MIG-004-112-1

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université Saad Dahlab, Blida
USDB.

Faculté des sciences.
Département informatique.

**Mémoire pour l'obtention**
**du diplôme d'ingénieur d'état en informatique.**
Option : I.A

Sujet :

# Etude Pour la Réalisation d'un Framework pour agents mobiles intelligents

**Présenté par :Sidoumou Mohamed Rédha** **Promoteur :Dr.Mahieddine Mohamed**

**Organisme d'accueil : LDRSI.**

**Soutenue le:18/07/2006, devant le jury composé de :**
**Président:Prof.Guessoum**
**Examinateur:Mr.Mazari**

**- 2005/2006 -**

MIG-004-112-1

# RESUME

Les systèmes multi-agents sont devenu très populaires et sont utilisé pour résoudre des problèmes à grande complexité. Ils utilisent plusieurs entités coopérantes pour atteindre un but final.

Nous avons créé auparavant des objets simple et maintenant nous voulons réutiliser ces objets, en les dotons de mobilité, communication et intelligence, d'où viennent les notions de 'Object' et 'True Object'. Une telle approche aurait un gain de temps et d'argents. C'est le but de notre Framework.

Notre travail est de créer un Framework à bases de patterns facilitant la création de système multi-agents intelligents en java, en prenant en compte l'interaction et la mobilité, pour permettre la création de système et une réutilisation de code assez efficace.

# REMERCIEMENTS

Je tiens à remercier,
Dieu pour nous avoir donner la capacité à faire ce travail.

Mon promoteur, Dr Mahieddine qui m'a énormément aidé

Le président du jury le prof Guessoum

Les membres du jury

Tous membres du Laboratoire LRDSI et particulièrement Mazari Rédha qui m a beaucoup aidés

Le chef département d'informatique, Mr Messiad

Tous mes professeurs

Mohamed,Merouane, Farouk, Alilo, Mounir, Djamel, HeyHey, Omar, Mireille, Sophia,
Ainsi que tous mes amis,

**Dédicaces :**

A mes parents

A mes frères et sœurs

A toute ma famille

A tout mes amis

## Cadre du travail,

Cette thèse a été effectuée à l'Université de Blida au sein de l'équipe *GLODOO* du laboratoire LRDSI.

L'équipe *GLODOO* a été formée par le Dr Mahieddine pour adresser la problématique générale des *applications orientées objets*.

Cette équipe s'intéresse aux nouvelles applications influencées par les avancés technologiques comme les mobiles, et d'autre part, les nouvelles technologies de constructions de logiciels.

L'objectif est l'étude et de la conception de services communs permettant de faciliter le développement d'applications des nouvelles technologies à base des nouvelles techniques de construction de logiciels.

Les axes de recherches de l'équipe sont :

- Le modèle de programmation par objets,
- Les patterns et les Frameworks pour la construction de logiciels appliqués aux nouvelles technologies (téléphones cellulaires, systèmes embarqués, « objets domestiques » intelligents, robotique, ...).
- La programmation des objets, acteurs, agents et logiciels (comme les jeux) intelligents,
- Le modèle de programmation par agents mobiles.

# SOMMAIRE

| Titre | Page |
| --- | --- |
| **Introduction** | 10 |
| **Chapitre 1 : Agent et multi-agents** | 12 |
| 1-1-Définition | 12 |
| 1-2-Système multi-agents | 13 |
| 1-2-1-Définition | 13 |
| 1-2-2-SMA ouvert | 14 |
| 1-2-3-SMA homogène/hétérogène | 14 |
| 1-2-4-L'émergence | 15 |
| 1-2-5-La décomposition Multi-agents | 16 |
| 1-2-5-1-Agent (A) | 16 |
| 1-2-5-2-Environnement (E) | 16 |
| 1-2-5-3-Interaction (I) | 17 |
| 1-2-6-Problèmes à résoudre | 17 |
| 1-2-7-Agents réactifs | 17 |
| 1-3-Liens avec d'autres disciplines | 18 |
| 1-4-Exemples d'applications | 19 |
| 1-5-Un peu d'histoire | 20 |
| **Chapitre 2 : Les patterns et les Frameworks** | 21 |
| 2-1-Les patterns | 21 |
| 2-1-1-Introduction | 21 |
| 2-1-2-Définition | 21 |
| 2-1-3-Classification des patterns de conception | 21 |
| 2-1-4-Exemple de pattern de conception | 22 |
| 2-1-4-1-L'architecture MVC | 22 |
| 2-1-4-2-Le pattern observer | 23 |
| 2-2-Les Frameworks | 24 |

2-2-1-Definition 24
2-2-2 Les Framework 25
2-2-3-Pourquoi les utilisés 25
2-2-4-Type de Framework et utilisation 25
2-2-5-Framework Blanc 26
2-2-6-Famework noir 26
2-2-7-Instanciation 27
2-2-8-Quelque exemple de l'approche multi-agents 27
2-2-8-1-Le système des Aglets 28
2-2-8-2-Tacoma 28
**Chapitre 3 : Le langage Java** 29
3-1-Introduction 29
3-2-Le langage JAVA 30
3-3-Fonction du langage java 30
3-3-1-Type de données 30
3-3-2-Objet, classes, méthodes 31
3-4-D'autres fonctions du langage java 31
3-4-1-Java.lang 31
3-4-2-Java.lang.reflect 31
3-4-3- Java.io 32
3-4-4-Java.util 32
3-4-5-Java.util.zip 33
3-4-6-Java.net 33
3-5-Utilisation de java pour les agents 34
3-5-1-Autonomie 34
3-5-2-Implémentation du pattern Observer 35
**Chapitre 4 : La mobilité sous java** 36
4-1-Introduction 36
4-2-La mobilité et les agents 36
4-3-La communication par socket en java 38
4-3-1-Java.net.InetAdress 39

4-3-2-La classe serverSocket 39
4-3-3-La classe java.net.Socket 40
4-3-3-Opération de la classe Socket 41
4-4-Le classLoader 41
4-4-1-Structure du classLoader 42
4-4-2-La méthode defineClass 43
4-4-3-La méthode findsystemClass 43
4-4-4-Méthode resolveClass 44
4-5-Le ByteCode Java 45
4-5-1-Format du fichier .class 45
4-5-2-Méthodes et champs 46
4-5-3-L'attribut de code 46
4-5-4-L'instruction JVM 46
4-5-5-Code semi interprété 47
**Chapitre 5: Système de raisonnement** 48
5-1-Le raisonnement à base de règles 48
5-2-Le chaînage avant 50
5-2-1-Avantages 51
5-2-2-Inconvénients 51
**Chapitre 6: la conception de notre Framework(AgentTo)** 52
6-1-Interaction et communication 52
6-1-1-Les mondes 52
6-1-1-1-Le MVC 53
6-1-1-2-Comment ça marche ? 54
6-1-2-Les agents 57
6-1-3-Manière de travail 59
6-1-4-La communication 60
6-2-Le raisonneur 61
6-2-1-Les règles 62
6-2-2-Les clauses 63
6-2-3-Le raisonneur 65

6-3-La mobilité des agents 68

6-3-1-Le serveur du monde 68

6-3-2-Le classLoader 71

6-3-3-Le choix des classes 72

6-3-4-Le byteCode java 72

6-3-5-L'envoi de message entre différents mondes 77

**Chapitre 7 : Comment utiliser AgentTo ?** 78

7-1-Manière de travail 78

7-2-L'exemple agentHomme 79

7-3-Sortie d'un exemple 82

7-4-Comparaison par rapport aux autres plate-formes 83

**Conclusion Générale** 84

**Référence Bibliographique** 85

## TABLES DES FIGURES


| Numéro de la figure | Description | page |
|---|---|---|
| Fig.1 | Liens avec d'autres disciplines | 18 |
| Fig.2 | L'architecture MVC | 22 |
| Fig.3 | Diagramme de classe du pattern observer | 23 |
| Fig.4 | Schéma d'une communication client/serveur | 38 |
| Fig.5 | Structure d'un fichier .class | 45 |
| Fig.6 | Diagramme de classe de AgentTo | 52 |
| Fig.7 | Le MVC | 54 |
| Fig.8 | L'application du MVC dans AgentTo | 55 |
| Fig.9 | Disposition des mondes et des agents | 57 |
| Fig.10 | Diagramme de classe du raisonneur | 62 |
| Fig.11 | La clause | 64 |
| Fig.12 | Le pattern Observer dans la clause | 65 |
| Fig.13 | La dépendance des classes | 75 |
| Fig.14 | Diagramme de class de l'agent Homme | 80 |

# INTRODUCTION

*Les parties du monde ont toutes un tel rapport et un tel enchaînement l'une avec l'autre que je crois impossible de connaître l'une sans l'autre et sans le tout. (Blaise Pascal)*

La question de faire des objets intelligents a toujours été posée. Surtout qu'il existe des tas d'objets classiques possédant beaucoup de fonctionnalité. Ça serai très intéressant de réutiliser ces objets et de les rendre intelligents, autonomes, coopérants,...

C'est pourquoi nous avons eu l'idée de crée ce Framework.

Notre projet consiste à construire un Framework à base de patterns pour la création des systèmes multi-agents mobiles en java. Il doit pouvoir crée un système ouvert, capable de transformer les 'Object' en 'True Object'. Ce Framework comporte trois parties:

-L'interaction qui est la manière dont interagissent les agents et aussi la manière dont ils sont organisés.

-L'intelligence qui est la façon de raisonnement des agents.

-La mobilité qui est la migration des agents dans un environnement répartie. Cette dernière apporte un bon gain de temps dans le cas de grosses ressources qui se trouve dans un autre terminal.

Ce Framework doit d'être assez général et assez simple.

Notre travail a été mené en deux phases:

1) La partie apprentissage.

2) La partie implémentation.

Nous verrons que les agents présentent au niveau collectif des capacités supérieures à la somme des capacités individuelles.

Nous allons suivre le plan suivant:

**Chapitre1 :** Nous définissons en premier lieu le terme agent et le terme multi-agents. Nous verrons les différentes approches de leur utilisation ainsi que quelques exemples.

**Chapitre2 :** Nous définissons les patterns dans le cas générale, les patterns que nous avons utilisés, leur but, approfondir notre connaissance sur les Framework, le pourquoi de leur utilisation. Nous verrons aussi quelques exemples.

**Chapitre3 :** Nous présentons dans ce chapitre les causes du choix du langage java. Nous présentons les points forts pour la construction d'agents dans un système hétérogène.

**Chapitre4 :** Nous verrons dans ce chapitre comment la mobilité des classes et des processus est mise en œuvre en utilisant le langage java. Nous verrons aussi les socket, les stream, le classloader et le byte code java.

**Chapitre5 :** Nous abordons ici le raisonnement à base de règles, son choix et le chaînage avant.

**Chapitre6 :** Nous introduirons notre Framework en détails en expliquant les interactions, le raisonnement et la mobilité.

**Chapitre7 :** Nous expliquerons la façon d'utiliser notre Framework et son utilité. Nous donnerons aussi un exemple concret.

# Chapitre 1 : agents et multi-agents

## 1.1. Définitions :

Ce chapitre a pour objectif d'introduire les systèmes multi-agents qui constituent le fondement de notre travail. Nous nous intéressons tout d'abord aux entités qui composent cette catégorie de systèmes: les agents. Cette notion définie, nous nous intéresserons alors à la caractérisation des systèmes multi-agents.

Depuis une dizaine d'années, les systèmes multi-agents ont connu un grand essor et sont appliqués à des domaines très variés comme, par exemple, le domaine de la simulation et de la vie artificielle, la robotique et le traitement d'images.

Les systèmes multi-agents sont issus de l'intelligence artificielle distribuée (IAD), une branche de l'intelligence artificielle qui s'articule autour de trois axes:

1) résolution distribuée des problèmes qui s'intéresse à la manière de diviser un problème en un ensemble d'entités distribuées et coopérantes et à la manière de partager la connaissance du problème afin d'en obtenir la solution.

2) l'intelligence artificielle parallèle qui développe des langages et des algorithmes parallèles pour l'intelligence artificielle (IA) visant ainsi l'amélioration des performances des systèmes d'IA.

3) Les systèmes multi-agents qui privilégient une approche décentralisée de la modélisation et mettent l'accent sur les aspects collectifs des systèmes.

Les systèmes multi-agents, comme leur nom l'indique, sont des systèmes constitués de plusieurs agents qui évoluent et collaborent dans un environnement commun. Il est donc nécessaire de commencer par définir ce que l'on appelle un agent. La notion d'agent définie, nous introduirons le concept de système multi-agents.

La notion d'agent est utilisée dans beaucoup de domaines: sociologie, biologie, psychologie cognitive, psychologie sociale, informatique

Pourquoi des agents? Que représentent-ils pour l'informatique?

Nous apportent-ils quelque chose de nouveau dans la modélisation et la conception?

Y a-t-il une différence entre les agents logiciels et les autres logiciels?

Agent – il n'y a pas une définition acceptée à l'unanimité.

Un agent est une entité qui **perçoit son environnement et agisse** sur celui-ci" (Russell, 1997). Un agent est un système informatique, **situé** dans un environnement, et qui agit d'une façon **autonome** pour atteindre les **objectifs** (buts) pour lesquels il a été conçu (Wooldrige et Jennings, 1995).

Les agents intelligents sont des entités logiciels qui *réalisent des opérations à la place d'un utilisateur* ou d'un autre programme, avec une sorte d'indépendance ou **d'autonomie**, et pour faire cela ils utilisent une sorte de connaissance ou de représentation des buts ou des désires de l'utilisateur.(L'agent IBM);

Un agent est une entité qui fonctionne continuellement et de manière **autonome** dans un environnement où **d'autres processus** se déroulent et **d'autres agents** existent. (Shoham, 1993)

Un agent est **une entité autonome**, réelle ou abstraite, qui est capable **d'agir** sur elle-même et sur son environnement, qui, dans un **univers multi-agents**, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents (Ferber, 1995).

Les Caractéristiques des agents sont les suivantes:

a) **Situé** – l'agent est capable d'agir sur son environnement à partir des entrées sensorielles qu'il reçoit de ce même environnement;
b) **Autonome** – l'agent est capable d'agir sans l'intervention d'un tiers (humain ou agent) et contrôle ses propres actions ainsi que son état interne
c) **Proactif** – l'agent doit exhiber un comportement proactif et opportuniste, tout en étant capable de prendre l'initiative au bon moment;
d) **Capable de répondre à temps** – l'agent doit être capable de percevoir son environnement et d'élaborer une réponse dans le temps requis
e) **Social** – l'agent doit entre capable d'interagir avec des autres agents (Logiciels ou humains) afin d'accomplir des taches ou aider ces agents à accomplir les leurs.

## 1.2. Systèmes multi-agents :

### 1.2.1. Définitions :

Un système multi-agents est un ensemble d'agents qui évoluent dans un environnement commun. Dans [Weiss, 1999]. Gerhard Weiss définit l'intelligence artificielle distribuée comme étant l'étude, la conception et la réalisation de systèmes multi-agents qu'il présente comme étant

des systèmes dans lesquels des agents intelligents interagissent et poursuivent un ensemble de buts ou réalisent un ensemble d'actions.

On peut donc se demander comment, à partir d'un ensemble d'agents pas forcément intelligents, on arrive à un système complexe où semble se dégager de l'intelligence. Dans un système multi-agents, l'intelligence provient de l'émergence d'un comportement global [Gasser, 1992] comme nous le verrons dans le paragraphe suivant.

Les systèmes multi-agents, depuis leur création, font l'objet d'un véritable engouement. Les raisons exprimées dans [Bourdon, 2001] indiquent qu'il ne s'agit pas simplement d'un phénomène passager. En effet, l'importance de l'approche de *systèmes multi-agents* croît considérablement et s'explique essentiellement par:

a) Le fait que les systèmes d'information soient de plus en plus distribués, ouverts, à grande échelle et hétérogènes. Cela rend les interconnexions tellement compliquées et croissantes qu'elles dépassent la compréhension globale que peut en avoir un être humain.
b) Le fait que cette approche soit utile pour expérimenter des réflexions sociologiques et psychologiques pour ce qui concerne par exemple les relations entre personnes dans les sociétés modernes.

### 1.2.2. Système multi-agents ouvert :

Un système multi-agents *ouvert* [Vercouter, 2000] partage les caractéristiques des systèmes ouverts. Pour ce qui concerne les entités élémentaires du système que sont les agents, ils n'ont pas la possibilité d'avoir une représentation complète de l'environnement. Pour ce qui est du système dans sa globalité, il doit être modulaire et extensible. La modularité concerne le fait que le système multi-agents est composé de plusieurs sous-systèmes mis en relation. Ces sous-systèmes ont chacun leur propre mode de fonctionnement. L'extensibilité se traduit par le fait que le système multi-agents supporte l'ajout et le retrait dynamique d'éléments. A l'inverse, le qualificatif *fermé* signifie que l'ensemble des agents qui compose le système reste le même.

### 1.2.3. Système multi-agents homogène/hétérogène:

Un système multi-agents *homogène* est composé d'agents homogènes. Deux agents ont cette particularité s'ils sont identiques du point de vue de leur modèle et de leur architecture. Le

qualificatif *hétérogène* est utilisé pour préciser que le système multi-agents est composé d'agents différents du point de vue de leurs modèles et de leurs architectures.

#### 1.2.4. L'émergence:

L'émergence traite de l'apparition soudaine non programmée et irréversible de phénomènes dans un système (éventuellement multi-agents) confirmant ainsi la maxime grecque "le tout est plus que la somme de toutes les parties".

