspect est une fonctionnalité (la sécurité, la persistance des données, la gestion des traces…).
Classe dans POO ≡ aspect dans POA

Il ya trois éléments principaux dans un aspect qui sont: les coupes (*pointcuts*), les codes *advice* et le mécanisme d'introduction.

7.2. Points de jonction (Join Point)

Un point de jonction est un Point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés. [Paw04]

L'utilisation des points de jonction se fait essentiellement sous forme d'ensemble. Une coupe définissant tous les points de l'application auxquels elle souhaite greffer l'aspect. Ces points de l'application peuvent être l'appel d'une méthode, l'exécution d'un constructeur, la lecture d'un attribut...etc. Nous détaillons les différents types de points de jonction ci-après:

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 différents scénarios d'exécution d'une application peuvent être exprimés en termes de séquences de messages qui déclenchent l'exécution de méthodes. Les appels et les exécutions de méthodes sont donc deux types de points de jonction couramment utilisés.

Les constructeurs : ils 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.

Classes et interfaces : Ne possédant pas d'équivalent en termes d'exécution. On peut certes vouloir désigner tout les points de jonction de toutes les méthodes de cette classe ou des classes qui implémente cette interface.

Les attributs : Les langages orientés aspect considèrent les opérations de lecture et d'écriture sur les attributs comme des types de points de jonction. Un exemple concret d'utilisation de ce type de point de jonction est la gestion transactionnelle de la persistance des données.

Les exceptions : ils sont levées (throw) pour signaler une situation d'exécution anormale et elles sont capturées (catch) pour exécuter un traitement particulier. Ces deux événements (le throw et le catch) sont des points d'exécution très importants dans une application. Ils sont tous les deux considérés par la plupart des langages orientés aspect comme des types de points de jonction.

Blocs de code statique (*seulement qui sont d'hors des méthodes***)** : Les blocs de code *statique* associes aux classes, sont prisent en compte par la POA. Alors on peut développer des aspects qui effectuent des traitements après le chargement d'une classe et avant toute instanciation de cette classe.

Instructions (*if, for, while, switch...etc.***):** Pratiquement les points de jonction correspondantes ne sont pas pris en compte par les outils de POA.

7.3. Coupe (Crosscut)

Une coupe désigne un ensemble de points de jonction [Paw04].
Les points de jonction et les coupes sont liés par leur définition, mais leur nature est très différente. Un point de jonction représente un point dans l'exécution d'un programme, une coupe est un morceau de code défini dans un aspect.

Dans les cas simples, une seule coupe suffit pour définir la structure transversale d'un aspect. Dans les cas plus complexes, un aspect est associé à plusieurs coupes.

En général, la définition d'une coupe passe par l'utilisation d'une syntaxe et de mots-clés. Chaque outil de la POA fournit sa propre syntaxe et ses propres mots-clés. Ces mots-clés identifient des ensembles de points de jonction. Ces ensembles sont unis dans la coupe par des opérations ensemblistes comme par exemple : intersection, union et complémentarité.

Les mots-clés utilisés par la coupe désignent chacun un ensemble de points de jonction en spécifiant son type (appel de méthode, lecture d'attributs, ...), et une expression qui précise le type (la méthode A, l'attribut B, ...). Ces expressions peuvent faire usage de quantificateurs, par exemple toutes les méthodes dont le nom est *add*.

7.4. Greffon : Transversalité dynamique

Un code greffon est un bloc de code définissant le comportement d'un aspect [Paw04].

Un code greffon est toujours associé à une coupe ou plus exactement aux points de jonctions sélectionnés par cette coupe. En effet, un code greffon n'est jamais appelé manuellement, mais il est invoqué chaque fois qu'un point de jonction, sélectionné par la coupe à laquelle il est associé, survient.

Avant d'écrire ce code greffon il faut décider à quel moment exécuter le code : avant l'événement, après ou autour de l'événement (before, after, around).

Lorsqu'il est exécuté autour du point de jonction, il peut carrément remplacer l'exécution de ce dernier, ou bien lui redonner le contrôle.

- ***Les différents types de code greffon :***

  Comme nous avons mentionné il y a trois types principaux de code greffon:

Before : Le code du greffon est exécuté avant l'exécution du code intercepté. Le déroulement d'une greffe **before** peut être récapitulé ainsi:

1. Exécution normale du programme.
2. Juste avant un point de jonction appartenant à la coupe, exécution du code advice.
3. Reprise de l'exécution du code du point de jonction intercepté.
4. Fin du point de jonction, suite de l'exécution du programme.

After : Le code du greffon est exécuté après l'exécution du code intercepté. Le déroulement d'une greffe **After** peut être récapitulé ainsi:

1. Exécution normale du programme.
2. Exécution du code du point de jonction appartenant à la coupe.
3. Juste après la fin du code de ce point de jonction, exécution du code advice.
4. Reprise de l'exécution du programme juste après le point de jonction.

Le greffon *after* possède deux sous-types qui sont:

Afterreturning : Le code du greffon est exécuté si l'exécution du code intercepté s'est correctement effectuée. Le déroulement d'une greffe **Afterreturning** peut être récapitulé ainsi:

1. exécution normale du programme.
2. Exécution du code du point de jonction appartenant à la coupe.
3. Juste après la fin du code de ce point de jonction et si aucune erreur n'est survenue, exécution du code greffon.

4. Reprise de l'exécution du programme juste après le point de jonction.

Afterthrowing : Le code du greffon est exécuté si l'exécution du code intercepté a provoqué une erreur. Le déroulement d'une greffe ***afterthrowing*** peut être récapitulé ainsi:

1. exécution normale du programme.
2. Exécution du code du point de jonction appartenant à la coupe.
3. Juste après la fin du code de ce point de jonction et si une erreur est survenue, exécution du code greffon.
4. Reprise de l'exécution du programme juste après le point de jonction.

Around : Le code du greffon *(advice)* est exécuté avant et après l'exécution du code intercepté. Ainsi, un greffon **around** est composé de deux parties **avant** et **après**. Le déroulement d'une greffe **around** peut être récapitulé ainsi:

1. Exécution normale du programme.
2. Juste avant un point de jonction appartenant à la coupe, exécution de la partie avant du code advice.
3. Si certains traitements de la partie **avant** imposent la non-exécution du code intercepté (par exemple, pour des raisons de sécurité), le point de jonction ne fera pas appel à la méthode **proceed** (*continuer*) du greffon. Cette méthode permet d'exécuter le code intercepté. Si elle n'est pas appelée, le code intercepté est remplacé par le code du greffon.
4. Exécution du code correspond au point de jonction.
5. Exécution de la partie *après* du greffon
6. Reprise de l'exécution du programme juste après le point de jonction.

7.5. Mécanisme d'introduction : Transversalité Statique

Le mécanisme d'introduction est un mécanisme d'extension permettant d'introduire de nouveaux éléments structuraux au code d'une application [Keb10].

Le mécanisme d'introduction de la POA permet d'étendre des classes en y ajoutant des éléments. Ces éléments sont essentiellement des attributs ou des méthodes, mais ce ne sont pas les seuls exemples. AspectJ propose notamment l'ajout d'interfaces Java et même l'ajout d'une superclasse.

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.

7.6. Tissage (weaving)

- *Tisseur d'aspect (aspect weaver) : Programme qui réalise une opération d'intégration entre un ensemble des classes et un ensemble d'aspects [Paw04].*
- *Le tissage (weaving) (figure 8) 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 [Keb10].*

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 (aspect weaving). Le programme qui réalise cette opération est appelé tisseur d'aspects (aspect weaver) ou bien tisseur (weaver) tout court.

Le tissage d'aspect peut être effectué soit:

- Statiquement, à la compilation de l'application
- Dynamiquement, durant l'exécution du programme

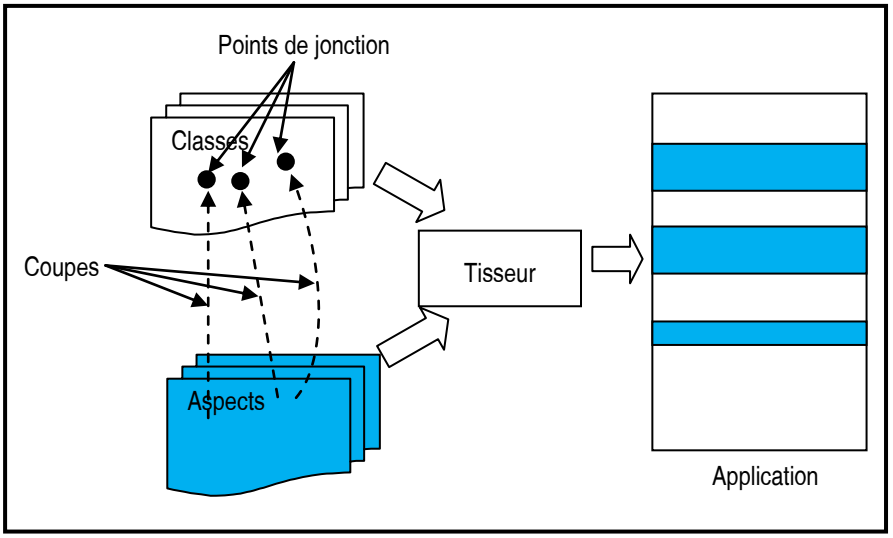


Figure 8. Tissage des aspects [Keb10].

8. Différents Outils de la POA

Nous avons ci-dessous une liste des outils de POA les plus connus, classés par langage de programmation [*2*] :

Langage	*Outil*	*Type*
JAVA	**AspectJ:** Extension du langage Java. Tisseur d'aspect au niveau du code source ou du bytecode.	Statique
	JbossAOP: Extension du langage Java. Existe en version standalone ou en web application JBoss.	statique / dynamique
	JAC (Java Aspect Components): Framework 100% Java. Le tisseur d'aspect intervient au niveau bytecode.	dynamique
	AspectWerkz: Framework 100% Java. Le tisseur d'aspect intervient au niveau bytecode.	statique / dynamique
	Spring AOP: Framework 100% Java. Le tisseur d'aspect intervient au niveau bytecode.	dynamique
.Net (C#, VB.Net, ...)	**AspectDNG**	statique
C	**Aspect-C**	(inconnu)
C++	**AspectC++**	(inconnu)
PHP	**phpAspect**	statique
Caml	**AspectualCaml**	(inconnu)

Table 1. Différents outils de la POA [2].

9. Le tisseur AspectJ

Historiquement, AspectJ est le premier outil de programmation orientée aspect en Java, créé par Gregor Kiczales et son équipe du Xerox PARC. Un premier prototype d'AspectJ a été réalisé en 1998. La première version officielle d'AspectJ, désignée AspectJ 1.0, a été réalisée en novembre 2001.

AspectJ étend le langage Java et fournit un compilateur pour traduire les aspects en programmes exécutables. Cette extension est disponible dans les projets open-source Eclipse, de manière autonome ou sous forme d'extension pour l'environnement de développement Eclipse « plugin AJDT ». AspectJ est devenu le standard, du fait de son utilisation répandue,

pour la Programmation Orientée Aspect en mettant l'accent sur la simplicité et la facilité de l'implémentation.

9.1. Construction des préoccupations transversales avec Aspectj

AspectJ permet également aux programmeurs de définir des constructions spéciales nommées "aspects". Les aspects peuvent contenir plusieurs entités inutilisables par des classes standard qui sont joinpoint, pointcut, advice, introduction, weave-time declaration. On les regroupe dans deux catégories possibles:

9.1.1. Construction statique *(inter-type declaration (introduction), weave-time declaration)*

- Modification de la structure statique du système : classes et interfaces.
- Ajout d'attributs et méthodes.
- Déclaration de messages d'avertissement (warnings) ou d'erreurs affichés lors de la compilation

9.1.2. Construction dynamique *(advice)*

- Ajout d'un nouveau comportement à l'exécution "normale" du programme.
- Étendre ou remplacer une opération.
- Action effectuée avant/après l'exécution de certaines méthodes ou de certains *traitements* d'exceptions dans des classes.

9.2. L'implémentation des concepts de la POA avec AspectJ

Les concepts d'Aspectj sont les mêmes concepts mentionnés précédemment dans la POA. Nous allons voir comment implémenter ces concepts avec aspectj.

9.2.1. Point de jonction

En réalité, un point de jonction est n'importe quel point d'exécution dans un système. Parmi tous les points de jonction possibles dans un système on cite de façon non exhaustive :

- L'appel à une méthode.
- L'exécution d'une méthode.
- L'affectation de variable (lecture, écriture).

- L'appel au constructeur d'une classe.
- Une instruction conditionnelle (i.e. IF/THEN/ELSE).
- Le traitement d'une exception.
- Les boucles (i.e. FOR, WHILE, DO/WHILE).

Mais en pratique, et par souci de prévenir la dépendance de l'implémentation et la transversalité instable, le modèle de points de jonction adopté par AspectJ n'offre qu'un sous ensemble de points de jonctions possibles. Par exemple, il ne considère pas les boucles comme des points de jonctions. L'ensemble des points de jonction offert par AspectJ est résumé dans la table suivant.

Type	***Description***
call*(method_expr)*	Appel d'une méthode
execution*(method_expr)*	Exécution d'une méthode
get*(attribut_expr)*	Lecture d'un attribut Exemple : **get**(int Point.x)
set*(attribut_expr)*	Ecriture d'un attribut
handler*(exception_expr)*	Exécution d'un bloc de récupération d'une exception
initialization*(constructor_expr)*	Exécution d'un constructeur
preinitialization*(constructor_expr)*	Exécution d'un constructeur hérité dont le nom vérifie *constructor_expr*
staticinitialization*(classe_expr)*	Exécution d'un bloc de code *static* dans une classe
adviceexecution*()*	Exécution d'un code *advice*

Table 2. Récapitulatif des points de jonction AspectJ [Paw04].

9.2.2. Les coupes (Crosscut)- mot clé « pointcut »

Une coupe désigne est un ensemble de points de jonction.

La coupe est définie à l'intérieur des aspects, un aspect peut contenir une ou plusieurs coupes.

La déclaration des coupes dans AspectJ se fait par le mot-clé *pointcut*.

La syntaxe de définition est :

Pointcut <nom de la coupe> (< paramètres >) : < ensembles des points de jonction > ;

Exemple :

```
pointcut addition():call(int operations.add(int,int));
```

- ***Notion de Jokers (wildcards)***

L'AspectJ fournit un ensemble de symboles (Table 3) qui nous permettent de créer des expressions englobant plusieurs méthodes.

Alors, au lieu d'écrire des expressions (*call*, *execution*,…) pour tout point de jonction quelque soit son type (méthode, attribut,…) dans une classe ou dans un package « càd un grand nombre d'expressions pour une seule coupe », nous allons créer des coupes dont des expressions utilisant les wildcards, alors on va obtenir un nombre d'expressions plus petit qu'avant.

Wildcard	Utilisation
*	Remplace un nom (de classe, de paquetage, de méthode, d'attribut, etc..) ou simplement une partie de nom. Il peut aussi remplacer un type de retour ou un paramètre. Il signifie « n'importe quel nom » ou « n'importe quel type ». Exemple pour une expression sur une méthode : **public** * org.aspectj.*.jade.*.init*(**int**, String, *)
..	Utilisé pour omettre les paramètres des méthodes ou le chemin complet des paquetages. Exemple pour une expression sur une méthode : **public void** org..Test.active(..) Cet exemple signifie « toutes les méthodes publiques *active* quel que soient leurs paramètres, retournant void et situées dans des classes *Test* situées dans n'importe quel sous paquetage de *org* ».
	Permet de définir n'importe quel sous-type d'une classe ou d'une interface.

+	Exemple pour une expression sur une méthode : ***** **void** org.jade.test.application+.set* (..) ; Cet exemple à deux sens : si *application* est une interface alors « toutes les méthodes commençant par *set* des classes implémentant l'interface application. » si *application* est une class alors « toutes les méthodes commençant par *set* de cette classe et toutes les classes qui l'héritent. »

Table 3. Récapitulatif des « Wildcards » dans AspectJ [Paw04]

- ***Les concepts de filtrage***

- Les operateurs logiques *('&&', '||','!') ;*

- Des conditions *' if '.*

- *L'*emplacement de point de jonction dans le code ou la classe est défini par *(withincode,within).*

-La source et la cible du point de jonction sont définies par *(this, target).*

-Le flot de contrôle *(cflow, cflowbelow).*

Mot-clé	***Description***
&&	ET logique
||	OU logique
!	Négation logique
if (expression)	Condition
withincode(méthod exp)	Le point de jonction est situé dans la méthode définie comme paramètre.
within(type expr)	Le point de jonction est situé dans la classe ou l'interface définie comme paramètre.

this(type expr)	Vrai lorsque le type de l'objet *Source* du point de jonction vérifie *type expr*.
target(type expr)	Vrai lorsque le type de l'objet *destination* du point de jonction vérifie *type expr*.
cflow(coupe)	Tout point de jonction situé entre l'entre dans la coupe et sa sortie « avec l'entrée et la sortie »
cflowbelow(coupe)	Tout point de jonction situé entre l'entre dans la coupe et sa sortie « sauf pour l'entrée et la sortie »

Table 4. Récapitulatif des opérateurs AspectJ [Paw04].

Exemple :

```
pointcut tracing_methods () :call(* *..*.*(int) ) &&
                        execution(* *..*.*(String) &&
                        (!within(packname.classe1)||
                            !within(packname.classe2)) ;
```

tracing_methods() : est une coupe qui capture toute les appels aux méthodes qui ont un *entier* seulement en paramètre, et les exécutions des méthodes qui ont un *String* seulement comme paramètre, à condition que, ces méthodes ne se trouvent pas dans la classe *classe1* ou dans la classe *classe2*.

9.2.3. Greffon (Advice)

L'advice c'est un code qui définit le comportement d'un Aspect. Chaque advice est attaché à une coupe et possède un type. Plusieurs codes advice peuvent être définis dans un aspect.

- ***Les blocs de code advice***

Toutes les instructions autorisées dans java peuvent être utilisées dans le code advice Appel des méthodes, affectation instanciation (new), boucle (for, while, do...while), test (if) Gestion d'exception (try/catch).etc...

En plus Aspectj fournit deux mots clés supplémentaires *proceed* et *thisjoinpoint.* Le mot-clé *Thisjointpoint* est utilisable pour tout type d'advice. Le mot-clé *proceed* est utilisable uniquement pour les advice de type *around.*

- ***Les types de code advice***

AspectJ fournit cinq types de code advice. Les trois types principaux sont : *Before, after, around.* Les deux autres sont *after returning, after throwing,* ces deux sont des raffinements de type *after.*

Type	Syntaxe

before	`before() : < nom du point de jonction>() {`
	`// comportement du greffon`
	`}`

after	`after() : < nom du point de jonction>() {`
	`// comportement du greffon`
	`}`

after returning	`after() returning (type de retour d): < nom du point de jonction>(){`
	`System.out.println("la valeur de retour est : "+d ) ;`
	`//……`
	`}`

after throwing	`after() throwing (Exception e) : < nom du point de jonction>(){`
	`System.out.println("l'exception levée est : "+e ) ;`
	`//……`
	`}`

around	`Object around() : < nom du point de jonction>{`
	`// avant l'exécution du point de jonction`
	`// emplacement des pré-conditions`
	
	`Object ret = proceed() ;`
	
	`// après l'exécution du point de jonction`
	`// emplacement post-conditions`
	
	`Return ret ;`
	`}`

Table 5. Implémentation des greffons en AspectJ.

9.2.4. Le mécanisme d'introduction

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

La syntaxe d'une déclaration inter-type est la suivante :

<Méthode d'accès> <type> <nom de la class>.<nom de variable> ;

Exemple d'une variable : `public int classe1.x;`

Exemple d'une fonction : `public int classe1.getx(){return x;}`

9.2.5. Aspect

La création d'aspect est similaire à celle d'une classe

```
public aspect Trace1 {

}
```

Elle peut contenir toutes les notions du langage java, plus les concepts de la POA.

Propriétés	***Description***

Héritage	Comme dans la notion de classe, l'aspect peut être hérité.
	`public aspect Trace1 extends Object{`
	`}`

Instanciation	Comme dans la notion de classe, l'aspect peut être instancie aussi
	Mot-clé disponible sont : perthis, pertarget, percflow, percflowbelow
	`aspect aspect1 perthis(coupe){}`
	`aspect aspect2 pertarget(coupe){}`
	`aspect aspect3 percflow(coupe){}`
	`aspect aspect4 percflowbelow(coupe){}`

Ordonnancement	Lorsque plusieurs aspects se tissent sur le même point de jonction alors il faut déterminer l'ordre d'exécution des ces codes Advice. AspectJ fournit deux mécanismes pour ordonner les aspects :

	10. Explicite : c'est au programmeur de définir l'ordre des aspects en utilisant des mots-clés prédéfinis. 11. Implicite : il existe quelque règles pour déterminer l'ordre. `declare precedence: Trace2,Trace1;`

Aspect privilégié	Un aspect privilégié a la possibilité d'accéder à tous les attributs et les méthodes quelques soient leurs visibilités (privates, *protected*, *public*).
	`privileged aspect Trace1 {`
	`}`

Table 6. Propriétés de l'aspect dans AspectJ.

10. Conclusion

Dans ce chapitre nous avons présenté un nouveau paradigme de programmation relativement récent appelée *Programmation Orientée Aspect*. Ce paradigme n'est pas lié à un langage de programmation particulier mais peut être mise en œuvre aussi bien avec un langage orienté objet comme Python qu'avec un langage impératif comme le C. Le seul pré-requis étant l'existence d'un *tisseur d'aspect* pour le langage cible. Le but de la POA est de compléter La POO pour avoir des programmes plus compréhensible, plus facile à maintenir, etc. par la séparation des exigences fonctionnelles et non-fonctionnelles. Parmi les outils de la POA on trouve l'AspectJ qu'est considéré comme le meilleur à cause des avantages qu'il procure.

Chapitre 03

Les techniques d'extraction d'Aspects

1. Introduction

Les tendances actuelles en matière de développement des systèmes distribués sont de plus en plus orientées vers des applications logicielles en termes de systèmes multi-agents. Ces applications sont intrinsèquement complexes et leur développement est encore techniquement difficile. Pour implémenter ces applications, les développeurs réutilisent souvent des plateformes multi-agents ne tenant pas compte de la qualité du code, ce qui rend leurs processus de compréhension et de maintenance difficiles à maîtriser. Une des principales causes de cette difficulté est la fusion des préoccupations non-fonctionnelles avec celles fonctionnelles dans le même code source.

La *programmation orientée aspect* apporte une solution élégante et simple à ce problème. Cette nouvelle méthode de programmation permet d'implémenter chaque problématique (préoccupation transverse) indépendamment des autres, puis de les assembler selon des règles bien définies. La programmation orientée aspect promet donc une meilleure modularité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements.

Afin d'appliquer les techniques orientées aspect aux anciennes applications Multi-Agent en usage (i.e. Migration vers l'approche orienté aspect) on a besoin de passer par deux étapes majeurs la première est la découverte (identification (semi) automatique) de préoccupations transverses dispersées dans le code qui peuvent être transformées en aspects (aspects candidats). Cette technique est connue sous le nom *Extraction d'aspects* (*Aspect Mining*). La deuxième étape est la restructuration du système *(Aspect refactoring)* par la transformation des aspects candidats identifiées dans la première étape vers des aspects réels dans le code tout en gardant le même comportement général. Nous nous intéressons dans le cadre de notre projet à la découverte (identification semi-automatique) d'aspects dans de codes sources d'applications multi-agent.

2. Extraction d'Aspects

L'implémentation de certaines préoccupations peut entrainer la duplication de code, la dispersion de préoccupations dans le système ainsi que l'enchevêtrement du code d'une préoccupation spécifique avec celle d'autres préoccupations, ce qui rend difficile à comprendre, maintenir et évoluer le système.

À l'aide de la technologie orientée aspect, ces préoccupations transversales peuvent être modulaire à l'aide de fonctionnalités de langage comme les coupures et les aspects [Lad03]. Dans le système résultant, les différentes préoccupations sont proprement séparées ce qui rende le système plus facile à maintenir et à étendre.

