Mémoire Présenté
Par
Tony Garneau

à l'Université du Québec à Trois-Rivières

Comme exigence partielle de la
Maîtrise en mathématiques et informatique appliquées

La Programmation Orientée-Agent

DMAS Builder : Un Environnement de Développement De Systèmes Multi-Agents Totalement Distribués

Département de mathématiques et d'informatique
Université du Québec à Trois-Rivières

Août 2003

Résumé

Ce document est le résultat de deux années de travail intensif sur les SMA (Systèmes Multi-Agents). Dans un premiers temps, nous avons étudié les travaux effectués sur les méthodologies de développement de SMA. Les concepts de programmation OA (Orientée-Agent) et de SMA sont des idées très intéressantes et les méthodologies développées fournissent des patrons théoriques pour la modélisation de ceux-ci. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont souvent difficiles à implémenter directement à partir des langages standards de programmation, comme Java ou C++. C'est pourquoi plusieurs méthodologies ont mené à la création d'outils d'aide au développement de SMA. De plus, plusieurs autres équipes de chercheurs axent leurs travaux dans l'élaboration de solutions quant au support pour le développement de SMA, étant donné la problématique d'application de ce type de méthodologie.

Nos intérêts étant beaucoup plus au niveau applicatif que théorique, il était logique que nous suivions cette tangente. Nous nous sommes donc intéressés aux outils d'aide au développement de SMA. Par la suite, nous les avons évalués et comparés [Garneau02]. Cette évaluation fût faite dans l'optique de trouver les forces et les faiblesses chroniques des outils par rapport à un environnement de développement complet de développement de SMA. Les conclusions de cette évaluation sont claires et non-ambiguës. En effet, plusieurs lacunes chroniques sont présentes dans la majorité des ED (Environnement de Développement) de SMA, ce qui rend leur utilisation inappropriée voir même impossible pour le développement de systèmes réels d'envergure intéressante.

Finalement, nous avons conçu un environnement complet de développement de SMA totalement distribués répondant [Garneau03], en grandes parties, aux exigences nécessaires identifiées dans notre étude comparative et dans ce document. DMAS Builder permet :

- Le développement simple et rapide d'applications distribuées.
- La spécification, via des interfaces, des différents sous-systèmes, de leurs agents, des bases de connaissances, des modes de communication entre les différentes machines et autres.
- Le développement OA grâce à la librairie de classes OA (environ 200 classes) intégrée.
- L'extensibilité des classes de la librairie OA.
- La validation des spécifications.
- La génération complète du code source des composants du système.
- L'exportation du système permettant le développement et l'exécution du système indépendant de l'outil.
- L'archivage des différents sous-systèmes et la création d'un exécutable pour chaque JVM (Java Virtual Machine). Ceci permet un déploiement extrêmement simple d'applications totalement distribuées.

- Plusieurs autres options non-disponibles sur les autres environnements de développement (orienté-objet ou OA) simplifiant la préparation et l'initialisation de l'outil, comme la recherche automatique des programmes nécessaires à l'ED (*java, javaw, rmic, javac, jar* et autres), l'initialisation automatique du *classpath* et du *sourcepath*.
- L'exécution de l'environnement sur différentes plate-formes (Windows, Unix, Linux, etc.) sans aucune modification.
- L'exécution des sous-systèmes (fichier exécutable « .jar ») sur les différentes plate-formes (Windows, Unix, Linux, etc.) sans aucune modification.

L'environnement que nous avons développé (DMAS Builder) est implémenté à 100% en Java. Nous sommes sur le point d'offrir la première version de l'outil. Celle-ci comprendra l'environnement de développement, la documentation et les exemples permettant une meilleure compréhension. Nous sommes très optimistes quant à l'utilité de l'environnement qui sera le premier à offrir ce genre de services pour le développement et le déploiement de SMA.

Abstract

This paper is a logical extension of our recent research on MAS. Initially we studied MAS development methodologies. Then, we considered MAS development tools and environments that we evaluated and compared. This comparative evaluation was carried out in order to identify tools' and environments' strong points, as well as weak points, in the hope of determining what a complete development environment should look like. Finally, we undertook the development of a complete MAS development environment essentially meeting the set of requirements identified in our comparative study. DMAS Builder offers several major advantages compared with most other existing tools. Indeed, DMAS Builder:

- Graphic specification of all MAS components: sub-systems, agents, knowledge bases, tasks, behaviours, communication modes between agents, etc.
- Packages of agent-oriented classes (about 200 classes) allowing agent-oriented development.
- Specifications validation.
- A complete source code generation (in Java) of all system components.
- An exportation mechanism allowing system development and execution independent of DMAS Builder.
- An archive mechanism allowing the creation of an executable archive for each sub-system (JVM). This allows very simple distributed application deployment.
- Several other options not available on standard development environments (OO or AO), thus simplifying tool preparation and initialization, such as automatic search of the necessary programs (java, javaw, rmic, javac, jar and others), automatic "classpath" and "sourcepath" initialization and much more.
- The environment execution on various operating systems (Windows, Unix, Linux) without any modification.
- Sub-systems deployment (executable jar files) on different operating systems without any modification.
- To develop MAS with OA classes and Java standards API
- To use an integrated help engine to access all OA and standards Java classes documentation specifications.

Everything related to our MAS development environment/tool in this paper is 100% implemented in Java. We are about to offer the first version of our development environment. This one will include the development environment, documentation and examples allowing an easier understanding of the tool's numerous capabilities. We are very optimistic as to the future of such a tool: as far as we know, it is the first to offer such a package of services for MAS development and deployment.

Dédicaces

Je dédie ce mémoire à ma mère,
Denise Deveault

et

à mon père décédé le 2 janvier 2002,
Gaston Garneau

Remerciements

Je tiens à remercier les personnes suivantes :

Mon directeur de mémoire, Sylvain Delisle, professeur d'informatique au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Je te remercie pour ton immense patience, ta disponibilité, ta grande implication ainsi que pour ton énorme soutien. Sans toi, la réalisation de ce mémoire n'aurait pas eu lieu. Encore une fois, merci beaucoup.

Bernard Moulin, professeur d'informatique au département d'informatique de l'Université Laval pour la poursuite de mes études au doctorat.

François Meunier et Louis Paquette, professeurs d'informatique au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Malgré les chaleurs de l'été, merci d'avoir pris quelques heures pour évaluer mon mémoire. Ce geste fut grandement apprécié.

Chantale Lessard, notre secrétaire au département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Merci pour ta disponibilité, ton aide omniprésente et pour tous les services que tu m'as rendus.

Mes professeurs du département de mathématiques et d'informatique de l'Université du Québec à Trois-Rivières. Merci.

Ma mère Denise, son mari Alain ainsi que toute ma famille pour leur soutien tout au long de mes études. Durant mon cheminement scolaire, votre appui fut très important pour moi. Merci beaucoup.

À mes plus fidèles amies et amis. Je prend le temps de vous remercier tous et toutes individuellement pour votre soutien, votre aide et les services que vous m'avez rendus. Je vous en serais éternellement reconnaissant.

Au CRSNG qui, grâce à son support financier, me permet de poursuivre mon cheminement scolaire.

Aux personnes travaillant au service des prêts et bourses de l'Université du Québec à Trois-Rivières. Mes visites furent très fréquentes et je tenais à vous remercier pour votre service toujours impeccable.

Merci 

Table des matières

Liste des figures

Acronymes, références et formatage du texte

Dans le but d'alléger le texte, plusieurs acronymes connus sont utilisés tout au long de ce document. Leur signification est donnée ci-dessous.

Acronyme	*Signification*
ACC	Agent Communication Chanel
ACL	Agent Communicating Language
AMS	Agent Management System
ANS	Agent Name Server
API	Application Programming Interface
API	Application Programming Interface
AWT	Abstract Window Toolkit
BDI	Beliefs Desires and Intentions
CORBA	Common Object Request Brokerage Architecture
CPU	Central Unit Processing
DF	Directory Facilitator
DF	Directory Facilitator
DMAS Builder	Distributed Multi-Agents System Builder
ED	Environnement de Développement
FIPA	Foundation For Intelligent Physical Agent
FIPA ACL	Foundation For Intelligent Physical Agent Agent Communicating Language
GUID	Global Unique IDentifier
JAL	Jack agent language
JIT	Just In Time compiler
JRE	Java Runtime Environment
JVM	Java Virtual Machine
KIF	Knowledge Interchange Format
KQML	Knowledge Query Manipulation Language
KS	Knowledge Source
LHS	Left Hand Side
MAS	Multi-Agents System
OA	Orienté-Agent
OMT	Object Modeling Technique
OO	Orienté-Objet
PLACA	PLAnning Communicating Agents
PRS	Procedural Resonning System
RA	Register Agent
RHS	Right Hand Side
RMI	Remote Method Invocation
SDK	Standard Development Kit
SMA	Système Multi-Agents
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unifed Modeling Language

Références

Deux types de références se retrouvent dans le texte. Les références à un document se retrouvent sous le format [référence]. Les références à des sites Web se retrouvent sous le format <référence>.

Utilisation de l'anglais

De plus, étant donné que plusieurs termes anglais sont difficilement traduisibles en français (ou que le mot ou l'expression traduit serait plus difficile à comprendre en français), quelques mots et expressions anglaises seront laissées intégrales en anglais. Cependant, le texte de ceux-ci sera en italique. *Mot ou expression en anglais difficilement traduisible ou plus facilement compréhensible en anglais.*

1. Introduction

La programmation a beaucoup évolué ces dernières années. Il n'y a pas si longtemps, on ne parlait que de programmation procédurale. Les besoins en termes de logiciels changent rapidement et ces derniers sont de plus en plus présents dans la vie de tous les jours. Les logiciels deviennent plus complexes et l'effort de conception de ceux-ci est en constante croissance. Les techniques de programmation devaient donc s'adapter et c'est, entre autres, ce qui a mené à la programmation OO (orientée-objet). La programmation OO et ses techniques de modélisation, comme OMT (Object Modeling Technique), permettent d'enrichir la programmation procédurale en y ajoutant des concepts essentiels comme l'encapsulation, l'abstraction et le polymorphisme.

Depuis quelques années, une évolution de celle-ci se dessine à l'horizon : la programmation OA. En effet, la programmation OO n'est pas toujours adaptée aux besoins des applications d'aujourd'hui. Les systèmes sont souvent distribués sur plusieurs machines. Ils interagissent entre eux et doivent s'exécuter indépendamment les uns des autres. Aussi, ils doivent communiquer entre eux. De plus, ils sont souvent divisés en sous-systèmes indépendants et chacun d'eux exécutent une partie du travail dans un but commun. La programmation OA offre une façon beaucoup plus naturelle de concevoir ce genre de système. Elle fut proposée pour la première fois par Yoav Shoham en 1993 [Shoam93] comme étant un nouveau paradigme de programmation basé sur une vue « sociétale de la programmation » et dont un des aspects fondamentaux est la programmation des agents en termes de notions mentales comme celles qu'impliquent les croyances, désirs et intentions des architectures BDI (*Beliefs, Desires and Intentions*).

1.1 Les agents

La définition d'un agent est probablement le sujet qui a donné lieu au plus de discussions dans les communautés étudiant le domaine. Cependant, la plupart des chercheurs s'entendent aujourd'hui pour dire qu'un agent est une entité (programme ou partie de programme) possédant un certain degré d'autonomie, de l'intelligence (bases de connaissances ou autres) et des buts bien définis. Un agent est situé, c'est-à-dire qu'il possède une représentation de son environnement. Il est donc capable d'interagir avec son environnement et, par le fait même, d'avoir une influence sur ce dernier. Enfin, il est capable de communiquer avec les autres composants du système.

1.2 Les systèmes multi-agents

Un SMA (système multi-agents) consiste en un ensemble d'agents interagissant dans le but d'atteindre un objectif global. Un SMA est un système où plusieurs agents coopèrent pour effectuer des tâches qui seraient a priori très difficiles voir mêmes impossibles de mener à terme par un seul agent. L'accomplissement de ces tâches tend vers la résolution d'un problème précis. Il est évident que le développement de systèmes où les

interactions, les communications et la coordination entre les divers composants sont essentiels, est significativement plus complexe que l'implémentation d'agents individuels.

L'émergence de ce nouveau paradigme de programmation a donné lieu à l'élaboration de plusieurs méthodologies et architectures pour la modélisation des SMA. Le concept de programmation OA est une idée très intéressante et les méthodologies développées fournissent des patrons théoriques pour la modélisation des SMA. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont difficiles à implémenter directement avec des langages de programmation standards, comme Java ou C++. En effet, les modèles utilisent des concepts et des schémas complexes. Ceux-ci sont la plupart du temps divisés en plusieurs couches conceptuelles difficilement séparables, ce qui rend les méthodologies difficilement transférables du modèle à l'implémentation (programmation). Il en résulte souvent un effort de programmation beaucoup trop élevé pour l'apport de l'utilisation de l'approche SMA. Il faut donc fournir aux programmeurs des outils leur permettant de développer de façon plus simple, rapide et efficace des applications utilisant ce nouveau paradigme de programmation (et les méthodologies s'y rattachant).

Récemment, plusieurs outils de différents types ont été développés pour la programmation OA. On peut considérer comme outil OA les éléments logiciels offrant des services pour le développement de SMA. Parmi ceux-ci, on retrouve les librairies de classes fournissant des fonctionnalités se rattachant aux SMA : définition des fonctionnalités de bases des agents, mécanismes d'interaction, de coordination, de communication, etc. On y retrouve aussi les ED, complets ou partiels, qui aident le programmeur dans l'élaboration et l'implémentation d'un SMA. Par contre, ces derniers sont incomplets et possèdent de grandes lacunes. Ils sont la plupart du temps inutilisables ou de faibles intérêts pour le développement de systèmes relativement complexes. Il faut donc fournir aux programmeurs un environnement qui leur donne des avantages significatifs et évidents tout en minimisant, ou idéalement en éliminant, les contraintes de l'utilisation de celui-ci.

1.3 Objectifs

Le principal objectif est de concevoir un environnement complet de développement de systèmes multi-agents totalement distribués. L'environnement développé permettra de combler plusieurs lacunes chroniques de la majorité des outils d'aide au développement de SMA existants. ***DMAS Builder*** permettra :

- De simplifier le développement de SMA totalement distribués.
- D'accélérer le développement de SMA totalement distribués.
- De développer des systèmes relativement complexes.
- Une bonne extensibilité du code.
- Le développement indépendant de la plate-forme.
- D'abstraire les mécanismes de communication.

- La création d'applications indépendantes de l'environnement.
- Le déploiement simple.

1.4 Survol

Le document est divisé de la façon suivante. Dans un premier temps, la problématique de l'approche agent est abordée. Dans cette section, différents problèmes reliés aux SMA et à la programmation OA sont abordés. Il y est aussi question des insuffisances des techniques de programmation actuelles. Le chapitre 3 se veut une revue de littérature survolant l'univers des agents et des SMA. Cette section met en relief les principaux travaux du domaine, c'est-à-dire les principales méthodologies de développement, architectures, outils, langages de communication, techniques de coordination, bases de connaissances et autres. Le chapitre suivant donne une description générale de la solution proposée. Le chapitre 5 donne une explication détaillée de l'environnement développé (DMAS Builder). Des exemples de l'utilisation de l'outil se retrouvent dans le chapitre 6. Le chapitre 7 expose des mises en situation pour l'utilisation de l'ED et une grille d'évaluation de l'outil pouvant être faite par les pairs. Enfin, une brève conclusion termine le document.

2. Problématique

Depuis une dizaine d'années, une évolution phénoménale s'est effectuée dans le monde de l'informatique. Les ordinateurs sont de plus en plus performants. Ces derniers sont accessibles à tous et de plus en plus de gens en possèdent un. Internet est devenu du même coup un mode de vie pour plusieurs d'entre nous. L'avènement d'Internet a fait augmenter de façon exponentielle la disponibilité de l'information. Ces informations sont souvent distribuées et hébergées sur des serveurs hétérogènes (systèmes d'exploitation et langages de programmation différents). De ce fait, les programmes traitant et recueillant ces informations sont souvent inadaptés. Internet est sans contredit un des grands responsables de l'émergence de la programmation OA. Il faut trouver une façon plus naturelle de concevoir des systèmes divisés en sous-systèmes sur différentes machines coopérants entre eux pour effectuer un travail précis. De cette façon, on allège la tâche de chacun en déléguant, ce qui permet de résoudre des problèmes de plus grande envergure et ce, de façon plus efficace.

2.1 Les systèmes d'aujourd'hui

2.1.1 Complexité des systèmes d'aujourd'hui

Comme nous l'avons mentionné un peu plus haut, les systèmes d'aujourd'hui sont souvent très complexes. Ils traitent une quantité énorme d'informations et effectuent des calculs très lourds. On n'a qu'à penser aux simulations à grande échelle, au traitement d'images, au traitement de signaux, la classification et la catégorisation (textes, images, sons, paroles, etc.), le traitement de multiples requêtes par un serveur Web, le forage de données (Data Mining) et la recherche d'information (agents informatifs). Ce ne sont que quelques exemples où les capacités des machines et de leurs programmes sont souvent utilisées à leurs limites. Lorsque les algorithmes sont optimisés, que le CPU et la mémoire sont utilisés à leurs maximums, les avenues ne sont pas nombreuses pour améliorer la performance des programmes. On peut acheter des ordinateurs plus puissants, ce qui impliquent la plupart du temps des investissements énormes. Cependant, il reste une alternative intéressante à explorer: l'approche distribuée que procurent les SMA.

2.1.2 Interaction, communication et coordination des systèmes

La nécessité d'utiliser plusieurs machines pour effectuer un travail en particulier implique plusieurs contraintes. Parmi les plus importantes, on retrouve l'interaction, la communication et la coordination des systèmes. Bien évidemment, si une tâche s'effectue sur plusieurs ordinateurs en même temps, il sera impératif de partager les résultats partiels avec les autres ordinateurs. Par exemple, prenons deux sous-programmes qui

effectuent une requête (qui effectue une somme) sur une base de données distribuée. Si le résultat est la somme des résultats des deux requêtes, il est impératif de synchroniser le programme qui attend le résultat en lui spécifiant de n'effectuer la somme que lorsqu'il aura le résultat des deux requêtes. Cet exemple simple illustre un problème de synchronisation et de coordination relié à la programmation de systèmes distribués. La communication entre les systèmes demande en général de bonnes connaissances en programmation réseau. L'interaction et la coordination nécessitent des modèles et des schèmes pour la représentation des différents états possibles du système global et de ses sous-systèmes. Toutes ces nouvelles contraintes complexifient de beaucoup le développement et l'implémentation de ce type de système.

2.1.3 Hétérogénéité des différents composants des systèmes

Dans un monde idéal, les ordinateurs seraient de même conception (IBM, Macintosh, Solaris ou autres). Ils évolueraient tous sur le même système d'exploitation (Windows, Linux, UNIX ou autres) et tous les programmes seraient implémentés avec le même langage de programmation (C, C++, Java ou autres). Cependant, nous sommes très loin de cet idéal. Il faut donc élaborer des modèles, protocoles et stratégies permettant aux programmes implémentés sous différents langages de communiquer entre eux. Aussi, il faut standardiser les informations transmises car même si les programmes sont implémentés avec le même langage de programmation, il est important qu'ils s'échangent les informations dans des formats compréhensibles par les différents interlocuteurs (les sous-programmes). Celui qui reçoit des informations doit être capable de les interpréter. Les sous-programmes doivent pouvoir dialoguer dans un langage connu par les différents composants du système.

2.1.4 Autonomie et autosuffisance des systèmes

Il est important qu'un système soit capable de fonctionner sans intervention externe (du moins, qu'il ne soit pas totalement dépendant de ces interventions). Un sous-système peut être isolé des autres composants pour de multiples raisons (bris du réseau, panne d'un autre ordinateur, maintenance, etc.). Le reste du système doit être capable de s'adapter à la situation et continuer d'effectuer les tâches pour lesquelles il a été conçu.

2.2 Insuffisance des modèles existants

2.2.1 Paradigme procédural et orienté-objet

Les concepteurs et les développeurs se sont rapidement aperçus que la programmation procédurale ne répondait plus aux exigences du marché en matière d'extensibilité, de fiabilité, de testabilité, de maintenabilité, de réutilisabilité et autres standards de qualité. Ils ont donc développé un nouveau paradigme de programmation : la programmation OO.

La programmation OO répond mieux aux critères demandés par les entreprises. Elle permet le développement modulaire, une bonne réutilisabilité des composants (ce qui est très important au niveau des coûts de développement), la protection des données (l'encapsulation), une plus grande facilité de maintenance, etc.

2.2.2 Insuffisances de l'approche objet

Depuis quelques années, les besoins en terme de logiciels ont évolués. La tendance est aux logiciels puissants, traitants des masses d'informations la plupart du temps hétérogènes. Il faut des programmes qui s'adaptent aux différentes situations sans toujours nécessiter une intervention externe (du programmeur). Il faut des programmes avec une certaine « intelligence », capable de « raisonner », de « prendre de l'initiative » et apte d'interagir avec son environnement (d'autres agents ou composants du système, des personnes ou composants externes au système).

Il est très difficile de modéliser et d'implémenter ce type de système avec la programmation OO. Le modèle objet ne donne aucune spécification sur le type de communication entre les différents composants (objets) du système. Il ne fait que spécifier qu'un objet est en relation avec un autre et le type de relation qu'il existe (un à un, un à plusieurs, plusieurs à un ou plusieurs à plusieurs). Au niveau des systèmes où l'interaction et la communication entre les différents composants occupent une place de choix, il est nécessaire de spécifier comment ces entités communiquent. Si les entités sont sur la même machine, ils n'ont pas à utiliser le réseau et créer des *sockets*. Par contre, s'ils sont sur des machines différentes, il faut penser au mode d'envoi des messages : RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture), *socket* TCP (Transmission Control Protocol), *socket* UDP (User Data Protocol), *multicast* ou autres. Les entités doivent communiquer via un protocole connu des différents composants du système. Les messages envoyés doivent être définis selon un certain format compréhensible de tous les composants. Si des tâches effectuées par certains composants du système sont dépendantes d'autres composants, il est nécessaire de coordonner les différentes actions de chacun pour ne pas tomber dans des *deadlocks*.

La grande majorité de ces spécifications ne sont pas directement supportées par la programmation OO et ses techniques de modélisation. Par exemple, le modèle dynamique ne fait que représenter les états internes d'un objet, ce qui est nettement insuffisant lorsqu'il faut déterminer l'état d'un système et de ses composants en fonction des sous-systèmes et de leurs environnements respectifs. Le modèle dynamique est centré sur l'objet (évidemment) ; il est très peu sensible à son environnement. Bien sûr, il change d'état en réponse à des messages qu'il reçoit des autres objets, mais il n'est peu ou pas influencé par l'état global du système (il ne change pas d'état en réponse à plusieurs facteurs parallèles). Le diagramme fonctionnel est aussi incomplet en regard aux interactions, modes de communication et autres protocoles essentiels à ce genre de systèmes.

La programmation qui s'en suit est très difficile. Il faut impérativement connaître les bases de la programmation réseau pour l'implémentation des connexions inter-machines. La plupart des objets étant actifs, il faut aussi maîtriser la programmation *multi-thread* et la synchronisation inhérente à la concurrence et au parallélisme. Il faut implémenter les protocoles et langages de communication, la coordination des entités, une façon de déléguer dynamiquement les tâches en fonction des disponibilités des composants, la représentation de l'environnement, la représentation des connaissances et autres. Il est clair que la programmation OO n'est pas suffisante. Elle offre beaucoup d'avantages comparativement à la programmation procédurale mais elle doit être étendue pour supporter les systèmes distribués « intelligents ».

2.3 Avantages de l'approche centrée sur l'agent

Un nouveau paradigme de programmation émerge pour palier aux manques des approches traditionnelles : la programmation orientée-agent.

2.3.1 Modélisation des systèmes

L'approche OA offre une façon beaucoup plus naturelle de concevoir ce type de système. Trois grandes techniques sont utilisées pour la modélisation des SMA : l'approche centrée sur les tâches, l'approche centrée sur les buts et celle centrée sur les rôles. La première se base sur le fait que chaque agent peut effectuer des tâches, et que chaque tâche peut être effectuée par un ou plusieurs agents. Avec cette approche, il faut déterminer les tâches à effectuer pour l'atteinte du but global (la raison d'être du programme). Par la suite, il faut déterminer un ordonnancement et une cédule des tâches que chaque agent doit exécuter et coordonner leurs activités. La deuxième approche, celle centrée sur les buts, demande la spécification des buts intermédiaires des agents (ou sous-systèmes) qui sont impératifs à l'atteinte du but global (but du système). De cette façon, lorsque tous les buts secondaires sont rencontrés, le but global est atteint et le système a donc atteint son objectif (les tâches pour lesquelles le système a été conçu). La troisième approche, soit celle centrée sur les rôles, nécessite une abstraction plus élevée que les deux premières. Les agents sont plutôt modélisés en fonction de leurs responsabilités au sein du groupe (sous-système ou système). Plusieurs méthodologies ont été développées pour chacune de ces approches. Cependant, de l'avis de la majorité des chercheurs dans le domaine, une méthodologie OA en particulier ne peut être utilisée pour la modélisation de tous les types de SMA. La majorité des méthodologies développées s'appliquent à un ou quelques domaines en particulier mais ne sont pas appropriées dans d'autres circonstances.

Que l'on utilise une méthodologie spécifique pour la modélisation d'un système ou que l'on applique des concepts génériques comme l'approche centrée sur les rôles, les buts ou celle centrée sur les tâches, l'approche agent offre une façon beaucoup plus naturelle de développer les systèmes « intelligents » et distribués d'aujourd'hui. L'approche OA permet une factorisation (décomposition en sous-systèmes) et une délégation naturelle

des tâches, buts et/ou rôles à ces sous-systèmes et leurs agents. L'approche permet aussi de déterminer les protocoles de communication, d'interaction, de coordination et de négociation entre les agents. De cette façon, il est possible de superposer le problème réel sur le système à base d'agents.

2.3.2 Décomposition du système

Un SMA peut être composé d'un seul système ou de plusieurs sous-systèmes. L'opération de décomposition peut être appliquée aux sous-systèmes. De cette façon, dépendamment à quel niveau on se trouve, il est possible de voir le système soit comme un SMA, soit comme un seul agent. Cette décomposition permet de concevoir plus facilement des systèmes distribués en décomposant le problème en sous-problèmes. Les sous-systèmes sont séparés sur différentes machines et les communications et interactions s'effectuent via des langages de communication qui sont la plupart du temps basés sur la théorie des actes de langages.

Évidemment, une forte décomposition implique souvent plus de difficultés au niveau de la coordination des différents agents et sous-systèmes. De plus, une forte décomposition augmente les communications entre les sous-systèmes. Par contre, cette parallélisation ou concurrence (dépendamment si on travaille sur un ou plusieurs processeurs) permet le traitement d'une plus grande quantité d'informations et la résolution efficace de problèmes très difficiles voir mêmes impossibles à résoudre en temps normal (avec un seul processeur).

2.3.3 Tolérance aux fautes

Les systèmes à base d'agents possèdent un avantage marqué au niveau de la tolérance aux fautes par rapport aux programmes centralisés (programme situé sur un seul terminal ou serveur). La majorité des SMA sont distribués sur plusieurs machines et effectuent les différentes tâches en parallèle et/ou en concurrence. Par exemple, prenons plusieurs agents sur différents ordinateurs qui ont les qualifications requises pour effectuer des tâches de même nature. Donc, si une machine (ou un sous-système du SMA) est indisponible pour quelque raison que ce soit, les tâches à effectuer par ce sous-système peuvent être redistribuées à d'autres agents ayant la capacité d'effectuer ces dernières sur d'autres machines.

La plupart des applications OO ne possèdent pas une telle tolérance. Bien des programmes ne sont exécutés que sur une machine qui effectue un traitement séquentiel (procédural ou événementiel) avec un seul *thread* ou en concurrence sur plusieurs *threads*. Le travail étant effectué sur le même ordinateur, si celui-ci tombe en panne pour une raison ou une autre, le même sort est réservé au sous-programme s'exécutant sur cette machine.

2.3.4 Autonomie des systèmes

L'approche OA suppose la mise en application des concepts inhérents aux agents. Il faut donc que les agents d'un système possèdent une certaine autonomie, un certain degré d'intelligence artificielle, une représentation de leur environnement, qu'ils puissent interagir avec celui-ci, qu'ils soient capables de « prendre de l'initiative », de communiquer entre eux et qu'ils soient capables de s'adapter à différentes situations. La nécessité de ces concepts à différents niveaux d'un programme indiquent un certain besoin d'autonomie et valide en quelque sorte le choix de l'approche SMA plutôt que l'approche OO.

2.3.5 Autres avantages

Plusieurs autres avantages sont attribuables aux agents et SMA :

- La facilité de « changement d'échelle d'une architecture » (*scalability*) car plusieurs agents peuvent s'ajouter ou se retirer dynamiquement d'un système.
- La configuration automatique d'un SMA grâce à la plus grande « proximité » du monde réel des agents.
- La résolution plus rapide et efficace de problèmes en exploitant le parallélisme.
- La réduction des coûts de développement de logiciel : plus un logiciel est modulaire, plus la complexité et les coûts de développement diminuent.
- Diminution de l'utilisations des ressources (répartition sur différents processeurs)
- Flexibilité des systèmes : les systèmes sont composés de plusieurs agents ayant des habiletés différentes, ils peuvent donc résoudre différents problèmes (ils peuvent aussi résoudre des problèmes en coopérant et en se partageant les tâches dépendamment des capacités de chacun).

2.4 Difficultés de l'approche orientée-agent

Dans de nombreuses situations, le développement OA procure plusieurs avantages marqués par rapport à la programmation OO. Cependant, l'utilisation de ce nouveau paradigme de programmation engendre plusieurs difficultés propres à cette nouvelle vision de la programmation. Voyons les principales difficultés de cette approche.

2.4.1 Méthodologies SMA relativement complexes

Ces dernières années, une multitude de méthodologies et architectures pour les SMA ont été développées. Plusieurs proposent des façons intéressantes de modéliser les SMA et ce, en adoptant des approches diversifiées. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont, la plupart du temps, très difficiles à implémenter directement avec des langages de programmation standards comme Java, C,

C++ ou autres. Une des principales difficultés provient du fait que plusieurs méthodologies représentent les SMA avec des modèles multi-couches très complexes.

2.4.2 Méthodologies non-standardisées

Le développement d'applications OO permet l'utilisation de standards comme OMT et UML (Unified Modeling Language) pour la modélisation, la conception et l'implémentation. Cependant, la même chose n'est pas possible lorsque l'on développe une application OA. Aucun standard n'existe encore aux niveaux des méthodologies et de la technique de développement. Quelques standards sont définis par la FIPA (Foundation for Intelligent Physical Agents) pour l'implémentation des SMA [FIPA97]. Par contre, les standards définissent beaucoup plus le niveau architectural et les modes de communication (le langage) que le niveau conceptuel qu'implique la modélisation et la conception. Si nous voulons impérativement utiliser une méthodologie pour la création d'un SMA, il faut donc déterminer la méthodologie qui représente (modélise) le mieux le type de système à développer. Un tel choix suppose une connaissance relativement bonne des différentes méthodologies disponibles, ce qui est rarement le cas.

2.4.3 Passage du modèle à l'implémentation

Pour la plupart des méthodologies, la grande difficulté est le passage du modèle défini pour le SMA à son implémentation. Les méthodologies OA découpent systématiquement le modèle en plusieurs couches qui sont elles-mêmes séparées en plusieurs concepts souvent très abstraits. Des concepts comme les rôles, les tâches, les sous-tâches, les buts, le but global et autres sont difficilement transférables en terme de programmes. De plus, ces concepts se chevauchent et se superposent sur différentes couches ou phases du SMA (comme par exemple, la couche de la définition ou celle de l'organisation de la méthodologie *Role Modeling* [Collins99]). On peut facilement imaginer le niveau de difficulté et de complexité d'implémentation d'un système qui possède 6 ou 7 couches qui sont chacune découpées en rôles, sous-systèmes, etc. Il est donc nécessaire de fournir aux concepteurs de SMA des outils de développement complets (environnements de développement) les aidant tout au long du développement, de l'implémentation et du déploiement.

2.4.4 Outils de développement de SMA : primitifs et/ou incomplets

Étant donné la complexité des méthodologies développées, plusieurs équipes de chercheurs ont tenté, avec plus ou moins de succès, de développer des outils d'aide au développement de SMA. La majorité des outils furent développés pour étudier et démontrer une idée ou un concept en particulier. De ce fait, le développement de ces outils négligent, volontairement ou non, plusieurs dimensions essentielles à l'implémentation d'un SMA. Ceci rend leur utilisation inappropriée ou impossible pour le développement de systèmes réels. D'autres outils furent conçus pour supporter une

méthodologie en particulier. Ce type d'outils met l'accent sur quelques phases en particulier de la méthodologie développée et ignore des aspects primordiaux à l'implémentation. Les outils supportant une méthodologie offrent en général très peu ou pas de souplesse au concepteur.

D'après une étude que nous avons récemment publiée [Garneau02], nous pouvons constater quelques faiblesses et manques chroniques retrouvés dans les différents outils pour le développement de SMA :

2.4.4.1 Le manque d'extensibilité des différents composants

- Impossible d'ajouter des fonctionnalités aux différents composants de l'outil.
- Impossible d'ajouter des fonctionnalités aux composants créés avec l'outil.
- Impossible d'ajouter du code aux composants existants.

2.4.4.2 Le manque de réutilisabilité des composants créés avec l'outil

- Impossible de réutiliser, à l'intérieur d'autres projets, les composants créés avec l'outil (agents, bases de connaissances ou autres).

2.4.4.3 Complexité d'utilisation des outils

- Les outils développés nécessitent un effort d'apprentissage considérable.
- Les concepts utilisés sont souvent très techniques et sortent du cadre général de la programmation OA.
- Les outils développés ne sont aucunement instinctifs aux niveaux de la navigation et de la conception.

2.4.4.4 SMA développés dépendants de l'environnement

- La majorité du temps, les SMA développés sont dépendants de l'environnement et ne peuvent être exécutés à l'extérieur de celui-ci.

2.4.4.5 Environnements incomplets

- Impossible de créer des SMA déployés sur plusieurs machines.
- Impossible de programmer à l'intérieur de l'environnement.
- Impossible d'exécuter le SMA à l'intérieur de l'environnement.
- Manque de support pour le déploiement des systèmes.
- Manque au niveau des interfaces utilisateur (interfaces aidant le concepteur dans la définition des différents composants du système).
- Aucune génération automatique de code source des différents composants de l'application.
- Aucune aide à l'intérieur de l'outil

2.4.5 Aucun environnement de développement satisfaisant

Après avoir évalué les différents outils pour le développement de SMA, nous en sommes venus à la conclusion qu'aucun outil ne répondait aux différents manques et faiblesses énumérés dans la section précédente [Garneau03]. Il faut donc offrir aux développeurs un environnement complet leur permettant d'implémenter de façon plus rapide, simple et efficace des systèmes à base d'agents.

	Outils								
Critères		***JADE***	***DECAF***	***AgentBuilder***	***Zeus***	***JAFMAS/JIVE***	***Jack***	***AgentTool***	***Madkit***
Méthodologie (1)		0	0	4	4	3	0	3	3
Facilité d'apprentissage (2)		0	3	1	1	1	0	3	2
Transition entre les étapes (3)		0	0	3	2	2	0	3	2
Souplesse de l'outil (4)		3	0	1	1	2	3	0	3
Communication inter-agents (5)		4	2	4	4	2	3	2	3
Outil de « débuggage » (6)		4	2	4	4	1	2	2	3
Support développement (7)		0	2	4	4	2	1	4	0
Support implémentation (7)		0	0	4	4	2	1	2	0
Gestion du SMA (8)		4	0	3	3	0	0	1	4
Effort et simplicité (9)		2	3	2	2	1	2	3	1
Bases de données (10)		0	0	1	2	0	3	0	0
Génération de code (11)		0	0	1	3	1	0	1	0
Extensibilité du code (12)		4	1	1	2	1	4	0	3
Déploiement (13)		4	1	2	2	1	2	1	3
Documentation disponible (14)		4	1	4	4	1	3	1	3
Total (sur 60)		**29**	**15**	**39**	**42**	**20**	**22**	**26**	**30**

Figure 1 : Grille d'évaluation de différents outils SMA

(Étude comparative d'outils SMA [Garneau02])

2.5 Synthèse de la problématique

Depuis quelques années, la complexité des programmes croît rapidement. De plus, avec l'effervescence d'Internet, les logiciels tendent à être de plus en plus souvent distribués simultanément sur plusieurs machines. Les caractéristiques demandées aux logiciels ont beaucoup évoluées : autonomie, indépendance, intelligence, initiative, etc. La programmation OO n'est pas toujours bien adaptée à ces demandes et c'est ce qui a mené à l'émergence d'un nouveau paradigme de programmation : la programmation OA.

La programmation OA offre bien des avantages non-négligeables :

- Une façon naturelle de modéliser les systèmes,
- Une décomposition (factorisation du problème) en sous-systèmes,
- Une grande efficacité des systèmes implémentés,
- Robustesse (tolérance aux fautes),
- Résolution de problèmes complexes,
- Etc…

La programmation OA est un nouveau concept et plusieurs obstacles à l'utilisation de ce paradigme sont présents :

- Les méthodologies développées sont très complexes,
- Aucun standard pour l'utilisation de celles-ci,
- Grandes difficultés de transition entre le modèle et l'implémentation.

Un désavantage encore plus marqué est le manque d'outils pour le développement d'application SMA. La complexité des concepts utilisés en programmation OA et la jeunesse de cette approche sont probablement les grands responsables de cette insuffisance.

3. État de l'art

L'expression « programmation OA » fût énoncée pour la première fois il y a une dizaine d'années seulement. Cependant, depuis ce temps, beaucoup de recherches ont déjà été effectuées sur les agents, les SMA et la programmation OA.

3.1 Les agents

Les agents sont à la programmation OA ce que sont les objets à la programmation OO. Il fallait donc s'assurer d'avoir un certain consensus au sein de la communauté multi-agents sur la définition et les caractéristiques d'un agent. Celà a donné lieu à plusieurs débats et a longtemps été une source de division entre les chercheurs. Heureusement, les diverges se sont estompées et la majorité des intervenants du domaine sont maintenant d'accord sur les caractéristiques globales que doivent posséder les agents.

3.1.1 Qu'est-ce qu'un agent ?

Avant de donner une définition d'un agent, il est important de déterminer les différentes caractéristiques que devraient posséder un agent. Bien sûr, dépendamment des situations et des circonstances, un agent peut être fortement caractérisé par un sous-ensemble des attributs indiqués et être difficilement associable aux autres spécifications.

3.1.1.1 Principales caractéristiques d'un agent

Autonomie

- Un agent a un certain degré d'autonomie.
- Un agent possède certains états (non-accessibles aux autres agents et composants du système).
- Un agent peut prendre certaines décisions par rapport à ses états (sans intervention externe directe).

Situé

- Un agent est situé dans son environnement (physique ou virtuel).
- Un agent a une représentation de son environnement.

Réactif

- Un agent peut percevoir son environnement via des senseurs.
- Un agent peut agir sur son environnement via des effecteurs.

Social

- Un agent est capable d'interagir et de communiquer avec les autres agents (par des langages de communication).
- Un agent est capable de coopérer pour résoudre des problèmes ou effectuer des tâches.

Proactif

- Un agent est capable de « prendre de l'initiative » pour atteindre son but ou effectuer des tâches (et d'adopter les comportements appropriés).

Actif

- Un agent est toujours actif. Il s'exécute donc nécessairement dans un *thread* ou un *process* indépendants.

Apprentissage

- Un agent est capable d'apprendre et d'évoluer en fonction de cet apprentissage.
- Un agent est capable de changer de comportement (en fonction des expériences passées).

3.1.1.2 Définition

Un agent est une entité située, réelle ou virtuelle, agissant dans un environnement, capable de le percevoir, d'agir sur celui-ci et d'interagir avec les différents composants l'entourant. Une entité est un agent si elle est capable d'exercer un contrôle local sur ses processus de perception, de communication, d'acquisition de connaissances, de raisonnement, de prise de décision et d'exécution.

3.1.2 Architectures d'agents

D'après Russell et Norvig [Russell99], il existe quatre principales architectures d'agents. Les architectures se distinguent par le niveau de complexité de l'agent. Évidemment, plus un agent est complexe et complet, plus il est en mesure d'accomplir des tâches intéressantes.

3.1.2.1 Agent réflexe simple

L'agent réflexe simple est le type d'agent le plus simple. Comme son nom le dit, ce type d'agent ne fait que réagir à des stimulis provenant de son environnement. Ces agents n'ont aucune représentation de leur environnement, aucune base de connaissances et aucune forme de mémoire. Comme le démontre la figure 2, les agents réflexes simples ne possèdent que trois composants :

- Les capteurs (senseurs) qui servent à percevoir les changements de leur environnement.
- Un ensemble de règles et de conditions (ou une fonction) déterminant les actions à effectuer lorsqu'un changement se produit dans l'environnement.
- Les effecteurs qui permettent aux agents d'agir sur leur environnement.

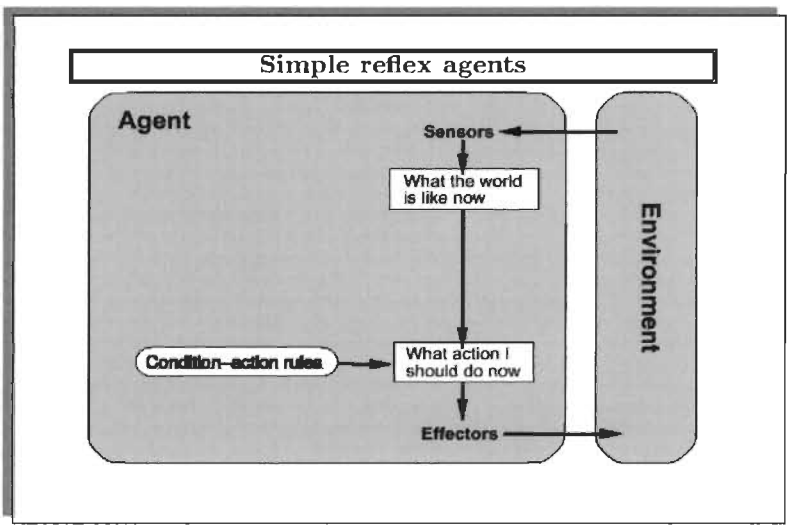

Figure 2 : Agent réflexe simple

(Artificial Intelligence: A Modern Approach [Russell99].)

3.1.2.2 Agent réflexe à états

Ce type d'agents est une version améliorée de l'architecture précédente. Ces agents possèdent aussi des capteurs, des effecteurs et des règles (ou fonctions). De plus, ce type d'agents possède différents états et il connaît les résultats des actions qu'il peut exécuter. Ces agents peuvent donc mieux déterminer quelle action poser en fonction des stimulis qu'il perçoivent de leur environnement via leurs senseurs.

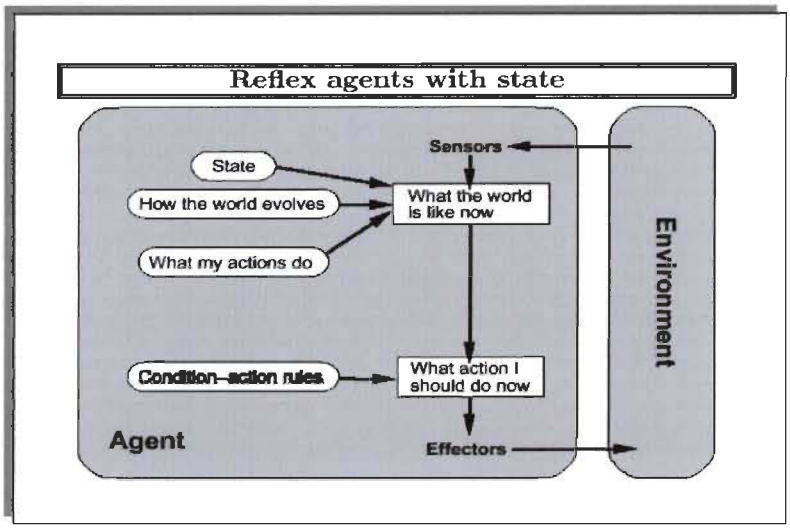

Figure 3 : Agent réflexe avec états

(Artificial Intelligence: A Modern Approach [Russell99].)

3.1.2.3 Agent à base de buts

Les agents à base de buts possèdent tous les composants des deux architectures précédentes. De plus, ces agents connaissent les conséquences de leurs actions sur leur environnement. Ce type d'agents agit en fonction de l'atteinte d'un but. De cette façon, si un agent détermine que l'exécution d'une action lui permettra d'atteindre son but, alors l'agent posera assurément cette action (lorsque c'est possible).

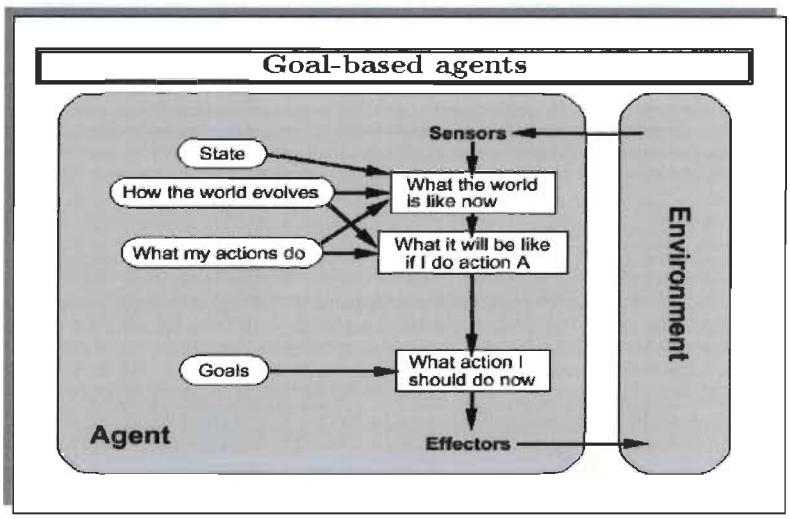

Figure 4 : Agent à base de buts

(Artificial Intelligence: A Modern Approach [Russell99].)

3.1.2.4 Agent à base d'utilité

Cette architecture est la plus évoluée. Ici aussi, les agents ont les caractéristiques des architectures précédentes. Au point de vue théorique, un agent à base d'utilité peut accomplir n'importe quel type de tâche. L'architecture permet à ces agents de résoudre des problèmes complexes. Comme, par exemple, des tâches qui nécessitent l'exécution d'enchaînement d'actions, ce qui était impossible avec les trois architectures précédentes.

Dans cette architecture, les agents possèdent une fonction qui permet d'évaluer l'utilité de chaque action. L'utilité d'une action est la détermination du rapprochement du but, dépendamment de l'action posée. Ici, contrairement à l'architecture à base de buts où les agents posent des actions aléatoires lorsqu'ils n'étaient pas en mesure d'atteindre leur but, les agents à base d'utilité peuvent déterminer l'action (ou les actions) à effectuer pour se rapprocher le plus possible du but à atteindre. Cela signifie que lorsqu'un agent ne peut atteindre son but en posant une action, alors il sélectionnera l'action qui le rapprochera le plus possible du but à atteindre ou l'action qui lui permettra d'atteindre ce but le plus rapidement possible. C'est à cette étape que l'on applique la fonction d'utilité pour déterminer l'action à poser.

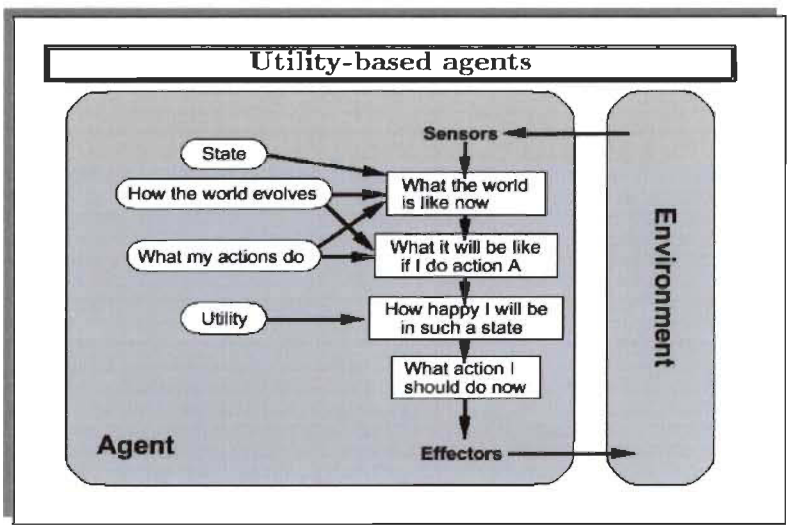

Figure 5 : Agent à base d'utilité

(Artificial Intelligence: A Modern Approach [Russell99].)

3.1.3 Les principaux types d'agents

Les types d'agents présentés ici sont parmi les types les plus connus et les plus utilisés dans les applications. Il existe plusieurs façons de catégoriser les agents et cette typologie n'est pas unique. Elle se veut seulement une idée générale des différents types d'agents que l'on peut retrouver dans la littérature. Une des typologies proposées [Nwana96] se base sur l'habilité des agents à coopérer, à apprendre et à agir de façon autonome. Dans

cette typologie, on retrouve les *Smart agent*, les agents interface, les agents collaboratif et les *collaborative learning agent (Figure 6)*.

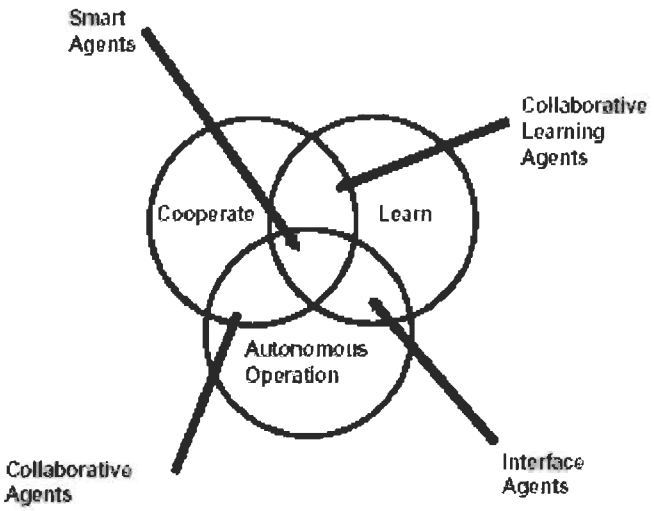

Figure 6 : Typologie des agents

(Software Agents: An Overview. Knowledge Engineering Review.[Nwana96])

3.1.3.1 Les agents collaboratif

Les agents collaboratif mettent l'emphase sur l'autonomie, la collaboration et la communication avec d'autres agents dans le but de résoudre des problèmes. Ce type d'agent est utilisé dans la résolution de problèmes distribués, donc dans les SMA. Ces agents impliquent plusieurs difficultés de conception : la coordination des actions, la définition des protocoles de communication entre les agents, la négociation entre les agents pour l'allocation des tâches, le partage des ressources (qui sont limitées), la synchronisation des actions, etc.

3.1.3.2 Les agents interface

Les agents interface sont des agents relativement autonomes qui utilisent différentes techniques d'apprentissage pour effectuer des tâches pour leur utilisateur. On peut voir un agent interface comme un assistant personnel qui effectue diverses tâches. Les principales tâches de ce type d'agent est de fournir du support et de l'assistance à l'utilisateur. Entre autre, ces agents peuvent filtrer des courriels, acheter ou vendre des articles sur le Web ou être votre compagnon dans *MS Word*. Un des principaux mandats de ces agents est de faciliter et accélérer quelques tâches spécifiques de l'utilisateur. La plupart du temps, les agents interface observent et enregistrent les actions de l'utilisateur. Par la suite, ils peuvent lui suggérer une meilleure façon d'effectuer ce travail. Les agents apprennent de différentes façons. Parmi les plus courantes, on retrouve l'apprentissage par observation,

par imitation de l'utilisateur, l'apprentissage par réception de *feedback* (négatif ou positif) de l'utilisateur et réception d'instructions explicites de l'utilisateur (Figure 7).

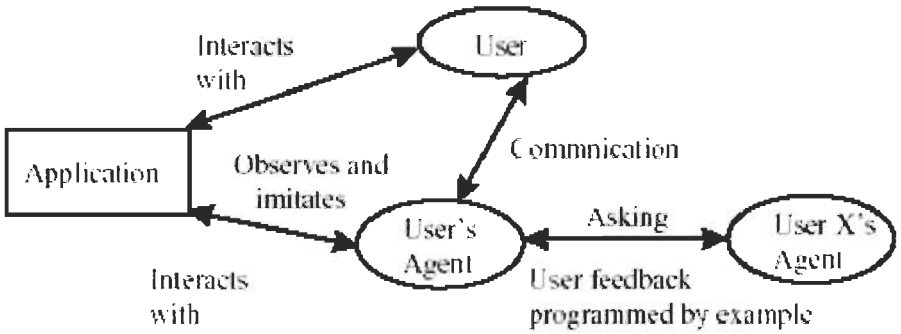

Figure 7 : Architecture d'un agent interface simple

(Extrait de [Maes95])

3.1.3.3 Les agents d'information

Les agents d'information sont des agents très populaires pour retrouver et analyser de grandes quantités d'information [Klusch99]. Les agents d'information sont des outils qui gèrent des masses énormes d'information qu'ils vont chercher sur des pages Web, sur des serveurs ou dans des bases de données. Les moteurs de recherche les utilisent pour indexer des sites Internet, pour vérifier les mises à jour des sites, etc. De plus, ils servent aux utilisateurs qui veulent retrouver de l'information sur un sujet, sur un service ou sur un bien en particulier. Ces agents récupèrent les informations, les filtrent, les classifient et les renvoient à l'usager qui peut déterminer plus rapidement et facilement si les informations recueillies contiennent les renseignements désirés.

3.2 Les systèmes multi-agents

3.2.1 Qu'est-ce qu'un système multi-agents ?

Avant de donner une définition d'un SMA, il est important de déterminer les différentes caractéristiques que devrait posséder ce type de système.

3.2.1.1 Caractéristiques d'un système multi-agents

- Ensemble d'agents agissant et travaillant indépendamment les uns des autres.
- Chaque agent est une partie du système.
- Chaque agent travaille dans le but d'accomplir ses tâches.
- Chaque agent est capable de communiquer et d'interagir avec d'autres agents.

- Un agent coopère avec les autres agents lorsque nécessaire.
- Un agent est capable de coordonner ses activités avec les autres agents pour accéder à des ressources et à des services partagés dont il a besoin (pour réaliser ses buts).
- Les agents ont un but commun (si ce n'est pas un SMA réactif ou émergent).
- Chaque agent a une vue partielle du SMA.

3.2.1.2 Définition

Un système multi-agents est une société d'agents où les interactions entre les agents et leur environnement mènent à un comportement propice à l'atteinte d'un but global.

3.2.2 Types d'architectures de SMA

Les SMA peuvent être conçus selon plusieurs types d'architectures. Le choix de l'architecture aura beaucoup d'impact sur le niveau de complexité de la modélisation, du développement et surtout de l'implémentation du système. En effet, les architectures qui tendent vers la décentralisation se modélisent plus difficilement. Par contre, la plus grande difficulté de ces architectures est leur implémentation. D'un autre côté, les systèmes adoptant une architecture plus centralisée (ou hiérarchisée) sont relativement plus simples à concevoir. Les architectures présentées ici sont les plus utilisées mais elles ne sont pas uniques. En effet, il existe plusieurs autres architectures hybrides empruntant quelques caractéristiques à plusieurs architectures.

3.2.2.1 Les systèmes centralisés (Blackboard-based)

Les systèmes centralisés se basent sur un concept assez simple et puissant : le partage des données. Dans ces systèmes, les agents ne se communiquent pas directement les données entre eux. Ils envoient et obtiennent les données du tableau (*board*). Cette structure permet aux agents d'échanger les données utiles à l'exécution du programme et de les garder dans une structure accessible à tous. Cette architecture possède trois sous-systèmes : KS (Knowledge Source) qui sont en fait les agents du système, le tableau qui garde et partage les données pour les agents ainsi qu'un contrôleur qui gère les conflits d'accès aux ressources entre les agents.

Les principaux désavantages de ce type d'architecture sont l'inefficacité des systèmes et leur manque de tolérance aux fautes. Comme toutes les données sont gardées sur la même machine, le manque d'espace mémoire devient souvent un problème. D'un autre côté, lorsque plusieurs agents tentent d'accéder aux mêmes données, un problème de partage des ressources est inévitable.

3.2.2.2 Les systèmes hiérarchisés (ou horizontal)

Ces systèmes se basent sur une structure où les entités répondent à leurs supérieurs hiérarchiques. Ce type de système est relativement simple à implémenter. Plusieurs systèmes utilisent cette architecture pour la réalisation de SMA. Cette architecture comporte des faiblesses. Parmi ces dernières, on note le peu de tolérance aux fautes, la limite imposée par les capacités de son supérieur hiérarchique, etc.

3.2.2.3 L'approche totalement distribuée

Cette approche est de loin la plus puissante mais aussi la plus complexe à développer et à implémenter. L'approche consiste à diviser le SMA en sous-systèmes indépendants effectuant chacun une partie du travail. Chaque sous-système est constitué d'un ou plusieurs agents pouvant effectuer une ou plusieurs tâches. Ces agents peuvent avoir un but auxiliaire ou tout simplement attendre des requêtes provenant des autres agents leurs demandant d'effectuer une ou des tâches en particulier.

L'approche distribuée procure de grands avantages comme la très grande tolérance aux fautes, le partage équitable du travail entre les sous-systèmes et/ou agents, l'utilisation plus uniforme des ressources, l'indépendance et l'autonomie des sous-systèmes, la répartition des tâches, l'efficacité du modèle concurrent et/ou parallèle, leur grande extensibilité, etc.