Comme nous l'avons constaté précédemment, un système multi-agents est un ensemble d'agents situés dans un même environnement et qui interagissent. De ces interactions peuvent émerger différents phénomènes que l'on peut classer en trois catégories [Marcenac, 1996] : l'émergence structurelle, l'émergence de comportements et l'émergence de propriétés. L'émergence traite donc de l'apparition soudaine non programmée et irréversible de phénomènes dans un système (éventuellement multi-agents).

Il est difficile de qualifier la caractéristique émergente d'un phénomène, cependant Jean-Pierre Müller en propose une définition intéressante [Müller and Van Dyke Parunak, 1998]. S'inscrivant dans le prolongement des travaux relatés dans [Lenay, 1996, Jean, 1997], il affirme qu'un phénomène est émergent si on a un ensemble d'agents interagissant via un environnement dont l'état et la dynamique ne peuvent pas être exprimés dans les termes du phénomène émergeant à produire mais dans un vocabulaire ou une théorie D. La dynamique des agents en interaction produit un phénomène global comme par exemple une trace d'exécution ou un invariant. Le phénomène global est observable soit par les agents (sens fort) soit par un observateur extérieur (sens faible) en des termes distincts de la dynamique sous-jacente D c'est-à-dire un autre vocabulaire ou une théorie D'.

Pour fournir à un système multi-agents une fonctionnalité particulière, on peut donc avoir recours à une autre méthode que celle utilisée traditionnellement et qui consiste à procéder à une décomposition fonctionnelle du problème en un ensemble de primitives qui seront implantées dans les agents. L'alternative proposée par Luc Steels [Steels, 1990] a pour objectif de faire émerger cette fonctionnalité à partir des interactions entre les agents (ou sous-systèmes multi-agents dans le cadre de système multi-agents récursif, c'est-à-dire composé de plusieurs systèmes multi-agents).

### 1.2.5. La décomposition multi-agents:

La décomposition Voyelles d'un système multi-agents a été introduite par Yves Demazeau [Demazeau, 1995]. La décomposition Voyelles identifie un axe Agent, un axe Environnement, des Interactions et une structure Organisationnelle explicite ou non.

SMA=Agent+Environemnt+Interation+Organisation (2.1).

De plus l'intelligence d'un système multi-agents étant plus que la somme de l'intelligence des agents qui la compose, un principe a été énoncé pour rendre compte du surcroît d'intelligence observé, de l'intelligente émergente.

Enfin, un système multi-agents peut être vu comme un agent d'un système multi-agents global. Il s'agit du principe de la récursivité. Récursion: A=A_elmentaire=SMA (2.3)

#### 1.2.5.1. Les agents (A):

Le concept d'agent a été défini.

Cette partie de l'analyse AEIO regroupe les éléments nécessaires à la construction des agents à savoir leur modèle, leur architecture, leur implémentation...

#### 1.2.5.2. L'environnement (E):

Dans un système multi-agents, on appelle *environnement* l'espace commun aux agents du système. Un environnement peut être [Russell and Norvig, 1995, Wooldridge et al, 2000]: *Accessible* si un agent peut, à l'aide des primitives de perception, déterminer l'état de l'environnement et ainsi procéder, par exemple, à une action. Si l'environnement est *inaccessible* alors il faut que l'agent soit doté de moyens de mémorisation afin d'enregistrer les modifications qui sont intervenues. L'environnement peut-être:

a) *Déterministe*, ou non, selon que l'état futur de l'environnement ne soit, ou non, fixé que par son état courant et les actions de l'agent.
b) *Episodique* si le prochain état de l'environnement ne dépend pas des actions réalisées par les agents.
c) *Statique* si l'état de l'environnement est stable (ne change pas) pendant que l'agent réfléchit. Dans le cas contraire, il sera qualifié de *dynamique*.
d) *Discret* si le nombre des actions faisables et des états de l'environnement est fini.

Cette partie de l'analyse consiste donc à trouver les éléments nécessaires à la réalisation des interactions extérieures au système comme par exemple la perception de cet environnement, les actions que l'on peut y faire.

#### 1.2.5.3. Les interactions (I):

Les interactions proviennent de la mise en relation dynamique de plusieurs agents par le biais d'un ensemble d'actions réciproques. Il existe plusieurs types d'interactions, qui dépendent de trois paramètres que sont les buts, les ressources et les compétences comme l'illustre le tableau référencé 2.1 tiré de l'ouvrage de Jacques Ferber [Ferber, 1995]. Cet axe de l'analyse AEIO comprend donc les éléments qui servent à la structuration des interactions externes entre les agents (langage d'interaction, protocole de communications...).

Le concept d'organisation est vaste mais on peut tout de même le définir comme étant une structure décrivant les interactions et autres relations qui existent (dans le but d'assouvir un objectif commun) entre les membres de la dite organisation [Fox, 1981]. L'organisation peut donc apparaître comme une structure de coordination et de communication [Malone, 1987]. Jacques Ferber [Ferber, 1995] va dans ce sens en affirmant que les organisations constituent à la fois le support et la manière dont se passent les inter-relations entre les agents, c'est-à-dire dont sont réparties les tâches, les informations, les ressources et la coordination d'actions. Il précise que ce qui rend l'organisation si difficile à cerner est qu'elle est à la fois le processus d'élaboration d'une structure et le résultat de ce processus.

L'axe O de la méthode d'analyse AEIO comprend donc les éléments permettant d'ordonner les ensembles d'agents en des organisations par la détermination des rôles des agents.

### 1.2.6. Problèmes à résoudre:

Comment construire des agents logiciels qui sont capable d'avoir un comportement indépendant, autonome pour accomplir leurs buts. Comment construire des agents logiciels qui sont capable d'interagir (coopération, coordination, négociation) avec d'autres agents pour accomplir leurs taches, en particulier dans la situation où les autres agents n'ont pas les mêmes buts.

**La conception des agents=La conception de la société ??**

### 1.2.7. Agents réactifs:

Composantes très simples qui perçoit l'environnement et sont capable d'agir sur celui-ci. Ils n'ont pas une représentation symbolique de l'environnement, des connaissances. L'intelligence est distribuée entre beaucoup d'agents réactifs. Le comportement intelligent devrait émerger de l'interaction entre ces agents réactifs et l'environnement.

Comment modéliser le problème avec des agents réactifs?

Comment modéliser le problème avec des agents cognitifs?

### 1.3. Liens avec d'autres disciplines:

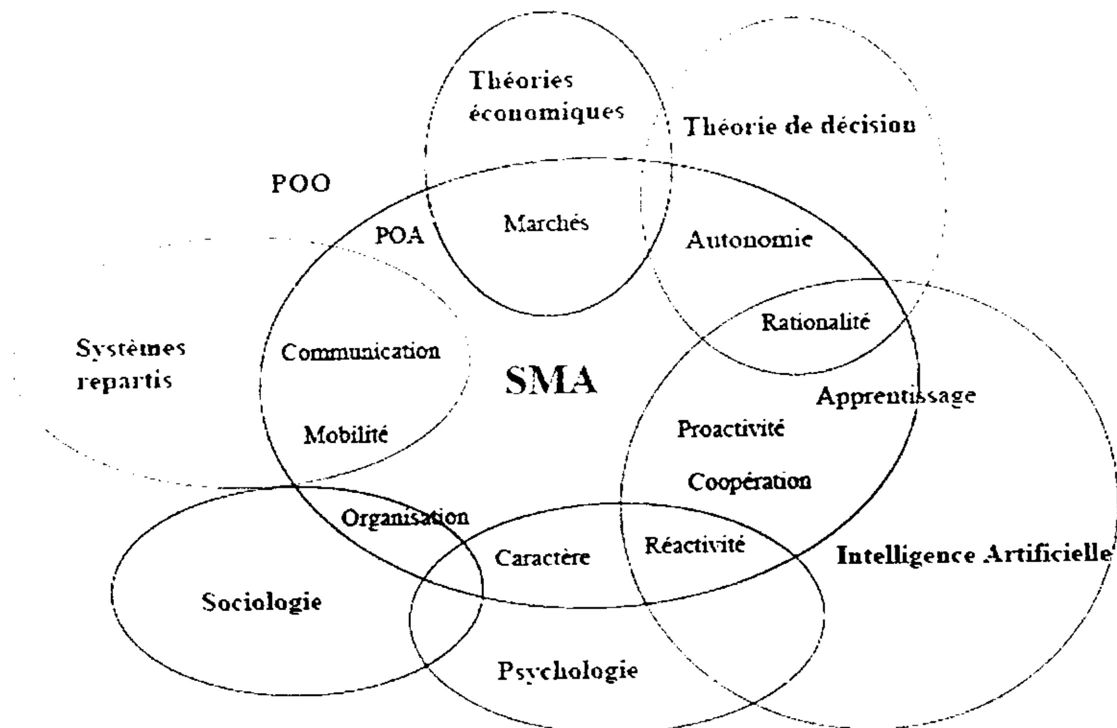



Fig.1: liens avec d'autre discipline

Est-ce que les agents nous apportent quelque chose de nouveau?

Comparaison des agents avec d'autres techniques:

**a) Agents vs Systèmes repartis/concurrents:**

Autonomie - Les structures de synchronisation et de coordination ne sont pas fixées → nécessités des mécanismes dynamiques

Agents individualistes - On ne peut pas supposer qu'ils sont toujours désireux à coopérer

**b) Agents vs Intelligence Artificielle:**

L'IA – surtout l'aspect de l'intelligence individuelle – un seul agent intelligent

SMA – l'aspect social, l'intelligence du comportement social, plusieurs agents intelligents

Est-ce que les agents nous apportent quelque chose de nouveau?

**c) Agents vs Objets :**

Autonomie – les agents ont le contrôle de leurs actions, ils peuvent refuser de coopérer

Les agents sont réactifs, comme les objets, mais aussi proactifs

Les agents sont d'habitude persistants et ils ont leurs propre "thread" de contrôle.

**d) Domaines de recherche:**

Architectures d'agents et des SMA

Représentation des connaissances: sur l'environnement, sur eux-mêmes, sur les autres agents.

Recherche distribuée de la solution.

Coordination.

Planification: partage des taches, partage des résultats, planifications distribuées.

Communication: langages, protocoles.

La prise des décisions: négociation, marchés, formation des coalitions.

Théories des organisations.

Apprentissage multi-agents.

Implémentation :

- Programmation orientée agents
- Langages spécialisés
- Platformes multi-agents
- Mobilité.

Sécurité et confiance.

Nous voyons que chaque année la somme investi dans le développement des agents accroît et beaucoup d'organisations ont optées pour cette approche.

### 1.4. Exemples d'applications :

Applications industrielles: contrôle en temps réel, production, réseaux de télécommunications, systèmes de transport, systèmes de distribution, etc.

Gestion de processus de business, support à la décision

Commerce électronique

Systèmes d'information coopératifs: découverte des sources, recherche de l'information, filtrage des informations, fusion des informations et personnalisation

Interaction homme-machine

Mondes virtuels

Divertissement

**1.5. Un peu d'histoire...**

Pour conclure ce chapitre, il est intéressant d'observer la place qu'occupent les systèmes multi-agents dans l'évolution des paradigmes de l'informatique tel que nous le voyons, du domaine des multi-agents. Il s'inspire de celui présenté dans l'ouvrage de génie logiciel multi-agents de Jürgen Lind [Lind, 2001] qui s'est arrêté à l'introduction du concept d'agent.

Un peu d'histoire...

Le *paradigme multi-agents*, où le collectif permet la résolution du problème..

Cette évolution des concepts est en fait très logique. Dans un premier temps l'objectif des personnes qui utilisaient l'outil informatique fut de simplifier la programmation : l'élaboration des langages dit évolués passa par l'agrégation d'instructions en procédures puis en fonctions lorsqu'elle retournait des résultats. Les besoins en dynamicité des applications puis en généricité ont conduit à l'utilisation de programmation structurée puis à introduire les objets (méthodes et langage de programmation). Les besoins en autonomie, induis par la répartition et la distribution des ressources a conduit à introduire la notion d'agent. Très peu de temps après, la coopération a amené les informaticiens à créer des systèmes à agents. La prise en compte d'objectif globaux et l'utilisation de l'émergence a conduit quant à elle à introduire le paradigme multi-agents.

Dans un système multi-agents, les agents offrent de la généricité de par leur découplage modèle/architecture/ implantation. Ce découplage permettra d'envisager plus simplement l'implantation sur des supports physiques variés.

La vision du collectif plutôt que de l'individu permet d'atteindre des objectifs globaux du système multi-agents dont il faudra garantir l'intégrité du fonctionnement global, notement dans le cas de système multi-agents ouvert.

# Chapitre 2 : Les Patterns et les Frameworks

## 2.1. Les patterns :

### 2.1.1. Introduction :

Nous présentons dans ce chapitre les concepts et les notions sur les Frameworks et les patterns, nous invitons le lecteur de ce référé aux livres [W. Cooper ,1998],[Mark Grand,1999] pour plus d'information sur les Frameworks et les patterns.
L'idée la plus répandue était celle des Frameworks ou squelettes d'application, avec le développement de cette dernière on a vu l'émergence des patterns.

### 2.1.2. Définition :

En général, un pattern décrit un problème qui se produit fréquemment dans la conception et l'implémentation des logiciels, et puis décrit la solution à ce problème de telle manière qu'il puisse être réutilisé [Eckel, 2001]. Selon Bruce Eckel [Eckel, 2001] on peut penser tout d'abord qu'un pattern est une manière perspicace et intelligente de résolution d'une classe particulière de problèmes. Les patterns sont introduits pratiquement pour documenter une bonne conception ; ils sont vus comme des véhicules du transfert de la connaissance et d'expérience à partir des experts au débutant. Pour cela, beaucoup de travaux ont été motivé pour documenter et découvrir des nouveaux patterns dans des divers domaines. Les patterns peuvent être classifiés, selon la phase de développement où ils sont employés, les patterns d'analyse [FOW,1997], les patterns d'architecture [BUS,1996] , les patterns de conception [GAMMA,1995], et les idiomes .
Le terme "design patterns" est souvent utilisé. Cependant il ne faudrait pas en déduire que l'approche patterns intervient seulement dans la partie conception (design).
Cela veut dire qu'un pattern diffère de l'approche traditionnelle qui veut qu'il est une phase d'analyse puis de conception et enfin d'implémentation.

### 2.1.3. Classification des patterns de conception:

Les patterns de conception (Design patterns) sont classifiés selon trois objectifs :

Créateur (Creational): La meilleure façon de créer des instances d'objets est l'objectif de cette classe de patterns, cela rend le programme indépendant de la manière avec laquelle les objets sont crées.

Parmi les patterns créateurs on peut citer:

• Singleton : comment assurer l'unicité de l'instance d'une classe.

Structurel (Structural): Les patterns structurels décrivent la manière avec la quelle la classe et les objets peuvent être combinés pour former des structures plus larges.

Parmi les patterns structurels on peut citer:

• Decorator : extension d'un objet de manière transparente.

Comportemental (Behavioral): Ce type de patterns traitent de la communication entre objets .Parmi les patterns Comportementaux on peut citer:

• Observer : mise à jour automatique des dépendants d'un objet.

• Strategy : abstraction pour sélectionner un algorithme parmi plusieurs.

• Iterator : parcours séquentiel de collections.

### 2.1.4. Exemples des patterns de conception (Design patterns) :

Nous présentons dans cette section quelques patterns de conception, plus précisément, les patterns que nous avons utiliser dans la conception et la description à un niveau plus abstraite les composants de notre Framework.

#### 2.1.1.1. L'architecture Model\View\Controller (MVC):

L'architecture MVC a pour but de faciliter la programmation dans le cas des systèmes ou on a besoin de présenter une même donnée de manière synchrone et multiple. Le modèle MVC sépare un composant logiciel en trois parties: un model, une vue, et un contrôler (voir la figure 2).

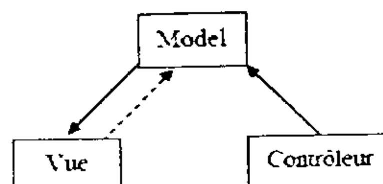



Fig.2: L'architecture MVC

- Le model est la partie qui représente le comportement et l'état du composant logiciel.
- La Vue (View) est la partie qui permet la visualisation de l'état représentant le model.
- Le contrôler (controller) est la partie qui gère l'interaction de l'utilisateur avec le model, i.e. permet de changer l'état du model.

Il est important de noter que :

- Le model n'as aucune connaissance particulière sur ces contrôleurs, c'est le système qui avertit les vues lors d'un changement d'état dans le model.
- Un model peut avoir plusieurs vues et plusieurs contrôleurs.

#### 2.1.1.2. Le pattern observer :

Le pattern Observer a deux interfaces qui laissent coordonner le sujet observé avec ses observateurs. Ces interfaces sont l'opération notify() dans le sujet et l'opération d'update() dans l'observateur.

Deux classes sont alors utilisées pour implémenter le pattern observer, la classe abstraite observer et la classe observable. Tous les objets qui désirent être avertis lors d'un changement dans un autre objet doivent implémenter la classe observer. Les objets qui notifient d'autres objets d'un changement qu'ils ont subis doivent quant a eux étendre (hériter) de la classe observable.