Pour migrer des systèmes existants vers des systèmes orientée aspects (figure 1) il faut utiliser des explorateurs avancés qui peuvent aider les développeurs à l'identification des préoccupations transverses dans les systèmes en usage (existants) ou des outils plus automatisé à la découverte de ces préoccupations, ainsi que la restructuration des préoccupations transversales identifiées en aspects.

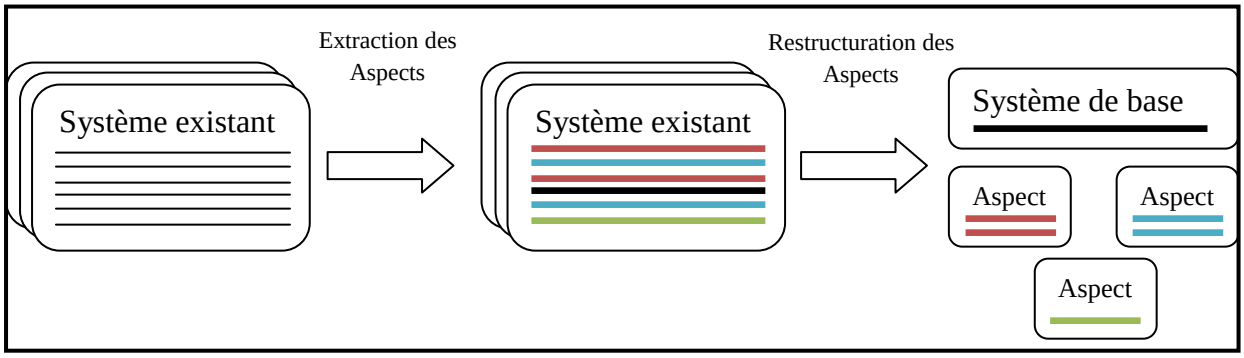


Figure 1. Migration d'un système existant vers un système orientée aspect.

L'extraction des aspects peut être définie comme étant l'activité de la découverte des préoccupations transversales dans le code source d'un logiciel donné, qui potentiellement pourraient se transformer en aspects.

Plusieurs approches ont été proposées et des outils ont été développés dont le but et d'aider les développeurs à détecter les aspects de manière automatique ou semi-automatique. On peut distinguer deux grandes catégories :

3. Les explorateurs dédiés

Les explorateurs dédiés nécessitent un point de départ appelé graine « seed » d'une préoccupation pour identifier manuellement ces préoccupations transversales en découvrant le système. Les Explorateurs dédiés peuvent avoir un langage de requête pour aider les développeurs dans la recherche des préoccupations transverses [Kel05].

L'avantage de ces explorateurs est que les développeurs peuvent identifier exactement les préoccupations qu'ils désirent, exactement autant de détails qu'ils ont besoin. L'inconvénient,

bien sûr, c'est qu'une grande partie de la charge cognitive est placée sur le développeur. L'outil joue le rôle d'un enregistreur plus qu'un assistant et les développeurs ont besoin d'une graine d'une préoccupation pour rechercher manuellement ces préoccupations transversales dans un système [Hon05].

Il existe plusieurs exemples d'explorateurs dédiés comme :

3.1. FEAT (Feature Exploration and Analysis Tool)

FEAT (figure 2) est un outil développé comme un plugin dans la plate-forme Eclipse [Rob02]. Il représente les préoccupations sous forme d'arbre dans le graphe de préoccupations. Un graph de préoccupations sert à sauvegarder un ensemble de préoccupations relatives à une tâche particulière. FEAT permet aux développeurs de rechercher, de naviguer, de comprendre et d'analyser le code qui implémente une préoccupation dans un système Java. Par la navigation visuelle dans les dépendances structurelles de programme, le développeur peut déterminer le code qui implémente une préoccupation et enregistrer le résultat comme une représentation abstraite composée de blocs de construction qui sont faciles à manipuler et à interroger. La représentation d'une préoccupation supportée par FEAT permet d'explorer les relations entre la préoccupation capturée et le code de base et entre les différentes parties de la préoccupation elle-même.

Avantages

FEAT a un certain nombre d'avantages comme:

- Le développeur peut rapidement déterminer et analyser les préoccupations dispersées dans un code.
- Les Graphes de préoccupations pourraient être étendus aux langages de programmation supplémentaires, y compris les langages procéduraux tels que C.
- Le principal avantage des graphes de préoccupation consiste à les utiliser pour sauvegarder les informations lorsque nous explorons les différentes préoccupations importantes dans un programme.
- Le concept clé de la comparaison de deux préoccupations est d'observer comment ils sont liées sans avoir à comprendre les préoccupations en totalité.

Inconvénients

FEAT a aussi un certain nombre d'inconvénients comme :

- Le développeur implémente les relations définies et les requêtes comme statiques par l'utilisation de FEAT.
- Le développeur ne peut pas ajouter de nouvelles requêtes afin d'explorer de nouveaux types de relations de fonctionnalité.
- Le développeur a besoin d'un point de départ des préoccupations pour démarrer l'analyse du code.
- Le développeur doit être bien familiarisé avec la plateforme Eclipse.

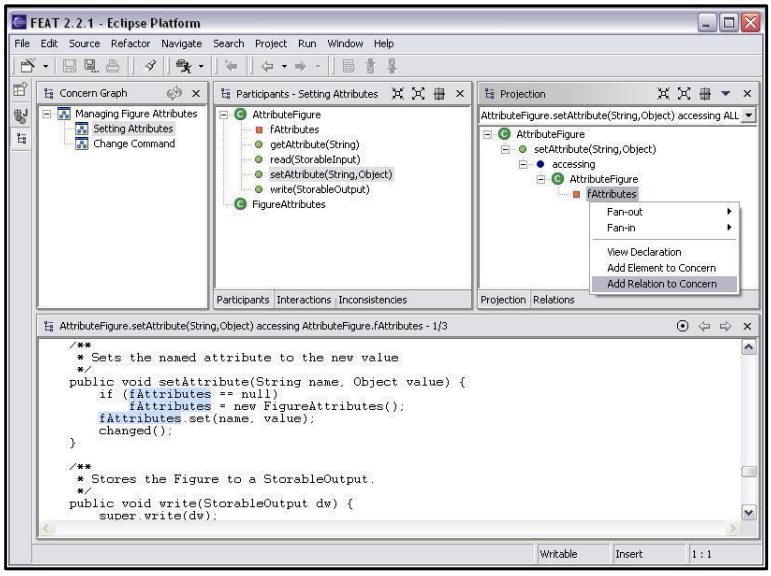


Figure 2. FEAT.

3.2. Aspect Browser

Développé en tant que plugin dans la plateforme Eclipse, **Aspect Browser** (figure 3) permet aux développeurs de visualiser les programmes dans une vue de type Seesoft-like [Eic92] par la recherche d'expressions régulières et l'affichage des résultats sous forme graphique. En outre, Aspect Browser inclut des fonctionnalités pour naviguer dans les résultats de la recherche et de gérer un ensemble potentiellement important d'expressions régulières [4].

Le but d'Aspect browser est d'aider les développeurs à afficher, explorer et gérer les préoccupations transversales. Ainsi, tous les fichiers dans un programme sont affichés comme une rangée de petites fenêtres dans lequel chaque ligne de code dans un fichier correspond à une ligne de pixels dans une fenêtre. Chaque occurrence d'une coupe transversale est mise en surbrillance dans une fenêtre avec une couleur spécifique, comme les symboles sur une carte.

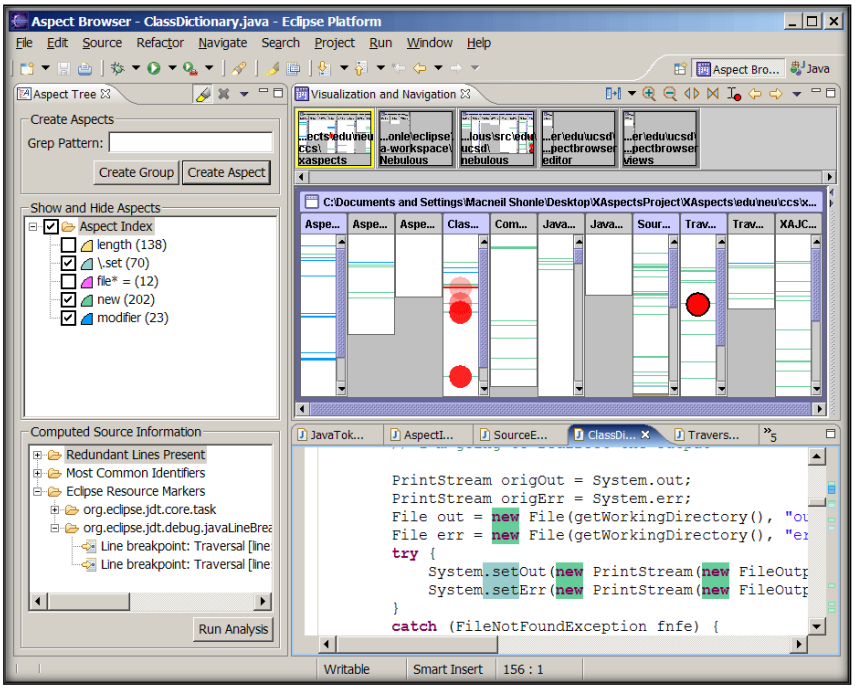


Figure 3. Aspect Browser.

Avantages

- Outil graphique qui aide les développeurs à trouver et à gérer les aspects.
- Il fournit au développeur une compréhension rapide sur comment une préoccupation transversale est dispersée à travers les fichiers.
- Il possède des fonctionnalités qui facilitent au développeur de trouver les représentants possibles de recoupement des préoccupations telles que l'identification des lignes redondantes du code.

Inconvénients

- C'est impossible d'afficher beaucoup d'aspects à la fois en raison du nombre impressionnant de couleurs. Aussi, sur un projet plus vaste, le nombre d'aspects augmentera, et une approche pour organiser les aspects sera essentielle.
- Il recherche seulement les pattern-textuelles; il ne fait pas de distinction entre un nom de package, un nom de type, un nom de variable, un nom de méthode ou un commentaire de code.
- Le développeur a besoin d'un point de départ de préoccupations pour démarrer l'analyse du code.
- Le développeur a besoin de beaucoup de temps pour analyser et filtrer les résultats.

3.3. Aspect Mining Tool

Aspect Mining Tool (AMT), développé par Jan Hannemann, offre un Framework d'analyse multimodale ouverte pour l'identification des préoccupations et la compréhension de système. AMT propose deux techniques d'analyse pour rechercher les aspects candidats [5] :

- Analyse lexicale (basé-texte): il offre un « pattern matching » simple comme aspect browser.
- Analyse basée-type : Avec l'analyse basé-type, l'enchevêtrement de code peut être détecté, et les mesures de la qualité de la modularité comme la cohérence et le couplage du code peuvent être visualisé.

Aspect Mining Tool se compose de deux programmes indépendants [5] :

- L'analyseur extrait toutes les statistiques orientées ligne nécessaires de programme (source code et les types utilisés) et des informations structurelles (les packages et les informations d'hiérarchie des classes). Toutes les informations extraites sont écrites dans un fichier de données.
- L'afficheur utilise le fichier de données pour afficher une vue (view) basés-lignes du système (par exemple, les unités de compilation sous forme d'ensembles de lignes de code). Les développeurs peuvent ensuite interroger la base de données de système (créée par le visualiser du fichier de données) interactivement.

Avantages

AMT a un certain nombre d'avantages comme :

• Il fournit un cadre d'analyse multimodale ouverte pour l''identification de préoccupation et la compréhension du système.

• L'analyse basée-type fonctionne très bien avec les objets et les variables.

Inconvénients

AMT présente quelques inconvénients comme :

• Il ne fonctionne que si les conventions de nommage pour les types, les méthodes, les variables et les classes sont respectées.

• L'analyse basée-type ne fonctionne pas avec les appels de méthodes. L'outil ne découvre pas les signatures des appels de méthode; ils doivent être détectés avec des recherches textuelles.

• Ce n'est pas possible de construire une structure de données de préoccupations afin de stocker les différents résultats d'une requête.

3.4. Comparaison entre les explorateurs dédiés

Les deux tables 1 et 2 montrent certaines fonctions de recherche pour les explorateurs dédiés discuté ci-dessus.

	Fonctionnalités de recherche			
	Analyse textuelle	Analyse basée sur le type	Appel aux méthodes	Joker utilisés
FEAT	n/a	n/a	n/a	n/a
Aspect Browser	Oui	No	No	"*"
AMT	Oui	Oui	No	rien

Table 1. Comparaison des explorateurs dédiés (partie 1).

	Fonctionnalités de recherche	Constructions de caractérisation valide	Analyse complémentaire atteinte
FEAT	- Constructions de Java - Relations	- Constructions de Java - Relations	Comparer entre deux préoccupations
Aspect Browser	n/a	Analyse textuelle	- Comptage de match - Lignes redondantes du code
AMT	n/a	- Analyse textuelle - Analyse basée sur le type	Non

Table 2. Comparaison des explorateurs dédiés (partie 2).

- **n/a:** non autorisé.
- **Constructions de Java** : type, méthode et attribut.
- **Relations:** déclare, déclaré par, appels, appelé par, etc.

4. Techniques automatisées d'extraction d'aspect (Automated aspect mining techniques)

Nous savons que les explorateurs dédiés ont besoin des graines de préoccupation pour rechercher manuellement par navigation des aspects candidat dans le code [Kel05]. Nous pouvons utiliser des techniques plus automatisées pour aider les développeurs à déterminer les points de départ ou les graines automatiquement. L'avantage de cette approche est qu'aucun entrée ou requête n'est requise afin d'identifier les préoccupations. Toutefois, l'inconvénient

est que seulement les préoccupations très fréquentes sont susceptibles d'être trouvées et le code qui implémente un problème donné est susceptible d'être manqué [Hon05].

Plusieurs techniques d'extraction d'aspects (automatiques ou semi-automatiques) ont été proposées dans la littérature (table 3).

Nom	Description
Patron d'exécution	Analyse des patrons récurrents présents dans les traces d'exécution
Analyse dynamique	Analyse des concepts formels des traces d'exécution
Analyse des identifiants	Analyse des concepts formels des noms de classes et des méthodes
Indice du langage	Processus d'analyse de langue naturelle appliquée au code
Méthode unique	Détection des méthodes uniques
Grappe (Clustring)	Identification par regroupement des méthodes qui sont reliées
Analyse de renvoi	Analyse des appels de renvoi entrant (fan-in)
Détection de clone	Détection de clone pour identifier les préoccupations transversales

Table 3. Liste des approches (semi) automatiques d'extraction des aspects.

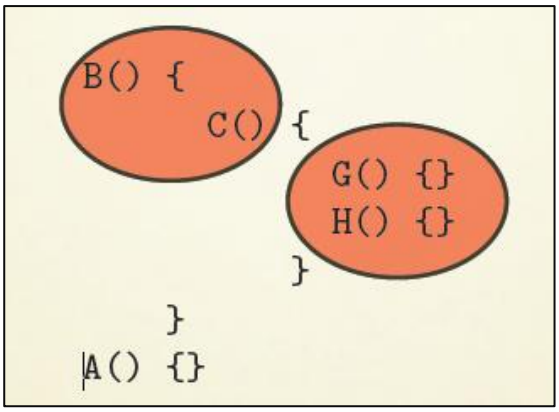
4.1. Analyse des patrons récurrents de traces d'exécution

Breu et krinkle [Bre04a] ont proposé une technique d'extraction des aspects nommée DynAMiT (Dynamic Aspect Mining Tool), qui analyse les traces d'exécution d'un programme en cherchant des patrons d'exécution récurrente. Ils ont introduit la notion de relations d'exécution entre l'appel des méthodes.

Prenons l'exemple suivant d'une trace d'évènement ou les lettres en majuscules représentent les noms des méthodes

Les auteurs distinguent entre 4 relations d'exécutions différentes:

1. **outside-before :** l'avant en extérieur (B est appelé avant A).
2. **outside-after** : l'après en extérieur (A est appelée après B).
3. **inside-first** : le premier en intérieur (G est le premier appel en C).
4. **inside-last** : le dernier en intérieur (H est le dernier appel en C).



Grâce à ces relations d'exécution, leur algorithme d'exploration identifie les aspects candidats basés sur des modèles récurrents d'invocations de méthodes. Si une relation d'exécution se produit plus d'une fois, et qui revient régulièrement (par exemple chaque invocation de la méthode B est suivie d'une invocation de méthode A), il est considéré comme un aspect candidat. Bien sûr, afin de s'assurer que les candidats sont suffisamment des préoccupations transversales, il y a une exigence supplémentaire que les relations récurrentes devraient apparaître dans différents contextes d'appel.

Bien que cette approche soit naturellement dynamique, les auteurs ont répété l'expérience à l'aide de graphes de flux de contrôle [Kri04], une technique statique, pour calculer les relations d'exécution.

Breu [Bre04b] rapporte également sur une approche hybride où l'information dynamique est complétée par des informations de type statique afin de lever les ambiguïtés et d'améliorer les résultats de la technique.

4.2. Analyse des concepts formels

La technique FCA (Formal Concept Analysis) [Gan99] d'analyse des concepts formels est assez simple. À partir d'un ensemble (potentiellement important) d'objets et de leurs attributs, FCA détermine les groupes maximaux des objets et des attributs. Ces groupes maximaux sont appelés des concepts. Chaque concept se compose d'un ensemble d'objets ayant un ou plusieurs attributs en commun [Tou04a].

a. Analyse des concepts formels de traces d'exécution

Tonella et Ceccato [Ton04] ont développé Dynamo, une technique d'exploration (extraction) qui applique l'analyse des concepts formels pour les traces d'exécution afin d'identifier les aspects possibles.

Lors de l'analyse d'un système utilisant Dynamo, une version instrumentée du système est utilisée pour exécuter un certain nombre de cas d'utilisation. La sortie de cette exécution est un nombre de traces d'exécution. Ces traces sont ensuite analysées en utilisant l'algorithme de FCA : où les cas d'utilisation sont les objets de l'algorithme et les méthodes qui sont invoqués lors de l'exécution d'un cas d'utilisation sont les attributs. Résultant des concepts qui sont spécifiques à un cas d'utilisation particulier, et qui sont des aspects candidats si les deux contraintes suivantes sont vérifiées:

- Dispersion: Les attributs (méthodes) du concept appartiennent à plus d'une classe.
- Enchevêtrement: Différentes méthodes d'une même classe sont contenues par plus d'un concept spécifique de cas d'utilisation (classe utilisée dans plusieurs cas d'utilisation)

b. Analyse des concepts formels des noms des classes et noms des méthodes

Tourwé et Mens [Tou04b] ont proposé une autre technique d'extraction d'aspect qui repose sur l'analyse des concepts formels. Contrairement à l'approche Dynamo que nous avons mentionné précédemment, l'outil DelfSTof de Tourwé et Mens analyse le code source d'un système (expériences ont été menées sur Smalltalk [Tou04b] et le code Java [Cec05]). Leur approche effectue une analyse d'identifiant (noms des classes et des méthodes) à l'aide de l'algorithme de FCA . L'hypothèse derrière cette approche est que les préoccupations intéressantes dans le code source sont reflétées par l'utilisation des conventions de nommage dans les classes et les méthodes du système. Comme entrées pour l'algorithme FCA, les classes et les méthodes du système sont utilisées comme des objets. Comme des attributs, l'algorithme de CAF prend comme entrée des sous-chaînes générées par les entités du programme utilisées comme des objets.

Par exemple, une classe nommée QuotedCodeConstant est subdivisée à «Quoted » «Code» et «Constant». Sous-chaînes avec peu de sens, comme 'a', 'avec', . . . sont rejetés à partir des résultats.

Les concepts résultants se composent de groupes maximaux d'entités de programme qui partagent un nombre maximal de sous-chaînes. Après filtration des concepts sans importance, un grand nombre de concepts demeurent qui doivent être inspectés manuellement. En plus d'être capable de détecter un certain nombre d'idiomes de programmation, design patterns et certaines opportunités de refactorisation [Men05], en limitant les concepts à ceux qui sont transversales (c'est à dire les méthodes et les classes concernées

appartiennent à au moins deux des hiérarchies de classes différentes) la même approche peut être utilisée pour l'extraction d'aspect [Tou04b].

4.3. Traitement du langage naturel sur le code source

Similaire à la technique précédente, Shepherd et al. [She05a], ont proposé une technique basée sur l'hypothèse que des préoccupations transversales sont souvent implémentées par les conventions de nommage et de codage.

Leur approche utilise le traitement du langage naturel (NLP) des informations comme un indicateur pour les aspects candidats possibles. Ils rendent compte d'une expérience dans laquelle ils utilisent une technique NLP appelé chaînage lexicale [Mor91] dans le but de trouver des groupes de code source liés à des entités qui représentent une préoccupation transversale. Le chaînage lexical prend en entrée un ensemble de mots et donne en sortie des chaînes des mots qui sont étroitement liés. Afin de créer la chaîne, l'algorithme utilise une mesure de distance sémantique entre deux mots. Ces auteurs ont utilisé WordNet [Bud01] qui est un catalogue de sillons sémantiques entre les mots, pour utiliser cette mesure en combinaison avec des informations sur la partie du discours de chaque mot. Afin d'extraire des préoccupations transversales, ils ont appliqué l'algorithme de chaînage sur les commentaires, les noms des méthodes, noms de champs et les noms de classes du système qu'ils analysent. Afin d'identifier les aspects candidats, l'utilisateur de leur approche doit inspecter manuellement les chaînes qui en résultent.

4.4. Détection des méthodes uniques

Gybels et Kellens [Gyb04, Gyb05] ont proposé l'utilisation des heuristiques pour identifier d'éventuelles préoccupations transversales. Ils ont observé qu'il y avait des préoccupations transversales ont été souvent implémentées de façon idiomatique. Certains de ces idiomes peuvent être considérés comme des «symptômes» des aspects candidats. Un exemple d'idiomes est l'implémentation d'une préoccupation transversale à l'aide d'une seule entité dans le système qui est appelée à partir de nombreux endroits dans le code (par exemple, « le Traçage » entité qui est appelée à partir de n'importe où dans le code).

Pour détecter les cas de ce modèle, les auteurs ont proposé une heuristique nommée "Méthodes uniques" définie comme suit: « une méthode sans valeur de retour qui implémente

un message qui n'est pas implémenté par d'autres méthodes ». Après avoir calculé toutes les méthodes uniques dans un système, il faut les trier en fonction du nombre de fois qu'une méthode est appelée et d'ignorer les méthodes non-importantes (comme par exemple les méthodes get et set), l'utilisateur doit inspecter manuellement les méthodes résultants pour trouver des aspects candidats adéquats.

Quelle que soit la simplicité de cette approche, les auteurs ont démontré l'applicabilité de leur technique en détectant des aspects typiques comme le traçage (figure 04), la notification de mise à jour et la gestion de la mémoire dans le contexte d'une application Smalltalk.

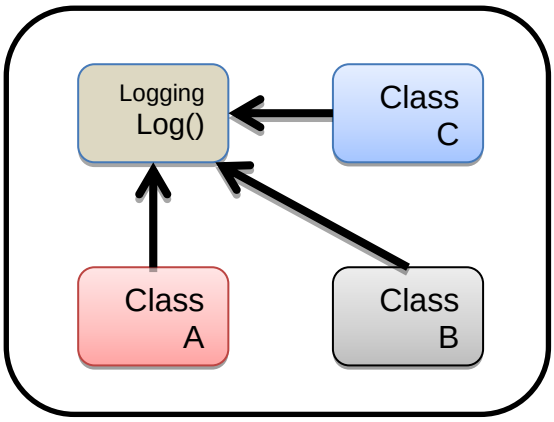


Figure 4. La méthode unique Logging.log().