Il existe peu de systèmes totalement distribués étant donné leurs complexité de développement et d'implémentation. Le développement d'un système distribué demande la résolution de plusieurs problèmes comme la façon de communiquer entre les sous-systèmes, les moyens utilisés pour déterminer l'affectation des tâches, la connaissance mutuelle des sous-systèmes et de leur environnement, la synchronisation et la coordination des actions des agents et des différents sous-systèmes, etc.

3.2.3 Agent, système ou sous-système ?

Deux aspects intéressants avec le modèle distribué sont les notions de système et sous-système. Avec une approche distribuée, si on regarde les sous-systèmes à partir d'un niveau supérieur, ils peuvent être considérés comme étant une entité (agent) [Ferber99]. On peut aussi faire l'opération inverse et regarder une entité comme étant constituée d'un ensemble de sous-systèmes à un niveau inférieur. De cette façon, il est possible d'effectuer une décomposition itérative du problème (but à atteindre) en sous-problèmes (sous-buts).

3.3 Concepts inhérents aux systèmes multi-agents

Depuis l'avènement de la notion de SMA, plusieurs concepts et théories ont été développés. Cependant, ces derniers n'ont pas tous survécus au fil du temps. Regardons quelques notions et concepts qui sont populaires et utilisées depuis quelques années.

3.3.1 Les langages de communication entre les agents (ACL)

Lorsque la notion de groupe d'agents est apparue, les recherches sur des mécanismes permettant aux agents de communiquer entre eux se sont amorcées. Celles-ci ont mené à trois langages de communication et de représentation de l'information qui sont devenus des « standards » en SMA : KQML, FIPA ACL et KIF.

3.3.1.1 KQML (Knowledge Query Manipulation Language)

Ce langage de communication se base sur la théorie des actes de langages [Finin93], [Finin94], [Finin94b], [Finin94c]. KQML détermine un format pour les messages et un protocole pour la réception et l'envoi de ces derniers. KQML permet aux agents de partager des informations avec les autres agents du système afin de coopérer pour résoudre un problème.

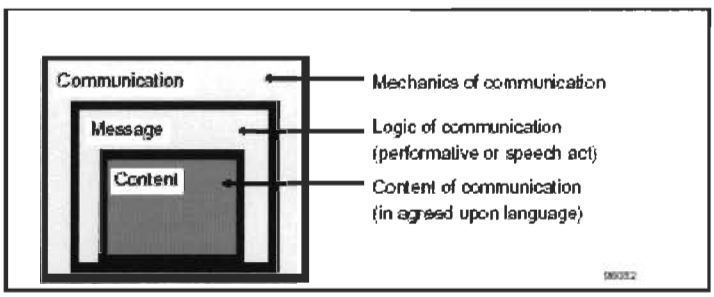

Figure 8 : Les trois couches d'un message KQML

(Guide de l'utilisateur, AgentBuilder [AgentBuilder U.G.])

Au niveau conceptuel, le langage KQML se divise en trois couches principales : la couche contenue, la couche message ainsi que la couche communication (Figure 8). La couche contenue est la représentation du message (son implémentation). KQML supporte la représentation des messages de n'importe quel type (chaînes de caractères, messages binaires ou autres). La couche communication encapsule les informations qui décrivent les paramètres nécessaires à la communication comme l'identité du propriétaire du message (l'agent qui envoie) et son GUID (Global Unique Identifier). La couche message détermine le type d'interaction avec l'interlocuteur comme le protocole d'envoi du message, le type de performative et les ontologies utilisées.

Le langage fourni au développeur une syntaxe standard pour les messages et plusieurs performatives qui spécifient la ou les forces du message tout en laissant l'utilisateur décider du contenu du message. Parmi les performatives les plus courantes, on retrouve *tell (Figure 9), inform, perform, reply* et bien d'autres. Chaque performative contient une liste de paramètres qui lui est associée. Le langage se base sur l'hypothèse que le mode de transport des messages est fiable et préserve l'ordre d'envoi et de réception des messages. Par contre, il n'existe aucune garantie que ces derniers se rendront à destination.

```
tell
      :content <expression>
      :language <word>
      :ontology <word>
      :in-reply-to <expression>
      :force <word>
      :sender <word>
      :receiver <word>
```

Figure 9 : Exemple de performative avec KQML

3.3.1.2 FIPA ACL

FIPA (Foundation For Intelligent Physical Agent) est une organisation qui tente d'instaurer des standards dans le domaine des SMA [FIPA97]. Le langage (FIPA ACL) est un langage semblable à KQML auquel des protocoles d'interaction comme le *contract-net* et autres protocoles populaires ont été ajoutés.

3.3.1.3 KIF (Knowledge Interchange Format)

Les agents ont besoin d'un langage pour communiquer entre eux mais ils ont aussi besoin de comprendre le contenu des messages qu'ils reçoivent. Ceci est possible par l'intermédiaire de KIF qui permet de représenter le contenu des messages par des prédicats et la logique du premier ordre. La description du langage se divise en deux parties : la spécification de la syntaxe (comment se présentera une information en particulier) et la représentation de cette information. Le langage possède aussi des méthodes pour la représentation de la « méta-connaissance » qui peuvent servir de liens entre les bases de connaissances et des langages de représentation des connaissances comme PROLOG et LISP.

3.3.1.4 Ontologie

Les ontologies ont été développées pour fournir des vocabulaires spécifiques dépendants du domaine d'application pour la communication entre les agents. Une ontologie définie

les concepts et les relations qui existent entre les mots d'un vocabulaire formel pour les agents qui utilisent ce dernier. Il est à noter que les agents d'un SMA partagent la même ontologie (le même vocabulaire) mais ceci ne veut pas dire pour autant qu'ils possèdent la même base de connaissances. Les agents doivent partager le même vocabulaire et connaître les mêmes concepts s'ils veulent communiquer d'une façon cohérente et consistante.

3.3.2 Communication, coordination et interaction

Plusieurs méthodologies, techniques et langages sont développés pour simplifier la coordination et l'interaction entre les agents à l'intérieur d'un système. Par contre, aucun standard ne semble émerger pour l'instant des travaux actuels ou passés. Cependant, quelques méthodes sont plus utilisées que les autres.

3.3.2.1 COOL

COOL [Barbuceanu97] est un langage basé sur les conversations permettant de représenter et d'appliquer explicitement les besoins en termes de coopération à l'intérieur d'un SMA. Ce langage permet de définir les comportements des agents à l'intérieur de leur organisation. La méthodologie consiste en cinq étapes : identifier les agents externes, identifier les interactions logiques, identifier les règles de conversation, identifier la règle de continuation et l'implémentation (en LISP). La communication entre les agents se fait via un langage qui est une extension à KQML. COOL n'est pas seulement un langage mais aussi un outil pour le développement de SMA. Par contre, il est très limité à plusieurs niveaux et son intérêt majeur est définitivement son langage permettant de modéliser les différentes interactions entre les composants du système.

3.3.2.2 Réseaux de Pétri *(Petri nets)*

Les réseaux de Pétri [David92] permettent de modéliser les interactions, la synchronisation, la concurrence et la cohérence des conversations entre les agents. De plus, on peut les utiliser pour modéliser l'accès à certaines ressources limitées ou pour résoudre les problèmes d'affectation entre les différentes tâches. En fait, les réseaux de Pétri sont très puissants. Ils permettent une modélisation rigoureuse basée sur un modèle mathématique à base de graphes, de logique du premier ordre et de matrices d'adjacences.

3.3.3 Connaissances et raisonnement

Fournir des connaissances aux agents et leur donner des outils pour bien les utiliser est primordial. La grande majorité du temps, lorsque des agents « intelligents » sont implantés dans un système, leur architecture en est une à base de BDI.

3.3.3.1 Agents « intelligents » ou agents BDI

Les agents BDI [Rao95] sont les dignes successeurs du travail de Shoham [Shoham93b] qui fut un des premiers à définir un type d'agents « intelligents » : Agent-0. Ces agents possèdent deux états mentaux internes : les croyances (*beliefs*) et les engagements (*commitment*). Ce type d'agents peut effectuer deux sortes d'actions : les actions privées et les actions d'obligation.

Un autre ancêtre du modèle BDI est le modèle PRS (Procedural Reasonning System) [Georgeff87] qui fût lui aussi un des premiers modèles de programmation à base de raisonnement (Figure 10). Un de ses successeurs est devenu très populaire dans le monde des agents : dMars (distributed Multi-Agents Reasonning System) qui est un modèle BDI classique.

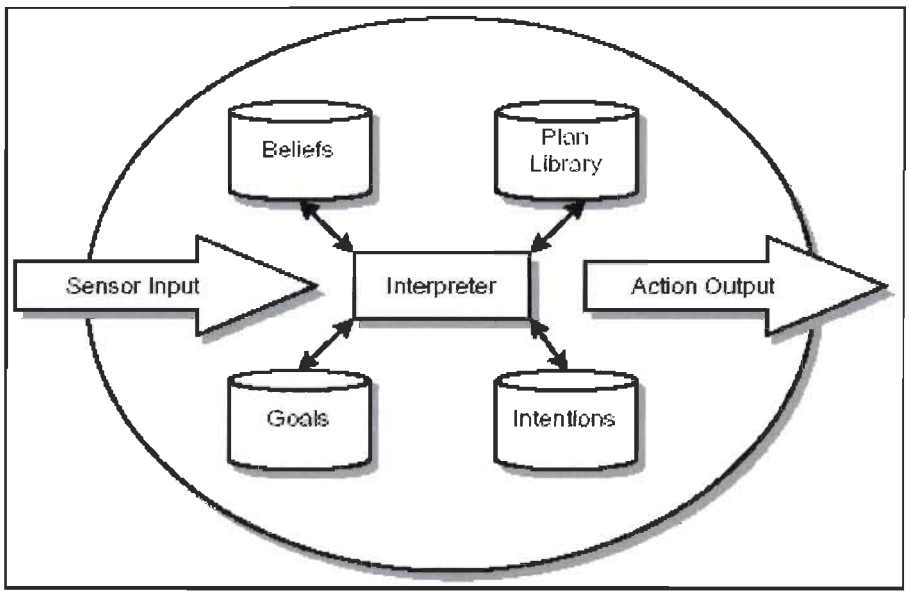

Figure 10 : Architecture d'un modèle BDI : PRS

(A formal specification of dMars [d'Iverno97])

Un autre type d'agents succéda à Agent-0 : PLACA (PLAnning Communating Agents) [Thomas93]. Ce type d'agent est une extension directe aux travaux de Shoham auxquels on ajoute une partie planification.

La plupart des types d'agents BDI sont des évolutions de l'un ou l'autre des modèles introduits ci-haut.

3.3.3.2 Bases de connaissances

Les agents « intelligents » nécessitent souvent l'utilisation de bases de connaissances. Les bases de connaissances sont des outils très puissants permettant à un agent d'utiliser la

logique propositionnelle pour permettre la déduction de connaissances qui, a priori, n'existent pas explicitement.

Une base de connaissances est composée de faits, de règles, et d'un engin d'inférence[Russell99]. Un fait est un constat qui est véridique et connu. Une règle consiste en un ensemble d'antécédents (LHS ou Left Hand Side) et de conséquents (RHS ou Right Hand Side). Pour vérifier si une règle est vraie, il faut vérifier si tous les antécédents de la règle sont vrais. Si c'est le cas, alors le ou les conséquents de cette règle sont tous vrais. On peut alors ajouter les conséquents dans la liste de faits (cette règle vient donc d'ajouter des connaissances à l'agent qui n'avait pas ces connaissances au départ). Ce processus s'appelle inférence. Ce travail se fait à l'intérieur du moteur d'inférence qui permet de déduire de nouveaux faits à partir des croyances de départ et des règles contenues dans la base de connaissances. Deux principaux types d'inférence sont possibles : le chaînage avant (*forward chaining*) et le chaînage arrière (*backward chaining*). La première technique consiste à tenter de vérifier la véracité de tous les antécédents des différentes règles pour trouver de nouveaux faits, les ajouter à la liste des faits et recommencer la vérification des antécédents des règles tant que l'on déduit de nouvelles informations. La deuxième technique consiste à vouloir vérifier la véracité d'un ou des conséquents d'une règle. À partir de la règle qui contient le conséquent, on tente de vérifier l'ensemble des antécédents de celle-ci.

3.3.3.3 JESS

<JESS> JESS est un engin d'inférence permettant d'effectuer le travail mentionné précédemment. L'engin est entièrement implémenté en java contrairement à plusieurs autres qui utilisent des langages comme LISP ou PROLOG pour ce type de travail. De plus, l'engin permet le chaînage arrière, ce qui n'est pas le cas de la majorité des engins d'inférence. De plus, il est possible de créer des scripts à l'intérieur de JESS. L'engin est beaucoup plus puissant que ses homologues. Une des raisons principales de cette puissance est qu'il est codé en Java plutôt qu'en LISP ou en PROLOG.

3.3.4 FIPA (Foundation for Intelligent Physical Agents)

Formée en 1996, la FIPA est une organisation qui tente de standardiser les concepts relatifs aux SMA. Les recherches sont axées sur la définition des propriétés des agents, les rôles, les architectures et les différents composants d'un SMA. Les premiers travaux de l'organisation furent la spécification en 1997 de standards [FIPA97] (semblable aux normes ISO) sur la gestion des agents, le système de nommage des agents et sur les communications entre agents. Leurs recherches se sont dirigées vers la communication, la négociation et les architectures.

3.4 La programmation orientée-agent et Java

Le choix du langage de programmation pour l'implémentation d'un SMA ou d'un outil pour les SMA est très important.

3.4.1 Langage orienté-objet

La programmation OA est une extension à la programmation OO. Il faut donc utiliser un langage supportant bien la programmation OO. Un langage OO permet de bénéficier de plusieurs avantages comme l'encapsulation, l'abstraction et le polymorphisme. Un langage OO simplifie aussi la phase de modélisation du système grâce aux notions d'objet et de classe permettant la création de modèles plus près du monde réel.

3.4.2 Langage *multi-thread*

L'utilisation d'agents indépendants et autonomes nécessite l'emploi d'un langage permettant la concurrence et/ou le parallélisme à l'intérieur d'un même programme. Le langage doit aussi permettre une synchronisation simple entre les différents *thread* ou *process* et un système de protection pour les données qui sont partagées entre ces derniers.

3.4.3 Langage de programmation réseau

Les SMA étant distribués par nature, il est indispensable d'utiliser un langage permettant la programmation réseau de façon simple et efficace. Évidemment, plus le langage est souple et puissant pour la communication via Internet, moins il impose de barrière pour la conception des systèmes. Si le langage permet l'utilisation de plusieurs protocoles et types de communication comme TCP, UDP, Multicast, CORBA ou autres, alors il est possible de choisir le mode de communication entre les agents en fonction des besoins.

3.4.4 Langage performant

L'exécution de plusieurs entités sur la même machine (concurrence), la transmission d'une quantité relativement grande d'informations par Internet, l'élaboration de conversations, l'utilisation de bases de connaissances et bien d'autres facteurs demandent l'utilisation d'un langage puissant, performant et efficace.

3.4.5 Langage sécuritaire

Étant donné que la plupart des SMA sont distribués en sous-systèmes sur plusieurs machines, il est nécessaire d'utiliser Internet pour la communication entre les sous-systèmes. L'utilisation d'Internet pour transporter des informations implique des considérations au niveau de la sécurité. Il faut donc un langage offrant des mécanismes permettant la sécurisation simple du système.

3.4.6 Langage portable

Un langage permettant l'exécution du SMA sur différents systèmes d'exploitation sans nécessiter de recompilation ou de changement d'implémentation (Figure 11).

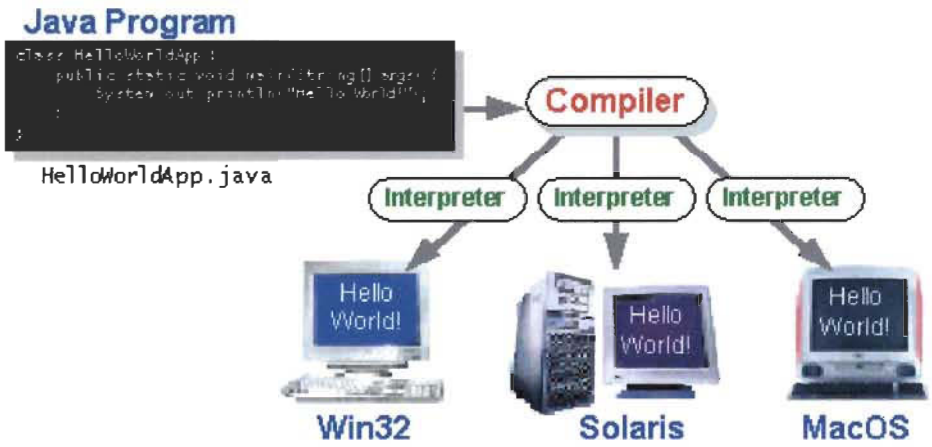

Figure 11 : Processus de compilation et d'exécution d'une application Java

(Tutorial Java <Java>)

Utilisation de Java :

Java est le seul langage qui répond à toutes les exigences des SMA mentionnées précédemment.

- Le langage est OO et offre une librairie de quelques milliers de classes pour supporter le développeur dans plusieurs axes de la programmation.

- Java est un langage qui supporte très bien la concurrence et le parallélisme. La classe *Thread* et l'interface *Runnable* permettent un développement simple et efficace d'applications *multi-threads*. Il est aussi facile de protéger l'intégrité des données et l'accès aux différentes ressources avec le mot clé *synchronized* qui permet d'obtenir un *lock*, soit sur l'objet spécifié ou sur la classe de celui-ci.

- Le langage supporte de façon remarquable la programmation réseau avec les classes de son package *java.net*. L'utilisation de ces classes permet la création et l'utilisation de connections réseaux selon plusieurs protocoles comme TCP, UDP et HTTP. Il est aussi possible de créer des communications *multicast*. Le *package java.rmi* permet l'utilisation d'appels à distance RMI comme type de communication et, par le fait même, offre une façon simple de répartir les objets qui sont des ressources sur différentes machines. De plus, le langage supporte CORBA grâce à plusieurs *packages* de classes.

- Les performances du langage sont impressionnantes et souvent comparables à celles des langages C et C++ depuis l'avènement des machines virtuelles JIT (Just In Time compiler).

- La sécurité des applications Java est supportée par l'utilisation de fichiers de sécurité dans lesquels on spécifie les différentes permissions autorisées. Le package *java.security* offre aussi plusieurs outils pour la sécurité comme l'encryption de données, l'attribution de clefs et autres.

- Une phrase dite par les développeurs de *Sun Microsystems* (créateur de Java) résume bien la portabilité du langage : ***Compile once, run everywhere.***

3.5 Méthodologies et architectures SMA

Depuis quelques années, une multitude de méthodologies et d'architectures pour le développement de SMA ont été proposées. Mais, comme mentionné précédemment, aucune de celles-ci ne se démarque vraiment.

3.5.1 MaSE (Multiagent Systems Engineering Methodology)

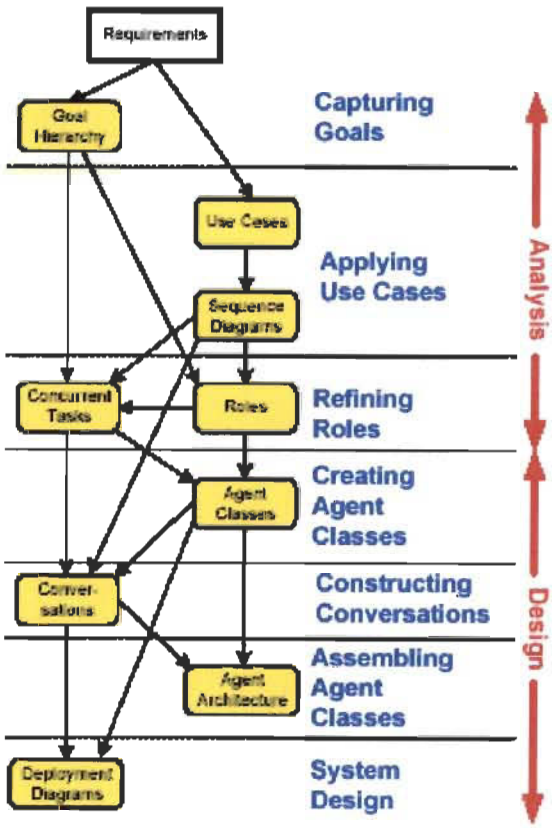

Figure 12 : Méthodologie MaSE

(Méthodologie MaSE [DeLoach])

Cette méthodologie comporte sept phases : trouver les buts, appliquer les cas d'utilisation, raffiner les rôles, créer les classes d'agents, construire les conversations, assembler les classes d'agents ainsi que l'implémentation (Figure 12). Cette méthode met l'accent sur l'analyse et le développement. MaSE est une des méthodologies les plus complète. De plus, elle est aussi relativement simple [DeLoach], [DeLoach01].

3.5.2 AGR (Agent/Groupe/Rôle)

Aussi appelée Aalaadin, la méthodologie AGR est basée sur un concept simple qui suppose qu'un agent possède un ou plusieurs rôles à l'intérieur d'un ou plusieurs groupes et que chaque groupe contient un ou plusieurs rôles [Ferber97], [Ferber97b], [Ferber97c].

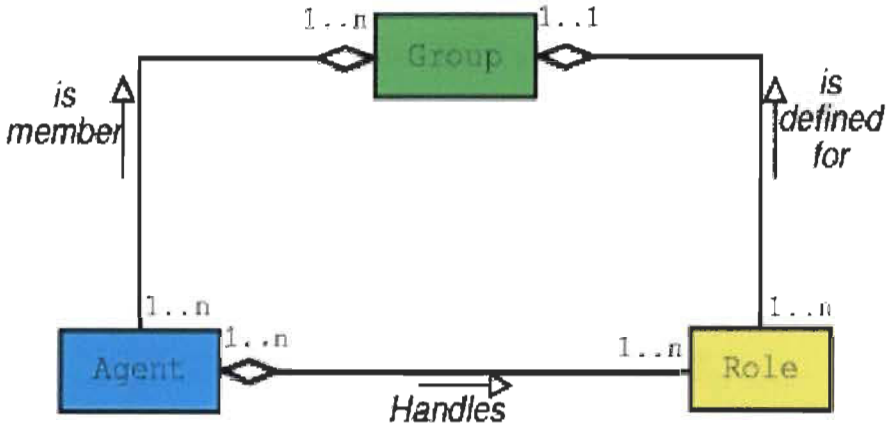

Figure 13 : Méthodologie AGR

(Documentation MadKit <MadKit>)

3.5.3 Role Modeling

Cette méthodologie est constituée de cinq couches dont trois déterminant les composants de l'agent [ZEUS R. M.G.], [ZEUS A. R. G]. La première couche des agents est celle de la définition où l'agent est vu comme une entité autonome capable de raisonner en termes de ses croyances, ses ressources et de ses préférences. La seconde couche est celle de l'organisation [Collins98], [Collins99]. Dans celle-ci, il faut déterminer les relations entre les agents. La dernière couche est celle de la coordination où les modes de communication, les protocoles, la coordination et autres mécanismes d'interactions sont déterminés.

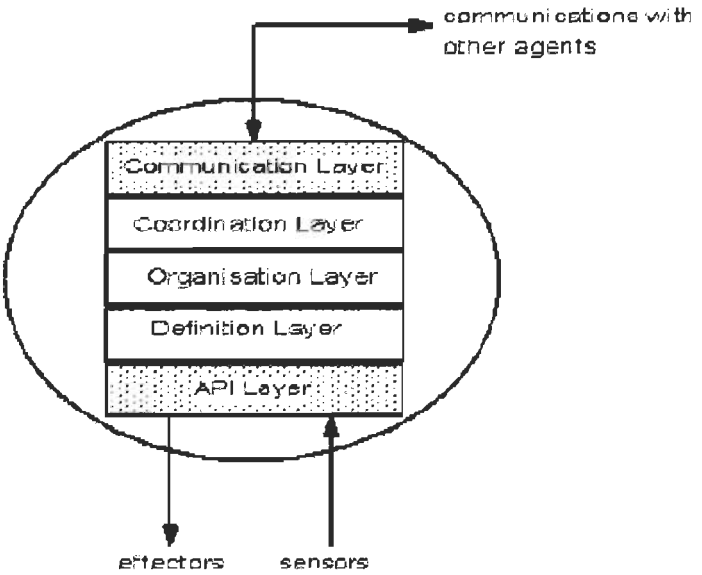

Figure 14 : Méthodologie Role Modeling utilisée pour Zeus

(Zeus: A collaborative agents toolkit [Collis98b])

3.5.4 JAFMAS

JAFMAS propose une méthodologie en cinq phases : identifier les agents, identifier les conversations, identifier les règles de conversation, analyser le modèle des conversations ainsi que l'implémentation [Chauhan97], [Baker].

3.5.5 Autres méthodologies et architectures

Plusieurs autres méthodologies et architectures furent développées. En voici quelques-unes : RETSINA [Sycara], dMARS [d'Iverno97], OAA (open agent architecture) [Martin99], DESIRE [Brazier97], Gaia [Zambonelli00], Tropos [Mylopoulos], Kaos [Bradshaw96] et bien d'autres [Delisle02].

3.6 Outils et utilitaires pour le développement de SMA

Le concept de programmation OA est une idée très intéressante et les méthodologies développées fournissent des patrons théoriques pour la modélisation des SMA. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont souvent difficiles à implémenter directement avec des langages de programmation standards comme Java ou C++. Plusieurs outils (éléments logiciels offrant des services pour le développement de SMA) de différents types ont été développés récemment pour la programmation OA.

3.6.1 AgentTool

[AgentTool U.M.], [DeLoach], [DeLoach01], <AgentTool> Cet outil se base sur une méthodologie qui se veut une extension au modèle OO : MaSE. Cette méthodologie met l'accent sur l'analyse et le développement mais laisse à l'utilisateur une liberté aux niveaux de l'implémentation et du déploiement. Ceci se reflète au niveau de l'outil qui ne supporte bien que les deux premières phases (analyse et développement). Il supporte ces deux étapes par ses différents éditeurs. Il permet aussi de vérifier la validité des conversations entre les agents. Cet outil peut être considéré comme un utilitaire de « débuggage ». La génération automatique du code (en Java) des conversations est disponible, mais on ne peut aller plus loin au point de vue implémentation. Un bref manuel de l'utilisateur spécifie les étapes à suivre. Cet outil est intéressant pour effectuer les premières étapes du développement d'un SMA. Par contre, il est insuffisant lorsque vient le temps d'implémenter le système.

3.6.2 AgentBuilder

<AgentBuilder> AgentBuilder est un environnement de développement complet [AgentBuilder R.M.], [AgentBuilder U. G.]. Toutes les phases du développement sont supportées par des interfaces. Une modélisation OO avec OMT est la base de la conception des systèmes à laquelle on ajoute une partie « ontologie ». Le développement demande une connaissance du modèle BDI « agent-0 » car l'élaboration du comportement des agents se fait à partir de ce dernier. KQML est utilisé comme langage de communication entre les agents. L'environnement est très peu extensible et y ajouter du code est difficile. Les utilisateurs sont limités (la plupart du temps) à implémenter les composants à partir des interfaces, ce qui limite les possibilités. L'exécution du système se fait à partir de l'engin d'exécution d'AgentBuilder, ce qui restreint son utilisation. Par contre, il est possible de créer des fichiers « .class » et les exécuter sur une JVM standard. La documentation est composée d'un guide de l'utilisateur et d'un manuel de références. Ces deux derniers sont assez volumineux et couvrent l'ensemble de l'environnement et la plupart des concepts utilisés. AgentBuilder est un outil complexe, qui demande d'énormes efforts d'apprentissage et de bonnes connaissances dans le domaine des SMA pour être utilisé de façon intéressante. L'outil est complet au niveau des composants mais est très limité aux niveaux de l'extensibilité, du déploiement (autre qu'avec l'engin conçu à cette fin) et de la réutilisabilité.

3.6.3 Brainstorm / J

<Branstorm/J> On ne peut parler ici d'environnement de développement car cet outil consiste en un ensemble de classes, abstraites pour la plupart, permettant le développement de SMA. Le cadre se base sur l'architecture Brainstorm. Celle-ci considère un SMA comme étant un système OO possédant un méta-système. Ce dernier est composé de plusieurs couches. Aucune méthodologie n'est fournie et très peu de documentation est disponible.

3.6.4 DECAF

<DECAF> DECAF est un environnement de développement de plans [DECAF01]. L'outil fourni quelques utilitaires dont un ensemble de classes pour l'élaboration de plans et la coordination des tâches. Un planificateur applique des heuristiques pour trouver une cédule aux tâches. Une interface permet la construction des tâches. De plus, DECAF fourni un éditeur d'agents utile pour le « débuggage ». [Decker] L'environnement permet le développement rapide de petits systèmes en Java via l'utilisation de classes déjà implémentées simplifiant le développement des agents. Aucune méthodologie n'est spécifiée pour la conception. De plus, la documentation est très limitée. Elle consiste en une petite introduction à la programmation avec DECAF. L'outil est incomplet; il est très peu documenté et les facilités permises sont très limitées.

3.6.5 Jack

<Jack>, [Busetta99] L'environnement Jack est constitué d'un éditeur gestionnaire de projet, d'un langage de programmation JAL (Jack agent language) et d'un compilateur qui transforme le langage JAL en Java pur [Jack D. E. U.G.], [Jack U.G.]. Le gestionnaire de projet est une interface qui possède un éditeur de textes où se fait l'implémentation du système. La sauvegarde des fichiers, la compilation (passage de JAL à Java) et l'exécution du système se font aussi à l'intérieur de cette interface. Le langage JAL est une extension de Java. Une bonne documentation est disponible. Elle comprend un manuel de l'utilisateur qui explique le langage JAL, un guide pour le développement, des exercices et quelques petits exemples. Aucune méthodologie n'est proposée. Elle est laissée libre aux utilisateurs. Les agents sont basés sur un modèle BDI qui se veut une évolution du modèle dMars. Aucun éditeur n'est disponible pour le développement ou le déploiement des systèmes. Jack est très long à maîtriser. Il faut apprendre le langage JAL et idéalement connaître le modèle BDI de dMars [d'Iverno97]. De plus, le manque de support graphique complique le développement, l'implémentation et le déploiement des systèmes.

3.6.6 Jade

<Jade> Jade est un environnement de développement qui répond aux normes FIPA97 [JADE A.G.], [JADE P.G.], [Pitt], [Rimassa], [Rimassa99], [Rimassa00]. L'outil ne se base sur aucune méthodologie en particulier. Il fournit plutôt des classes qui implémentent JESS pour la définition du comportement des agents [JADE J.T.]. Jade possède trois modules principaux (nécessaire aux normes FIPA).

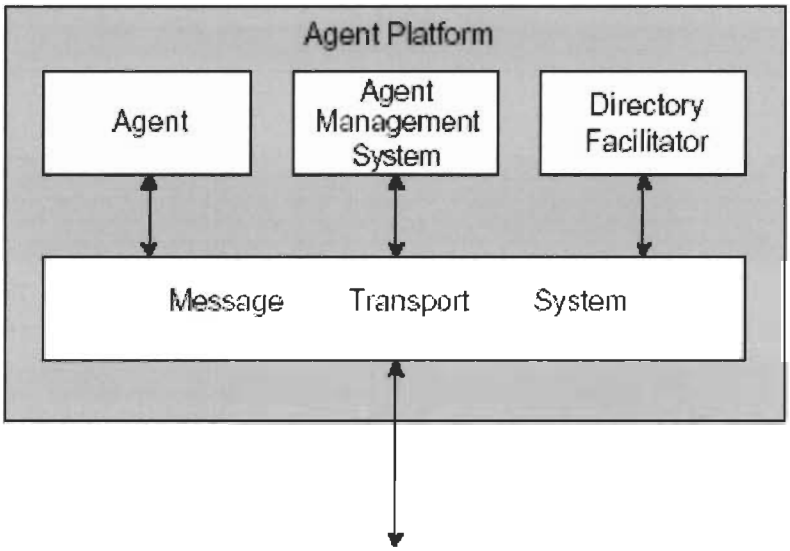

Figure 15 : Architecture d'un agent FIPA

(guide du programmeur Jade [JADE P.G.])

Le DF (Directory Facilitator) fourni un service de pages jaunes à la plate-forme. Le ACC (Agent Communication Chanel) gère la communication entre les agents. Le AMS (Agent Management System) supervise l'enregistrement des agents, leur authentification, leur utilisation ainsi que l'accès aux ressources du système (Figure 15).

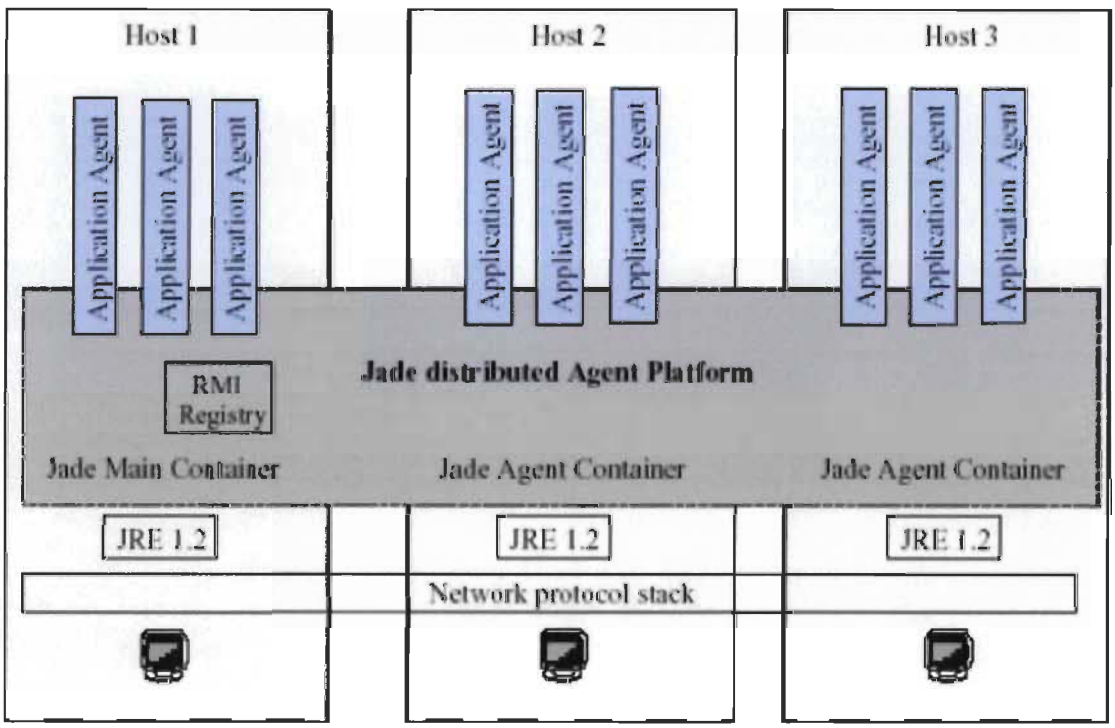

Figure 16 : Architecture d'une application Jade

(guide du programmeur Jade [JADE P.G.])

FIPA ACL est utilisé comme langage de communication entre les agents. Un guide du programmeur, un manuel de l'administrateur et un tutorial complètent la documentation. L'outil met l'accent sur la standardisation, les mécanismes de communication entre les agents, le déploiement et l'optimisation. Les premières phases du développement sont laissées à la discrétion de l'utilisateur. Un éditeur est disponible pour l'enregistrement des agents et leurs suivis (il est possible de voir le parcours suivi par les messages). Il est également possible d'envoyer des messages aux agents via une interface utilisateur. Cependant, aucune autre interface n'est disponible pour le développement ou l'implémentation. Par le fait même, l'implémentation est difficile et demande beaucoup d'efforts. Elle nécessite une bonne connaissance des classes et des différents services offerts.

3.6.7 JAFMAS et JiVE

<JAFMAS> JAFMAS met l'emphase sur les protocoles de communication, l'interaction entre les agents, la coordination et la cohérence à l'intérieur du système [Chauhan97], [Galan], [Galan00], [Baker]. L'éditeur graphique (JiVE) est un outil de support pour le développement qui propose une interface aidant l'utilisateur dans sa démarche. Une particularité de JiVE (JAFMAS integreted Visual Environment) est la possibilité de travailler en groupe sur un projet. La documentation sur ces deux outils est déficiente. Les réseaux de Pétri et l'utilisation de COOL rendent la création de conversations et la coordination très complexes. Aucun support pour le déploiement n'est disponible.

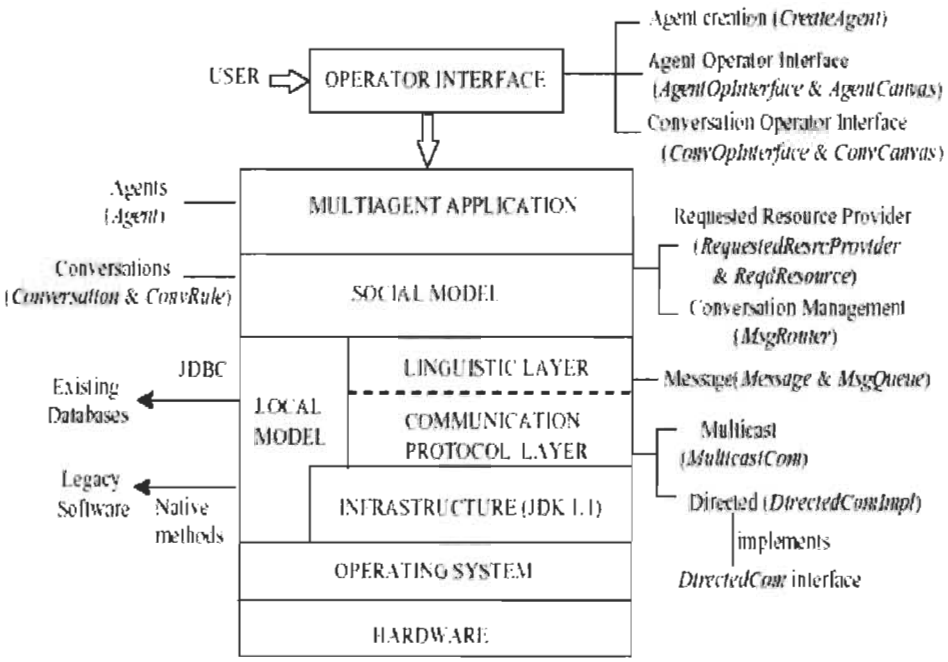

Figure 17 : Architecture JAFMAS

(A Multiagent Application Development System [Baker])

3.6.8 MadKit

<MadKit> Madkit est un environnement basé sur la méthodologie Aalaadin ou AGR [Ferber97], [Ferber97b], [Ferber97c], [Gutknecht01]. L'outil fourni un éditeur permettant le déploiement et la gestion des SMA (G-box). La documentation offerte couvre les principaux points de l'environnement. De plus, l'outil offre un utilitaire pour effectuer des simulations à grande échelle.

3.6.9 RMIT

Cet outil se veut un *framework* pour le développement de SMA. La méthodologie est incomplète. Elle ne se situe qu'au niveau des agents qui devraient être développés en sept couches : les sens, les croyances, le raisonnement, les actions, la collaboration, la translation et la mobilité. Le cadre proposé est très complexe et aucun outil d'aide n'est disponible. De plus, la documentation est presque inexistante.

3.6.10 Zeus

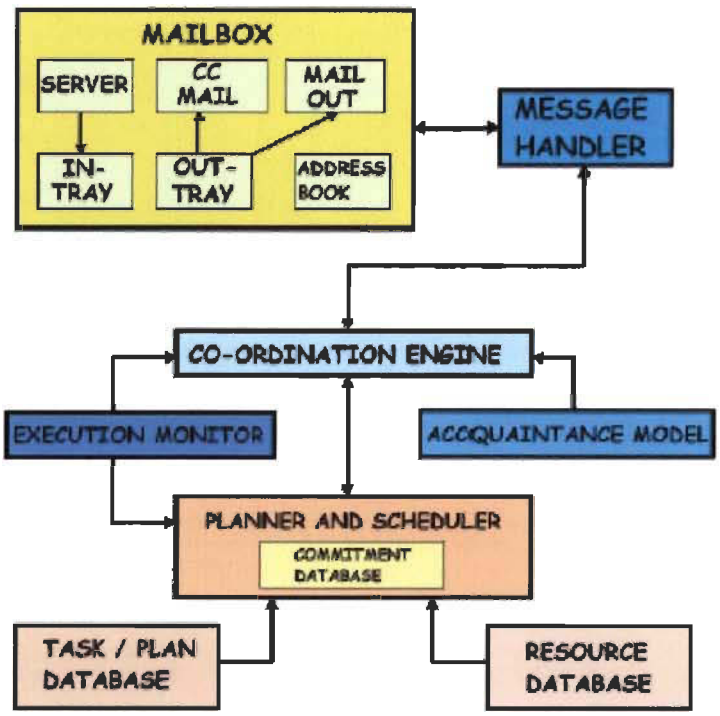

Figure 18 : Architecture d'un agent Zeus

(Manuel technique [ZEUS T. M.])

<Zeus> Zeus est un environnement complet qui utilise la méthodologie *Role Modeling* pour le développement de systèmes collaboratifs [ZEUS A.R.G.], [ZEUS R. M..G.], [ZEUS R. G.], [ZEUS T. M.]. L'outil couvre toutes les phases du processus de développement [Collis98], [Collis98b], [Collis99]. Les différentes étapes du

développement se font à l'intérieur de plusieurs éditeurs : ontologie, description des tâches, organisation, définition des agents, coordination, faits et variables ainsi que les contraintes. L'outil est assez complexe et sa maîtrise nécessite beaucoup de temps et d'efforts. Zeus possède une documentation complète qui couvre les différentes étapes dans l'élaboration de système selon la méthodologie suggérée. La documentation propose quelques exemples pratiques d'utilisation de l'outil. Cette méthodologie est parfois difficile à utiliser et à appliquer. Le développement avec Zeus est cependant conditionnel à l'utilisation de la méthodologie *Rôle Modeling* ainsi qu'à l'un de ses trois patrons proposés.

3.6.11 ADE (Agent Development Environment)

ADE est un environnement de développement de systèmes multi-agents distribués. L'outil fourni un environnement graphique pour le développement OO. ADE fourni un langage ADEGraphcet qui utilise les réseaux de Pétri pour la spécification du comportement des agents.

3.6.12 Autres outils

<MASSYVE Kit> MASSYVE Kit (Multiagent Agile manufacturing Scheduling System for Virtual Entreprise), <ADK> ADK (Agent Development Kit), <CABLE> CABLE, <Comet Way JAK> Comet Way JAK, <Cougaar> Cougaar, <FIPA OS> FIPA OS, <SIM_AGENT> SIM_AGENT et bien d'autres [Mangina02].

3.7 Objectifs, faiblesses et manques de ces outils

La majorité des outils développés le furent pour exploiter ou démontrer un concept ou une idée en particulier. De ce fait, le développement de ces outils négligent, volontairement ou non, le développement de plusieurs dimensions essentielles à l'implémentation d'un SMA. Ceci rend leur utilisation inappropriée voir même impossible pour le développement de systèmes réels.

3.8 Vers des environnements plus complets

Comme mentionné ci-haut, la majorité des outils furent développés pour combler des besoins spécifiques. Les concepts comme le langage de communication, les protocoles d'interaction, les bases de connaissances et autres se rapprochent de la standardisation. Il est maintenant temps de fournir des environnements plus complets supportant toutes les étapes du développement de SMA (développement, implémentation et déploiement).

Il est impératif de développer des environnements complets qui :

- Mettent l'accent sur la facilité et la simplicité du développement.
- Fournissent aux programmeurs plusieurs options pré-implémentées.

- Offrent plusieurs types d'agents ayant chacun des caractéristiques et des comportements différents.
- Fournissent une implémentation appropriée de ceux-ci.
- Définissent des mécanismes (encapsulés) simples de communication inter-machines (sockets, multicast, RMI, etc).
- Permettent la génération automatique des sources (bases de connaissances, mécanismes de communication, squelettes des méthodes nécessaires, attributs et spécifications du système et des agents).
- Permettent une bonne extensibilité du code généré et offrent quelques mécanismes d'interaction (entre les agents) simples et extensibles.
- Offrent un utilitaire d'aide intégré
- Permettent l'implémentation, la compilation et l'exécution du système créé.

De cette façon, on obtient des environnements qui combinent les principales qualités des meilleurs outils tout en palliant à leurs faiblesses par des ajouts substantiels.

Bien sûr, cette idée représente une vision « idéaliste » des environnements de développement de SMA. Cependant, un constat est évident : les outils existants ne permettent pas encore le développement complet et, de façon relativement simple, de SMA appliqués à des systèmes réels d'envergure intéressante. De ce fait, la programmation OA n'intéresse que ceux qui étudient le domaine des agents ou un domaine connexe. Plusieurs éléments essentiels à un environnement de développement de SMA existent déjà dans l'un ou l'autre des différents outils présentés. Un mariage des différents concepts et caractéristiques de ceux-ci auxquels seraient ajouté quelques utilitaires laisse entrevoir le développement d'outils encore plus adéquats et productifs.

3.9 Synthèse de l'état de l'art

Les concepts d'agent, SMA et de programmation OA sont très abstraits et impliquent plusieurs notions complexes comme la coordination, l'interaction, la communication, la représentation des connaissances et autres. Dans ce chapitre, nous avons fait un survol rapide de quelques-uns des principaux ouvrages reliés au domaine des agents. Comme nous l'avons vu, plusieurs utilitaires ont été développés pour supporter différentes caractéristiques de ce genre de systèmes. Cependant, il reste encore beaucoup de travail à faire aux niveaux de la standardisation et de l'uniformisation. De plus, les équipes de chercheurs ont axé leurs recherches sur des sujets précis et pointus, ce qui s'est reflété sur les outils et environnements qu'ils ont développés. Aucun de ces outils ne peut être utilisé de façon simple, rapide, efficace et performante pour le développement, l'implémentation et le déploiement de SMA totalement distribués.

4. Solution proposée : Fournir un ED de SMA complet

Comme mentionné à plusieurs reprises, les outils développés pour la programmation OA sont incomplets et ne peuvent être utilisés pour le développement, l'implémentation et le déploiement de SMA. Une approche plus généraliste visant à fournir un environnement de développement de SMA complet (développement, implémentation et déploiement) a été choisie.

L'outil développé se rapproche, à quelques niveaux, des environnements de développement d'applications OO comme VisualCafé ou JBuilder.

Similitudes

Comme ces ED, l'outil permet :

- De créer des projets.
- D'ajouter des fichiers existants au projet (fichiers .java ou autres ressources).
- De créer des fichiers et de les ajouter au projet (fichiers .java ou autres ressources).
- D'implémenter (de coder) des applications en Java à l'intérieur de ces fichiers.
- De compiler un fichier en particulier.
- De compiler un projet en entier.
- D'exécuter le projet (en spécifiant la classe à exécuter).
- D'initialiser plusieurs propriétés de l'environnement comme le chemin du programme d'exécution (*java* ou *javaw*), du programme de compilation (*javac* ou *rmic*), des fichiers sources (*sourcepath*), des classes (*classpath*), etc.
- D'utiliser les classes de l'API (Application Programming Interface) Java 1.4.1 dans les applications.
- D'obtenir de l'aide (*JavaDoc*) sur les différentes classes de Java.

Additions de l'outil

Contrairement aux environnements de développement standards, l'outil permet :

- Le développement simple et rapide d'applications distribuées.
- La spécification, via des interfaces, des différents sous-systèmes, de leurs agents, des bases de connaissances, des modes de communication entre les différentes machines et autres.
- Le développement OA grâce à la librairie de classes OA (environ 200 classes) intégrée.
- L'extensibilité des classes de la librairie OA.
- La validation des spécifications.
- La génération complète du code source des composants du système.
- L'exportation du système permettant le développement et l'exécution du système indépendant de l'outil.

- L'archivage des différents sous-systèmes et la création d'un exécutable pour chaque JVM. Ceci permet un déploiement extrêmement simple d'applications totalement distribuées.
- Plusieurs autres options non-disponibles sur les autres environnements de développement (OO ou OA) simplifiant la préparation et l'initialisation de l'outil comme la recherche automatique des programmes nécessaires à l'ED (*java, javaw, rmic, javac, jar* et autres), l'initialisation automatique du *classpath* et du *sourcepath*.
- L'exécution de l'environnement sur différentes plate-formes (Windows, Unix, Linux, etc.) sans aucune modification.
- L'exécution des sous-systèmes (fichier exécutable .jar) sur les différentes plate-formes (Windows, Unix, Linux, etc.) sans aucune modification.

4.1 Environnement pour l'implémentation d'applications Java

L'outil développé est implémenté en Java à 100% et permet le développement d'applications ou de SMA distribués en Java. Le langage Java a été choisi pour plusieurs raisons (voir section 3.4). Java est un langage OO, portable, très fiable, sécuritaire, extensible, polyvalent et de plus en plus performant grâce à l'amélioration constante des compilateurs et interpréteurs Java (Figure 19).

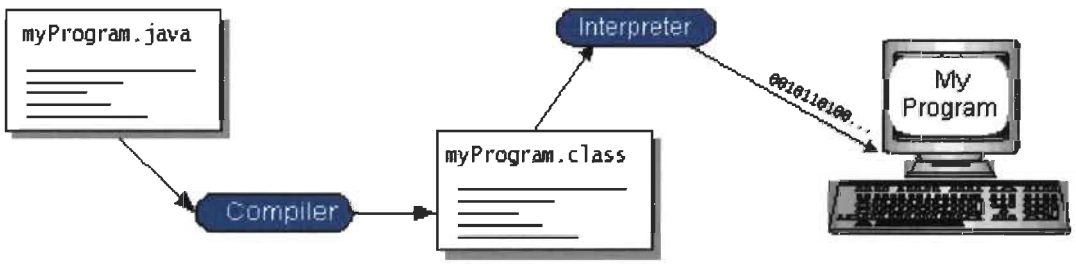

Figure 19 : Composants clés dans la performance en Java (compilateur et interpréteur)

(Tutorial Java <Java>)

Plusieurs évaluations démontrent que Java, en termes de rapidité d'exécution, se rapproche et même quelques fois surpasse les performances d'applications équivalentes implémentées en C pur. De plus, le développement d'applications en Java est beaucoup plus rapide que la majorité des langages de programmation. Le choix était donc facile quant au langage utilisé pour l'environnement et la librairie de classes.

4.2 Librairie de classes pour le développement OA

Une librairie de plus de 200 classes permet l'implémentation simple et rapide de SMA totalement distribués. La librairie supporte le développement de plusieurs types d'agents (BDI et autres). Un support pour le développement de bases de connaissances est aussi

fourni pour ces agents. Plusieurs comportements pré-implémentés s'intègrent facilement aux agents. Un mécanisme générique d'attribution et d'exécution de tâches permet l'ajout d'habilités aux différents agents du système. La création de communications inter-machines est extrêmement simplifiée grâce à un ensemble de classes encapsulant les différents types de communication possibles entre ordinateurs (*socket* TCP, *socket* UDP, RMI, *socket multicast*, CORBA). Plusieurs classes utilitaires sont aussi disponibles. Par exemple, une classe permettant l'envoi et la réception de messages dans la file des événements AWT (Abstract Window Toolkit) de Java. Un autre exemple est le ANS (Agent Name Server), une instance d'une classe créée automatiquement contenant des informations utiles sur les adresses et modes de communication des agents d'un système. Les DF (Directory Facilitator) et les RA (Register Agent) sont aussi des exemples d'utilitaires essentiels aux SMA que cette librairie fournie. Dans le prochain chapitre, nous verrons en détails la description et l'utilité de chacune de ces classes.

L'utilité de cette librairie est double. Premièrement, elle facilite la génération automatique du code source du système. En générant un ensemble de classes, qui pour la plupart dérivent de classes de cette librairie, la quantité de code générée est moindre et la compréhension de celui-ci est simplifié. Deuxièmement, elle permet à ceux qui veulent implémenter des SMA sans l'utilisation de l'environnement, d'incorporer la librairie à leur outil de développement (Java SDK, VisualCafé, Forte ou autre) et utiliser les facilités offertes par celle-ci.

4.3 Interfaces pour le développement et l'implémentation d'un MAS

Une des grandes forces de l'environnement est la possibilité de spécifier, via des interfaces, les composants du système. Les interfaces permettent :

- De spécifier les adresses IP des sous-systèmes.
- De spécifier les différentes JVM (sous-systèmes).
- De spécifier les types d'agents.
- De créer les agents et leurs différentes propriétés.
- De spécifier les comportements des agents
- De créer les tâches des agents.
- De créer en totalité les différentes bases de connaissances utilisées par certains agents (lorsque nécessaires).
- De lier les adresses IP avec les JVM.
- De déterminer quel type de DF est nécessaire pour chaque JVM (si on désire en ajouter un).
- De déterminer la façon dont les agents s'enregistrent auprès des autres JVM.

D'autres possibilités sont aussi disponibles. La description et la visualisation de chacune sera faite dans le prochain chapitre.

4.4 Plusieurs fonctionnalités automatisées

4.4.1 Validation des spécifications

Une fois les spécifications du système complétées, une validation permet d'obtenir un *feedback* sur le système qui sera généré. La validation permet de vérifier l'ensemble des noms utilisés dans le système. Cette étape vérifie que l'ensemble des noms des composants du système soient valides (que ceux-ci ne provoquent aucune erreur de compilation). De plus, la validation vérifie plusieurs erreurs de logiques comme la non-duplication des adresses IP, des noms et GUID, la logique des bases de connaissances, la logique communicative (les modes, ports et adresses utilisés), etc. De cette façon, plusieurs erreurs de conception peuvent être corrigées avant la génération du code et la poursuite de l'implémentation.

4.4.2 Génération automatique du code source du SMA

La génération automatique du code source du système et de ses composants accélère considérablement l'implémentation de l'application. Il est évident que la spécification du système serait inutile sans une génération de celui-ci. Une fois les spécifications terminées et la validation effectuée, la génération du code source permet de créer tous les fichiers sources nécessaires à chaque sous-système. Un répertoire est créé pour chaque JVM. Chacun de ces répertoires contient les fichiers d'un sous-système en particulier. Une fois les fichiers générés, la compilation peut être faite (si la validation ne mentionne aucune erreur, aucune erreur de compilation ne se produira).

4.4.3 Exportation des sous-systèmes

Une fonctionnalité permet de rendre chaque sous-système indépendant de l'ED. Une fois cette opération effectuée, chaque sous-système est totalement indépendant de l'outil. L'implémentation de cette JVM peut alors se continuer sur n'importe quel ordinateur possédant un environnement de développement Java standard SDK (Standard Development Kit). L'exécution peut être effectuée sur un ordinateur ne possédant qu'une JRE (Java Runtime Environment) de base.

4.4.4 Archivage des sous-systèmes

L'archivage permet de créer automatiquement une archive compressée Java « .jar » exécutable totalement indépendante de l'ED. En exécutant cette option, une archive est créée pour chaque JVM. De cette façon, il suffit d'amener l'archive d'une JVM sur la machine où elle sera déployée et l'exécutée (ligne de commande simple : ***java –jar <nom_de_fichier>.jar***). Cette fonctionnalité est primordiale si l'on veut simplifier au maximum le déploiement de SMA distribués.

4.5 Documentation

L'environnement offre une documentation adéquate de l'outil et de ses différentes classes. Il est essentiel de fournir une documentation facilement accessible de l'environnement. De plus, il est utile d'offrir la documentation du langage de programmation (ici on parle de Java, donc permettre la consultation de la JavaDoc de l'API java). La documentation doit aussi fournir un index permettant une recherche simple et efficace. La documentation des classes de l'environnement doit être séparée de celle du langage de programmation, ce qui permet une recherche ciblée plus rapide et efficace.

5. Développement de l'environnement

Ce chapitre détaille l'outil développé et ses différentes caractéristiques. Dans cette section, nous verrons plusieurs détails reliés à l'implémentation de l'outil et à l'architecture développée. De plus, les différents choix qui ont guidé le développement de l'environnement seront abordés.

5.1 Architecture développée

Au début de nos recherches sur les outils SMA, nous supposions que les outils SMA devaient proposer une méthodologie pour le développement. Cependant, au fil des évaluations [Garneau02] et après plusieurs discussions avec des collègues travaillant dans le domaine des SMA et avec des équipes utilisant ce type d'outil, nous avons conclu qu'il est préférable de n'imposer aucune méthodologie spécifique à l'utilisation d'un environnement de développement de SMA. En effet, d'après plusieurs chercheurs et collègues, la méthodologie est souvent dépendante du domaine d'application. De ce fait, l'imposition d'une telle restriction à l'intérieur d'un environnement de développement de SMA limite les domaines d'application dans lesquels celui-ci peut être utilisé. Par contre, nous ne minimisons aucunement l'utilité et l'importance des méthodologies. Nous affirmons seulement que la méthodologie utilisée devrait être laissée libre à l'utilisateur. L'environnement n'impose donc aucune méthodologie spécifique au développement du SMA [Garneau03].

Cependant, il faut suivre une certaine structure pour le développement de SMA avec l'outil. L'utilisation de l'environnement nécessite le développement en sous-systèmes, permettant ainsi le développement de SMA totalement distribués. Chaque agent possède un système de communication indépendant des autres. De ce fait, chaque agent peut communiquer avec d'autres agents ou composants Java. De plus, chaque agent détermine la façon dont il reçoit les messages provenant des autres sous-systèmes. Un agent peut écouter via RMI, TCP, UDP, Multicast ou en mode direct. Cependant, il est souvent utile et plus efficace de mettre un DF. Celui-ci fait le routage des messages sur une machine lorsque plusieurs agents d'une JVM veulent communiquer avec des agents situés sur d'autres ordinateurs. L'environnement permet de créer plusieurs types de DF qui offrent différentes fonctionnalités.