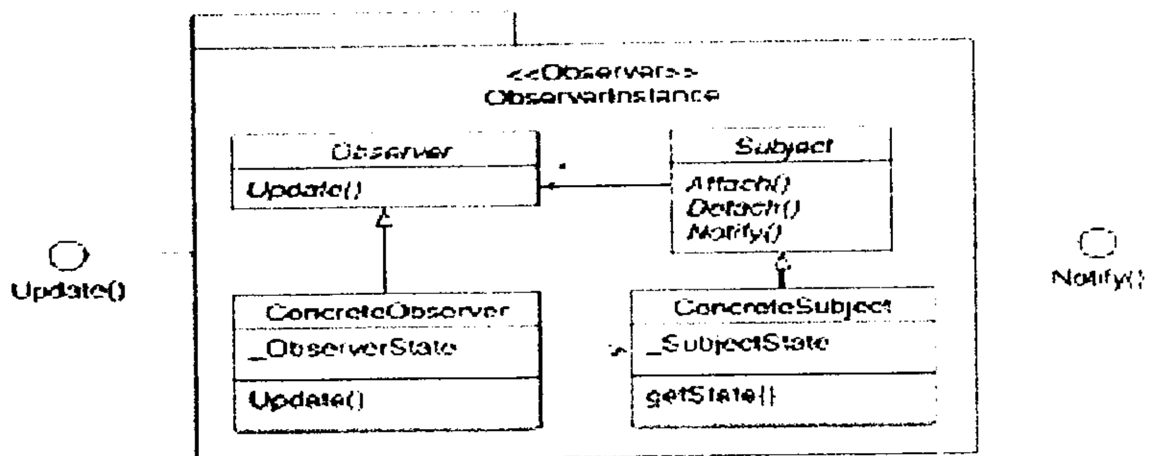



Fig.3: Diagramme de classe du pattern Observer

Le principe de ce pattern est très simple :

• L'observable (le model) subit un changement qui peut intéresser d'autres objets (de type observer), il avertit alors ces derniers de ce changement.

• L'observer ayant reçu un avertissement effectue les opérations nécessaires ou prévues lors d'un changement dans l'observable émetteur de cet avertissement.

### 2.2. Les Frameworks :

Les Frameworks constituent avec les composants, les schémas de conception (*design patterns*) et les bibliothèques (*libraries*), des techniques de réutilisation du logiciel.

Afin de mieux comprendre quels sont les buts de chacune de ces techniques, nous détaillons tout d'abord les Frameworks qui font l'objet de cette thèse, puis les comparons aux autres.

#### 2.2.1 Définition :

Afin de définir un Framework, nous préférons citer quelques définitions plutôt que d'en créer une nouvelle. Ainsi, voici une définition de Ralph Johnson [Joh, 1997] : *« a Framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their* instances interact ». Celle-ci peut être complétée de la définition suivante : *« a Framework is the skeleton of an application that can be customized by an application developer »* [Joh,1993]. Ces définitions sont les plus utilisées, cependant certaines personnes omettent la dimension liée au domaine d'application, et d'autres vont jusqu'à affirmer qu'un Framework est *« A design pattern implementation »* [Lor, 1993] quand il ne s'agit pas d'une simple « classe abstraite » [Pre, 2000]. Nous indiquons au lecteur que nous adhérons principalement aux définitions qui introduisent la notion d'application, et qu'ainsi nous n'adhérons, parmi les définitions précédemment citées, qu'à celles de Johnson.

En offrant modularité, extensibilité et un comportement minimal, un Framework a pour but de faciliter la construction d'une application en fournissant un petit ensemble de composants (classes) connus et éprouvés à composer ou à étendre. Il va au-delà de la simple réutilisation de code en favorisant la réutilisation d'analyse et de conception.

Nous retiendrons principalement de cet ensemble de classes son caractère adaptable et extensible, et la représentation partielle incomplète qu'il propose d'une application.

Le terme Framework sera gardé en anglais faute d'une traduction simple.

### 2.2.2Les Frameworks :

Un Framework a donc à la fois un aspect structurel (une architecture de classes) et un aspect comportemental (les fonctionnalités fournies par ces classes).

Quelques-uns des Frameworks les plus connus sont HotDraw [Bra.1995], ET++ [WGM88], CORBA [JMG,1997], SanFransisco [Boh,1998], JUnit [BG,1998], Eclipse [OTI,2001], AWT [Mic,1997], JavaBeans [Mic,1997], COM, X11, EJB [Mic]... Ils couvrent des domaines aussi variés que les interfaces graphiques au sens large (HotDraw, ET++, AWT, X11), la construction d'environnements de développement (Eclipse), la gestion d'entreprise (SanFransisco), les bus logiciels (DCOM, CORBA), les modèles de composants (COM, EJB, Java-Beans), etc.

### 2.2.3. Pourquoi utiliser un Framework ?

Un Framework est utilisé comme base à l'écriture d'une application, car il fournit des éléments à partir desquels il est plus facile de construire une application.

Les bénéfices liés à l'utilisation d'un Framework sont [SBF,1996] :

- la cohérence entre les différents produits dérivés de celui-ci ;
- l'augmentation de la productivité [MN,1996] ;
- la réduction de la maintenance ;
- l'économie d'argent.

Paradoxalement, ces bénéfices ne deviennent réels que si le Framework est utilisé sur une longue période de temps [FSJ, 1999, p. 11], [SBF, 1996]. En effet, l'apprentissage d'un Framework est plus long que l'apprentissage d'une simple bibliothèque, et peut parfois être aussi long que de concevoir une application. Cela est dû à la complexité du Framework qui ne résout pas un problème particulier, mais une généralisation de ce problème.

Il est donc conseillé de réutiliser le même Framework plusieurs fois. Par ailleurs, la construction d'un Framework est une activité encore plus coûteuse. Elle requiert une très grande expertise du domaine à modéliser de manière à fournir une flexibilité suffisante aux développeurs et la validation d'un Framework n'est faite qu'une fois la mise en oeuvre d'applications suffisamment différentes effectuée. [Joh,1993].

### 2.2.4. Types de Frameworks et utilisation :

L'utilisation d'un Framework implique l'extension, la paramétrisation, la composition, ou la définition de classes afin de réaliser une application. Nous appelons cette application, application dérivée.

Bien qu'on puisse caractériser les Frameworks par leur domaine d'application, leur taille, leur langage d'implémentation, le fait qu'ils possèdent ou non leur boucle de contrôle... nous avons décidé de les étudier par leur type d'utilisation et leurs mécanismes d'adaptation.

Ainsi, on identifie deux grands types de Frameworks : les Frameworks blancs et les Frameworks noirs. La couleur ne représente pas une dichotomie stricte, mais décrit un style d'utilisation d'un Framework qui n'est jamais ni complètement blanc, ni complètement noir.

Un Framework est le plus souvent développé par généralisation à partir d'applications existantes Frameworks et autres techniques de réutilisation

### 2.2.5. Frameworks blancs :

Le terme Framework blanc désigne à la fois une forme de Framework et sa technique de réutilisation. Ainsi l'utilisateur d'un Framework blanc a accès au code source et doit généralement l'étudier avant de pouvoir l'étendre. L'utilisation d'un tel Framework, principalement basée sur l'héritage, est très liée à des fonctionnalités du langage d'implémentation (héritage, liaison dynamique, polymorphisme), et se fait en ajoutant de nouvelles classes, sous-classes, surchargeant des méthodes... Bien qu'avec un tel Framework toutes les modifications soient possibles, elles se produisent dans des endroits identifiés où l'adaptation a été prévue. On identifie ces endroits par les termes de « *hotspots*

» [Pre94] ou « *axis of variation* » [DMNS97]. Nous traduirons ces termes par point de variation. Un point de variation identifie une classe ou un ensemble de classes à modifier afin de particulariser le Framework. Ces points de variation sont décrits dans la documentation qui est associée au Framework.

HotDraw et JUnit sont des représentants de cette catégorie de Frameworks.

### 2.2.6. Frameworks noirs :

Comme le terme Framework blanc, le terme Framework noir identifie à la fois une catégorie de Framework et une technique de réutilisation. Un Framework noir est caractérisé par l'impossibilité d'accéder au code source, mais surtout par la technique d'utilisation qui lui est

liée. Ainsi l'utilisation d'un Framework noir se fait par paramétrisation de classes et par l'assemblage de composants. Tout comme dans les Frameworks blancs, ces modifications sont apportées aux points de variation.

COM est un représentant de cette catégorie de Framework.

Cette distinction entre Framework noir et blanc résulte de la dualité entre la facilité de la conception et la facilité d'utilisation. Un Framework noir est plus difficile à développer [RJ, 1996] mais plus simple à utiliser, alors qu'un Framework blanc est plus simple a développer, mais plus complexe à utiliser. En effet, les difficultés liées à l'utilisation d'un Framework noir sont principalement comportementales alors que pour un Framework blanc elles sont principalement structurelles mais aussi comportementales.

Certains Frameworks noirs sont livrés avec un ensemble de composants prêts à l'emploi.

On parle alors de component Frameworks. Tout comme les Frameworks gris, ils représentent un compromis entre la facilité d'utilisation et la facilité de conception en fournissant à la fois des zones « noires » et des zones « blanches »..

### 2.2.7. Instanciation :

La seconde signification du terme utilisation décrit l'instanciation d'un Framework pour la réalisation d'une application. Ceci consiste en la création et la connexion d'instances soit des classes du Framework soit des classes obtenues par personnalisation.

Ainsi, pour un Framework blanc elle se produit après la personnalisation, alors que pour un Framework noir elle consiste plus généralement à sélectionner les composants intéressants et à les connecter. La difficulté de cette utilisation réside dans la compréhension des dépendances comportementales existant entre les instances à l'exécution, et le protocole d'utilisation de chacune de ces instances. En effet, de par sa complexité, un Framework implique inévitablement des dépendances comportementales entre les éléments qui le composent. Ainsi certaines classes doivent être initialisées avant d'être appelées, ou les appels de méthodes de plusieurs composants doivent être enchevêtrés dans le temps. Les erreurs générées par la mauvaise utilisation du protocole du Framework sont d'autant plus difficiles et fastidieuses à corriger que, pour les reproduire, il faut parfois rejouer de longues séquences de tests.

### 2.2.8. Quelque exemple de l'approche multi-agents:

#### 2.2.8.1 Le système des Aglets :

Le système d'IBM *Aglets* propose un environnement de progammation d'agents mobiles dans le langage *Java*.

Le modèle d'objet des *Aglets* définit un ensemble d'abstractions et de comportements pour le développement d'agents mobiles sur Internet.

- Un *aglet* est un objet *Java* comportant un flot d'exécution et disposant de méthodes pour le déplacement, le clonage, le rapatriement, la désactivation sur support persistant, l'activation depuis un support persistant et la destruction.
- Un *context* est un espace d'adressage sur une machine pouvant abriter une ensemble d'aglets, il fournit un environnement sécurisé pour l'exécution des aglets, il peut y avoir plusieurs contextes par machine.
- Un *proxy* est une capacité sur un aglet, il sert d'intermédiaire pour la manipulation des aglets les protégant ainsi de tout accès direct.
- Un *message* est un objet échangé entre les aglets via un proxy.

#### 2.2.8.2 Tacoma :

Le projet TACOMA (*Tromso And COrnell Moving Agents*) a comme objectif de fournir un support dans les systèmes d'exploitation pour la programmation d'agents mobiles. Le modèle ne fait pas de distinction entre un client, un serveur et un agent. Il propose simplement la notion d'agents et des mécanismes de migration et de communication. Potentiellement tous les agents peuvent se déplacer dans le réseau de machines. C'est au niveau applicatif que l'on peut instancier un agent avec un comportement de client mobile, de serveur statique ou réaliser un monde d'agents intelligents. Un agent est un processus exécutant un script *Tcl* qui est capable de se déplacer volontairement sur une autre machine. Un agent peut manipuler les données locales et transporter des données lors de ses déplacements. TACOMA introduit pour cela la notion de *Folder* qui correpond à une unité de donnée manipulable par un agent. Chaque *Folder* est identifié par un nom et contient une valeur. Localement un agent manipule des *Folders* par l'intermédiaire d'un *File Cabinet* qui est un espace de stockage persistant.

# Chapitre 3 : Le langage Java

## 3.1.Introduction :

Pourquoi le langage java ?

Actuellement le langage java est le mieux adapté pour la construction de Framework pour multi-agents (Bigus, 1998).

Un langage simple avec de puissantes briques préfabriquées pour éviter de réinventer la roue.

**Un langage multi plateformes:**

Grâce à sa capacité d'être un langage multi plateforme indépendant de la machine sur lequel il s'exécute car après compilation, on génèrent un byte Code qui sera interprété par une JVM et exécuté sur la machine hôte

Et comme la mobilité du code est un élément indispensable dans les SMA cet atout rend java un langage très puissant pour la modélisation d'applications distribuées

**Un langage orienté objet:**

Etant donné que java est un langage OO cela facilite beaucoup les choses car les agents sont avant tout des objets

**La réflexion :**

L'un des points les plus remarquable de ce langage est la réflexivité qui grâce à elle on peut retiré des information sur n'importe quelle classe et c'est information peuvent être le type des attributs, les méthodes que cette classe utilise ainsi que leur paramètre et leur valeur de retour

De cette manière on peut avoir des informations sur les dépendance des classe, et une création dynamique des objets et même une invocation dynamique des méthodes.

Exemple :

```
Class c1=new Class.forName(" name ");//création dynamique d'une classe
 Object o=c1.getInstance();
Method m;
m.Invoque("Object Name",paramètres) ; //invocation de la méthode m d'une façon dynam
```

**3.2. Le langage java:**

Ce chapitre présente le langage de programmation Java développé par Sun Microsystems et les fonctions fournies par la Trousse d'outils JDK. Ce chapitre fournit une vue d'ensemble rapide du langage, qui a été développer par les programmeurs orientés objets expérimentés. Nous comparons brièvement et contrastons Java contre le C++ et Smalltalk comme un langage de programmation universel, orienté objets et discutons comment Java satisfait les pré-requis pour des agents intelligents, y compris l'autonomie, l'intelligence et la mobilité. La documentation complète des spécifications de langage Java et les interfaces de programmation d'application (des API) est disponible sur le Web http://www.javasoft.com. Il y a beaucoup de livres excellents en programmation à Java. *Java dans une Coquille de noix* (Flanagan) et *Java Fondamental* (Cornell etHorstmann) est deux que nous avons trouvé particulièrement utile.

**3.3. Fonctions du Langage Java**

Cette section n'inclut pas la pleine spécification du Java, mais est destinée pour fournir assez d'information pour un programmeur C++ pour obtenir un sens pour Java. Pour une pleine description du langage, voir le site Web JavaSoft à < http: // www.javasoft.com / >.

**3.3.1. Types de données:**

Bien que Java soit un langage orienté objets, les types de données primitifs ne sont pas des objets. On fournit des objets équivalents comme la partie du langage, mais le primitive permet une performance meilleur en éliminant l'aérien associé aux objets..

### 3.3.2. Objets, Classes et Méthodes:

Java est un langage de programmation orienté objets. Ainsi, donc vous aller programmer en définissant des classes, instantiant des objets et appelant des méthodes sur ces objets.

## 3.4. D'autres Fonctions de Langage Java:

Une fonction intéressante de Java est qu'il utilise le jeu de caractères Unicode 16 bits, plutôt que le codage ASCII 8 bits. Unicode définit plus de 34,000 caractères codés couvrant les langues écrites principales dans le monde entier. Ce support de langage national intégré (NLS), signifie que les Petites applications Java pourraient être écrites pour un auditoire mondial.

Parce que runtime Java supporte le multi-thread la spécification du langage inclut une déclaration *synchronisée* qui peut être utilisée pour marquer les sections critiques de code pour empêcher la corruption d'objets. Le multi-thread est une fonction importante pour les applications distribuées, parce qu'il permet aux programmes serveur d'entretenir des clients multiples en filant de nouveaux thread pour chaque requête. Cette approche donne de meilleure performance que le départ d'un nouveau processus (comme des programmes de CGI-Garbage) quand une requête entre.

### 3.4.1. Java.lang :

Le paquetage *java.lang* contient les classes de langage Java de base. Ceux-ci incluent des classes pour des objets et des classes eux-mêmes **'Object'**, quelques types de données de base **(Boolean, Byte, Char, Double, Float, Int, Long, Math, short, String, StringBuffer)**, des thread **(thread, ThreadGroup)** et quelques autres classes liées à la compilation Java et le temps d'exécution. La majorité des classes de type de données est assez évidente. **La String** et les classes **de StringBuffer** ont de certain intérêt parce que les objets **String** à Java sont immuables, qui signifie que les chaînes ont les valeurs constantes qui ne peuvent pas être changées une fois qu'ils sont définis. **StringBuffer** est utilisé pour des valeurs de chaîne modifiables.

### 3.4.2. Java.lang.reflect :

Le paquetage *java.lang.reflect* inclut des classes pour supporter l'introspection des classes et des objets fonctionnant actuellement dans une Machine virtuelle Java. Ceux-ci incluent des classes pour créer et avoir accès aux tableaux **(le Tableau)** et fournir l'information et l'accès aux classes, des constructeurs, des champs des méthodes et des modificateurs (**Class, Constructor, Field,Method, Modifier**). Notez **que la Class** supporte des méthodes de réflexion, mais fait en réalité partie *de java.lang* pour la compatibilité descendante avec les versions précédentes de Java.

### 3.4.3. Java.io

Le paquetage *java.io* contient un grand nombre de classes, dont la plupart sont descendues **d'InputStream** ou **OutputStream**. Les classes de flot définissent des méthodes pour lire et écrire des données d'une variété de flots. En plus de l'entrée-sortie de flot, *java.io* inclut des classes pour l'entrée-sortie de fichier et le filtrage de nom de fichier (**le Fichier, FilenameFilter, RandomAccessFile**). Le paquetage *java.io* inclut aussi un certain nombre d'interfaces. Deux des interfaces plus intéressantes sont **Serializable** et **Externalizable**. L'interface **Serializable** permet à un objet d'écrire son état à un flot et le lire en arrière de nouveau. C'est utile pour passer des objets comme des paramètres dans une architecture distribuée (comme le PROTOCOLE RMI) et pour faire l'état d'un objet persistant. L'interface **Externalizable** est aussi utilisée pour des objets en continu, mais le réalisateur a plus de contrôle du format et le contenu du flot.