4.5. Classification hiérarchique des méthodes reliées

Shepherd et Pollock [She05b] ont rapporté sur une expérience dans laquelle ils ont utilisé la classification hiérarchique ascendante [Kar98] pour un groupe de méthodes reliées. Cette technique met chaque méthode dans un groupe séparé et fusionne récursivement ces groupes pour lesquelles la distance entre les méthodes est inférieure à un certain seuil. Ils ont implémentés cette technique comme une partie dans un IDE orienté aspect nommé (Aspect Miner and Viewer) ce qui permet d'adapter facilement la mesure de distance utilisée par l'algorithme. Pour une première expérience, ils ont utilisé une mesure de distance simple opposée proportionnellement à la longueur de la sous-chaîne commune des noms des méthodes.

Cet algorithme d'extraction est utilisé en combinaison avec l'outil de visualisation de l'IDE qui ne liste pas seulement tous les clusters (groupes) qui ont été trouvés, mais qui

consiste également en une section des préoccupations transversales qui affiche les méthodes associées à un cluster et une section de l'éditeur, dans laquelle le contexte de la classe d'une méthode particulière est affichée.

He et Bai [He04] ont proposé une autre technique d'extraction d'aspect basée sur l'analyse des clusters. Ils partent de l'hypothèse que si les méthodes apparaissent ensemble dans un certain nombre de modules différents, cela peut être une bonne indication qu'une préoccupation transversale est présente. En entrée de l'algorithme de classification, un ensemble de méthodes est proposé avec une mesure de distance sur la base de SDIR (Static Direct Invocation Relationship), la Relation d'Invocation Statique Directe, entre les méthodes. Cette mesure de distance est une représentation de la dissemblance des méthodes, en testant si une méthode fait appel à une autre méthode dans un autre contexte d'appel différente ».

4.6. Analyse de renvoi

Marin et al. [Mar04] ont remarqué que les préoccupations transversales bien connues sont implémentées en utilisant une technique qui présente un grand renvoi. Ils proposent d'utiliser une métrique de renvoi dans le système pour découvrir des préoccupations transversales dans le code source. Ils définissent le renvoi d'une méthode *m* comme le nombre de corps des méthodes distinctes qui peuvent appeler *m*. En raison de polymorphisme, un appel à une méthode *m* contribue au renvoi de toutes les méthodes de raffinage de *m* ainsi que toutes les méthodes qui raffinent *m*. Leur algorithme d'extraction comprend:

- Calcul de la métrique du renvoi pour toutes les méthodes dans le système.
- Filtrage des résultats: filtrage des méthodes d'accès (get*()) et de mutation (set*()), ainsi les méthodes utilitaires comme toString (), le nombre de méthodes considérés est également limité en ne considérant que les méthodes avec un renvoi en valeur supérieure à un certain seuil.
- Analyse manuelle des méthodes restantes

Les auteurs présentent une expérience dans laquelle des préoccupations transversales ont été extraites avec une grande précision: un tiers de toutes les méthodes à forte renvoi étaient les graines (seed) conduisant à un aspect. De plus, 60% des deux tiers restants ont été supprimés automatiquement.

4.7. Détection des Clones

Le symptôme de « duplication de code » peut être un bon indicateur d'une préoccupation transversale dans le code source d'un système parce que les préoccupations transversales ne pourraient pas être proprement modulaires, certaines parties de l'implémentation montrent des niveaux élevés de code dupliqué. Il y a deux techniques qui s'appuient sur cette observation à l'extraction des aspects candidats.

a. Détection des aspects par la détection de clone basée PDG

Une première technique, présentée par Shepherd et al. [She04] est implémentée comme un outil appelé *Ophir*, utilise des Graphes de Dépendance du Programme (GDP) pour détecter les aspects possibles.

Dans un GDP, chaque instruction dans le code est représentée par un nœud ; les arêtes du graphe se composent des relations de dépendance du contrôle ou de données entre les instructions.

En comparant les GDPs [Kom01, Kri01], cette technique est capable de reconnaître la duplication de code au début d'une méthode (c.-à-d. les aspects candidats pour un griffon « before »). Après le filtrage et la coalescence de la GDP qui en résulte, il reste un certain nombre d'aspects candidats possibles.

b. Détection de clone basée sur l'AST (Abstract Syntax Tree) et le jeton (Token)

Bruntink et al. [Bru04a, Bru05] ont aussi utilisé des techniques de détection de clone pour l'exploration des aspects. Ils comparent la détection de clone basée jeton [Bak95], qui repose sur une analyse lexicale du code source, avec la détection de clone [Bax98] basée AST, qui prend en compte l'arbre d'analyse du code source. Les deux techniques offrent en sortie plusieurs classes de clone, c'est-à-dire des groupes de fragments de code qui sont considérés des clones l'un de l'autre. Les auteurs ont appliqué les techniques de détection de clone à un grand programme en C où les différentes préoccupations transversales ont été annotées par un développeur. Afin de mesurer l'efficacité des techniques, ils ont comparé empiriquement les classes résultantes de clone avec la documentation manuelle des préoccupations transversales. Dans [Bru04b], une amélioration a été apportée à ce travail , dans lequel un certain nombre de paramètres pour les classes de clones a été décrit pour filtrer les résultats.

5. Comparaison

5.1. Critères de comparaison

Nous allons voir quelques comparaisons entre les approches (semi) automatiques existantes en se basant sur les critères suivantes définis par Kellens, Mens et Tonella [Kel05]:

a. **Analyse textuel ou structurel / comportemental**: quel est le type d'analyse faite par la technique, basant sur l'analyse lexicale de code, les expressions régulières, chaines de caractères, etc. Ou bien sur l'analyse structurel ou comportemental, l'arbre d'analyse, envoi de messages.
b. **Données Statiques ou dynamiques :** Est-ce que les données en entrée de la technique sont obtenues statiquement par l'analyse de code source ou dynamiquement par son exécution ou les deux.
c. **Dispersion ou enchevêtrement :** est-ce que la technique cherche des symptômes de dispersion ou d'enchevêtrement ou les deux.
d. **Granularité :** Quel est le niveau de granularité des aspects candidats explorés? Alors que certaines techniques fonctionnent au niveau des méthodes, d'autres prennent en compte des expressions individuelles ou des fragments de code.

e. **Validation empirique :** à quel degré les approches existantes sont validées quantitativement sur des cas réels, est ce qu'elles ont rapporté combien d'aspects ont été trouvés et combien d'entre eux sont des faux positives (des codes détectés comme aspects, mais réellement ne sont pas des préoccupations transversales)
f. **Taille de système :** quel est la taille la plus large des systèmes analysés par l'approche.
g. **Pré-conditions :** quels sont les pré-conditions qu'ils doivent être satisfaites par une préoccupation pour la classifier comme aspect.
h. **Participation de l'utilisateur :** quelles sont les actions faites par l'utilisateur, l'effort nécessaire par lui, est ce qu'il doit participer au début de l'analyse (entrée) ou au niveau des résultats

La table 4 montre que la plus part des techniques vues fonctionnent sur des données provenant d'un analyse statique de code, les deux approches qui nécessitent des traces

d'exécution sont les seules approches dynamiques où le patron d'exécution possède une version statique basée sur le graph de flux de contrôle ainsi une version hybride. Pour le type d'analyse, quatre techniques sont basées sur l'analyse lexicale où ils supposent que les préoccupations transversales sont implémentées par des conventions de nommage, les autres techniques sont soient structurelle soeint comportementales.

Nom	Type de donnée d'entrée		Type d'analyse	
	Statique	Dynamique	Lexicale (Token)	Structurel/Comport-emental
Patron d'exécution	X	X	-	X
Analyse dynamique	-	X	-	X
Analyse des identifiants	X	-	X	-
Indice du langage	X	-	X	-
Méthode unique	X	-	-	X
Grappe	X	-	X	-
Analyse de renvoi entrant	X	-	-	X
Détection de clone (PDG-AST)	X	-	-	X
Détection de clone(Token)	X	-	X	-

Table 4. Types de données et d'analyse.

La table 5 présente les techniques qui cherchent les symptômes de dispersion et ceux d'enchevêtrement au niveau de granularité des méthodes ou des fragments de code. La plus part des techniques cherchent sur aspects qui sont des méthodes dispersées dans le code source, sauf la détection de clone qui cherche la dispersion des fragments (expressions); différemment à l'analyse dynamique qui peut détecter des aspects par les deux symptômes au même temps.

Nom	Granularité		Symptômes	
	Méthode	Fragment de code	dispersion	enchevêtrement
Patron d'exécution	X	-	X	-
Analyse dynamique	X	-	X	X
Analyse des identifiants	X	-	X	-
Indice du langage	X	-	X	-
Méthode unique	X	-	X	-
Grappe	X	-	X	-
Analyse de renvoi	X	-	X	-
Détection de clone (PDG-AST-Token)	-	X	X	-

Table 5. Granularité et symptômes cherchés.

La table 6 montre la taille de plus grand programme analysé par chaque technique et si cette analyse est empiriquement validée ou non (combien des aspects pré-connus sont réellement rapportées et combien des faux positives et négatives sont rapportées). Toutes les technique sont appliquées sur des systèmes larges sauf la technique de grappe des appels où elle a analysé un petit code d'un jeu. Nous remarquons aussi que les techniques ne sont pas empiriquement validées mais sauf la détection de clone basée jeton et AST.

Nom	Validation empirique	Taille Système
Patron d'exécution	-	3100 méthodes/82KLOC
Analyse dynamique	-	2800 méthodes/18KLOC
Analyse des identifiants	-	2800 méthodes/18KLOC
Indice du langage	-	10KLOC
Méthode unique	-	3400 classes/66000 méthodes
Grappe des méthodes	-	2800 méthodes/18KLOC

Grappe des appels		12 méthodes
Analyse de renvoi	-	2800 méthodes/18KLOC, 172KLOC
Détection de clone (PDG)	-	38KLOC
Détection de clone (token/AST)	**X**	20KLOC

Table 6. Évaluation de la validation des méthodes.

La table 7 montre la participation d'utilisateur lors de l'application de chaque technique. Lorsqu'aucune technique n'est complètement automatique ils ont besoin d'un processus de navigation et filtrages dans les résultats pour avoir les aspects adéquats. D'autres techniques ont besoin de donnés en entrée fournies par l'utilisateur comme le nombre des cas d'utilisation à performer dans la méthode d'analyse dynamique.

Nom	Participation d'utilisateur
Patron d'exécution	Inspection des patrons récurrents résultants
Analyse dynamique	Sélection des cas d'utilisation et interprétation manuelle des résultats
Analyse des identifiants	Navigation sur des aspects explorés à l'aide d'intégration d'IDE
Indice du langage	Interprétation manuelle des chaînes lexicales qui en résulte
Méthode unique	Inspection des méthodes uniques; facilitée par le tri sur l'importance.
Grappe des méthodes	Navigation sur des aspects explorés à l'aide d'intégration d'IDE
Grappe d'appels	Inspection manuelle des grappes qui en résulte
Analyse de renvoi (fan-in)	Sélection des candidats de la liste des méthodes, triées sur la plus haute entrance (la plus invoquée).
Détection de clone	Navigation et interprétation manuelle des clones découverts

Table 7. Participation d'utilisateur.

La table 8 est considérée comme un résumé de la supposition de chaque technique. Nous remarquons que l'analyse des identifiants, grappe, indice de langage, détection de clone basée jeton, s'appuient sur la supposition que les développeurs utilisent des conventions de nommage lors de l'implémentation des préoccupations transversales.

Le patron d'exécution, grappe (AST, PDG) suppose que les méthodes appelées ensemble dans des différents contextes sont des aspects candidats.

L'analyse de renvoi entrant suppose que les préoccupations transversales implémentées par des méthodes qui sont appelées plusieurs fois ou par des méthodes qui font appel à des méthodes sont des aspects candidats.

Nom	Pré-conditions sur des préoccupations transversales dans le programme analysé
Patron d'exécution	Ordre des appels dans un contexte de préoccupation est toujours le même.
Analyse dynamique	Il existe au moins un cas d'utilisation qui expose la préoccupation transversale et un autre qui ne le fait pas
Analyse des identifiants	Noms des méthodes qui implémentent la préoccupation se ressemblent.
Indice du langage	Le Contexte de la préoccupation contient des mots clés qui sont synonymes des préoccupations transversales
Méthode unique	La préoccupation est implémentée exactement par une seule méthode
Grappe des méthodes	Noms des méthodes qui implémentent la préoccupation se ressemblent.
Grappe des appels	La préoccupation est implémentée par des appels aux mêmes méthodes à partir des différents modules.
Analyse de renvoi	La préoccupation est implémentée dans une méthode séparée qui est appelée un grand nombre de fois ou plusieurs méthodes qui implémentent la préoccupation font appel à la même méthode.
Détection de clone	La préoccupation est implémentée en réutilisant un certain fragment de code

Table 8. Pré-conditions pour trouver les aspects.

5.2. Taxonomie des approches

Les auteurs dans [Kel05] ont proposé une taxonomie (figure 5) qui peut guider les chercheurs dans l'évolution des techniques d'extraction des aspects. Les quatre grands rectangles distinguent entre les techniques qui fonctionnent au niveau des méthodes et au niveau des fragments ainsi que ceux qui sont de nature statique ou dynamique.

Les techniques dynamiques existantes fonctionnent totalement au niveau de granularité des méthodes, mais celles qui sont statiques fonctionnent aux deux niveaux: le niveau fragment de code et le niveau méthode. Les types d'analyse sont présentés comme des petits rectangles ronds qui font référence aux méthodes structurelles / comportementales et textuelles.

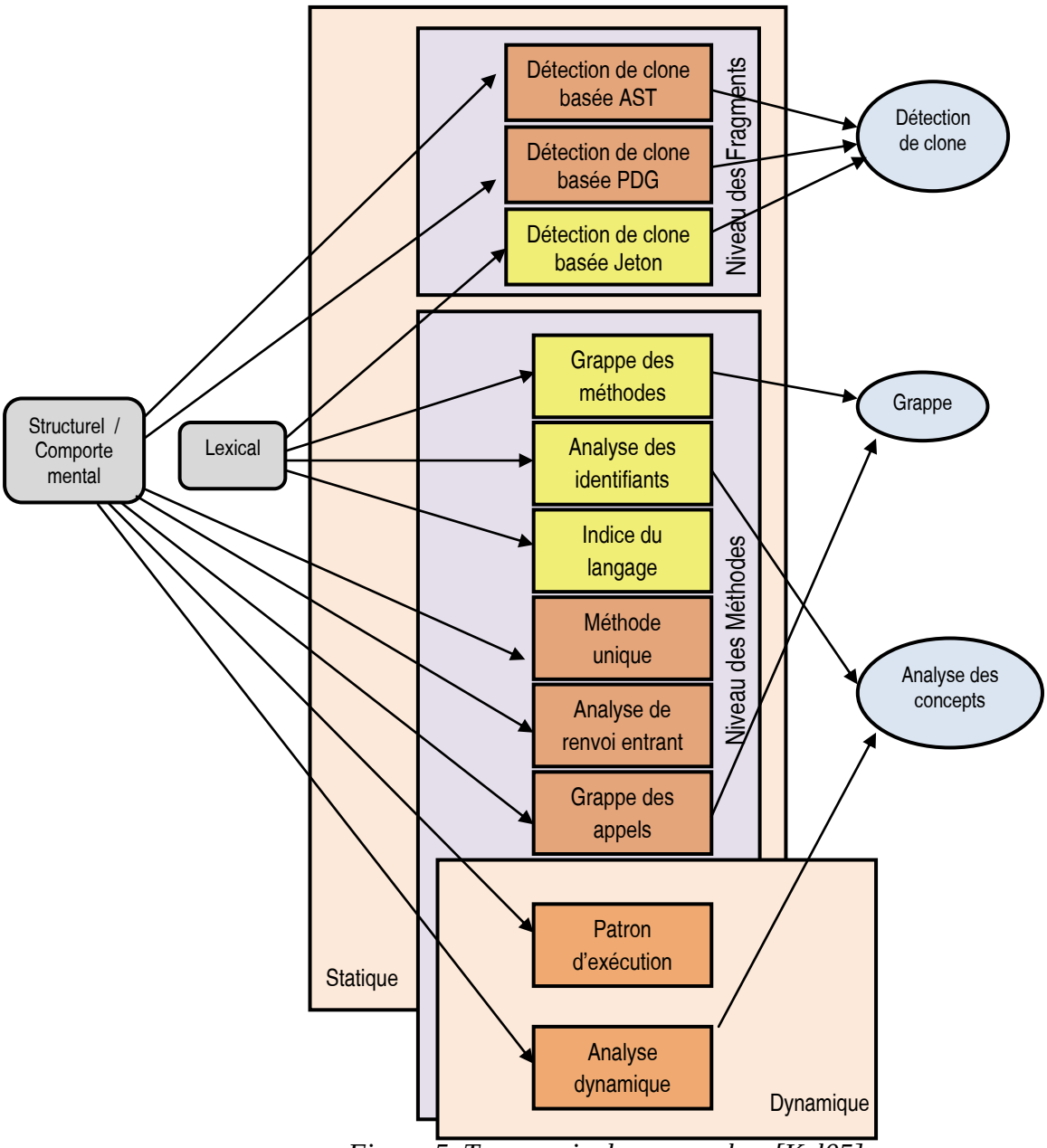


Figure 5. Taxonomie des approches [Kel05].

6. Discussion

Chaque technique parmi celles que nous avons présenté est caractérisée par le type d'analyse, le niveau de granularité, les symptômes d'existence des aspects, la validation empirique et la référence commune. Nous allons dans ce qui suit discuter ces propriétés.

6.1. Statique vs dynamique

La plus part des techniques d'extraction actuelles sont des approches statiques, parce qu'elles n'ont pas beaucoup de contraintes, contrairement à l'approche dynamique qui oblige la compilation et l'exécution de code, ainsi un environnement d'exécution approprié.

Ces informations peuvent être utiles pour les utilisateurs dans le choix de la meilleure technique.

Le problème de l'analyse dynamique est que pour les grands systèmes, les donnée fournies par les traces d'exécutions prouvent être énorme, et un bon moyen pour les réduire est par l'hybridation avec l'analyse statique.

6.2. Granularité

Une bonne granularité est assurée si la méthode en totalité implémente une préoccupation transverse donc l'emplacement où cette méthode a été invoquée peut être considérée comme un point de jonction pour l'intervention de l'aspect restructuré.

Cependant, Il existe des préoccupations transverses qui peuvent être implémentées seulement comme un modèle (pattern) dispersé dans les différentes expressions du code. Dans ce cas, pour marquer les fragments de codes qui font partie des préoccupations transverses, l'utilisateur doit fournir un effort supplémentaire lors de l'analyse des résultats ou par l'utilisation d'une technique qui fonctionne au niveau des expressions des méthodes.

6.3. Dispersion vs enchevêtrement

La plus part des techniques prennent en compte la dispersion comme étant l'indicateur (Symptôme) des préoccupations transverses « appel d'une méthode à partir de plusieurs classes» contrairement à l'analyse dynamique qui s'intéresse aussi au problème d'enchevêtrement parce qu'elle est capable de fournir des informations de haut niveau à

propos des différentes exigences dans le système en utilisant les cas d'utilisation (scenarios) où les méthodes d'une classe doivent participés dans un seul scenario.

6.4. Validation empirique

C'est une importance fondamentale pour une comparaison quantitative des techniques. Dans la plus part des études elle a été remplacée par des évaluations. Parce qu'il est difficile de définir (a priori) l'ensemble des aspects appropriés à l'extraction et de décider a posteriori quels sont les faux aspects rapportés (faux positives, faux négatives). C'est impossible à ce domaine d'extraction d'aspects de progresser sans passer par cette validation.

6.5. La référence commune

La plus part des techniques d'extraction sont représentées comme des preuves de leurs algorithmes où elles démontrent que des aspects réelles sont identifiés. Pour comparer la qualité de ces techniques dont le but est d'avoir une meilleure vue sur ses points forts et ses points faibles, il est conseillé de les valider sur un cas d'étude et des métriques communes.

7. L'extraction des aspects dans les systèmes multi agents

Les techniques d'extraction d'aspects présentés précédemment visent les préoccupations transverses des applications orientées objet comme la persistance et le traçage, nous nous intéressons aux préoccupations transverses des systèmes orienté agent, comme la mobilité, l'interaction, le matchmaking…etc. Dans la littérature il existe seulement quelques approches de séparation de ces préoccupations transverses dans les phases préliminaires de développement.

Dans [Sil06], les auteurs ont proposé une approche systématique pour spécifier les aspects de manière générique et utiliser ces spécifications dans la conception architectural des SMA. Pour cela, ils ont spécialisé un méta model d'agent pour le diagramme architectural des SMA par l'utilisation du concept *rôle.* La notation proposée a été associée à des constructions d'AspectJ et JADE. Les idées ont été illustrées à l'aide d'un système de gestion de contenu, appelé *e-News.*

Les auteurs en [Gar04a] ont présenté une méthode orientée aspect qui permet une meilleure séparation des préoccupations, en soutenant une nature des aspects systématique des propriétés de l'agent à travers la définition architecturale, la conception et l'implémentation détaillée. Un système multi-agents pour la gestion de revue du papier est considéré comme un cas d'étude.

Les auteurs de [Gar02] ont présenté leur proposition basée aspect sur une nouvelle proposition basée sur des modèles pour la création de logiciels multi-agents. Ils ont démontré l'applicabilité de deux propositions à travers le système *Portalware*, un environnement basé sur le web pour le développement de portails e-commerce.

Le travail [Gar06] présente un cadre de méta-modélisation pour soutenir la représentation modulaire des préoccupations transversales dans la conception orientée agent. Le travail est centré sur la notion d'aspects pour décrire ces préoccupations transverses. Il définit également des nouveaux opérateurs de composition pour permettre la spécification sur la façon dont les aspects influent les objectifs et les actions de l'agent. Le travail proposé est le résultat d'une expérience précédente [Gar02] de ces auteurs qui consiste en l'utilisation des techniques orientées aspect pour la conception et l'implémentation des SMAs et l'intégration des abstractions orienté aspect dans un langage de modélisation, appelé *ANote.*

Dans [Lob04], un cadre orientée aspect appelé le cadre *AspectM* a été proposé. Il prend en charge l'amélioration de la modularisation de la préoccupation de la mobilité et une intégration flexible avec plusieurs plates-formes de la mobilité. L'utilisation de leur cadre minimise la réplication de code, et augmente la réutilisabilité et la maintenabilité du problème de la mobilité et d'autres préoccupations de l'agent.

Garcia et al. [Gar05] ont introduit une approche pour séparer les préoccupations relatives à l'interaction par l'utilisation des aspects en proposant un modèle (patron) d'interaction. Ce dernier découple le comportement interactif de l'agent de l'implémentation de ses fonctionnalités de base et d'autres préoccupations spécifiques à l'agent, ce qui améliore la réutilisabilité et la maintenabilité du système. Autres préoccupations telles que la mobilité et l'apprentissage ont été prises en compte par les auteurs dans [Gar04c, Sar04].

Garcia et al. ont présenté dans [Gar04b] une étude empirique qui évalue le degré de quelques abstractions associées à deux techniques orientées objet qui permettent la modularisation des préoccupations des SMA. Les techniques sélectionnées impliquent des concepts orienté objets

tels que les classes et les objets, et autres concepts comme les modèles de conception et les aspects. Sur la base de plusieurs critères (par exemple couplage et cohésion), les résultats obtenus montrent que l'utilisation des aspects a permis la construction des SMA avec une amélioration significative de la modularisation de différentes préoccupations.

Afin de combler le vide entre les exigences orientée aspect et les approches de conception détaillée pour les SMAs, Silva et al. [Sil09] ont défini un langage aspectuel de description d'architecture (Aspectual ADL) pour les SMA comme un profil UML (Unified Modeling Language). Ce langage permet essentiellement de : (1) construire des SMAs en utilisant des composants aspectuelles de modularisation des préoccupations transversales (amélioration de la modularité), et (2) utiliser le développement dirigé par modèles pour effectuer des transformations des modèles de spécification des exigences à l'implémentation de code (amélioration de la traçabilité).