Il est impossible de créer plus d'un sous-système par ordinateur. Nous avons déterminé qu'il n'y avait aucune raison valable d'exécuter plus d'une JVM sur la même machine. En fait, exécuter deux JVM sur le même ordinateur provoquerait un comportement erroné. Pour cette raison, dans notre cas, parler d'une JVM ou d'un sous-système est équivalent. Chaque sous-système est constitué d'un ou plusieurs agents. Pour chaque agent, il est nécessaire de définir les tâches que celui-ci peut effectuer. L'implémentation des agents est donc orientée-tâche contrairement à d'autres architectures qui sont orientées-buts ou orientées-rôles.

Nous reviendrons en détail sur l'architecture lors de la description de la librairie de classes OA développée. Cette brève introduction permet d'avoir un premier aperçu de l'architecture mise en œuvre (Figure 20).

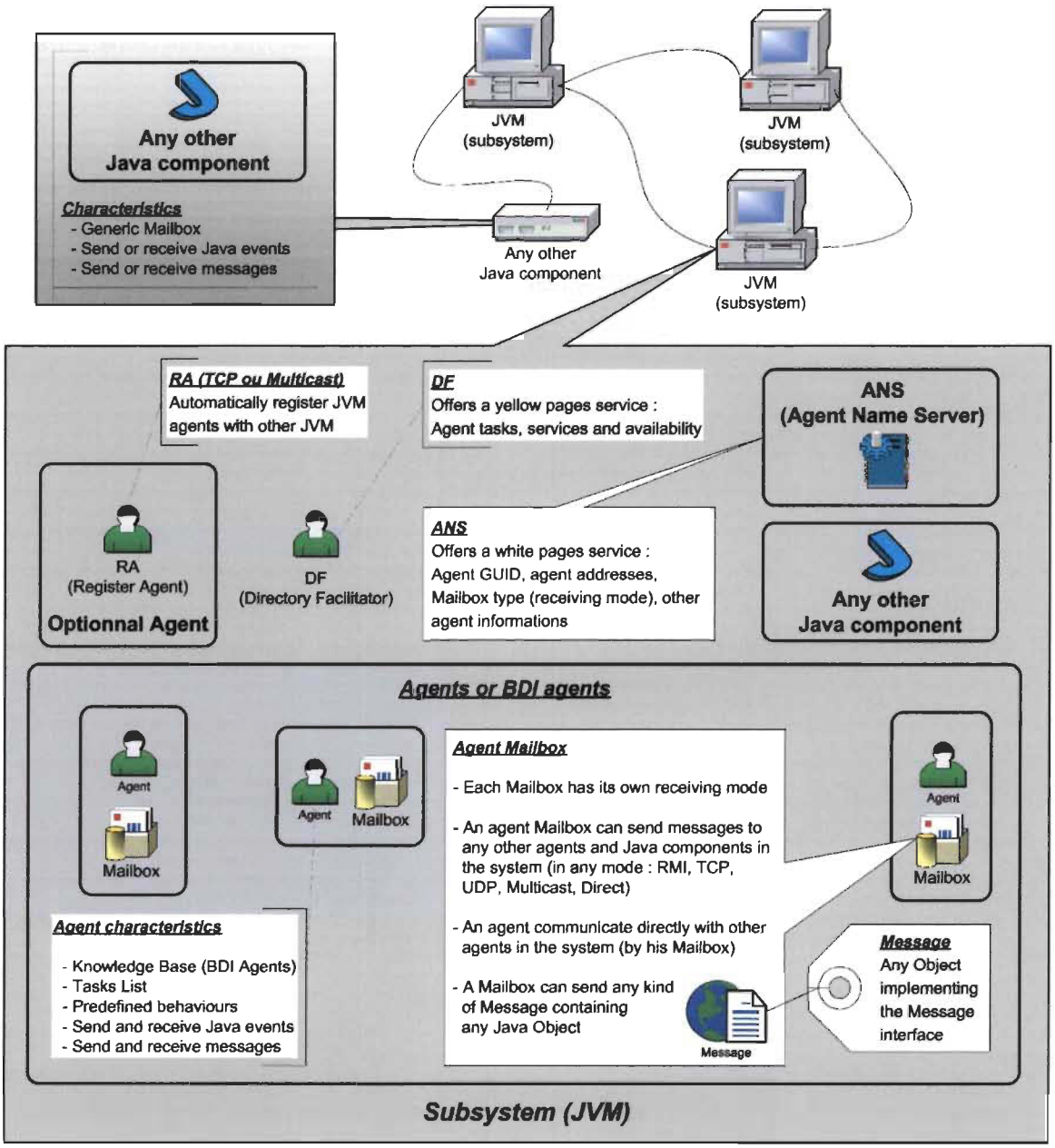

Figure 20 : Représentation des composants d'un SMA développé avec DMAS Builder

L'environnement est totalement implémenté en Java et permet le développement de SMA en Java. On peut diviser l'environnement en deux composants principaux : l'environnement de développement et la librairie de classes pour le support à la programmation OA.

5.2 *L'environnement de développement*

Un peu comme les environnements de développement d'applications populaires (JBuilder, VisualCafe ou autres), l'environnement développé est un utilitaire graphique permettant le développement d'applications Java standards et / ou de systèmes à base d'agents en Java totalement distribués. L'outil possède plusieurs interfaces utilisateur. Commençons par regarder les options accessibles à partir de l'environnement indépendamment d'un projet, c'est-à-dire qu'il n'est pas nécessaire de créer un projet (application ou SMA) pour accéder à ces options (comme les options en rapport avec la JVM, utilisation de l'aide, etc.).

5.2.1 Fonctionnalités générales (indépendantes d'un projet)

Voici l'interface de départ (Figure 21), celle qui apparaît à l'écran lorsque l'on exécute l'environnement :

5.2.1.1 Environnement de départ

Figure 21 : Interface apparaissant lors de l'exécution de l'environnement

Normalement, chaque menu écrit d'une teinte plus faible que les autres nécessite qu'un projet ait préalablement été ouvert pour permettre leur utilisation. Dans un souci de clarté et de lisibilité, les captures d'écrans ont été faites lorsqu'un projet était ouvert.

Cependant, pour différencier les deux contextes (aucun projet ouvert versus un projet en cours), les descriptions suivies du mot *(projet)* spécifient qu'un projet doit être ouvert pour avoir accès à celles-ci.

5.2.1.2 Menu Fichier

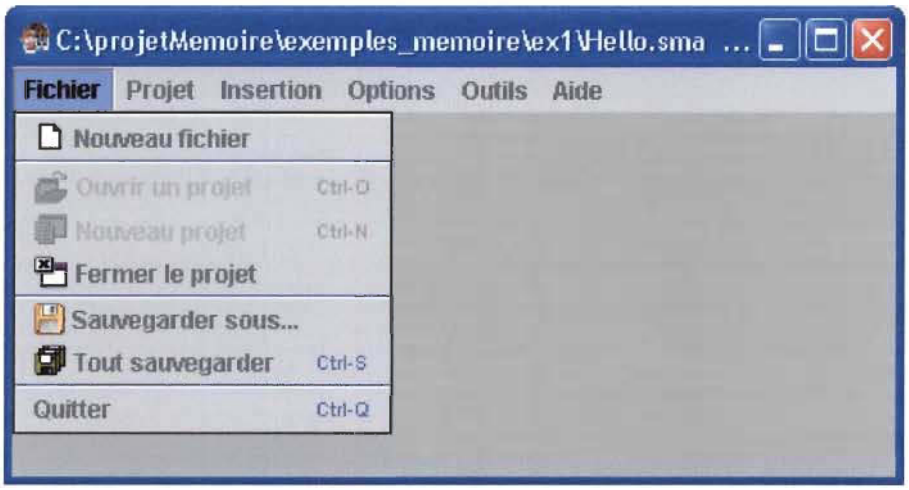

Figure 22 : Options du menu Fichier

Nouveau fichier

- Permet de créer un fichier et de l'ajouter au projet *(projet)*.

Ouvrir un projet

- Permet d'ouvrir un projet déjà existant.

Nouveau projet

- Permet de créer un nouveau projet.

Fermer le projet

- Permet de fermer un projet *(projet)*.

Sauvegarder sous

- Permet d'enregistrer un projet et tous ses fichiers dans un autre répertoire (recopie tous les fichiers du projet) *(projet)*.

Tout sauvegarder

- Permet d'enregistrer tout le projet dont ses fichiers, ses options et ses spécifications *(projet)*.

Quitter

- Permet de fermer l'environnement.

5.2.1.3 Menu projet

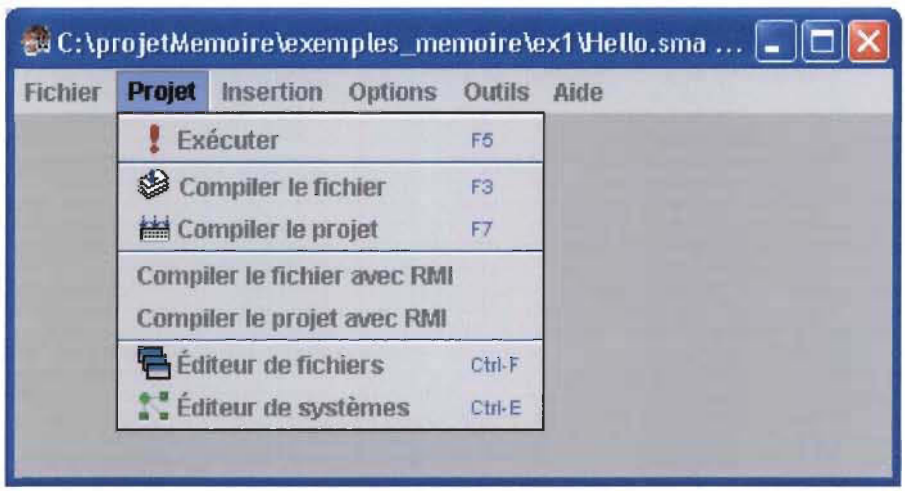

Figure 23 : Options du menu Projet

Exécuter

- Exécute le projet courant, la classe à exécuter est spécifiée dans l'éditeur des options du projet *(projet)*.

Compiler le fichier

- Compile le fichier qui est présentement édité dans l'éditeur des fichiers *(projet)*.

Compiler le projet

- Compile tout le projet *(projet)*.

Compiler le fichier avec RMI

- Compile avec RMIC (compilateur RMI) le fichier qui est présentement édité dans l'éditeur des fichiers *(projet)*.

Compiler le projet avec RMI

- Compile tout le projet avec RMIC (compilateur RMI) *(projet)*.

Éditeur de fichiers

- Permet d'accéder à l'éditeur de fichiers où se fait l'édition et l'implémentation des fichiers du projet *(projet)*.

Éditeur de systèmes

- Permet d'accéder à l'interface où se fait la spécification des différents composants du système *(projet)*.

5.2.1.4 Menu Insertion

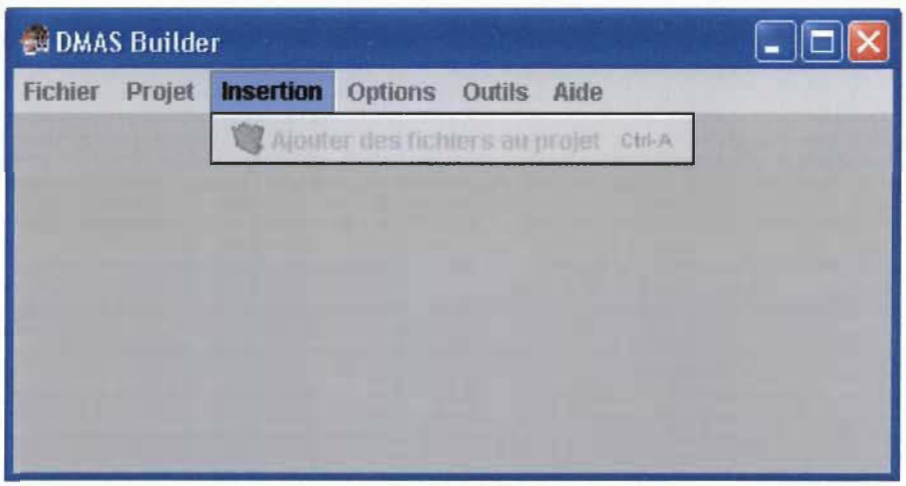

Figure 24 : Option du menu Insertion

Il est à noter que le menu insertion ne contient qu'une option pour l'instant. Par contre, dans la prochaine version, ce menu permettra l'insertion d'autres types de composants. C'est la principale raison pour laquelle le menu n'est pas fusionné avec un autre.

Ajouter des fichiers au projet

- Permet d'ajouter un ou plusieurs fichiers au projet *(projet)*.

5.2.1.5 Menu Options

Plusieurs éditeurs simplifient l'initialisation des différents paramètres du système comme l'emplacement des programmes nécessaires à l'exécution (*java* ou *javaw*), à la compilation (*javac* et *rmic*), à l'archivage (*jar*) et autres.

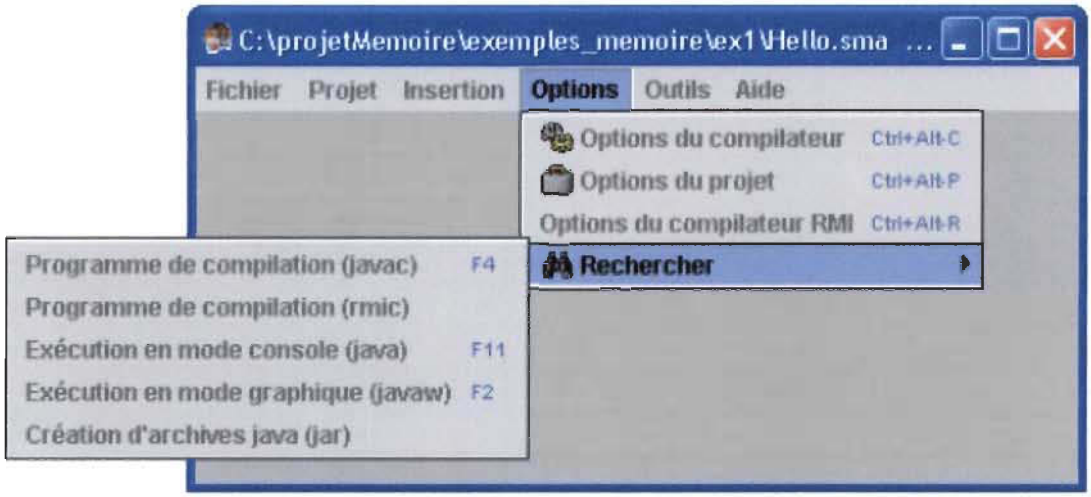

Figure 25 : Options du menu Options

Options du compilateur

- Cette fenêtre offre une façon simple de choisir et d'initialiser le compilateur nécessaire à l'environnement pour la compilation des applications Java (voir figure 26). De plus, cette interface permet de spécifier les dossiers et fichiers contenants les classes de l'application (fichiers sources et binaires). On peut déterminer, en choisissant parmi les différentes options, les paramètres à envoyer au compilateur.

Chemin javac

- Le chemin du programme à utiliser pour la compilation des applications Java.

Répertoires des fichiers .class (classpath)

- Répertoires et fichiers « .jar », « .zip » où se trouvent les fichiers binaires nécessaires à la compilation des applications.

Répertoires des fichiers .java (sourcepath)

- Répertoires où se trouvent les fichiers sources « .java » à compiler. Ici, il n'est pas nécessaire de mettre les répertoires des fichiers du projet car, lors de la compilation, les chemins de tous les fichiers sources de l'application seront spécifiés automatiquement par l'environnement au compilateur. Cependant, cette option est très utile lors de l'exportation du projet (voir section 5.3.10.3). Il est aussi utile pour spécifier de compiler les fichiers sources de ces répertoires même si des fichiers binaires correspondants existent dans le *classpath* (voir la documentation et la spécification Java pour plus de détails). Cette option permet aussi de spécifier plusieurs répertoires, ceux-ci sont séparés par un point-virgule au lieu d'être dans une liste comme le *classpath*. Si l'on ajoute un dossier, il sera concaténé avec le contenu de la boîte d'édition.

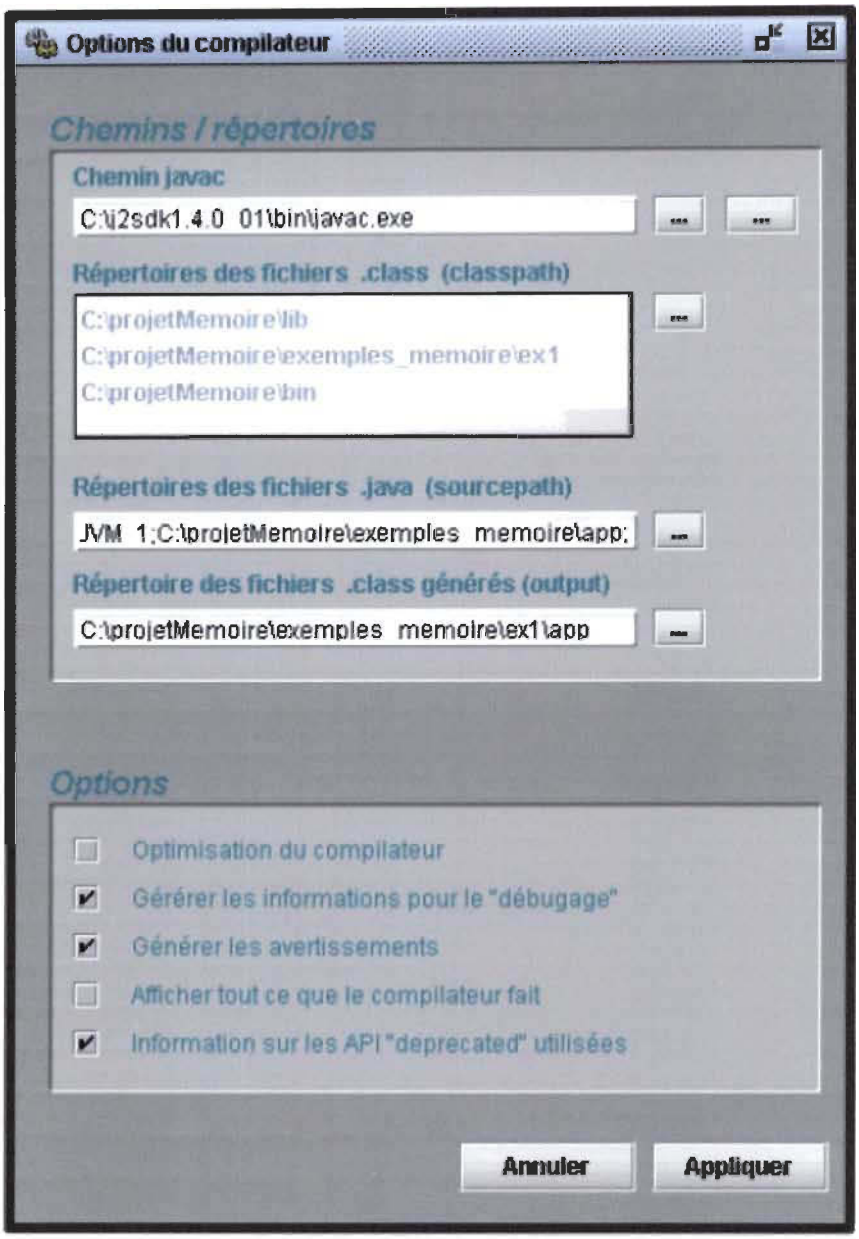

Figure 26 : Fenêtre permettant l'initialisation simple des paramètres du compilateur

Répertoire des fichiers .class générés (output)

- Répertoire où écrire les fichiers binaires créés par le compilateur.

Options

- Les cinq cases à cocher sont des indicateurs (*flags*) que l'on peut spécifier au compilateur. Il suffit de cocher les cases correspondantes aux options que le compilateur doit prendre en considération.

Options du projet

- Permet d'afficher la fenêtre des options du projet *(projet)*.

Options du compilateur RMI

- Plusieurs des options du compilateur RMI sont semblables aux options du compilateur *javac*.

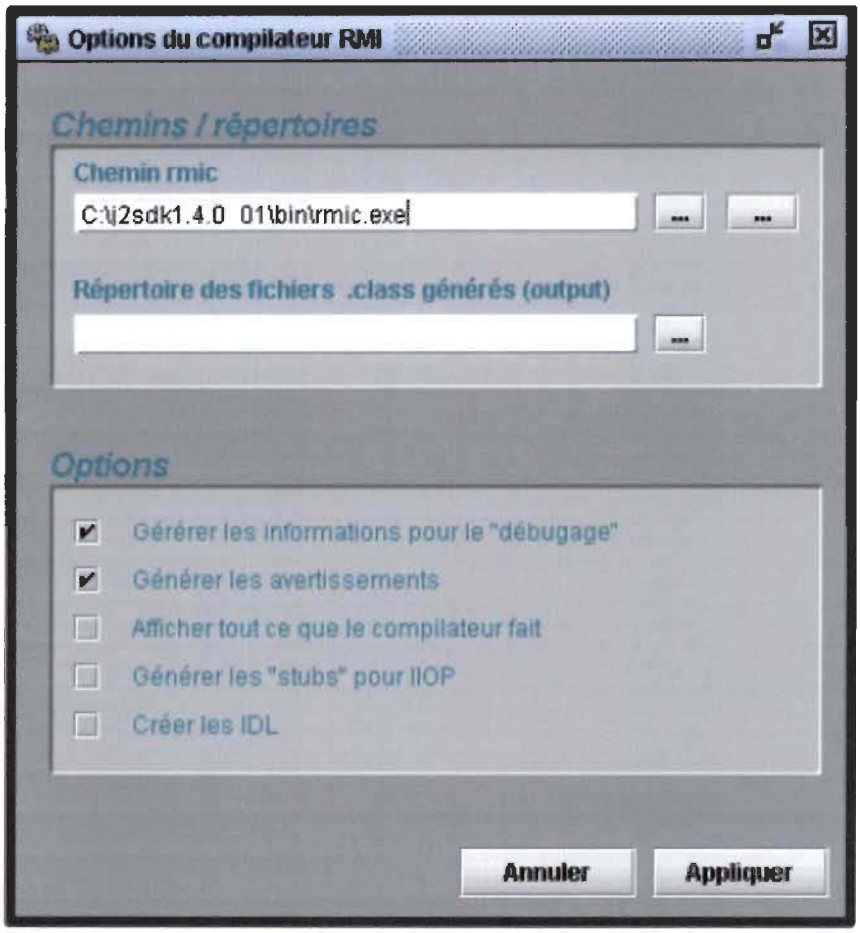

Figure 27 : Fenêtre permettant l'initialisation simple du compilateur RMI

Chemin rmic

- Le chemin du programme à utiliser pour la compilation des composants RMI des applications Java.

Répertoire des fichiers « .class » générés (output)

- Répertoire où écrire les fichiers binaires créés par le compilateur RMI.

Options

- Les cinq cases à cocher sont des indicateurs (*flags*) que l'on peut spécifier au compilateur RMI. Il suffit de cocher les cases correspondantes aux options que le compilateur doit prendre en considération.

Rechercher

- L'environnement possède un utilitaire de recherche et d'initialisation automatique des différents programmes nécessaires à l'environnement. Ceci facilite grandement les différentes initialisations du système et permet une grande flexibilité quant à l'utilisation de la version du SDK utilisé (Figure 28).

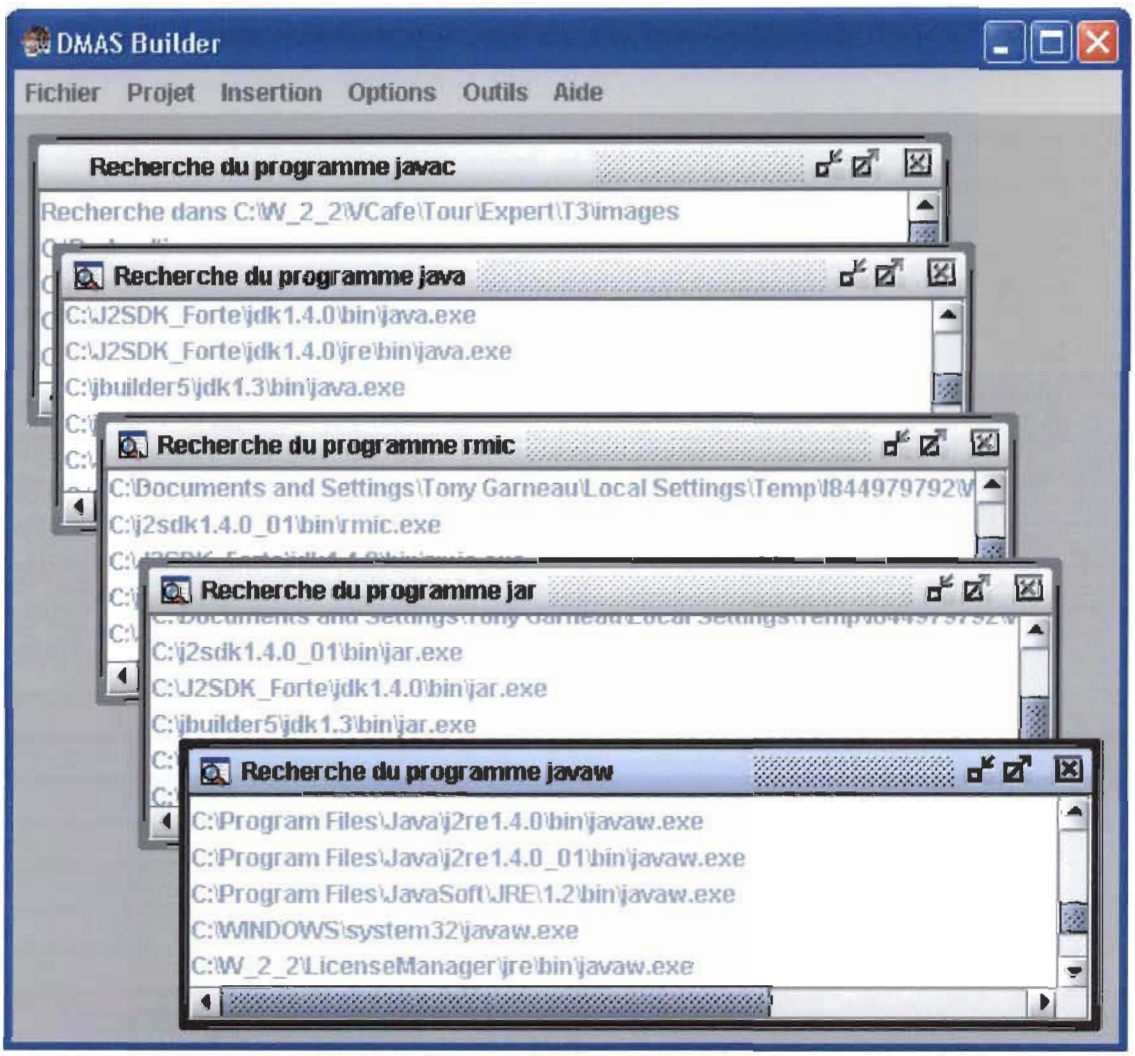

Figure 28 : Recherche automatique des programmes nécessaires à l'ED

Recherche de javac

- Recherche automatique du programme nécessaire à la compilation des applications Java.

Recherche de rmic

- Recherche automatique du programme nécessaire à la compilation des composants RMI des applications Java.

Recherche de java

- Recherche automatique du programme nécessaire à l'exécution des applications Java en mode console.

Recherche de javaw

- Recherche automatique du programme nécessaire à l'exécution des applications Java en mode graphique.

Recherche de jar

- Recherche automatique du programme nécessaire à la création des archives Java.

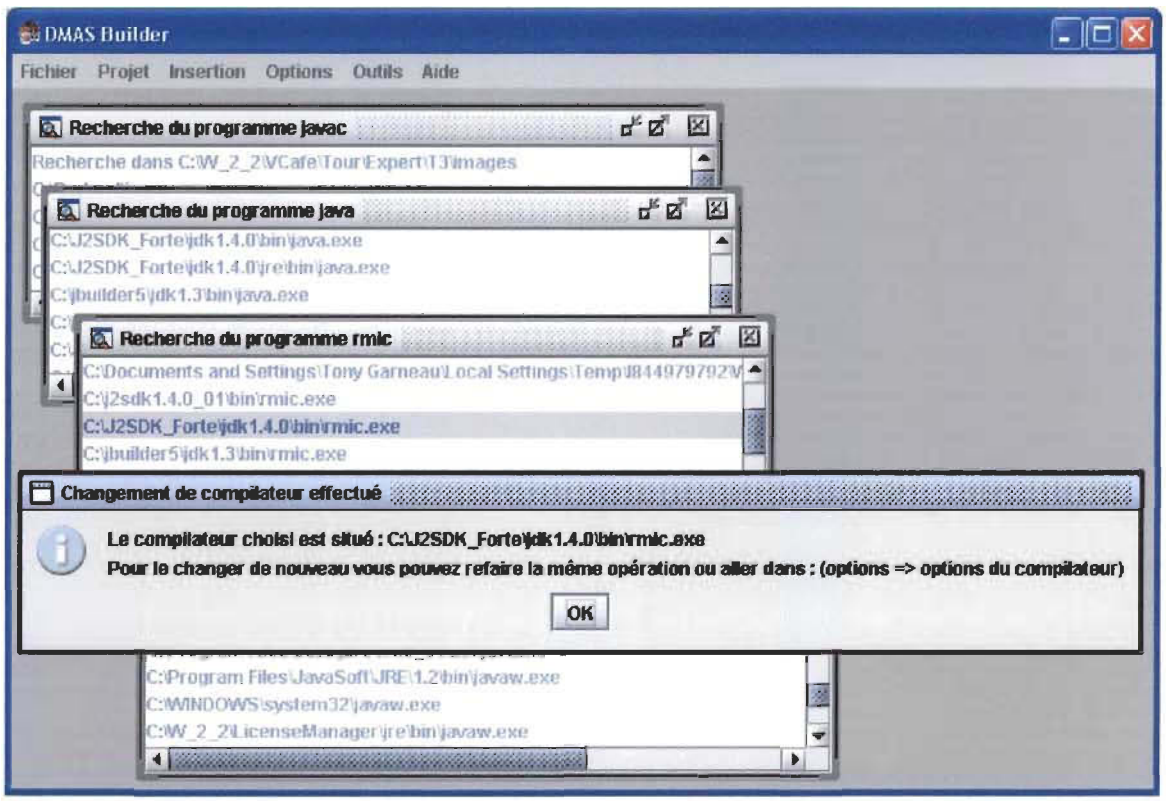

Figure 29 : Message de confirmation du changement de programme

Une fois que la recherche commence à donner des résultats, l'utilisateur doit simplement cliquer sur le programme désiré. Un message confirmant l'opération apparaîtra à l'écran comme dans la figure 29.

5.2.1.6 Outils

Ce menu ne contient qu'une option pour l'instant. Cependant, il sera enrichi de plusieurs autres options dans la prochaine version.

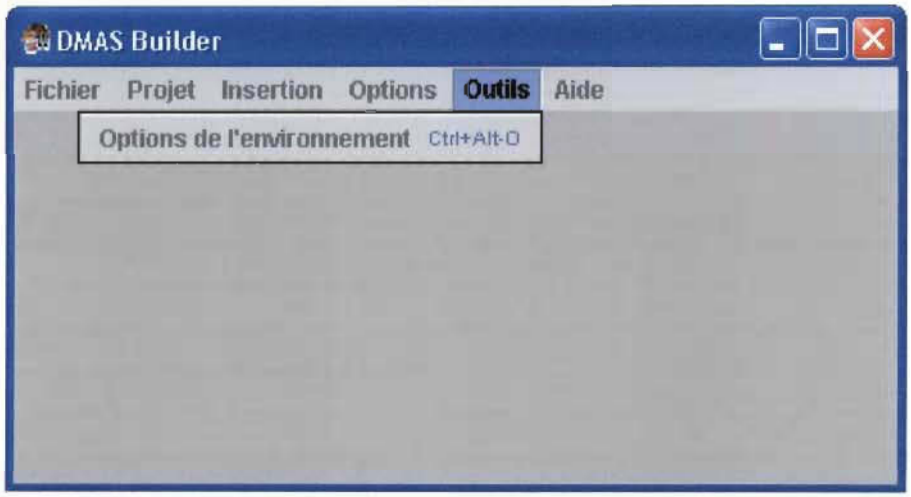

Figure 30 : Option du menu Outils

Options de l'environnement

- Un éditeur permet de personnaliser l'environnement à l'aide de plusieurs options.

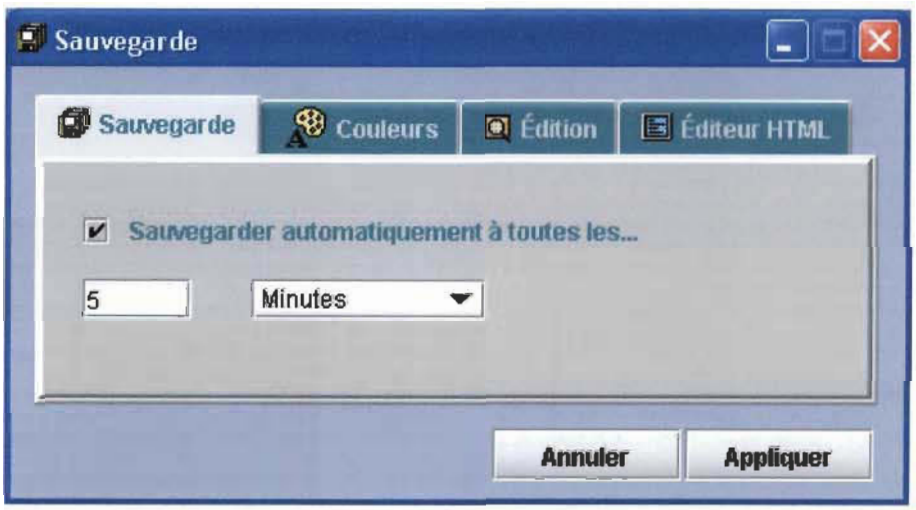

Figure 31 : Interface permettant de spécifier la sauvegarde automatique des projets

Sauvegarde automatique

Si cette option est activée (en sélectionnant la case à cocher), la sauvegarde du projet se fera automatiquement à tous les intervalles de temps spécifiés. Si l'option n'est pas cochée, la sauvegarde du projet ne se fera que si on le demande explicitement (Figure 31).

Choix des couleurs

Le choix des couleurs permet de déterminer la couleur des différentes catégories de mots à l'intérieur des fichiers de code source « .java » (Figure 32). De cette façon, il est possible de choisir une couleur en particulier pour les mots clés du langage Java tout en spécifiant d'autres couleurs pour les autres catégories de mots. Les catégories disponibles sont les mots clés du langage Java, les noms de classes de la librairie de classes OA, les commentaires et les chaînes de caractères. Il est également possible de déterminer si une catégorie de mots est éditée en caractères gras. Le choix de la taille des caractères est imputable à tout le fichier et non à une catégorie de mots en particulier.

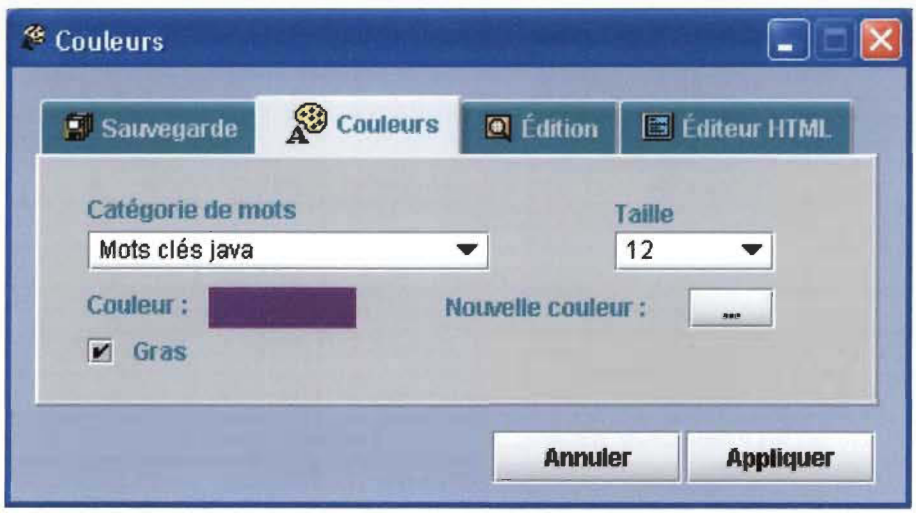

Figure 32 : Options des couleurs du texte pour l'édition des fichiers sources Java

Pour changer la couleur d'une catégorie de mots, il suffit de cliquer sur le bouton nouvelle couleur. Par la suite, la fenêtre des choix de couleurs apparaît à l'écran comme présenté à la figure 33.

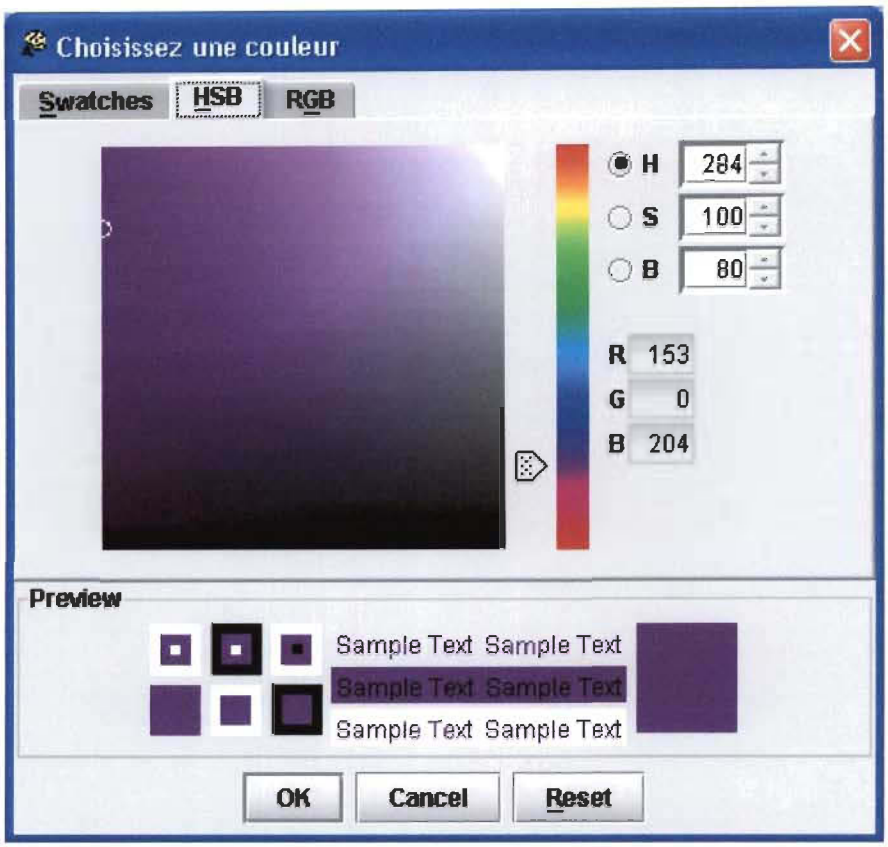

Figure 33 : Sélection d'une couleur pour une catégorie de mots en particulier

Édition des différents messages

Ces options permettent de spécifier à l'environnement si il doit ou non afficher les différents types de messages.

Figure 34 : Fenêtre permettant de spécifier les types de messages à afficher

Confirmer la suppression

- Spécifie à l'environnement d'afficher un message de confirmation lors de la suppression de divers éléments du système.

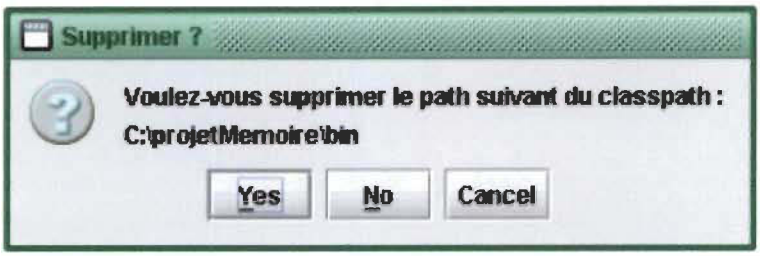

Figure 35 : Message standard de suppression

Confirmer les divers changements

- Spécifie à l'environnement d'afficher un message de confirmation lors d'un changement à l'intérieur de l'environnement

Figure 36 : Message de confirmation standard spécifiant un changement

Afficher les messages d'avertissement

- Spécifie à l'environnement d'afficher un message d'avertissement lorsqu'une action incertaine pouvant entraîner des erreurs ou un comportement non-désiré se produit.

Figure 37 : Message standard d'avertissement

Afficher les messages d'erreur

- Spécifie à l'environnement d'afficher un message lorsqu'une erreur se produit

Figure 38 : Message standard d'erreur

Afficher les changements

- Spécifie à l'environnement d'afficher un message lorsqu'un changement qui entraîne une modification à l'environnement se produit

Figure 39 : Message standard affichant les différents changements

Éditeur HTML

Cette option permet de spécifier l'éditeur HTML à utiliser pour la visualisation de la documentation. La grande utilité de cette option est la possibilité d'utiliser l'éditeur de notre choix. Le principal avantage de l'éditeur de fichiers HTML par défaut de Java est son indépendance par rapport à la plate-forme. Il peut donc être utilisé sur n'importe quel système d'exploitation. Cependant, il possède un inconvénient majeur : son manque de performance. L'éditeur de Java n'est pas très rapide surtout lorsque la page à éditer est importante en terme de grosseur (nombre d'octets) ou si elle possède des cadres « *Frames* ». De plus, cet éditeur ne supporte pas toutes les fonctionnalités de scripts des pages HTML courantes. C'est pourquoi l'environnement offre à l'utilisateur la possibilité de choisir l'éditeur ou le navigateur de son choix (troisième option) en spécifiant le chemin du programme à utiliser. Il est aussi possible de spécifier l'utilisation de l'éditeur par défaut du système d'exploitation. Par exemple, sur Windows, le chemin serait celui d'*Internet Explorer* comme présenté à la figure 40 :

Figure 40 : Fenêtre permettant de choisir l'éditeur HTML à utiliser

5.2.1.7 Menu Aide

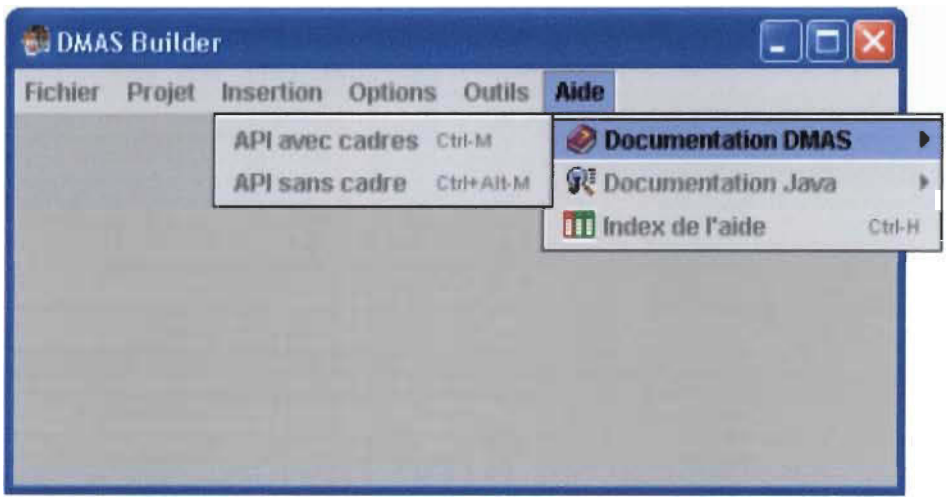

Figure 41 : Options du menu Aide

Documentation DMAS

- *API avec cadres*
  - Permet d'obtenir, en « format plusieurs fenêtres », l'aide de l'API de la librairie de classes supportant la programmation OA.
- *API sans cadre*
  - Permet d'obtenir, en « format une seule fenêtre », l'aide de l'API de la librairie de classes supportant la programmation OA.

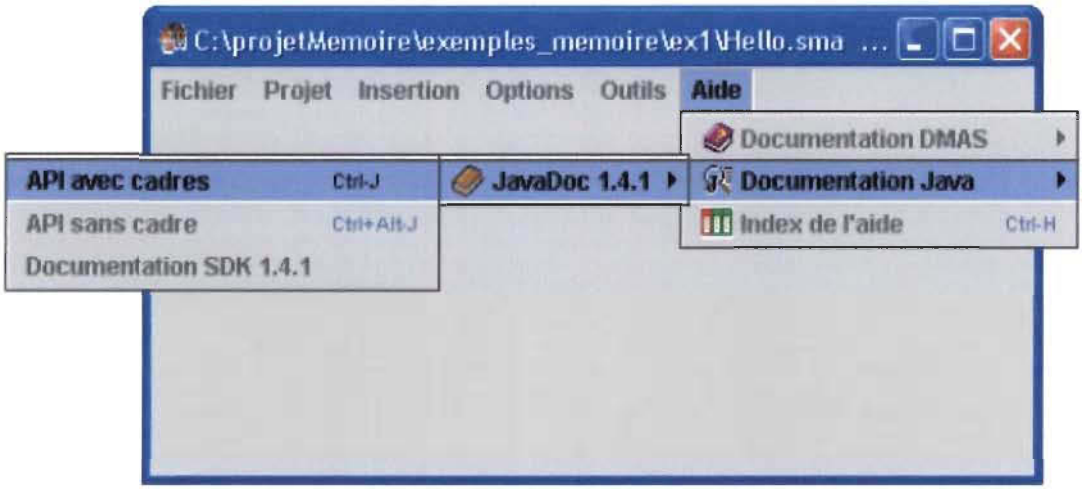

Figure 42 : Options du sous-menu JavaDoc 1.4.1

Documentation Java

- *JavaDoc 1.4.1*
  - *API avec cadres*
    - Permet d'obtenir, en « format plusieurs fenêtres », l'aide de l'API de toutes les classes du SDK 1.4.1 de Java.
  - *API sans cadre*
    - Permet d'obtenir, en « format une seule fenêtre », l'aide de l'API de toutes les classes du SDK 1.4.1 de Java.
  - *Documentation SDK 1.4.1*
    - Permet d'obtenir de la documentation supplémentaire sur le SDK 1.4.1 de Java.

Remarque :

Le sous-menu « Documentation Java » ne contient qu'un sous-menu « JavaDoc 1.4.1 ». La conception est ainsi car de la documentation Java supplémentaire pourra facilement être ajoutée dans les versions ultérieures. Comme par exemple, dans la prochaine version, il sera possible d'accéder au tutorial de Java via un sous-menu de « Documentation Java » qui se trouvera dans le même sous-menu que « JavaDoc 1.4.1 ». Plusieurs autres options de la barre de menus sont conçus de cette façon permettant une grande extensibilité de l'interface.

Index de l'aide

- Mène à la fenêtre où il est possible de chercher dans la documentation en spécifiant le mot recherché.

Cet éditeur d'aide permet une recherche par index soit à l'intérieur de la librairie de classes OA de l'environnement, soit dans la *JavaDoc* de Java ou dans les deux à la fois. De cette façon, une documentation complète est disponible en tout temps pour l'utilisateur. Celui-ci détermine la documentation à utiliser pour une recherche plus ciblée, concise et efficace. Ainsi, il peut séparer les classes de l'API de Java de celles de la librairie de l'environnement. De plus, un choix à l'intérieur de l'environnement permet d'obtenir à l'écran la documentation globale de l'API Java ou celle de l'outil.

Ce type d'aide accélère grandement la recherche de classes, de méthodes et d'attributs dans les API. Il suffit d'entrer les premières lettres du mot recherché et la liste se positionne à l'endroit qui correspond au premier mot commençant par ces lettres dans l'index. L'aide est divisée en trois index pour simplifier et accélérer la recherche. La recherche des même lettres « ***ma*** », comme si on voulait la documentation sur les « *Mailbox* » (voir la section 5.4.2) est utilisée dans les trois fenêtres pour montrer la différence entre les résultats obtenus.

Index général

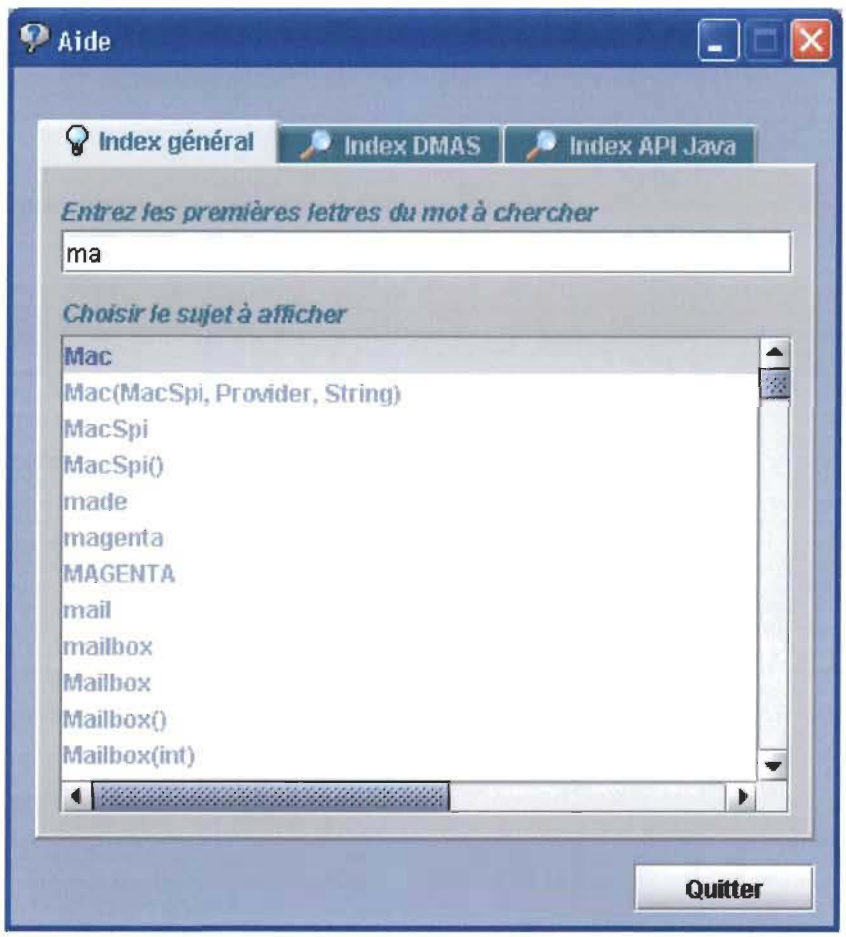

Figure 43 : Aide générale permettant une recherche par lettre, début de mot ou mot

L'index général (Figure 43) permet une recherche à la fois dans l'index de l'API de Java et dans celle de la librairie de classes OA. Cela permet à l'utilisateur de faire une recherche globale.

Index DMAS

Cet index permet d'effectuer une recherche dans la documentation de la librairie de classes OA. De cette façon, il est plus simple de trouver des informations sur ces classes. La recherche à l'intérieur de l'index DMAS permet de filtrer les informations trouvées. De cette façon, on ne retrouve dans la liste (index) que les informations relatives aux classes de la librairie OA (Figure 44).

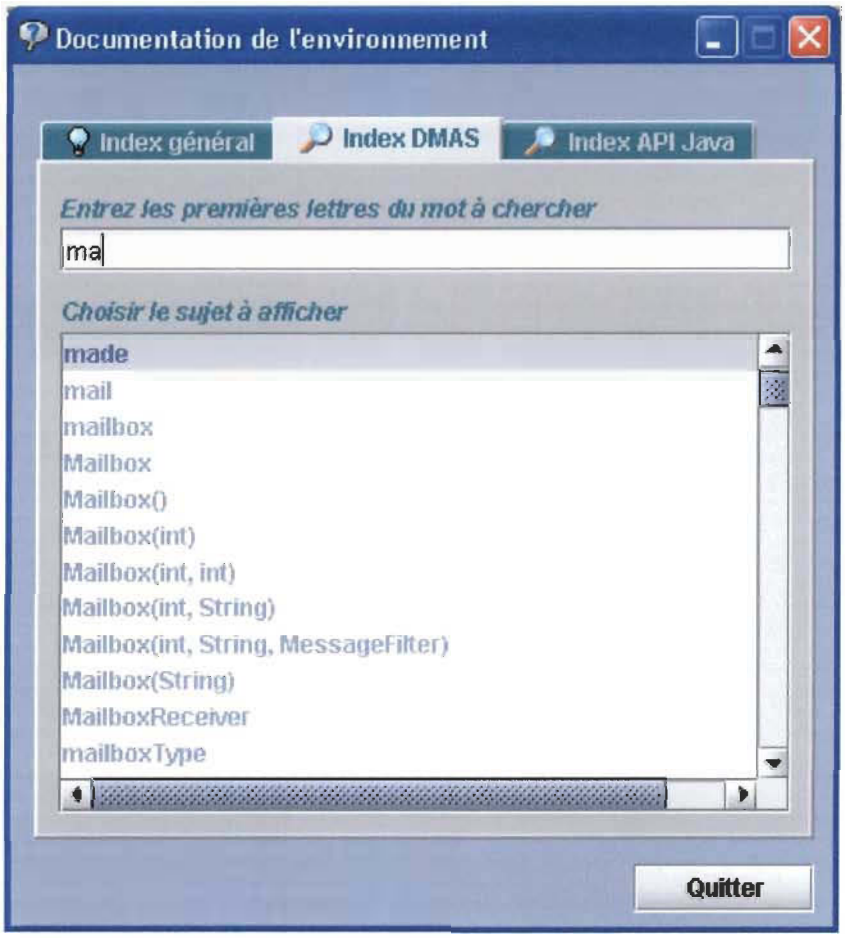

Figure 44 : Aide de la librairie OA pour une recherche par lettre, début de mot ou mot

Index API Java

Cet index permet de chercher uniquement dans la documentation de l'API Java 1.4.1. En effet, il est intéressant d'avoir la possibilité de faire une recherche exclusivement dans la documentation de Java. Par exemple, on peut vouloir trouver le nom d'une méthode

d'insertion d'objets dans une liste ou une « *Map* ». Dans ce cas, il n'est pas utile d'obtenir les résultats de la recherche de la librairie OA (Figure 45).

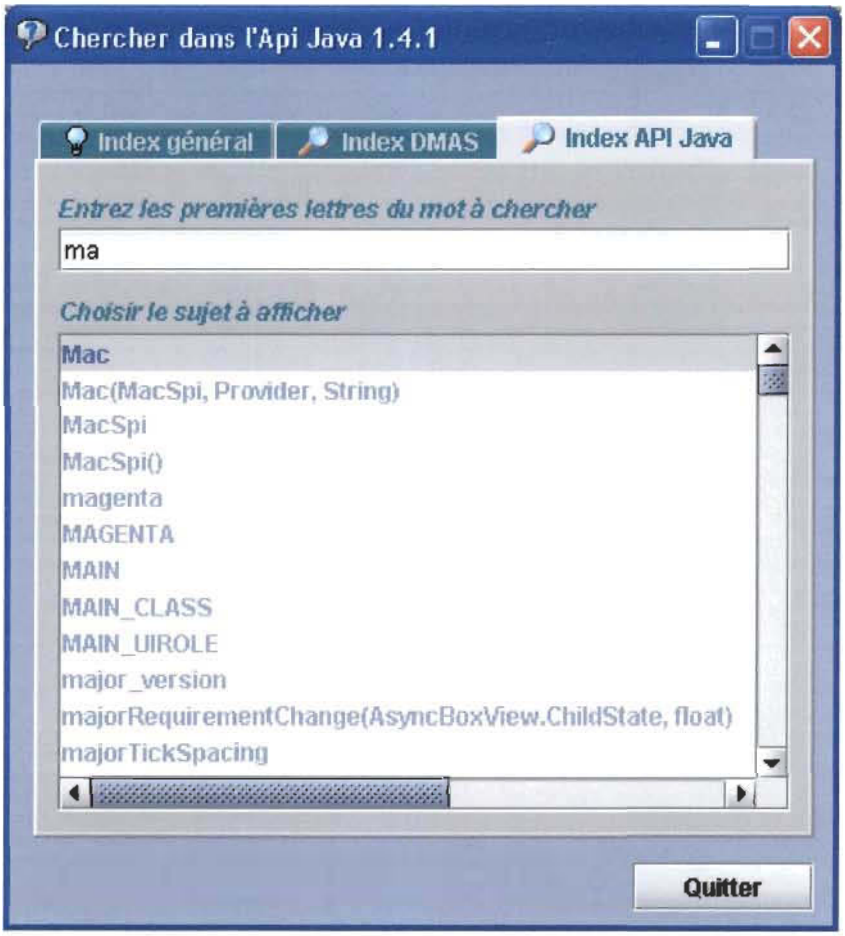

Figure 45 : Aide de Java 1.4.1 pour une recherche par lettre, début de mot ou mot

Remarque :

On remarque que la recherche dans l'index général ne donne que quelques résultats sur les Mailbox parce que plusieurs noms d'attributs, de méthodes et de classes de l'API Java commencent par « ma ». Cette recherche est « polluée » par l'API Java. Par contre, avec la recherche dans l'index DMAS, on retrouve presque exclusivement des mots en rapport avec les Mailbox étant donné que la recherche filtre et retire tout ce qui provient de la documentation de l'API Java. Dans la troisième fenêtre, celle de l'index de l'API Java, il est évident que l'on ne retrouve plus aucun mot en rapport avec les Mailbox car la recherche filtre et retire tous les sujets en rapport avec la librairie de classes OA de l'environnement.