### 3.4.4. Java.util:

Le paquetage *java.util* contient beaucoup de structures de données généralement utilisées dans des applications. Ceux-ci incluent des classes et des interfaces pour la date et *la manipulation* de temps ( **Calendar, Date, GregorianCalendar, SimpleTimeZone, TimeZone**), *des structures* de données communes (**BitSet, Dictionnary, enumeration, Hashtable, Random, stack, StringTokenizer, Vector**), (**EventListener, EventObject, Observable, l'Observateur**) et des propriétés, des lieux et *des paquets* de ressource(**ListResourceBundle, PropertyResourceBundle, ResourceBundle**). Tandis que la plupart d'entre ceux-ci sont évidents, quelques-uns d'entre ceux-ci méritent un regard plus proche.

Les classes d'avis et les interfaces peuvent être utilisés pour mettre en oeuvre le modèle de conception d'observateur (le Gamma et d'autres., 1995), le plus familier dont est le paradigme de

modèle/vue. Un objet **Observable** est celui dont les changements doivent être observés par d'autres objets dans une application. Les objets qui veulent être notifiés quand les changements d'objet **Observables** doivent implémenter l'interface **Observer** et doivent étendre avec l'objet **Observable**.

La classe **Property** permet indexent/estiment des paires à être lu et écrit à un flot. Les objets **property** peuvent être utilisés pour chercher des valeurs personnalisables basées sur une clé, semblable aux variables d'environnement dans beaucoup de systèmes d'exploitation. Si une application doit être exécutée dans des lieux différents, les propriétés peuvent être chargées dans **un PropertiesResourceBundle**, qui est une sous-classe **de ResourceBundle**. La classe **ResourceBundle** contient les objets spécifiques de lieu qui peuvent être chargés comme approprié **au Lieu** actuel. Cela permet à une application d'être écrit indépendant **du Lieu** de l'utilisateur. Ces classes sont très utiles en écrivant le code qui peut être téléchargé du Web et exécuté n'importe où dans le monde. **Le ResourceBundle** et **le Lieu** classent, avec des classes dans le progiciel *java.text*, fournir le support d'internationalisation de Java.

#### 3.4.5. Java.util.zip :

Le paquetage *java.util.zip* fournit le support pour la compression de FICHIERS ZIP et la décompression. Cela inclut des classes pour compresser et décompresser les données (**Deflater, DeflaterOutputStream, GZIPInputStream, GZIPOutputStream, Inflater, InflaterInputStream, ZipEntry, ZipFile, ZipInputStream, ZipOutputStream**) aussi bien que des classes et des interfaces pour faire des sommes de contrôle sur ces fichiers (**Adler32, CRC32, CheckedInputStream, CheckedOutputStream, la Somme de contrôle**).

#### 3.4.6. java.net

*java.net* le progiciel fournit un jeu de classes pour des communications à travers des réseaux. Cela inclut des classes pour travailler avec des adresses Internet et des Repères de Ressource Uniformes (des URL){*(des adresses Internet)*} (**INetAddress, URL,URLConnection, URLEncoder, URLStreamHandler** et **URLStreamHandlerFactory**) et des coupleurs (**Coupler, ServerSocket, DatagramSocket, SocketImpl** et **SocketImplFactory**) aussi bien que des données (**ContentHandler, ContentHandlerFactory, DatagramPacket**). Ces classes qu'admettent les programmes Java pour travailler à n'importe

lequel de trois niveaux logiques sur des réseaux, à l'URL lisent un niveau de page HTML, au niveau de coupleurs cru et au niveau de datagramme encore inférieur. Ce pouvoir et flexibilité n'étonnent pas le centre central de réseau de Java donné.

## 3.5. Utilisation de Java pour Agents Intelligents

Dans cette section, nous parlons des fonctions spécifiques de Java qui supportent des applications d'agent intelligentes. Ces fonctions seront explorées plus en détail dans ce qui suit.

### 3.5.1. Autonomie :

Pour un logiciel être autonome, cela doit être un processus séparé ou un thread. Les applications Java sont des processus séparés et comme tel peut être longtemps fonctionnelle et autonome. Une Application Java peut communiquer avec d'autres programmes utilisant des coupleurs. Dans une application, un agent peut être un thread séparé du contrôle. Les supports de Java des applications chaînées et fournissent le support pour l'autonomie utilisant les deux techniques.

Dans l'Introduction, nous avons décrit des agents intelligents comme des programmes autonomes ou des processus. Comme tel, ils attendent toujours, prêt à répondre à une requête d'utilisateur ou un changement de l'environnement. Une question qui vient à l'esprit est "Comment l'agent sait-il quand quelque chose change ?" Dans notre modèle, comme avec plusieurs d'autres, l'agent est informé en l'envoyant un événement. D'une perspective de conception orientée objets, un événement n'est rien plus qu'un appel de méthode ou un message, avec l'information faite passer sur l'appel de méthode qui définit ce qui est arrivé ou quelle action nous voulons que l'agent exécute, aussi bien que des données requises pour traiter l'événement.

En Java il y a un mécanisme traitant l'événement qui est utilisé dans la Boîte à outils AWT (AWT) pour passer des événements définis par l'utilisateur comme des mouvements de souris et des sélections par menu aux composants de fenêtre sous-jacents. Selon le type d'agent que nous construisons, nous pouvons devoir traiter ces événements à bas niveau, comme quand un contrôle de GUI obtient le focus, ou une fenêtre est redimensionnée ou déplacée, ou l'utilisateur appuie une clé. Le paquetage *awt.event* supporte aussi un niveau plus haut des événements sémantiques comme des actions d'utilisateur.

### 3.5.2. Implémentation du pattern *Observer* :

Java supporte aussi une autre interface d'événement de niveau plus haut appelée la structure **Observable/observateur** et c'est l'implémentation direct du pattern observer. Dans cette approche, objets intéressés ou registre d'Observateurs eux-mêmes avec **l'objet Observable**. Chaque fois qu'une méthode est s'adressent à l'objet **Observable** ou le modèle qui cause un changement significatif d'état ou quelque chose d'équivalent à un événement de haut niveau, alors tous les objets **d'Observateur** enregistrés sont notifiés.

# Chapitre 4: La Mobilité sous java

## 4.1. Introduction :

La migration d'activité est définie comme le mouvement d'une entité en cours d'exécution sur une machine distante. Elle fournit un mécanisme d'exécution répartie puissant mais qui est complexe à mettre en oeuvre.

La migration d'activité dans les systèmes répartis pour les réseaux locaux est un mécanisme introduit pour répondre à trois types de problèmes :

- La répartition de charge : il s'agit d'optimiser la ressource de calcul globale en jouant sur la charge des processeurs disponibles, on déplace une activité s'exécutant sur un processeur très chargé vers un processeur moins chargé.
- La tolérance aux pannes : il s'agit de gérer les pannes en déplaçant les activités vers des processeurs de secours pour assurer une disponibilité constante des services.
- Le partage d'information : le partage d'information entre activités réparties est mis en oeuvre en déplaçant les activités vers la machine où les données sont en mémoire plutôt que de gérer la réplication des données dans plusieurs mémoires.

On peut distinguer deux types de migration dans ces systèmes. La migration à gros grain proposant le déplacement au niveau du processus ou de la tâche et la migration à grain fin le plus souvent par l'intermédiaire d'une mémoire d'objet répartie.

## 4.2. La mobilité et les agents :

L'architecture des réseaux d'information *Client-Serveur*, qu'elle soit mise en œuvre par une technique directe d'échange de message, ou par des mécanismes d'appel de procédure ou d'évaluation à distance demande aux deux parties d'être connectées durant toute l'interaction. La migration d'activité dans les systèmes répartis modernes impose également cette contrainte pour l'accès aux ressources communes et le chargement à la demande du code de l'activité se déplaçant dans le réseau local (système de gestion de fichier réparti, mémoire partagée répartie).

Le paradigme des agents mobiles propose d'utiliser la migration d'activité en supprimant cette contrainte de connexion constante qui n'est pas évidente ni dans les réseaux de grande envergure ni pour les stations nomades.

On demande aux deux parties d'être connectées seulement durant la phase de migration.

Ainsi un agent mobile peut se déplacer dans un réseau de machines offrant des services pour réaliser une tâche complexe.
Un client donne une mission à un agent qui pour la réaliser se déplace dans le réseau de machines accédant localement aux services offerts par ces machines. On peut distinguer trois phases :

- L'activation de l'agent mobile avec la description de sa mission;
- L'exécution de la mission par l'agent qui se déplace pour accéder aux services;
- La récupération éventuelle des résultats de l'agent mobile.

Ce paradigme introduit un fonctionnement *asynchrone* qui permet à une station cliente de se déconnecter pendant la réalisation d'une tâche dans le réseau, l'agent n'a pas besoin du client il est *autonome*. Le schéma de construction de nouveaux services est *dynamique*, un agent réalisant un service est programmé par le client à partir des opérations offertes par les serveurs du réseau.

Il faut différencier la migration d'activité dans les systèmes répartis conçus pour les réseaux locaux et la mobilité d'un agent dans les réseaux de grande envergure.

- Le temps de réponse d'une action réalisée à distance dans les réseaux locaux est relativement prédictible et aisément bornable, ce qui n'est pas le cas dans un réseau de grande envergure du fait de l'engorgement et des défaillances de l'infrastructure de transport.
- Les systèmes d'agents mobiles pour les réseaux de grande envergure utilisent la mobilité pour réduire le coût des communications en amenant le calcul sur les serveurs et pour permettre la déconnexion du client. Le déplacement d'un agent correspond à une migration d'activité à gros grain en une seule communication.

On déplace l'ensemble de l'image d'exécution de l'agent avant de le restaurer, contrairement à la migration d'activité dans les systèmes répartis modernes où le transfert du code et des données se fait après démarrage "à la demande".
Une différence majeure tient au fait que les systèmes répartis proposent des solutions de migration d'activité pour la construction d'un système d'exploitation uniforme sur un réseau local en cachant la répartition. Alors que la mobilité des agents constitue un modèle d'exécution répartie pour les réseaux de grande envergure et les stations nomades. Dans le premier cas c'est le système qui décide de la migration des activités, alors que l'agent mobile utilise explicitement la mobilité pour naviguer dans le réseau selon la stratégie applicative.

Pour mettre en œuvre une mobilité des agents en java nous devront étudier :

-La communication par socket en java.

-Le classLoader.

-La sérialisation.

-Le byteCode java.

**4.2. La communication par socket en Java:**

**http://www.enseeiht.fr/~queinnec/Ens/Chat/socket-java.html**

Les sockets sont les objets de programmation réseau la plus courante. Ils permettent de façon classique d'utiliser une communication par socket selon le schéma habituel client-serveur. La classe sockets Java des (package java.net) offre un accès simple aux sockets sur IP. La figure 4 montre le schéma d'une communication client/serveur.

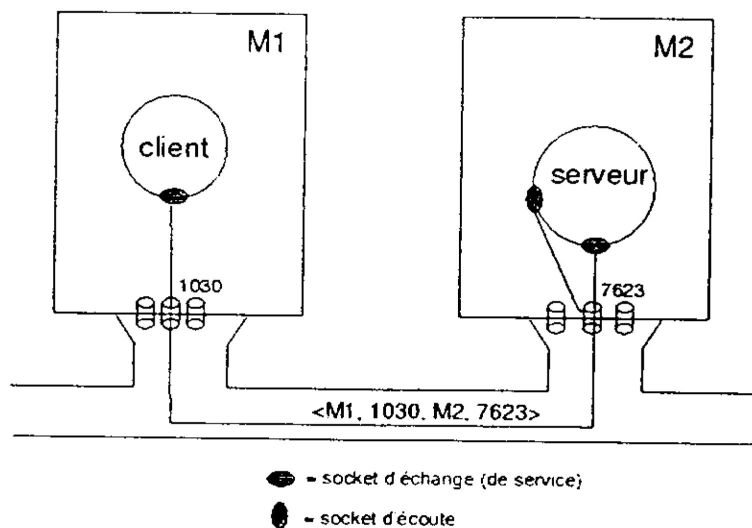



Fig.4: Schéma d'une communication client/serveur

Plusieurs classes interviennent lors de la réalisation d'une communication par sockets. La classe java.net.InetAddress permet de manipuler des adresses IP. La classe java.net.SocketServer permet de programmer l'interface côté serveur en mode connecté. La classe java.net.Socket permet de programmer l'interface côté client et la communication effective par flot via les

sockets. Les classes java.net.DatagramSocket et java.net.DatagramPacket permettent de programmer la communication en mode datagramme.

#### 4.1.1. La classe java.net.InetAddress:

Cette classe représente les adresses IP et un ensemble de méthodes pour les manipuler. Elle encapsule aussi l'accès au serveur de noms DNS.

Un premier ensemble de méthodes permet de créer des objets adresses IP.

public static InetAddress getLocalHost() throws UnknownHostException

Cette méthode renvoie l'adresse IP du site local d'appel.

public static InetAddress getByName(String host) throws UnknownHostException

Cette méthode construit un nouvel objet InetAddress à partir d'un nom textuel de site. Le nom du site est donné sous forme symbolique (bach.enseeiht.fr) ou sous forme numérique (147.127.18.03).

Enfin,

public static InetAddress[] getAllByName(String host) throws UnknownHostException

permet d'obtenir les différentes adresses IP d'un site.

Des méthodes applicables à un objet de la classe InetAddress permettent d'obtenir dans divers formats des adresses IP ou des noms de site. Les principales sont :

public String getHostName() obtient le nom complet correspondant à l'adresse IP

public String getHostAddress() obtient l'adresse IP sous forme %d.%d.%d.%d

public byte[] getAddress() obtient l'adresse IP sous forme d'un tableau d'octets.

#### 4.1.2. La classe ServerSocket:

Cette classe implante un objet ayant un comportement de serveur via une interface par socket.

public class java.net.ServerSocket

Une implantation standard du service existe mais peut être redéfinie en donnant une implantation explicite sous la forme d'un objet de la classe java.net.SocketImpl. Nous nous contenterons d'utiliser la version standard.

Les constructeurs ServerSocke sont:

public ServerSocket(int port) throws IOException

public ServerSocket(int port, int backlog) throws IOException

public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException

Ces constructeurs créent un objet serveur à l'écoute du port spécifié. La taille de la file d'attente des demandes de connexion peut être explicitement spécifiée via le paramètre backlog. Si la machine possède plusieurs adresses, on peut aussi restreindre l'adresse sur laquelle on accepte les connexions.

Les opérations de la classe ServerSocket sont:

Nous ne retiendrons que les méthodes de base. La méthode essentielle est l'acceptation d'une connexion d'un client :

```
public Socket accept() throws IOException
```

Cette méthode est bloquante, mais l'attente peut être limitée dans le temps par l'appel préalable de la méthode setSoTimeout. Cette méthode prend en paramètre le délai de garde exprimé en millisecondes. La valeur par défaut 0 équivaut à l'infini. À l'expiration du délai de garde, l'exception java.io.InterruptedIOException est levée.

```
public void setSoTimeout(int timeout) throws SocketException
```

La fermeture du socket d'écoute s'exécute par l'appel de la méthode close.

Enfin, les méthodes suivantes retrouvent l'adresse IP ou le port d'un socket d'écoute :

```
public InetAddress getInetAddress()
public int getLocalPort()
```

#### 4.1.3. La classe java.net.Socket:

La classe java.net.Socket est utilisée pour la programmation des sockets connectés, côté client et côté serveur.

```
public class java.net.Socket
```

Comme pour le serveur, nous utiliserons l'implantation standard bien qu'elle soit redéfinissable par le développement d'une nouvelle implantation de la classe java.net.SocketImpl.

Constructeurs:

Côté serveur, la méthode accept de la classe java.net.ServerSocket renvoie un socket de service connecté au client. Côté client, on utilise :

```
public Socket(String host, int port) throws UnknownHostException, IOException
public Socket(InetAddress address, int port) throws IOException
public Socket(String host, int port, InetAddress localAddr, int localPort)
        throws UnknownHostException, IOException
```

public Socket(InetAddress addr, int port, InetAddress localAddr, int localPort)throws IOException

Les deux premiers constructeurs construisent un socket connecté à la machine et au port spécifiés. Par défaut, la connexion est de type TCP fiable. Les deux autres interfaces permettent en outre de fixer l'adresse IP et le numéro de port utilisés côté client (plutôt que d'utiliser un port disponible quelconque).

*Appels système:*

Ces constructeurs correspondent à l'utilisation des primitives socket, bind (éventuellement) et connecté.

**4.1.4. Opérations de la classe Socket :**

La communication effective sur une connexion par socket utilise la notion de flots de données (java.io.OutputStream et java.io.InputStream). Les deux méthodes suivantes sont utilisés pour obtenir les flots en entrée et en sortie.

public InputStream getInputStream() throws IOException

public OutputStream getOutputStream() throws IOException

Les flots obtenus servent de base à la construction d'objets de classes plus abstraites telles que java.io.DataOutputStream et java.io.DataInputStream (pour le JDK1), ou java.io.PrintWriter et java.io.BufferedReader (JDK2).

Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer un délai de garde à l'attente de données (similaire au délai de garde du socket d'écoute : levée de l'exception java.io.InterruptedIOException) :

public void setSoTimeout(int timeout) throws SocketException

Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :

| | |
|---|---|
| public InetAddress getInetAddress() | fournit l'adresse IP distante |
| public InetAddress getLocalAddress() | fournit l'adresse IP locale |
| public int getPort() | fournit le port distant |
| public int getLocalPort() | fournit le port local |

L'opération close ferme la connexion et libère les ressources du système associées au socket.

**4.2. Le ClassLoader: Understanding the Java ClassLoader,ibm.com/developerWorks**

Qu'est un ClassLoader ?

Parmi les langages de programmation commercialement populaires, le langage Java se distingue en fonctionnant sur un Machine virtuelle Java (JVM). Cela signifie que la compilation des programmes est exprimé dans un format spécial, indépendant de la plate-forme, plutôt que dans le format de la machine.

Ce format diffère des formats de programmes exécutables traditionnels.

Particulièrement le programme Java, à la différence d'un programme écrit dans les langages C ou C++, n'est pas un fichier exécutable simple, mais composé de plusieurs fichiers de classe individuels. De plus, ces fichiers de classe ne sont pas chargés dans la mémoire immédiatement, mais sont plutôt chargés sur demande. Le ClassLoader est la partie de la JVM qui charge des classes dans la mémoire.

Java ClassLoader, en outre, est écrit en langage Java lui-même. Cela signifie qu'il est facile de créer votre ClassLoader propre sans avoir à comprendre les détails de la JVM.

Pourquoi écrire un ClassLoader ?

Si la JVM a un ClassLoader, pourquoi écrire un autre ?

Le ClassLoader de la JVM peut seulement charger des fichiers de classe du système de fichiers local. C'est excellent pour des situations régulières, c'est-à-dire quand vous compiler votre programme Java entièrement et localement sur Votre ordinateur.

Le langage Java est facile pour JVM pour obtenir des classes dans des places autres que le disque dur local ou le réseau. Par exemple, les navigateurs utilisent une tradition ClassLoader pour charger le contenu exécutable d'un site Web.

Il y a beaucoup d'autres façons d'obtenir des fichiers de classe. Par le chargement de fichiers du disque local ou d'un réseau, on peut utiliser le ClassLoader de la JVM:

### 4.2.1. La structure du ClassLoader :

Le but de base du ClassLoader est d'entretenir une demande d'une classe. La JVM a besoin d'une classe, avec le nom de cette classe comme paramètre pour le classLoader. Le ClassLoader essaie de retourner un objet 'Class' qui représente la classe.

Dans le reste de cette section, nous parlerons des méthodes critiques de Java ClassLoader.

Nous découvrirons comment il accorde le processus de charger des fichiers de classe. Nous découvrirons aussi quel code nous devons écrire en créant notre propre ClassLoader.

ClassLoader.loadClass () est le point d'entrée à ClassLoader. Sa signature est comme Suit:

Classe loadClass (String name,boolean resolve);

Le paramètre de nom spécifie le nom de la classe pour les besoins JVM, dans la notation de paquetage, comme java.lang.Objet.

Le paramètre de résolution informe méthode si la classe doit être résolue. La résolution de classe est comme la tâche qui prépare complètement la classe pour l'exécution.

La résolution n'est pas toujours nécessaire. Si le JVM doit seulement décider si la classe existe ou découvrir sa superclasse, alors la résolution n'est pas exigée.

### 4.2.2. Méthode defineClass :

La méthode defineClass est le mystère central de ClassLoader. Cette méthode prend le tableau cru d'octets et le transforme en objet Class. Le tableau contient les données de cette classe qui a été chargé du système de fichiers ou à travers le réseau.

### 4.2.3. Méthode findSystemClass:

La méthode findSystemClass charge des fichiers du système de fichiers local. Elle cherche un fichier de classe dans le système de fichiers local et le transforme en classe en utilisant defineClass.

Pour notre ClassLoader, nous utiliserons findSystemClass avant d'essayer de charger la classe à partir du réseau.

La raison est simple: notre ClassLoader est responsable de l'exécution des étapes spéciales pour charger des classes, mais pas pour *toutes* les classes. Par exemple, même si notre ClassLoader charge quelques classes d'un serveur distant, il y a toujours l'abondance de bibliothèques Java de base sur la machine locale qui doit aussi être chargée.Donc nous demandons à la JVM de charger les classes du système de fichiers local. C'est que findSystemClass fait.

La procédure travaille comme suit :

*Nous supposons que cette classe est une des bibliothèques Java de base et appeler

FindSystemClass pour la charger à partir système de fichiers.

*S'il la classe n'existe pas en local on demande à notre ClassLoader de charger la classe.

* Nous vérifions le serveur distant,pour voir si la classe est là.

* S'il est, excellent,nous chqrgeons la classe et c'est bon.

#### 4.2.4. Méthode resolveClass:

Comme j'ai mentionné précédemment, le chargement d'une classe peut être fait partiellement (sans résolution) ou complètement (avec résolution). Quand nous écrivons notre version de loadClass, nous pouvons devoir appeler resolveClass, selon la valeur du paramètre de résolution à loadClass.

FindLoadedClass sert d'une mémoire cache : quand on demande à loadClass de charger une classe, il peut appeler cette méthode et voir si la classe a déjà été chargée par ce ClassLoader, pour ne pas recharger une classe qui a déjà été chargée. Cette méthode devrait être appelée lq première.

Voyons comment toutes ces méthodes vont ensemble.

Notre mise en œuvre de loadClass effectue les étapes suivantes. (Nous ne spécifierons pas ici quelle technique spéciale sera utilisée pour obtenir le fichier de classe - il pourrait être chargé via le réseau, ou d'archives, ou a compilé en temps réel. Quoi qu'il soit, on obtient les octets du fichier class d'une manière ou d'une autre)

* Appeler findLoadedClass pour voir si nous avons déjà chargé la classe.

* Si nous n'avons pas chargé la classe, nous trouvons une maniére pour obtenir les octets crus.

* On Appele findSystemClass pour voir si nous pouvons charger la classe du système de fichiers local.

* Si on peut pas charger la classe du système de fichier local nous devont charger les octets crus à partir du réseau ou autre endroit susceptible et appelez defineClass pour les transformer en objet 'Class'.

* Si le paramètre de résolution est vrai, appelez resolveClass pour résoudre l'objet de Classe.

* Si nous n'avons pas toujours de classe, jetez un ClassNotFoundException.

* Autrement, retournons la classe.

**4.3. Le ByteCode java:** Security Aspects in Java Bytecode Engineering Blackhat Briefings 2002, Las Vegas, Aug 01, 02).

La figure 5 montre la structure du fichier .Class.

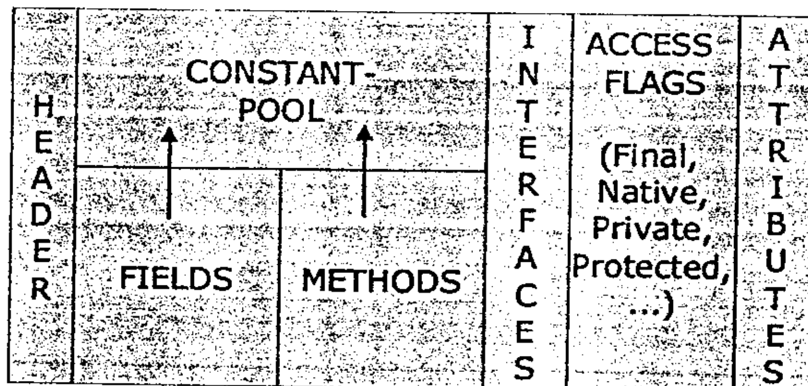



Fig.5: Structure du fichier .Class

**4.3.1. Le Format du Fichier .class :**

Pourquoi étudier le format du fichier .class ?

Nous avons vu que la classLoader s'occupe de charger des classes dans la mémoire, mais avant de les charger nous devons connaître quelles sont les classes qui doivent être charger.

Lors du chargement d'une classe nous devons aussi charger toutes les classes dont elle aura besoin, alors comment connaître ces classes là.

Nous avons vu précédemment que java 2 contient de puissant concepts pour détecter de tels classes. Grâce à la réflexion nous pouvons détecter tous les attributs ainsi que toutes les méthodes et leurs paramètres. Nous pourrons alors savoir quelles sont les classes que nous devrions charger.

Jusque là ça va, mais les classes dont nous avons besoin ne se trouvent pas seulement dans les attributs et les paramètres des méthodes, nous avons aussi les corps des méthodes qui peut aussi avoir besoin d'une autre classes et qui ne se trouve pas dans les attributs et les paramètres. C'est pourquoi nous devons étudier la structure du byte code java afin de pouvoir détecter les classes utilisées dans le corps des méthodes et par la même occasion les classes utilisés ailleurs.

• Des fichiers de classe Java sont apporté dans la JVM via le Classloader

• Le fichier de classe est essentiellement juste un tableau d'octet, après les règles du vérificateur du code objet.

• Toutes les quantités 16 bits et 32 bits sont formé par lecture dans deux ou quatre octets.

#### 4.3.2. Méthodes et Champs:

• Le type d'un champ ou d'une méthode est indiqué par une chaîne appelé sa signature.

• Des Champs peuvent avoir un attribut complémentaire donnant la valeur initiale au champ.

•Les Méthodes ont un attribut de CODE complémentaire l'octroi du code objet qui va être exécuté.

#### 4.3.3. L'Attribut de CODE:

L'attribut code contient :

• L'espace maximum à empiler

• Nombre maximal de variables locales

• Le code semi-interprété réel pour exécuter la Méthode.

• Une table de gestionnaires des exceptions.

Les types JVM :

• TYPES JVM et leurs préfixes :

- byte **b**
- short s
- int **i**
- long **l**
- char **c**
- Float **f**
- double **d**
- Fait référence à une Classes, Interfaces, Tableaux

• Ces Préfixes sont utilisés dans les opcodes (iadd, astore...)

#### 4.3.4. Les instructions JVM:

Mnémotechnique:

• (pop, swap, dup, ...)

• Calcul (iadd, isub, imul, idiv, ineg,...)

• Conversion (d2i, i2b, d2f, i2z,...)

• storage(iload, istore,...)

• Operation sur les tableau (arraylength, newarray,...)

• Utilisation d'objet (get/putfield, invokevirtual, new)

• Push operation (aconst_null, iconst_m1,....)

• Control de saut (nop, goto, jsr, ret, tableswitch,...)

• Threading (monitorenter, monitorexit,...)

#### 4.3.5. Code semi-interprété :

• Le Code objet Java (JBC) est suivi par le zéro ou plus d'octets d'information d'opérande complémentaire.

•les instructions de consultation de Table (tableswitch, Lookupswitch) ont une longueur flexible

• Aucun code auto modificateur.

• Aucun branchement à un emplacements arbitraires, seulement au début des instructions limitées à portée de méthode actuelle.

# Chapitre 5 : Systèmes de Raisonnement

Il y a eu diverse méthodes pour la conception de l'intelligence, tel que l'intelligence à base de cas, l'intelligence des jeux de réflexion comme le jeu d'échecs qui est implémenté avec les algorithmes du même type que alpha/beta, mais l'intelligence à base de règles a eu beaucoup de succès et est la plus utilisé surtout dans le domaine des système multi-agents. C'est pourquoi nous l'avons utilisé.

## 5.1. Raisonnement à base de Règles :

Les règles Si alors sont devenues la forme la plus populaire de représentation de connaissances déclaratives utilisées dans des applications d'intelligence artificielle. Il y a plusieurs raisons pour cela. La connaissance représentée comme les règles de type ' si alors ' sont facilement compréhensibles. La plupart des personnes sont à l'aise en lisant ses règles, par contraste avec la connaissance représentée dans la logique d'attribut. Chaque règle peut être vue comme un morceau autonome de connaissance ou d'une unité d'information dans une base de connaissance. La nouvelle connaissance peut être facilement ajoutée et la connaissance existante et peut être changée simplement en créant ou modifiant des règles individuelles.

Les règles sont facilement manipulées par des systèmes raisonnants. Le chaînage avant peut être utilisé pour produire de nouveaux faits et le chaînage arrière peut déduire si les déclarations sont vraies ou non. Des systèmes à base de règles étaient un des premiers succès commerciaux à grande échelle de recherche d'intelligence artificielle. Il consiste en trois éléments principaux, *une base de connaissance* (le jeu de si alors des règles et des faits connus), *une mémoire de manoeuvre* ou une base de données de faits tirés et des données et *un moteur d'inférence*, qui contient la logique raisonnante qui traite les règles et les données.

Avant que nous n'entrions dans les détails de raisonnement avec des règles, regardons une règle simple :

Si nb_roues = 4 et moteur = oui alors Type = automobile

On a deux clauses *d'antécédent* jointes par une conjonction (nb_roues = 4 et le moteur = oui) et une ou plusieurs clauses *conséquentes* simples (Type = automobile). Une règle expose un rapport entre des clauses (des affirmations ou des faits) et selon la situation, peut être utilisée pour produire la nouvelle information ou prouver la vérité d'une affirmation. Par exemple, si nous savons qu'un véhicule a quatre roues et un moteur, alors en utilisant la règle, nous pouvons conclure que le Type est une automobile et ajouter le fait à notre base de connaissance. D'autre part, si nous essayons de prouver que le Type est une automobile, nous devons découvrir si le véhicule a quatre roues et un moteur. Dans le premier cas, nous sommes dans le chaînage avant. Une règle dont les clauses d'antécédent sont toutes vraies peut être *déclenché*. Nous *renvoyons* une règle déclenchée en affirmant la clause conséquente et l'ajoutant comme un fait à notre mémoire de manoeuvre. À tout moment, une base de règle peut contenir plusieurs règles qui sont prêtes à être déclenchée. C'est à la stratégie de contrôle du moteur d'inférence de décider la quel est a déclenché. Nous discuterons ce point plus en détail plus tard dans ce chapitre.

La plupart des systèmes à base de règle permettent aux règles d'avoir des noms ou des étiquettes comme **Rule1 :** ou **Automobile :** pour facilement identifier des règles à la rédaction ou à tracer pendant l'inférence. Quelques systèmes permettent des disjonctions **(ou)** entre des clauses d'antécédent. C'est une sténographie qui réduit la taille d'une base de règle. Par exemple :

La règle 1 : si nb_roues = 2 alors Type = cycle

La règle 2 : si nb_roues = 3 alors Type = cycle

La règle 3 : si (nb_roues = 2 ou nb_roues = 3) alors Type = cycle

La règle 3 pourrait remplacer la Règle 1 et la Règle 2 dans la base de règle. La plupart des systèmes de règle permettent aussi aux opérateurs de condition booléens comme <,> et! =, en plus d'égalité. Nous pourrions réécrire la Règle 3 sans utiliser des disjonctions :

La règle 4 : si (nb_roues > 1) et (nb_roues < 4) alors vehicleType = cycle

Beaucoup de systèmes à base de règles permettent aux fonctions d'être appelés dans des clauses d'antécédent. Ces fonctions sont appelées *des détecteurs*, parce qu'ils sortent du moteur d'inférence et testent une certaine condition dans l'environnement. Les détecteurs rendent d'habitude des valeurs booléennes, mais ils peuvent aussi rendre des données ou des faits à la mémoire de manoeuvre. Quand on permet des fonctions dans le conséquent, ils sont appelés *des effecteurs*, qui étendent énormément la capacité du système à base de règles. Un effecteur ajoute

à une règle un mécanisme produisant un fait dans un périphérique produisant une action. Ces règles d'action permettent aux agents intelligents de faire des choses pour nous. Par exemple, un agent intelligent qui traite le courrier électronique contient la règle suivante :

La règle6 . si détecteur (Mail_Arriver) alors effecteur (Traiter_Mail)

Où Mail_Arriver et Traiter_Mail sont définis comme des clauses et le détecteur () et l'effecteur () sont des méthodes fournies par le système d'inférence pour invoquer ces fonctions.

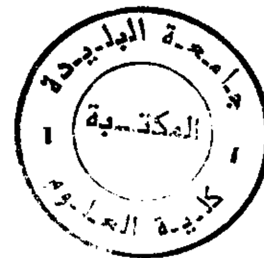
Les règles peuvent représenter beaucoup de types de connaissance de domaine. Cependant, comme le nombre de règles grandit, les aspects intuitifs des règles si alors sont diminués et ils perdent leur efficacité d'une perspective de lisibilité. Dans les deux sections, nous utiliserons une base de règle simple comme un exemple. Cette base de règle devrait être familière. Nous l'appelons la Base de Règle de Véhicules .Il a seulement neuf règles et sept variables, avec une variable intermédiaire. Sept des règles définissent le sorte de véhicule et deux sont utilisé pour déterminer si le véhicule est un cycle ou une automobile. La brièveté et la clarté de cette petite base de règle nous aideront à se nous concentrer sur les questions liées à l'inférence à base de règles.

Notre domaine de véhicules peut identifier trois types de cycles (un avec un moteur et deux sans) et sept types d'automobiles. Nous différencions des cycles comme ayant moins de quatre roues et des automobiles comme ayant exactement quatre roues et un moteur. Les types divers d'automobiles sont identifiés par leur taille relative et le nombre de portes qu'ils ont.

**5.2. Chaînage avant :**

Le chaînage avant est un processus de raisonnement conduit par des données où un ensemble de règles et est utilisé pour tirer de nouveaux faits d'un jeu initial de données. Il n'utilise pas l'algorithme de résolution utilisé dans la logique d'attribut. L'algorithme de chaînage avant produit de nouvelles données par l'application simple et directe ou le renvoi des règles. Comme une procédure d'inférence, le chaînage avant est très rapide.