Kulesza et al. [Kul04] ont proposé une approche qui facilité le développement de systèmes multi-agents. Les motivations essentielles de l'approche proposée sont les suivants: (1) soutenir les préoccupations fonctionnelles et non fonctionnelles des agents logiciels depuis les étapes préliminaires de développement, (2) Minimiser les caractéristiques et variables communes, et (3) permettre la génération de code orienté aspect des agents.

8. Conclusion

Nous avons présenté dans ce chapitre, d'une part, les différentes techniques d'extraction d'aspects à partir des applications orientées objet, et d'autre part, celles qui s'intéressent à la séparation des préoccupations transverses des systèmes orienté agent pendant leurs premières étapes de développement. Bien que ces dernières aient apporté des éléments de réponses très importants au domaine, elles n'ont pas traité les codes sources des applications multi-agent déjà existantes. Par conséquent, l'extraction d'aspects dans les SMA est un domaine qui n'est pas encore abordé.

Chapitre 04

L’Approche Proposée

1. Introduction

Les patrons sociaux et les préoccupations soient *agent Hood* ou *additionnels* vues précédemment sont des préoccupations transverses. Cependant, ils ne sont pas forcement tous totalement séparables, puisque il existe des parties de leurs codes qu'ils doivent être implémentés à l'intérieure du code métier ou dans des blocs du code des autres préoccupations (par exemple: la préparation des messages). Par contre, d'autres préoccupations sont totalement indépendantes du code métier comme les modèles sociaux broker, matchmaker, etc.

Nous présentons dans ce chapitre notre nouvelle approche d'identification des aspects liés au modèle social « MatchMaker » dans les systèmes multi agents basés sur JADE.

2. Extraction d'aspects des applications SMA (L'étude de cas « L'agent DF sous Jade »)

L'extraction d'aspects dans la plupart des applications complexes, en l'occurrence, celles orientées agent, est une tâche très difficile à automatiser. L'intervention de l'utilisateur est indispensable durant l'accomplissement de cette activité. Décider qu'un fragment de code fait partie de telles préoccupations n'est pas facile. En effet, les patrons d'exécution récurrente ou la duplication de code ne signifient pas la présence d'une préoccupation.

L'état de l'art prouve que le bon choix des graines de chaque préoccupation donne une excellente extraction. L'approche proposée (figure 1) est une approche hybride et semi-automatique appliquée sur des applications Multi-agents JADE dans le but d'explorer la préoccupation transverse de l'agent DF.

Cette approche combine entre l'analyse dynamique et l'analyse statique et est basée sur la notion de granularité à deux niveaux: (1) niveau des fragments du code (les appels aux méthodes « l'API + code source ») et (2) niveau de méthodes (les méthodes en totalité implémentées par le développeur « code source »). Le processus de notre approche est accompli dans 10 étapes (figure 1).

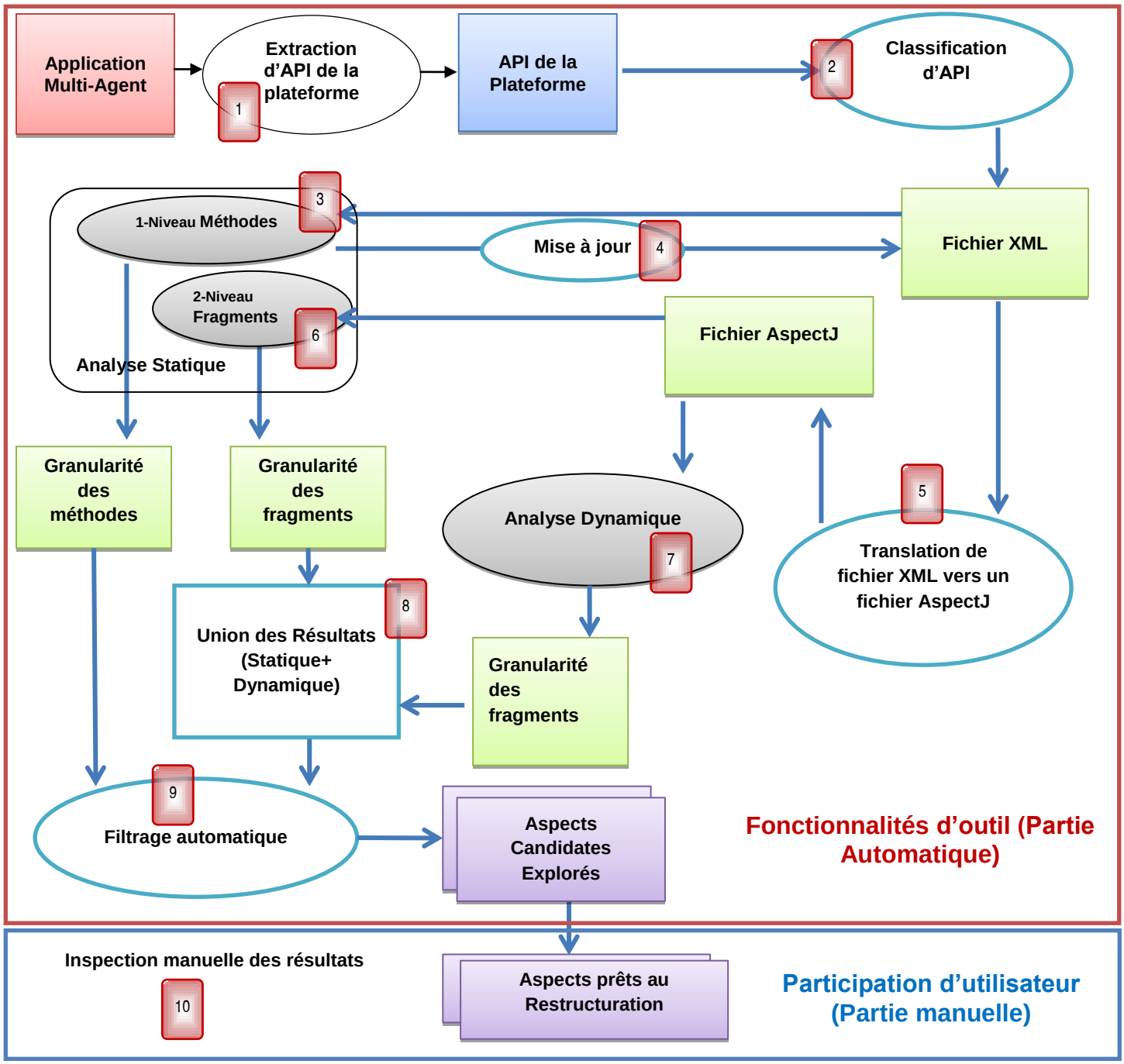


Figure 1. Une approche pour l'extraction des aspects dans les SMA.

2.1. Extraction de l'API de la Platform d'agent (Étape 1)

Les applications multi-agents sont implémentées sur des plateformes offrant des bibliothèques spéciales. Ces dernières offrent un ensemble de packages et classes qui peuvent être utilisées pour implémenter certaines préoccupations comme les protocoles d'interaction, la mobilité, etc. Ces bibliothèques sont appelées API (Application Programming Interface) de la plateforme. Cette phase consiste à extraire tous les éléments Java (packages, interfaces, classes, méthodes) de la plate-forme multi-agent utilisé (dans notre cas JADE) pour développer le système en cours d'analyse.

2.2. Classification de l'API (Étape 2)

Le premier pas est le choix des graines de la préoccupation transverse « MatchMaker » en se basant sur l'API de la plateforme utilisée pour le développement de l'application sous analyse. Nous sélectionnons les packages, les interfaces, les classes et les méthodes fournies par cette API selon la préoccupation transverse qu'ils implémentent.

Cette classification peut être automatisée selon une méthode de classification basée texte (distance sémantique) ou semi-automatique ou bien manuelle. L'objectif est d'explorer les fragments de codes (les appels des méthodes implémentées par les fondateurs de l'API « Ex. JADE ») dispersée dans le système.

Exemple : Cet exemple montre la classification de quelques éléments java (paquetage, classe…) de la Plateforme multi agent JADE (figure 2) :

- Pour la préoccupation Interaction on peut classifier :

  FIPA.ContractNet,
  FIPA.request
  ACLMessage…

- Pour la préoccupation Mobilité on a par exemple :

  jade.domain.*mobility*
  MoveAction,
  MobilityOntology…

- Pour le MatchMaker

  jade.domain.FIPAAgentManagement.ServiceDescription
  jade.domain.DFService…

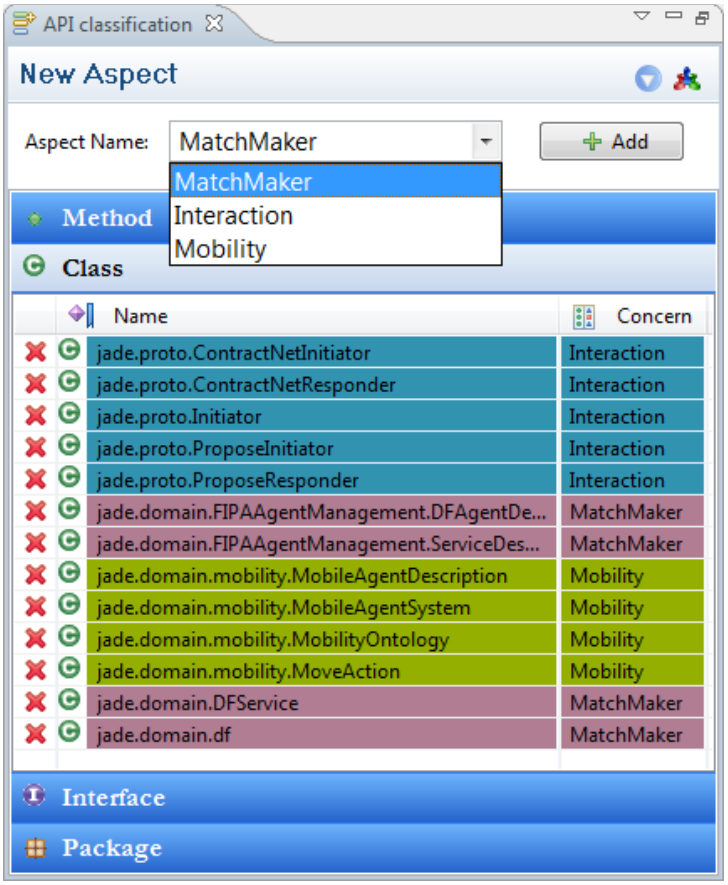


Figure 2. Classification des préoccupations transverses "MatchMaker, Mobilité, Interaction".

Les éléments java classés forment un catalogue des graines d'une préoccupation transverse. Ces graines seront sauvegardées dans un fichier XML, pour une meilleur présentation et compréhension des données. La figure 3 montre la classification de trois préoccupations transverses le MatchMaker, l'interaction et la Mobilité ; chacune des trois contient des sous balises montrant le type de graine. Si la graine est un paquetage ça veut dire que toutes les classes et les interfaces de ce paquetage font partie de cette préoccupation transverse. Si elle est une interface alors toutes les classes qui l'implémentent font partie de la même préoccupation transverse. Dans le cas où la graine est une classe, ses instances et toutes ses méthodes font partie de sa préoccupation transverse.

Figure 3. Le fichier .xml généré après la classification de trois préoccupations transverses de l'API de JADE.

Nous classifions dans notre étude de cas seulement la préoccupation transverse « MatchMaker » de la plateforme JADE (Agent DF) (figure 4).

Figure 4. Classification de la préoccupation transverse « MatchMaker » de JADE.

2.3. L'analyse statique (Étapes 3, 4, 5, 6)

Le processus d'analyse statique est réalisé en quatre étapes (3, 4, 5 et 6). Le catalogue des graines (fichier xml) fournies par l'API nous permet d'identifier statiquement tous les appels des méthodes et les instanciations des classes de l'API JADE qui implémente le matchmaker (analyse du niveau de fragment de l'étape 6). Comme algorithme de recherche, nous réutilisons la classe recherche des marqueurs fournies par le AJDT (AspectJ Development Tools) [6]. Pour cela nous avons besoin d'une autre représentation du catalogue des graines en tant que module AspectJ pour qu'il soit compris par le plugin AspectJ. Cette transformation sera expliqué plus tard (étape 5), mais avant ces phases nous recherchons les méthodes implémentés par le développeur (analyse au niveau des méthodes de l'étape 3). Le

but de ce processus est de classer les méthodes comme des graines dans le fichier xml dans la phase de mise à jour (étape 4), elles peuvent être identifiées en tant que code de matchmaking seulement si le symptôme de l'existence de cette préoccupation transversale est vérifié. L'ordre de ces deux phases (3 et 6) est très important pour assurer que les fragments identifiés statiquement qui contiennent aussi les appels de méthodes explorées récemment dans la première phase d'analyse statique.

Le symptôme d'existence d'une préoccupation transverse au niveau de granularité des méthodes (le critère de classification des méthodes) :

L'exemple suivant (figure 5) montre un pseudo code d'une méthode qui utilise des instances des classes, des invocations aux méthodes, et une exploitation des différents paquetages, ces derniers peuvent être classées ou non dans des préoccupations. Selon la préoccupation dominante (majorité) « *MatchMaker* dans le pseudo code suivant » on affecte la méthode en totalité à la préoccupation appropriée. Nous identifions statiquement tous les types simples (non primitifs) utilisés par la méthode (type de retour , arguments, bloc du code). Le rapport du nombre des occurrences des types simples classés dans la préoccupation transverse du MatchMaker sur le nombre totale des occurrences des types simples sera comparé avec un seuil prédéfini par l'utilisateur «par exemple >50% ». Pour le cas des plusieurs préoccupations transverses, nous pouvons comparer leurs rapports de participation à la méthode qu'elle sera classifiée dans la préoccupation transverse la plus dominante où la différence entre eux doit être importante.

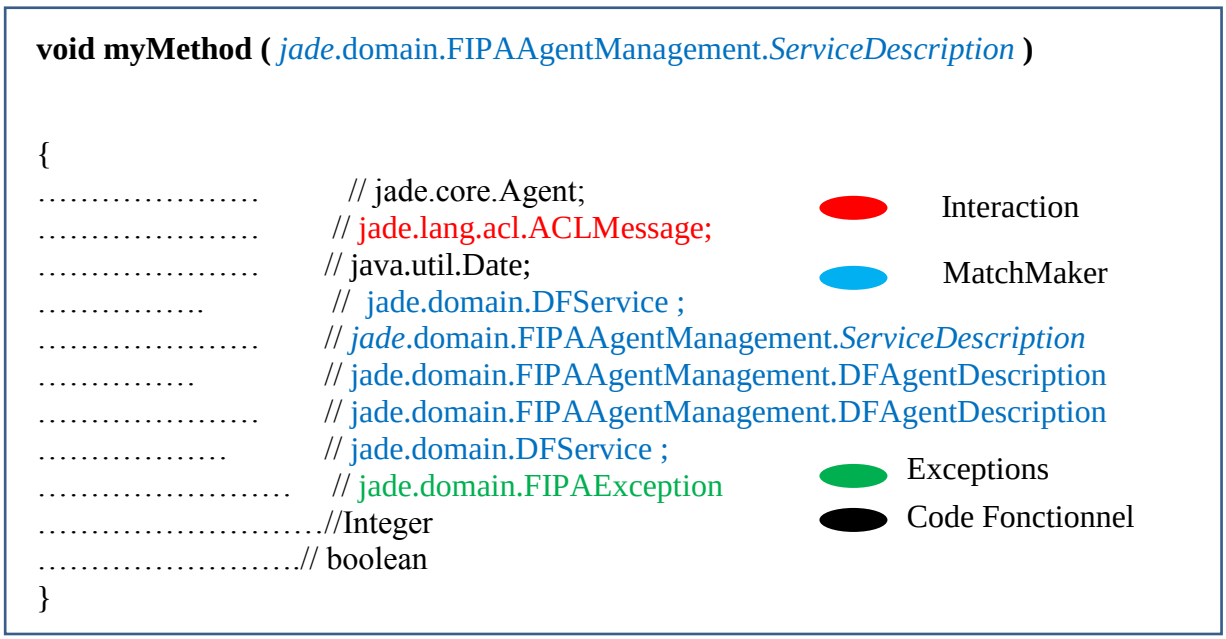


Figure 5. Les types simples d'une méthode.

Le rapport de participation de la préoccupation transverse MatchMaker à l'implémentation de la méthode myMethod() est de (6/10) 60% du code, les types « Integer » et « boolean » sont des types primitifs est n'entrent pas dans le calcul. Si ce rapport est supérieur au seuil défini par l'utilisateur alors cette méthode est un aspect candidat (préoccupation transverse candidate à la restructuration vers un aspect).

Si on cherche à identifier la préoccupation transverse d'interaction aussi on calcule son rapport, qui est dans ce cas égal à (1/10) 10% du code, le rapport de l'interaction est très inférieur à celui du MatchMaker.

La figure 6 présente les étapes de l'analyse statique. Nous remarquons qu'après l'identification de toutes les méthodes qui sont possibles d'être des graines à cette préoccupation transverse nous faisons une mise à jour au catalogue des graines, ensuite nous performons la deuxième étape de l'analyse statique qui est l'identification des invocations de toutes les méthodes classées dans le catalogue (granularité des fragments).

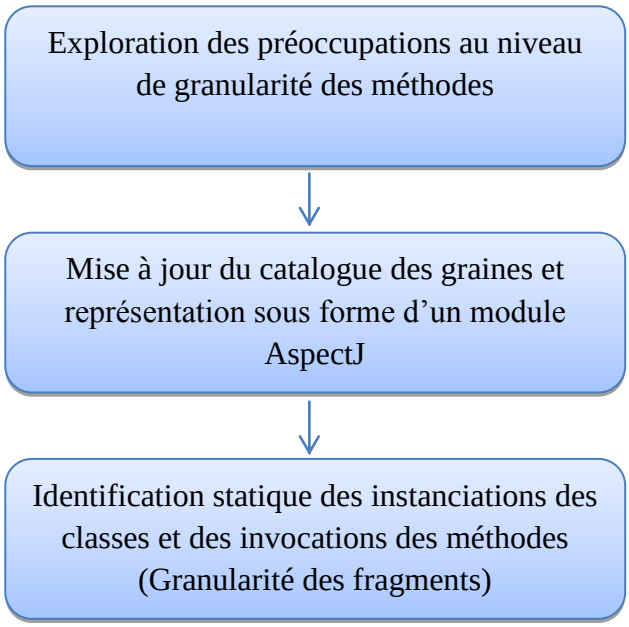


Figure 6. Les étapes de l'analyse statique.

La figure 7 montre le catalogue des graines en format .xml, la balise de type « *method* » comporte une signature qui commence par le nom du paquetage racine de l'application sous analyse, donc ceux sont les graines fournies par le développeur du système et non par l'API et qui sont ajoutées après l'identification des déclarations des méthodes dans la première étape de l'analyse statique.

Figure 7. Mise à jour du catalogue des graines après la première étape de l'analyse statique.

- **La Transformation de fichiers XML vers un module:**

Un moyen simple pour explorer les invocations est par l'exploitation de tisseur AspectJ. Pour cela une conversion automatique de la nature du projet sous analyse vers la nature AspectJ est nécessaire. Cette transformation (figure 8) est faite par l'ajout de la bibliothèque de l'AspectJ au projet.

Seeds.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Aspects>
 <MatchMaker>
  <classe callorexeuction="call">jade.domain.FIPAAgentManagement.DFAgentDescription</classe>
  <classe callorexeuction="call">jade.domain.FIPAAgentManagement.ServiceDescription</classe>
  <classe callorexeuction="call">jade.domain.DFService</classe>
  <classe callorexeuction="call">jade.domain.df</classe>
  <method callorexeuction="call">agentes09.gestor.behaviours.RecibirMensajesGestion.action</method>
  <method callorexeuction="call">agentes09.agentes.AgenteProductor.registrarEnDF</method>
  <method callorexeuction="call">agentes09.agentes.AgenteConsumidor.registrarEnDF</method>
  <method callorexeuction="call">agentes09.agentes.AgenteGasolinera.registrarEnDF</method>
  <method callorexeuction="call">agentes09.gestor.behaviours.EventoDeControl.actualizarAgentesRegistradosTipo</method>
  <method callorexeuction="call">agentes09.gestor.AgenteGestor.registrarEnDF</method>
  <method callorexeuction="call">agentes09.agentes.AgenteCamion.registrarEnDF</method>
 </MatchMaker>
</Aspects>
```

Extraction.aj

```
privileged aspect Extraction  {
pointcut MatchMaker():
call(* jade.domain.FIPAAgentManagement.DFAgentDescription.*(..)) ||
call(jade.domain.FIPAAgentManagement.DFAgentDescription.new(..)) ||
call(* jade.domain.FIPAAgentManagement.ServiceDescription.*(..)) ||
call(jade.domain.FIPAAgentManagement.ServiceDescription.new(..)) ||
call(* jade.domain.DFService.*(..)) ||
call(jade.domain.DFService.new(..)) ||
call(* jade.domain.df.*(..)) ||
call(jade.domain.df.new(..)) ||
call(* agentes09.gestor.behaviours.EventoDeControl.actualizarAgentesRegistradosTipo(..)) ||
call(* agentes09.agentes.AgenteProductor.registrarEnDF(..)) ||
call(* agentes09.gestor.behaviours.RecibirMensajesGestion.action(..)) ||
call(* agentes09.agentes.AgenteConsumidor.registrarEnDF(..)) ||
call(* agentes09.gestor.AgenteGestor.registrarEnDF(..)) ||
call(* agentes09.agentes.AgenteCamion.registrarEnDF(..)) ||
call(* agentes09.agentes.AgenteGasolinera.registrarEnDF(..)) ;