Lorsque le choix du mot à éditer est fait, il est possible que plusieurs liens différents existent pour celui-ci. Dans ce cas, une nouvelle fenêtre apparaît à l'écran et demande le choix à éditer. Prenons par exemple la recherche décrite à la figure 46 :

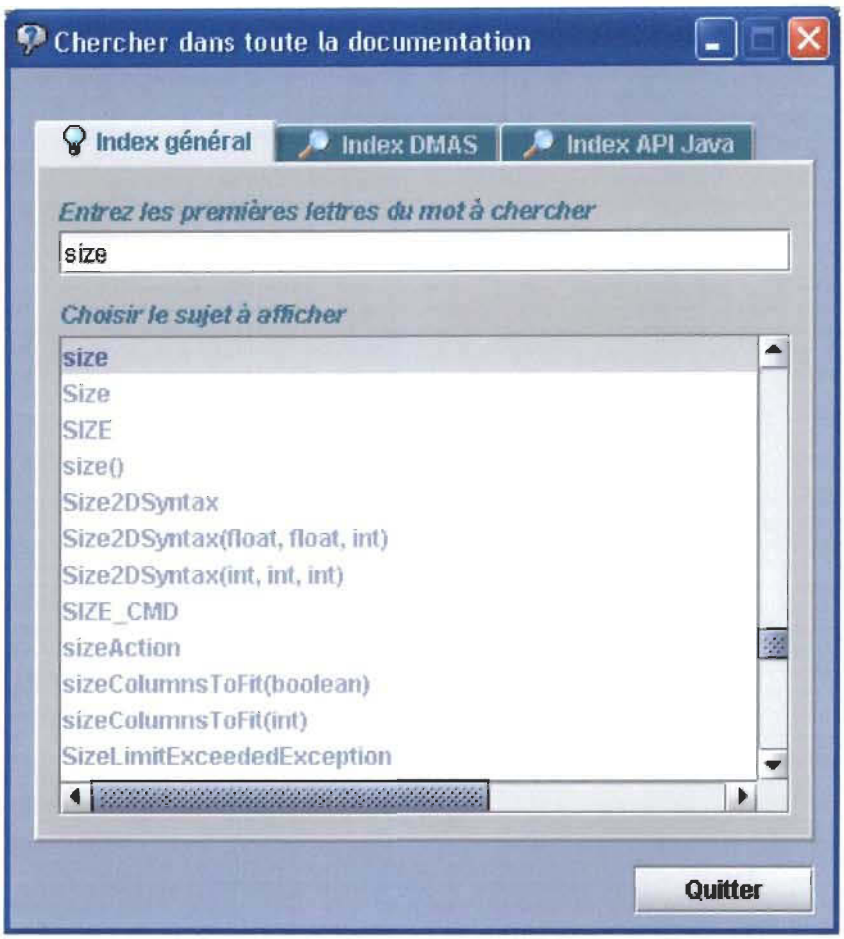

Figure 46 : Exemple de choix d'aide menant à un choix multiple

Lorsque l'on appuie sur *Enter* ou que l'on clique sur le mot « *size* », une autre fenêtre apparaît (Figure 47) étant donné que plusieurs liens différents existent pour ce mot :

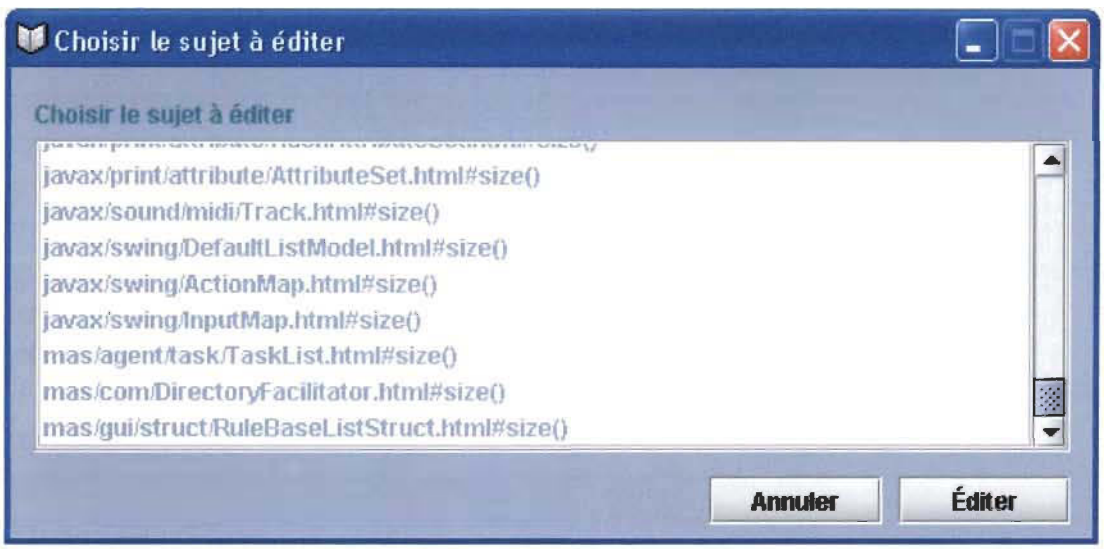

Figure 47 : Fenêtre permettant d'obtenir la documentation d'un choix multiple

De cette fenêtre, si un choix est fait, alors un éditeur affichera la page HTML de la documentation correspondante.

Remarque :

> *Le mécanisme d'aide fourni par l'environnement est beaucoup plus complet que celui fourni par les environnements de développement commerciaux comme VisualCafe. L'environnement fourni une documentation complète de l'API Java 1.4.1 et de* ***tous*** *les liens HTML existant dans cette documentation, ce qui n'est pas le cas des environnements standards. Ces index ont été élaborés à partir d'un programme conçu spécifiquement pour cette tâche (pour DMAS Builder) et celui-ci fait partie intégrante de l'ED. Le programme est un petit utilitaire indépendant de l'environnement qui pourra être utilisé pour la mise à jour de la documentation de l'environnement (librairie OA) ou pour changer la version de la documentation de L'API Java. Par exemple, il pourrait servir pour une mise à jour de la documentation Java (la nouvelle version 1.4.1_02). L'utilitaire est un programme puissant qui extrait tous les liens HTML des fichiers « index » de la JavaDoc de l'API Java et les retranscrit avec leur alias dans des fichiers correspondants (chaque lettre correspond à un fichier de « liens, alias » (voir le guide de maintenance).*

Nous venons de passer en revue les différentes options de l'environnement accessibles sans nécessiter la création d'un projet. On peut considérer ces options comme étant indépendantes des applications ou des systèmes développés. Par conséquent, les options sont les mêmes pour tous les programmes qui sont créés (tant qu'aucune d'entre elles n'est changées). Cependant, ici il faut mettre un bémol car l'environnement effectue parfois quelques changements « intelligents » à des variables (comme le *classpath*) pour permettre une compilation spécifique de *package* et ainsi permettre l'exportation du projet.

5.2.2 Options de projet (accessibles seulement à l'intérieur d'un projet)

Les options regardées dans cette section sont accessibles lorsqu'un projet est ouvert. La première étape est donc d'ouvrir ou de créer un projet.

5.2.2.1 Nouveau projet

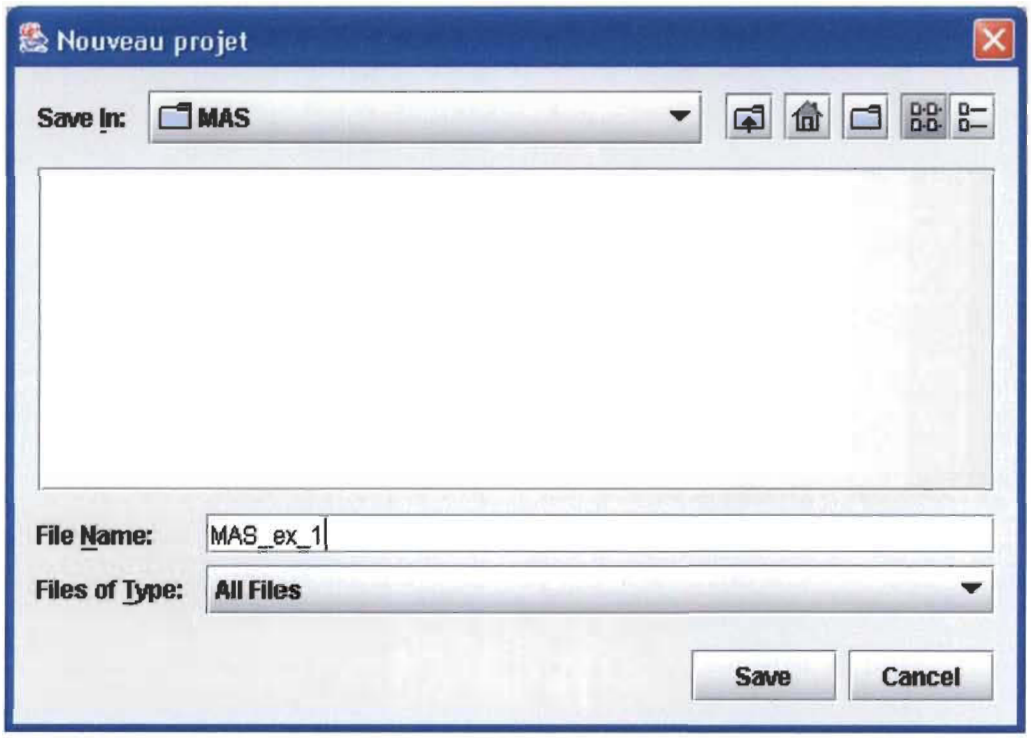

Figure 48 : Création d'un nouveau projet (Menu : Projet → Nouveau projet)

Le nouveau projet sera situé dans le répertoire ***C:\MAS*** et s'appellera ***MAS_ex_1.sma***. Lors de la création d'un nouveau projet, un répertoire nommé « ***app*** » sera créé dans le même répertoire que le projet (dans cet exemple, le projet est ***C:\MAS\MAS_ex_1.sma*** et le répertoire créé est ***C:\MAS\app***). Ce répertoire est très important car il contiendra tous les fichiers sources (et leurs répertoires) générés automatiquement par l'environnement.

Une fois le projet créé, la majorité des options de l'environnement sont accessibles. Il est donc possible d'ajouter des fichiers au projet par le menu ***Insertion → Ajouter des fichiers au projet***. Lorsque l'ajout des fichiers est confirmé, les fichiers ajoutés se retrouvent dans l'éditeur de fichiers.

5.2.2.2 Éditeur de fichiers

L'éditeur de fichiers est un utilitaire où se fait la visualisation des fichiers (« .java » ou autres), l'implémentation en Java et la compilation du projet. L'éditeur permet aussi d'utiliser l'aide de l'API Java 1.4.1 et celle de l'environnement (en surlignant un mot et en appuyant sur la touche « F1 ».

La liste des fichiers (à gauche) contient des informations importantes :

Le nom du fichier

- Le nom du fichier suivi de l'extension (car il est possible d'ajouter des fichiers au projet qui ne sont pas nécessairement des fichiers source « .java »

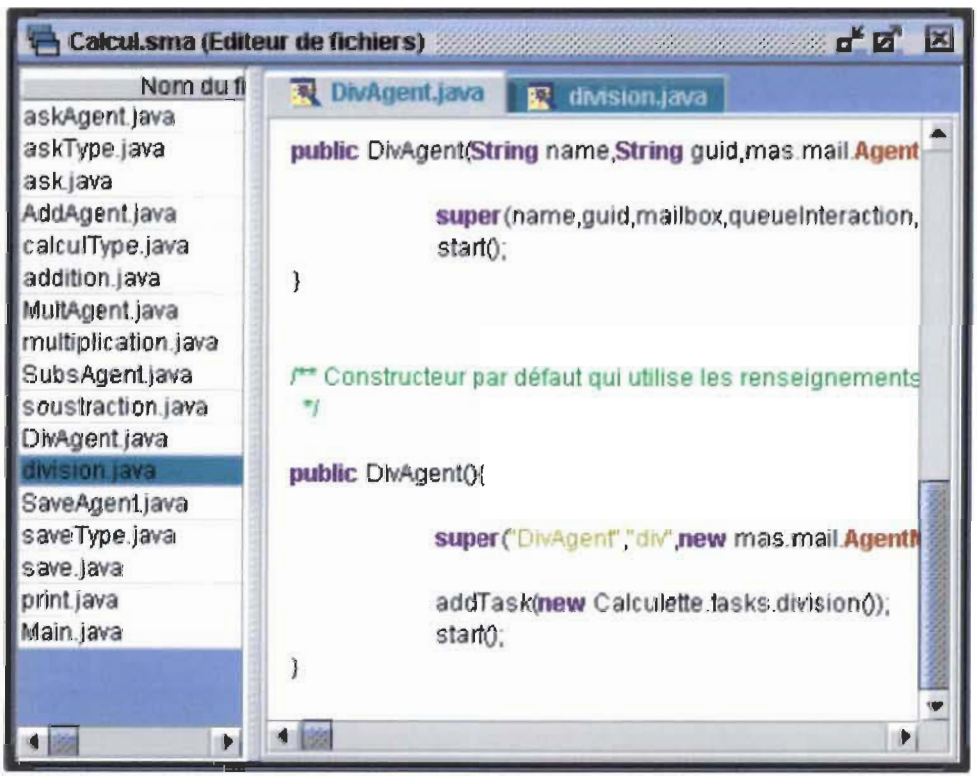

Figure 49 : Éditeur de fichiers

Le type de fichier

- Fichier source
  - « .java » ajouté au projet par le développeur.
- Agent généré
  - « .java » représentant la spécification (via l'éditeur de systèmes) d'un agent en particulier.

- Type d'agent généré
  - « .java » représentant la spécification d'un type d'agent en particulier (via l'éditeur de systèmes).

- Base de connaissances générée
  - « .java » représentant la spécification d'une base de connaissances en particulier (via l'éditeur de systèmes).

- Ressource externe
  - N'importe quel autre fichier ajouté par l'utilisateur dans le projet.

Le chemin du fichier

- Le chemin absolu où se situe le fichier.

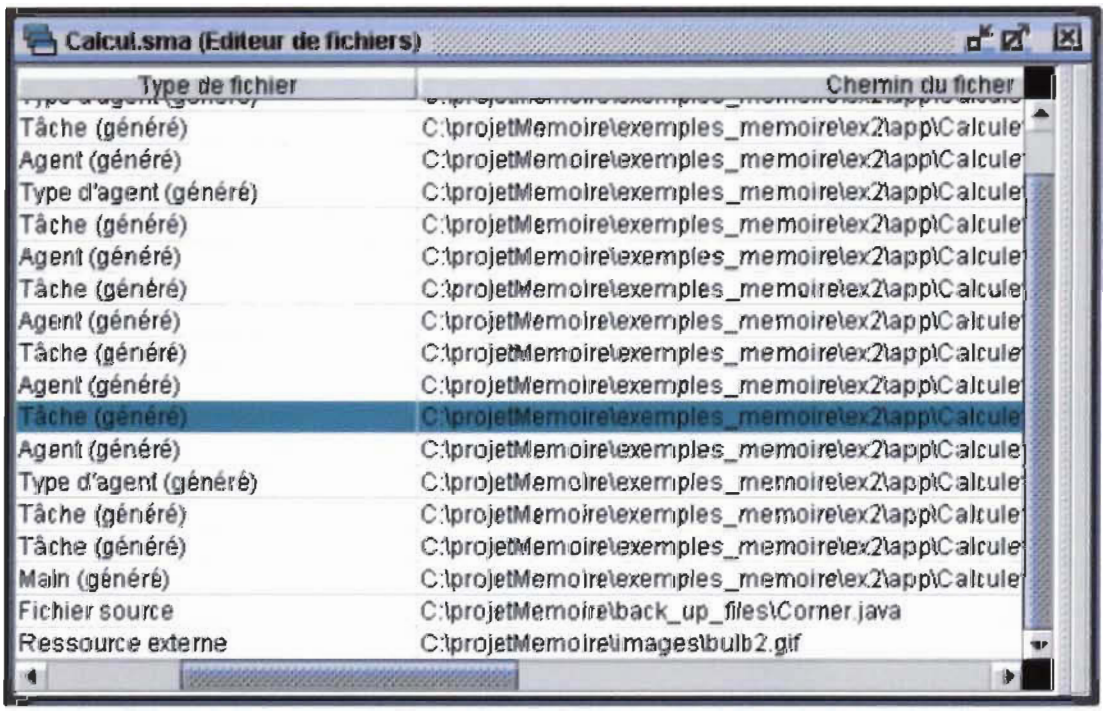

Figure 50 : Table des fichiers (éditeur de fichiers)

Dernière modification

- La date de la dernière modification du fichier.

Fichier sauvegardé

- Oui si le fichier est présentement sauvegardé ou non s'il ne l'est pas.

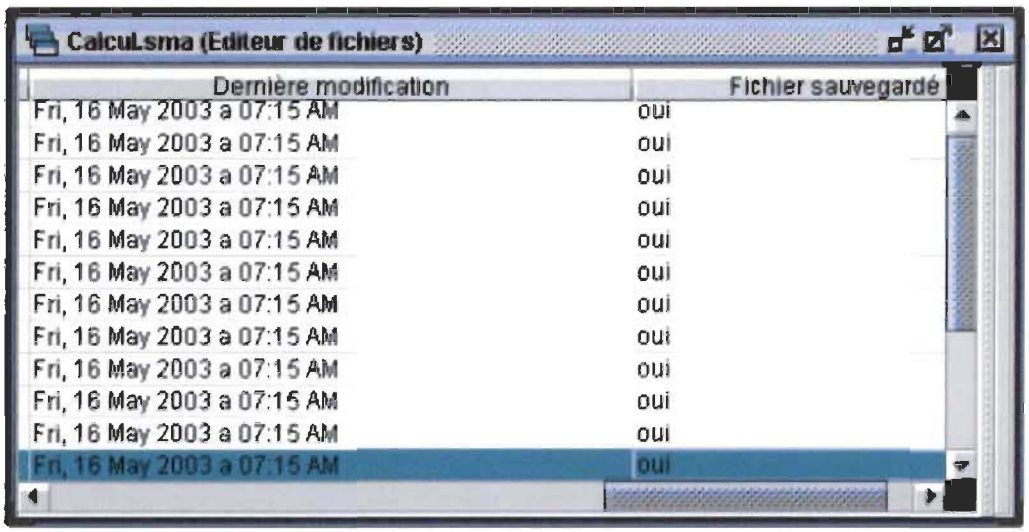

Figure 51 : Table des fichiers (suite)

5.2.2.3 Compilation et exécution

Projet → <une_des_cinq_commandes>

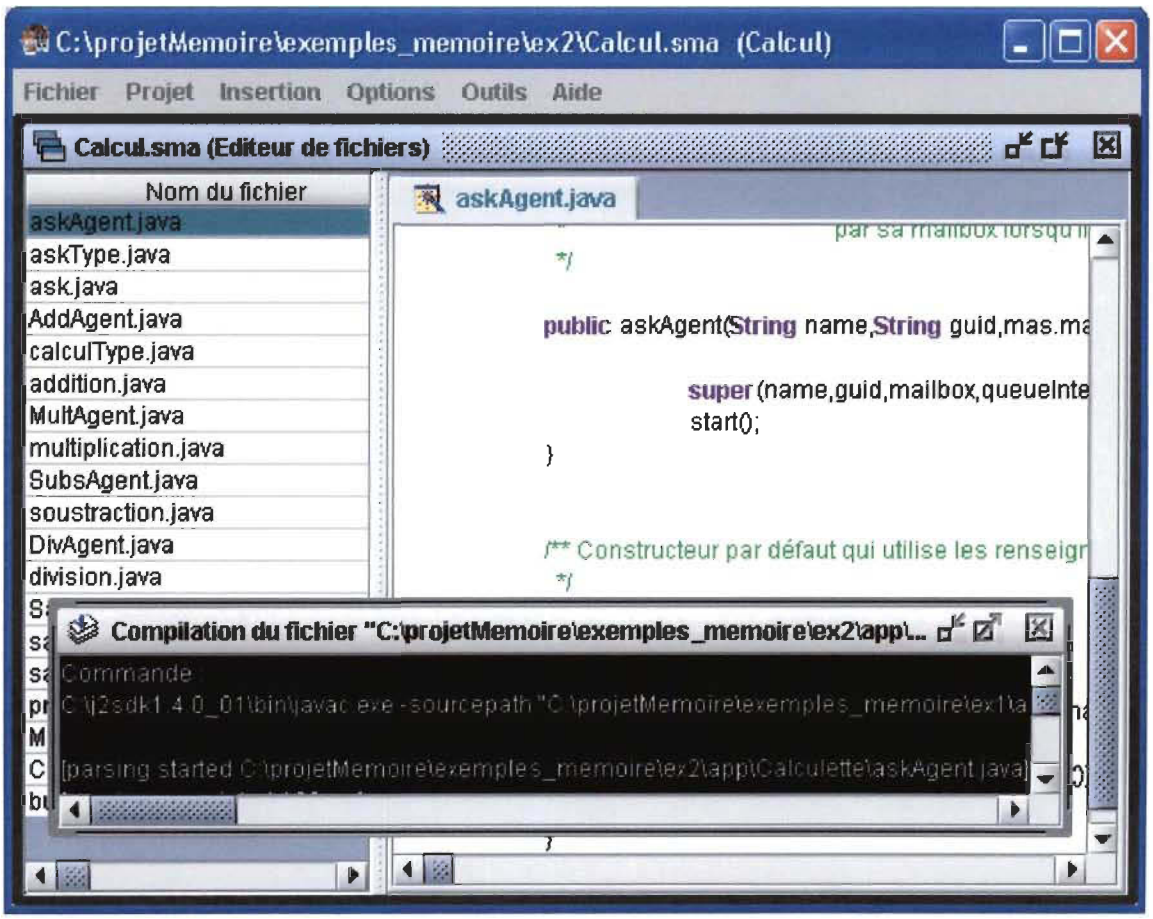

Figure 52 : Fenêtre affichant la commande et les actions effectuées

Lorsqu'un projet contient des fichiers « .java », il est possible de compiler ces fichiers ou de les exécuter avec une des options suivantes du menu Projet :

- *Projet → Exécuter,*
- *Projet → Compiler le fichier,*
- *Projet → Compiler le projet,*
- *Projet → Compiler le fichier avec RMI,*
- *Projet → Compiler le projet avec RMI.*

Dans les cinq cas ci-dessus, une fenêtre apparaîtra à l'écran pour spécifier la ligne de commande qui est exécutée et les détails des actions faites.

Les quatre autres choix dans le menu projet mèneront à un comportement semblable à la figure 52 (apparition de la fenêtre de messages avec la ligne de commande et les actions associées).

5.2.2.4 Options du projet

Cette interface permet de spécifier quelques paramètres propres au projet. De cette façon, lorsque le projet est sauvegardé, il garde ses paramètres. Lors de la réouverture du projet, les paramètres sont réinitialisés aux valeurs propres à ce projet.

Le nom du projet

- Le nom du projet est initialisé automatiquement lors de la création du projet (nom du fichier « .sma »). Cependant, il est possible de changer ce nom.

Le type de projet

- Deux choix sont possibles : l'utilisation de *java* ou de *javaw* pour l'exécution du projet

Description du projet

- Cette boîte de texte n'a pas de limite d'espace : elle permet au développeur du projet de mettre du texte en abondance. Cette option est intéressante car on peut soit mettre une description du projet, soit mettre des étapes qu'il reste à faire, soit mettre des « *bugs* » ou tout ce qui peut être utile au développement du projet. Cette boîte de texte peut servir d'aide mémoire pour le projet.

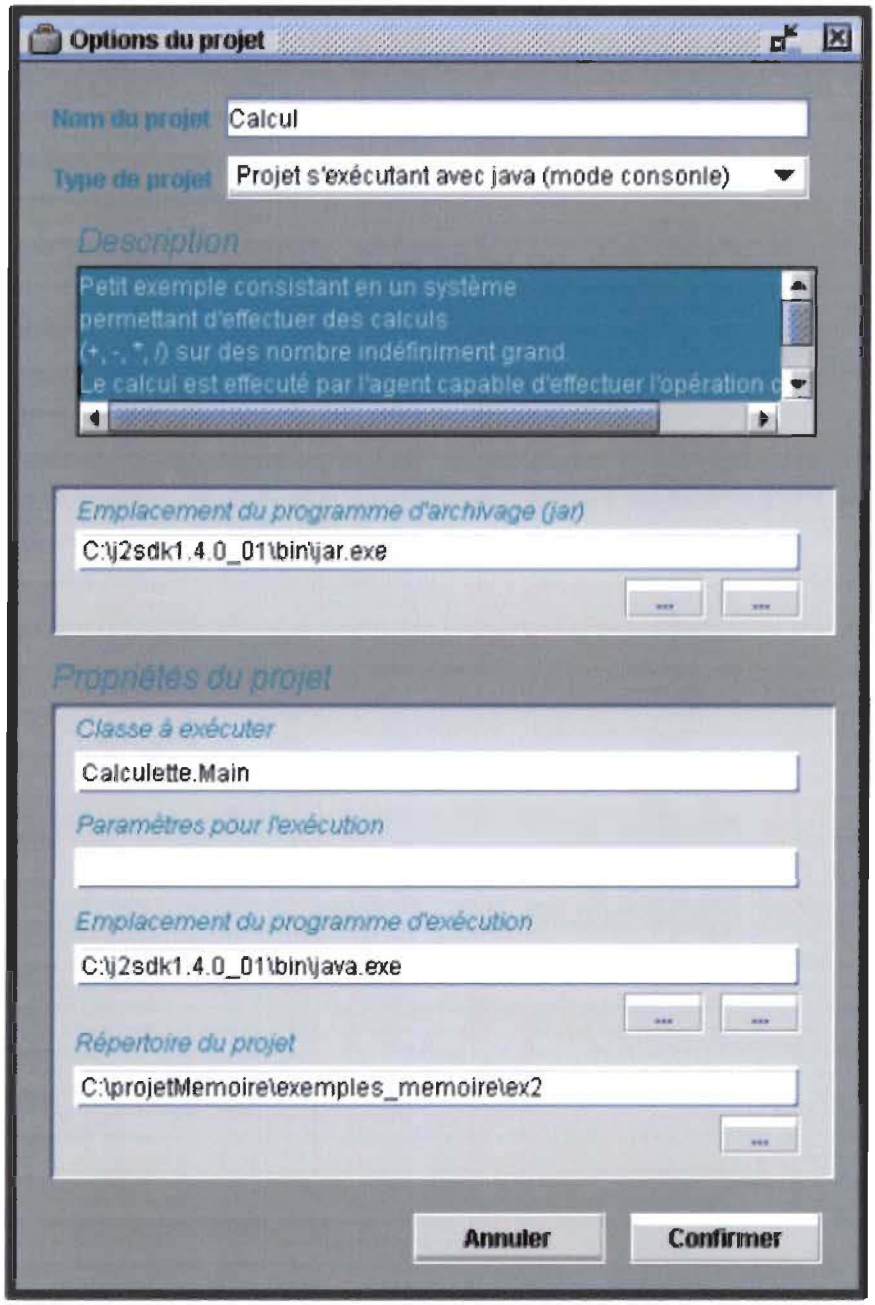

Figure 53 : Options d'un projet

Plusieurs options permettent une meilleure gestion d'un projet en particulier (Figure 53) :

Emplacement du programme d'archivage (jar)

- Cette option permet d'initialiser le programme d'archivage qui servira à l'exportation du projet et à la création des fichiers exécutables pour chaque JVM.

Note

Cette option ne devrait pas être ici étant donné que c'est une option qui est en théorie la même pour tous les projets. Des considérations d'implémentation ont mené à ce choix pour l'instant (pour cette version).

Classe à exécuter

- Nom de la classe à exécuter (classe possédant le « *main* » pour l'exécution).

Paramètres pour l'exécution

- Paramètres à passer lors de l'exécution (paramètres qui sont normalement passés lorsque l'on lance un programme en ligne de commande suivi d'une liste de paramètres)

Emplacement du programme d'exécution

- Chemin du programme nécessaire pour l'exécution du projet (*java* ou *javaw*)

Note

Cette option ne devrait pas être dans cet éditeur étant donné que cette option est la même pour tous les projets (en théorie). Des considérations d'implémentation ont mené à ce choix pour l'instant (pour cette version).

Répertoire du projet

- Le répertoire où se trouve le fichier « .sma » et le répertoire « *app* » . Ce répertoire est initialisé automatiquement lors de la création ou l'ouverture d'un projet. ***Il est fortement déconseillé de changer ce répertoire.***

5.3 L'éditeur de systèmes multi-agents

L'éditeur principal de l'environnement est celui où se fait toutes les spécifications par rapport au système (Figure 54). Cet éditeur permet de spécifier chaque sous-système (hôte) et leurs différents composants :

- Les adresses des JVM (si nécessaire).
- Les différents types d'agents qui pourront être utilisés pour créer les agents. Un type d'agent est déterminé en fonction de son comportement (plusieurs comportements sont disponibles), de son architecture (BDI ou autre) et de quelques autres spécifications.
- Les tâches qui seront effectuées par l'un ou l'autre des agents.
- Les bases de connaissances que certains agents posséderont.
- Le ou les agents pour chaque hôte (sous-système) et leurs paramètres :
  - Le mode de communication (RMI, *socket* TCP, *socket* UDP, *multicast* ou direct),
  - La base de connaissances utilisée (si nécessaire),
  - Les différentes tâches que chacun pourra effectuer,
  - Autres...
- Pour chaque hôte, il est possible de spécifier un DF. L'outil fourni différents types de DF, dépendamment des situations et du comportement désirés.
- Il est également possible de spécifier aux sous-systèmes d'enregistrer automatiquement leurs agents auprès du ANS (*Agent Name Server*). Dans ce cas, un agent *RegisterAgent* sera créé pour l'hôte.
- Et bien d'autres composants...

Quatre utilitaires très importants sont fournis par cet éditeur :

- La validation du système.
- La génération automatique du code source (en Java) des différents composants du système.
- La création des fichiers sources nécessaires à l'indépendance du système par rapport à l'environnement.
- La création d'archives exécutables pour chaque sous-système.

Cette interface (éditeur de systèmes) est l'endroit où sont définis tous les composants du système et de ses sous-systèmes. L'idée de tout englober les spécifications dans une seule interface est de faciliter la tâche des utilisateurs. Une des faiblesses chronique des environnements de développement de SMA offrant un développement graphique est la complexité de naviguer à l'intérieur de l'outil. L'intention ici est de promettre à l'utilisateur que toutes les spécifications sont faites à l'intérieur de cette interface (et de ses sous interfaces pour les bases de connaissances). De cette façon, l'utilisateur n'a pas à composer avec la création de composants conditionnels à la création d'autres objets du système via d'autres interfaces. L'utilisateur peut donc se concentrer sur cette interface sans avoir à se demander s'il n'a pas omis des définitions ailleurs dans l'environnement.

De plus, comme toutes les spécifications sont faites à l'intérieur de la même interface, les différentes informations sont facilement et rapidement accessibles.

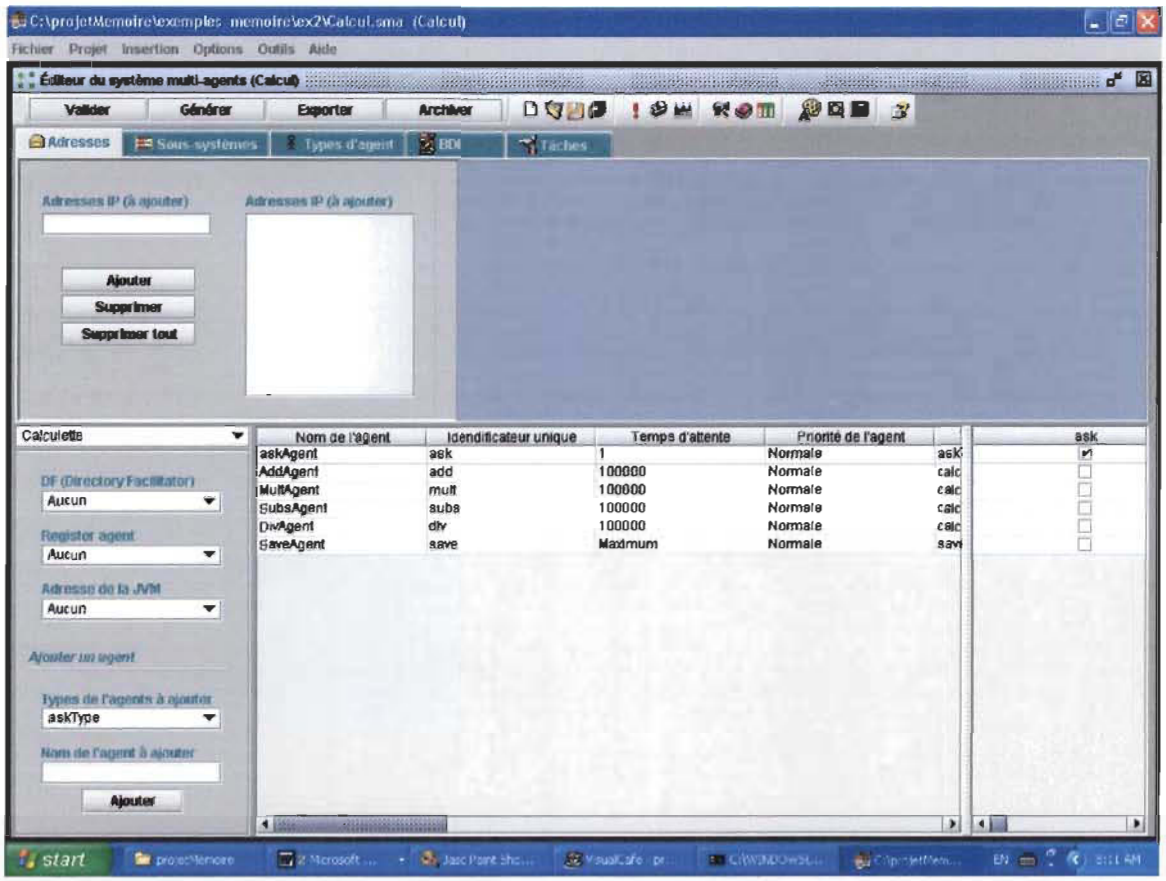

Figure 54 : Éditeur de systèmes

Une des grandes forces de l'éditeur de systèmes est la possibilité de construire le système dans n'importe quel ordre (à quelques exceptions près). On peut donc par exemple commencer par spécifier les tâches, suivi des bases de connaissances, suivi des JVM, etc. On pourrait aussi commencer par créer les types d'agents, suivi des adresses IP suivi des tâches, etc. Cette façon de procéder permet une grande flexibilité et une grande extensibilité. On peut construire un système et par la suite ajouter un sous-système ou des nouvelles tâches et les ajouter à un agent en particulier sans aucune difficulté.

L'éditeur de systèmes est très puissant. Il comporte plusieurs options fournissant des services et plusieurs spécificités qu'il est nécessaire d'expliquer en détails. Il faut donc regarder et expliquer les différentes possibilités offertes par cet éditeur.

5.3.1 Adresses IP

Cette fenêtre (Figure 55) permet de spécifier les adresses IP des différentes JVM qui feront partie du système. Chaque adresse pourra être associée avec une JVM (voir plus bas). Cela permet de procéder à l'enregistrement automatique des agents de chaque JVM auprès des autres JVM. Si aucune adresse n'est spécifiée, la tâche d'enregistrer les agents d'une JVM auprès des autres revient au programmeur (sauf dans le cas où les JVM possèderaient chacune un *MulticastRegisterAgent*, voir la section *RegisterAgent*).

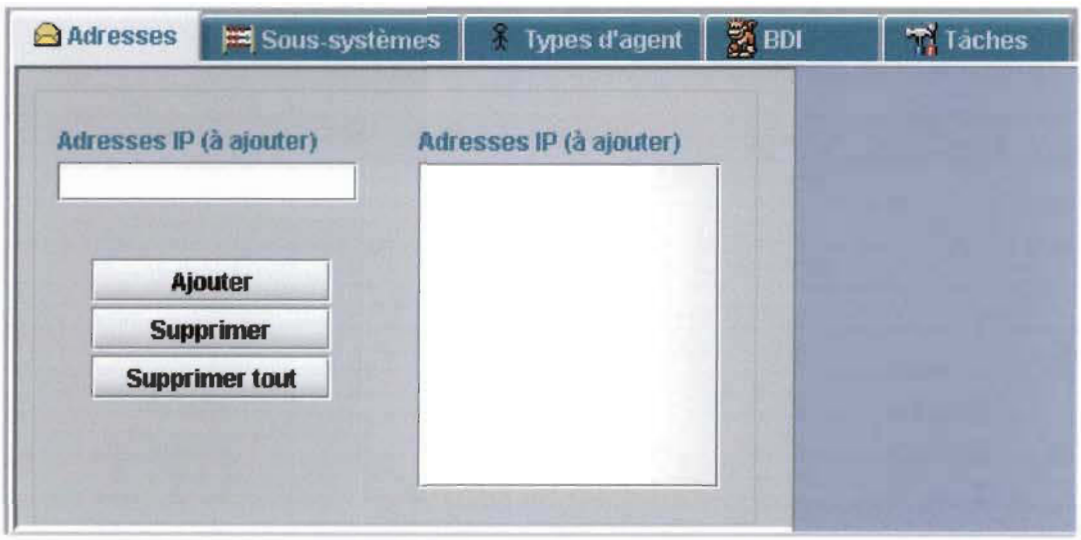

Figure 55 : Onglet permettant de spécifier les adresses IP de chaque JVM

5.3.2 Sous-systèmes

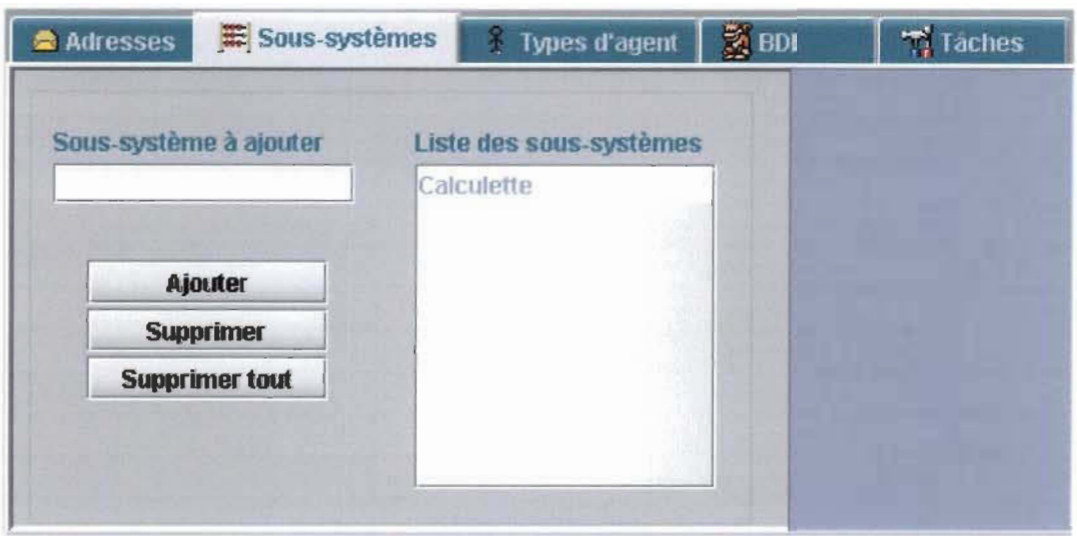

Figure 56 : Onglet pour la création des sous-systèmes (JVM)

Cette interface (Figure 56) permet de spécifier les noms des JVM (sous-systèmes). Évidemment, il est nécessaire de donner un nom à un sous-système avant de spécifier ses différents composants. Aussitôt qu'un nom est ajouté à la liste des sous-systèmes, une JVM est créée pour ce système.

5.3.3 Types d'agent

Le type d'agent est un peu plus complexe que les deux autres (Figure 57). Une couche se situe au-dessus d'un agent : son type. De cette façon, il est possible de créer rapidement plusieurs agents possédant des caractéristiques semblables.

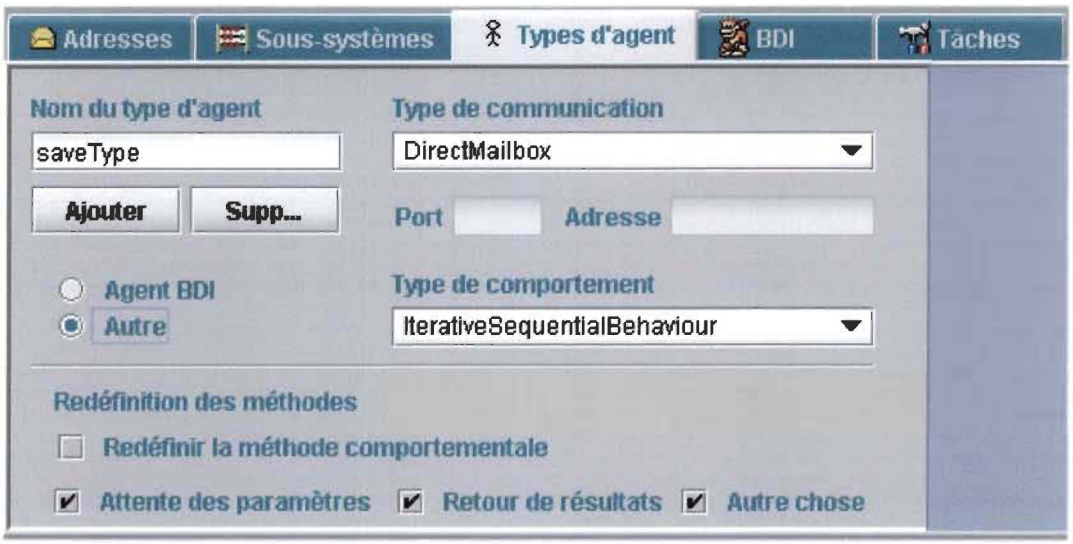

Figure 57 : Onglet permettant de créer un type d'agent en particulier

La première étape avant de créer un agent pour une JVM en particulier est de créer un type d'agent. Celui-ci pourra soit être utilisé que pour cet agent ou soit pour d'autres agents possédant des caractéristiques similaires.

5.3.3.1 Nom et type d'agent

Nom du type d'agent

- Le nom qui sera utilisé pour représenter ce type d'agent.

Agent BDI ou autre

- Si l'agent possède une base de connaissances, alors il est impératif de spécifier BDI. Sinon, il sera un agent « Autre ».

5.3.3.2 Type de communication

La façon de recevoir des messages par un agent. Pour l'instant, les modes suivants sont disponibles :

RMIMailbox

- Le type d'agent écoute les message via RMI, c'est-à-dire qu'il reçoit les messages par des appels distants des autres agents sur une méthode que sa Mailbox rend accessible.

SocketMailbox

- Le type d'agent écoute les messages par un *socket* via le protocole TCP.

MulticastMailbox

- Le type d'agent écoute tous les messages qui sont envoyés sur une adresse *multicast*.

UDPSocketMailbox

- Le type d'agent écoute les messages par un *socket* via le protocole UDP.

DirectMailbox

- Le type d'agent ne possède pas de méthode de réception de messages externe (provenant d'agents des autres JVM). Il peut cependant envoyer des messages à n'importe quel agent (sur la même JVM ou sur une autre). De plus, il peut aussi recevoir des messages des agents qui se trouvent sur la même JVM.

Remarque :

Il est à noter que le type de communication pour un type d'agent est plus un utilitaire qu'une véritable propriété du type d'agent. En effet, le type de communication est une propriété de l'agent plutôt que de son type. Il est possible de spécifier le type de communication au niveau du type d'agent pour donner un mode par défaut à ce type d'agent. Par contre, dans la table des agents (voir plus bas) il est possible de changer ce mode de communication pour un agent en particulier.

5.3.3.3 Adresse et port de communication

Port

- Le port sur lequel la *Mailbox* de l'agent écoute (lorsque nécessaire).

Adresse

- Adresse *multicast* sur laquelle la *Mailbox* écoute. Champ nécessaire seulement pour les *Mailbox* écoutant en mode *multicast*.

5.3.3.4 Types de comportement

Le type de comportement pré-implémenté de l'agent. Pour l'instant, les comportements suivants peuvent être donnés à un agent :

Aucun

- Dans ce cas, l'utilisateur n'aura qu'à redéfinir la méthode « *void execute()* ». D'ailleurs, cette méthode sera redéfinie lors de la génération automatique du type d'agent. Cependant, le corps de la méthode sera vide. Cette méthode ne doit pas être appelée par le programmeur; elle est automatiquement appelée par le système.

OneShotSequentialBehaviour

- Spécifie qu'un agent de ce type exécute une seule fois sa liste de tâches une à la suite de l'autre. Lorsque l'exécution des tâches est terminée, l'agent cesse son exécution et quitte le système.

IterativeSequentialBehaviour

- Spécifie qu'un agent de ce type exécute sa liste de tâches une à la suite de l'autre et ce de façon itérative tant qu'il est vivant. Lorsque l'exécution de la liste de ses tâches est terminée, l'agent recommence l'exécution de sa liste de tâches.

OneShotRandomBehaviour

- Spécifie qu'un agent de ce type exécute une seule fois sa liste de tâches dans un ordre indéterminé. Lorsque l'exécution des tâches est terminée, l'agent cesse son exécution et quitte le système.

IterativeRandomBehaviour

- Spécifie qu'un agent de ce type exécute sa liste de tâches dans un ordre indéterminé et ce de façon itérative tant qu'il est vivant. Lorsque l'exécution de la liste de ses tâches est terminée, l'agent recommence l'exécution de la liste.

RandomBehaviour

- Spécifie qu'un agent de ce type exécute une seule tâche choisie au hasard dans sa liste et termine ensuite son exécution et quitte le système.

OnRequestBehaviour

- Ce comportement est très utile. Si un agent adopte ce comportement, il se met en attente de messages entrant et lorsqu'il reçoit un message, il vérifie si le sujet du message correspond à un nom de tâche qu'il peut effectuer. Si c'est le cas, il effectue la tâche. Sinon, il ne fait rien.

ComplexBehaviour

- Ce type de comportement permet de redéfinir des comportements plus appropriés dans des circonstances particulières.

5.3.3.5 Redéfinition de méthodes

La redéfinition de méthodes nécessite la compréhension de l'ordre des appels de méthodes lors de l'exécution d'une tâche (voir la spécification de la classe *mas.agent.Agent* pour une meilleure description). Voici une idée générale du concept :

Méthode comportementale

- Si cette méthode est redéfinie par le type d'agent, alors le comportement associé décrit ci-haut n'est plus valide. Il faut redéfinir complètement ce comportement. Un squelette de la méthode sera généré où il est nécessaire de mettre le code à exécuter. Cependant, il ne faut pas appeler cette méthode; l'appel de celle-ci est automatiquement fait par le système.

Attente de paramètres

- Si cette méthode est redéfinie dans le type, alors chaque fois qu'une tâche sera sur le point d'être exécutée, cette méthode sera automatiquement appelée juste avant l'exécution de la tâche. Ceci permet d'attendre de plusieurs façons des paramètres nécessaires à l'exécution de la tâche

Retour de résultats

- Si cette méthode est redéfinie dans le type, alors chaque fois que l'exécution d'une tâche se terminera, cette méthode sera automatiquement appelée juste après l'exécution de la tâche. Ceci permet de retourner des résultats à l'agent ou d'envoyer des messages à d'autres agents.

Autres choses

- Si cette méthode est redéfinie dans le type, alors chaque fois que l'exécution d'une tâche se terminera, cette méthode sera automatiquement appelée juste après l'appel à la méthode de retour de résultats. Cette méthode est plus appropriée

lorsque la tâche n'a aucun résultat à retourner mais qu'il est nécessaire d'effectuer des actions en particulier.

5.3.4 Tâches

La description des tâches est simple (Figure 58). Elle consiste à donner un nom à chaque tâche qui pourra être effectuée par un ou des agents et de lui donner une description (optionnelle).

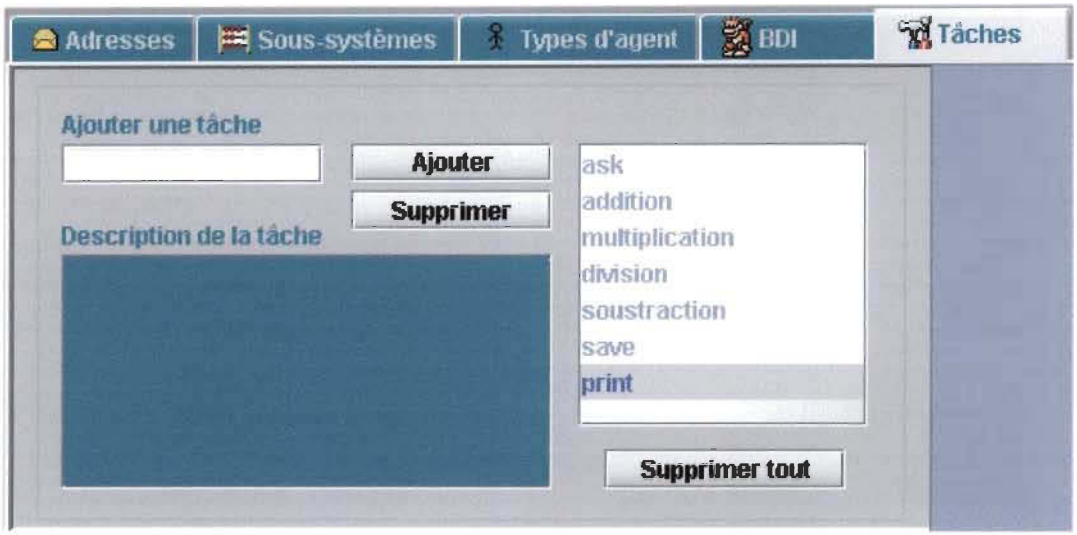

Figure 58 : Onglet permettant d'ajouter des tâches

Remarque :

Il est important de mentionner qu'une tâche en particulier ne sera pas nécessairement exécutable par tous les agents. La liste de tâches est la liste globale des différentes tâches pouvant être effectuées par un ou des agents du système. La table des agents permet de déterminer quels agents peuvent effectuer quelles tâches. Chaque agent peut effectuer plusieurs tâches et chaque tâche peut être effectuée par plusieurs agents. Cependant, un agent n'effectue pas nécessairement toutes les tâches et une tâche n'est pas nécessairement effectuée par tous les agents.

5.3.5 Base de connaissances (agent BDI)

Les bases de connaissances peuvent être implémentées directement à partir de la librairie OA de l'environnement. Cependant, l'utilisation de trois interfaces permettent l'implémentation simple et relativement rapide des bases de connaissances. Les bases de connaissances sont composées des éléments suivants :

Un ensemble de variables

- Les variables sont les éléments sur lesquels il est possible de faire des comparaisons logiques.

Un ensemble de clauses

- Une clause est une condition pré-requise (antécédente) d'une règle ou le conséquent d'une règle (ce qui sera effectué) si tous les antécédents sont vrais. Une clause peut être une comparaison logique d'une variable avec une valeur ou une chaîne de caractères. Une clause peut aussi être un effecteur ou un senseur.

Un ensemble de règles

- Une règle est constituée d'un ensemble de clauses pré-requises (liste des antécédentes) et d'une clause conséquente (clause qui sera effectuée si toutes les clauses antécédentes sont vraies).

Un ensemble de faits

- Clauses étant toujours vraies.

Remarques :

La création d'une base de connaissances doit se faire de façon linéaire. Il est nécessaire de commencer par définir la liste des variables de la base de connaissances. Par la suite, il faut définir les différentes clauses de la base de connaissances (les clauses dépendent des variables). Enfin, il faut créer les règles (qui dépendent des clauses). Le développement d'une base de connaissances est linéaire car avant de créer une règle, il faut que les clauses qui la composent soient déjà créées. La même logique s'applique à une clause. Si l'on veut créer une clause, il faut que la variable qui constitue la partie de gauche existe déjà. Même si le développement est linéaire, il est possible d'ajouter des variables, clauses et règles. Il faut cependant que les composants nécessaires à la création d'un autre composant (clause ou règle) soient préalablement créés.

Pour ajouter une base de connaissances à la liste, il suffit de spécifier un nom pour celle-ci. Par la suite, pour ajouter des variables à une base de connaissances, il est nécessaire de choisir dans la liste le nom de la base de connaissances à modifier. Celle-ci apparaîtra dans le champ « nom » (comme dans la figure 59). Une fois le nom de la base de connaissances dans le champ « nom », il est possible de choisir « variables » pour ajouter des variables à cette base de connaissances, « clauses » pour lui ajouter des clauses et « règles » pour lui ajouter des règles.

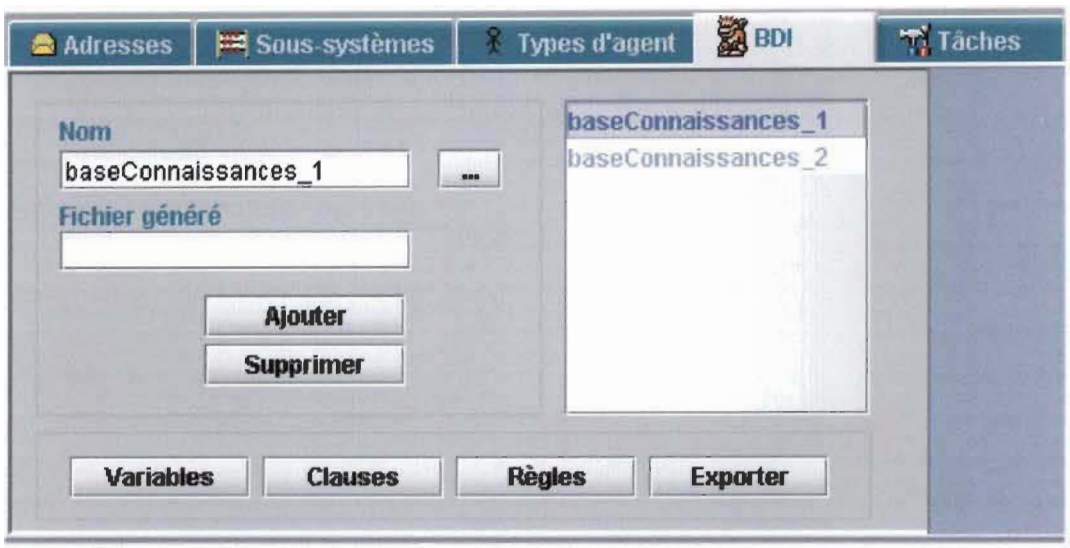

Figure 59 : Onglet permettant de créer des bases de connaissances

Une autre option très importante est disponible : l'exportation d'une base de connaissances. Les spécifications contenues dans l'éditeur de systèmes n'étant applicables qu'à un seul système, il est impossible de récupérer une partie de celles-ci (sauf en récupérant le code généré et en l'ajoutant à un nouveau projet). Cependant, contrairement aux autres spécifications d'un système, les bases de connaissances peuvent être extraite de la spécification d'un système pour être réutilisées dans d'autres projets. Cette option a été ajoutée étant donné que les bases de connaissances sont longues à développer et celles-ci peuvent souvent être réutilisées dans d'autres projets pour d'autres agents. Il serait vraiment désagréable d'être obligé de recréer la même base de connaissances (refaire les mêmes opérations via les interfaces de bases de connaissances) pour des agents de systèmes différents. De plus, une base de connaissances peut être exportée et modifiée dans un autre projet, ce qui permet une bonne flexibilité et extensibilité de celle-ci.

5.3.5.1 Variables

Pour ajouter des variables à une base de connaissances, il suffit de spécifier un nom et, si désiré, une valeur par défaut pour celle-ci. Si aucune valeur n'est spécifiée, la valeur « *null* » lui sera affectée (Figure 60).

Figure 60 : Éditeur permettant d'ajouter des variables dans une base de connaissances

5.3.5.2 Clauses

Plusieurs éléments sont à prendre en considération lors de l'ajout de clauses dans une base de connaissances (voir la figure 61). La première chose à laquelle il faut penser est qu'il faut définir toutes les clauses qui seront nécessaires dans les différentes règles de la base de connaissances. Il ne faut pas oublier que les clauses conséquentes des règles doivent aussi être créées.