Regardons le processus de raisonnement plus en détail. Comme mentionné auparavant, n'importe quel système expert exige trois éléments de base, une base de connaissance de règles et des faits,



une mémoire de manoeuvre pour stocker des données pendant l'inférence et un moteur d'inférence. **Les étapes suivantes font partie du cycle de chaînage avant :**

**1. Charger la base de règle dans le moteur d'inférence et n'importe quels faits de la base de connaissance dans la mémoire de manoeuvre.**

**2. Ajouter n'importe quelles données initiales complémentaires dans la mémoire de travail.**

**3. *Correspondre* les règles et les données dans la mémoire de travail et déterminer quelles règles sont à déclencher, signifiant que toutes leurs clauses d'antécédent sont vraies. Ce jeu de règles déclenchées est appelé *le jeu de conflit.***

**4. Utiliser la procédure *de résolution de conflit* pour choisir une règle simple du jeu de conflit.**

**5. *Renvoyer* la règle choisie en évaluant la clause (s) conséquente; mettez à jour la mémoire de travail si c'est une règle produisant un fait, ou appeler la procédure d'effecteur, si c'est une règle d'action. C'est mentionné comme l'étape *d'acte.***

**6. Répéter des étapes 3, 4 et 5 jusqu'à ce que le jeu de conflit soit vide.**

### 5.2.1. Avanteges ;

-Fonctionne bien lorsque le problème se présente naturellement avec des faits initiaux.

-Produit une grande quantité de faits.

-Adapté au contrôle.

### 5.2.2. Inconvénient :

Souvent ne perçoit pas certaines évidences.

# Chapitre 6 : Conception de AgentTo

Le schéma de la figure 6 montre le diagramme de classes de AgentTo

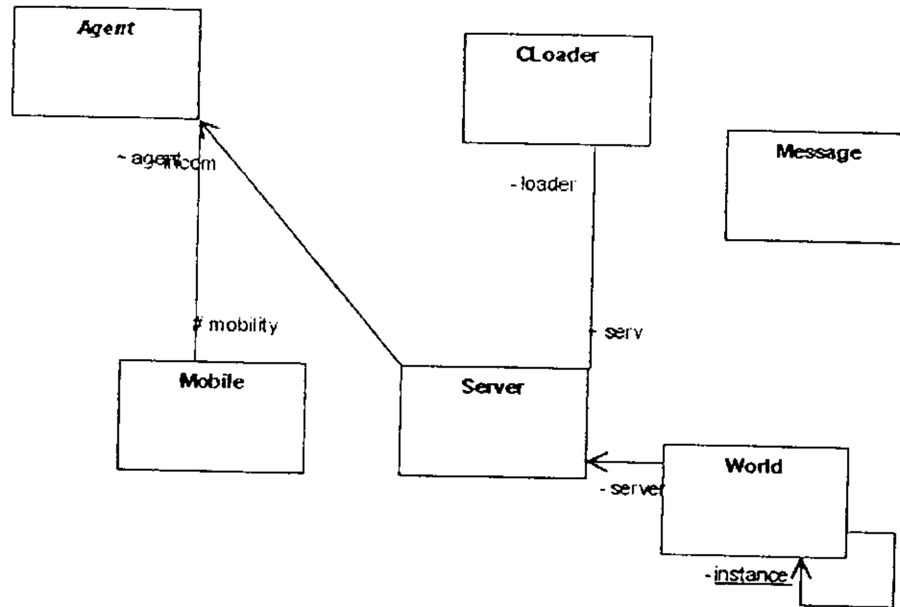



Fig.6: Diagramme de classes de AgentTo

## 6.1. Interaction et communication :

Dans ce chapitre nous allons parler des grands axes de notre Framework AgentTo.

La société dans laquelle notre système évolue se compose de mondes, pour chaque virtuelle machine nous avons un et un seul monde qui lui abrite un nombre d'agents. Pour chaque monde nous avons une adresse IP et un numéro de port ce qui fait qu'on peut avoir plusieurs monde dans le même hôte mais avec différents ports et différentes machines virtuelles. Un agent peut circuler entre ces mondes pour qu'il soit plus proche des ressources nécessaires.

### 6.1.1. Les Mondes :

Un monde est unique dans chaque machine virtuelle et donc dans chaque instance du programme pour cela nous avons utilisé le pattern singleton.

Le But du singleton est de créer une seule instance d'un objet. Pour ce faire, le constructeur du monde doit être privé. On construit une méthode qui permet de créer le monde s'il n'existe pas ou de renvoyer une référence sur ce monde là s'il existe. Le code est le suivant:

**Code :**

```
class World{

        public static synchronized World getInstance(){

                if (instance == null)instance = new World();

return instance;}

        private World() {

                MSGList=new ALinkedList();}

...}
```

A part regrouper un nombre d'agents et leur servir de boite aux lettres, un monde sert aussi à recueillir d'autres agents qui viennent d'autres mondes grâce à son serveur mais nous verrons cela plus loin.

Un autre rôle primordial pour le monde est d'envoyer des messages d'un agent à un autre. Pour cela un agent crée un message et l'envoie à un autre agent et demande au monde de délivrer ce message au destinataire. Le monde incorpore le message dans une liste et demande aux agents de ce monde là de rechercher les messages reçus. Pour cela on a utilisé le pattern MVC (Model, View, Controller).

#### 6.1.1.1 Le MVC :

Le pattern MVC est utilisé lorsqu'on a plusieurs vues pour une seule donnée: Dans ce cas on a 3 catégories d'objets:

i. Les models : utilisé comme données.
ii. Les vues : Pour l'utilisation des données.
iii. Les contrôleurs : Pour l'affectation des models et des vues.

#### 6.1.1.2. Comment ça marche :

Un événement peut conduire un contrôleur à changer un model ou une vue ou les deux à la fois. Quand un contrôleur modifie les données d'un model toutes les vues associées à ce model sont automatiquement notifiées et les vues reçoivent les données de modification par le model.

La figure 7 montre l'architecture du MVC avec deux vues.

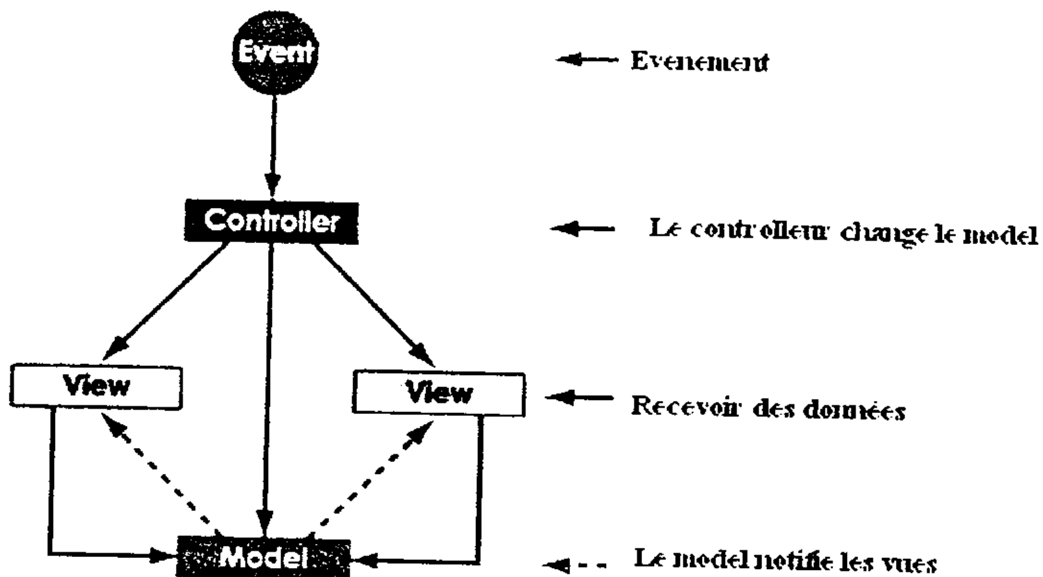



Fig.7: Le MVC

Dans notre cas le monde est le contrôleur car il est le seul à pouvoir modifier la liste des messages. Le model est la liste des messages car il est l'objet à modifier par le monde. Les vues sont les agents de ce monde. Dés que le message arrive dans la liste, il y a notification pour chaque agent.

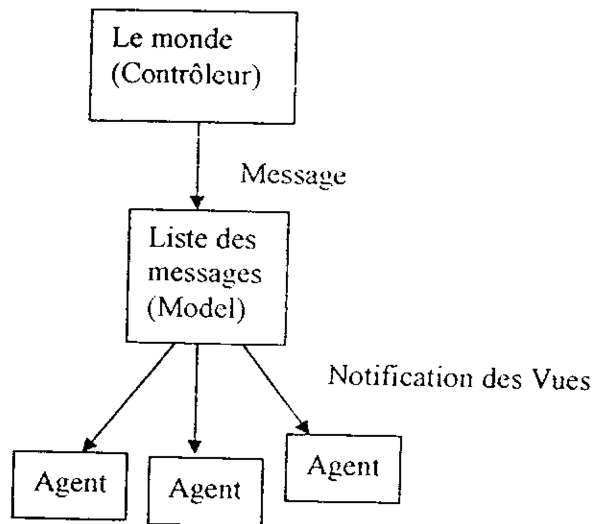



Fig.8: Application du MVC dans AgentTo

Le monde compte aussi une liste de référence sur les agents. Le monde est un 'observable'. Les agents sont des 'observer'. L'interface 'observer' et la classe 'observable' de JAVA nous facilite beaucoup la mise en œuvre du MVC. Pour implémenter le 'pattern' Il suffit juste d'en hériter et de redéfinir la méthode 'update' de la classe abstraite 'observer'.

```
public class World extends Observable{

private Vector AgentsList=new Vector();

public void AddAgent(Agent My){// méthode qui sert à ajouter un observeur sur la liste de message

        AgentsList.add(My);

        this.addObserver(My);

}

...
```

```
}

public abstract class Agent implements Observer,Linkable, Messageable{

//Update est la méthode qui va être invoquer lors d'une notification et elle va essayer de //
retirer le message qui lui est destiné est faire un traitement spécifique

    public  final synchronized void update(Observable observable, Object object){

        World world=World.getInstance();

        if(observable == world){

            Message message=world.getMessage(this);

            if(message != null){

            boolean first=(WaitingMessages.getLegth()==0);

            WaitingMessages.insertAtTail(message);

            try{

            if(first)processMessages();}

            catch(Exception e){

            }

            System.out.println("message "+message.getID()+" received"); }

        } else UserUpdate( observable,  object);

...

}
```

### 6.1.2. Les Agents :

Les agents sont les unités de traitement auxquels on affecte des tâches bien précises. Les agents sont abstraits c'est-à-dire que chaque agent concret doit hériter de la classe Agent. Dès qu'une classe en hérite elle devient automatiquement un agent et nécessite la redéfinition des méthodes.

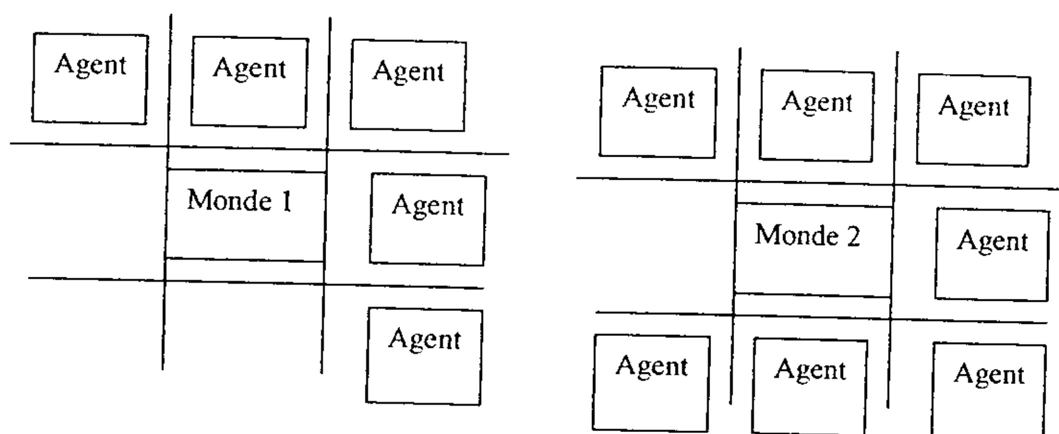



Fig.9 : Disposition des mondes et des agents

La méthode qui doit être absolument définie est la suivante:

public abstract void processMessages() ;

Cette méthode qui doit être définie par l'utilisateur du Framework sert à spécifier le comportement de l'agent en fonction du message reçu (en attendant le système expert). Donc l'aspect recommandé du corps de cette méthode est de la forme suivante:

**Message msg;**

**while((msg=GetMessage())!=null){**

**//Traitement en fonction du message reçu**

}

L'utilisateur de ce Framework reste toujours libre de choisir le corps qui lui conviendra.

Une autre bonne manière est de créer un 'thread' qui s'occupe du traitement des messages. Ce qui veut dire que la méthode ProcessMessages ne fait que lancer ce thread avec la méthode start du thread comme dans l'exemple mais il faut prendre en compte la synchronisation des objets partagés.

L'agent possède aussi son propre monde mais sans une référence sur ce monde.

Mais comment l'agent va communiquer avec son monde s'il ne possède pas de référence dessus ?

Puisque chaque instance du programme ne possède qu'un et un seul monde pour connaître le monde, il suffit d'appeler la méthode 'static' du monde qui est getInstance(). Cette méthode va créer un monde s'il n'existe pas encore ou nous renvoyer une référence sur ce monde.

De cette manière, le pattern MVC est utilisé sans avoir une référence sur le monde comme attribut dans notre agent.

L'agent est identifié par successivement :

1-son nom qui est un type string. Il est unique dans tout le système pour éviter toute confusion lors de la mobilité.

2-la machine hôte : c'est la machine sur laquelle l'agent ce trouve à ce moment là. Cet attribut est utilisé pour faciliter la communication.

3-un identificateur unique de type *long* qui peut être utilisé de la même façon que le nom mais avec un traitement moins lourd car les types numériques sont bien plus faciles et plus simples à manipuler que les chaînes de caractères.

### 6.1.3. Manière de travail :

De cette manière on peut profiter des travaux antérieurs et les rendre intelligents. Nous pouvons aussi composer avec d'autres travaux faits par le passé et refaire appel à la réutilisation du code ce qui veut dire transformer des 'Object' en 'True Object'.

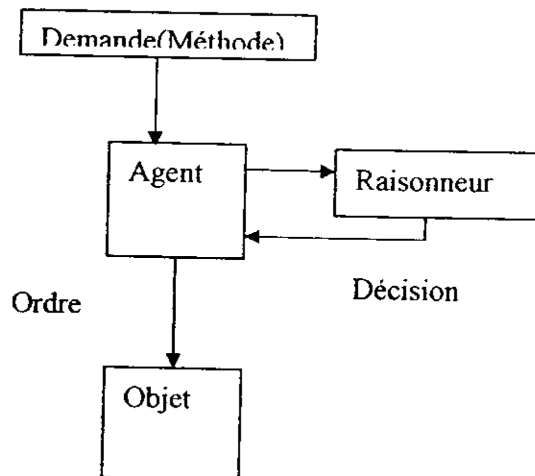



Fig.9 : Schémas de fonctionnement

Comment faire ?

Par exemple nous avons une classe Homme qui était faite par le passé et a été compilé. Nous voulons qu'une instance de cette classe ait un comportement intelligent et qu'elle coopère avec d'autres objets, qu'elle devienne un vrai objet intelligent capable de prendre un décision et de se déplacer.

On construit une autre classe agent (par ex AgentHomme). Cette classe va hériter de la classe agent comporter ses propriétés. Cette classe va implémenter une référence sur l'objet Homme et grâce au système expert de notre agent on va avoir un comportement intelligent de notre Homme et le rendre coopératif et capable de migrer d'un site à un autre. De cette façon on va profiter de tous les travaux antérieurs et on aura un grand gain en terme de coûts et de temps.

Pour cela on utilise le pattern decorator.

Le décorateur est un pattern très puissant qui fait en quelque sorte un héritage d'une manière un peu spécial. C'est un héritage sans les points faibles de l'héritage en java (Par exemple héritage multiple).

De cette façon, l'agent Homme va décorer les méthode de l'objet homme et on aura héritage de la classe agent et un héritage par enveloppement de la classe Homme.

Tout cela a été utilisé dans l'exemple.

Pour un comportement intelligent, l'agent a besoin d'un système expert, c'est pourquoi nous avons intégrés dans l'agent une référence sur un raisonneur.

### 6.1.4. La communication :

La communication entre agents se fait par messages. Un agent qui désire envoyer une information, un ordre ou une demande à un autre agent, il fait passer un message. Donc l'agent va construire un message qui a la structure suivante :

```
public final class Message implements Linkable, Serializable{
private Linkable next;
        private String type;// Le type de message
        private String name;//Le nom du message
        private int ID;//L'identificateur du message
```

```
private Agent sender;//L'agent qui envoi le message

private Agent receiver;//L'agent destinataire du message

private String methodName;//Le nom de la méthode a exécuté

private Class[]parameters;//Les paramètres de la méthode

private Object[]arguments; //Les arguments de la méthode

private Calendar sendTime;//l'heure d'envoi

private Calendar recTime;//L'heure de réception

}
```

Nous avons introduit dans notre message une méthode avec des paramètres et des arguments. Cette méthode va nous permettre d'effectuer une demande auprès de l'agent receveur en lui demandant de l'exécuter. L'agent va avoir un comportement en fonction du système expert et l'envoi du message va se faire en demandant au monde de posté ce message. Le monde va mettre le message dans la liste des messages et va demander aux agents de ce monde d'effectuer une recherche et d'essayer de trouver les messages qui leurs sont destinés.