Object around(): MatchMaker(){
// Enregistrement des informations sur le point de jonction intercepté
return proceed();   } }
```

Figure 8. Représentation du catalogue des graines sous la forme d'un Aspect « .aj ».

Le but de cette conversion est l'instrumentation du projet sous analyse par la création d'un aspect d'extraction « Extraction.aj ». Cet aspect contient une coupe (pointcut) qui représente la préoccupation transverse « MatchMaker » à explorer où la coupe possède des points de jonctions qui représentent les graines classées précédemment. L'aspect .aj contient aussi le griffon (advice) de la coupe définie qu'est le code qui sera exécuté lorsqu'un point de jonction (graine) est intercepté. Ce griffon qui enregistre l'information correspondante à la localisation de ce point de jonction tel que :

- L'appelant : la méthode qui fait l'invocation.
- L'appelé : la méthode (constructeur) invoqué.
- Numéro de la Ligne : numéro de la ligne dans la classe appelante où a été passée cette invocation.

- **L'identification des aspects candidates au niveau des fragments:**

AspectJ possède une classe nommée : **UpdateAJMarkers.java** qui permet d'identifier tous les points de jonction déclarés dans un module aspect (.aj) sans exécution du programme. Cette classe est responsable sur le tissage statique de l'outil AspectJ.

Nous instrumentons le système à analyser par l'ajout du catalogue des graines sous forme d'un fichier (.aj) et l'API (Application programming Interface) du plugin AspectJ dans la bibliothèque (Pour que Java comprendre le module .aj), et nous instancions la class **UpdateAJMarkers.java** qui fournir tous les aspects candidats qui ont des graines déclarés dans le catalogue.

2.4. L'analyse Dynamique (Étape 7)

L'exécution du système par tissage des aspects (figure 9) permet d'intercepter les points de jonction déclarés dans la coupe MatchMaker, c'est beaucoup mieux que la génération des traces d'exécutions et leur analyse ultérieure.

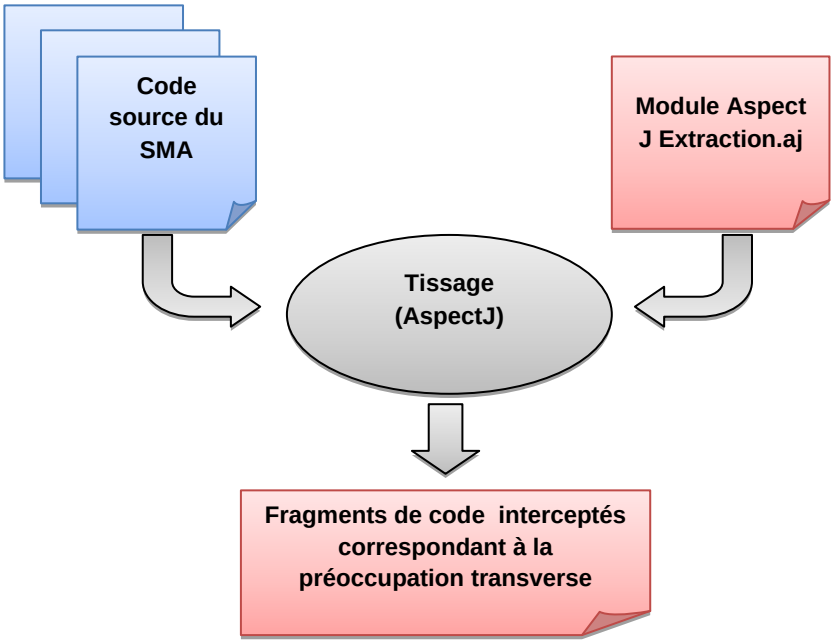


Figure 9. L'analyse dynamique

2.5. Union des résultats (Étape 8)

Les résultats des deux techniques peuvent fournir différents résultats. Si un fragment est identifié par l'analyse statique seulement, alors il faut exécuter plusieurs scenarios dans l'analyse dynamique, ou performer le test dynamique pour une long période. Sinon le système contient un code inutile qu'il faut éliminer. Pour cela il est préférable de fusionner les deux résultats pour avoir tous les aspects candidats possibles fournis par ces deux techniques d'analyse.

2.6. Filtrage Automatique (Étape 9)

La phase de filtrage est très importante. Si un fragment de code de l'API d'une préoccupation transverse est exploré et qu'il est implémenté dans une méthode classée dans la même préoccupation transverse, alors on ignore ce fragment et on garde la méthode entière comme étant un aspect candidat. Si la méthode est classée dans une autre préoccupation

transverse on garde les deux où la méthode sera déclarée comme déclaration inter type par le mécanisme d'introduction et le fragment comme code griffon lors du processus de restructuration.

2.7. Inspection manuel (Étape 10)

Pour avoir la possibilité de manipuler, contrôler et vérifier les résultats avant de les enregistrer, l'utilisateur peut inspecter et modifier manuellement les aspects candidats

3. Conclusion

L'extraction des aspects dans les systèmes multi agents est un nouveau domaine de recherche. Dans la littérature, aucun travail n'est réalisé dans ce contexte. Seulement quelques travaux qui consistent à proposer des architectures pour la séparation des préoccupations d'agent dans les phases préliminaires du développement des SMA.

Après avoir étudié les techniques existantes dans le domaine de l'extraction des aspects, nous avons combiné plusieurs approches comme l'analyse statique dans la granularité des méthodes et des fragments et l'approche dynamique dans la granularité des fragments de codes seulement. Il est clair que la technique dépend totalement des graine sélectionnées au début, donc elle nécessite une pré-connaissance de l'API utilisée. Pour avoir un bon catalogue, la phase de sélection peut être automatisée par plusieurs techniques de classification existantes qui sont basées sur les conventions de nommage des éléments (java dans ce cas). Pour concrétiser notre approche, nous avons développé un outil graphique implémenté en java et basé sur Eclipse RCP, sa présentation et validation est décrite dans le chapitre suivant.

Chapitre 05

Validation de l’approche

1.Introduction

Dans le cadre de notre projet nous avons développé un outil appelé MAMIT (Mas Aspect MIning Tool). Cet outil implémente et concrétise notre approche d'extraction d'aspects dans les applications multi-agents.

Avant de passer à la présentation de notre outil, nous préférons présenter sa conception en utilisant différents diagrammes UML. L'objectif étant de donner plus de détails sur l'outil et de ne pas se limiter à présenter directement son utilisation et les résultats qui peut donner.

2.Conception de l'outil

Nous utilisons trois diagrammes UML: diagramme de cas d'utilisation, diagramme de séquences, et diagramme de classe.

2.1. Diagramme de cas d'utilisation

Dans la modélisation par cas d'utilisation (figure 1) deux concepts fondamentaux doivent être présentés

- **les acteurs ou les utilisateurs du système** : ceux sont les entités qui gèrent le système. Ils doivent connaitre toutes les fonctionnalités fournies par l'outil.
- **Les cas d'utilisation** qui décrivent les fonctionnalités fournies par le système.

Notre outil offre les fonctionnalités suivantes :

1. **La préparation d'extraction:**
    - L'utilisateur importe le système à analyser et demande l'extraction de l'API utilisé dans le code source.
    - le système extrait les éléments java et les classifie dans des paquetages, interfaces, classes, méthodes.
    - l'utilisateur crée une nouvelle classe (catégorie) de la préoccupation transverse à identifier et choisit les éléments java de l'API utilisés à son implémentation.
    - le système sauvegarde cette classe dans un fichier xml (catalogue).

2. **L'analyse Statique**
    - L'utilisateur lance l'analyse statique
    - le système extrait les graines à partir du fichier xml, avant qu'il performe la première phase d'analyse (niveau méthodes), il transforme la type du projet

vers le type AspectJ, et le fichier xml vers un fichier aspectJ et il l'ajoute au paquetage par default du projet. Après l'analyse, il met à jour le fichier xml,

- Avant la deuxième phase d'analyse (niveau fragments) l'outil performe une mise à jour de module AspectJ.

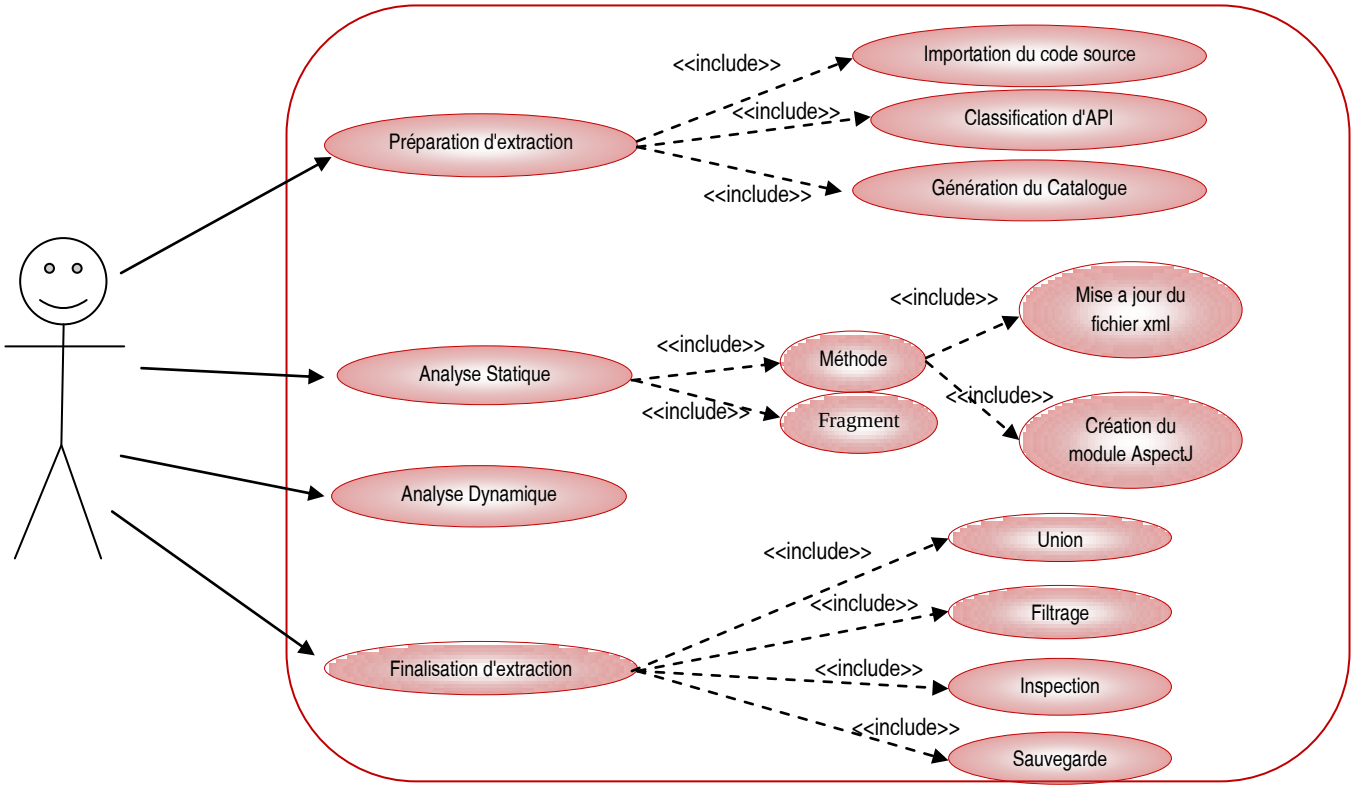


Figure 1. Diagramme des cas d'utilisation.

3. L'analyse dynamique

- l'utilisateur exécute le projet
- le système intercepte les fragments correspondants aux graines du catalogue
- l'utilisateur arrête l'exécution
- le système affiche les fragments interceptés

4. La finalisation d'analyse:

- Union des résultats:
  - L'utilisateur demande d'unifier les résultats des deux approches au niveau des fragments.
  - Le système affiche les résultats unifié dans une seule liste
- Filtrage automatique:

- L'utilisateur demande le filtrage automatique des fragments identifiés
- Le système élimine les fragments en fonction de méthodes appelantes, identifiés et affiche les résultats.

- Inspection Manuelle et sauvegarde des résultats
  - L'utilisateur élimine les aspects candidats jugés inappropriés ou non ré-structurable et sauvegarde les résultats finaux.
  - Le système fait le changement et sauvegarde les aspects candidats finaux.

2.2. Diagramme de Séquences

Le diagramme de séquences est la représentation graphique des interactions entre les acteurs et le système selon un ordre chronologique. Il peut montrer les interactions dans le cadre du scénario d'un cas d'utilisation ou même les scénarios globaux d'un système. Dans la figure 2, nous donnons un diagramme de séquence qui représente les interactions globales entre les différentes fonctionnalités de notre outil et l'utilisateur.

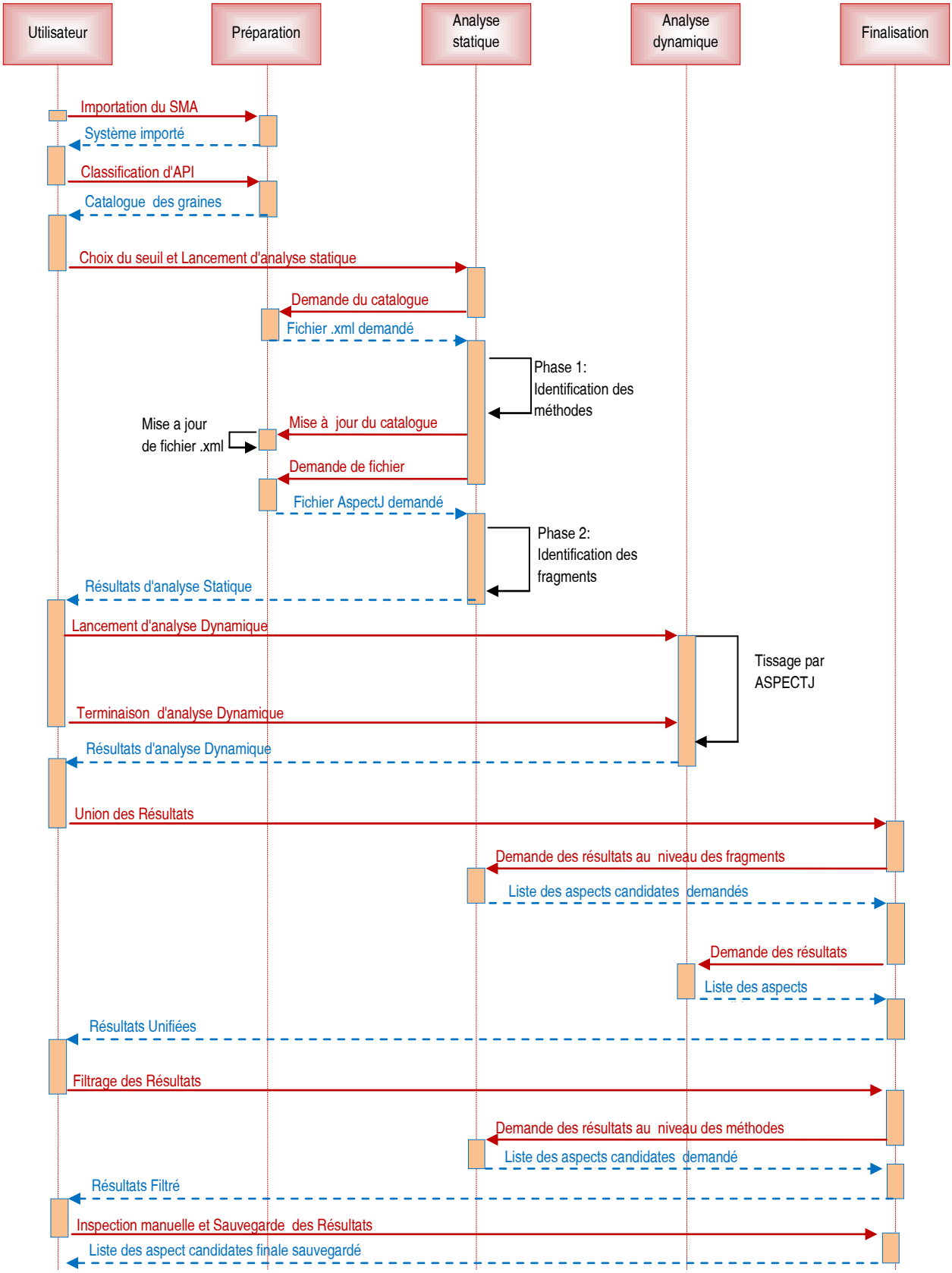


Figure 2. Diagramme de séquence.

2.3. Diagramme de classes

Le diagramme de classes est un schéma pour présenter les classes et les interfaces d'un système ainsi que les différentes relations entre elles. Nous l'utilisons pour montrer les entités essentielles de notre système.

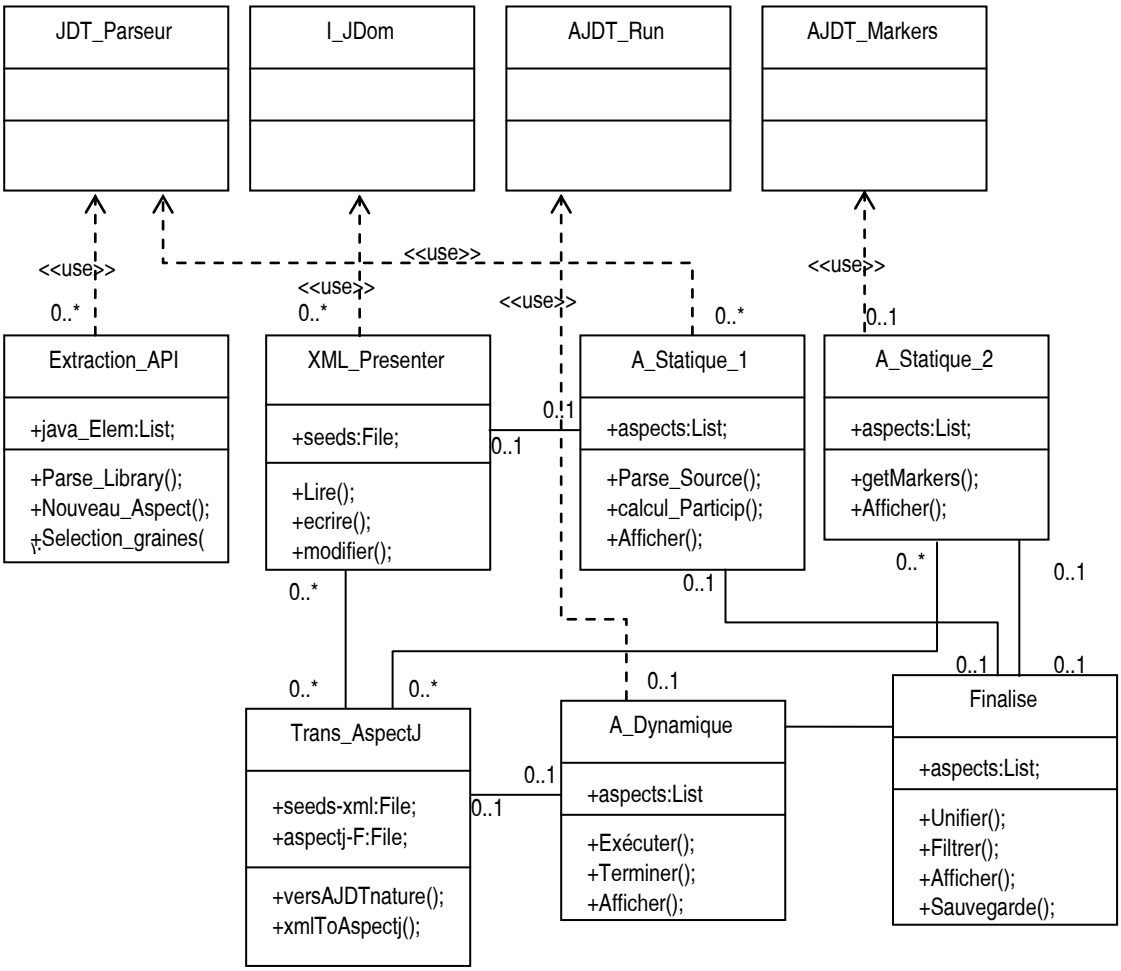


Figure 3. Diagramme des classes.

L'implémentation des plugins sous ECLIPSE RCP dépend des notions importantes comme "*views*" ," *Actions*" , "*Commands*", "*extension points*" , "*dependencies*" ...etc. Ce diagramme de classe (figure 3) présente une vue générale et simplifié des classes et leur relations.

- ***JDT_Parseur:*** est le parseur de code java fourni par le plugin JDT d'Eclipse (Java Developpement Tool). Il nous permet de parcourir tous les éléments d'un projet Java.

- ***I_JDom:*** est l'interface du pluging nommé "*Jdom*" qui facilite la lecture et l'écriture des fichiers .xml.

- ***AJDT_Run :*** est la classe responsable sur le tissage des aspects dans un projet AspectJ. Elle est fournie par le plugin AJDT (AspectJ développement Tools).
- ***AJDT_Markers:*** est la classe responsable de la recherche des points de jonctions déclarés dans les coupes des aspects d'un Project.
- ***Extraction_API:*** est la classe responsable de l'extraction de la plateforme multi agent utilisée (JADE), de la création de la préoccupation transverse à explorer et de la sélection des éléments java de l'API qui implémente cette préoccupation transverse.
- ***XML_Presenter:*** est la classe qui permet la création, la lecture, l'écriture et la modification du fichier .xml
- ***A_Statique_1:*** est la classe qui cherche à identifier les préoccupations transverses candidates au niveau des méthodes. Elle parcourt le code source et calcul la participation de la préoccupation transverse à explorer dans chaque déclaration de méthode.
- ***A_Statique_2:*** cette classe peut déceler les fragments du code qui implémente le MatchMaking à l'aide du plugin AJDT par l'invocation de la classe ***AJDT_Markers***.
- ***Trans_AspectJ:*** cette classe nous permet d'ajouter la bibliothèque AspectJ au projet en cours d'analyse pour qu'il puisse comprendre le fichier aspectJ créé à partir du fichier .xml.
- ***A_Dynamique:*** permet d'exécuter le projet sous analyse par le tisseur ASPECTJ. Elle permet aussi d'arrêter l'exécution et afficher les fragments interceptés.
- ***Finalise:*** c'est la classe qui réunit , filtre et sauvegarde les résultats, et qui permet aussi à l'utilisateur de les modifier.

3. Présentation de l'outil

Pour valider notre approche, nous avons développé un outil visuel nommé MAMIT (Mas Aspect MIning Tool).

3.1. Langage utilisé

Cet outil est développée en java, suivant l'approche de développement des plugins sous Eclipse RCP (rich client platform). L'avantage de suivre cette méthode d'implémentation est l'architecture extensible qui permet de réutiliser d'autres plugins comme JDT (Java Développement Tool) la possibilité de l'intégration de l'outil dans Eclipse qui permet sa réutilisation par d'autres utilisateurs.

3.2. MAMIT (Mas Aspect MIning Tool)

MAMIT est un environnement qui automatise l'extraction des préoccupations transverses dans les applications Multi-Agent suivant l'approche proposée. En utilisant MAMIT, l'utilisateur doit performer les étapes ci-après et qui sont également illustrées par la figure 2:

1- Importation du code source du système multi agent à l'explorateur des packages,
2- Extraction de l'API de la plateforme agent utilisé à l'implémentation,
3- Création de la préoccupation transverse à explorer et sélection de ses éléments java (packages, interfaces, classes, méthodes) appropriés qui forment ses graines où ils seront représentés sous la forme d'un fichier .xml,
4- Faire une extraction de cette préoccupation transverse par l'analyse statique, où le projet sera transformer vers l'approche orientée aspect par l'ajout automatisé de la bibliothèque du tisseur AspectJ et la transformation du catalogue des graines à un aspect d'extraction,
5- Faire la deuxième extraction dynamique par compilation du code source en utilisant le tisseur AspectJ pendant une période limitée,
6- Unifier les résultats des deux extractions statiques et dynamiques au niveau de la granularité des Fragments,
7- Filtrer automatiquement les résultats unifiés au niveau des fragments de code,
8- Inspecter les aspects candidats explorés au niveau des méthodes et des fragments,
9- Sauvegarder des aspects candidats explorés pour les utiliser dans le processus de la restructuration.

3.3. Étude de cas : La préoccupation transverse « MatchMaking » sous JADE

Afin de valider notre approche et de montrer le fonctionnement de notre outil, nous avons appliqué MAMIT sur un exemple concret. Il s'agit de l'extraction de la préoccupation transverse de « MatchMaking » dans Jade implémentée comme un agent nommé le facilitateur d'annuaire (DF) dans une application multi-agent appelée agentes09 [1], qui simule un système d'enchères. La table 1 montre quelques informations sur le code source de l'application.

Statistiques	Valeur
Total Packages	12
Total Classes	97
Total Méthodes	263
Lignes du code	2885

Table 1. Statistiques d'Agentes09[1].

La figure 4 présente sa complexité cyclomatique qui est une mesure de logiciel développé par Thomas J. McCabe, Sr. en 1976. Elle est utilisé pour indiquer la complexité d'un programme. Il mesure directement le nombre de chemins linéairement indépendants dans un code source du programme.

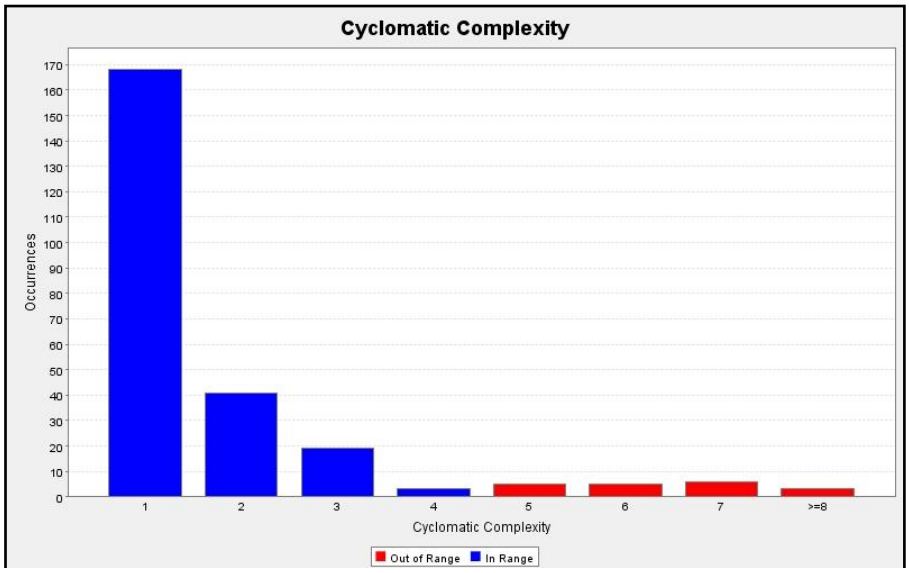


Figure 4.La complexité cyclomatique de l'application agentes09[1].

La figure 5 présente une classe de l'application qui est responsable de la création des agents de l'enchère.