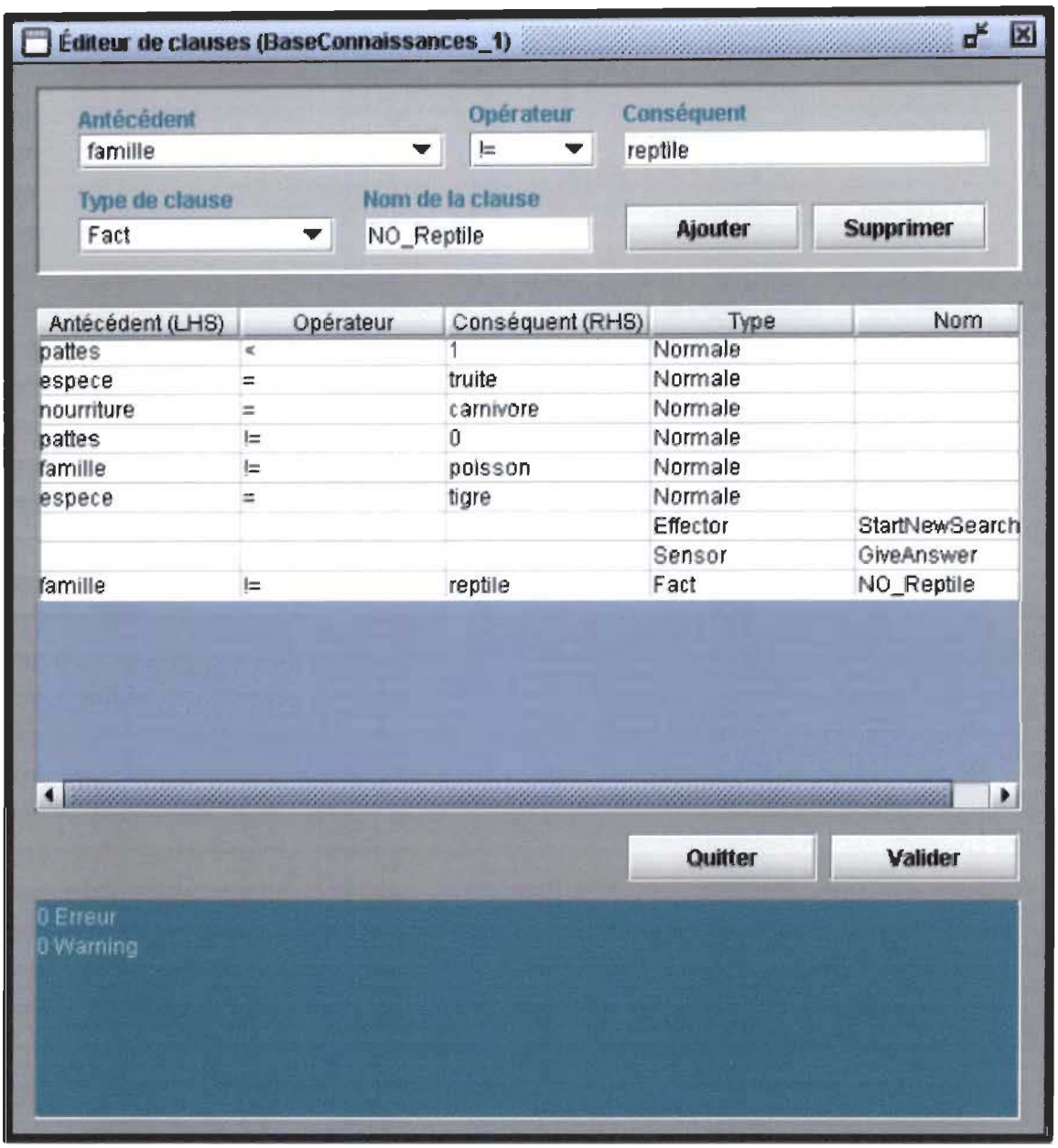

Figure 61 : Éditeur permettant d'ajouter des clauses à une base de connaissances

Remarque :

Il est important de noter que le signe d'égalité dans la colonne opérateur sert d'opérateur d'égalité lorsque la clause est utilisée dans la liste des antécédents d'une clause mais qu'il sert d'opérateur d'affectation lorsque la clause est utilisée en tant que conséquent d'une règle. Une même clause peut être utilisée dans les deux circonstances sans problème car c'est le moteur d'inférence qui déterminera si l'opérateur « = » doit être utilisé comme opérateur de comparaison d'égalité ou comme opérateur d'affectation. Cette technique peut sembler un peu bizarre mais elle permet de sauver l'écriture de plusieurs clauses. Prenons l'exemple suivant : l'utilisateur veut écrire deux clauses : ***« si***

famille est égal à félin *» et la clause «* ***affecter à la variable famille la valeur félin*** *» car la première est utilisée dans les antécédents d'une règle et la deuxième est utilisée comme conséquent d'une règle. L'utilisateur aurait normalement deux clauses à écrire si l'opérateur « = » n'était pas surchargé. Par contre, ici l'utilisateur n'écrit qu'une seule clause «* ***famille*** *=* ***félin*** *» et l'engin d'inférence s'occupe de choisir la signification de l'opérateur « = ».*

Les différents éléments à considérer lors de la création d'une clause (Figure 62) sont les suivants :

Antécédent

- L'antécédent est une des variables déterminées dans l'éditeur de variables. Les variables se retrouvent dans le menu déroulant; il suffit de choisir celle qui est nécessaire.

Opérateur

- L'opérateur logique de comparaison entre l'antécédent et le conséquent (<, <=, >, >=, !=, =). Comme mentionné ci-haut, l'opérateur « = » a deux fonctions : l'égalité et l'affectation.

Conséquent

- Une chaîne de caractères ou une valeur numérique avec laquelle la variable est comparée.

Nom

- Nom de la clause.

Type

Type de clause (*Normal, Fact, Sensor, Effector*).

Normal

- « *Normal* » est le type de clause standard. Une clause de type « *Normal* » est une clause qui utilise une variable de la liste des variables de la base de connaissances et effectue une comparaison logique (un des six opérateurs) avec une valeur numérique ou une chaîne de caractères. Ce type de clause n'utilise pas l'attribut nom.

Fact

- Clause qui est un fait dans la base de connaissances. Clause qui utilise une variable de la liste des variables de la base de connaissances et spécifie qu'une

condition logique (un des six opérateurs) avec une valeur numérique ou une chaîne de caractères est toujours vraie. Ce type de clauses nécessite la spécification du nom.

Sensor

- Clause qui est très utile. Elle permet d'exécuter une partie de code et retourne vrai si la clause est validée (vérifiée) et faux si la clause n'est pas validée. Ce type de clause n'utilise ni l'antécédent, ni l'opérateur, ni le conséquent; cependant, elle utilise l'attribut nom. Il sera possible de spécifier le code que le senseur aura à effectuer une fois la génération du code terminée.

Effector

- Clause qui est très utile. Elle permet d'exécuter une partie de code si tous les antécédents d'une règle sont vérifiés. Une clause effecteur permet de remplacer une clause conséquente dans une règle lorsqu'il faut exécuter une méthode au lieu d'affecter une valeur à une variable dans un conséquent. Ce type de clause n'utilise ni l'antécédent, ni l'opérateur, ni le conséquent; cependant, elle utilise l'attribut nom. Il sera possible de spécifier le code que l'effecteur aura à effectuer une fois la génération du code terminée.

	Antécédent	Opérateur	Conséquent	Nom
Normal	X	X	X	
Fact	X	X	X	X
Sensor				X
Effector				X

Figure 62 : Champs obligatoires en fonction du type de clause

Si l'utilisateur détermine que des variables sont manquantes pour la création de clauses, il n'a qu'à retourner dans l'éditeur de variables et ajouter les variables nécessaires.

En tout temps, il est possible de faire une validation partielle des clauses. Cette option permet de vérifier quelques erreurs communes qui peuvent se glisser lors de la rédaction des clauses. Comme par exemple, la duplication de clause, la duplication de nom, etc. D'autres validations plus importantes sont aussi faites. Étant donné que les clauses utilisent des variables (antécédents), ces dernières doivent exister en tout temps dans la table des variables de la base de connaissances. Si une variable est supprimée dans l'éditeur des variables mais que des clauses utilisent cette variable, alors il y a incohérence dans la base de connaissances. La vérification permet de valider ce type de cas et envoie un message lorsque de telles aberrations surviennent.

Voici des messages typiques d'erreurs pouvant être affichés dans l'espace de texte situé au bas de l'éditeur de clauses. Dans la figure 63, des erreurs ont été volontairement introduites (retrait de la variable espèce dans l'éditeur de variable et duplication de la

dernière clause) dans l'exemple montré un peu plus haut pour permettre l'apparition de messages d'erreurs :

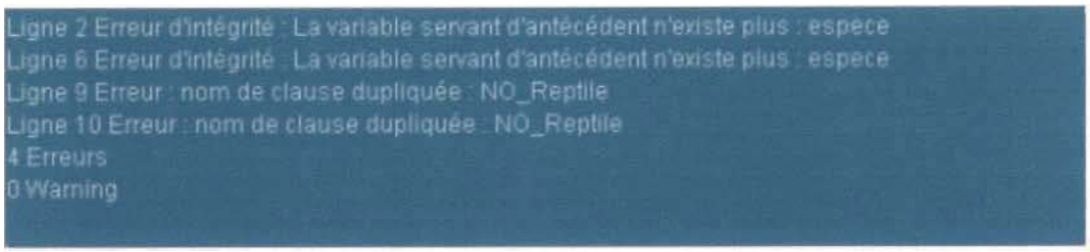

Figure 63 : Message d'erreurs de la fenêtre (éditeur de clauses)

Remarque :

La validation offerte aux niveaux de l'éditeur de clauses et de l'éditeur de règles est beaucoup moins puissante que celle offerte au niveau de l'éditeur de systèmes. L'idée ici est de permettre à l'utilisateur de valider une partie seulement d'une base de connaissances lors du développement. La validation du système effectue beaucoup plus de vérifications et de validations que la validation des clauses. Cependant, elle se fait dans un cadre plus global. La validation offerte au niveau de l'éditeur de clauses n'est qu'une validation partielle. Par exemple, aucune validation n'est faite ici sur les noms des clauses qui doivent être des identificateurs. Par contre, une vérification de la sorte sera effectuée au niveau de la validation du système.

5.3.5.3 Éditeur de règles

Comme mentionné un peu plus haut, une règle est composée d'une liste de clauses antécédentes et d'une clause conséquente. Une règle se lit comme suit :

- **Règle r** :

  Si antécédent 1 (clause) est vérifié (vrai) **et que**
  antécédent 2 (clause) est vérifié (vrai) **et que**
  :
  antécédent N (clause) est vérifié (vrai)
  Alors
  exécution du conséquent (clause)

Pour créer une règle, il est nécessaire de lui spécifier un nom. Par la suite, il faut choisir parmi la liste des clauses disponibles (la liste des clauses de l'éditeur de clauses), les antécédents de la règle et les ajouter à la liste des antécédents de la règle. Par la suite, il faut choisir le conséquent de la règle à partir de la même liste de clauses. Si l'utilisateur détermine que des clauses sont manquantes pour la création de règles, il n'a qu'à retourner dans l'éditeur de clauses et ajouter les clauses nécessaires (Figure 64).

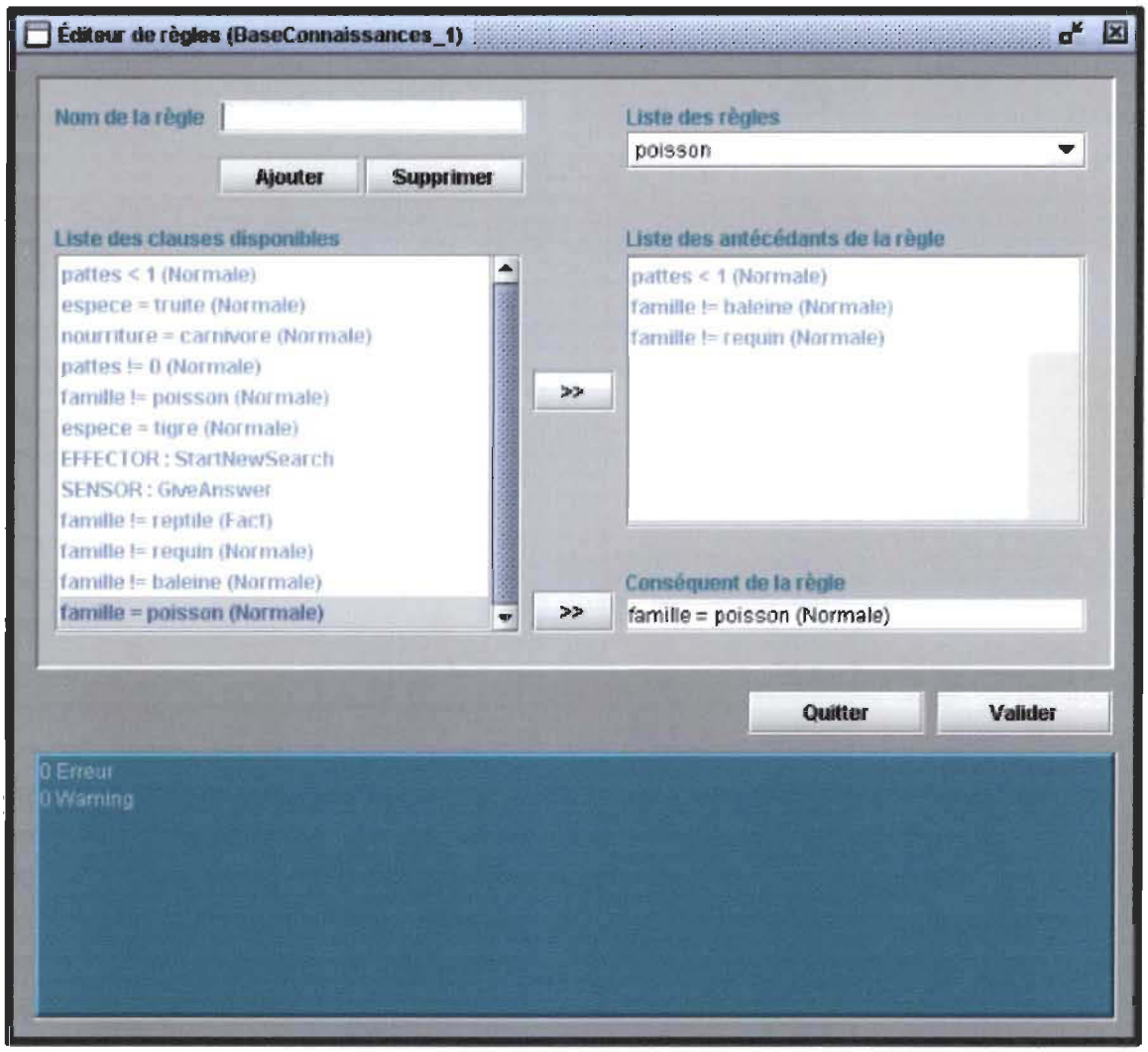

Figure 64 : Éditeur de règles

Remarque :

Des restrictions s'imposent sur le choix des clauses qui composent les règles. Les clauses antécédentes de la règle doivent être de type Sensor ou Normal. La clause conséquente d'une règle doit être de type Effector ou Normal. De plus, pour une clause conséquente d'une règle, si le type est Normal alors l'opérateur doit nécessairement être « = », qui sera utilisé dans ce cas-ci comme opérateur d'affectation. Ceci est tout à fait plausible car si la liste des antécédents est vérifiée, alors la règle doit déterminer qu'une variable possède une valeur en particulier ou effectuer une action en particulier dans le cas d'un Effector.

L'éditeur de règles possède un utilitaire de validation partielle semblable à celui de l'éditeur de clauses. Celui-ci permet de vérifier que les règles ne possèdent pas d'erreurs de duplication de clauses, de clauses ne possédant pas le bon type (soit dans les antécédents ou comme conséquent), que l'opérateur de la clause conséquente est bien l'opérateur d'affection. Comme dans la validation offerte dans l'éditeur de clauses, une validation permettant de vérifier la cohérence des règles est offerte. Cette validation permet de vérifier que les clauses qui font parties des règles existent bien dans la table des clauses de la base de connaissances.

5.3.5.4 Exportation et importation des bases de connaissances

Exportation

Comme mentionné un peu plus haut, il est possible de sauvegarder une base de connaissances dans un fichier binaire dans le but de la réutiliser dans un autre projet. Il suffit de choisir dans la liste des bases de connaissances celle qui est à exporter; le nom de celle-ci apparaîtra dans la boîte de texte « nom ». Par la suite, il faut choisir l'option « exporter ». Une fenêtre demandera l'endroit où sauvegarder la base de connaissances et le nom à donner au fichier contenant celle-ci. Une fois la sauvegarde effectuée, la base de connaissances est déjà exportée. Elle est sauvegardée dans un fichier binaire portant le nom spécifié avec l'extension « .rlb ».

Importation

L'importation d'une base de connaissances est aussi simple. En effet, il suffit de cliquer sur le bouton à droite de la boîte de texte « nom ». Une fenêtre apparaît à l'écran; elle permet de choisir un fichier contenant une base de connaissances. Lorsque la sélection est faite, le nom de la base de connaissances est ajouté à la liste des bases de connaissances du projet en cours. Il est désormais possible d'utiliser cette base de connaissances comme toutes les autres qui ont été créées à l'intérieur de ce projet (il est possible d'ajouter, modifier ou de supprimer des variables, clauses ou règles).

5.3.6 Options des JVM (sous-systèmes)

Quelques options sont déterminantes pour le comportement de chaque JVM. La liste déroulante au haut de la figure 65 détermine sur quelle JVM les actions seront effectuées.

Figure 65 : Différentes options d'un sous-système

5.3.6.1 DF (*Directory Facilitator*)

Les DF offrent plusieurs possibilités intéressantes. La première est la possibilité d'utiliser un DF pour faire le pont entre les agents d'une JVM et les autres JVM. C'est l'utilité principale du « Simple DF ». En effet, si plusieurs agents s'exécutent sur la même machine et que chacun possède une *Mailbox* qui écoute les messages provenant des agents des autres sous-systèmes (via RMI ou un type de *socket*), alors la quantité de ressources demandées peut vite se dégrader et rendre le programme moins efficace. Le problème vient du fait que chaque *Mailbox* de chaque agent de la JVM écoute les messages via un *Thread* et que chacun de ceux-ci nécessite des ressources supplémentaires de la JVM. Pour éviter cet inconvénient, il est nécessaire de spécifier à tous les agents de la JVM d'écouter en mode direct (un agent qui écoute en mode direct peut recevoir des messages seulement des agents qui se trouvent sur la même JVM que lui). De cette façon, les agents ne créeront pas de *Thread* inutiles pour la communication externe. En contrepartie, il faut aussi spécifier à la JVM un « *Simple DF* ». Celui-ci recevra tous les messages provenant des autres JVM et les redistribuera à leurs destinataires (les agents de la JVM peuvent recevoir des messages du DF car il est sur la même machine). De plus, le « *Simple DF* » s'occupe de l'envoi des messages des agents de sa JVM aux destinataires se situant sur les autres JVM. *Tout ce mécanisme est totalement automatisé. Il suffit de spécifier à une JVM qu'elle aura un « Simple DF ».* D'autres types de DF sont disponibles. Comme par exemple, un DF permettant de déterminer quel agent peut exécuter une tâche en particulier.

5.3.6.2 RA (*Register Agent*)

Ce type d'agent est très utile car il permet un enregistrement automatique des agents d'une JVM auprès des autres JVM. Deux types de *Register Agent* sont disponibles : *SocketRegisterAgent* et *MulticastRegisterAgent*.

SocketRegisterAgent

- Lorsqu'un agent de ce type est spécifié, il est nécessaire de connaître les adresses IP des JVM. Il faut spécifier la liste des adresses dans l'onglet adresses comme mentionné un peu plus haut et jumeler à chaque JVM une adresse contenue dans le menu déroulant « adresse de la JVM ». De cette façon, lorsque les JVM seront exécutées, les agents de chaque JVM s'enregistreront automatiquement auprès des autres JVM (dont les adresses IP sont connues). Lors de la génération automatique du code de la classe « *main* » des JVM, si aucune adresse ne se trouve dans la liste des adresses IP (onglet), la liste des arguments (liste de *String*) sera passée au *RegisterAgent*. Concrètement, cela signifie que les adresses IP pourront être spécifiées en ligne de commandes.

MulticastRegisterAgent

- Lorsqu'un agent de ce type est spécifié, il n'est pas nécessaire de connaître les adresses IP des JVM. L'agent communique avec les autres *MulticastRegisterAgent* se situant sur d'autres machines par l'envoi de messages sur une adresse *multicast* où tous ces RA écoutent. De cette façon, lorsque les JVM seront exécutées, ces agents enregistreront automatiquement les agents de leur JVM auprès des autres JVM (dont les adresses IP sont inconnues). Il est à noter ici qu'aucune intervention du développeur n'est nécessaire. En aucun moment les adresses des sous-systèmes ne sont spécifiées.

5.3.6.3 Adresse de la JVM

Adresse à laquelle la JVM est jumelée. Cette option est utile lorsque l'on connaît au préalable les adresses des JVM. Ceci évite d'avoir à entrer les adresses des sous-systèmes à chaque exécution du système.

5.3.6.4 Ajout d'agents à une JVM

Type de l'agent à ajouter

- Ce menu déroulant contient la liste des noms de types qui ont été créés avec l'onglet « types d'agent ». Pour obtenir la spécification d'un type d'agent en particulier, il faut choisir le type voulu et retourner à l'onglet « types d'agent ».

Nom de l'agent à ajouter

- Nom de l'agent qui sera ajouté à cette JVM.

Ajouter

- Ajoute un agent dont le type sera celui spécifié dans le menu déroulant « type de l'agent à ajouter » et portant le nom spécifié dans la boîte de texte « nom de l'agent ». L'agent sera ajouté au sous-système qui porte le nom sélectionné dans le menu déroulant au haut du formulaire présenté à la figure 65.

5.3.7 Table des agents

Lorsqu'un agent est ajouté à une JVM, il est ajouté à la table des agents de celle-ci. Du même coup, une liste d'attributs de l'agent est ajoutée à cette table (Figures 66 à 69). Plusieurs de ces attributs sont modifiables directement dans la table (comme le nom) tandis que d'autres ne le sont pas (comme le type d'agent).

Nom de l'agent	Idendificateur unique	Temps d'attente	Priorité de l'agent
Agent_1	Automatiquement	100000	Normale
Agent_2	ident_tt_x	Maximum	Normale
Agent_N	Automatiquement	Maximum	2

Figure 66 : Table des agents et de leurs attributs (partie 1)

5.3.7.1 Attributs d'un agent

Nom de l'agent

- Nom que possédera la classe générée de cet agent (modifiable).

Identificateur unique (GUID)

- C'est l'identificateur de l'agent. Cet identificateur doit être unique à l'intérieur du système. Par défaut, l'identificateur sera attribué automatiquement (modifiable).

Temps d'attente

- Temps que l'agent reste inactif (en millièmes de secondes) lorsqu'un appel à la méthode « *waiting()* » est fait. Par défaut, le temps d'attente est initialisé à la valeur « *Long.MAX_VALUE* » (modifiable).

Priorité de l'agent

- La priorité d'exécution de l'agent dans la JVM. Par défaut, la priorité de l'agent est initialisée à « *normal* ». Les valeurs possibles sont des valeurs entières comprises entre 1 et 10 inclusivement; 10 étant la priorité maximale qu'un agent puisse avoir et 1 la priorité minimale (modifiable).

Type d'agent (créé)	Type de mailbox	Port (si nécessaie)	Adresse multicast (si néc...
TypeAgent_1	SocketMailbox	8080	
TypeAgent_1	SocketMailbox	8080	
TypeAgent_2	RMIMailbox	1099	

Figure 67 : Table des agents et de leurs attributs (partie 2)

Type d'agent

- Le type d'agent spécifié dans le menu déroulant « type de l'agent à ajouter » (non-modifiable).

Type de Mailbox

- Le type de boîte de messagerie pour la réception des messages. Le type est celui qui a été spécifié au niveau du type de l'agent. Par contre, celui-ci n'est qu'une valeur par défaut. (modifiable).

Port

- Le port sur lequel la *Mailbox* de l'agent écoute les messages (lorsque nécessaire). Par défaut, chaque type de *Mailbox* possède un port par défaut. Cependant, si deux agents sur la même JVM possèdent le même type, il est nécessaire de différencier les ports car deux agents d'une JVM ne peuvent écouter sur le même port.

Adresse multicast

- L'adresse nécessaire lorsque le type de *Mailbox* est *MulticastMailbox*. Ce type de *Mailbox* possède une adresse *multicast* par défaut (modifiable).

Type de comportement	Redéfinir le comportement	Attente de paramètres	Envoie de résultats
OnRequestBehaviour	☐	☐	☑
OnRequestBehaviour	☐	☐	☑
OnRequestBehaviour	☐	☑	☑

Figure 68 : Table des agents et de leurs attributs (partie 3)

Type de comportement

- Type de comportement spécifié au niveau du type d'agent (non-modifiable).

Redéfinir le comportement

- Spécification au niveau du type d'agent (non-modifiable).

Attente de paramètres

- Spécification au niveau du type d'agent (non-modifiable).

Envoi de résultats

- Spécification au niveau du type d'agent (non-modifiable).

Figure 69 : Table des agents et de leurs attributs (partie 4)

Faire autres choses

- Spécification au niveau du type d'agent (non-modifiable).

Type

- « *BDIAgent* » ou « Autre » Spécification au niveau du type d'agent (non-modifiable).

Base de connaissances

- La base de connaissances associée à l'agent (si nécessaire). La liste des bases de connaissances est celle qui est définie dans l'onglet BDI. Il faut noter que si le type de l'agent est « Autre » il est inutile de spécifier une base de connaissances (la figure 69 démontre que la liste des bases de connaissances est disponible directement dans la table des agents) (modifiable).

Interaction avec UI

- Détermine si l'agent pourra envoyer des messages à la file des événements AWT de Java (modifiable).

5.3.8 Table des tâches

Cette table est composée de la liste de tâches qui ont été spécifiées dans l'onglet « tâches ». Pour spécifier qu'un agent peut effectuer une tâche, il suffit de cocher la tâche pour l'agent.

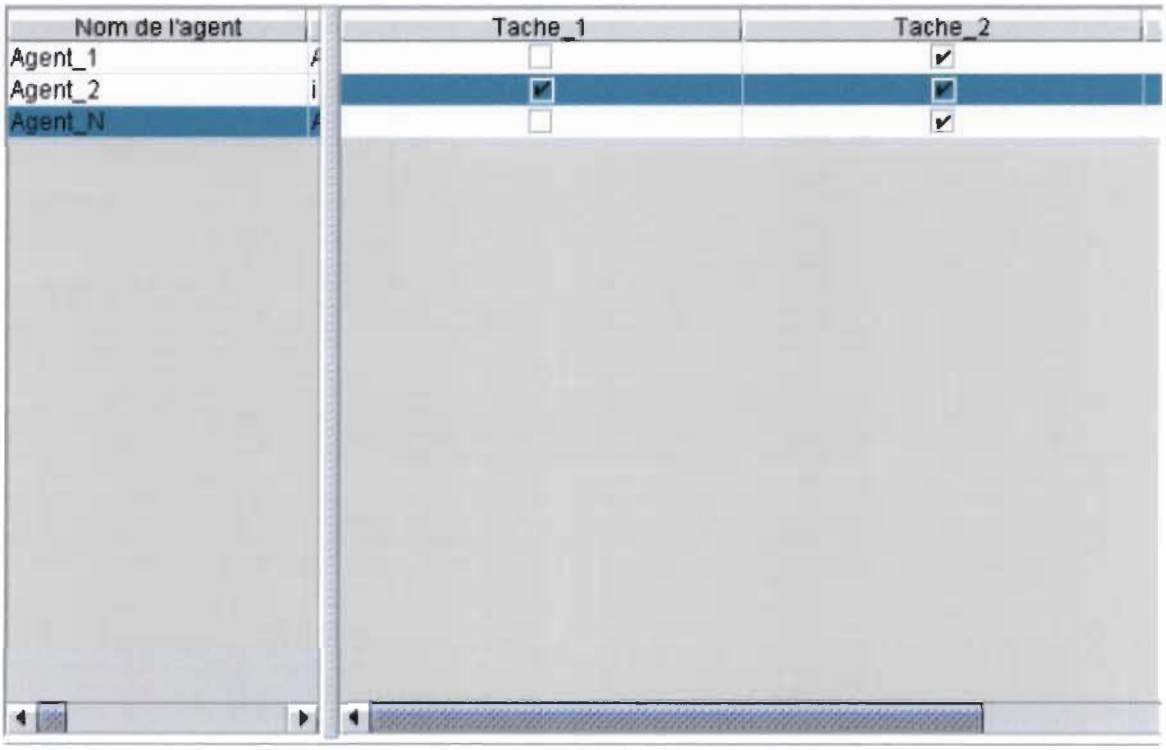

Figure 70 : Table des agents et table des tâches

5.3.9 Liste des éléments spécifiés graphiquement

La section précédente a passé en revue les différents éléments permettant la spécification graphique des différents composants d'un SMA. La grande majorité des éléments d'un système peuvent être spécifiés par l'intermédiaire des interfaces utilisateurs :

- Les adresses IP des différents sous-systèmes (JVM).
- Les sous-systèmes (JVM).
- Les différents types d'agents du système :
  - Leur comportement,
  - Leur mode de communication,
  - Leur type (BDI ou autre).
- Les tâches de chaque agent.
- Les bases de connaissances des agents :
  - Les variables,
  - Les faits,

- Les clauses,
- Les règles.

- Les DF (*Directory Facilitator*).
- Les RA (*Register Agent*).
- Les agents des sous-systèmes (et leurs attributs) :
  - Le nom,
  - Le GUID (*Global Unique Identifier*),
  - Le temps d'attente de l'agent,
  - La priorité d'exécution de l'agent,
  - Le type de l'agent,
  - Le mode de communication de l'agent,
  - Le port pour la communication,
  - L'adresse *multicast* (si nécessaire),
  - Le type de comportement,
  - La redéfinition de la méthode comportementale,
  - La redéfinition de la méthode d'attente de paramètres,
  - La redéfinition de la méthode de retour de résultats,
  - La redéfinition de la méthode des autres actions à effectuer,
  - Le type d'agent (BDI ou Autre),
  - La base de connaissances (si nécessaire),
  - L'envoi d'événements à la file d'événements AWT de Java.

5.3.10 Validation, génération, exportation et archivage du SMA

Toutes ces spécifications ne seraient pas vraiment utiles et intéressantes s'il n'y avait pas les quatre options suivantes : la validation, la génération, l'exportation et l'archivage.

Figure 71 : Barre d'outils où sont situés les 4 principales options

5.3.10.1 La validation automatique du système

Cette option permet de vérifier la logique de la spécification des différents composants du système. La validation permet de trouver des incohérences à l'intérieur des composants ou dans les liens qui les unissent. De plus, la validation permet de vérifier que les mots utilisés (les noms, les types et les variables) sont valides pour la compilation. Ceux-ci doivent être acceptables en tant qu'identificateur au sens de Java. La validation permet aussi d'éviter les erreurs de compilation lors de la génération des composants du système.

La validation du système vérifie tous les composants énumérés dans la section précédente et tente de trouver des irrégularités, des inconsistances ou des incohérences qui sont des sources potentielles d'erreurs. Celles-ci sont exprimées à l'utilisateur sous formes de

messages d'erreurs ou de messages d'avertissements, dépendamment du niveau de risque d'erreur qu'entraîne cette irrégularité (voir figure 72).

Éléments vérifiés pour chaque composant (sauf les adresses IP)

- Tous les noms utilisés pour les différents composants doivent être des identificateurs au sens de Java (pour éviter des erreurs de compilation une fois la génération du système effectuée).

Autres éléments vérifiés

Les adresses IP

- Toutes les adresses IP doivent être uniques et utilisées par une JVM.

Les JVM

- Les noms des JVM ne sont pas dupliqués,
- Chaque JVM possède au moins un agent,
- Les *Register Agent* sontdu même type (s'ils en ont),
- Deux JVM n'ont pas la même adresse IP.

Type d'agent

- Chaque type d'agent doit être utilisé par au moins un agent.

Tâche

- Chaque tâche doit être utilisée par au moins un agent.

Agent

- Tous les agents d'une JVM doivent écouter sur des ports différents.
- Les agents doivent avoir des noms différents.
- Les agents doivent avoir des GUID différents.
- Le temps d'attente doit être valide (compris entre 0 et *Long.MAX_VALUE*).
- La priorité doit être valide (comprise entre 1 et 10 inclusivement).
- Les agents de type « Autres » ne doivent pas avoir de base de connaissances.
- Les agents de type « *BDIAgent* » doivent posséder une base de connaissances.
- Chaque agent peut effectuer au moins une tâche.
- La *Mailbox* est valide :
  - Si elle est de type *Direct*, elle ne possède pas de port,
  - Si elle n'est pas de type *Direct*, elle possède un port valide,
  - Si elle n'est pas de type *multicast*, elle ne possède pas d'adresse *multicast*.

Les bases de connaissances

- Aucune duplication de nom de base de connaissances.
- Aucune duplication de nom de variable.
- Une validation complète des clauses :
  - Aucune duplication de clauses,
  - Vérification de l'intégrité des clauses. Dépendamment du type de clause (*Normal, Sensor, Fact, Effector*), les éléments nécessaires à chacune sont spécifiés (voir figure 62),
  - Les variables utilisées à l'intérieur des clauses existent dans la base de connaissances.
- Validation complète des règles :
  - Aucune duplication de règles,
  - Aucune duplication de clauses à l'intérieur d'une règle,
  - Type des clauses antécédentes valide (*Senseur ou Normale*),
  - Type des clauses conséquentes des règles valides (*Effecteur ou Normale*),
  - Les règles possèdent au moins un antécédent,
  - Les règles possèdent une clause conséquente,
  - L'opérateur des clauses conséquentes est « = » ou la clause est un effecteur,
  - Les clauses existent dans la liste des clauses de la base de connaissances.

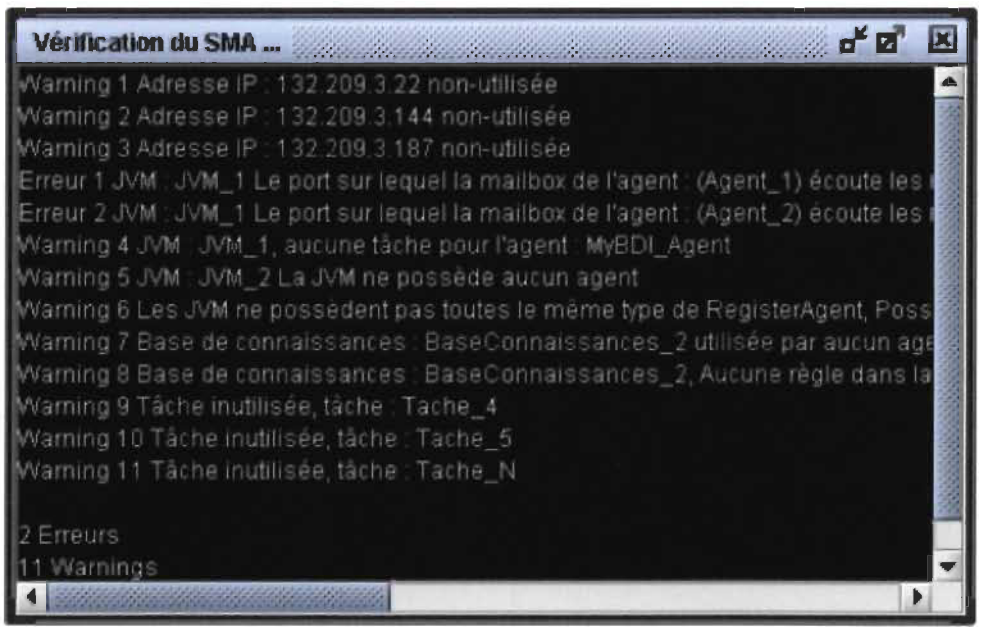

Figure 72 : Messages typiques de la validation d'un système

5.3.10.2 Génération automatique du code source de l'application

La génération automatique du code source du système est probablement la tâche la plus importante fournie par l'environnement. Elle permet la génération complète et effective du code source de tous les composants de l'application (Figure 80). Le code source

généré est compilable et exécutable. Tous les fichiers source générés sont commentés pour une meilleure compréhension de la structure du système et pour trouver facilement les endroits où ajouter du code (Figure 82). Le code généré suit une hiérarchie bien définie (Figure 81). Le projet « **.sma** » se situe dans le dossier où l'utilisateur l'a déterminé. Dans ce même répertoire, un dossier « **app** » est créé automatiquement lors de la création du projet (Figure 73).

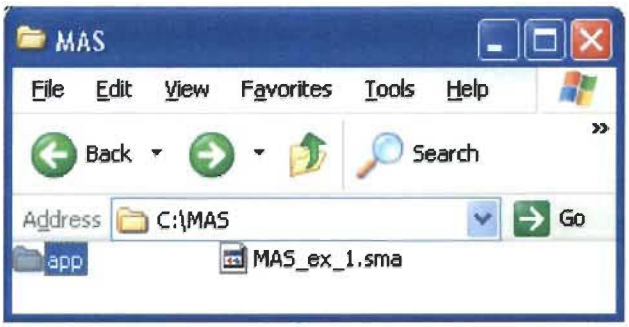

Figure 73 : Dossier où le projet est créé (MAS_ex_1.sma)

Ce dernier contient tous les sous-répertoires et les fichiers générés. Chaque sous-répertoire correspond à une JVM (Figure 74).

Figure 74 : Dossier « app » contenant les dossiers de chaque JVM

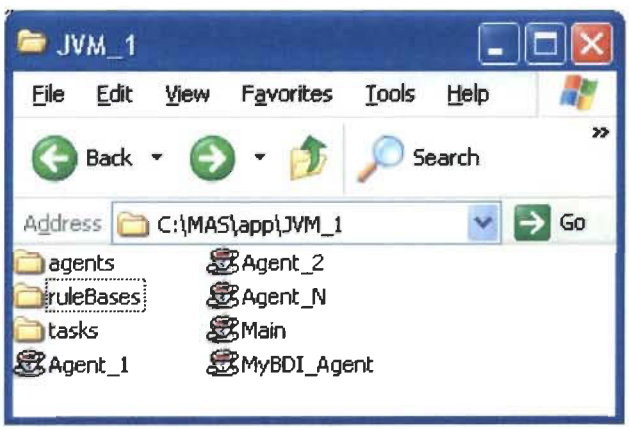

Figure 75 : Fichiers et dossiers contenus à l'intérieur du dossier d'une JVM

À l'intérieur d'un répertoire correspondant à une JVM, on retrouve un dossier correspondant aux tâches (***tasks***) effectuées par un ou des agents de cette JVM (le dossier contient les fichiers sources des différentes tâches).

Figure 76 : Dossier contenant les tâches utilisées par les différents agents d'une JVM

À l'intérieur d'un répertoire correspondant à une JVM, on retrouve aussi un dossier correspondant aux types d'agents utilisés à l'intérieur de cette JVM (le dossier contient les fichiers sources des différents types « ***agents*** »).

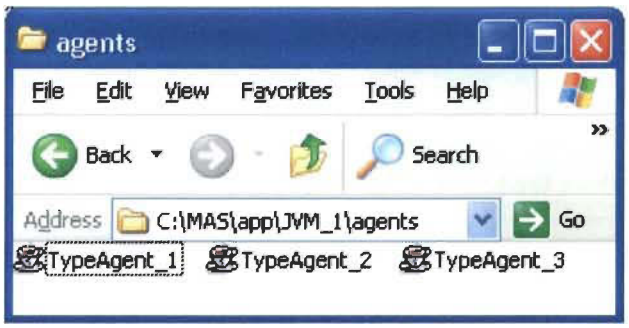

Figure 77 : Dossier contenant les types d'agents utilisés pour les agents d'une JVM

On y retrouve aussi un dossier correspondant aux bases de connaissances utilisées par un ou des agents de cette JVM (le dossier contient les fichiers sources des différentes bases de connaissances « ***ruleBases*** »).

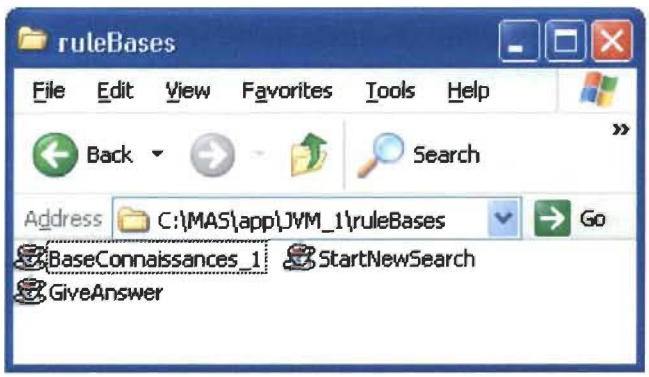

Figure 78 : Bases de connaissances utilisées par au moins un agent de la JVM

Dans le dossier d'une JVM, on retrouve un fichier source pour chaque agent et un fichier « ***Main.java*** ». Ce dernier étant le point d'entrée du programme.

Figure 79 : Point d'entrée du sous-système et le code de chaque agent de la JVM

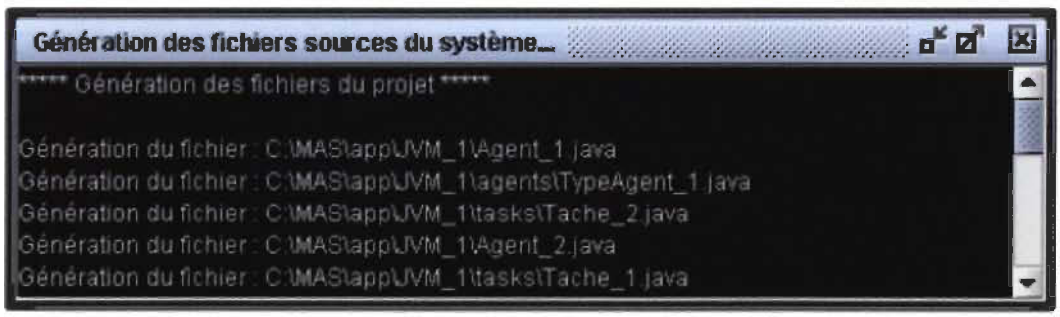

Figure 80 : Fenêtre apparaissant lorsque la génération automatique est effectuée

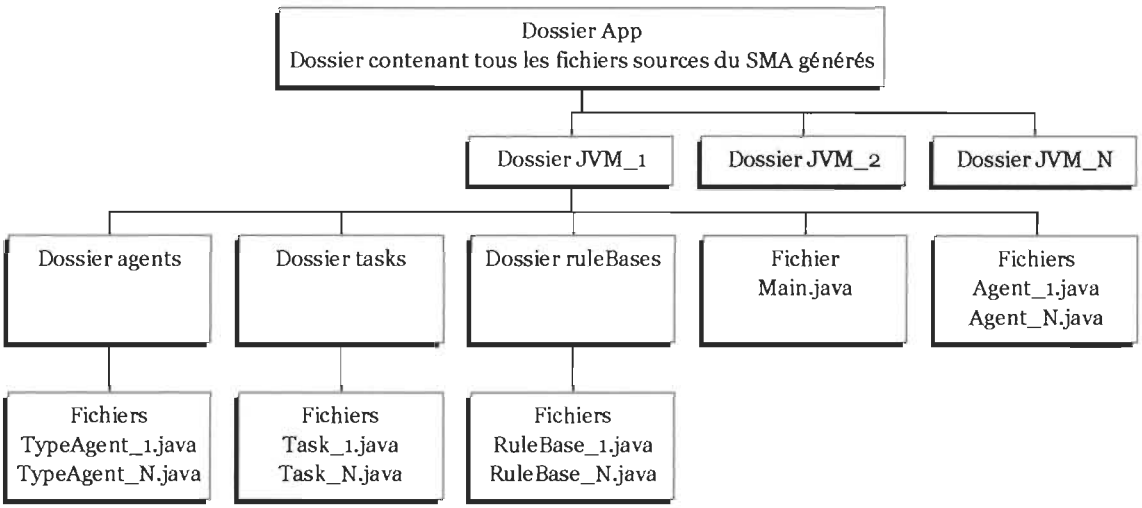

Figure 81 : Hiérarchie des fichiers sources générés d'un SMA

Remarque :

La génération du code source crée tous les fichiers nécessaires à une JVM en particulier. Si une base de connaissances, une tâche, un type d'agent ou autre composant n'est pas utilisé par une JVM, le code source de ce composant n'est pas généré pour cette JVM.

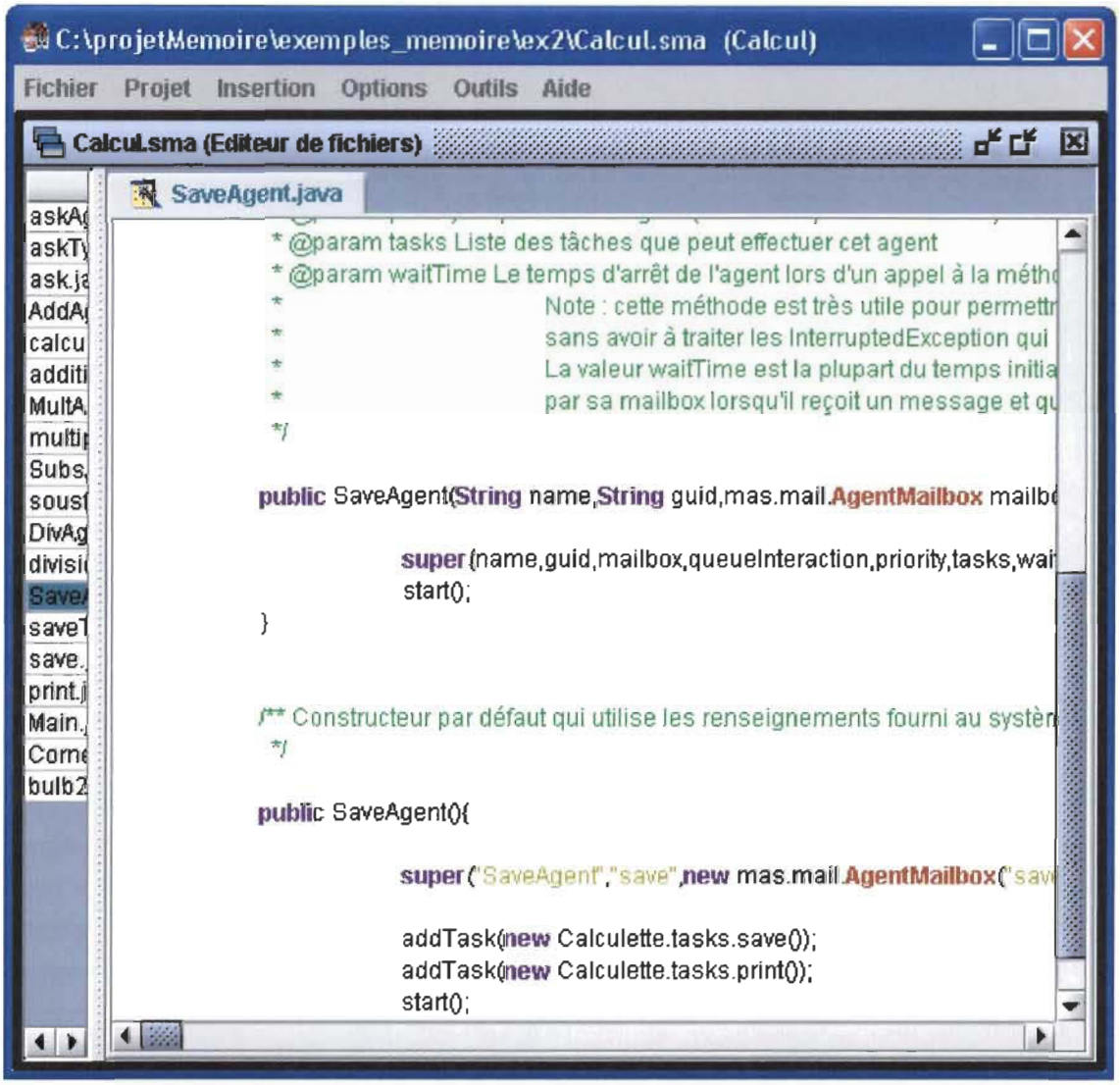

Figure 82 : Agent généré à partir des spécifications de l'éditeur de systèmes

5.3.10.3 Exportation du projet

Cette option permet de créer (copier) les fichiers appartenant à la librairie de classes de l'environnement nécessaires à l'exécution indépendante des sous-systèmes (Figure 83). Chaque sous-système possèdera les fichiers qui sont nécessaires à son exécution. Les fichiers nécessaires à un sous-système ne sont pas générés dans les dossiers des autres JVM. Cette procédure permet d'alléger chaque sous-système (en terme de volume). L'exportation du système permet une grande extensibilité du code et l'indépendance de

l'outil. Étant donné que les fichiers nécessaires appartenant à la bibliothèque SMA sont copiés dans le dossier du sous-système, il est possible d'exporter le dossier sur une autre machine et de continuer l'implémentation. De plus, il est possible de modifier les fichiers sources de la librairie SMA qui ont été copiés dans le répertoire du sous-système. De cette façon, il est possible d'étendre directement à l'intérieur d'un sous-système les fonctionnalités de la librairie OA.

Figure 83 : Messages apparaissant lors de l'exportation des classes

Cette opération créée un dossier « ***mas*** » à l'intérieur de chaque JVM contenant tous les fichiers de la librairie de classes OA nécessaires à l'exécution de cette JVM, et ce de façon indépendante de l'environnement (Figure 84).

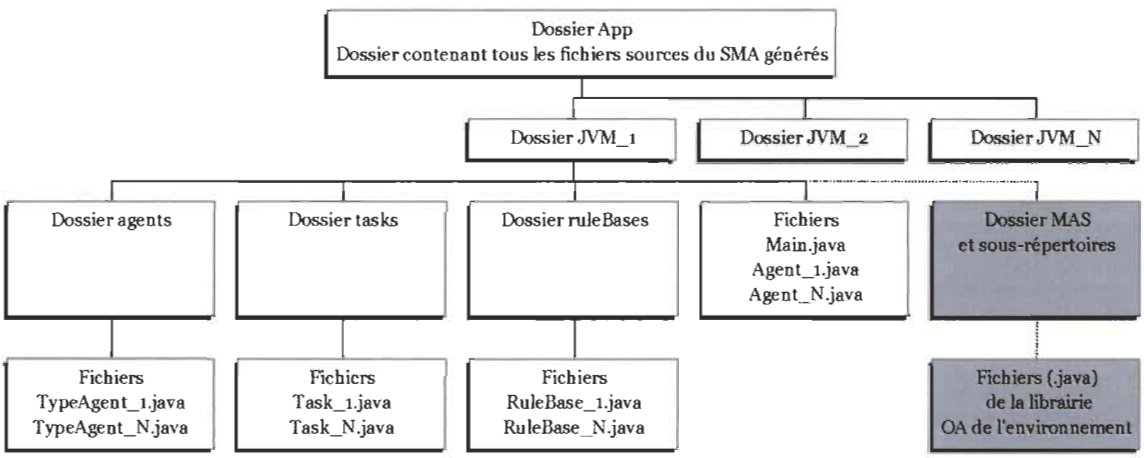

Figure 84 : Hiérarchie des fichiers après l'exécution de l'option "exportation"

Remarque

> *Les fichiers copiés pour une JVM ne sont pas nécessairement les mêmes que pour une autre JVM. Ce processus copie, dans le répertoire de la JVM, seulement les fichiers qui sont nécessaires à son exécution.*

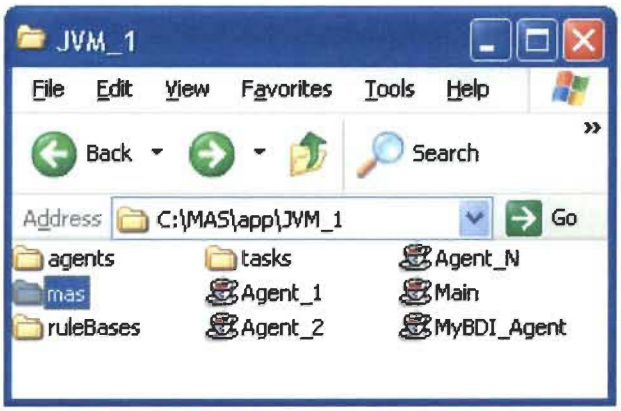

Figure 85 : Dossier « mas » contenant les sources de la librairie de classes OA

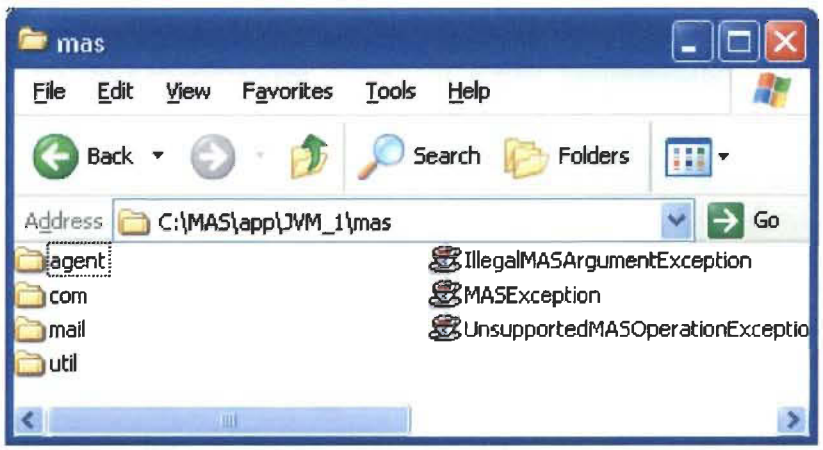

Figure 86 : Classes de la librairie OA nécessaires à l'exécution de cette JVM

5.3.10.4 Archivage des sous-systèmes

Cette fonctionnalité permet de créer une archive compressée « .jar » exécutable pour chaque sous-système (Figure 89). Celle-ci contient les fichiers binaires du sous-système et ceux de la librairie SMA nécessaires à l'exécution de la JVM. De plus, un fichier spécifiant le point d'entrée dans l'application et permettant l'exécution de l'archive s'y retrouve aussi. L'archive contient aussi les fichiers sources de l'application pour permettre de continuer l'implémentation de la JVM en dehors de l'environnement de développement. Il est aussi nécessaire de compiler le projet avant de créer les archives; sinon, les fichiers binaires ne se retrouveront pas dans l'archive.

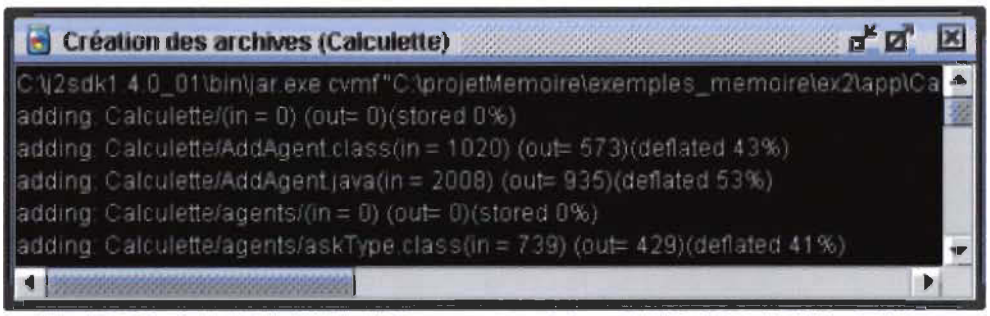

Figure 87 : Messages spécifiant la création des archives de la première JVM

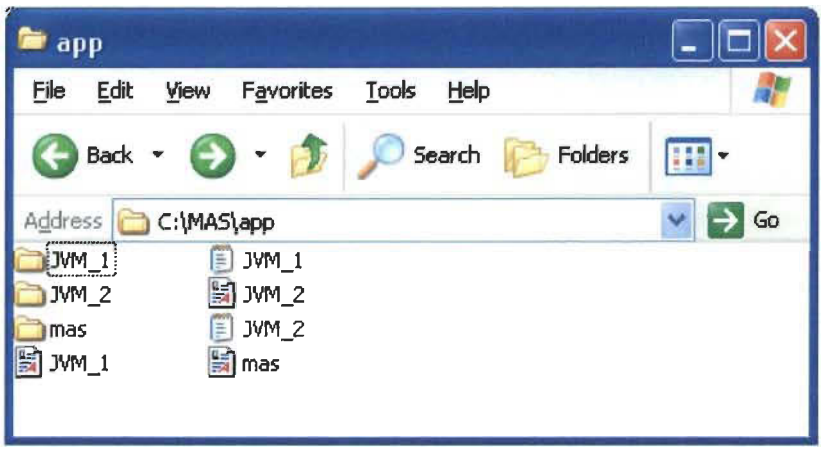

Figure 88 : Archives exécutables créées pour chaque JVM

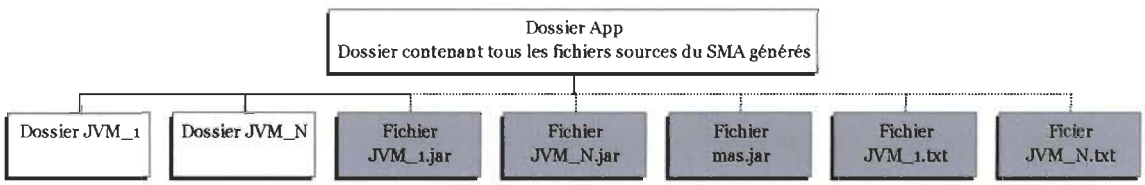

Figure 89 : Hiérarchie des fichiers après l'exécution de l'option "archivage"

5.4 Librairie de classes orientées-agent

L'environnement de développement est supporté par une librairie de classes OA permettant le développement rapide, simple, efficace et facilement extensible de SMA distribués. Compte tenu du nombre de classes de la librairie, il est impossible de passer en revue la majorité de celles-ci. Cependant, quelques-unes de ces classes doivent être abordées. De plus, les diagrammes UML de plusieurs classes ne peuvent être présentés dans ce document, étant donné leur taille imposante. Cela rend leur compréhension et leur lisibilité impossible.