Un message peut être destiné à un agent qui n'est pas de ce monde, si c'est le cas, le monde va l'envoyer au monde du message receveur et ce monde va faire le même travail que si le message lui a été envoyé par l'agent de son propre entourage.

Nous avons donc un serveur dans chaque monde qui s'occupe d'un tel transfert et la classe message va hériter de l'interface 'serialisable' et un objet message va être envoyé de serveur en serveur via des streams.

**6.2. Le raisonneur :** La figure 10 montre le diagramme de classe du raisonneur.

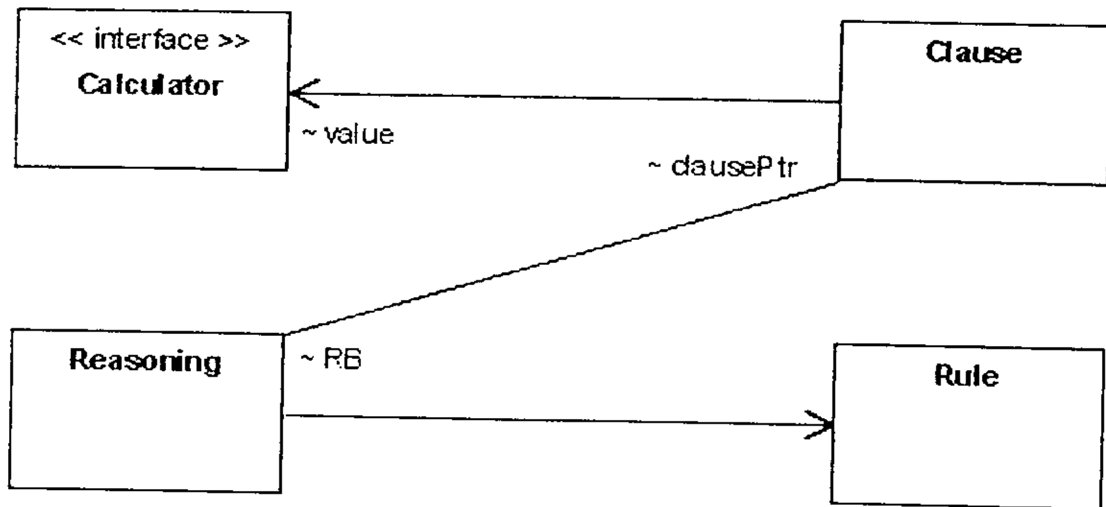



Fig.10: Diagramme de classe du raisonneur

Chaque agent possède son propre raisonneur qui va avoir une très grande influence sur le comportement de l'agent.

Voici comment est implémenté notre système expert :

**6.2.1. Règles :**

La classe **Rule** est utilisée pour définir une règle simple. Elle contient les méthodes qui soutiennent le processus d'inférence. Chaque **Règle** a un nom, un vecteur d'antécédent, **des Clauses** et **une Clause** conséquente simple. La valeur de vérité **de la Règle** est stockée dans *un* **booléen**. Notez, c'est un objet **Booléen**, pas une variable booléenne élémentaire. Cela nous permet d'utiliser une valeur nulle pour indiquer que la vérité de la règle ne peut pas être décidée (parce qu'une des variables qui fait référence dans une clause est aussi nulle ou non définie).

Le membre booléen ***fired*** indique si cette règle a été appliquée ou non.

Ce qui fait que la régle peut avoir 3 états :

1. Vrai : Si on l'a appliquée
2. NULL : Si une des clauses antécédentes est inconnue
3. Faux : sinon

Le constructeur de Rule est comme suit :

**public Rule(RuleBase Rb, String Name, Vector lhs, Clause rhs);**

Rb : est la base de règle, auquel la règle va appartenir.

Name : est le nom de notre règle.

lhs: est un vecteur qui contient les clauses antécédentes de notre règle.

rhs: est la clause conséquente de la règle.

### 6.2.2. Clause :

Les clauses sont employées dans les parties antécédentes et conséquentes d'une règle. Une clause se compose habituellement d'une **variable** du côté gauche, d'une **condition** qui examine l'égalité, supérieur, inférieur, et différent.

Par exemple la règle:

**age**: **Si** age>18 **alors** adulte =true   Contient deux clauses. La première clause antécédente se compose **de la variable** age, **la condition** ">", et un entier 18. L'autre clause est idem.

**Une clause** contient également un vecteur **des règles** qui contiennent cette **clause**, un booléen **consequent** qui indique si la clause apparaît dans l'antécédent ou le conséquent de la règle, et un entier **truth** qui indique si la clause est vraie, faux, ou inconnu.

Les membres de la clauses sont les suivants:

```
public class Clause{
private String name;      //le nom de la clause
private int ID;      //identificateur de la clause
private LinkedList rules;    //les règles utilisant cette clause
private Boolean truth;//la valeur de vérité de la clause: true,'false'ou inconnu
private boolean consequent ; // si la règle est dans les clauses conséquentes
```

Notre Framework doit être le plus général possible et traiter tous les cas. On doit alors trouver une bonne représentation de la clause. C'est pourquoi nous avons decider de ne pas mettre une variable gauche et une chaine à droite avec une condition ,mais plutôt utilisés une interface Calculator.

L'implémentation de l'interface a la forme suivante:

```
public interface Calculator{

Boolean Calc(Clause owner);

}
```

Pour toute clause, il se doit de lui référencer un objet héritant de l'interface calcultor

```
public class Clause{
//...
Calculator value;

synchronized public void Check(){

        truth=value.Calc(this);

}

}
```

Cet objet va définir la méthode Calc(Clause) qui va calculer la valeur booléenne de la clause. Dans cette méthode on n'est pas restreint à juste un coté gauche, une condition ou un coté droit mais àn'importe quel traitement complexe. C'est ce que nous avons appeler l'ordre 0 extensible. La figure11 montre le diagramme de la clause.

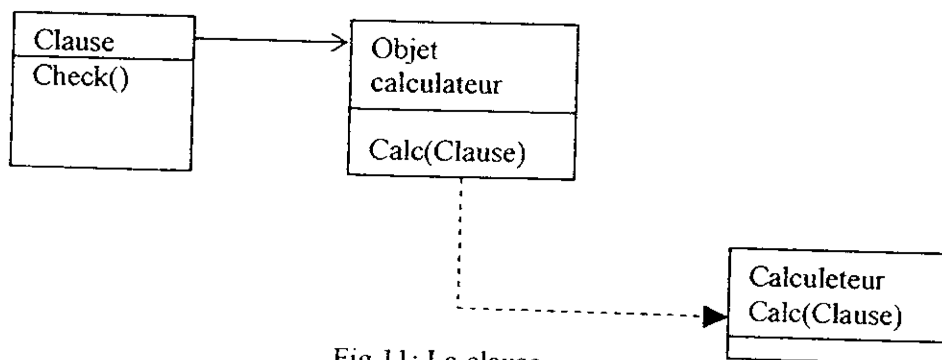



Fig.11: La clause

La classe de **clause** contient aussi La méthode *d'addRuleRef() qui* est employée par le constructeur de **règle** pour enregistrer la **règle** avec cette **clause**.

**void AddRule(Rule Rl){rules.addElement(Rl);}**

La clause est aussi un 'Observable' et dés sa modification on peut notifier tout les 'Observer'.

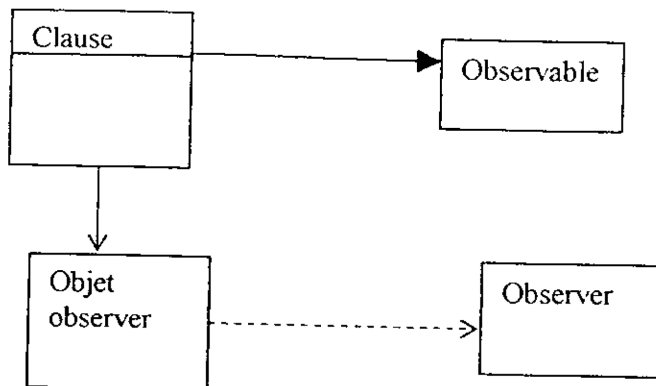



Fig.12 : Le pattern 'Observable' dans la clause

Et contient un traitement spécial que nous verrons un peu plus tard.

**6.2.3. Le raisonneur :**

**La class reasoning:**

Cette classe définit le comportement d'un système expert à base de règle, elle possèdent deux vecteurs, le premier pour la liste des clauses et le deuxième pour ses règles. Elle implémente deux méthodes, la première **forwardChain()** pour le chaînage avant et la deuxième pour la réinitialisation du système.

```
class reasoning
{
String name ;//le nom de la RuleBase
    LikedList clauseList;//la list des Clauses
    LinkedList ruleList ;//la listes des regles
    void forwardChain();
    void reset();
```

```
//...
}
```

Le constructeur de la classe reasoning alloue deux vecteurs pour la liste des règles et clauses et admet comme paramètre son nom.

```
public Reasoning(String name) {
        this.name = name;
                                        }
```

**L'implémentation du Chaînage avant :**

Le chaînage avant est implémenté comme une fonction **forwardChain()** dans la classe **Reasoning**.

La méthode crée d'abord le vecteur de **conflictRuleSet.** Elle appelle La méthode **match()** avec un paramètre booléen vrai pour forcer un premier essai de toutes les règles dans la base de règle. Elle retourne ensuite *le* **conflictRuleSet** initial, un vecteur des règles qui sont déclenchées et pourraient être inférés.

Nous mettrons une boucle *while()* qui fonctionne jusqu'à ce que nous ayons *un* **conflictRuleSet** vide. A l'intérieur de la boucle, nous appelons d'abord la méthode **selectRule()** avec *le* **conflictRuleSet** comme paramètre. La méthode **selectRule()** recherche et retourne la meilleur règle a inférer. Nous appelons **Rule.fire()** pour mettre à vrai la règle et la clause conséquente et puis on essais de nouveau toutes les clauses et règles qui se rapportent à la variable mise à jour.

Ensuite, nous appelons la méthode **match()**.

Le code de la méthode **forwardChain()** est comme suit:

```
synchronized public void forwardChain(boolean cycle) {

   LinkedList conflictRuleSet = new Vector() ;

   // test1

   conflictRuleSet = match(true); // selection

   while(conflictRuleSet.size() > 0) {
```

```
Rule selected = selectRule(conflictRuleSet);
selected.fire() ;
   System.out.println(selected.GetName());
                          this.setChanged();
            this.notifyObservers();
conflictRuleSet = match(cycle);
  }
}
```

Maintenant regardons avec plus de détail les différentes méthodes.

**Reasoning.match()**: la méthode prend un paramètre booléen simple. Elle crée un vecteur de règles **matchList**. Si le paramètre **test** est vrai, il appelle **Rule.check()** pour examiner toutes les clauses antécédentes de règle et placer la valeur de vérité de la règle. Si *l'essai* est **test**, *le match()* regarde simplement la valeur de vérité de la règle courante. Si la règle est vraie, et elle n'a pas été déjà inférée, elle l'ajoutera au vecteur *de* **matchList**. Sinon, nous passerons à la prochaine règle sur **le ruleList**.

La signature de la méthode **Reasoning.match()** est comme suit:
**public Vector match(boolean test) ;**
**Reasoning.selectRule()**: la méthode prend un vecteur des règles comme paramètre représentant l'ensemble des règles en conflit. Notre fonction est assez simple, elle recherche la meilleure règle à inférer, pour elle une meilleure règle qui possède le plus grand nombre d'antécédents et peut utiliser d'autres critères.

La signature de la méthode **Reasoning.selectRule()** est comme suit:
**public Rule selectRule(Vector ruleSet) ;**

Le raisonneur possède aussi un thread qui va servir pour l'inférence.

Nous allons maintenant retourner à notre clause. Nous avons définis une méthode setTruth(boolean); cette méthode sert à modifier la valeur booléenne de la clause et lancer le raisonneur, mais si le raisonneur possède un thread, elle ne va que lancer ce thread sinon le raisonneur va utiliser le thread courrant pour faire son traitement sur les règles.

De cette manière dés qu'une clause change elle va entraîner le changement de toutes les règles qui lui sont associées.

## 6.3. La Mobilité des agents:

La mobilité est sans doute la phase la plus complexe pour mettre en œuvre un système multi-agents. Il y a deux sortes de mobilité: la forte mobilité et la faible mobilité.

La forte mobilité : cette forme de mobilité fait migrer non seulement tout l'agent ainsi que tout ses paramètres mais aussi l'état dans lequel le traitement s'est arrêté.

La faible mobilité : la faible mobilité ne fait migrer que l'agent et ses paramètres mais pas l'état du thread du traitement.

Vu que le langage java ne dispose pas de mécanisme pour la mis en œuvre de la forte mobilité, nous avons juste appliqué la faible mobilité et pour la forte mobilité la virtuelle machine nécessite une modification pour localiser l'état d'un thread, de plus la forte mobilité est rarement nécessaire.

L'objet principal que nous devront rendre mobile c'est l'agent, c'est pourquoi il doit être serialisable et on premier lieu nous devront établir une connexion entre l'agent et le monde vers lequel il va migrer.

Chaque monde implémente un serveur qui s'occupe du dialogue entre l'agent et le monde.

### 6.3.1. Le serveur du monde:

Le rôle du serveur est d'être en écoute des demandes des agents des mondes extérieurs. Les taches qu'il doit faire sont :

-Recevoir les agents migrants d'autres mondes.

-Intercepter les messages venus d'un autre monde.

Le serveur contient deux identificateurs principaux: Une adresse et un port.

L'adresse est une adresse IP classique qui lui sert d'identificateur et le port est un numéro de communication. Donc, quand l'agent décide de se déplacer vers un autre site, il ouvre une communication avec le serveur du monde. La communication est à base de socket.

Donc le serveur possède aussi un socket de communication.

Pour le traitement continue des requêtes, notre serveur a un thread .Il implémente l'interface Runnable et définit la méthode run();

La méthode run() est définit comme suit:

```
public void run () {//...
ServerSocket s = new ServerSocket(port);
while (true) {
synchronized(soc){
soc = s.accept();
int length;
byte [] code;
DataInputStream Din = new DataInputStream(soc.getInputStream());
NumC=Din.readInt();
```

```
for(int i=0;i<NumC;i++){
name  = Din.readUTF();
System.out.println(name);
length = Din.readInt();
code = new byte[length];
Din.read(code);
loader.loadClass(name,true);
}
ObjectInputStream obj=new ObjectInputStream(Din);
incom=(Agent)obj.readObject();
//...
}
```

Nous remarquons que le serveur attend une communication de la part d'un agent et celui-ci commence par faire migrer la classe qu'il aura à utiliser dans l'autre site. Il va envoyer au serveur le nombre de classes qu'il aura à envoyer.

Il commence par la première classe. Il va envoyer son nom puis sa taille. Le serveur possède aussi un classLoader. Il va essayer de la charger à partir du système de fichier local, s'il la trouve, il va dire à l'agent de ne pas envoyer son code et s'il ne la trouve pas, il va demander à l'agent son code puis l'agent transmet le code de la classe au serveur et la le serveur va utiliser son classLoader pour charger la classe dans sa virtuelle machine. Ce qui va nous obliger à créer un classLoader personnalisé. Et ainsi de suite avec tous les autres classe. A la fin l'agent devra envoyer son propre code avec ses paramètres:

Le code de chargement des attributs de l'agent est le suivant:

```
ObjectInputStream obj=new ObjectInputStream(Din);
    incom=(Agent)obj.readObject();
```

Ensuite on lance le traitement de l'agent soit par un thread soit par la méthode processMessages() de l'agent.

### 6.3.2. Le ClassLoader :

Comme nous l'avons dit le serveur possède son propre classLoader personnalisé qui va servir à charger des classes.

Le code de mon ClassLoader est le suivant:

```
public class CLoader extends java.lang.ClassLoader{
Hashtable Loaded;
byte[] code;
int len=0;
Server serv;
//...
}
```

Le classLoader possède une HashTable qui contient toutes les classes qu'il a déjà chargé pour empêcher le double chargement d'une même classe.

Donc en premier lieu, le ClassLoader va vérifié s'il n'a pas déjà charger cette classe auparavant, sinon il va utiliser le classLoader du Système pour essayer de charger la classe du système du

fichier local et s'il ne la trouve pas, il va alors recevoir les octets qui sont reçus par le serveur et va les enregistrer avec la méthode suivante:

**defineClass(name,code,0,code.length);**

Où name est le nom complet de la classe et code est le code binaire de la classe.

### 6.3.3. Le choix des classes :

Jusque là il n'y a aucun problème. Nous avons une communication, envoie d'information et chargement de classe. Comment va faire l'agent pour envoyer et choisir les classes dont il aura besoin dans l'autre monde ? Nous avons vu précédemment que grâce à la réflexion, on peut déterminer le type des attributs de chaque classe et aussi les types de paramètres des méthodes. Le problème qui reste est le corps des méthodes, celui-ci contient des objet ayant d'autres types, c'est pourquoi nous avons pensés à une méthode un peu brute mais très efficace et qui marche à tous les coups, cette méthode consiste à scanner le byte code java et à déterminer les classes dont nous avons besoin.

### 6.3.4. Le Byte Code java :

La méthode qui scan le byte code pour déterminer les classes utiles est la suivante:

**public static String[] getClasses(byte[] bytecode);**

Cette méthode prend en compte un Byte code d'une classe et retourne un tableau de String contenant les noms des classes dont la classe à besoin:

On commence la lecture de notre Byte Code:

En premier lieu, nous devons lire un nombre magique qui sert à identifier toute classe java et qui est le 0xCAFEBABE. Si on ne lit pas ce code, on jette une exception qui dit que le code d'entrée n'est pas un code d'une classe java. On lit Ensuite deux unsignedShort qui signifient deux version (la version mineur et la version majeur) mais on ne va pas se préoccuper de ça.