```
package agentes09.agentes;

import agentes09.negotiation.behaviours.CrearAgenteGestor;
import agentes09.negotiation.behaviours.CrearAgentesProductor;
import agentes09.negotiation.behaviours.CrearAgentesGasolinera;
import jade.core.Agent;


public class AgenteCreador extends Agent{

@Override
protected void setup() {
super.setup();
addBehaviour(new CrearAgenteGestor());
addBehaviour(new CrearAgentesProductor());
addBehaviour(new CrearAgentesGasolinera());
}        }
```

Figure 5. Un échantillon de code source de l'application agentes09.

3.4. Aperçu général

MAMIT est déployé comme une application autonome et indépendante de l'EDI Eclipse et aussi peut être déployé sous la forme d'un Plugin. La figure 6 présente un aperçu général de quelques fenêtres (vues) fourni par l'outil.

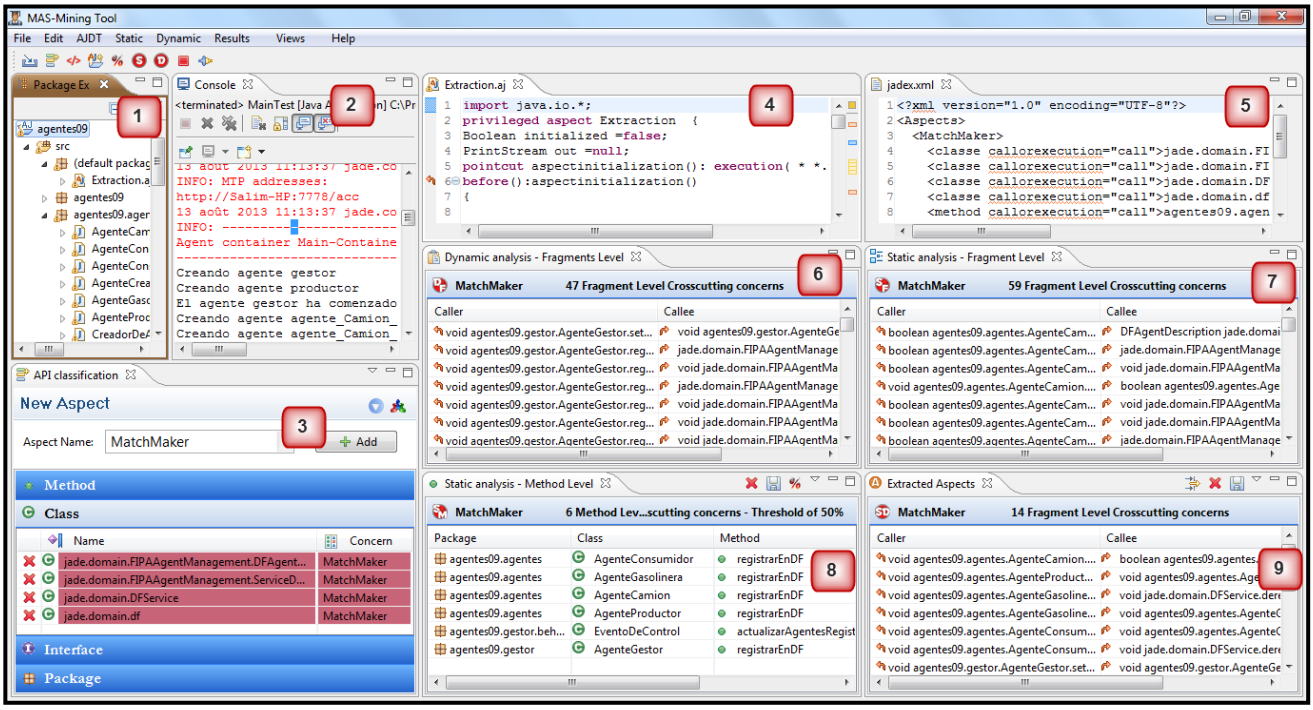


Figure 6. Aperçu général des quelques vues d'outil.

- Section 1 : l'explorateur des packages.
- Section 2 : la console.
- Section 3: vue pour la classification de l'API.

- Section 4 : L'éditeur Java.
- Section 5 : L'éditeur XML.
- Section 6 : résultat de l'analyse dynamique au niveau des fragments
- Section 7 : résultats de l'analyse statique au niveau des fragments
- Section 8 : résultat de l'analyse statique au niveau des méthodes
- Section 9 : résultats finaux réunis et filtrés au niveau des fragments

3.4.1. Préparation d'extraction

La figure 7 présente les premières étapes de la technique. La section1 montre l'importation du projet, la 2ème section offre la liste de tous les éléments java de l'API jade où nous sélectionnons ceux qui implémentent la préoccupation transverse MatchMaker. Dans cette figure seulement les éléments classés sont affichés. La section3 montre le fichier .xml généré après la classification. Section 4 est l'editor java contenenant une classe du système Multi agent. Section 5 présente les graines sous la forme d'un aspect « Extraction.aj »

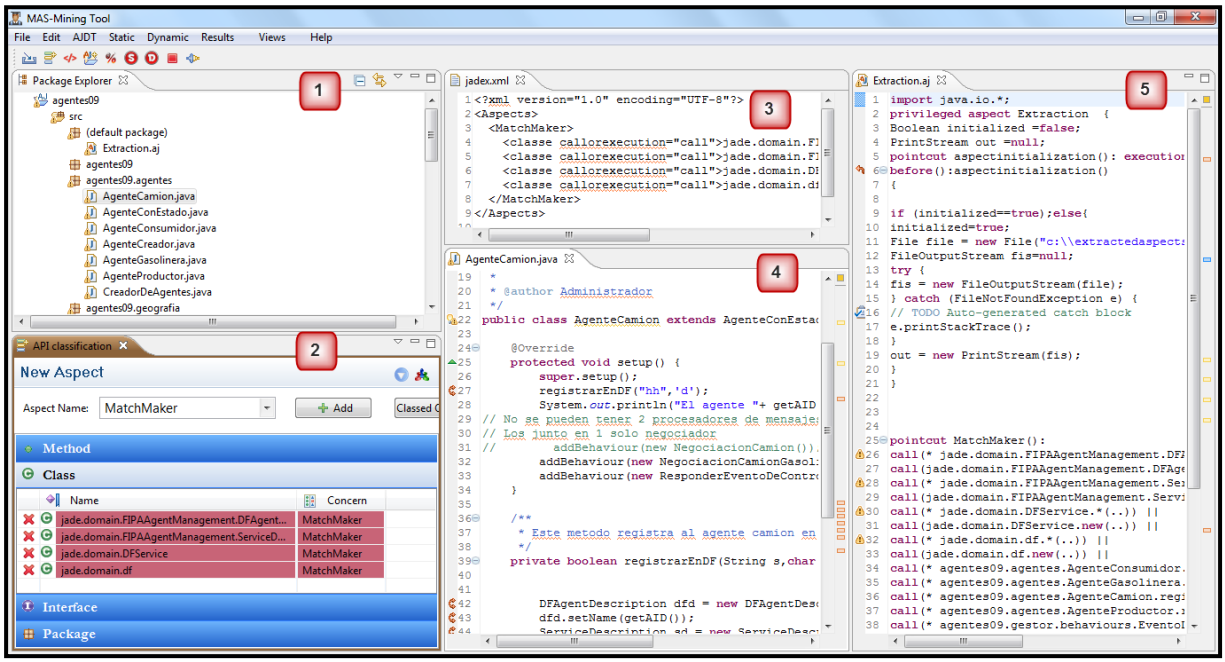


Figure 7. Préparation de l'extraction.

❖ **Classification de l'API sous JADE**

La figure 8 présente la vue responsable de la sélection des éléments java qui forment les graines de la préoccupation transverse de l'agent DF. Nous en avons classé seulement quatre qui participent dans l'implémentation des différentes tâches de DF comme l'enregistrement des- enregistrement, etc.

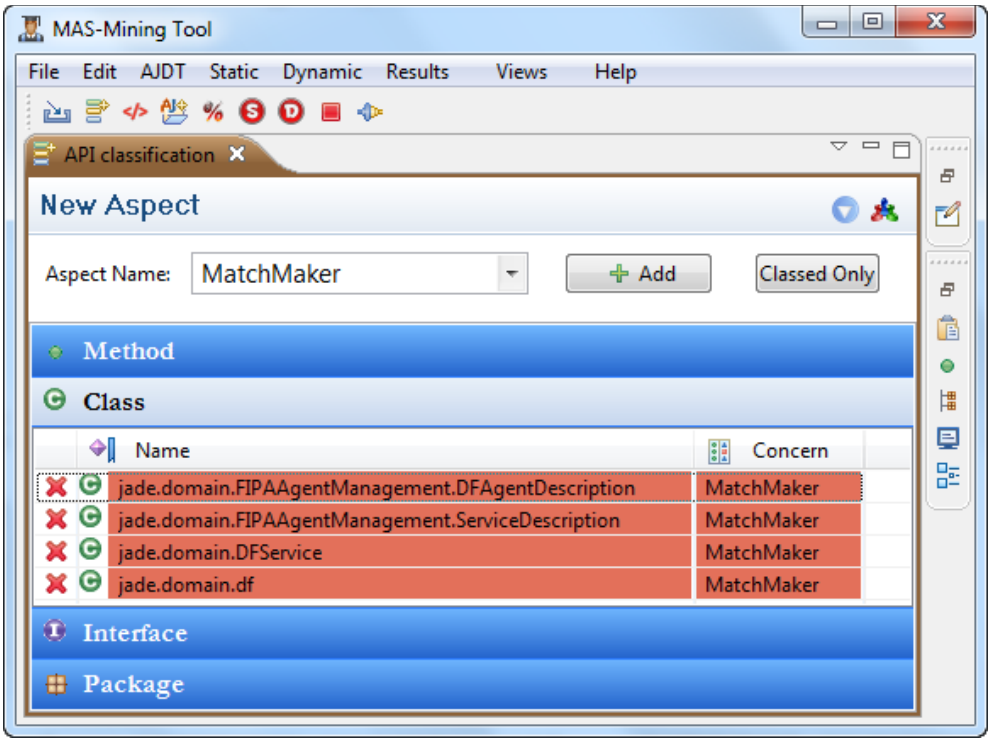


Figure 8. Classification de l'API(MatchMaker).

3.4.2. L'analyse statique

La première étape de l'analyse statique est l'extraction des préoccupations transverses au niveau de granularité des méthodes à l'aide du catalogue généré dans la phase de classification d'API. La figure 9 montre la configuration initiale du catalogue et sa représentation en un module(Aspect) AspectJ.

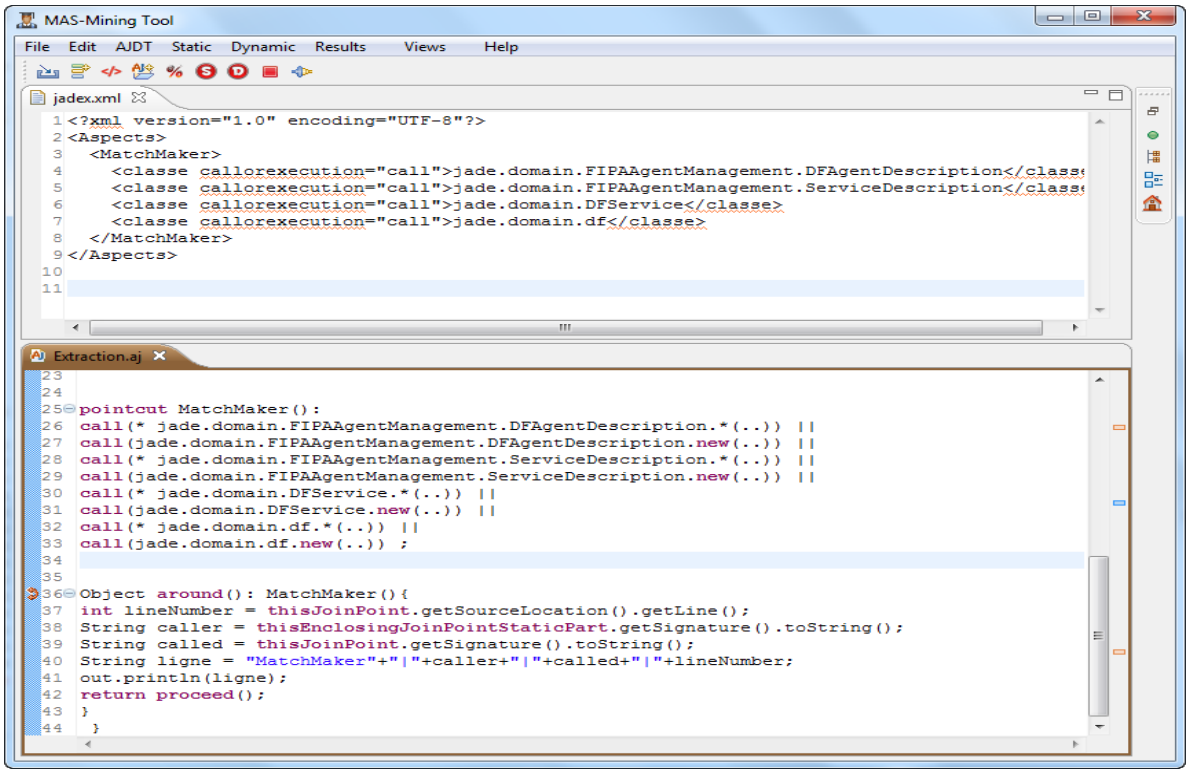


Figure 9. Le catalogue des graines initiales et l'aspect correspondant.

Les résultats de cette phase sont présentés dans la figure 10 qui montre la détection de six méthodes dont la préoccupation transverse dominante est le MatchMaker avec un seuil de 50%.

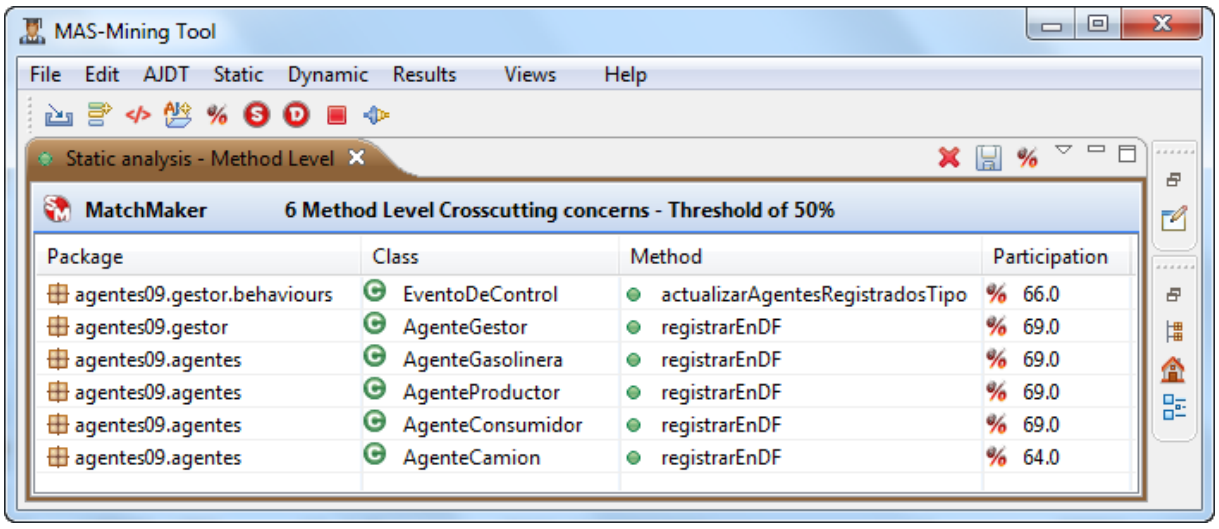


Figure 10. Résultats de l'analyse statique au niveau des méthodes.

Ces méthodes doivent être ajoutées au catalogue des graines pour avoir la possibilité de détecter leurs appels. La figure 11 montre la mise à jour du catalogue et le module aspect correspondant. Dans le fichier xml, les balises « method » dont le contenu débute par le nom du paquetage racine du système sont les nouvelles graines.

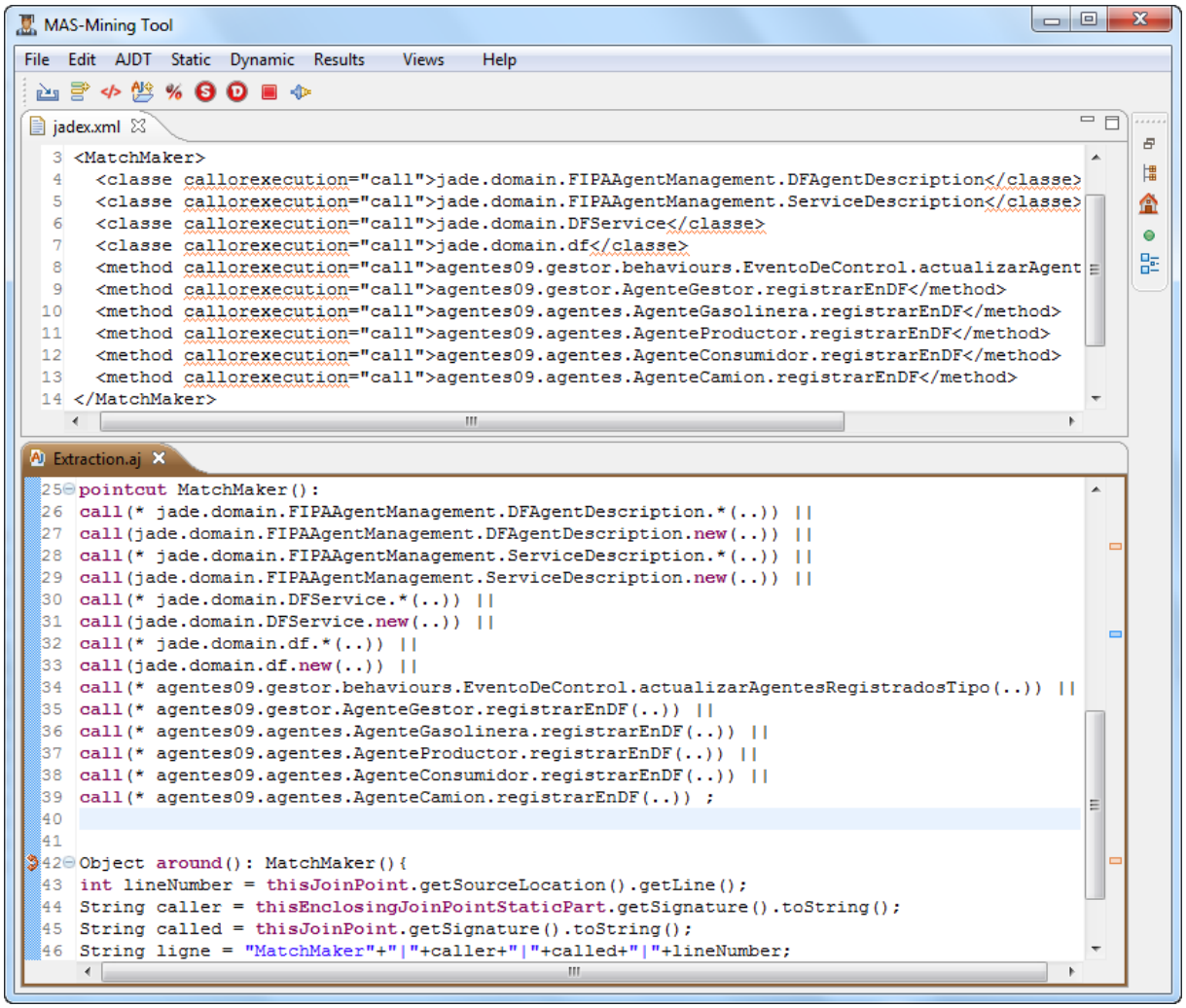


Figure 11. Le catalogue des graines final et l'aspect correspondant.

La deuxième partie de l'analyseur statique est l'identification des appels. La deuxième section de la figure 12 présente l'identification de 59 aspects candidats au niveau de fragment de code qui sont des invocations des méthodes ou des constructeurs fournis par l'API comme DFSercice.register(..) et qui sont fournis par le système comme registerEnDF(..).

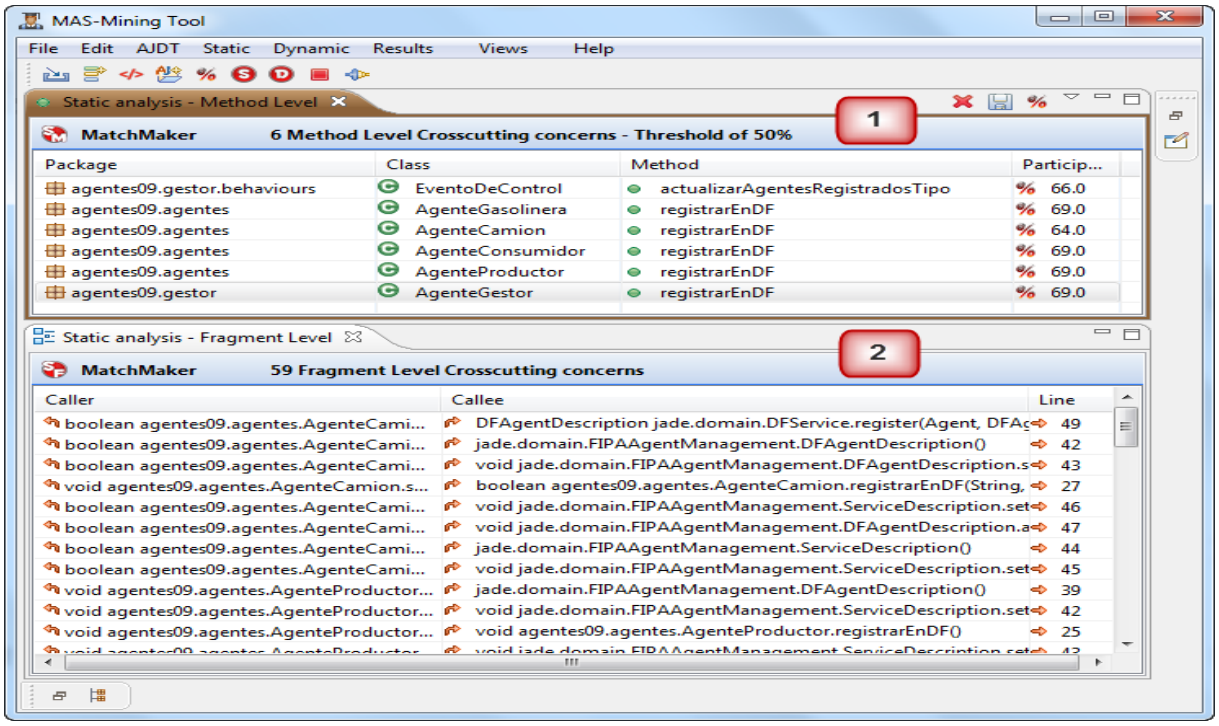


Figure 12. Résultats de l'analyse statique au niveau des méthodes et des fragments.

Lorsque nous ne mettons pas à jour le catalogue après la détection des aspects candidats au niveau des méthodes nous tomberons sur un cas erroné. Dans ce cas le nombre est inférieur au cas précèdent ce qui est présenté dans la deuxième section de la figure 13 où les appels des méthodes identifiées précédemment sont manquants.

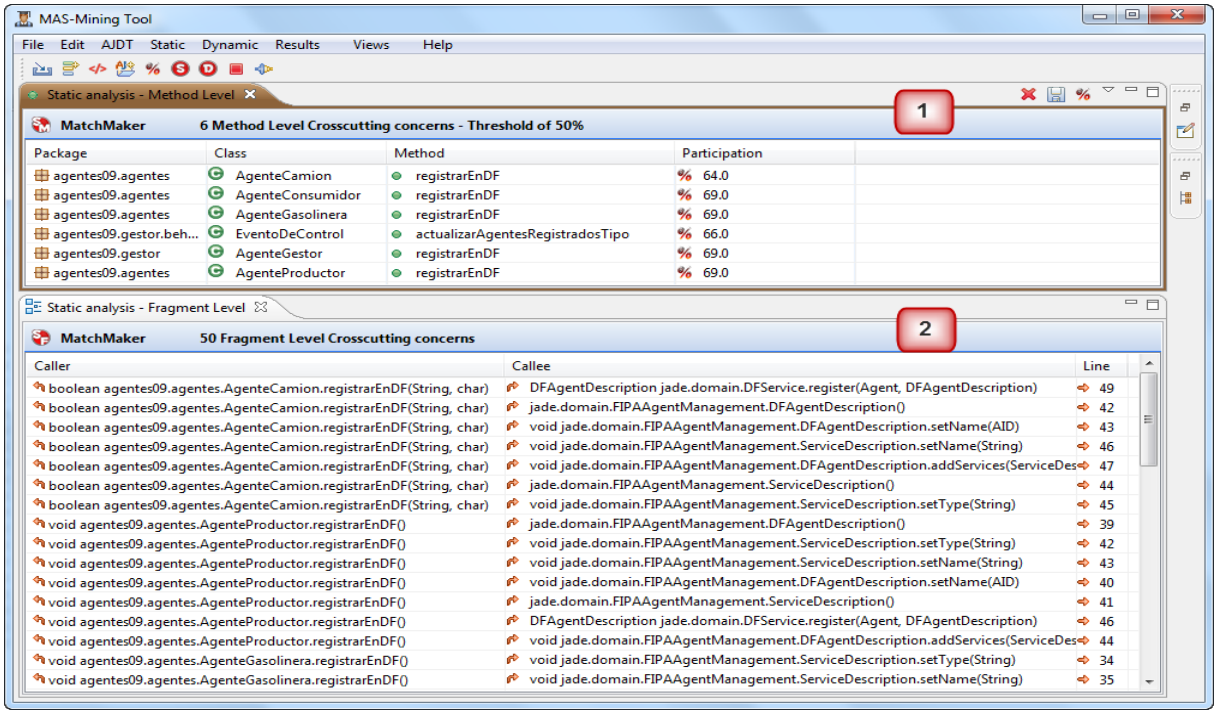


Figure 13. Résultats de l'analyse statique en inversant l'ordre de ses étapes.

3.4.3. L'analyse Dynamique

Dans la figure 14 nous faisons une extraction dynamique des aspects candidats. Les résultats sont seulement au niveau de granularité des fragments. Nous avons identifié 47 invocations des méthodes et constructeurs classés comme MatchMaker (section 4).

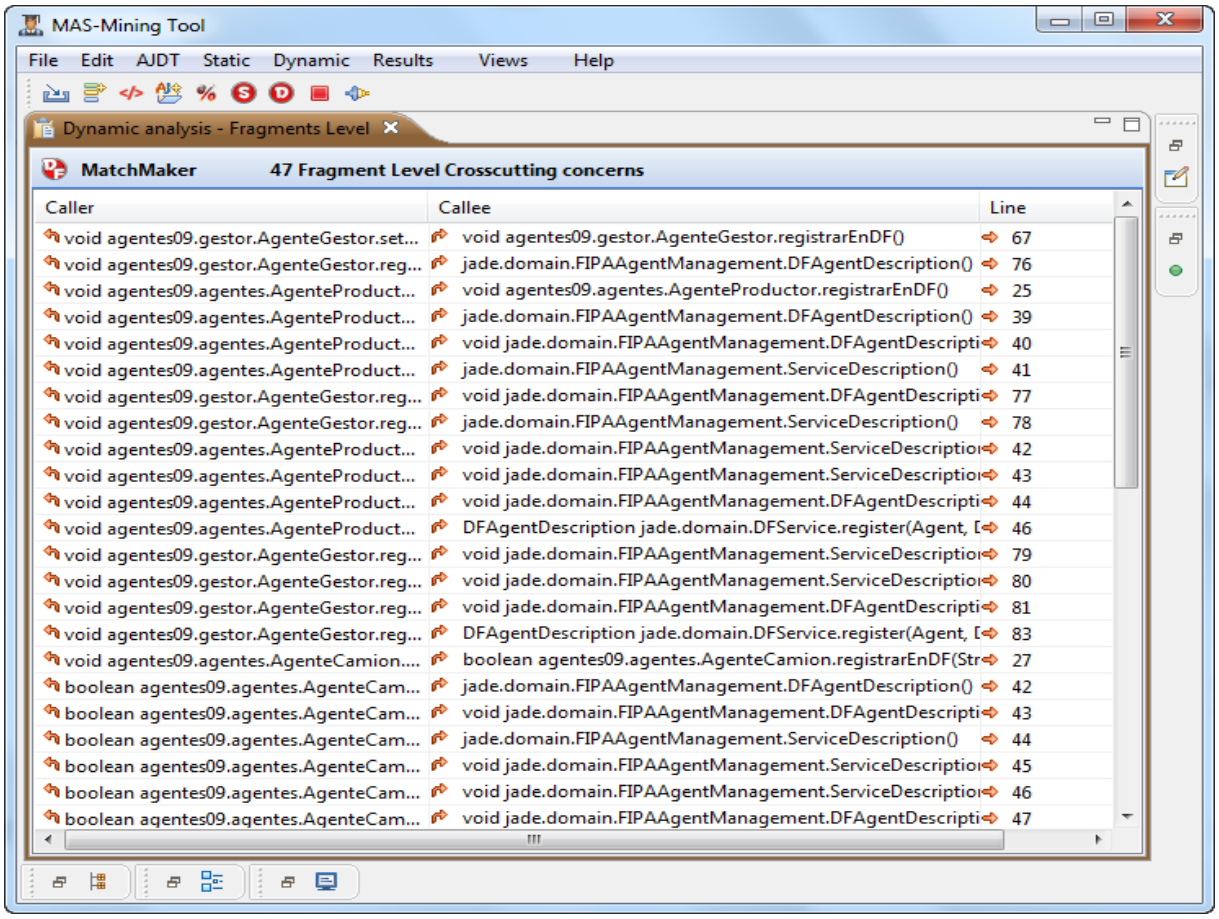


Figure 14. Résultat de l'analyse dynamique au niveau des fragments.

Ce nombre est inférieur au nombre des aspects candidats au niveau des fragments identifié dans l'analyse statique c'est que signifie qu'il y a des faux négatifs (des aspects candidats non identifiés). Nous les comparons avec ceux de l'analyse statique on peut déterminer les fragments non exécutés où la cause peut être:

- Le délai du test dynamique n'est pas suffisant,
- Ces fragments ne participent pas dans le scenario passé,
- Une male implémentation cause un code inutile, qu'il faut l'éliminé.

La figure 15 montre les résultats de l'analyse dynamique où les lignes colorées sont les fragments identifiés statiquement seulement.

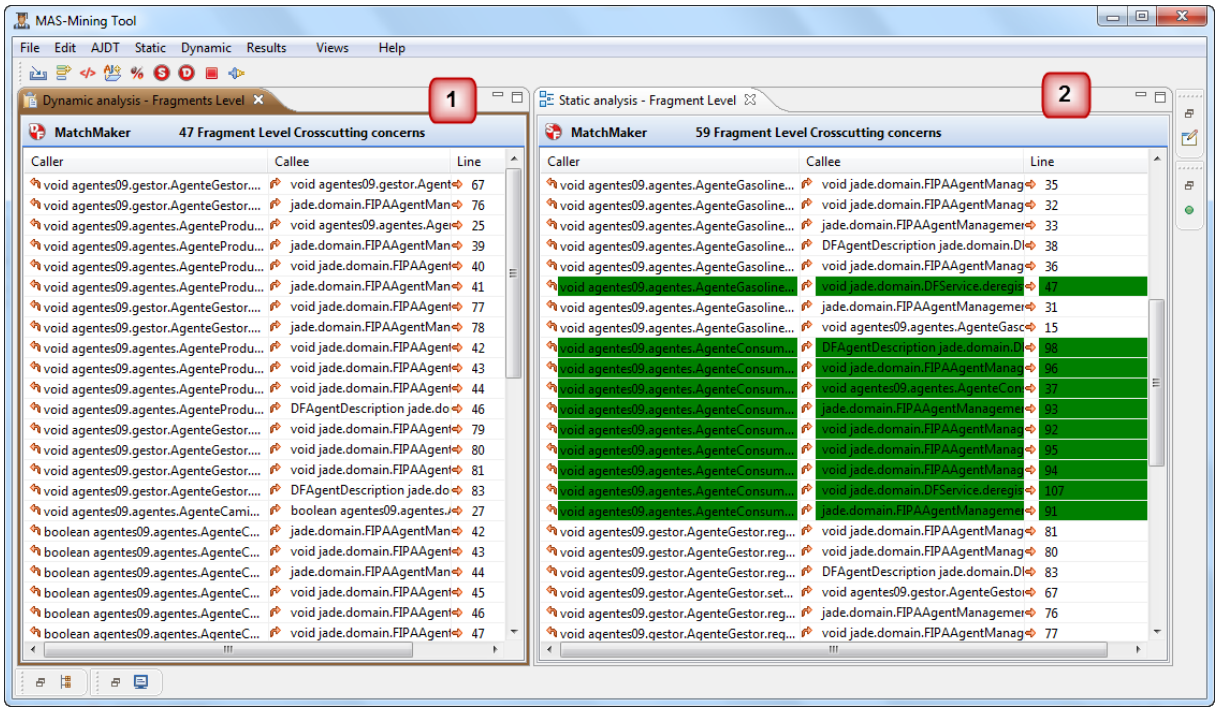


Figure 15. La différence entre les résultats des deux approches au niveau des fragments (fragments manquants dynamiquement).

- **Est-ce que le résultat de l'analyse dynamique peut avoir des fragments non identifiés statiquement ?**

La réponse est oui. Bien que ce soit rare, le développeur du système peut utiliser des techniques d'ajout dynamique du code au projet.