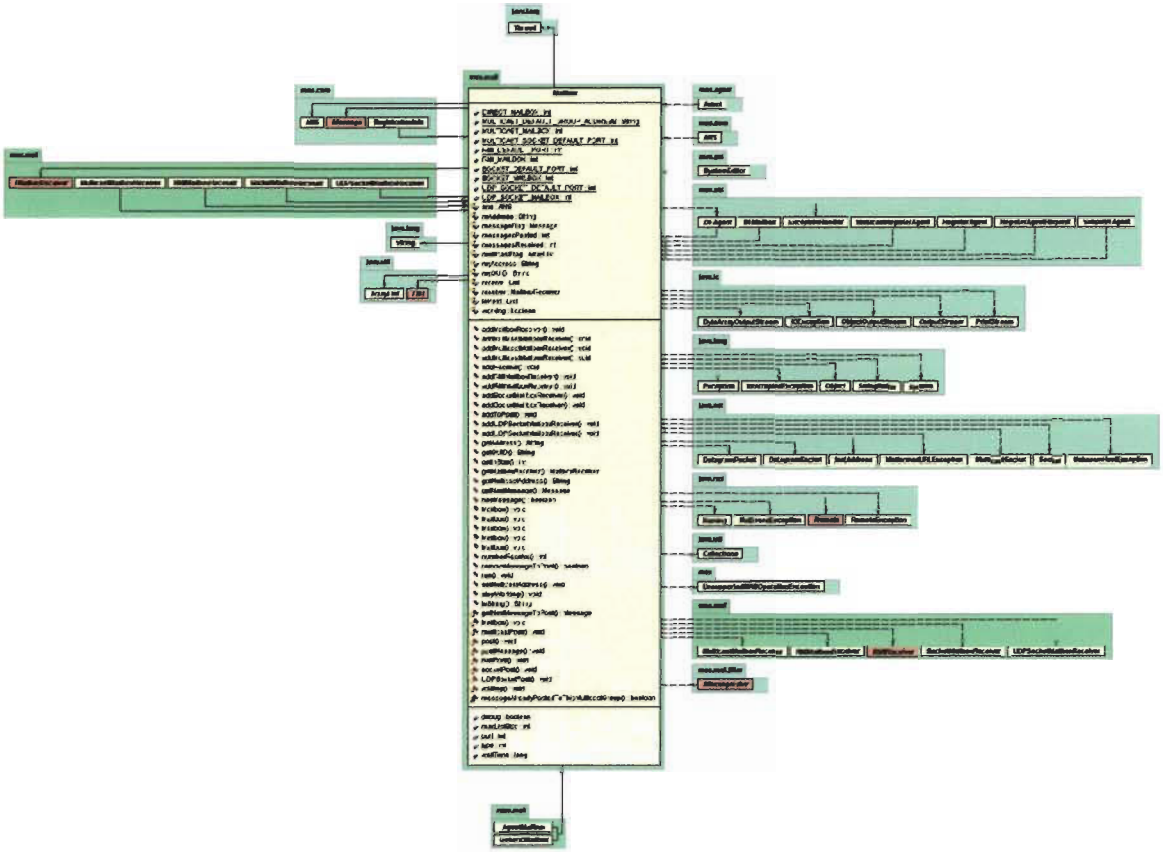

Figure 90 : Diagramme UML de la classe Mailbox

Un diagramme, comme celui présenté dans la figure 90, est difficilement lisible. Dans un souci de lisibilité, les caractéristiques de quelques classes seront donc présentées sous forme d'arbre hiérarchique. D'autres seront présentées sous formes de liste de méthodes, attributs et constantes.

5.4.1 La classe Agent

La classe *Agent* est la superclasse de tous les agents du système (Figure 91). Cette classe est un *Thread*; donc chaque agent s'exécute indépendamment des autres. Chaque agent possède un nom et un GUID. Ce dernier doit être unique dans tout le système. C'est à partir du GUID que l'agent est enregistré dans le système et que les autres agents peuvent lui envoyer des messages. Chaque agent possède une *AgentMailbox*, qui est une sous-classe de la classe *Mailbox* (qui est sa boîte d'envoi et de réception de messages). L'agent possède une liste de tâches qu'il peut accomplir. Il est possible de spécifier le comportement d'un agent. Pour se faire, il suffit d'implémenter l'interface correspondant au type de comportement souhaité. Les interfaces correspondantes aux comportements possèdent des méthodes qui, dans une application Java standard devraient normalement être définies dans la classe qui implémente l'interface. ***Par contre, dans le cas présent, il est déconseillé de redéfinir ces méthodes***. La classe redéfinie toutes les méthodes nécessaires à l'adoption de ces comportements. L'exécution d'une ou de plusieurs de celles-ci est dépendante de l'interface implémentée. Plusieurs autres utilités sont

comprises avec cette classe. Par exemple, un utilitaire qui permet d'envoyer et de recevoir des messages par la file des événements de Java.

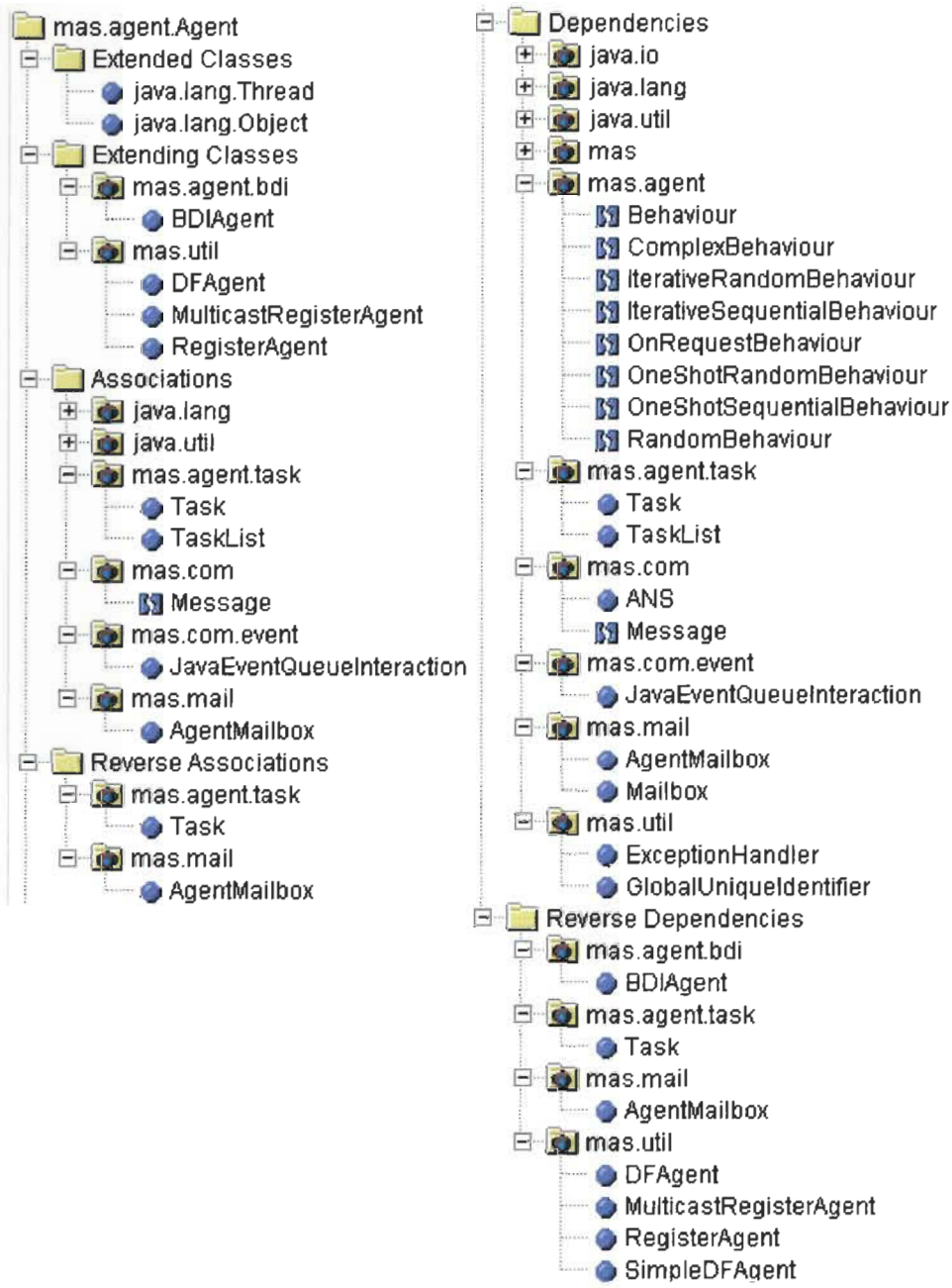

Figure 91 : Dépendances et associations de la classe Agent sous forme d'arbre

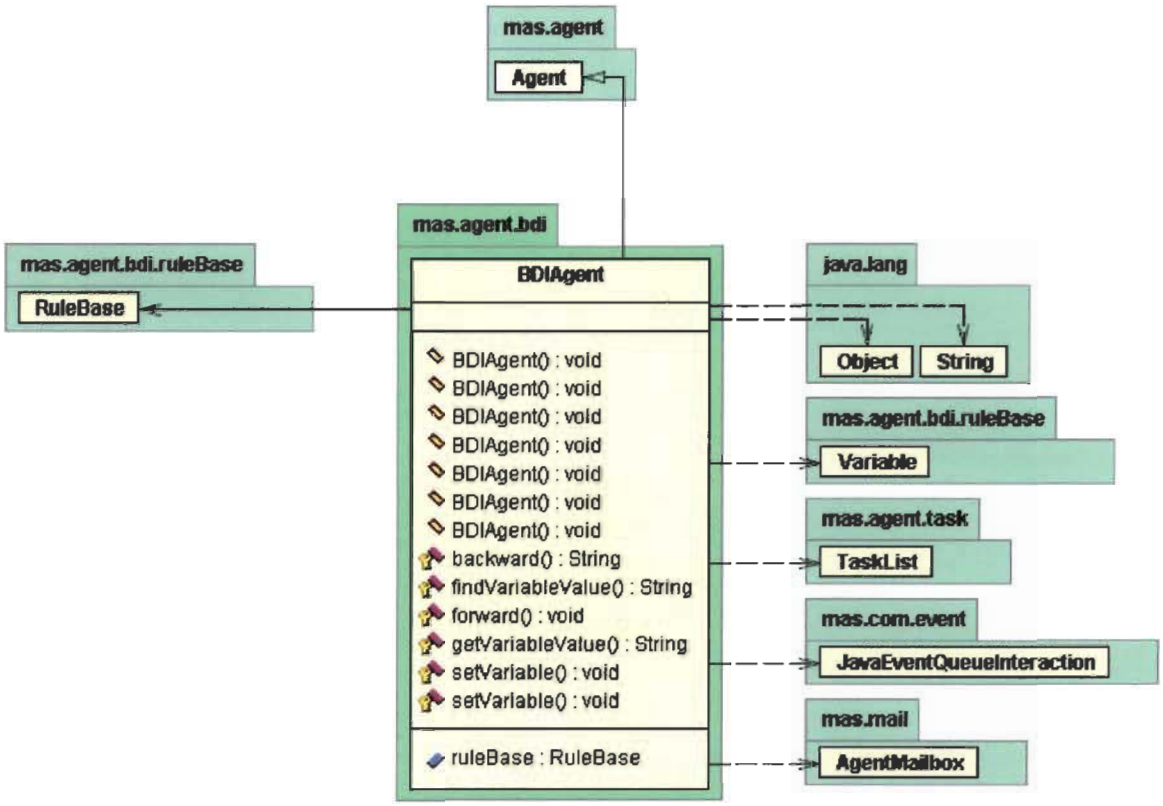

Figure 92 : Diagramme UML de la classe BDIAgent

5.4.2 La classe Mailbox

La classe *Mailbox* et ses sous-classes, comme *AgentMailbox* et *GenericMailbox*, ont deux mandats : recevoir et envoyer les messages. Le premier mandat de ces classes est de recevoir les messages provenant des autres agents (ou autres composants Java), qu'ils soient sur la même JVM ou sur un autre ordinateur. Lorsque la *Mailbox* de l'agent (ou du composant) reçoit un message, elle l'ajoute dans la file des messages entrants. S'il n'y avait aucun message dans la file, elle notifie à l'agent (lorsque c'est une *AgentMailbox*) la réception de ce message au cas où celui-ci serait en attente de messages. La réception des messages par une *Mailbox* peut se faire de différentes façons. Le mode direct permet d'ajouter des messages directement dans la boîte. Ceci implique que les agents et / ou composants qui s'envoient des messages sont soit sur la même JVM ou qu'un *SimpleDFAgent* se trouve sur la JVM de la *Mailbox* réceptrice. Les autres modes de réception sont : RMI, *socket* TCP, *socket* UDP et *multicast (Figure 93)*. Ces modes de réception peuvent recevoir des messages provenant des autres agents et / ou composants qui sont sur la même JVM que cette *Mailbox* ou sur d'autres sous-systèmes même, si aucun DF n'existe sur l'hôte.

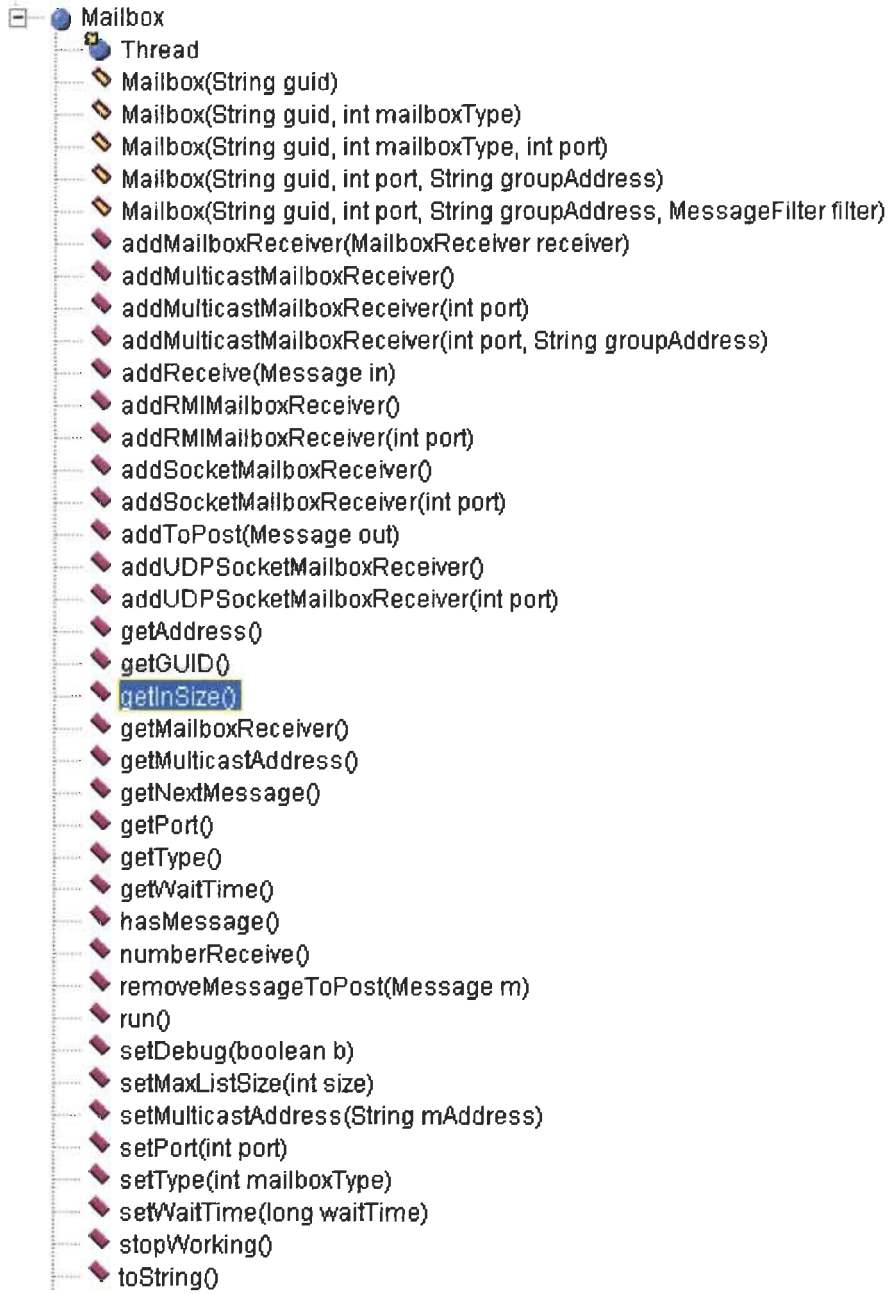

Figure 93 : Méthodes publiques d'une Mailbox

Le deuxième mandat de la classe *Mailbox* est d'envoyer les messages sortants de l'agent (ou composant) à leurs destinataires. Dans cette situation, le propriétaire de la *Mailbox* (l'agent ou le composant) ne fait qu'ajouter un message à envoyer dans la liste d'envoi. Sa *Mailbox* détermine les destinataires et leur mode de communication. Par la suite, la *Mailbox* envoie le message à chaque destinataire par le mode de réception du destinataire. Par exemple, si la boîte du destinataire écoute en *socket* TCP sur le port 5555, alors elle lui envoie le message en mode TCP sur le port 5555. Si un autre récepteur possède une boîte qui écoute sur une adresse *multicast*, alors elle lui envoie le message sur cette adresse. Pour permettre une meilleure efficacité et une continuité dans l'exécution d'un

agent (ou composant), le mécanisme d'envoi est exécuté à l'intérieur d'un *Thread* indépendant.

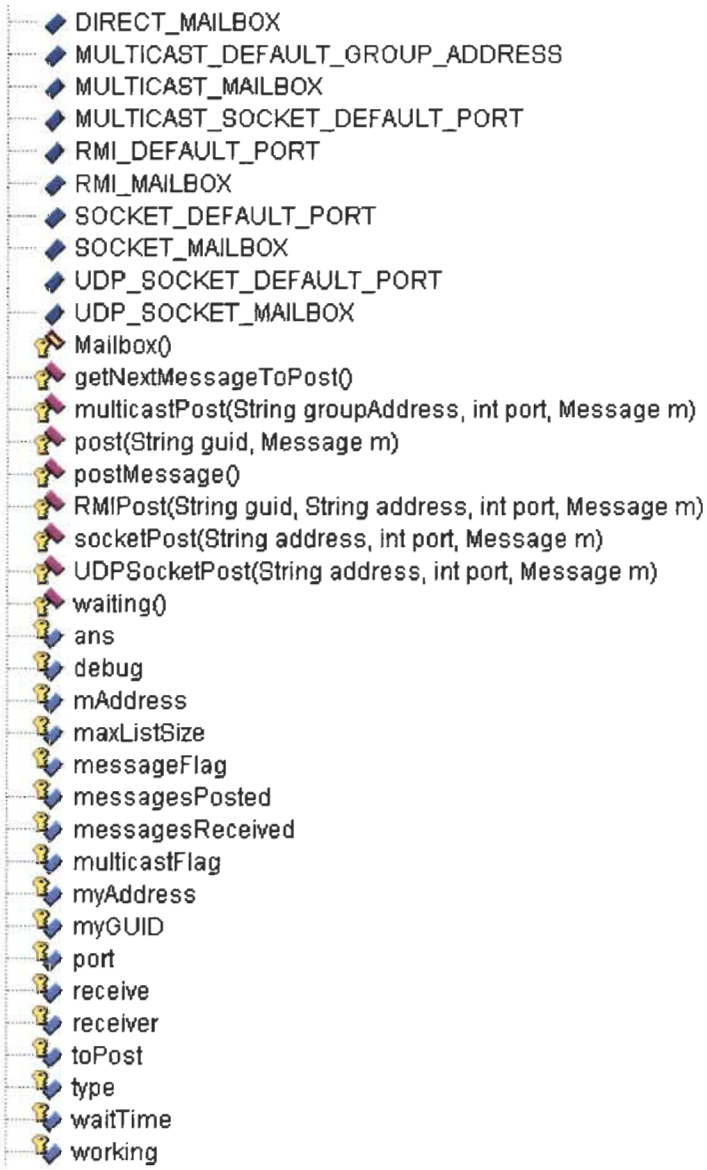

Figure 94 : Attributs, constantes publiques et méthodes privées d'une Mailbox

Ce mécanisme de communication permet de concevoir des SMA totalement distribués car chaque agent est libre de communiquer avec qui il veut sans avoir à passer par un « *broker* » et sans partager de ressources. Ce mécanisme est indépendant des autres entités du système. L'engin abstrait complètement les mécanismes de communication. De plus, il est totalement automatisé.

5.4.3 La classe Task

La classe *Task* est une classe générique qui permet de créer une tâche qui pourra être ajoutée à la liste des tâches qu'un agent peut effectuer. Cette classe contient une méthode « *execute* » qui doit être redéfinie. Cette méthode contient les actions qui doivent être effectuées pour considérer la tâche comme étant complétée. Un mécanisme permet de passer des paramètres pour l'exécution de la tâche et un autre mécanisme permet de retourner des résultats.

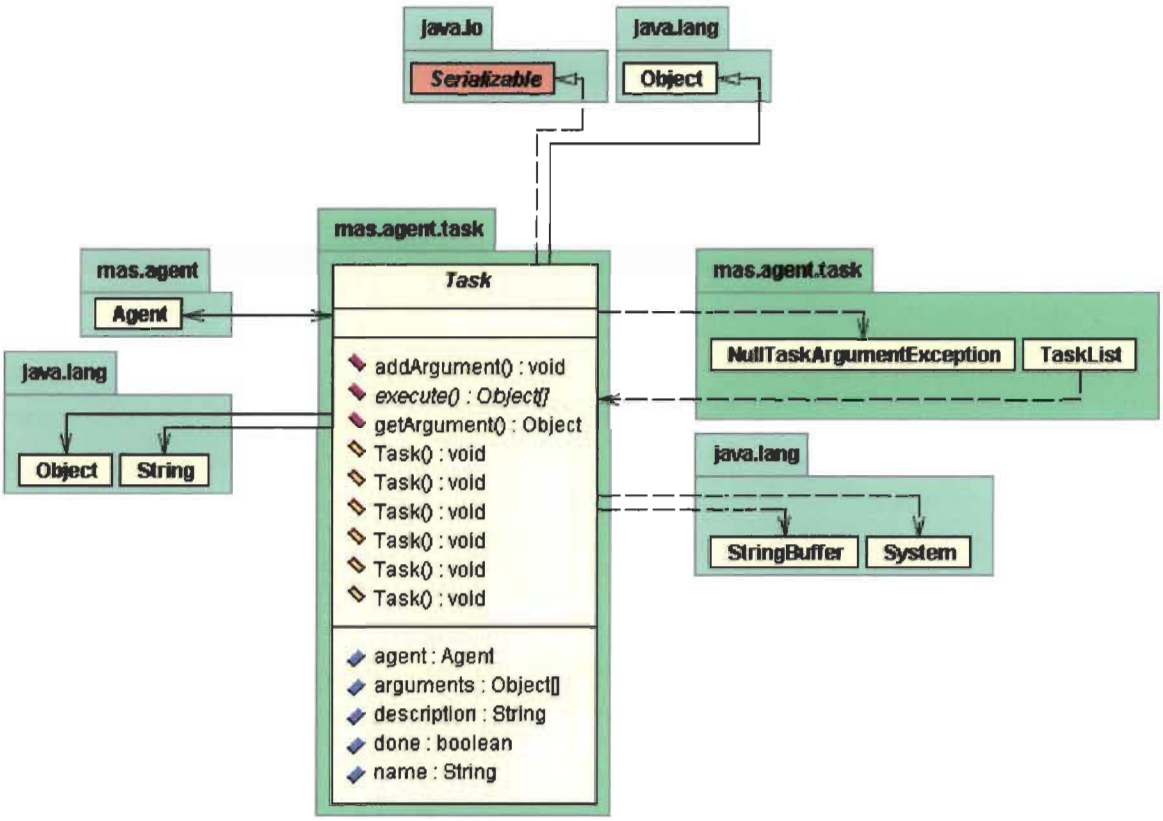

Figure 95 : Diagramme UML de la classe Task

5.4.4 L'interface Message

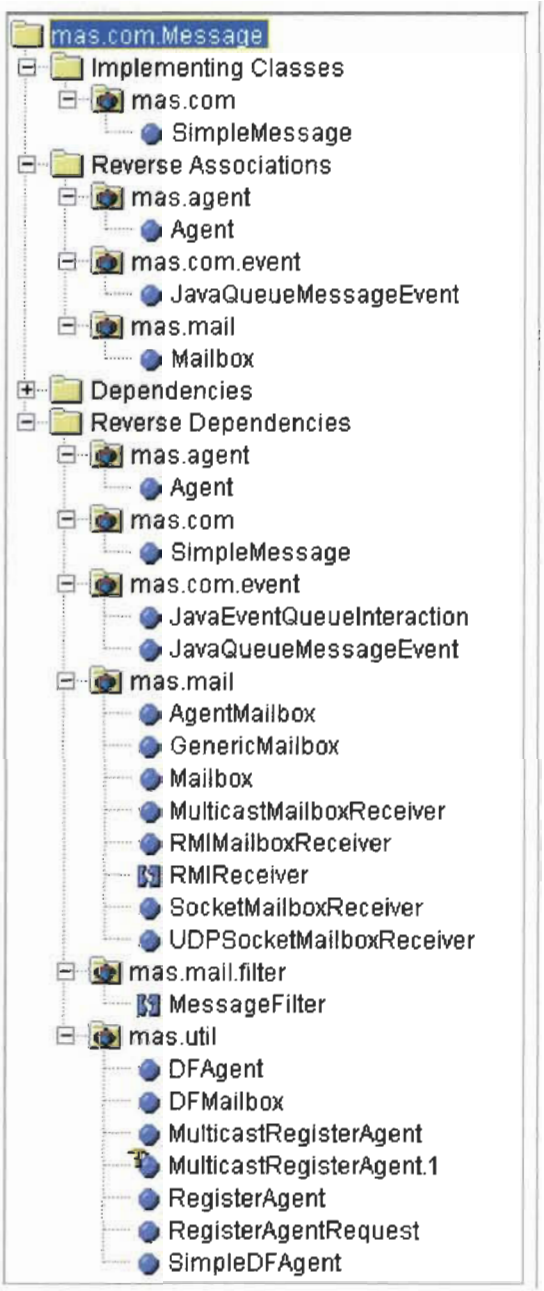

Figure 96 : Liens entre l'interface Message et les autres classes de la librairie OA

Cette interface doit être implémentée par tous les types de messages pouvant être envoyés d'un agent à un autre (Figure 97). Elle possède plusieurs méthodes utiles qui permettent à la *Mailbox* de déterminer à qui, quand, comment et où envoyer les messages.

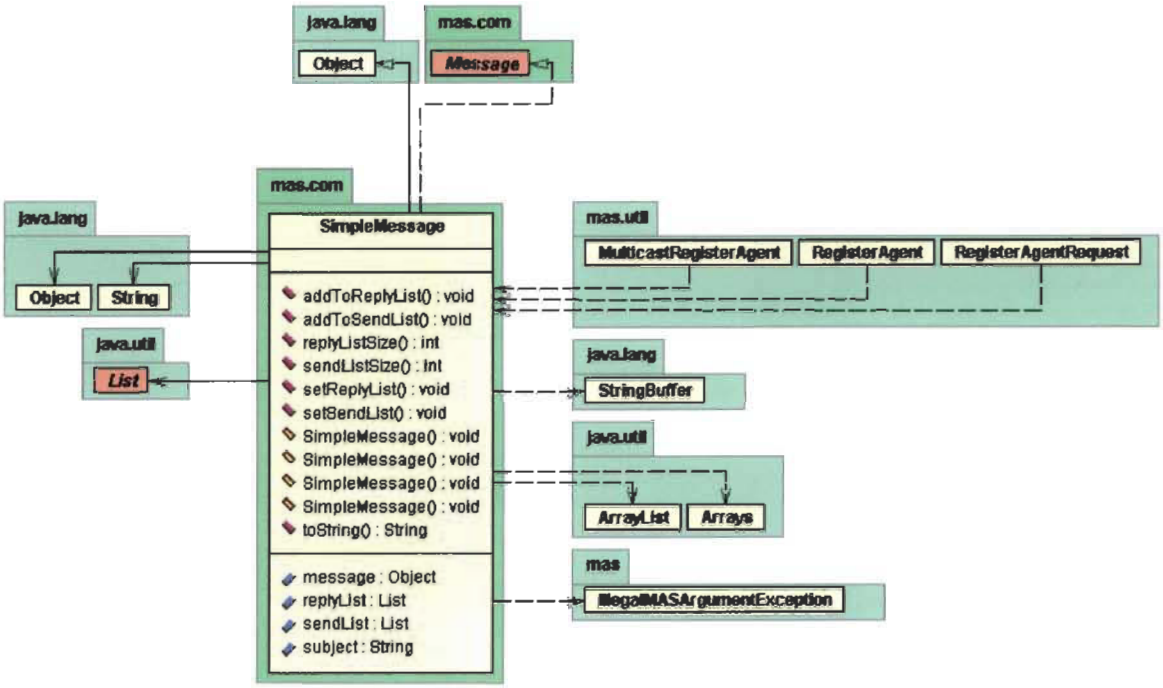

Figure 97 : Une des classes utilitaires implémentant l'interface Message

5.4.5 La classe ANS (Agent Name Server)

La classe ANS sert d'annuaire aux agents et aux *Mailbox*. Elle permet de déterminer les informations nécessaires à la communication. En effet, elle permet de trouver l'adresse, le mode de communication, le port (si nécessaire) et le nom d'un agent (Figure 98). Toutes ces informations sont disponibles en autant que l'on connaisse le GUID de l'agent dont on veut obtenir les informations.

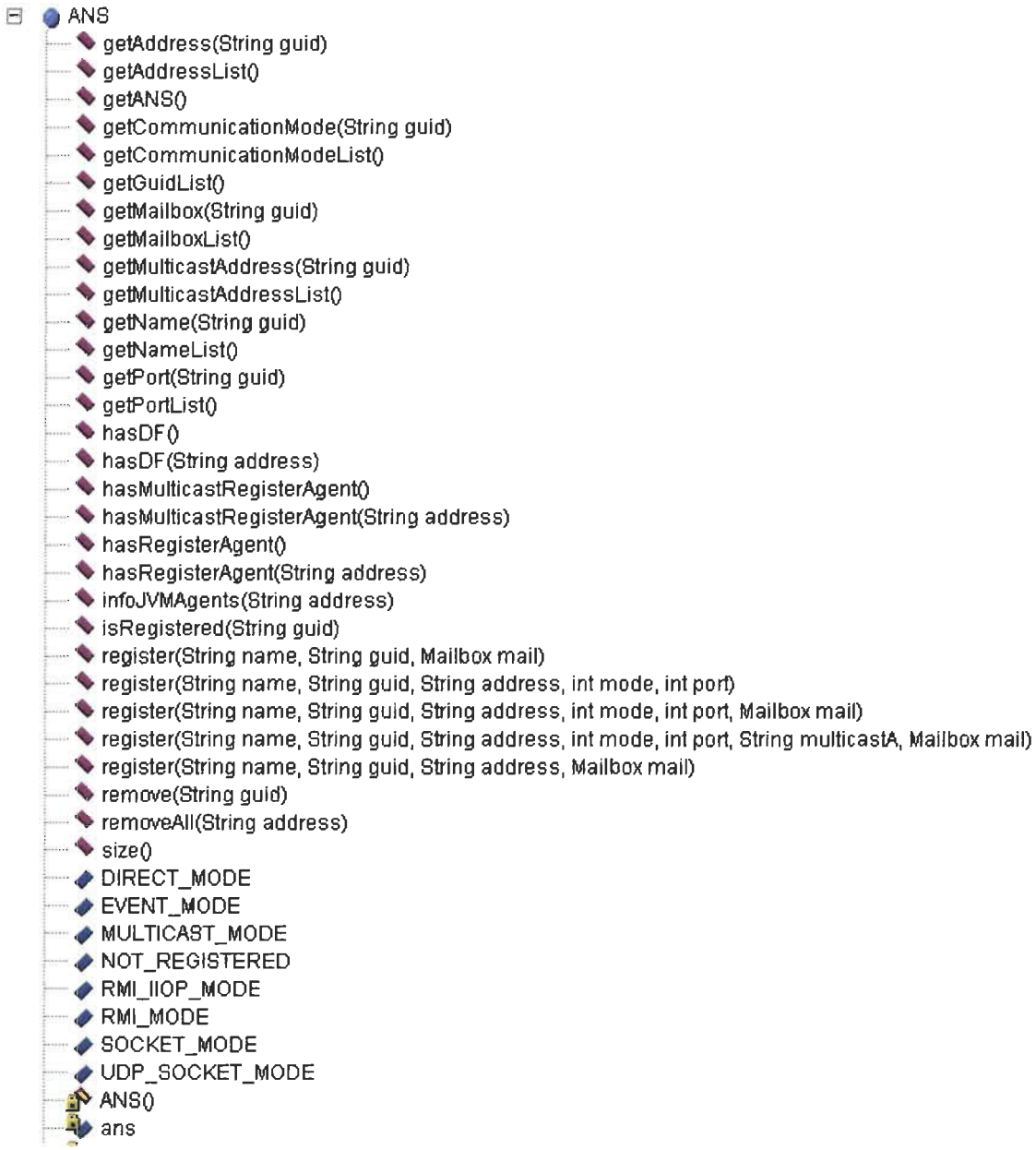

Figure 98 : Méthodes utilitaires de la classe ANS

5.4.6 Plusieurs classes pour les bases de connaissances

Il est possible de créer des bases de connaissances grâce à une quinzaine de classes et trois interfaces (voir figure 99). On peut les créer en utilisant les interfaces graphiques ou directement dans le code source. Ces bases de connaissances peuvent être automatiquement incorporées aux agents.

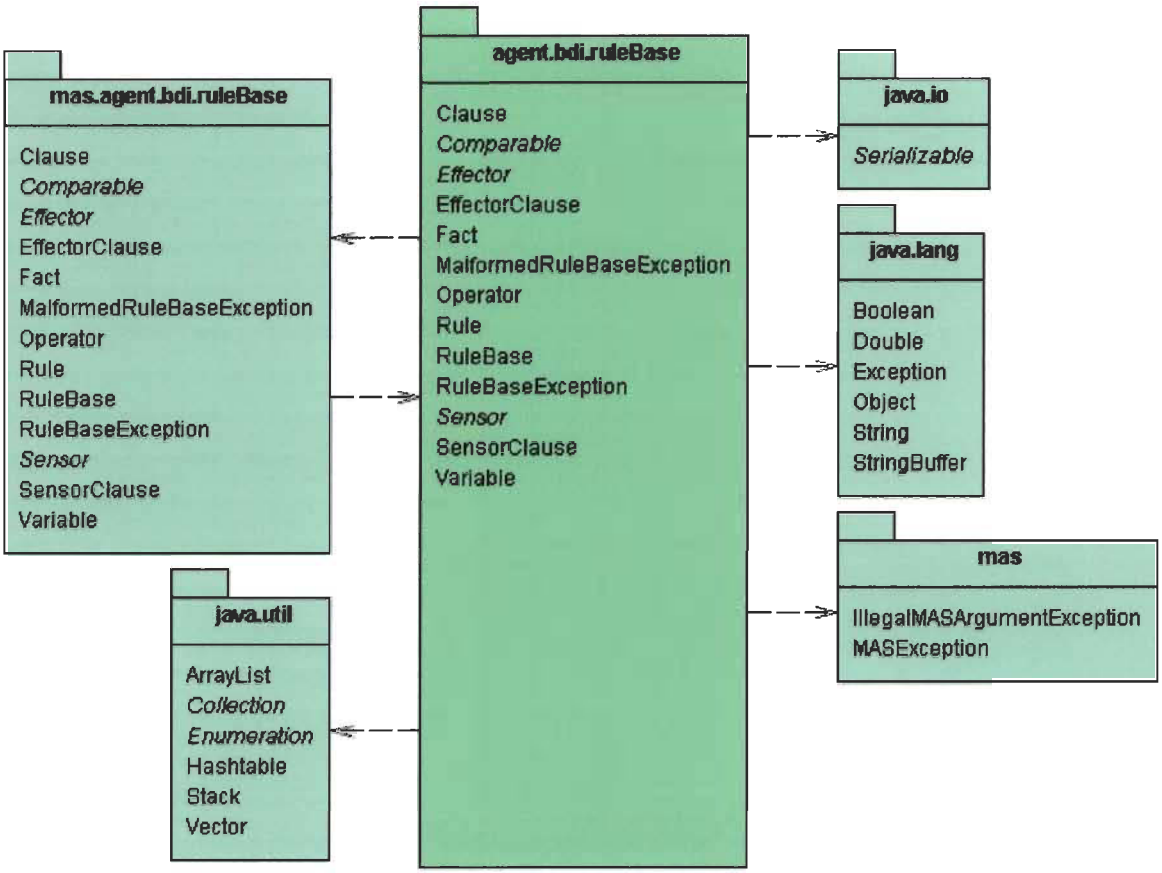

Figure 99 : Diagramme UML du package supportant les bases de connaissances

5.4.7 Le DF (Directory Facilitator)

La classe *DFAgent* et ses sous-classes procurent à une JVM plusieurs services intéressants : la possibilité de trouver un agent pour une tâche en particulier, envoyer les messages de tous les agents d'une JVM ou recevoir tous les messages provenant de l'extérieur vers une JVM en particulier pour éviter la surcharge de *Thread* nécessaires pour les communications inter-machines, etc.

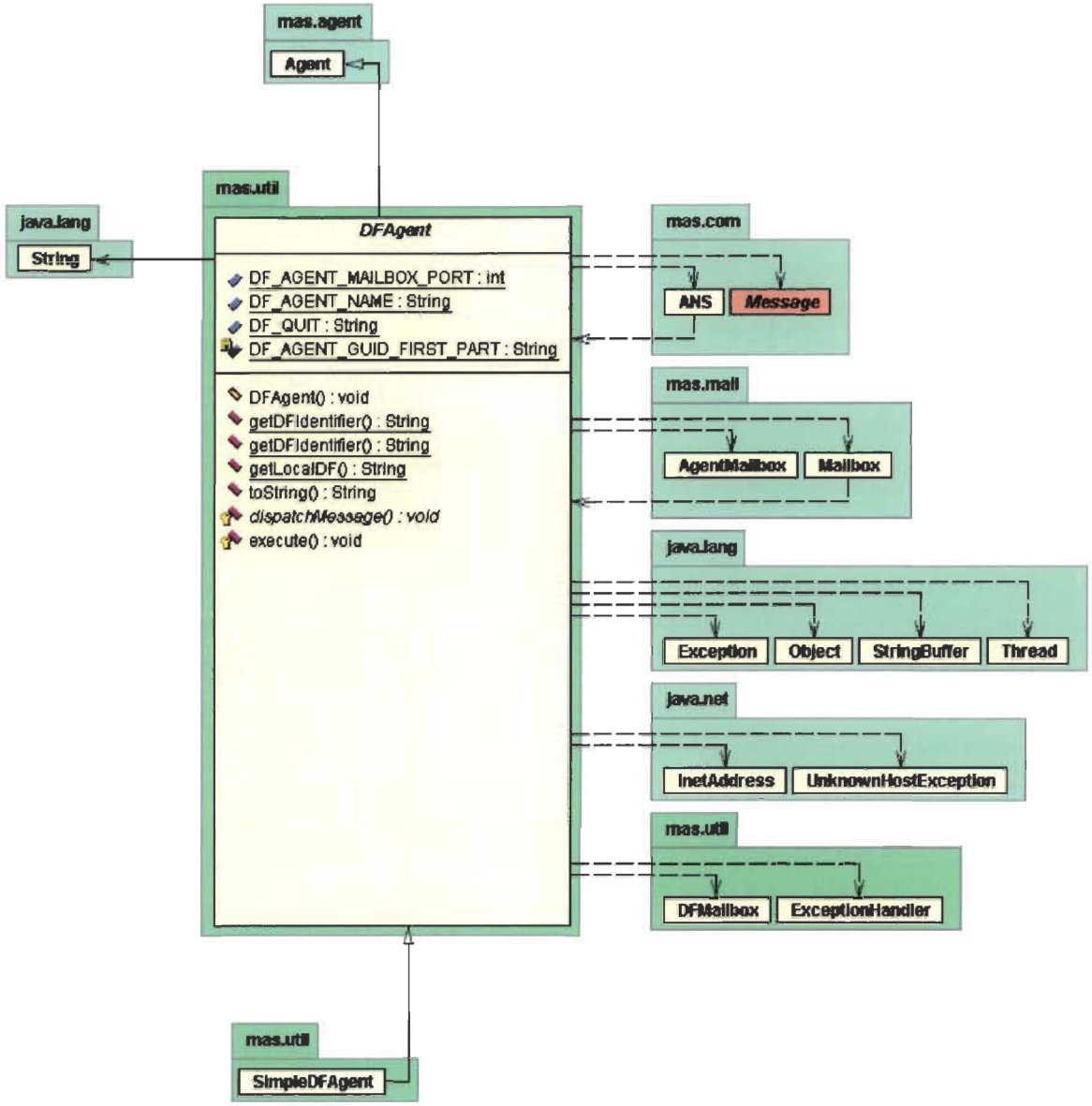

Figure 100 : Diagramme UML de la classe DFAgent (Directory Facilitator)

5.4.8 Les classes RegisterAgent et MulticastRegisterAgent

Ces deux classes sont des classes très utiles. En effet, elles permettent d'enregistrer automatiquement les agents des différents sous-systèmes auprès du ANS (*Agent Name Server*) des autres JVM. Avec la classe *MulticastRegisterAgent* il est possible d'exécuter les sous-systèmes sans spécifier les adresses des machines. Les agents de ces sous-systèmes s'enregistreront seuls (sans intervention du développeur) auprès des autres JVM.

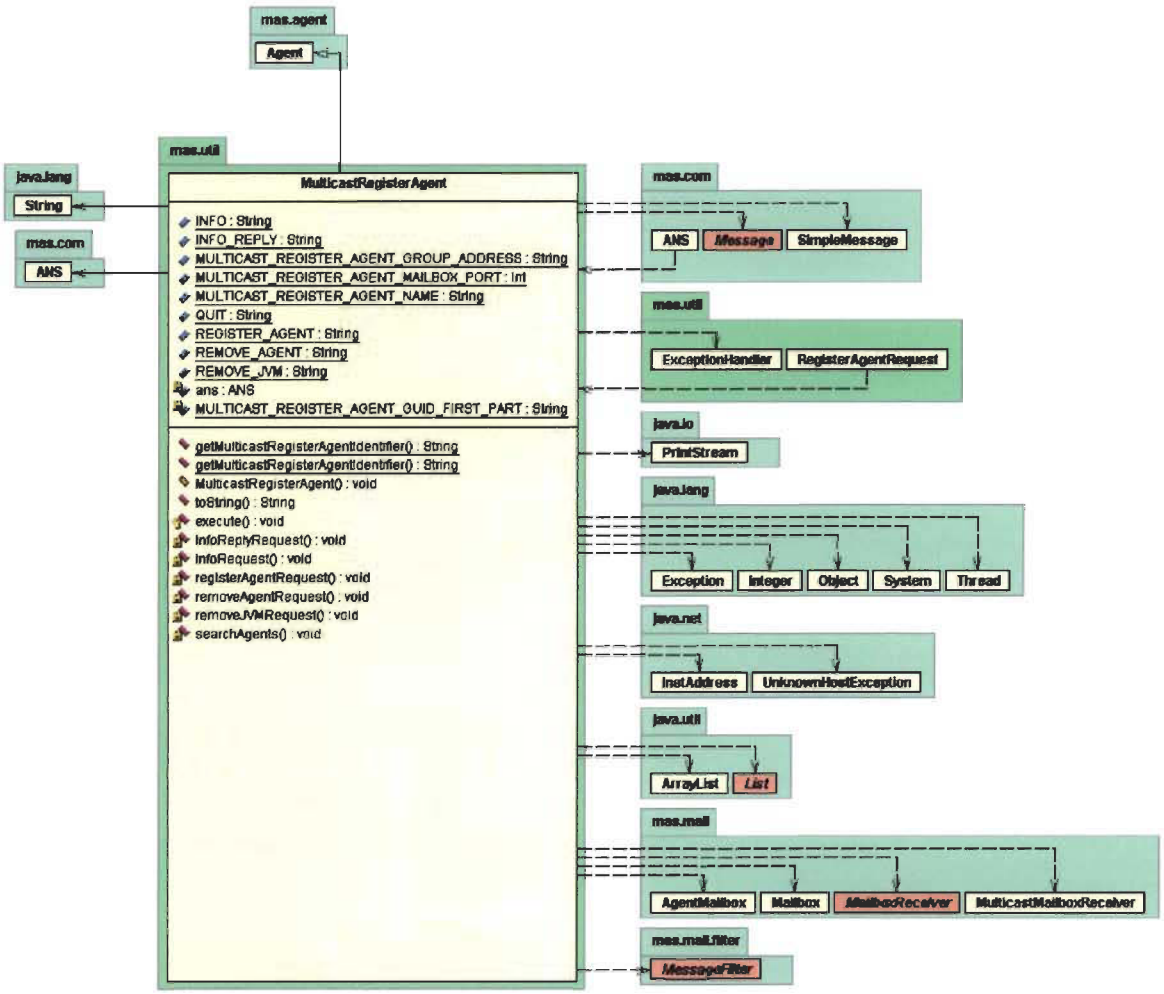

Figure 101 : Diagramme UML d'un RegisterAgent de type Multicast

5.4.9 L'interface Behaviour

L'interface *Behaviour* et ses sous-interfaces permettent aux agents d'adopter des comportements déjà implémentés à l'intérieur de la classe *Agent*.

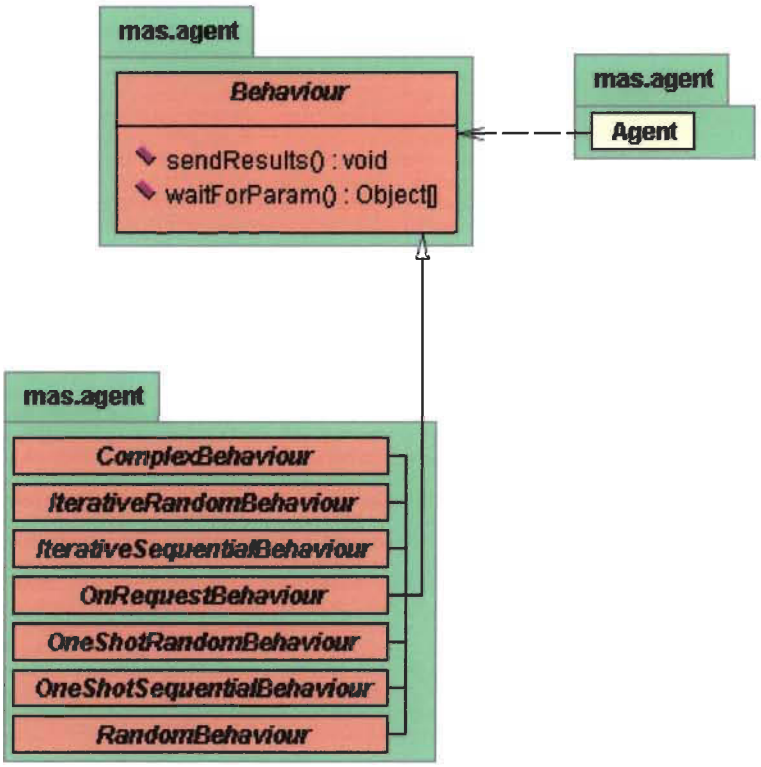

Figure 102 : Diagramme UML de l'interface Behaviour

5.4.10 Interaction avec la file d'événements AWT Java

Un utilitaire (composé de quelques classes) permet d'envoyer des messages dans la file des événements de Java (Figures 103 à 105). Ces messages peuvent être récupérés avec une gestion standard des événements AWT (en implémentant un « *listener* » conçu à cet effet) .

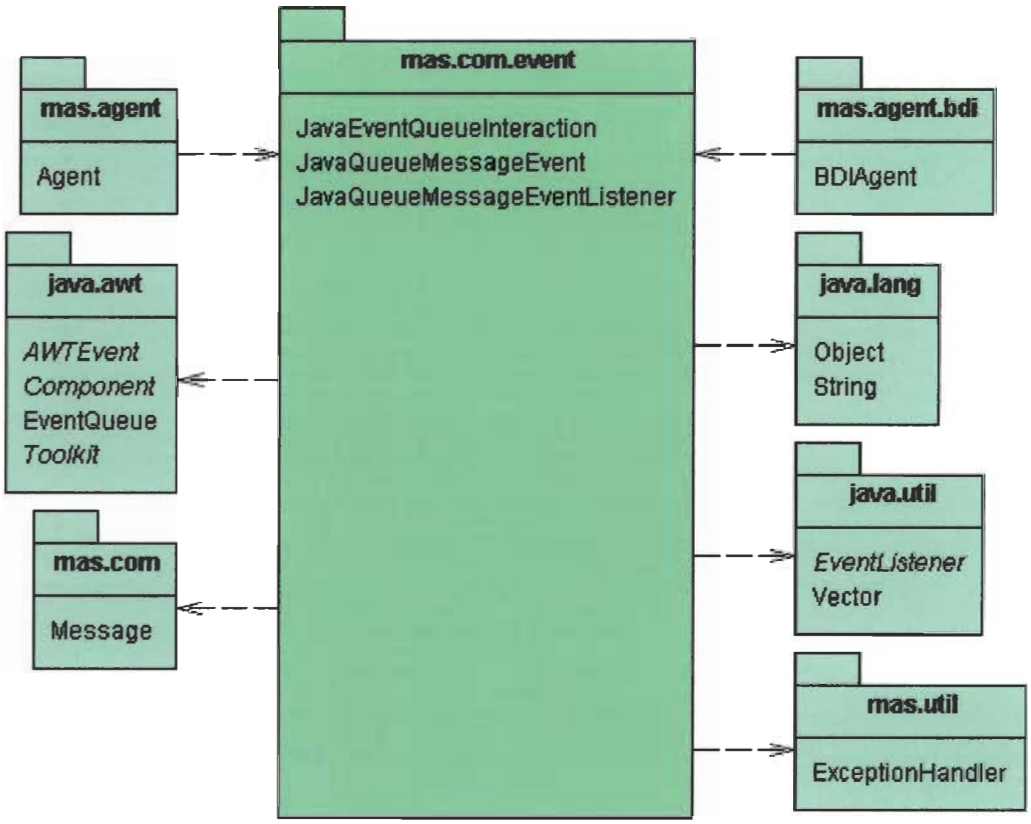

Figure 103 : Package de classes pour l'interaction avec la file d'événements AWT

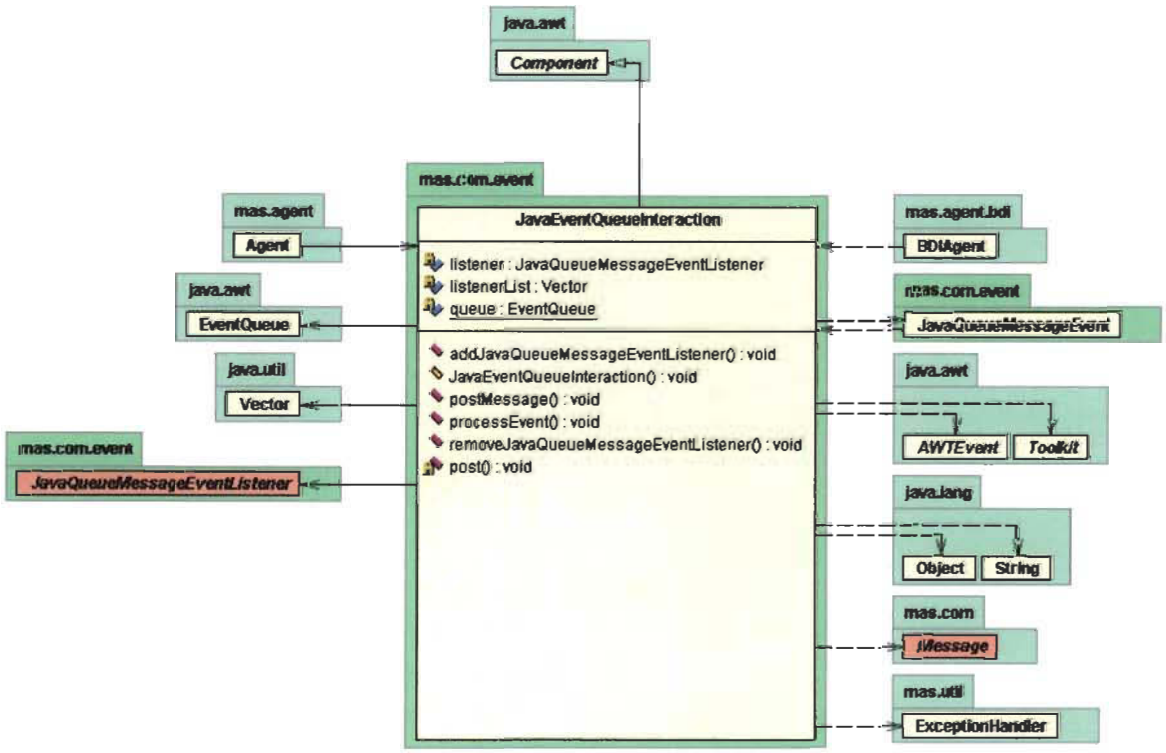

Figure 104 : Classe permettant l'interaction avec la file d'événements AWT

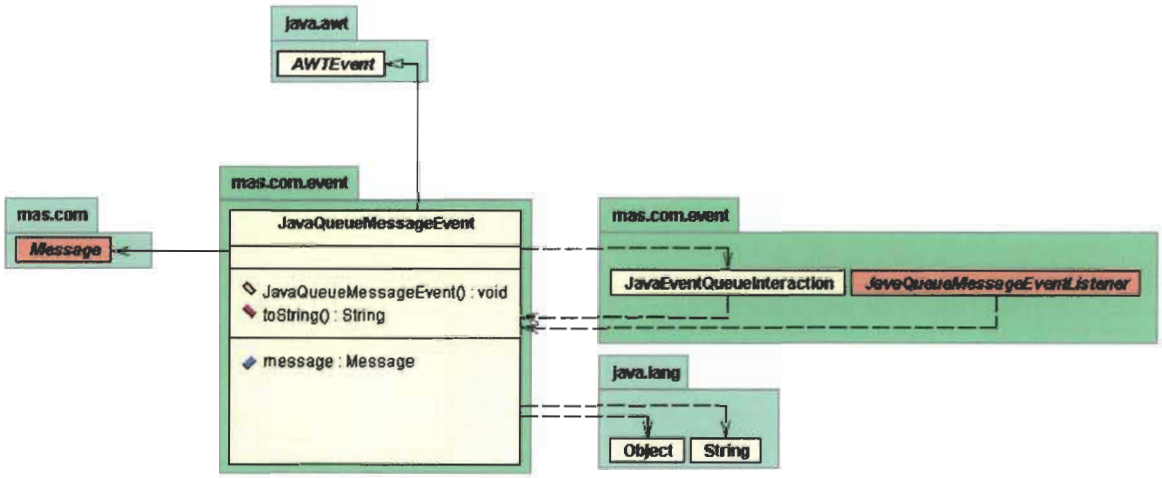

Figure 105 : Type d'événements envoyés à la file d'événements AWT

5.4.11 Autres classes et interfaces

La librairie de classe fournie plusieurs autres classes utilitaires visant à simplifier et accélérer le développement des systèmes. Par exemple, on y retrouve une classe *GenericMailbox* (sous-classe de la classe *Mailbox*) qui permet à une interface graphique ou autre composant Java de recevoir et d'envoyer des messages à un ou des agents (Figure 106).

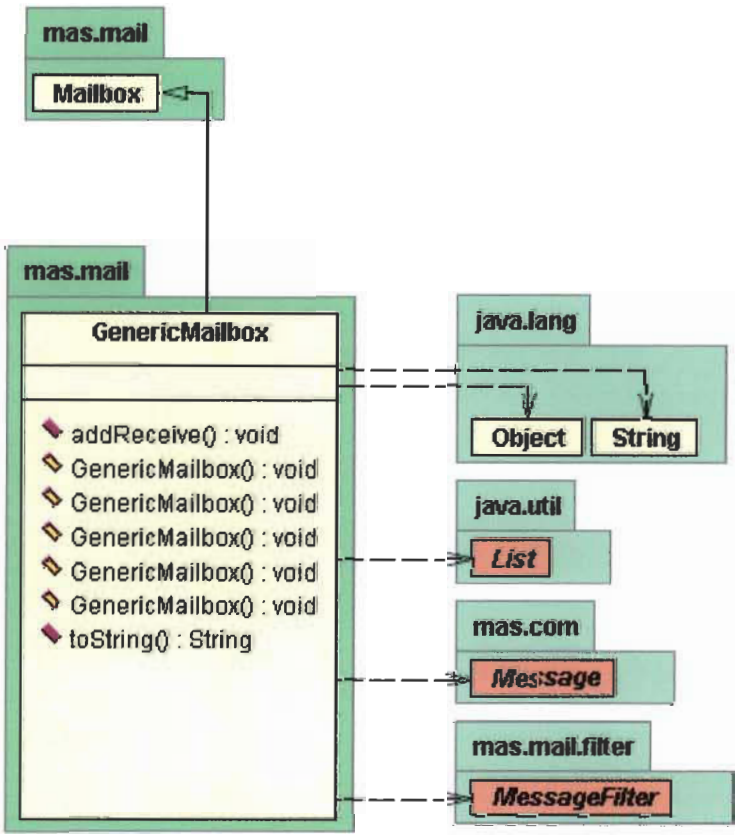

Figure 106 : Diagramme UML de la classe GenericMailbox

6. Exemples

Ce chapitre introduit deux exemples permettant d'illustrer l'utilisation de l'environnement. Pour une meilleure compréhension des étapes à suivre pour la création d'un SMA avec DMAS Builder, les exemples sont expliqués en détails. Ces exemples sont d'ailleurs les réponses aux deux mises en situation du chapitre 7.

6.1 Étapes générales du développement

Comme mentionné précédemment, il n'est pas nécessaire de suivre un ordre en particulier pour la création d'un SMA avec DMAS Builder. Cependant, une liste d'étapes « génériques » est proposée pour simplifier le développement et s'assurer qu'aucun oubli ne soit fait.

6.1.1 Étapes à suivre

La façon la plus simple de créer un SMA avec DMAS Builder est de suivre les étapes suivantes :

6.1.1.1 Étape 1 : Créer le dossier et le projet

La première étape est de créer un dossier qui contiendra le projet et tous les fichiers du SMA. Lors de la création d'un nouveau projet, il suffit de se placer à l'intérieur de ce dossier et de spécifier le nom du projet à créer. Le projet sera alors sauvegardé à l'intérieur de ce dossier et portera l'extension « **.sma** ». De plus, un dossier « **app** » sera créé à l'intérieur de ce dossier. Lors de la génération automatique, les fichiers sources seront générés à l'intérieur du dossier « **app** ».

6.1.1.2 Étape 2 : Spécifier les Adresses IP (si nécessaire)

S'il y a plus d'une JVM et que les adresses IP sont connues à l'avance, il est important de les spécifier à l'intérieur de l'onglet « Adresses IP ». Celles-ci seront associées avec leur JVM respective dans une étape subséquente.