Ensuite on va lire le nombre de constantes utilisées, puis on lis chaque constante une par une.

Par exemple: Si le nombre de constante est 5, on fait 5 fois in.readUnsignedByte() qui signifie un type. Examinons le source de prés:

```
for(int i = 1; i < constantPoolCount; i++) {// constantPoolCount est le nombre de
//constantes

   int tag = in.readUnsignedByte();

   switch(tag) {

    // Une chaîne de caractère c'est un nom d'une classe

    case CONSTANT_Utf8:

     strings[i] = in.readUTF();

     break;

   // Un index de nom de Classe il pointe vers le nom de la classe

   case CONSTANT_Class:

    classIndexes.addElement(new Integer(in.readUnsignedShort()));// on //sauvegarde les
index

    break;

   // Pas très important mais lisant la suite proprement

   case CONSTANT_Long:

   case CONSTANT_Double:

    i++; // long & double prennent la taille de 2 constantes

    in.readInt();
```

```
 // suivant car sa ne nous interesse pas
 case CONSTANT_Integer:
 case CONSTANT_Float:
  in.readInt();
  break;
//on passe
 case CONSTANT_Fieldref://Si c'est un champ
 case CONSTANT_Methodref://ou nom de méthode
 case CONSTANT_InterfaceMethodref://ou nom d'une méthode d'interface
 case CONSTANT_NameAndType:
  in.readUnsignedShort();
  // on passe
 case CONSTANT_String:
  in.readUnsignedShort();
  break;
 default:
  throw new IOException("bytecode Invalide: " )
 }
}
```

Apres cela le reste du byte code ne nous intéresse pas.

Maintenant nous avons toutes les classes dont nous avons besoins du moins les classes du premier niveau. Le type d'un tableau est inclus séparément, c'est pourquoi on doit ignorer tous les types qui commence par un '['.

Pour avoir vraiment toutes les classes nécessaires, on doit parcourir les classes de façon hiérarchique.

Exemple: La dépendance de classes de la figure 13.

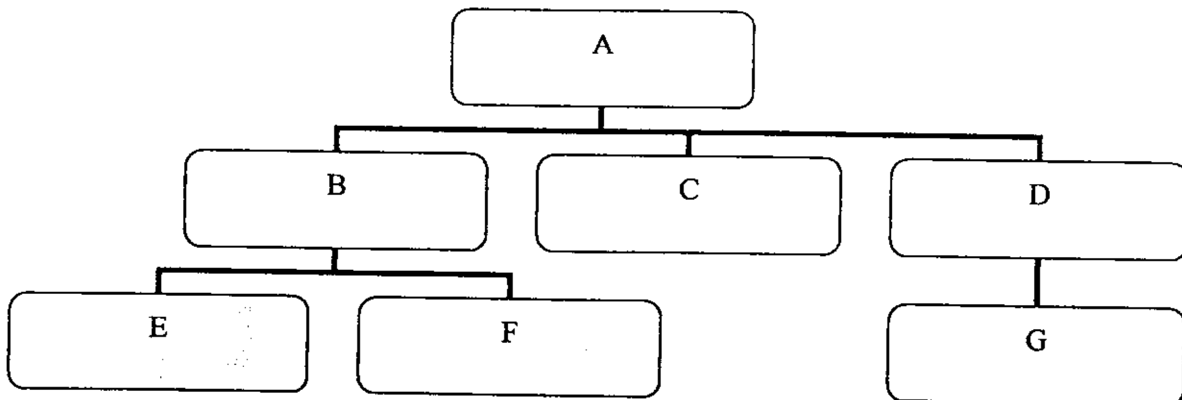



Fig.13: La dépendance de classes

On voit que la classe A utilise les classes B, C et D mais en réalité elle a besoin des classes B, C, D, E, F, G. Donc ce qu'on doit faire, c'est que pour chaque classe on doit la parcourir pour déterminer d'autres classes nécessaires jusqu'à ce qu'il n'en restera plus tout en évitant les circuits.Dés qu'on trouve une classe on l'ajoute a la classe nécessaire mais si elle y est déjà on s'abstient de l'ajouter et on fait de même avec toutes les classes trouvées.

Le chargement de la classe se fait à partir du système de fichier local et à partir de son nom. On la cherche dans le classpath:

Le code de chargement du **byteCode** de la classe **className** est le suivant:

```
public static byte[] getClassBC(String className)
  throws IOException {
  byte[] byteCode = null;
File f = null;
  className = className.replace('.', File.separatorChar) + ".class";
 for(int i = 0; i < classpath.length; i++) {
   f = new File(classpath[i] + className);// on cherche dans classpath
   if(f.exists()) {
    byteCode = new byte[(int) f.length()];
    // Lire le byte code à partir du fichier
    FileInputStream fis = new FileInputStream(f);
    fis.read(byteCode);
    fis.close();
    break;
  }
 }
 return byteCode;
}
```

De cette manière nous avons fait le nécessaire pour qu'un agent devienne mobile à 100% mais sans le thread d'exécution.

### 6.3.6. L'envoi de message entre différents mondes:

Nous avons vu précédemment qu'un agent peut communiquer avec un autre agent via des messages mais sans qu'il soit dans un autre monde.

Pour qu'un agent puisse envoyer des messages vers un autre monde, il doit communiquer avec un serveur de nom. Ce serveur de nom contient une table où il y a tous les agents et tous les mondes. Les étapes d'envoie des messages sont:

a) L'agent demande au monde de poster un message.
b) Le monde se rend compte que l'agent destinataire est ailleurs.
c) Il demande au serveur de nom (via une communication avec socket) de lui dire ou l'agent se trouve.
d) Le serveur de nom répond avec l'adresse et le port du monde ensuite le monde envoi le message au monde destinataire.
e) Ensuite le monde destinataire fera comme si le message était un message local.

# Chapitre 7 : Comment utiliser AgentTo

Notre Framework utilise une bibliothèque pour la création de système multi-agents. On a travailler de telle manière qu'il soit le plus général possible afin qu'il soit utile dans n'importe quel domaine. Le système peut être exécuter à partir de tout terminal compatible avec java 2.

## 7.1. Manière de travail:

Au début nous avons le monde qu'on peut instancier on appellent World.getInstance().

Une fois ce monde créé, l'utilisateur du Framework n'a pas à s'on préoccuper par la suite.

Il doit créer une nouvelle classe(La classe qui deviendra l'objet agent) héritant de la classe AGENT.IA.Agent et obligatoirement redéfinir la méthode procesmessages() de celle-ci, cette méthode va servir à traiter les messages qui sont envoyé à l'agent. Un aperçu de la méthode procesmessages() est donnée ci-dessous:

```
Message msg;

    while((msg=GetMessage())!=null){

    //C'est là que se trouve le comportement de l'agent
```

Si GetMessage() renvoi un null le traitement va s'arrêter mais dés qu'un nouveau message arrive, la méthode processMessages() va être automatiquement invoquer.

L'utilisateur du Framework va redéfinir cette méthode ou bien créer un thread pour l'agent, donc cette méthode ne va que lancer le thread conçu par l'utilisateur. On voit que c'est très simple de créer des agents concrets. L'utilisateur doit aussi créer le raisonneur de notre agent. Il doit définir toutes les clauses ainsi que toutes les règles pour que l'agent ait un comportement intelligent en fonction des résultats du raisonneur.

L'utilisateur devra créer les clauses avec leur calculateur qui déterminera la valeur de la clause et construire les règles en fonction des clauses puis ajouter toutes les règles au raisonneur.

Ensuite le raisonneur redémarrera automatiquement. Comme l'agent comprend une hashTable de clauses (pour chaque méthode il y a une clause) le comportement de chaque méthode est en fonction de sa clause.

L'utilisateur doit lancer le serveur de nom qui s'occupera tout seul de se paramétrer mais il doit le faire avant toute exécution car dés sa création l'agent va s'enregistrer auprès du serveur de nom. L'utilisateur peut relier son agent avec n'importe quel travail antérieur pour en bénéficier. En cas de manque de ressources ou de nécessité à la mobilité, l'agent va juste exécuter la méthode Agent.move(host,port). On voit bien que la tâche de l'utilisateur est très réduite, il n'a qu'à définir le système expert (raisonneur), le comportement de l'agent et le moment de migration. Nous allons voir un exemple concret qui illustre tout cela.

### 7.2. L'exemple AgentHomme:

La classe Homme qui est la classe de l'objet homme simple est définit comme suit:

```
public class Man  implements Maneable{

//...

public void eat(String o);//mange

public void rest();//repos

public void wake();//réveil

}
```

Nous voulons donner un comportement intelligent à cette classe. Nous allons donc la relier à notre agenthomme:

```
public class AgentHomme extends Agent implements Maneable

//...
```

```
public void eat(String o){

        if(Clauses.get("eat").getTruth()==true)man.eat(o);

}

        public void rest(){if(Clauses.get("rest").getTruth()==true)man.rest();

        }

        public void wake(){if(Clauses.get("wake").getTruth()==true)man.wake();

        }
```

Nous remarquons ici que nous avons utilisé le pattern decorator.

La figure 14 montre le diagramme de classe de l'agent Homme.

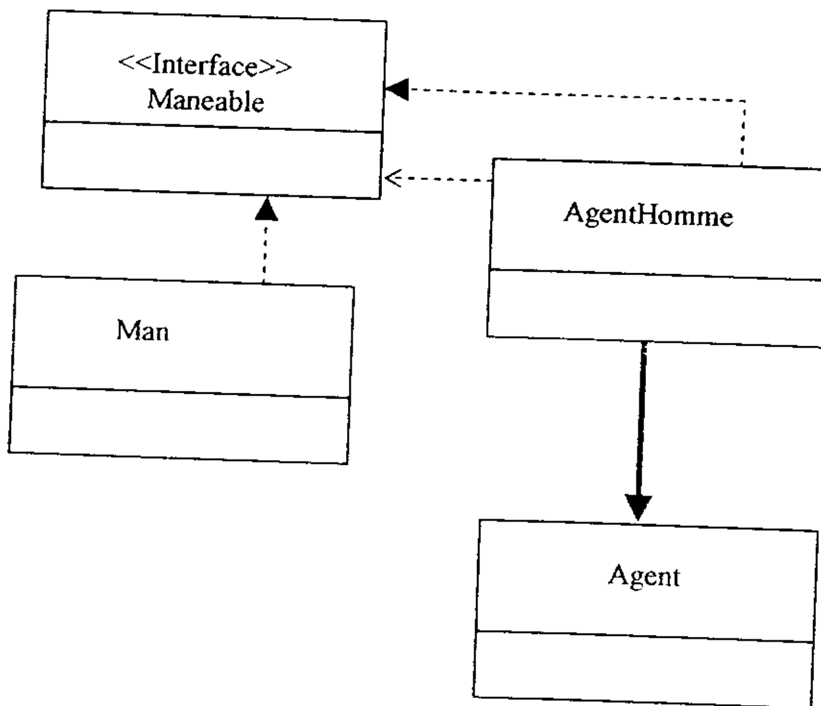



Fig.14: Diagramme de classe de l'agent Homme

De cette manière l'AgentHomme hérite d'une manière spécial la class Man sans les points faible de l'héritage classique.

Ensuite en crée le raisonneur:

```
Clause r=new Clause();
r.setName("eat");
        Clause c1=new Clause();
        c1.setName("notSick");
        Clause c2=new Clause();
        c2.setName("happy");
        Clause c3=new Clause();
        c3.setName("hungry");
```

Maintenant notre règle devient si c1 & c2 & c3 alors r,soit si notsick & happy & hungry alors eat.

Pour le raisonneur nous avons un thread. Dés qu'il y a un changement il démarre pour faire une inférence. On a aussi utilisé un thread pour le traitement des messages pour chaque agent. Ce thread attend les messagse et les traite un par un:

Le thread invoque cette méthode :

```
public void run() {
    Message msg;
    while((msg=GetMessage())!=null){//Réception d'un message
```

```
System.out.println("process messages");

try{

Method ToIN=msg.getMethod(this);

if(ToIN.getName().equals("eat")){// on voi quelle est la méthode à exécuter

// test aussi des paramétres de la méthode

ToIN.invoke((Object)man,msg.getArgs());}}//On invoque cet méthode

catch(Exception ex){

ex.printStackTrace();

}

}

}
```

Ensuite dans le main on va instancier quelque agents Homme et on va envoyer quelque message et on notera le résultat:

```
eat
message posted id=2
message 2 received
process messages
objet manger pain
Press any key to continue..._
```

**Sortie d'un exemple:**

On voix en premier que la demande est eat.

Ensuite un message est envoyé.

Ensuite il est reçu.

Puisque toutes les conditions sont satisfaites, la méthode eat est exécutée.

Nous avons un agent capable de recevoir des messages avec un raisonnement.

Pour la mobilité il suffit d'exécuter la méthode move de l'agent pour le déplacer d'un site à un autre. L'utilisateur n'a pas à ce préoccupé de la manière de fonctionnement du move.

Et de cette façon en remarque qu'on a un nouvelle objet qui est agentHomme, qui intègre les fonctionnalité de Homme avec une intelligence, de communication et capable de migrer.

### 7.4. Comparaison par rapport aux autres plates-formes:

Par rapport aux plates-formes multi-agents présentées, notre Framework se caractérise par pouvoir transformer facilement un 'Object' en 'True Object' vu qu'il définit les interactions et les communications, l'intelligence et la mobilité. Il utilise des mécanismes spéciaux pour garantir une mobilité très efficace, néanmoins il a la faiblesse de ne pas supporter la forte mobilité (tout comme les systèmes développés en java) ,mais les statistiques montre que la forte mobilité et beaucoup moins utilisé que la faible. Une différence par rapport à aglet c'est l'absence du proxy qui est du à l'absence du traitement de sécurité dans AgentTo.

# CONCLUSION GENERALE

Ce travail m'a permis d'apprendre beaucoup de choses sur java, la communication par socket, le multithread, le classLoader, le byteCode java, le raisonnement à base de règles, quelques patterns et les différentes techniques de réutilisation. Nous avons aussi vu que pour développer un tel travail ce n'est pas très simple c'est pourquoi nous avons fait beaucoup d'études sur les travaux antérieurs.

Nous avons introduit une manière de création d'agents assez efficace et assez simple avec un grand niveau de flexibilité et de réutilisation.

Pour les perspectives à venir, il faut ajouter d'autres formes d'intelligence comme le raisonnement à base de cas, l'apprentissage en utilisant les réseaux de neurone par exemple, ajouter une interface graphique pour l'entrée des bases de règles et un moniteur pour voir l'état des agents. On peut aussi ajouter la sécurité et la mobilité forte et appliquer d'autres exemples avec ce Framework.

J'espère que vous utiliseriez AgentTo quand vous aurez à développer un système multi-agents.

# REFERENCES BIBLIOGRAPHIQUES


Understanding the Java ClassLoader, **ibm.com/developerWorks**

**Joseph P. Bigus, Jennifer Bigus, Joe Bigus**, Constructing Intelligent Agents Using Java: Professional Developer's Guide, 2nd Edition, Wiley(1998).

**Jim Farley**, Java Distributed Computing (O'Reilly Java) (1998)

**James W. cooper**, Java Design Patterns , Addison Wesley (2000)

**G. Weiss (Ed.),** ***Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*****, MIT Press, 1999.**

**Bourdon Alain, S. Kornman S. Pinson**, LAMSADE, Apprentissage des connaissances de contrôle en environnement **multi-agent**, MESR, 2001.

**Jean-Pierre Muller. H. Van Dyke Parunak**, Multi-agent systems and manufacturing, 9th Symposium on Information Control in Manufacturing INCOM98, France, 1998 Lmay.

**Steels, L. (1990).** Cooperation between distributed agents through self organization. In Demazeau, Y. and Müller. J.-P., editors, *Decentralized AI - Proceedings of the First European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds (MAAMAW-89)*, pages 175-196. Elsevier Science Publishers B.V.: Amsterdam, The Netherlands. Demazeau,1995

**Russell, S. J. and P. Norvig (1995):** ***Artificial Intelligence*****, A Modern Approach, Prentice Hall.**

**J. Ferber** - 1995 - Addison-Wesley, Multi-Agent Systems: Towards a Collective Intelligence

**Mark S. Fox, Norman Sadeh, and Can Baykan**. Constrained Heuristic Search. In *Proceedings of the Eleventh International Joint Conference on Artificial*

*Intelligence*, pages 309-315. 1989.

**TW Malone, J Yates, RI Benjamin** - Communications of the ACM, 1987.

**J Lind** - 2001 - Iterative Software Engineering for Multiagent Systems: The Massive Method, Springer-Verlag New York, Inc. Secaucus, NJ, USA

**Grand, Mark** 1999. Patterns in Java, Volume 1. Wiley.Hunter, Jason & Crawford, Java Servlet Pro-gramming. O'Reilly & Asssociates.

**Eckel**, B.: *Thinking in Java* (3rd edition). Prentice-Hall, 2001.

**J. Gama and P. Brazdil**. Linear tree. Intelligent Data Analysis, vol. 3(3), 1995.

**D. Y. Joh**, (1997), Security Aspects in Java Bytecode Engineering Blackhat Briefings 2002, Las Vegas.

http://www.enseeiht.fr/~queinnec/Ens/Chat/socket-java.html

**Pierre-Michel Ricordel**, Programmation Orientée Multi-Agents (2001) : Développement et Déploiement de Systèmes Multi-Agents. Voyelles