Un autre cas est relié à l'approche proposée. si les deux étapes de l'analyseur statique (les étapes 3et 6) sont inversées alors le résultat de l'analyse dynamique montre l'existence des fragments non identifiés statiquement qui sont les appels des méthodes marquées comme aspects candidats. La figure 16 présente ce cas où les lignes colorées représentent la différence entre les résultats des deux analyses (fragments manquants par les deux approches).

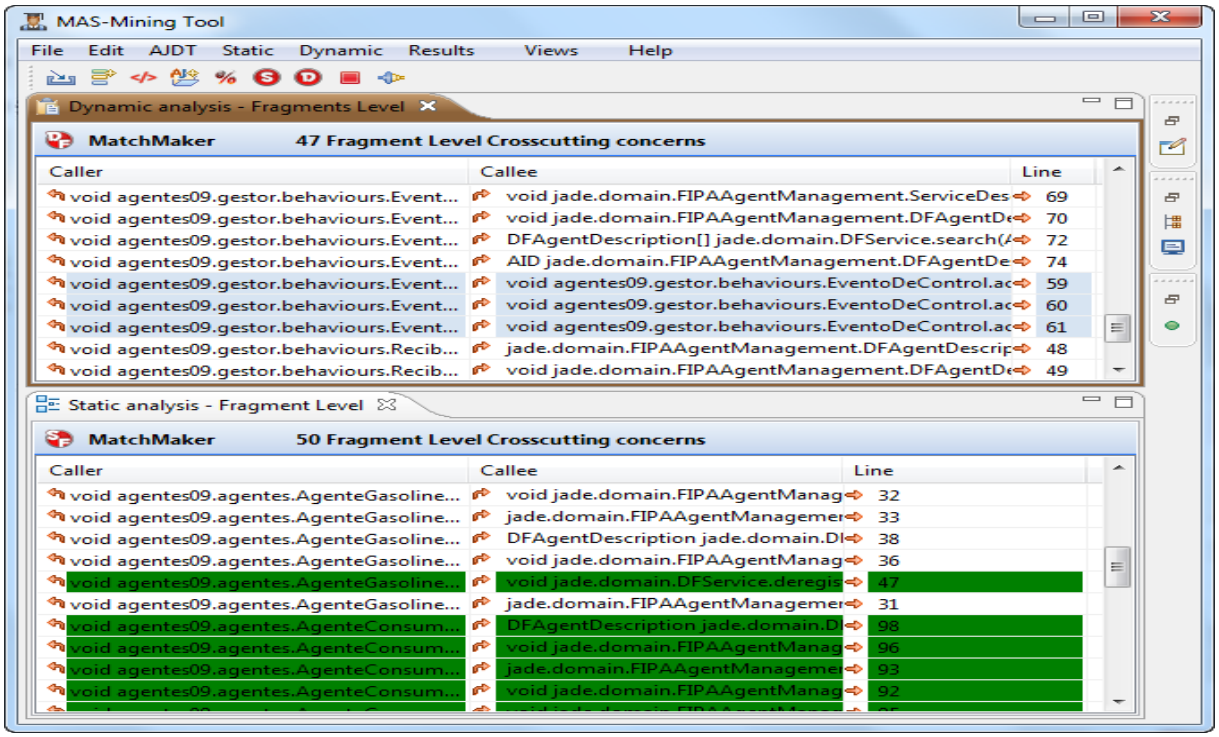


Figure 16. La différence entre les résultats dans le cas d'inversement des étapes de l'approche statique.

3.4.4. Finalisation des résultats

❖ **Union des résultats**

Comme nous avons mentionné, cette phase permet d'unifier les fragments de code identifiés statiquement et dynamiquement pour avoir tous les aspects candidats possibles dans une seule liste. La figure 17 présente les aspects candidats possibles au niveau des fragments

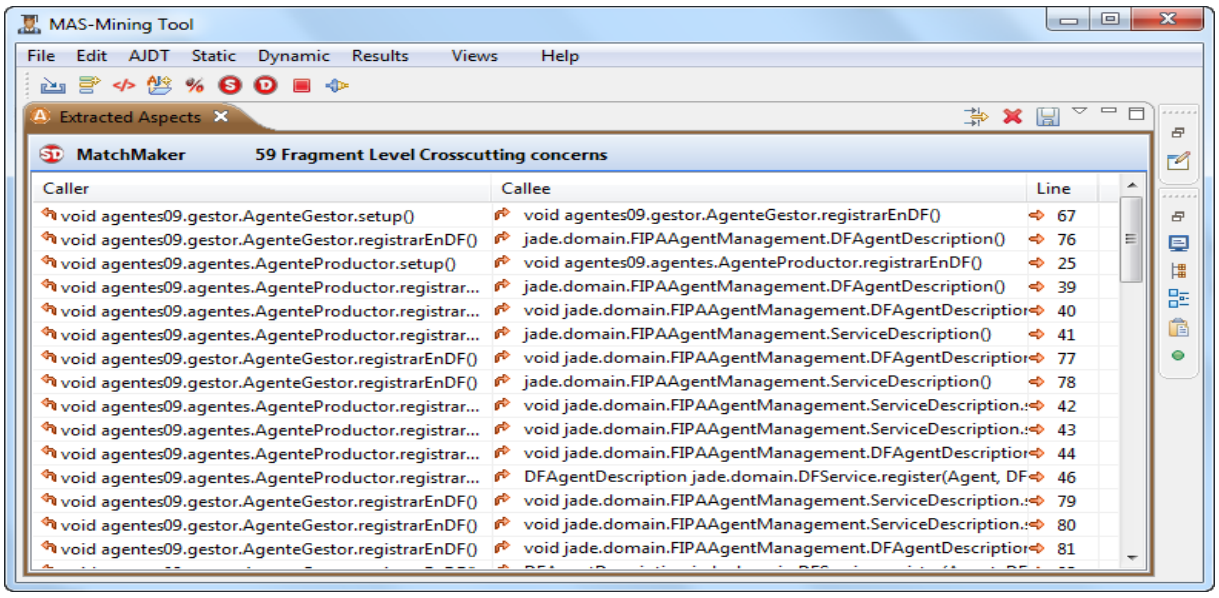


Figure 17. Union des résultats au niveau des fragments.

La figure 18 montre l'union des résultats dans le cas d'inversement des étapes de l'analyse statique (3 et 6), où nous avons identifié 47 aspects candidats par l'analyse dynamique et 50 aspects candidats par l'analyse statique, et après l'union nous avons obtenu 58 aspects candidats. Donc l'union nous permet d'éviter la perte de plusieurs aspects candidats qui reste inferieure à celle du cas précédent de 59 aspects candidats où l'ordre des étapes a été respecté.

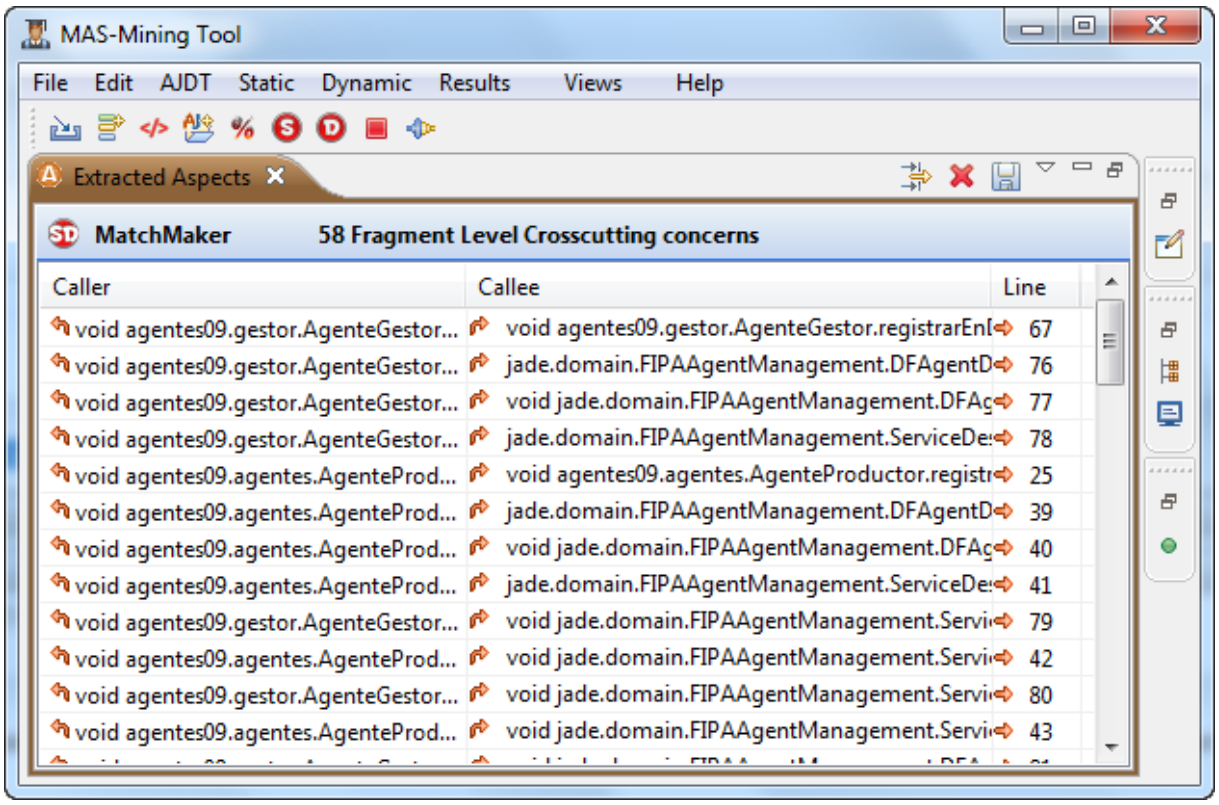


Figure 18. Union des résultats dans le cas d'inversement des étapes de l'analyse statique.

❖ **Filtrage automatique**

Le filtrage est l'élimination des aspects condidats au niveau des fragments unifiés précedement où la methode appelante d'aspects condidats de la section 2 (figure 19) est identifée dans la section 1 de la figure 19. Cette figure montre que 18 aspects condidats au niveau des fragments sont identifés au lieu de 59 condidats, donc 41 fragments sont implementés dans les 6 methodes identifiées statiquement.

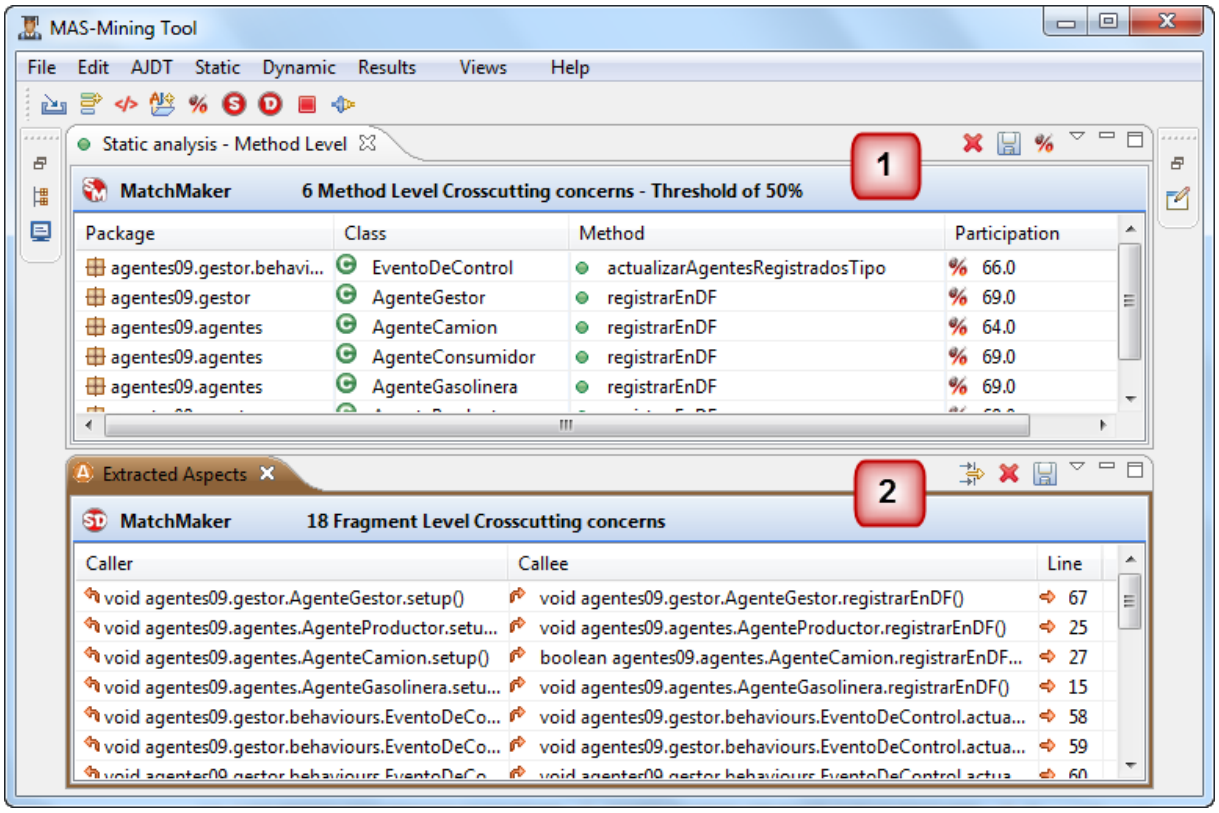


Figure 19. Filtrage automatique des résultats au niveau des fragments.

❖ **Inspection manuelle et Sauvgarde des resultats**

Avant de sauvegarder les résultats finaux, l'utilisateur peut inspecter les aspects candidats explorés au niveau des invocations.

4. Discussion

Nous discutons dans cette section les résultats obtenus au niveau des granularités des fragments et des méthodes séparément, et nous évaluons l'approche proposée selon deux critères: la précision et le rappel qui sont calculés en utilisant la matrice de confusion (table 2) des résultats.

		Réel	
		Positif	Négatif
prédit	Positif	VP	FP
	Négatif	FN	VN

Table 2.Définition de la matrice de confusion.

Où :

- VP (Vrai positifs) sont les aspects candidats correctement identifiés
- FP (Faux positifs) sont les aspects candidats incorrectement identifiés
- FN (Faux négatifs) sont les aspects candidats incorrectement non-identifiés
- VN (Vrai négatifs) sont les aspects candidats correctement non-identifiés

Et:

Précision= VP/(VP+FP)

Rappel=VP / (VP+FN)

a) Niveau des fragments

Les résultats obtenus au niveau des fragments de l'analyse statique et dynamique et de leur union sont organisés dans la table 3.

		Réel					
		Statique		Dynamique		Union	
		Positive	Négative	Positive	Négative	Positive	Négative
Prédit	Positive	59	0	47	0	59	0
	Négative	0	-	12	-	0	-

Table 3.La matrice de confusion des analyses au niveau des fragments

Après avoir compté manuellement les fragments du code responsable de l'implémentation de la préoccupation transverse de MatchMaker , nous avons trouvé 59 fragments.
Dans la table 4, nous allons calculer la précision et le rappel de notre approche à ce niveau de fragments.

Analyse	Précision	rappel
Statique	100% (59/59)	100% (59/59)
Dynamique	100%(47/47)	79,6% (47/59)
Union	100% (59/59)	100% (59/59)

Table 4.Précision et rappel des analyses au niveau des fragments.

Dans l'analyse statique les résultats montrent que tous les aspects candidats identifiés sont liés à la préoccupation transversale matchmaker (précision de 100%), et aucun aspect candidat réel a été manqué (rappel de 100%). Dans l'analyse dynamique tous les résultats

sont liés au matchmaker avec une précision de 100%, mais seulement 79,6% de l'ensemble réel des aspects candidats ont été identifiés, ce qui signifie que 20,4% des aspects candidats réels n'ont pas été identifiés, la cause est que les fragments manqués n'ont pas été exécutés, et pour améliorer la précision et le rappel nous fusionnons leurs résultats.

b) Niveau des Méthodes

Les résultats obtenus au niveau des méthodes (analyse statique) sont présentés dans la table 5 où plusieurs seuils ont été utilisés, nous présentons quatre parmi eux seulement où nous avons obtenu des résultats différents (15, 25, 60 et 70%).

		Real							
		15%		25%		60%		70%	
		P	N	P	N	P	N	P	N
Prédit	Positive	6	4	6	1	6	0	0	0
	Négative	0	-	0	-	0	-	6	-

Table 5.La matrice de confusion des analyses au niveau des méthodes.

Notre cas d'étude a 06 véritables méthodes qui implémentent la préoccupation transverse de MatchMaker. Comme présenté dans la table 6, en utilisant 15% comme seuil des types simples utilisés dans les déclarations méthodes, nous avons identifiés 10 méthodes où 04 parmi elles sont des faux positifs et les 6 autres sont de vrais positifs qui nous donnent une précision de 60% et un rappel de 100%

Threshold	Precision	recall
15%	60%(6/10)	100% (6/6)
25%	85,7% (6/7)	100% (6/6)
60%	100% (6/6)	100% (6/6)
70%	-	0% (0/6)

Table 6.Précision et rappel au niveau des Méthodes.

Le seuil de 25% réduit les faux positifs à une seule méthode (Figure 20) où les autres ont été correctement identifiés avec une précision de 85,7%, le seuil de 60% est le meilleur dans ce cas d'étude avec une précision et un rappel de 100%.