6.1.1.3 Étape 3 : Créer les sous-systèmes (JVM)

La création des sous-systèmes est simple. Elle consiste à spécifier un nom pour chaque sous-système et de les ajouter au SMA. Ceci est fait par l'onglet « Sous-systèmes ».

6.1.1.4 Étape 4 : Associer les JVM aux adresses IP (si nécessaire)

Une fois les JVM créées, si des adresses IP ont été spécifiées à l'étape 2, il faut associer les JVM avec les adresses. Ceci est fait par la partie de l'interface de l'éditeur de

systèmes se trouvant en bas à gauche. Il suffit de sélectionner chaque JVM (dans le premier menu déroulant) et choisir l'adresse IP à laquelle elle est associée (dans le quatrième menu déroulant).

6.1.1.5 Étape 5 : Déterminer le type de DF et de RA pour chaque JVM (si nécessaire)

Si les JVM doivent posséder un type de *Directory Facilitator*, il faut les spécifier via le deuxième menu déroulant (encore en bas à gauche de l'éditeur de systèmes). Pour chaque JVM, il faut lui déterminer un type de DF en particulier et lui associer. La même opération doit être faite pour le type de *Register Agent*. S'il y a plus d'une JVM et que les adresses IP ont été spécifiées, un « *Socket RA* » doit être spécifié pour chaque JVM. S'il y a plusieurs JVM mais que les adresses IP sont variables ou inconnues, il est possible de spécifier un « *Multicast RA* » permettant l'enregistrement automatique des JVM.

6.1.1.6 Étape 6 : Créer les types d'agent

Il faut créer un type d'agent pour chaque agent ayant au moins une propriété différente des autres agents (onglet Types d'agent). En général, il est préférable de créer un type d'agent pour chaque agent. Celui-ci sert de couche intermédiaire entre la superclasse *Agent* et l'agent en particulier. Cette couche est utile pour traiter l'envoi de paramètres aux tâches et la réception des résultats de l'exécution de celles-ci.

6.1.1.7 Étape 7 : Créer les tâches

Cette étape consiste à créer toutes les tâches qui pourront être effectuées par un ou des agents du système. Pour créer les tâches, il suffit de spécifier le nom de chaque tâche (onglet Tâches). Celles-ci seront associées aux agents un peu plus loin.

6.1.1.8 Étape 8 : Créer les bases de connaissances des agents (si nécessaire)

L'étape de la création des bases de connaissances se fait à l'intérieur de l'onglet « BDI » et de ses trois éditeurs : Variables, Clauses et Règles.

6.1.1.9 Étape 9 : Créer les agents et spécifier leurs attributs

La création des agents se fait au bas à gauche de l'éditeur de systèmes. Il faut donner un nom à chaque agent et spécifier le type de chacun. Par la suite, il faut déterminer les valeurs des attributs de chacun dans la table des agents.

6.1.1.10 Étape 10 : Associer les bases de connaissances aux agents (si nécessaire)

Si un ou des agents possèdent une base de connaissances, il faut associer chaque base de connaissances aux agents correspondants. Ceci est fait par l'attribut « Base de connaissances » de la table des agents.

6.1.1.11 Étape 11 : Associer les tâches aux agents

Cette étape consiste à associer les tâches aux agents qui pourront les effectuer. Pour faire cette opération, il suffit de sélectionner les tâches (dans la table des tâches) pouvant être effectuées par chaque agent de chaque JVM.

6.1.1.12 Étape 12 : Valider le système

Une fois toutes les spécifications terminées, il est important de valider le système. La validation permet de vérifier plusieurs aspects du SMA. Il suffit de choisir l'option « Valider » de la barre d'outils.

6.1.1.13 Étape 13 : Générer le système

Lorsque la validation est complétée et qu'il n'y a plus aucun message d'erreur, les fichiers sources du système peuvent être générés. Il faut choisir l'option « générer » de la barre d'outils.

6.1.1.14 Étape 14 : Terminer l'implémentation

Lorsque la génération du code source du SMA est complétée, il faut terminer l'implémentation (les tâches, les méthodes des agents pour les paramètres de tâches, les senseurs et effecteurs des bases de connaissances, etc.). La plupart des endroits stratégiques où il faut ajouter du code sont spécifiés par le commentaire suivant : « *to do : code goes here* ».

6.1.1.15 Étape 15 : Compiler et exécuter le système

Il reste seulement à compiler et exécuter le projet. Pour l'exécution, la classe contenant la méthode « *main* » doit être spécifiée dans l'éditeur « Options du projet ». Évidemment, on ne peut exécuter qu'un sous-système sur une machine.

6.1.1.16 Étape 16 : Archiver le système

Cette option peut être utilisée lorsque l'on veut créer des archives compressées exécutables pour chaque JVM.

6.2 Le fameux « Hello World »

Le fameux programme « *Hello World* » prend tout son sens ici car il permet de se familiariser avec l'environnement de développement. Le but de ce premier programme est de créer un projet contenant deux agents sur la même JVM. Un des deux agents demande le nom de l'utilisateur à l'écran (ici la demande du nom peut se faire en mode console ou par une fenêtre d'entrée de texte). Par la suite, une fois le nom entré, il l'envoie à l'autre agent qui dit « salut… » suivi du nom entré à l'écran.

Ce petit programme semble anodin. Cependant, il est beaucoup plus complexe qu'un simple programme qui affiche une chaîne de caractères à l'écran. Ici, il est nécessaire d'utiliser les *Mailbox* des agents. De plus, il faut créer un sous-système, une tâche pour chaque opération, spécifier des comportements aux agents, générer le code et terminer l'implémentation dans le code source.

La façon la plus simple pour concevoir un système avec DMAS Builder est de suivre les étapes énumérées dans la section précédente (section 6.1). Cette technique semble complexe. Cependant, elle est très simple et rapide. Voyons maintenant en détails comment faire l'implémentation de ce petit système. Il est à noter que certaines étapes ne sont pas toujours nécessaires, c'est pourquoi certaines sont excluses de l'exemple.

6.2.1 Créer un dossier et un projet (Étape 1)

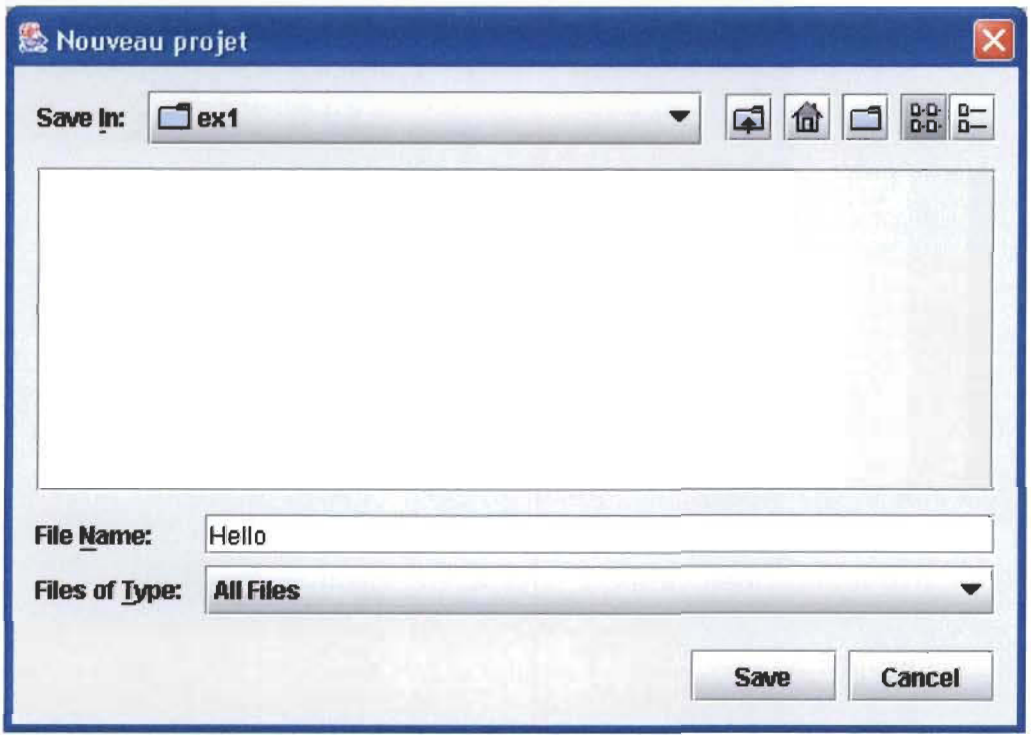

Figure 107 : Fenêtre pour la création du projet (étape 1)

La première étape est d'aller dans ***Fichier → Nouveau*** projet et de créer un répertoire (si cela n'est pas déjà fait) qui contiendra le projet. Ici, le répertoire **ex1** contiendra tout le projet. Par la suite, toujours dans cette fenêtre, il faut spécifier le nom du projet (dans cet exercice, le projet s'appelle ***Hello***).

6.2.2 Création de la JVM (Étape 3)

Toutes les spécifications du système se font à l'intérieur de l'éditeur de systèmes. Il faut donc éditer cette interface : ***Projet → Éditeur de systèmes***. Dans cette interface, il faut se placer dans l'onglet Sous-systèmes (Figure 108) et ajouter une JVM (lui donner un nom et appuyer sur *Enter*). Dans cet exemple, la JVM (sous-système) s'appelle (***JVM_1***).

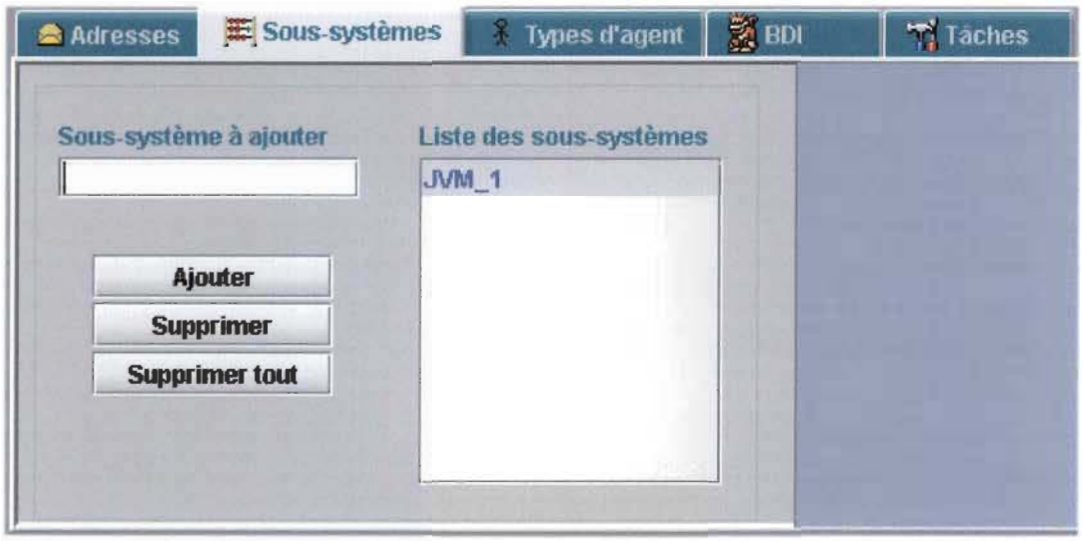

Figure 108 : Onglet pour la création du sous-système (étape 3)

6.2.3 Création des types d'agents (Étape 6)

Dans cette étape, il faut créer deux types d'agents car il faut donner deux types de comportement différents.

6.2.3.1 Premier type d'agent

Le premier type est celui qui permettra à l'agent de demander le nom de l'utilisateur et de l'envoyer au deuxième agent. Par la suite, ce premier agent termine immédiatement son exécution. Pour spécifier un type d'agent, il faut se placer dans l'onglet Types d'agent (Figures 109 et 110).

Pour créer un type d'agent, il faut spécifier le nom du type, le type (BDI ou autres), le mode de communication et le type de comportement. Pour cet exemple, le nom du type est « *AskType* »; l'agent n'a pas de base de connaissances (il est donc de type « Autre »). Pour ce cas-ci, le mode de communication est *DirectMailbox* car les deux agents sont sur la même JVM. Comme ce premier agent demande le nom de l'utilisateur, l'envoie et termine son exécution, il faut utiliser un comportement approprié pour son type. Le comportement « *OneShotSequentialBehaviour* » est un comportement qui, lorsque

associé à un type d'agent, spécifie qu'il exécutera sa liste de tâches une seule fois (une après l'autre). Par la suite, il termine son exécution. C'est exactement le type de comportement nécessaire pour le premier agent (il exécutera sa liste de tâches qui en contiendra qu'une seule et terminera son exécution).

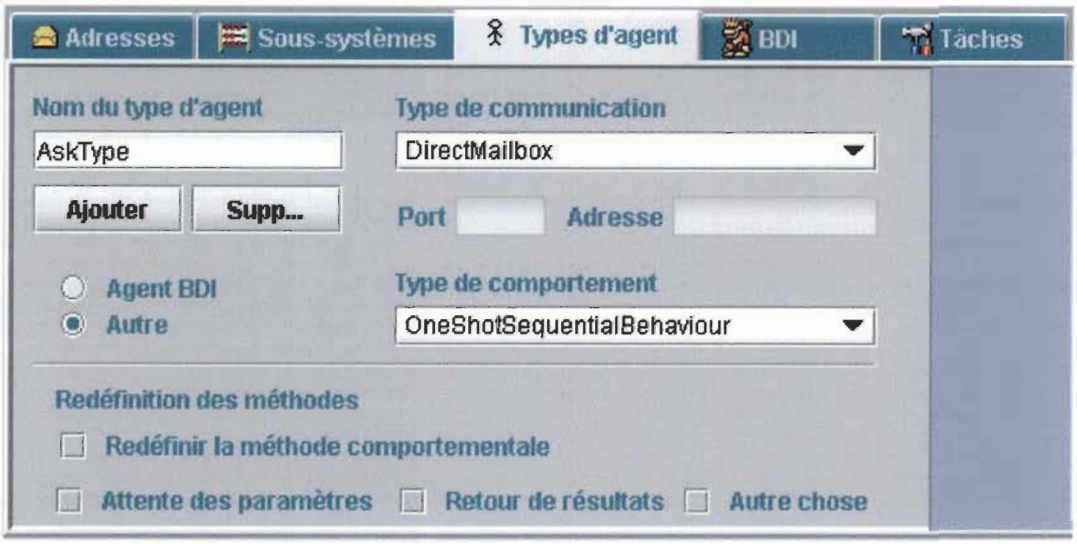

Figure 109 : Onglet pour la création du type d'agent de l'agent demandeur (étape 6)

6.2.3.2 Deuxième type d'agent

Le deuxième type est celui qui servira pour l'agent qui ne fait qu'afficher le nom de l'utilisateur et lui dit « salut ». Par la suite, cet agent termine immédiatement son exécution. Pour spécifier un type d'agent, il faut encore se placer dans l'onglet Types d'agent.

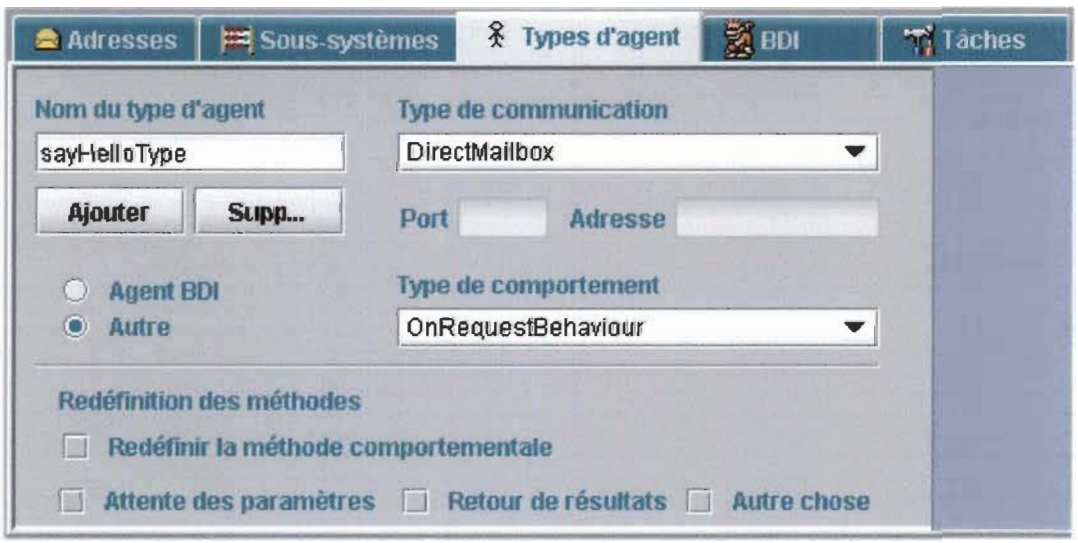

Figure 110 : Onglet pour la création du type d'agent de l'agent répondeur (étape 6)

La création de ce type d'agent est semblable au premier sauf pour la spécification du comportement. Le comportement du premier type n'est pas approprié pour le deuxième agent car il ne faut pas qu'un agent de ce type exécute la tâche tant qu'il ne reçoit pas de message. Pour le second agent, le bon type de comportement est le « *OnRequestBehaviour* ». Ce type de comportement spécifie que les agents de ce type exécute des tâches sur demande. L'agent reste en attente de messages et lorsqu'il reçoit un message, il vérifie si le sujet de ce message correspond au nom d'une de ses tâches. Si oui, il exécute cette tâche. Sinon, il retourne en attente.

6.2.4 Création des tâches (Étape 7)

Figure 111 : Onglet pour la création des tâches (étape 7)

Pour la spécification des tâches, il faut se placer dans l'onglet Tâches (Figure 111).

La création des tâches est très simple. Il suffit de spécifier le nom de la tâche qui pourra être effectuée par un ou des agents. Une description de chaque tâche est optionnelle. Le premier agent aura besoin d'effectuer une seule tâche qui sera de demander le nom de l'utilisateur et d'envoyer un message à l'autre agent « *AskName* ». Le deuxième agent n'aura aussi qu'une seule tâche à effectuer : recevoir le message et afficher un message « *SayHello* ».

6.2.5 Créer les agents et spécifier leurs attributs (Étape 9)

La création des agents se fait dans le bas à gauche dans l'éditeur de systèmes. Il suffit de spécifier le type de l'agent à créer et de donner un nom à l'agent (Figures 112 et 113).

Figure 112 : Création des agents (Étape 9) (agent demandeur)

Figure 113 : Création des agents (Étape 9) (agent afficheur)

Une fois les agents créés, il se retrouvent dans la table des agents. Il faut maintenant personnaliser les attributs des deux agents (Figure 114).

Nom de l'agent	Idendificateur unique	Temps d'attente
AskAgent	Automatiquement	100
HelloAgent	hello	100

Figure 114 : Table des attributs des agents (Étape 9) (partie 1)

Dans ce cas-ci, la majorité des attributs ne nécessitent aucune modification. Cependant, il est plus simple de changer l'identificateur de l'agent « *HelloAgent* ». En effet, l'agent « *AskAgent* » devra connaître cet identificateur pour envoyer un message au deuxième agent « *HelloAgent* ». De plus, le temps d'attente des agents était initialisé à « Maximum ». Cela signifiait que les agents avaient un temps d'attente immense entre l'exécution de leurs tâches et ainsi le programme ne se terminerait pas immédiatement (100 millisecondes).

6.2.6 Association des tâches aux agents (Étape 11)

Il faut maintenant associer chaque tâche aux agents (Figure 115). Cette opération est très simple. Il suffit de sélectionner les tâches à effectuer pour chaque agent (dans la liste des tâches).

Nom de l'agent		AskName	SayHello
AskAgent	Au	☑	☐
HelloAgent	he	☐	☑

Figure 115 : Table des agents et table des tâches (Étape 11)

Ici, il faut associer la tâche « *AskName* » avec l'agent « *AskAgent* » et la tâche « *SayHello* » avec l'agent « *HelloAgent* ».

6.2.7 Valider le système (Étape 12)

La validation du système permet de vérifier plusieurs caractéristiques. Entre autre, elle permet de vérifier si tous les éléments d'un système sont présents dans l'éditeur et que les noms utilisés sont conformes pour la compilation Java. Pour valider le système, il suffit de cliquer sur « Valider » dans la barre d'outils.

Figure 116 : Options principales de la barre d'outils

Lors de la validation, une fenêtre de messages apparaîtra à l'écran. Si aucune erreur n'a été commise, la fenêtre présentée à la figure 117 sera à l'écran.

Figure 117 : Fenêtre de validation du système

6.2.8 Génération du système (Étape 13)

La génération du système permet de générer automatiquement le code source de l'application. Pour générer les fichiers sources de l'application, il suffit de cliquer sur Générer dans la barre d'outils.

Figure 118 : Options principales de la barre d'outils

Lors de la génération, une fenêtre de messages apparaîtra à l'écran. Si aucune erreur n'était présente au niveau de la validation, la fenêtre de la figure 119 sera à l'écran.

Figure 119 : Fenêtre de messages de génération automatique

Une fois la génération du code terminée, les fichiers auront été ajouté à la liste des fichiers du projet dans l'éditeur de fichiers : ***Projet → Éditeur de fichiers*** (Figure 120).

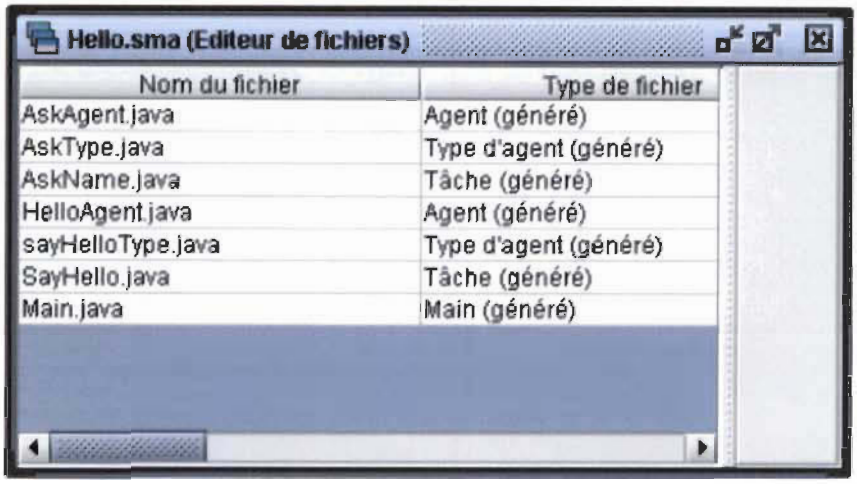

Figure 120 : Liste des fichiers générés dans l'éditeur de fichiers

6.2.9 Terminer l'implémentation (Étape 14)

6.2.9.1 Implémentation de la tâche « *AskName* »

Il faut importer le *package java.io* dans cette classe. Le seul endroit où il faut ajouter du code à l'intérieur de cette tâche est dans la méthode « *execute()* » (Figure 121).

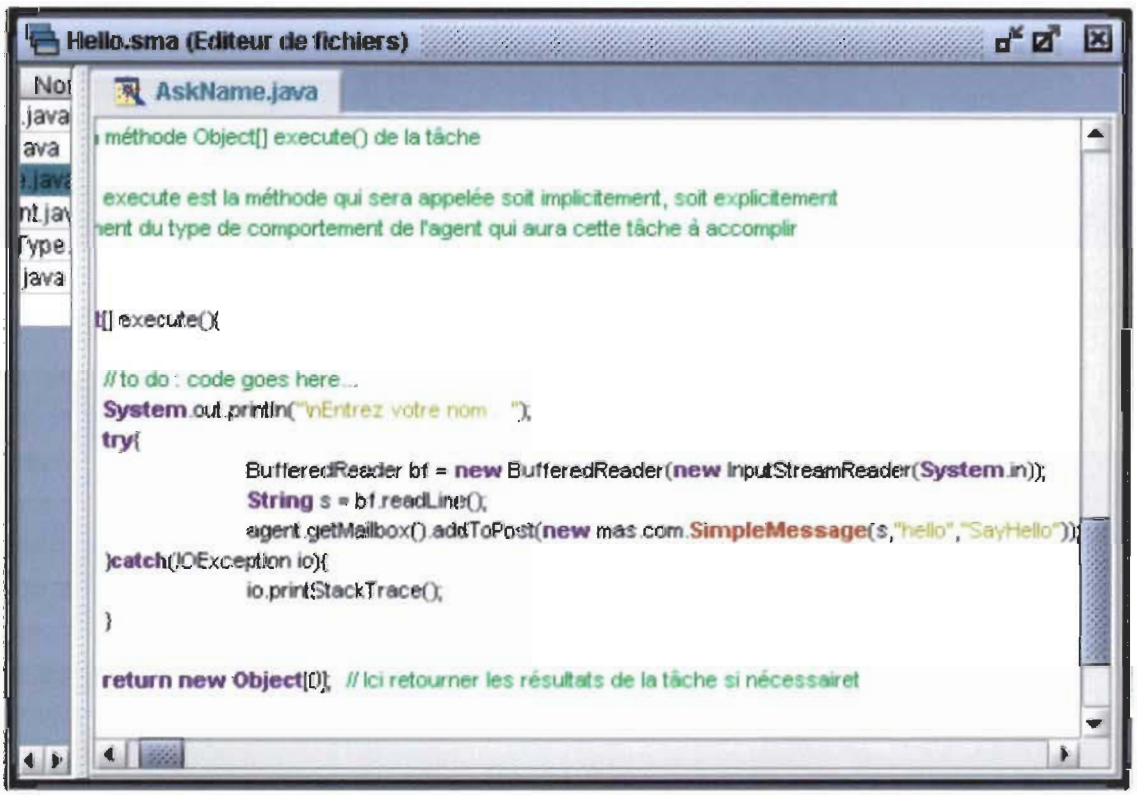

Figure 121 : Éditeur de fichiers (Méthode « execute() » de la tâche «AskName »)

Code de base ajouté (Java standard)

- Afficher (*System.out*).
- Instanciation d'un objet pour la lecture à la console (*BufferedReader*).
- Lecture à la console (*bf.readLine()*).

Code spécifique (API OA)

- ***agent.getMailbox().addToPost(new mas.com.SimpleMessage (s,"hello","SayHello"));***

Note : La ligne précédente est équivalente à :

```
mas.mail.AgentMailbox mail = agent.getMailbox() ;
mas.com.SimpleMessage mess = new
mas.com.SimpleMessage(s,"hello","SayHello") ;
mail.addToPost(mess);
```

- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).
- ***getMailbox() :*** Retourne la *Mailbox* de l'agent.
- ***addToPost(Message m) :*** Ajoute un message dans la liste d'envoi de la *Mailbox* de l'agent.
- ***new mas.com.SimpleMessage(Object o, String guid, String subject) :*** Est un message à envoyer prenant comme paramètres n'importe quel objet (le message à envoyer), l'identificateur de l'agent auquel on envoie le message et le sujet. Dans cet exemple, étant donné que l'agent « *Hello* » est un agent qui a un comportement « *OnRequestBehaviour* », il effectuera une tâche que si le sujet du message correspond au nom d'une de ses tâches. Note : Le sujet ne sert d'appel de tâches que pour les comportements « *OnRequestBehaviour* ». Dans les autres cas, le sujet est libre d'utilisation pour le concepteur.
- ***return new Object[0]; :*** S'il y a des résultats à retourner (qui seront « attrapés » par la redéfinition de la méthode d'attente de résultats du type de l'agent), c'est ici qu'il faut les retourner. Sinon, il faut retourner un tableau vide d'objets.

6.2.9.2 Implémentation de la tâche « *SayHello* »

Une seule ligne est à ajouter dans la méthode « *execute()* » (Figure 122).

Code spécifique (API OA)

- ***(String)agent.getCurrentMessage().getMessage()***

Note : La ligne précédente est équivalente à :

```
Message m = agent.getCurrentMessage() ;
String s = (String)m.getMessage() ;
```

- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).

- ***getCurrentMessage() :*** Est les messages présentement traités (cet objet aussi est implicite à l'agent lorsqu'il a un comportement « *OnRequestBehaviour* ») car s'il exécute une tâche, c'est parce qu'il vient de recevoir une requête avec un sujet qui correspondait au nom d'une de ses tâches. Dans ce cas, une référence au message courant est gardée implicitement avec l'agent et est accessible via cette méthode. De plus, la méthode « *getCurrentMessage()* » ne sert pour le moment que pour les agents qui ont un comportement « *OnRequestBehaviour* ».
- ***(String)getMessage() :*** Méthode qui retourne le contenu du message, qui était le premier paramètre du message envoyé par « *AskAgent* » dans sa tâche « *AskName* » (peut être n'importe quel type d'objet). Par contre, dans ce cas-ci que c'est une *String*).
- ***return new Object[0]; :*** S'il y a des résultats à retourner (qui seront « attrapés » par la redéfinition de la méthode d'attente de résultats du type de l'agent), c'est ici qu'il faut les retourner. Sinon, il faut retourner un tableau vide d'objets.

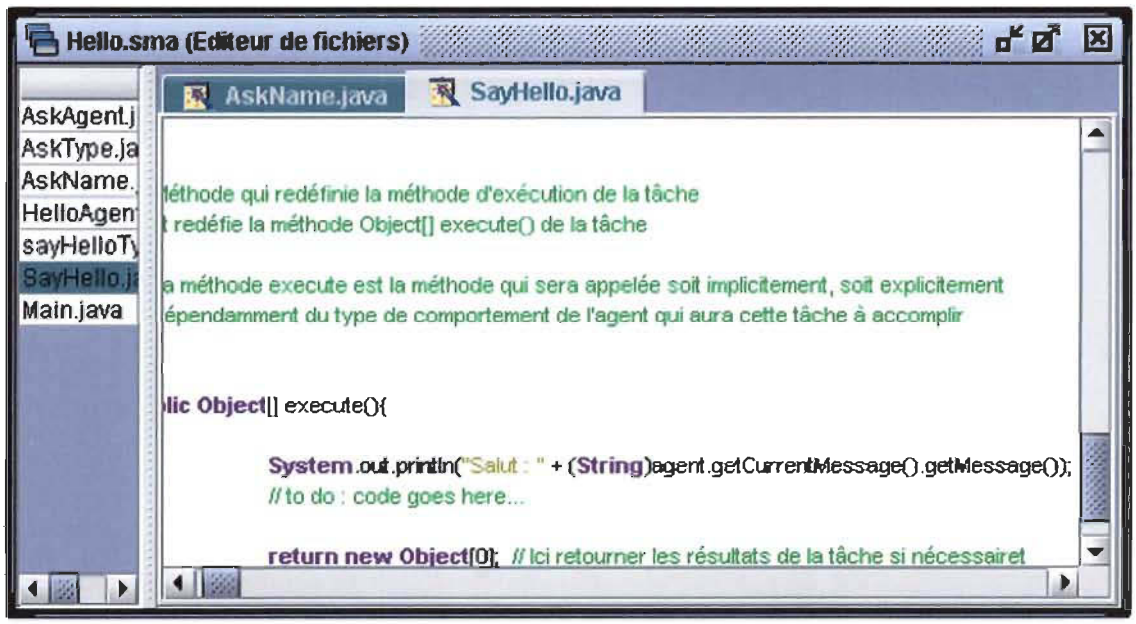

Figure 122 : Éditeur de fichiers (Méthode « execute() » de la tâche «SayHello»)

6.2.10 Compiler et exécuter le projet (Étape 15)

6.2.10.1 Compilation du projet

Lorsque l'implémentation est terminée, il ne reste qu'à compiler le projet : ***Projet → Compiler le projet***. Une fenêtre de messages apparaîtra et donnera les messages d'usages. Si aucune compilation ne se produit, c'est probablement dû au fait que le chemin du programme de compilation n'est pas initialisé : ***Options → Options du compilateur*** (Chemin *javac*).

6.2.10.2 Exécution du projet

L'exécution se fait en choisissant ***Projet → Exécuter le projet***. Si une erreur « *NoClassDefFoundError* » est levée, c'est probablement parce que la classe à exécuter n'est pas spécifiée. Pour spécifier la classe à exécuter, il est nécessaire d'aller dans ***Options → Options du projet*** (Classe à exécuter) et spécifier le nom de la classe (avec le nom de *package*). Pour ce projet, la classe à spécifier est : ***JVM_1.Main***.

6.3 La calculatrice distribuée

Le deuxième programme est une petite calculatrice distribuée permettant d'effecteur des opérations mathématiques de base. Le programme doit pouvoir additionner, soustraire, multiplier et diviser deux nombres indéfiniment grands. De plus, toutes les opérations mathématiques sont écrites dans un fichier sous la forme :

N1 op1 n2 = res1
N3 op2 n4 = res2
N5 op3 n6 = res3

Le programme fonctionne de la façon suivante :

- Le programme demande à l'utilisateur d'entrer un calcul.
- L'utilisateur entre le calcul (ex.: 44 + 66).
- Le programme donne le résultat à l'écran et sauvegarde le résultat dans le fichier.
- À nouveau, le programme demande d'entrer un calcul…

Contraintes :

Le programme doit posséder 6 agents :

- Un agent qui demande le calcul à effectuer et délègue l'opération au bon agent.
- Un agent qui effectue l'addition (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la multiplication (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la soustraction (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la division (et envoie le calcul à l'agent de sauvegarde).
- Un agent de sauvegarde qui enregistre les calculs dans le fichier et affiche le résultat à l'écran.

6.3.1 Créer un dossier et un projet (Étape 1)

La première étape est d'aller dans ***Fichier → Nouveau projet*** et de créer un répertoire (si cela n'est pas déjà fait) qui contiendra le projet. Pour cet exemple, le répertoire ***ex2***

contiendra tout le projet. Par la suite, toujours dans cette fenêtre, il faut spécifier le nom du projet (ici, le projet s'appelle Calcul).

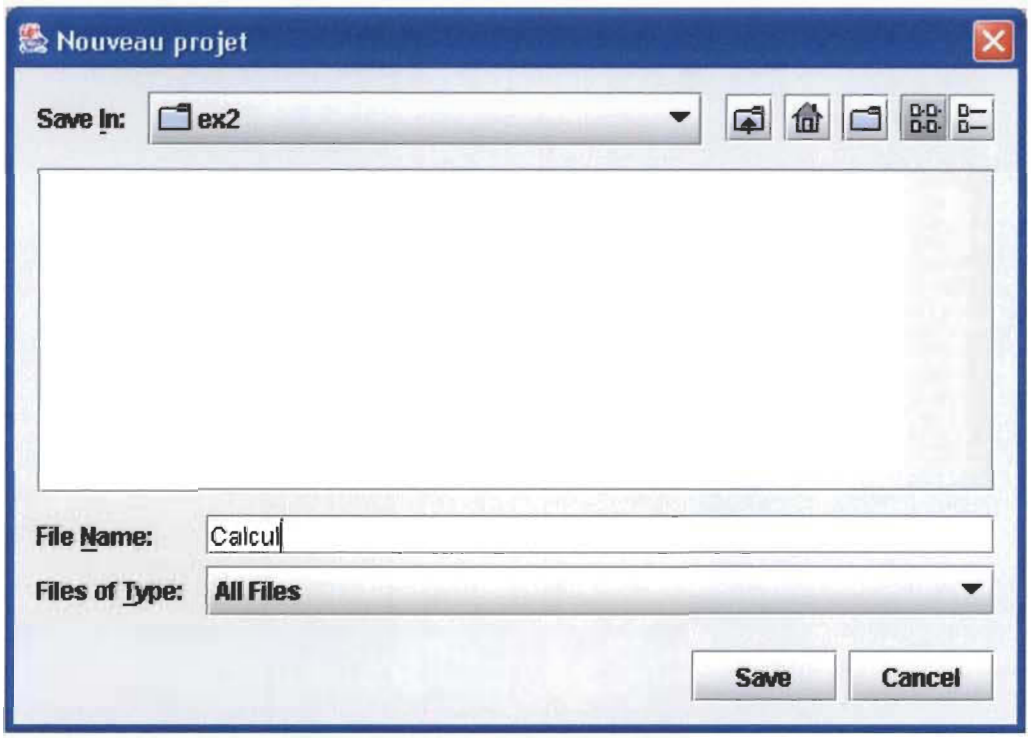

Figure 123 : Fenêtre pour la création du projet (étape 1)

6.3.2 Création de la JVM (Étape 3)

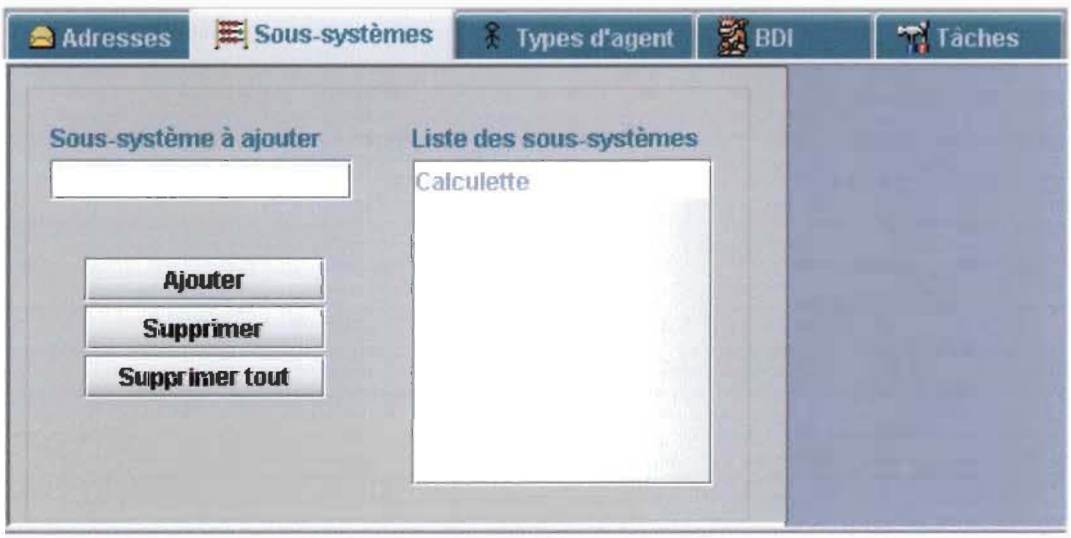

Figure 124 : Onglet pour la création du sous-système (étape 3)

Toutes les spécifications du système se font à l'intérieur de l'éditeur de systèmes. Il faut donc éditer cette interface : ***Projet → Éditeur de systèmes***. Dans cette interface, il faut se placer dans l'onglet Sous-systèmes (Figure 124) et ajouter une JVM (lui donner un nom et appuyer sur *Enter*). Dans cet exemple, le sous-système se nomme (Calculette).

6.3.3 Création des types d'agents (Étape 6)

Dans cette étape, il faut créer trois types d'agents.

6.3.3.1 Premier type d'agent

Le premier type est celui qui servira pour l'agent qui demande à l'utilisateur un calcul et l'envoie à l'agent qui doit effectuer l'opération (un des quatre agents, dépendamment de l'opération demandée). Une fois l'envoi effectué, il recommence les mêmes opérations. Le type d'agent se spécifie via l'onglet Types d'agent (Figures 125 à 127).

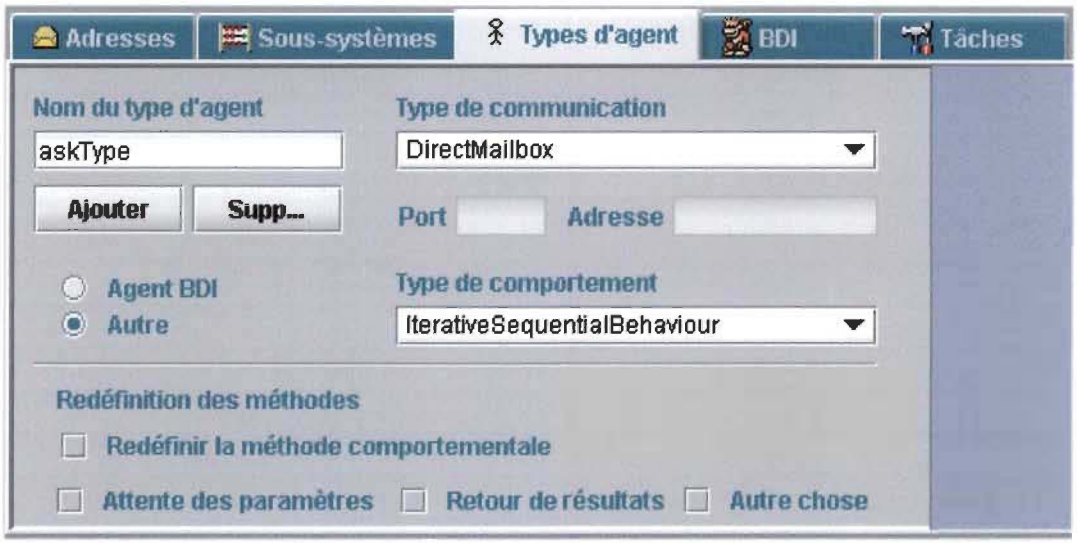

Figure 125 : Onglet pour la création du type d'agent de l'agent « demandeur »

Pour créer un type d'agent, il faut spécifier le nom du type, le type (BDI ou Autre), le mode de communication et le type de comportement. Pour cet exemple, le nom du type est « *AskType* »; l'agent n'a pas de base de connaissances (il est donc « Autre »). Dans cet exemple, le mode de communication est *DirectMailbox* car tous les agents sont sur la même JVM. Étant donné que l'agent demande à l'utilisateur d'entrer un calcul, détermine à quel agent l'envoyer, envoie le message et recommence les mêmes opérations sans arrêt, il faut utiliser un comportement approprié pour son type. Le comportement « *IterativeSequentialBehaviour* » est un comportement qui, lorsque associé à un type d'agent, spécifie qu'un agent de ce type exécutera sa liste de tâches (une après l'autre) de façon itérative (l'agent recommence l'exécution de la liste une fois l'exécution de celle-ci terminée). C'est exactement le type de comportement nécessaire pour l'agent demandeur

de calcul (il exécutera sa liste de tâches qui ne contiendra qu'une seule tâche) et recommencera l'exécution de celle-ci indéfiniment.

6.3.3.2 Deuxième type d'agent

Le deuxième type est celui qui servira pour les quatre agents qui exécutent les calculs. Les quatre agents ayant des caractéristiques semblables et surtout le même comportement (faire un calcul sur demande). Il n'est donc pas utile de leur spécifier des types différents.

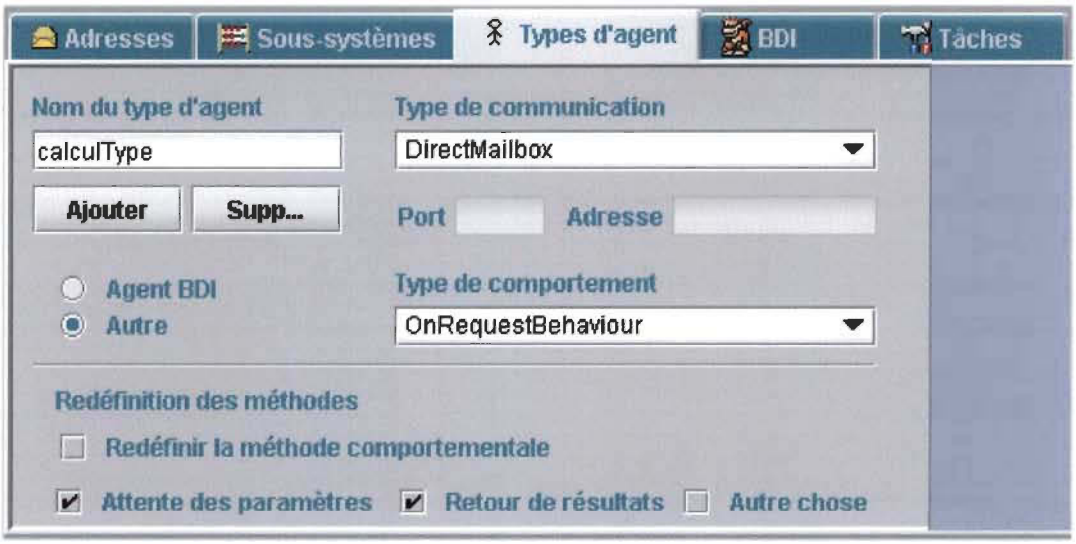

Figure 126 : Onglet pour la création du type d'agent de calcul (étape 6)

La création de ce type d'agent est semblable au premier type sauf pour la spécification du comportement. Le comportement du premier type n'est pas approprié pour le type des agents de calcul car ceux-ci doivent exécuter des tâches sur demande seulement (non de façon continue comme pour le premier type). Le bon type de comportement à employer est le « *OnRequestBehaviour* ». Ce type de comportement spécifie que les agents de ce type exécutent des tâches sur demande seulement. L'agent reste en attente de messages. Lorsqu'il reçoit un message, il vérifie si le sujet du message correspond au nom d'une de ses tâches. Si oui, il exécute cette tâche. Sinon, il retourne en attente.

6.3.3.3 Troisième type d'agent

Le troisième type est celui qui servira pour l'agent qui affiche les résultats et qui écrit ces derniers dans un fichier.

La création de ce type d'agent est identique au deuxième type. Cet agent doit exécuter des tâches sur demande seulement. Lorsqu'il reçoit un message, il doit afficher le résultat et écrire ceux-ci dans un fichier. Il adopte donc le même genre de comportement que les agents de calculs, soit le « *OnRequestBehaviour* ».

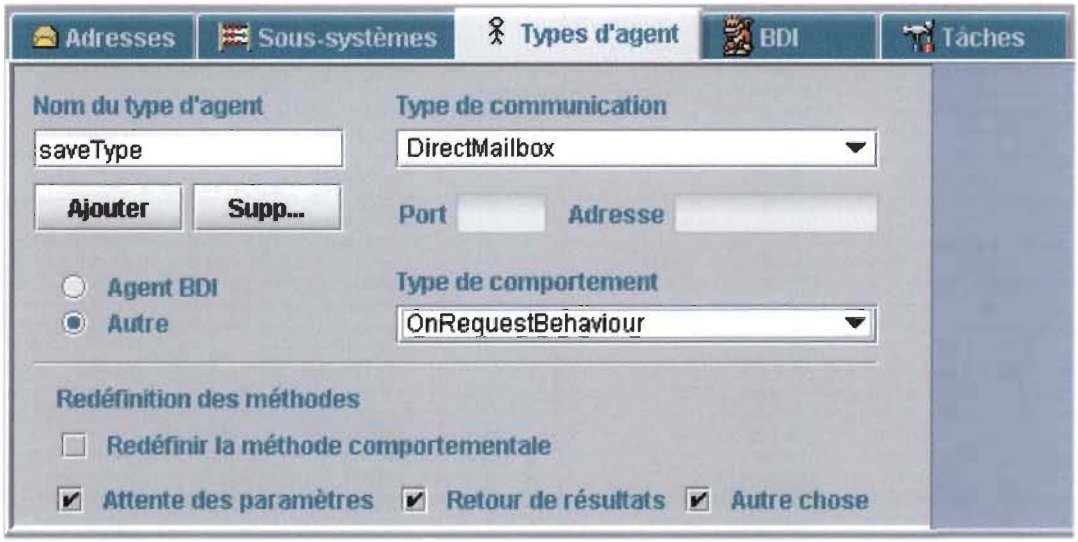

Figure 127 : Onglet pour la création du type d'agent de sauvegarde (étape 6)

Remarque

Il aurait été possible de ne pas créer ce troisième type et spécifier lors de la création des agents que celui-ci est du même type que les agents de calcul. Cependant, cette alternative restreint l'extensibilité du SMA et elle est sémantiquement erronée. L'ajout de propriétés dans le type se propagent dans les agents qui héritent de ce type. Si on veut ajouter une propriété au type de l'agent qui affiche et que celui-ci est du même type que les quatre agents de calcul, alors cette propriété se propagent dans ces quatre agents aussi. De plus, un problème se pose au niveau des redéfinitions de méthodes. Si une méthode est redéfinie dans le type (sélection de méthodes à redéfinir dans l'onglet Types d'agent), alors les méthodes redéfinies sont appelées lors de l'exécution de chaque agent héritant de ce type d'agent.

6.3.4 Création des tâches (Étape 7)

Pour la spécification des tâches, il faut se placer dans l'onglet Tâches (Figure 128). La création des tâches est très simple. Il suffit de spécifier le nom de la tâche qui pourra être effectuée par un ou des agents. Une description de chaque tâche est optionnelle. Le premier agent devra effectuer une seule tâche qui sera de demander un calcul et d'envoyer le calcul à l'agent capable d'effectuer cette opération « *ask* ». Les quatre agents de calcul peuvent chacun effectuer une et une seule tâche « *addition* », « *multiplication* », « *division* » ou « *soustraction* ». L'agent d'affichage et de sauvegarde effectuera la tâche « *save* » sur demande.

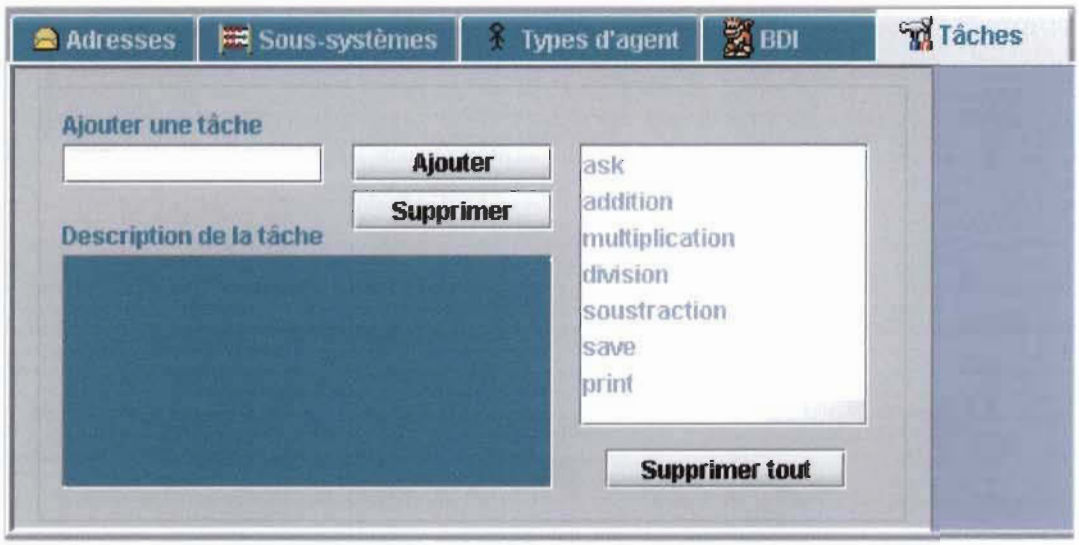

Figure 128 : Onglet pour la création des tâches (étape 7)

6.3.5 Créer les agents et spécifier leurs attributs (Étape 9)

La création des agents se fait dans le bas à gauche dans l'éditeur de systèmes. Il suffit de spécifier le type de l'agent à créer et de donner un nom à l'agent (Figure 129).

6.3.5.1 Création de l'agent « demandeur »

Figure 129 : Création des agents (Étape 9)

6.3.5.2 Création des 5 autres agents

La création des cinq autres agents est semblable. Tout d'abord, pour les quatre agents de calcul, il suffit de choisir « *calculType* » dans la liste déroulante « type de l'agent à ajouter » et de donner un nom à chaque agent. Enfin, pour l'agent d'affichage et de sauvegarde, il faut choisir « *saveType* » et lui donner un nom.

Une fois les agents créés, il se retrouvent tous dans la table des agents. Il faut maintenant personnaliser les attributs des six agents (Figure 130).

Nom de l'agent	Idendificateur unique	Temps d'attente	Priorité de l'agent
askAgent	ask	1	10
AddAgent	add	100000	Normale
MultAgent	mult	100000	Normale
SubsAgent	subs	100000	Normale
DivAgent	div	100000	Normale
SaveAgent	save	Maximum	10

Figure 130 : Table des attributs des agents (Étape 9)

Il est important de donner des noms simples et significatifs aux identificateurs uniques des quatre agents de calculs et de l'agent de sauvegarde. Le GUID (Global Unique Identifier) d'un agent est utilisé par les autres agents pour lui envoyer des messages. L'attribution d'un nom significatif pour le GUID simplifie l'implémentation qui suit les spécifications. Les temps d'attente et de priorité des agents ne sont pas très significatifs pour ce SMA. Cependant, il est intéressant de spécifier à l'agent qui demande les calculs et à l'agent qui affiche les résultats qu'ils ont une priorité maximale. Ceci permet aux agents qui font les calculs de les effectuer en « *Background* » pendant que le système continue ses opérations normales (demander des calculs et sauvegarde de ceux-ci).

6.3.6 Association des tâches aux agents (Étape 11)

Il faut maintenant associer chaque tâche aux agents (Figures 131 à 133). Cette opération est très simple. Il suffit de sélectionner les tâches à effectuer pour chaque agent (dans la liste des tâches).

Nom de l'agent	ask	addition
askAgent	☑	☐
AddAgent	☐	☑
MultAgent	☐	☐
SubsAgent	☐	☐
DivAgent	☐	☐
SaveAgent	☐	☐

Figure 131 : Table des agents et table des tâches (« ask » et « addition ») (Étape 11)

Nom de l'agent	multiplication	division
askAgent	☐	☐
AddAgent	☐	☐
MultAgent	☑	☐
SubsAgent	☐	☐
DivAgent	☐	☑
SaveAgent	☐	☐

Figure 132 : Table des agents et table des tâches (« multiplication » et « division ») (Étape 11)

Nom de l'agent	soustraction	save
askAgent	☐	☐
AddAgent	☐	☐
MultAgent	☐	☐
SubsAgent	☑	☐
DivAgent	☐	☐
SaveAgent	☐	☑

Figure 133 : Table des agents et table des tâches (« soustraction » et « save ») (Étape 11)

6.3.7 Valider le système (Étape 12)

La validation du système permet de vérifier plusieurs caractéristiques. Entre autre, elle permet de vérifier si tous les éléments d'un système sont présents dans l'éditeur et que les noms utilisés sont conformes pour la compilation Java. Pour valider le système, il suffit de cliquer sur « Valider » dans la barre d'outils.

Figure 134 : Options principales de la barre d'outils

Lors de la validation, une fenêtre de messages apparaîtra à l'écran. Si aucune erreur n'a été commise, la fenêtre de la figure 135 sera à l'écran.

Figure 135 : Fenêtre de validation du système (Étape 12)

6.3.8 Génération du système (Étape 13)

La génération du système permet de générer automatiquement le code source de l'application. Pour générer les fichiers sources de l'application, il suffit de cliquer sur « Générer » dans la barre d'outils. Lors de la génération, une fenêtre de messages (voir figure 136) apparaîtra à l'écran. Si aucune erreur n'était présente au niveau de la validation, la fenêtre suivante sera à l'écran.

Figure 136 : Fenêtre de messages de génération automatique (Étape 13)

Une fois la génération du code terminée, les fichiers auront été ajoutés à la liste des fichiers du projet dans l'éditeur de fichiers : ***Projet → Éditeur de fichiers*** (Figure 137).

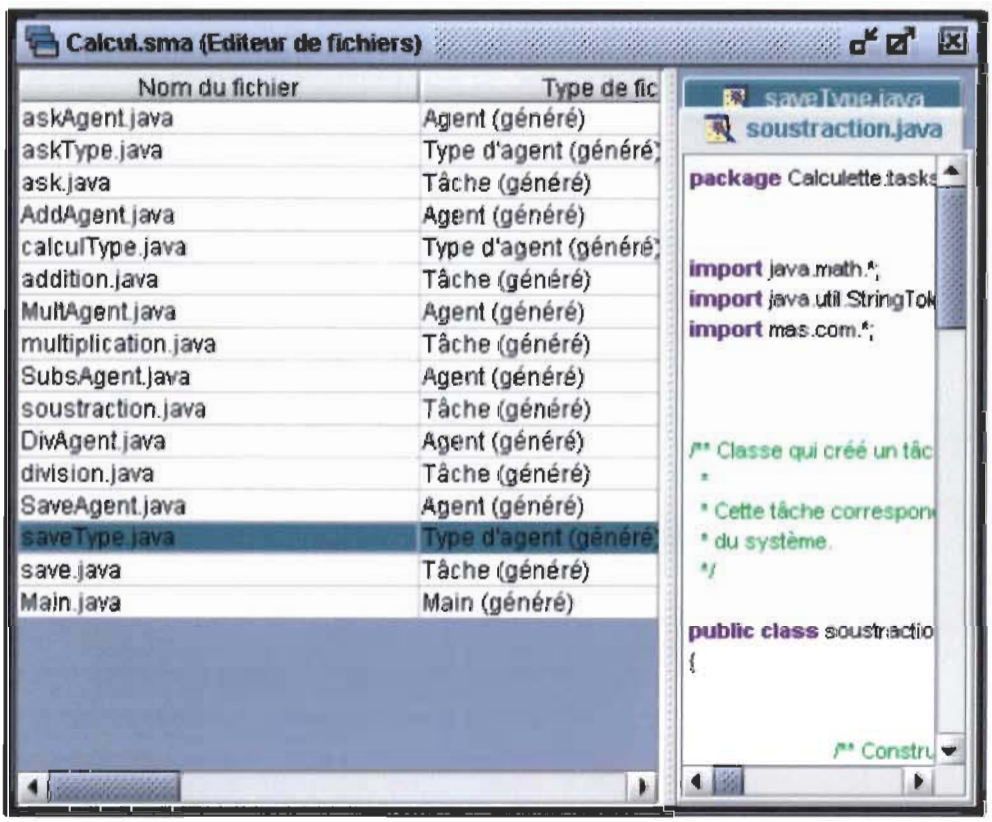

Figure 137 : Liste des fichiers générés dans l'éditeur de fichiers

6.3.9 Terminer l'implémentation (Étape 14)

Il ne reste plus qu'à implémenter les six tâches.