```
protected void takeDown() {
      try {
                  DFService.deregister(this);
        } catch (FIPAException fe) {
                  fe.printStackTrace();
      }
System.out.println("El agente "+getAID().getName()+"
ha terminado");
           }
```

Figure 20.Un faux positif au niveau des méthodes.

En utilisant le seuil de 70% nous ne pouvions pas identifier des méthodes qui implémentent le Matchmaker.

Dans notre approche, la qualité des résultats (précision et rappel) et le temps d'analyse sont pris en considération. Pour ces deux, nous avons combiné : (1) le principe de navigateurs dédiés à la phase de sélection API où l'avantage est la haute précision et rappel et (2) l'identification automatisée en utilisant le plug-in AspectJ qui offre une vitesse et une excellente efficacité c'est à dire que si la graine est déclarée dans la coupe alors elle sera identifiée.

5. Conclusion

Dans ce chapitre nous avons présenté l'outil que nous avons développé pour supporter notre approche d'extraction d'aspects dans les SMA. Nous avons présenté tous les cas possibles des analyses statique et dynamique tout en justifiant le choix d'ordre des étapes. L'utilisation de catalogues des graines et AspectJ pour identifier les aspects candidats nous a offert une bonne précision et un bon rappel dans un temps d'analyse réduit et une participation minime de l'utilisateur.

Conclusion Et Perspectives

Conclusion et Perspectives

L'extraction d'aspects dans applications multi-agents est un nouveau domaine de recherche qui n'est pas encore exploré. Seulement quelques propositions d'architectures ont été proposées pour la séparation des préoccupations d'agents dans les étapes préliminaires de cycle de développement dans la littérature. Dans ce mémoire, nous avons présenté une nouvelle approche hybride et semi-automatique pour l'extraction d'aspects dans les SMAs qui combine les techniques d'analyse statique et dynamique. L'analyse statique est utilisée au niveau des granularités des méthodes et des fragments, alors que l'analyse dynamique est utilisée au niveau des fragments. Notre approche est supportée par un outil visuel appelé MAMIT (Mas Aspect MIning Tool). En utilisant l'outil développé, nous pouvons importer une application à l'explorateur de paquetages, extraire l'API de la plate-forme SMA choisie, puis nous sélectionnons les éléments Java liés à la préoccupation transversale qui nous désirons identifier. Ensuite, nous performons l'analyse statique d'abord, puis l'analyse dynamique, nous fusionnons les aspects candidats identifiés au niveau des fragments puis nous filtrons automatiquement les résultats pour éviter la duplication. Finalement, nous pouvons inspecter les aspects candidats identifiés manuellement avant de les enregistrer. Nous avons validé notre outil sur une application multi agent développée sous JADE nommé « Agentes09 » qui simule un système d'enchère. Les résultats obtenus sont très satisfaisants et montrent une très bonne précision des aspects candidats avec une participation minime de l'utilisateur.

Comme perspective à moyen termes, nous prévoyons d'étendre notre approche et notre outil MAMIT pour : (1) tenir compte des autres préoccupations transversales inhérentes au SMA, et (2) implémenter un processus de restructuration (en anglais *refactoring*) des applications multi-agents.

Bibliographie

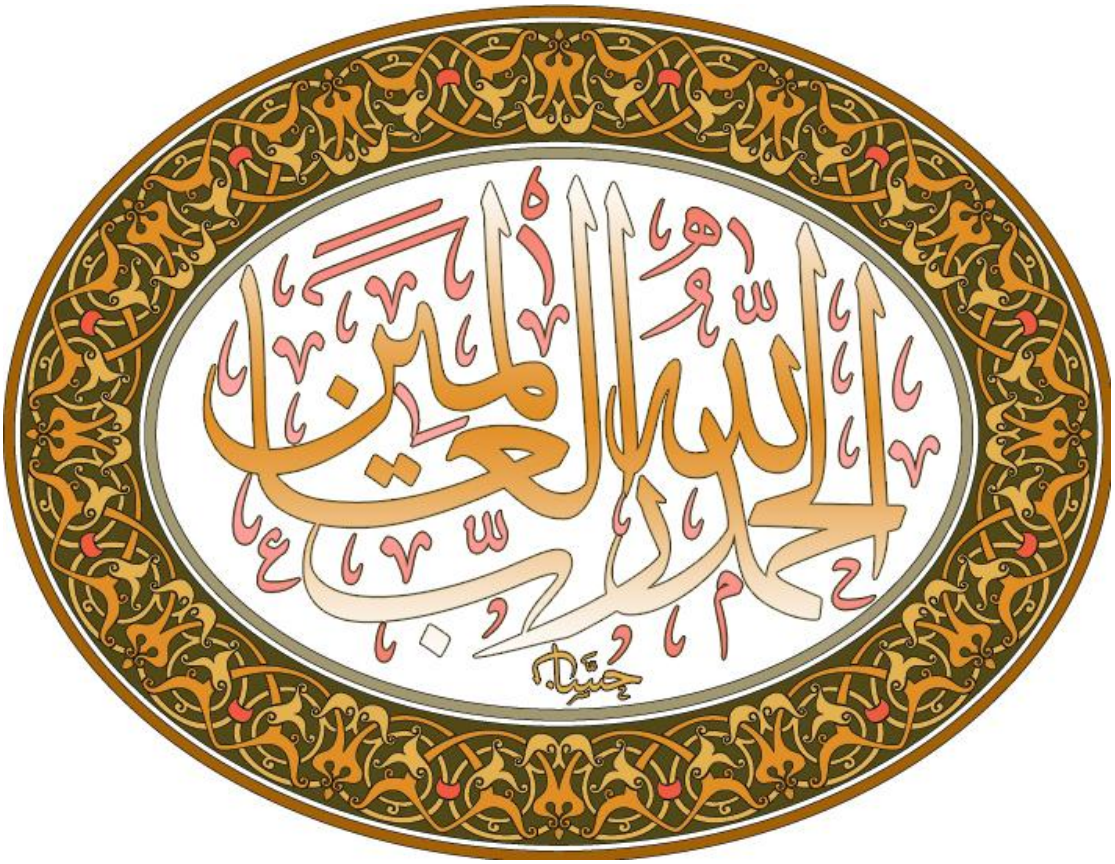
- ***[Abb12]*** *A.R. Abbasi. "Automated Separation of Crosscutting Concerns: Earlier Automated identification and Modularization of Cross-Cutting Features at Analysis Phase" , Multitopic Conference (INMIC), 15th International (2012).*
- ***[Bak95]*** *B. Baker, "On finding duplication and near duplication in large software systems". In: Working Conference on Reverse Engineering (WCRE 1995), IEEE Computer Society Press 86–95 ,1995*
- ***[Bal02]*** *J. Baltus, « La Programmation Orientée Aspect et AspectJ: Présentation et Application dans un Système Distribué. » Facultés Universitaires Notre-dame de la Paix, Namur, Belgique. 2002*
- ***[Bax98]*** *I. Baxter, A. Yahin , L. Moura, M. Sant' Anna, , L . Bier, « Clone detection using abstract syntax trees ». In: International Conference on Software Maintenance (ICSM 1998), IEEE Computer Society Press 1998*
- ***[Bel07]*** *F. Bellifemine, G. Caire, D. Greenwood, "Developing Multi-Agent Systems with JADE" (Wiley Series in Agent Technology) 2007.*
- ***[Bell00]*** *F. Bellifemine, C. Giovani, G. Rimassa., "Jade Programmer's Guide" Jade version 2.6 (http://sharon.cselt.it/projects/jade/), 2000.*
- ***[Bell99]*** *F. Bellifemine, A. Poggi, G. Rimassa, "JADE -- A FIPA-compliant agent framework", CSELT internal technical report. Part of this report has been also published in Proceedings of PAAM'99, London, pp.97-108, April 1999.*
- ***[Boi04]*** *O. Boissier, S. Gitton, P. Glize, « Caractéristiques des Systèmes et des applications. Systèmes Multi-Agents », Observatoire Français des Techniques Avancées, ARAGO 29, Diffusion Editions TEC & DOC, p. 25-54, 2004.*
- ***[Bre04a]*** *S. Breu, J. Krinke, "Aspect mining using event traces". In: Conference on Automated Software Engineering (ASE). 2004*
- ***[Bre04b]*** *S. Breu. "Towards hybrid aspect mining: Static extensions to dynamic aspect mining". In 1st Workshop on Aspect Reverse Engineering, 2004.*
- ***[Bru04a]*** *M. Bruntink, A.v. Deursen, R.v. Engelen, T. Tourwé, " An evaluation of clone detection techniques for identifying crosscutting concerns" In: International Conference on Software Maintenance (ICSM 2004), IEEE Computer Society Press, 2004*
- ***[Bru04b]*** *M. Bruntink "Aspect mining using clone class metrics". In: 1st Workshop on Aspect Reverse Engineering. 2004*

- ***[Bru05]*** *M. Bruntink, A.v. Deursen, , R.v. Engelen, , T. Tourwé, "On the use of clone detection for identifying crosscutting concern code". IEEE Transactions on Software Engineering 31(10) 804–818, 2005*
- ***[Bub01]*** *A. Budanitski. "Semantic distance in wordnet: an experimental,application-oriented evaluation of five measures". 2001.*
- ***[Cec05]*** *M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonello, T. Tourwé. "Aqualitative comparison of three aspect mining techniques". In Inter-national Workshop on Program Comprehension (IWPC), 2005.*
- ***[Eic92]*** *S. Eick, J. Steffen, E. Summer, "Seesoft - A Tool For Visualizing Line Oriented Software Statistics", IEEE TSE, vol. 18, no. 11, pp. 957-968, November 1992.*
- ***[Fer05]*** *A. Ferguen, « La plate-forme JADE », 2005.*
- ***[Fer95]*** *J. Ferber, « les systèmes multi agents : vers une intelligence collective ». InterEditions, 1995.*
- ***[Gan99]*** *B. Ganter, R. Wille. "Formal Concept Analysis: Mathematical Foundations". Spring-Verlag, 1999.*
- ***[Gar02]*** *A. Garcia, V. da Silva, C. Chavez, C. Lucena, "Engineering Multi-Agent Systems with Patterns and Aspects", Journal of the Brazilian Computer Society, SBC, Special Issue on Software Engineering and Databases, September 2002.*
- ***[Gar04a]*** *A.Garcia, U. Kulesza, C. Lucena. « Aspectizing Multi-Agent Systems: From Architecture to Implementation". "Software Engineering for Multi-Agent Systems III". pp. 121-143. Springer-Verlag, LNCS 3390, December 2004*
- ***[Gar04b]*** *A. Garcia, C. Anna, C. Chavez, V. da Silva, C. Lucena, A. Staa, "Separation of concerns in multi agent systems: An empirical study, Software Engineering for Multi-Agent Systems" II Lecture Notes in Computer Science Volume 2940, pp 49-72, 2004*
- ***[Gar04c]*** *A. Garcia, U. Kulesza, C. Anna, C. Lucena. «The Mobility Aspect Pattern". Fourth Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'04, Fortaleza, Brazil.2004*
- ***[Gar05]*** *A. Garcia, U. Kulesza, C. Chavez, C. Lucena, "The interaction aspect pattern", In proceeding of: EuroPLoP' 2005, Tenth European Conference on Pattern Languages of Programs, Irsee, Germany, July 6-10, 2005*
- ***[Gar06]*** *A. Garcia, C. Chavez, R. Choren, "An Aspect-Oriented Modeling Framework for MAS Design". 7th Workshop on Agent-Oriented Software Engineering, AAMAS'06, Hakodate Japan, May 2006.*
- ***[Gra03]*** *J.D. Gradecki, N. Lesiecki, "Mastering AspectJ Aspect-Oriented Programming in Java". 2003*

- ***[Gyb04]*** *K. Gybels, A. Kellens. "An experiment in using inductive logic programming to uncover pointcuts". In First European Interactive Workshop on Aspects in Software,2004.*
- ***[Gyb05]*** *K. Gybels, A. Kellens. "Experiences with identifying aspects in smalltalk using unique methods". InWorkshop on Linking Aspect Technology and Evolution, 2005.*
- ***[He04]*** *L. He, H. Bai. "Aspect mining using clustering analysis". Technical report, Jilin University, 2004.*
- ***[Hon05]*** *T. Hon, M. Tkatchenko. "Refactoring JQuery with AspectJ: an experience report". CPSC 511 Project Report.April 29, 2005.*
- ***[Jen94]*** *N.R. Jennings, M. Wooldridge, J. Michael, " Intelligent Agents" , ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, The Netherlands, August 8 - 9, Proceedings 1994.*
- ***[Kar98]*** *S. Karanjkar. "Development of graph clustering algorithms". Master's thesis, University of Minnesota, 1998.*
- ***[Keb10].*** *S. Kebir, « "Programmation orientée aspect en Java avec AspectJ ». 2 février 2010*
- ***[Kel05]*** *A. Kellens, K. Mens. "A Survey of Aspect Mining Tools and Techniques". June 30, 2005.*
- ***[Kic97]*** *G. Kiczales, J. Lamping, A. Mendhekar,C. Maeda, C. Lopes, J. Loingtier, J. Irwin, "Aspect-Oriented Programming". In Proc. of European Conference on Object-Oriented Programming. ECOOP'97, Finland. Springer-Verlag. 1997*
- ***[Kol02]*** *M. Kolp, P. Giorgini, J. Mylopoulos, "Information Systems Development through Social Structures".14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ishia, Italy, 2002.*
- ***[Kol05]*** *M. Kolp, T. T. Do, S. Faulkner, H. T. T. Hoang, "Introspecting Agent Oriented Design Patterns", In: S. K. Chang Introspecting Agent Oriented Design Patterns, In: S. K. Chang (Eds), Advances in software Engineering and Knowledge Engineering, vol. III, World Publishing, 2005.*
- ***[Kom01]*** *R. Komondoor, S. Horwitz, "Using slicing to identify duplication in source code". In: International Symposium on Static Analysis, Springer-Verlag 40–56. 2001*
- ***[Kri01]*** *J. Krinke, " Identifying similar code with program dependence graphs". In:Working Conference on Reverse Engineering (WCRE'01), IEEE Computer Society Press 301–309. 2001*
- ***[Kri04]*** *J. Krinke, S. Breu. "Control-flow-graph-based aspect mining". In 1st Workshop on Aspect Reverse Engineering, 2004.*
- ***[Kul04]*** *U. Kulesza, A. Garcia, C. Lucena. "Generating Aspect-Oriented Agent Architectures". Proceedings of the 3rd Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design, 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK 2004*

- ***[Lad03]*** *R. Laddad. "AspectJ in Action: Practical Aspect-Oriented Programming". Manning Publications, 2003.*
- ***[Lad09]*** *R. Laddad, "AspectJ in Action, Second Edition Enterprise AOP with Spring Applications", September, ISBN: 1-933988-05-3. 2009*
- ***[Lob04]*** *C. Lobato, A. Garcia, A. Romanovksy,C. Sant'Anna, U. Kulesza, C. Lucena. "Mobility as an Aspect: The AspectM Framework". Proceedings of the 1st Brazilian Workshop on Aspect-Oriented Software Development – WASP'04, SBES'04, Brasília, Brazil, October 2004.*
- ***[Mar04]*** *M. Marin, A. van Deursen, L. Moonen. « Identifying aspects using fan-in analysis". In Working Converence on Reverse Engineering (WCRE), 2004.*
- ***[Men05]*** *K. Mens, T. Tourwé. "Delving source-code with formal concept analysis". Elsevier Journal on Computer Languages, Systems & Structures,2005.*
- ***[Mok08]*** *F. Mokhati, M. Badri, L. Badri, F. Hamidane, S. Bouazdia: "Automated testing sequences generation from AUML diagrams: a formal verification of agents' interaction protocols". IJAOSE 2(4): 422-448 2008.*
- ***[Mor91]*** *J. Morris, G. Hirst. "Lexical cohesion computed by thesaural relations as an indicator of the structure of text". Comput. Linguist.,17(1):21–48, 1991.*
- ***[Noe07]*** *V. Noel, F. Cassignol, « TER : JADE et Gorgias », 21 mai 2007*
- ***[Par12]*** *P. Parreira, W. Mendes, V. de Camargo, R. A. D. Penteado, H. A. Xavier Costa, "Mining Crosscutting Concerns with ComSCId: A Rule-Based Customizable Mining Tool", Informatica (CLEI), XXXVIII Conferencia Latinoamericana En 2012*
- ***[Paw04].*** *R. Pawlak, J. Retaillé, L. Seinturier, « Programmation Orientée Aspect pour Java/J2EE » ISBN: 2-212-11408-7. 2004*
- ***[Pél02]*** *L. Pélissier, R. Lhoste, « Jade, Java Agent DEvelopment framework » 2002*
- ***[Ran12]*** *R. Rand, M. F. J. Mitropoulos, "Aspect Mining Using Model-Based Clustering", Southeastcon, Proceedings of IEEE 2012*
- ***[Rob02]*** *M. Robillard, G. Murphy. Capturing Concern Descriptions During Program Navigation. A position paper for the OOPSLA Workshop on Tool Support for Aspect Oriented Software Development. 2002*
- ***[Sar04]*** *J. Sardinha, A. Garcia, R. Milidiú, C. Lucena. "The Learning Pattern. Fourth Latin American Conference on Pattern Languages of Programming", Sugar Loaf PLoP'04, Fortaleza, Brazil. 2004*
- ***[Seg04]*** *A. Seghrouchni , I. Cartault. Modelling, " Control and Validation of Multi-agent plans in Dynamic Context". AAMAS'04, 19-23, New York, USA. 2004*
- ***[She04]*** *D. Shepherd, E. Gibson, L. Pollock. "Design and evaluation of an automated aspect mining tool". In International Conference on Software Engineering Research and Practice, 2004.*

- ***[She05a]*** *D. Shepherd, T. Tourwé, L. Pollock. "Using language clues to discover crosscutting concerns". In Workshop on the Modeling and Analysis of Concerns, 2005.*
- ***[She05b]*** *D. Shepherd,L. Pollock. "Interfaces, aspects and views. In Linking Aspect Technology and Evolution" (LATE) Workshop, 2005.*
- ***[Sil06]*** *C. Silva, J. Araújo, A. Moreira, J. F. B. Castro, D. Penaforte, A. Carvalho, "Towards an Aspect Oriented Modeling in Multi-agent Systems", Workshop on AOSD, WASP'06, 20th Brasilian Symposium on Software Engineering (SBES'06), Florianópolis, Brazil, 2006.*
- ***[Sil09]*** *C. Silva, M. Lucena, J. Castro, J. Araujo, A. Moreira, F. Alencar, "Support for aspectual modeling to Multiagent system architecture Aspect-Oriented Requirements Engineering and Architecture Design", 2009*
- ***[Tah99]*** *Y. Tahara, A. Ohsuga, S. Honiden, "Agent system development method based on agent patterns". Proceedings of the 21st International Conference on Software Engineering, Los Angeles, California, United States. pp.: 356 –367. ISBN: 1-58113-074-0 1999*
- ***[Tha08]*** *R.P. Thao, « Amélioration de la communication dans un système multi-agents perturbé », Institut de la Francophonie pour l'Informatique, 2008*
- ***[Ton04]*** *P. Tonella, M. Ceccato. "Aspect mining through the formal concept analysis of execution traces". In 11th IEEE Working Conference on Reverse Engineering, 2004.*
- ***[Tou04a]*** *T. Tourwé, K. Mens. "Mining aspectual views using formal concept analysis". In Proc. of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004). IEEE Computer Society, September 2004.*
- ***[Tou04b]*** *T. Tourwé, K. Mens. "Mining aspectual views using formal concept analysis". In Source Code Analysis and Manipulation Workshop(SCAM), 2004.*
- ***[Woo97]*** *M. Wooldridge. "An Introduction to Multiagent Systems". John Wiley and Sons, 1997.*
- ***[Zha12]*** *C. Zhang, H.A. Jacobsen, "Mining Crosscutting Concerns through Random Walks". IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 38, NO. 5,2012*
- ***[1]*** *http://code.google.com/p/agentes09/*
- ***[2]*** *http://www-igm.univ-mlv.fr/~dr/XPOSE2005/gmasquelier/contents/tisseur_aspect.html*
- ***[3]*** *http://www.technoscience.net/?onglet=glossaire&definition=5385*
- ***[4]*** *AspectBrowser for Eclipse, http://www.cs.ucsd.edu/users/wgg/Software/AB/.*
- ***[5]*** *The Aspect Mining Tool, http://www.cs.ubc.ca/labs/spl/projects/amt.html.*
- ***[6]*** *http://www.eclipse.org/ajdt/*


الحمد لله رب العالمين