6.3.9.1 Implémentation de la tâche « *ask* »

Il faut importer le *package java.io* et la classe *mas.com.SimpleMessage* dans cette classe. Il faut aussi créer une variable statique qui permet de lire les chaînes de caractères entrées au clavier (Figure 138).

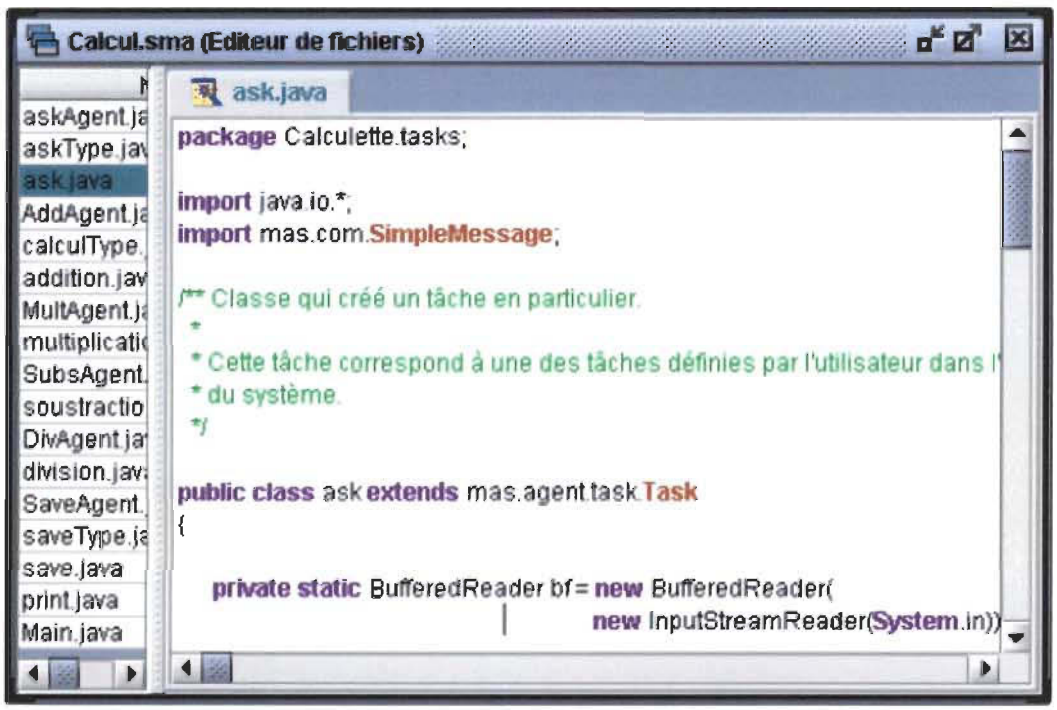

Figure 138 : Éditeur de fichiers de la tâche «ask »

Le code ***private static BufferedReader bf = new BufferedReader (new InputStreamReader (System.in));***

est du code Java standard permettant de créer un objet « *bf* » qui pourra lire les chaînes de caractères entrées au clavier. La variable est statique pour que celle-ci ne soit créée qu'une seule fois. La même variable servira pour la lecture de toutes les entrées.

Par la suite, il faut implémenter la méthode « *execute()* » de la tâche (voir la figure 139).

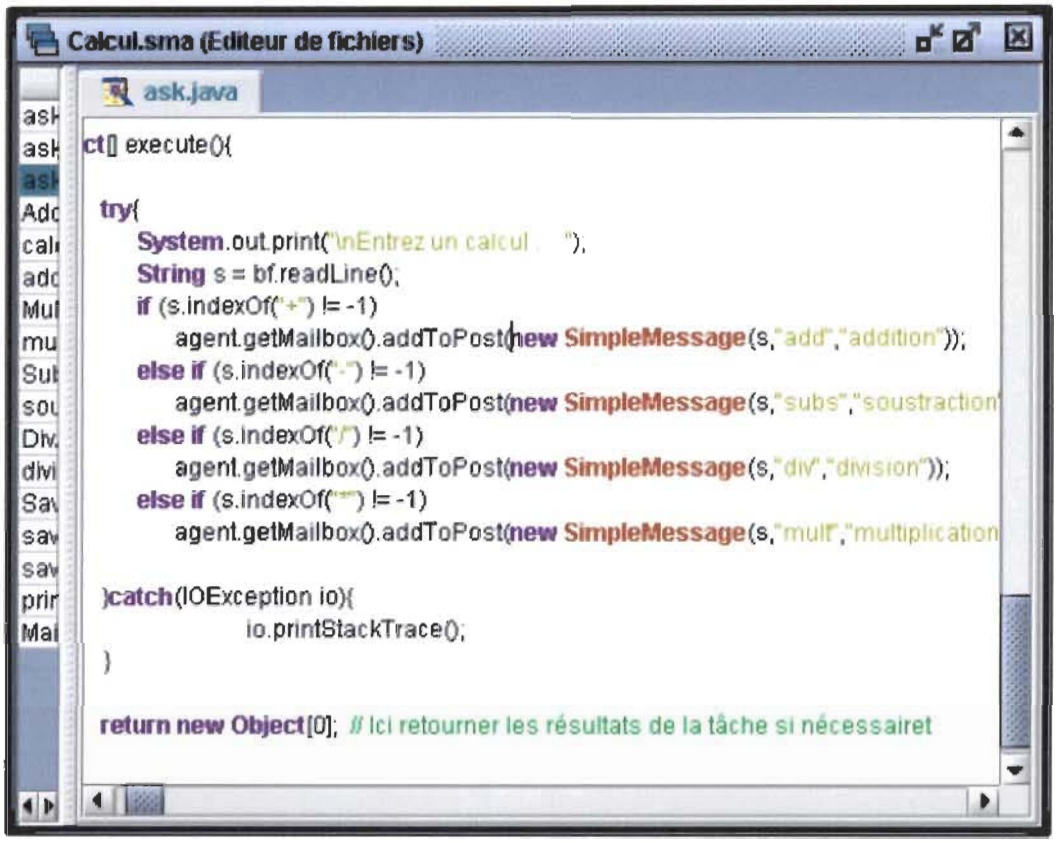

Figure 139 : Éditeur de fichiers (Méthode « execute() » de la tâche «ask»)

Code de base ajouté (Java standard)

- Afficher (*System.out*).
- Lecture de la chaîne de caractères (le calcul) entrée au clavier (*bf.readLine()*).
- Vérification de l'opération effectuée :

```
if (s.indexOf("+") != -1)
        ou
else if (s.indexOf("-") != -1)
        ou
 else if (s.indexOf("/") != -1)
        ou
else if (s.indexOf("*") != -1)
```

Note :

*La méthode indexOf(String s) de la Classe String de Java retourne l'index de la sous-chaîne passée en paramètre si elle se trouve à l'intérieur de la chaîne. Sinon, elle retourne -1. De cette façon, on vérifie si un des opérateurs (+, -, *, /) se trouve à l'intérieur de la chaîne. Si l'opérateur « + » se trouve à l'intérieur de la chaîne, on envoie le message à l'agent qui fait l'addition et ainsi de suite.*

Code spécifique (API OA)

- ***agent.getMailbox().addToPost(new mas.com.SimpleMessage (s,"add","addition"));***

Note : La ligne précédente est équivalente à :

```
mas.mail.AgentMailbox mail = agent.getMailbox() ;
mas.com.SimpleMessage mess = new
mas.com.SimpleMessage(s,"add","addition") ;
mail.addToPost(mess);
```

- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).
- ***getMailbox() :*** Retourne la *Mailbox* de l'agent.
- ***addToPost(Message m) :*** Ajoute un message dans la liste d'envoi de la *Mailbox* de l'agent.
- ***new mas.com.SimpleMessage(Object o, String guid, String subject) :*** Est un message à envoyer, prenant comme paramètres n'importe quel objet (le message à envoyer), l'identificateur de l'agent auquel on envoie le message et le sujet. Dans ce cas-ci, étant donné que l'agent « *add* » est un agent qui a un comportement « *OnRequestBehaviour* », il effectuera une tâche si le sujet du message correspond au nom d'une de ses tâches. Note : Le sujet ne sert d'appel de tâches que pour les comportements « *OnRequestBehaviour* ». Dans les autres cas, le sujet est libre d'utilisation pour le concepteur. De plus, pour le moment, la méthode « *getCurrentMessage()* » ne sert que pour les agents qui ont un comportement « *OnRequestBehaviour* ».
- ***return new Object[0]; :*** S'il faut retourner des résultats (qui seront « attrapés » par la redéfinition de la méthode d'attente de résultats du type de l'agent), c'est ici qu'il faut les retourner. Sinon, il faut retourner un tableau vide d'objets.

6.3.9.2 Implémentation des tâches « opérateurs »

L'implémentation des quatre tâches des opérateurs est presque identique. Pour cette raison, seulement l'implémentation de la tâche soustraction sera expliquée (Figure 140). Pour cette tâche (de même pour les trois autres opérateurs), il est nécessaire de faire les importations suivantes :

- *import java.math.BigDecimal;*
- *import java.util.StringTokenizer;*
- *import mas.com.**;

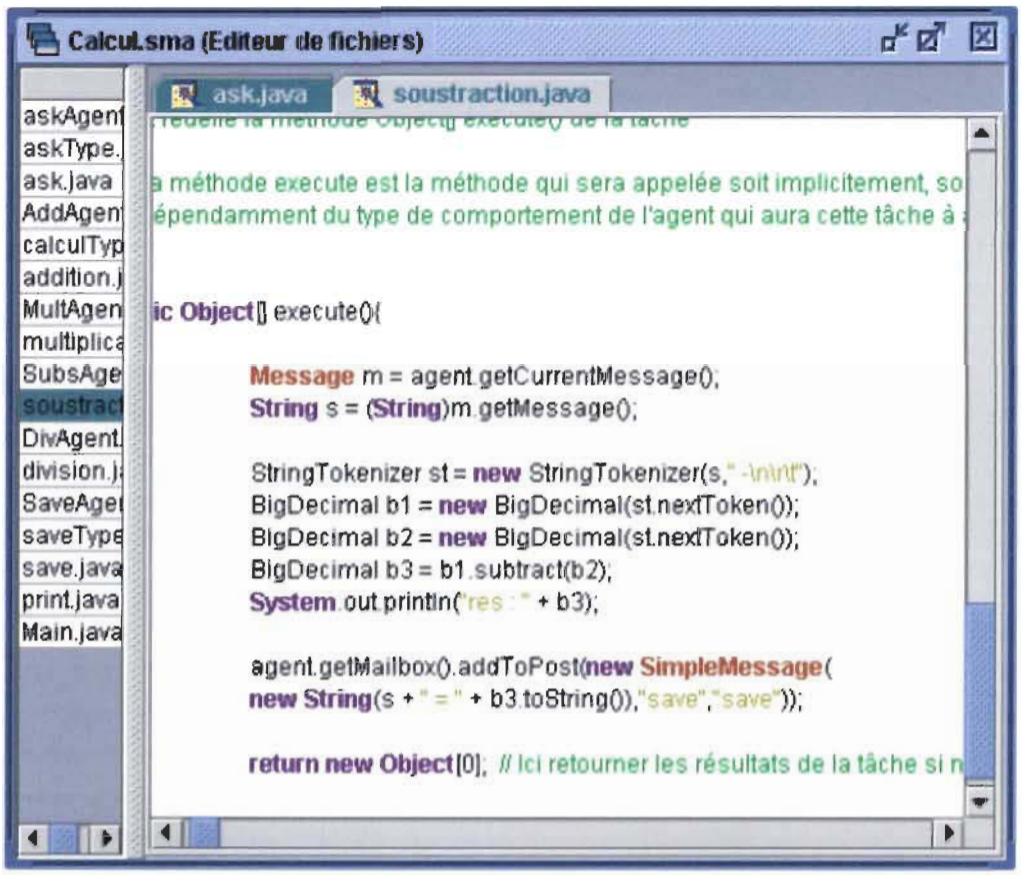

Figure 140 : Éditeur de fichiers (Méthode « execute() » de la tâche « soustraction »)

Code de base ajouté (Java standard)

- Création d'un « *Tokenizer* » pour découper la chaîne de caractères qui contient l'opération entrée au clavier. Il faut noter ici qu'étant donné que c'est l'implémentation de la soustraction, les caractères de délimitation des « *tokens* » contiennent le caractère « - ». Pour les autres opérations, il est nécessaire de changer un des caractères du deuxième paramètre (le caractère moins) par le caractère de l'opérateur approprié. *(StringTokenizer st = new StringTokenizer(s," -\n\r\t");)*.
- Création d'un objet contenant un nombre indéfiniment grand. Celui-ci contient le premier nombre de l'opération *(BigDecimal b1 = new BigDecimal(st.nextToken());)*.
- Création d'un objet contenant un nombre indéfiniment grand. Celui-ci contient le deuxième nombre de l'opération *(BigDecimal b2 = new BigDecimal(st.nextToken());)*.
- Soustraction de deux nombres indéfiniment grands. Le résultat se retrouve dans un nouvel objet *(BigDecimal b3 = b1.subtract(b2);)*.
- Affichage à l'écran du résultat *(System.out.println("res : " + b3);)*.

Code spécifique (API OA)

- ***Message m = agent.getCurrentMessage();***
- ***Message :*** Le message couramment traité par l'agent.
- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).
- **getCurrrentMessage() :** Retourne le message couramment traité par l'agent
- ***String s = (String)m.getMessage(); :*** Cette méthode retourne le contenu du message qui a été reçu. Celui-ci contient l'opération de soustraction à effectuer. Le message étant de type *Object* il est nécessaire de le « *caster* » en un objet de type *String*.
- ***agent.getMailbox().addToPost(new SimpleMessage(new String(s + " = " + b3.toString()),"save","save"));***

Note : La ligne précédente est équivalente à :

```
mas.mail.AgentMailbox mail = agent.getMailbox() ;
String s2 = new String(s + " = " + b3.toString()),"save","save"));
mas.com.SimpleMessage mess = new
mas.com.SimpleMessage(s2,"add","addition") ;
mail.addToPost(mess);
```

- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).
- ***getMailbox() :*** Retourne la *Mailbox* de l'agent.
- ***addToPost(Message m) :*** Ajoute un message dans la liste d'envoi de la *Mailbox* de l'agent.
- ***new mas.com.SimpleMessage(Object o, String guid, String subject) :*** Est un message à envoyer, prenant comme paramètres n'importe quel objet (le message à envoyer), l'identificateur de l'agent auquel il faut envoyer le message et le sujet. Dans ce cas-ci, étant donné que l'agent « *save* » est un agent qui a un comportement « *OnRequestBehaviour* », il effectuera une tâche si le sujet du message correspond au nom d'une de ses tâches. Note : Le sujet ne sert d'appel de tâches que pour les comportements « *OnRequestBehaviour* ». Dans les autres cas, le sujet est libre d'utilisation pour le concepteur. De plus, pour le moment, la méthode « *getCurrentMessage()* » ne sert que pour les agents qui ont un comportement « *OnRequestBehaviour* ». Ici, il faut envoyer un message à l'agent qui sauvegarde les résultats. Le premier paramètre est le message à envoyer à l'agent (le résultat à sauvegarder). Le deuxième paramètre est le GUID de l'agent et le dernier paramètre est le sujet, qui se trouve à être le nom de la méthode à exécuter par cet agent.
- ***return new Object[0]; :*** S'il faut retourner des résultats (qui seront « attrapés » par la redéfinition de la méthode d'attente de résultats du type de l'agent), c'est ici qu'il faut les retourner. Sinon, il faut retourner un tableau vide d'objets.

6.3.9.3 Implémentation de la tâche « *save* »

La tâche « *save* » est implémentée d'une façon simple et non-optimisée. Normalement, il n'aurait pas été nécessaire de garder en mémoire toutes les opérations. De plus, il ne faudrait pas réécrire celles-ci à chaque fois. L'exemple se veut seulement une illustration de la communication entre plusieurs agents. Pour cette tâche, il est nécessaire de faire les importations et les déclarations illustrées dans la figure 141 :

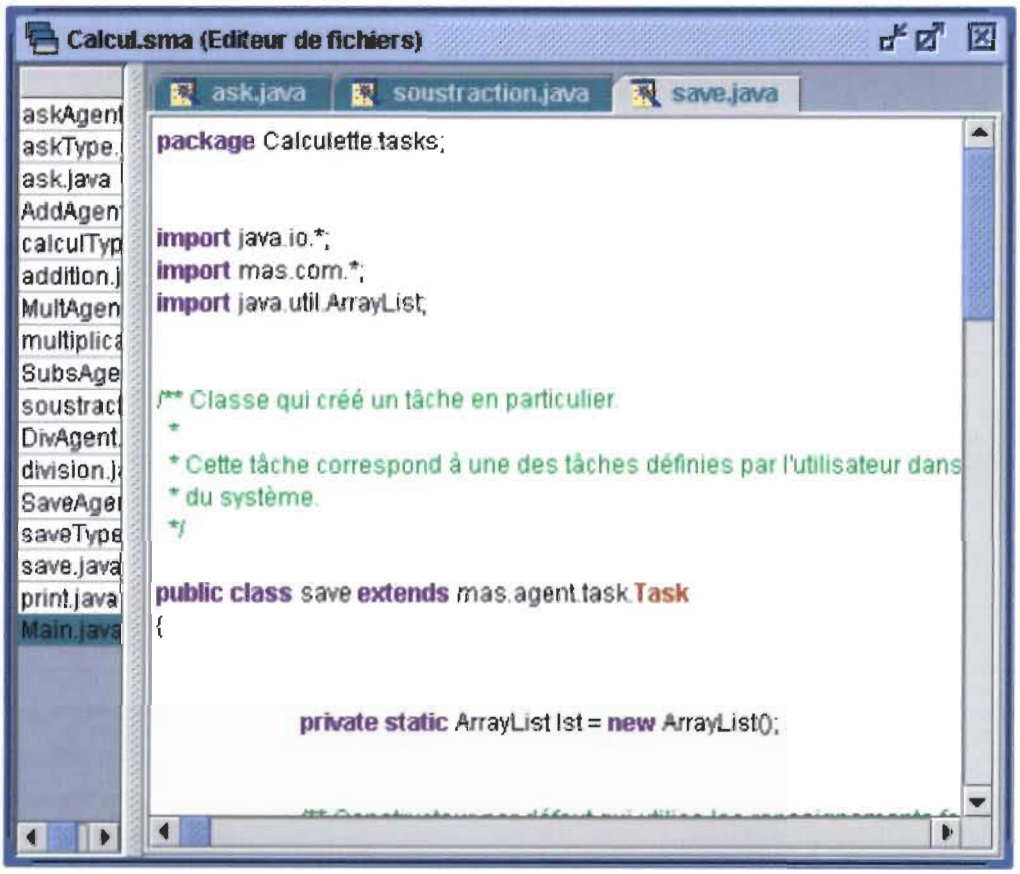

Figure 141 : Éditeur de fichiers de la tâche « save »

Le code ***private static ArrayList lst = new ArrayList() ;***

est du code Java standard permettant de créer un objet « *lst* » qui pourra garder en mémoire toutes les opérations reçues des agents. Par la suite, il faut implémenter la méthode « *execute()* » de la tâche.

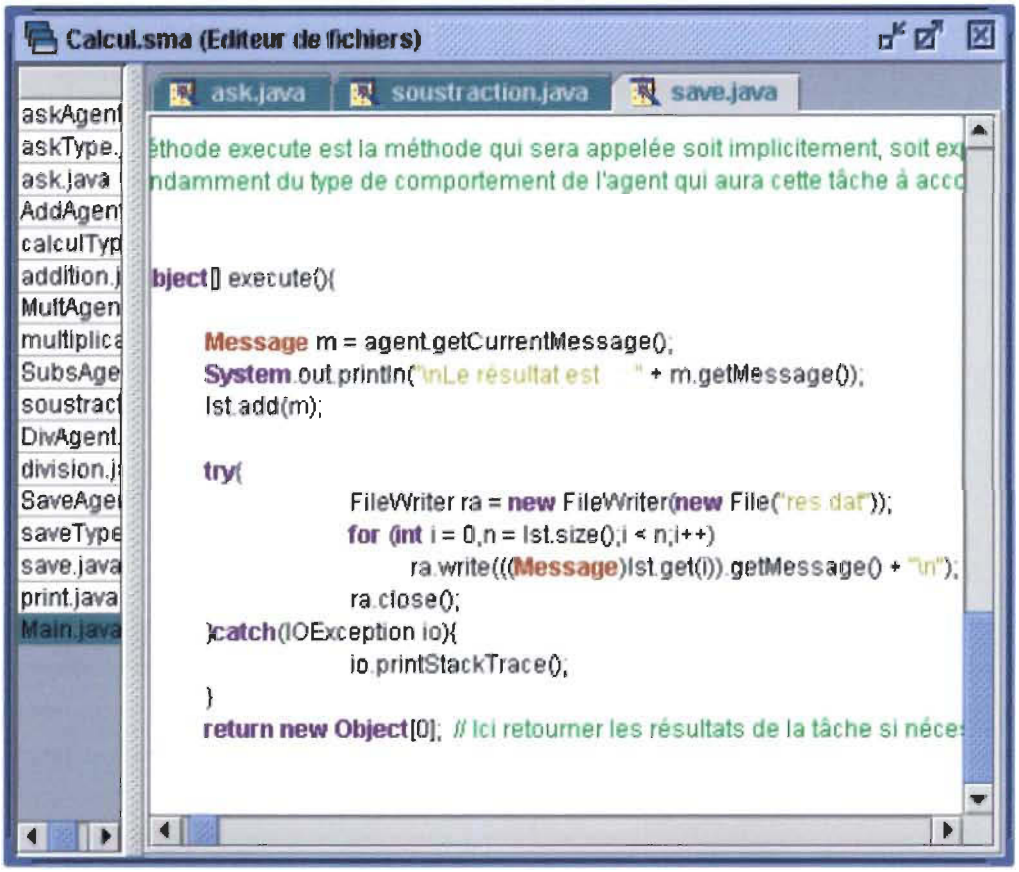

Figure 142 : Éditeur de fichiers (Méthode « execute() » de la tâche « save »)

Code de base ajouté (Java standard)

- Affichage du résultat à l'écran *(System.out.println("\nLe résultat est : " + ...)*.
- Insertion du résultat (sous forme d'objet de type *Message*) dans la liste (*lst.add(...);*).
- Gestion des erreurs de *IO* (« *try* » et « *catch* »).
- Création du fichier dans lequel les résultats seront écrits. (*FileWriter ra = new FileWriter(new File("res.dat"));*).
- Boucle « *For* » pour boucler dans toute la liste (*for (int i = 0,n = lst.size();i < n;i++)*).
- Écriture de tous les résultats dans le fichier. Ici, il est nécessaire de « *caster* » les objets qui sont retournés par la méthode « *get* » car celle-ci retourne des objets de type *Object*. Il faut donc les retourner sous la forme du type qu'ils avaient dans la liste (type *Message*). Comme précédemment, la méthode « *getMessage()* retourne l'objet qui a été encapsulé dans le message (celui-ci peut être de n'importe quel type). Par contre, dans cet exemple, celui-ci est nécessairement de type *String*. *(ra.write(((Message)lst.get(i)).getMessage() + "\n");)*.
- Fermeture du fichier *(ra.close();)*.

Code spécifique (API OA)

- ***Message m = agent.getCurrentMessage();***
- ***Message :*** Le message couramment traité par l'agent.
- ***agent :*** Est l'agent qui exécute la méthode (cet objet est implicite).
- **getCurrrentMessage() :** Retourne le message couramment traité par l'agent.
- ***return new Object[0]; :*** S'il faut retourner des résultats (qui seront « attrapés » par la redéfinition de la méthode d'attente de résultats du type de l'agent), c'est ici qu'il faut les retourner. Sinon, on retourne un tableau vide d'objets.

6.3.10 Compiler et exécuter le projet

6.3.10.1 Compilation du projet

Lorsque l'implémentation est terminée, il reste à compiler le projet : ***Projet → Compiler le projet***. Une fenêtre de message apparaîtra et donnera les messages d'usages. Si aucune compilation ne se produit, c'est probablement dû au fait que le chemin du programme de compilation n'est pas initialisé : ***Options → Options du compilateur*** (Chemin *javac*).

6.3.10.2 Exécution du projet

L'exécution se fait en choisissant ***Projet → Exécuter le projet***. Si une erreur « *NoClassDefFoundError* » est levée, c'est probablement parce que la classe à exécuter n'est pas spécifiée. Pour spécifier la classe à exécuter, il est nécessaire d'aller dans ***Options → Options du projet*** (Classe à exécuter) et spécifier le nom de la classe (avec le nom de *package*). Pour ce projet, la classe à spécifier est : Calculette.Main.

7. Évaluation

Sans critères d'évaluation bien précis, l'analyse d'outils comme celui-ci se veut très difficile. De plus, l'évaluation est dépendante des besoins d'utilisation et des conditions dans lesquelles l'ED est utilisé. Comme nous l'avons mentionné dans un des chapitres précédents, l'évaluation d'un ED doit se faire en fonction de critères bien précis et bien définis. Cela est bien important si on veut comparer l'évaluation avec celles des autres environnements.

Nous avons déjà publié une étude comparative des différents outils d'aide au développement de SMA [Garneau02]. Cette dernière évalue une dizaine d'outils plus ou moins complets. De plus, elle commente les différents résultats obtenus.

Pour évaluer DMAS Builder, notre premier choix était de faire analyser notre environnement par des personnes externes à notre équipe et publier cette évaluation dans ce travail (mémoire). Cependant, comme nous nous sommes heurté à des contraintes de temps (l'évaluation n'était pas prête pour le premier dépôt), nous avons dû opter pour une solution intermédiaire. Nous avons donc créé une nouvelle grille d'évaluation (adaptée aux objectifs du document) que nous expliquons dans cette section (Figure 143). De plus, pour faciliter et uniformiser l'évaluation de l'environnement, nous avons ajouté quelques mises en situation. Ces dernières suggèrent aux évaluateurs les applications à développer pour l'évaluation. Ces mises en situation sont simples mais permettent d'utiliser plusieurs facettes de l'environnement. Les problèmes à résoudre doivent être clairs, concis et précis si l'on veut obtenir une évaluation relativement cohérente des différents évaluateurs. Ceux-ci doivent être capable d'arriver à résoudre le problème et à obtenir les résultats escomptés pour, par la suite, effectuer une évaluation des différents critères.

7.1 Mises en situation

Les différentes mises en situation ont été choisies pour forcer l'évaluateur à utiliser les différents outils de l'environnement de développement.

7.1.1 Le projet Hello World

Le premier programme « *Hello World* » prend tout son sens car il permet de se familiariser avec l'environnement de développement. Le but de ce premier programme est de créer un projet contenant deux agents sur la même JVM. Un des deux agents demande le nom de l'utilisateur à l'écran (ici, la demande du nom peut se faire en mode console ou par une fenêtre d'entrée de texte). Une fois le nom entré, il l'envoie au second agent qui dit « salut… » suivi du nom entré à l'écran.

Ce petit programme semble anodin. Cependant, il est beaucoup plus complexe qu'un simple programme qui affiche une chaîne de caractères à l'écran. Ici, il est nécessaire

d'utiliser les *Mailbox* des agents. Il faut créer un sous-système, une tâche pour chaque opération, spécifier des comportements aux agents, générer le code et terminer l'implémentation dans le code source.

7.1.2 La calculatrice distribuée

Le deuxième programme est une petite calculatrice distribuée permettant d'effecteur des opérations mathématiques de base. Le programme doit pouvoir additionner, soustraire, multiplier ou diviser deux nombres infiniment grands. De plus, toutes les opérations mathématiques sont écrites dans un fichier sous la forme :

N1 op1 n2 = res1
N3 op1 n4 = res2
N5 op1 n6 = res3

Le programme fonctionne de la façon suivante :

- Le programme demande à l'utilisateur d'entrer un calcul.
- L'utilisateur entre le calcul (ex.: 44 + 66).
- Le programme donne le résultat à l'écran et sauvegarde le résultat dans le fichier.
- À nouveau, le programme demande d'entrer un calcul…

Contraintes :

Le programme doit posséder 6 agents :

- Un agent qui demande le calcul à effectuer et délègue l'opération au bon agent.
- Un agent qui effectue l'addition (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la multiplication (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la soustraction (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui effectue la division (et envoie le calcul à l'agent de sauvegarde).
- Un agent qui enregistre les calculs dans le fichier et affiche le résultat à l'écran.

7.2 *Évaluation*

L'évaluation se divise en 20 points que l'utilisateur évalue selon la pondération suivante :

- 5 si l'ED répond très bien au critère.
- 4 si l'ED répond bien au critère.
- 3 si l'ED répond moyennement au critère.
- 2 si l'ED répond peu au critère.
- 1 si l'ED répond très peu au critère.
- 0 si l'ED ne répond pas du tout au critère.

7.2.1 Facilité d'apprentissage de l'ED

Ce critère est déterminé en fonction de plusieurs facteurs : la qualité de la documentation, la complexité des composants et les concepts utilisés. Les connaissances préalables à son utilisation comme le langage de programmation, le langage de communication entre les agents, les protocoles d'interaction et autres sont aussi à prendre en considération.

7.2.2 Facilité de transition entre le développement et l'implémentation

Facilité de passer du modèle à son implémentation. Plusieurs méthodologies développées sont très intéressantes au niveau conceptuel mais difficilement applicables, notamment en ce qui concerne l'implémentation.

7.2.3 Souplesse de l'outil

C'est la flexibilité et la polyvalence de l'outil par rapport à l'utilisation de ses composants.

7.2.4 Communication inter-agents

Le programmeur ne doit pas avoir à se préoccuper de l'implémentation des connexions entre les différentes machines, des protocoles de communication, de la sécurité, de la synchronisation, des services de messagerie et autres. Ces services doivent donc être déjà implémentés.

7.2.5 Support graphique pour le développement

L'environnement propose des interfaces graphiques facilitant et accélérant le développement du SMA. Ces dernières peuvent servir à la création du modèle, la création des agents, l'élaboration de conversations, la création des tâches, permettre une validation du système, etc.

7.2.6 Support pour l'implémentation

L'environnement offre des facilités pour l'implémentation du SMA. Il permet l'implémentation directement à l'intérieur de l'environnement (programmation). De plus, il permet de compiler les composants (lorsque nécessaire). Aussi, il permet l'édition de fichier ainsi que la sauvegarde. Enfin, il supporte la recherche de documentation.

7.2.7 Support pour l'exécution

L'environnement offre des facilités pour l'exécution du SMA. Il permet son exécution à l'intérieur de l'environnement. L'environnement offre aussi des facilités pour l'exécution à l'extérieur de l'environnement. De plus, il offre des services pour la connexion entre les sous-systèmes qui sont distribués. Enfin, il offre des services pour le déploiement des agents sur les différentes plate-formes.

7.2.8 Diminution de l'effort demandé

La diminution de l'effort demandé pour le développement en terme de quantité de code à écrire, de complexité des composants à implémenter, de facilité d'utilisation des composants existants. La spécification des composants et de leurs caractéristiques est simple.

7.2.9 Réutilisabilité du code

Les composants spécifiés, générés et/ou implémentés sont facilement réutilisables. Le code généré est récupérable pour une utilisation future. De plus, les sous-systèmes peuvent servir à nouveau et être incorporés à d'autres SMA.

7.2.10 Simplicité d'implémentation

Un langage supportant bien la programmation OO, le *multi-threads* et la programmation réseau procurent des avantages importants. De plus, les composants doivent être facilement identifiables (nom, *packages*, documentation, paramètres, etc.). Enfin, les classes et les services disponibles doivent être faciles à utiliser.

7.2.11 Génération automatique de code

Si les spécifications du système sont possibles au niveau des interfaces, il est important de pouvoir générer le code source du système (au moins les squelettes) et de ses différents composants.

7.2.12 Extensibilité du code

Les utilitaires fournis par les outils comme les modules, les agents pré-définis ou le code généré doivent être facilement modifiables. De plus, il faut pouvoir ajouter facilement du code à celui déjà existant.

7.2.13 Déploiement inter-machines

La possibilité de répartir le système sur plusieurs machines est un critère très important au niveau de l'exécution.

7.2.14 Documentation

La documentation disponible est de qualité. Elle couvre l'ensemble des composants de l'outil. De plus, elle est claire, concise et non-ambiguë. Elle est intégrée à l'environnement et facile d'accès.

7.2.15 Application indépendante

L'environnement permet de créer des applications indépendantes de l'environnement de développement.

7.2.16 Services standards offerts

L'environnement offre les services standards aux ED comme la compilation de l'application, l'exécution de l'application, l'accès aux différents modules nécessaires à l'implémentation du système, une documentation « *inline* », etc.

7.2.17 Déploiement simple

L'exécution de systèmes distribués est généralement très complexe. L'environnement doit offrir un mécanisme simple de déploiement du système (des différents sous-systèmes).

7.2.18 Applications réelles développées

L'environnement de développement permet de développer des systèmes d'envergure intéressantes permettant de résoudre des problèmes réels.

7.2.19 Facilité d'utilisation

L'environnement est simple d'utilisation. Les interfaces sont conviviales. Les concepts et les termes utilisés sont facilement compréhensibles. De plus, la navigation à l'intérieur des interfaces est simple et les options disponibles sont faciles à utiliser.

7.2.20 Polyvalence

L'environnement peut servir à créer plusieurs types d'application. Il permet de créer des applications dans différents domaines d'application. De plus, il permet l'utilisation de méthodologies diverses, de différentes architectures (distribuée, centralisée ou autres) ainsi que différents modes de communication.

Critères	0	1	2	3	4	5
1. Facilité d'apprentissage de l'ED						
2. Transition (développement, implémentation)						
3. Souplesse de l'outil						
4. Communication inter-agents						
5. Support pour le développement						
6. Support pour l'implémentation						
7. Support pour l'exécution						
8. Diminution de l'effort demandé						
9. Réutilisabilité du code						
10. Simplicité d'implémentation						
11. Génération automatique de code						
12. Extensibilité du code						
13. Déploiement inter-machines						
14. Documentation						
15. Application indépendante						
16. Services standards offerts						
17. Déploiement simple						
18. Applications réelles développées						
19. Facilité d'utilisation						
20. Polyvalence						
Total /100						

Figure 143 : Grille d'évaluation

8. Conclusion

Ce mémoire présente ***DMAS Builder***, un environnement complet de développement de systèmes multi-agents totalement distribués et orienté-tâches.

8.1 Résumé

Ce document est le résultat de deux années de travail intensif sur les SMA. Dans un premiers temps, nous avons étudié les travaux effectués sur les méthodologies de développement de SMA. Les concepts de programmation OA et de SMA sont des idées très intéressantes et les méthodologies développées fournissent des patrons théoriques pour la modélisation de ceux-ci. Cependant, les systèmes à base d'agents spécifiés à partir de ces méthodologies sont souvent difficiles à implémenter directement avec des langages de programmation standards, comme Java ou C++. C'est pourquoi plusieurs méthodologies ont mené à la création d'outils d'aide au développement de SMA. De plus, plusieurs autres équipes de chercheurs axent leurs travaux dans l'élaboration de solutions quant au support pour le développement de SMA, étant donné la problématique d'application de ce type de méthodologie.

Nos intérêts étant beaucoup plus au niveau applicatif que théorique, il était logique que nous suivions cette tangente. Nous nous sommes donc intéressés aux outils d'aide au développement de SMA. Par la suite, nous les avons évalués et comparés [Garneau02]. Cette évaluation fût faite dans l'optique de trouver les forces et les faiblesses chroniques des outils par rapport à un environnement de développement complet de développement de SMA. Les conclusions de cette évaluation sont claires et non-ambiguës. En effet, plusieurs lacunes chroniques sont présentes dans la majorité des ED de SMA, ce qui rend leur utilisation inappropriée voir même impossible pour le développement de systèmes réels d'envergure intéressante.

Finalement, nous avons conçu un environnement complet de développement de systèmes multi-agents totalement distribués répondant [Garneau03], en grandes parties, aux exigences nécessaires identifiées dans notre étude comparative et dans ce document.

8.2 DMAS Builder

DMAS Builder est un environnement complet permettant le développement rapide de systèmes multi-agents totalement distribués.

L'outil est constitué de deux modules principaux : la librairie de classes orientées-agent et l'environnement de développement.

8.2.1 La librairie de classes OA

La librairie de classes OA permet une implémentation simple, rapide et efficace des composants d'un SMA distribué. La librairie possède plusieurs classes (environ 200) encapsulant plusieurs concepts et services nécessaires aux agents :

- Un service de messagerie complet permettant une communication extrêmement simple via plusieurs modes et protocoles (RMI, TCP, UDP, *multicast* et autres).
- Un service de nommage permettant de retrouver les entités, leurs coordonnées et autres informations essentielles à la communication entre agents.
- Un services d'annuaire permettant de trouver les ressources nécessaires à l'exécution de tâches et la résolution de problèmes.
- Un service complètement automatisé d'enregistrement des sous-sysèmes (JVM) auprès des autres JVM.
- Un service totalement encapsulé d'envoi et de réception d'événements AWT Java.
- Une dizaine de comportements déjà implémentés pour les agents.
- Un patron générique pour l'ajout et l'exécution de tâches par les agents.
- La création simple de bases de connaissances et un engin d'inférence pour celles-ci.
- L'encapsulation et automatisation des mécanismes d'inférence pour les bases de connaissances à l'intérieur des agents.

8.2.2 L'environnement de développement

L'ED simplifie et accélère grandement le développement de SMA totalement distribués. Il offre plusieurs caractéristiques et fonctionnalités nécessaires dont quelques-unes, jugées essentielles, n'existant pas dans la majorité des outils d'aide au développement de SMA :

- L'environnement permet la spécification, via des interfaces graphiques, de tous les composants d'un SMA : les JVM, les agents et leurs attributs (dont leur système de communication ou messagerie), les bases de connaissances, les services de nommages, les services d'annuaire, les services d'enregistrement automatique, les tâches, les interactions avec la file des événements AWT Java et autres. L'environnement permet la validation automatisée des spécifications du SMA.
- L'environnement permet la génération automatique du code source de tous les composants du SMA (tous ceux spécifiés au premier point).
- Une « exportation » du système permet aux systèmes générés d'être indépendants de l'environnement de développement en recopiant les fichiers sources de la librairie OA nécessaires à l'exécution indépendante du SMA.
- L'environnement offre un service automatisé d'archivage permettant de créer une archive compressée Java « .jar » exécutable pour chaque sous-système du SMA.

- Une extensibilité et flexibilité complète par sa possibilité de modifier les classes de la librairie OA (qui ont été recopiées lors de l'exportation et qui servent maintenant pour l'exécution du SMA).
- Une bonne réutilisabilité :
    - Par son mécanisme d'exportation et d'importation de bases de connaissances.
    - Par sa sauvegarde de projet et de fichiers.
    - Par une recopie de projets.
    - Par sa réutilisation des fichiers sources générés.
- L'environnement met l'accent sur la facilité et la simplicité du développement.
- L'environnement permet l'implémentation, la compilation et l'exécution de SMA en Java.
- Il fournit une aide complète (intégrée à l'environnement et supportée par un outil de recherche) sur la librairie OA et les API Java.
- Il offre plusieurs options globales pour la personnalisation de l'environnement.
- Il facilite l'initialisation et l'utilisation de plusieurs paramètres par un travail dynamique « intelligent » de l'environnement.
- Il permet une navigation simple et intuitive à l'intérieur des différentes interfaces.

8.3 Discussion

Un des principaux problèmes de l'approche SMA est la grande difficulté de transfert du modèle à l'implantation. La complexité des méthodologies et des modèles qui en découlent décourage la plupart des développeurs à utiliser cette approche. Dans cette optique, les travaux et les publications effectués ces dernières années dans le domaine des SMA tendent à être peu rassurants. Quelques équipes s'efforcent d'appliquer ces concepts. Cependant, beaucoup trop de chercheurs ignorent complètement le niveau applicatif, ce qui fait que le gouffre entre le théorique et la pratique est encore immense. Nous avons fait ce travail dans l'optique de fournir aux développeurs un outil permettant de mettre en pratique quelques concepts de base de l'approche agent. L'environnement développé ne permet pas pour l'instant l'application de concepts orientés-agent très complexes. Par contre, il fournit tous les éléments de base nécessaires à la conception, l'implémentation et le déploiement rapide de systèmes multi-agents réels d'envergures intéressantes. Il est aussi une base pour de futurs travaux.

8.4 Conclusion et travaux futurs

Bien sûr, ***DMAS Builder*** n'est pas le premier outil d'aide au développement de SMA. Cependant, il est un produit innovateur et original permettant enfin un développement efficace de SMA totalement distribués. Il est le premier de sa lignée par sa complétion au niveau des possibilités d'implémentation (spécification, validation, génération automatique, programmation, aide, compilation, exécution, exportation, archivage, création d'exécutable). Aucun outil avant lui ne permettait un développement aussi efficace, portable, extensible, simple, intuitif, rapide et complet de SMA distribués

d'envergure intéressante. L'environnement est implémenté à 100% en Java et permet le développement en Java.

Il est évident que plusieurs améliorations sont à faire dans ***DMAS Builder***. Il était inconcevable d'implémenter tous les services nécessaires aux SMA dans le cadre d'une maîtrise. Pour la première version, nous avons donc laissé plusieurs aspects en suspend. Cependant, ils seront ajoutés au fur et à mesure que des collègues mettront la main à la pâte.

Par contre, étant donné que nous continuons le développement de ***DMAS Builder*** au cours des prochains mois (et probablement des prochaines années), plusieurs améliorations et ajouts seront fait d'ici peu :

- Ajout sous peu d'un outil de « *débuggage* » distribué qui permettra d'obtenir les informations en temps réel sur tous les agents et autres composants des différentes JVM du système.
- Ajout d'autres comportements pré-implémentés.
- Quelques protocoles de communication et d'interaction.
- Quelques mécanismes de coordination de base.
- Ajout de types de DF plus spécialisés offrant des services bien définis.
- Et à plus long terme, permettre la mobilité des agents.

Enfin, la première version est sur le point d'être disponible à tous. Elle comprendra l'environnement de développement, la documentation et les exemples permettant une meilleure compréhension. Nous sommes très optimistes quant à l'utilité de l'environnement qui sera le premier à offrir ce genre de services pour le développement et le déploiement de SMA.

9. Références

[AgentBuilder R.M.], *An Integrated Toolkit for Constructing Intelligent Sofware Agents*, AgentBuilder, Reference Manual, Avril 2000.

[AgentBuilder U.G.], *An Integrated Toolkit for Constructing Intelligent Sofware Agents*, AgentBuilder, User's Guide, Avril 2000.

[AgentTool U.M.], AgentTool, User's Manual, Juin 2001.

[Baker], A. D. Baker, D. Chauhan, *JAFMAS : A Multiagent Application Development System.*

[Barbuceanu], M. Barbuceanu, M.S. Fox, *The Design of COOL: A Language for Representing Cooperation-Knowledge in Multi-Agent Systems.*

[Barbuceanu97], M. Barbuceanu, M.S. Fox, *Integrating Communicative action, conversation and decision theory to coordinate agents*, Proceedings of First International Conference on Autonomous Agents, pp. 49-58. 1997.

[Bradshaw96], J. M. Bradshaw, S. Dutfield, P. Benoit, J. D. Woolley, *KAoS: Toward an Industrial-Strength Open Agent Architecture*, G. M. O'Hare, N. R. Jennings, eds. Foundations of Distributed Artificial Intelligence, John Wiley & Sons, p. 375 – 418, 1996.

[Brazier97], F. M. T. Brazier, B. M. Dunin-Keplicz, N. R. Jennings, J. Treur, *DESIRE : Modelling Multi-Agent Systems in a Compositional Formal Framework*, Int Journal of Cooperative Information Systems, Vol 6, No 1, p. 67-94, 1997.

[Busetta99], P. Busetta, R. Ronnquist, A. Hodgson, A. Lucas, *Jack intelligent agents - components for intelligent agents in java*, AgentLink News Letter, Janvier 1999.

[Chauhan97], D..Chauhan, *JAFMAS:A Java-Based Agent Framework for Multi-Agent Systems Development and Implementation*, Masters Thesis, ECECS Department, University of Cincinnati, Juillet 1997.

[Collis98], J. C. Collis, L. C. Lee, *Building Electronic Marketplaces with the {ZEUS} Agent Tool-kit*, AMET, p. 1-24, 1998.

[Collis98b], J. C. Collis, H. Nwana, D. Ndumu, L. C. Lee, *Zeus: A collaborative agents toolkit*, A collaborative agents toolkit. In Proceedings of the Third International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, p. 377-392, 1998.

[Collis99], J. C. Collis, L. C. Lee, D. T. Ndumu, H. S. Nwana, *{ZEUS}: a toolkit and approach for building distributed multi-agent systems*, ACM Press, p. 360-361, 1999.

[David92] R. David, H. Alla, *Petri Nets and Grafcet - Tools for modelling discrete event systems*, Prentice Hall, 1992.

[DECAF INTRO], F. McGeaty, *DECAF Programming : An Introduction*, Avril 2001.

[Decker], K. Decker, F. McGeary, *DECAF Programming: Agents for Undergraduates.*

[Delisle02] Delisle S., Sabas A., Badri M. *A Comparative Analysis of Multiagent System Development Methodologies: Towards a Unified Approach*, Third International Symposium "From Agent Theory to Agent Implementation" (AT2AI-3), Sixteenth European Meeting on Cybernetics and Systems Research, Vienne (Autriche), 2-5 avril 2002, Volume 2, 599-604.

[DeLoach01], S. A. DeLoach, M. Wood, *Developing Multiagent Systems with agentTool*, (ATAL'2000), Berlin, 2001.

[DeLoach], S. A. DeLoach, *Analysis and Design using MaSE and agentTool.*

[d'Iverno97], M. d'Inverno, D. Kinny, M. Luck, M. Wooldridge, *A Formal Specification of dMARS*, Agent Theories, Architectures, and Languages, p. 155-176, 1997.

[Crnogorac96], L. Crnogorac, A. S. Rao, K. Ramamohanarao, *Inheritance by Extensions and Restrictions in Agent Systems*, No 96/22, 1996.

[Elammari99], M. Elammari, W. Lalonde, *An agent-oriented methodology: High-level and intermediate models*, Agent-Oriented Information Systems, 1999.

[Ferber97], J. Ferber, O. Gutknecht, *Aalaadin : a meta-model for the analysis and design of organizations and multi-agent systems*, rapport de recherché présenté à l'Université Montpellier, 1997.

[Ferber97b], J. Ferber, O. Gutknecht, *MadKit : Organizing heterogeneity with groups in a platform for multiple multi-agent systems*, rapport de recherche présenté à l'Université Montpellier, 1997.

[Ferber99], J. Ferber, *Multi-Agent Systems : An Introduction to Distributed Artificial Intelligence*, Addison-Wesley, 1999.

[Finin93], T. Finin, G. Wiederhold, A*n Overview of KQML: A Knowledge Query and Manipulation Language,* Department of Computer Science, Stanford University, 1993.

[Finin94], T. Finin, R. Fritzon, D. McKay, R. McEntire, *KQML as an agent communication language*, In Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), ACM Press.

[Finin94b], T. Finin, R. Fritzon, D. McKay, R. McEntire. *KQML - A Language and Protocol for Knowledge and Information Exchange* (Technical Report No. CS-94-02). University of Maryland, Department of Computer Science.

[Finin94c], T. Finin, J. Weber, G. Wiederhold, M. Genesereth, R. Fritzon, J. McGuire, S. Shapiro, D. McKay, R. Pelavin, C. Beck, *Specification of the KQML Agent-Communication Language plus example agent policies and architectures* (DRAFT Report), DARPA Knowledge Sharing Initiative External Interface Working Group.

[Finin93] F. Finin, *Specification of the KQML Agent Communication Language,* The DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993.

[Ferber97c], J. Ferber, O. Gutknecht, *The MadKit agent platform architecture.*

[FIPA97] Foundation for Intelligent Physical Agents, *FIPA 97 Specification Part 1 Agent Management*, Specification Oct 10, 1997.

[Franklin96], S. Franklin, A. Graesser, *Is it an Agent, or Just a Program?: A Taxonomy for Autonomous Agents*, Agent Theories, Architectures, and Languages, p. 21-35, 1996.

[Galan], A. K. Galan, A. D. Beker, *Multi-Agent Communication in JAFMAS.*

[Galan00], A. K. Galan, *JiVE : JAFMAS integrated Visual Environment*, Thèse présentés à l'Université de Cincinnati, 2000.

[Garneau02], T. Garneau. S. Delisle. *Programmation orientée agent : évaluation comparative d'outils et environnements*, Actes des Journées Francophones pour l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'02), Lille (France), Octobre 28-30 2002. *Systèmes multi-agents et systèmes complexes (ingénierie, résolution de problèmes et simulation)* JFIADSMA'02, Hermes Sciences, 2002, 111-123.

[Garneau03], T. Garneau, S. Delisle. *A new general, flexible and Java-based software development tool for multiagent systems,* The 2003 International Conference on Information Systems and Engineering (ISE 2003), july 20-25 2003.

[Georgeff87], M. P. Georgeff, A. L. Lansky. *Reactive reasoning and planning.* In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), pages 677–682, Seattle,WA, 1987.

[Graham], J. R. Graham, D. Mchugh, F. McGeary, M. V. Windley, D. Cleaver, K. S. Decker, *A Programming and Execution Environment for Distributed Multi Agent Systems*.

[Graham00], J. R. Graham, D. Mchugh, M. Mersic, F. McGeary, M. V. Windley, D. Cleaver, K. S. Decker, *Tools for Developing and Monitoring Agents in Distributed Multi-Agent Systems*, Agents Workshop on Infrastructure for Multi-Agent Systems, p. 12-27, 2000.

[Graham00b], J. R. Graham, *Real-Time Scheduling for Distributed Agents*.

[Gutknecht01], O. Gutknecht, J. Ferber, F. Michel, *Integrating tools and infrastructures for generic multi-agent systems*.

[Hassoun95], M. H. Hassoun, *Fundamentals of artificial neural networks*, MIT Press, 1995.

[Hindriks00], K. Hindriks, M. d'Inverno, M. Luck, *Architecture for agent programming languages*, In ECAI 2000: Proceedings of the Fourteenth European Conference on Artificial Intelligence, 2000.

[Horling98], B. C. Horling, *A Reusable Component Architecture for Agent Construction*, KEYWORD:: Agent framework, No UM-CS-1998-049, 1998.

[Hui99], P. M. D. Gray, K.Hui, A. D. Preece, *Finding and moving constraints in cyberspace*, IEEE Press intelligent agents in cyberspace p. 121, mars 1999.

[Iglesias99], C. Iglesias, M. Garrijo, J. Gonzalez, *A Survey of Agent-Oriented Methodologies*, Proceedings of the 5th International Workshop on Intelligent Agents {V} : Agent Theories, Architectures, and Languages ({ATAL}-98), Vol 1555, p. 317-330, 1999.

[Jack D. E. U.G.], Jack Development Environment, User Guide, 2001.

[Jack U.G.], Jack Intelligent Agents, User Guide, 2001.

[JADE A.G.], F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, JADE, Administrator's Guide, Janvier 2002.

[JADE P.G.], F. Bellifemine, G. Caire, T. Trucco, G. Rimassa, JADE, Programmer's Guide, Février 2002.

[JADE J.T.], G. Caire, *JADE TUTORIAL APPLICATION-DEFINED CONTENT LANGUAGES AND ONTOLOGIES*, Février 2002.

[Kendall], E. A. Kendall, P. V. Krishna, C. V. Pathak, C.B. Suresh, *A Java Application Framework for Agent Based Systems*.

[Klusch99], M. Klusch, *Intelligent Information agents : Agent-based information discovery and management on the internet*, Springer preface, 1998.

[Lee98], L. C. Lee, D. T. Ndumu, H. S. Nwana, *ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems*, An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. In Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems, p. 377-392, Londre, 1998.

[Lee98b], L. C. Lee, D. T. Ndumu, H. S. Nwana, *The ZEUS agent building tool-kit*, BT Technology Journal, Vol 16, No 3, Juillet 1998.

[Maes95], P. Maes, *Artificial Intelligence meets Entertainment: Lifelike Autonomous Agents*, Communications of the ACM 38 (11), pp. 108-114, Nov. 1995.

[Mangina02] E. Mangina, *Review of software products for Multi-agent Systems,* Applied intelligence (UK) Ltd for AgentLink, June 2002.

[Martin99], D. Martin, A. Cheyer, D. Moran, *The Open Agent Architecture: a framework for building distributed software systems*, Applied Artificial Intelligence, Vol 13, No ½, p. 91-128, 1999.

[Miles00], S. Miles, M. Joy, M. Luck, *Designing Agent-Oriented Systems by Analysing Agent Interactions*, AOSE, p. 171-184, 2000.

[Mylopoulos], J. Mylopoulos, M. Kolp, P. Giorgini, *Agent-Oriented Software development*.

[Nwana96], H. S. Nwana. *Software Agents: An Overview. Knowledge Engineering Review.* 1996.

[Pitt97], J. Pitt, F. Bellifemine, *A Protocol-BasedSemantics for FIPA'97ACL and its Implementation in JADE.*

[Rao95], A. S. Rao, M.P. Georgeff, *BDI Agents: From Theory to Practice*, In Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, USA, pp. 312-319, 1995.

[Russel99] S.J. Russell, P. Norvig, *Artificial Intelligence: A Modern Approach.* Englewood Cliffs, NJ: Prentice Hall, 1995.

[Ricordel00], P.-M. Ricordel, Y. Demazeau, *From Analysis to Deployment : a Multi-Agent Platform Survey*, ESAW, p. 93-105, 2000.

[Rimassa00], G. Rimassa, A. Poggi, P. Turci, *An Object-Oriented Framework to Realize Agent Systems*, WOA2000 Workshop, p. 52-57, Parma, Mai 2000.

[Rimassa99], JADE - A FIPA – *compliant agent framework*, PMAA`99, p. 97-108, Londre, Avril 1999.

[Rimassa], G. Rimassa, A. Poggi, M. Tomajuolo, *Multi-User and Security Support for Multi-Agent Systems*.

[Shohan93], Y. Shohan, *Agent-Oriented Programming*, Artificial Intelligence, Vol. 60, No. 1, p. 51-92 Mars 1993.

[Shohan93b] Y. Shoham, *AGENT-0: A simple agent language and its interpreter.* In Proceedings of the Ninth National Conference on Artificial Intelligence, Vol II. (pp. 704 - 709). Anaheim, CA: MIT Press, 1993

[Sichman98], J. Sichman, R. Conte, N. Gilbert, *Multi-agent systems and agent-based simulation*, Lecture notes in artificial intelligence, 1998.

[Sim99], W. T. J. Sim, C. H. Chi, *Framework for an agent-based electronic market place: design and implementation*, IEEE Press intelligent agents in cyberspace p. 164, mars 1999.

[Sycara], K. Sycara, Massimo Paolucci, M. V. Velsen, J. Giampapa, *The RETSINA MAS Infrastructure*.

[Thomas93], S. R. *PLACA, An Agent Oriented Programming Language*. PhD Thesis, Stanford University, 1993.

[Tveit01], A. Tveit, *A survey of Agent-Oriented Software Engineering*, 2001.

[Wooldridge00], M. Wooldridge, P. Ciancarini, *Agent-Oriented Software Engineering: The State of the Art*, AOSE, p. 1-28, 2000.

[Zambonelli00], F. Zambonelli, N. Jennings, A. Omicini, M. Wooldridge, *Agent-Oriented Software Engineering for Internet Applications*, Coordination of Internet Agents: Models, Technologies and Applications, Springer, 2000.

[ZEUS A.R.G.], J. Collis, D. Ndumu, *The Zeus Agent Building Toolkit*, The Application Realisation Guide, Septembre 1999.

[ZEUS R. M..G.], J. Collis, D. Ndumu, *The Zeus Agent Building Toolkit*, The Role modelling Guide, Août 1999.

[ZEUS R.G.], J. Collis, D. Ndumu, *The Zeus Agent Building Toolkit*, The Runtime Guide, Novembre 1999.

[ZEUS T.M.], J. Collis, D. Ndumu, *The Zeus Agent Building Toolkit*, Technical Manual, Septembre 1999.

[Zunino], A. Zunino, A. Amandi, *Building Multi-Agent Systems From Reusable Software Components*.

[Zunino00], A. Zunino, A. Amandi, *Brainstorm/J: a Java Framework for Intelligent Agents*, Argentinian Symposium on Artificial Intelligence, 2000.

10. Sites webs

<ADK>	http://www.tryllian.com
<AgentBuilder>	http://www.agentbuilder.com/
<AgentTool>	http://en.afit.af.mil/ai/agentool.htm
<Brainstorm/J>	http://www.exa.unicen.edu.ar/~azunino/brainstormj.html
<CABLE>	http://public.logica.com/~grace/Architecture/Cable/public
<Comet Way JAK>	http://www.cometway.com
<Cougaar>	http://www.cougaar.org
<DECAF>	http://www.eecis.udel.edu/~decaf/
<DESIRE>	http://www.cs.vu.nl/vakgroepen/ai/projects/desire
<FIPA OS>	http://fipa-os.sourceforge.net/index.htm
<Jack>	http://www.agent-software.com.au/shared/products/index.html
<Jade>	http://sharon.cselt.it/projects/jade/
<JAFMAS>	http://www.ececs.uc.edu/~abaker/JAFMAS/Home.html
<Java>	http://java.sun.com
<JiVE>	http://www.ececs.uc.edu/~abaker/JiVE/index.html
<JESS>	http://herzberg.ca.sandia.gov/jess
<MadKit>	http://www.madkit.org/
<MASSYVE Kit>	http://www.gsigma-grucon.ufsc.br/massyve/mkt.htm
<SIM_AGENT>	http://www.cs.bham.ac.uk/_7Eaxs/cogaff/simagent.html
<Zeus>	http://www.labs.bt.com/projects/agents/zeus/