

Inventaires systématiques et sémantiques du patrimoine et des musées



Societas Cooperativa Europaea / Geneva



**Travail de Bachelor réalisé en vue de l'obtention
du titre Bachelor HES en Informatique de Gestion**

par :

Miguel TAVARES DOS SANTOS

Conseiller au travail de Bachelor :

Johann SIEVERING

Carouge, le 21 janvier 2013

Haute École de Gestion de Genève (HEG-GE)

Informatique de Gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l'examen final de la Haute école de gestion de Genève, en vue de l'obtention du titre de Bachelor HES en Informatique de Gestion. L'étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J'atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Carouge, le 21 janvier 2013

Miguel TAVARES DOS SANTOS

Remerciements

Premièrement, je souhaite remercier les personnes ayant fait partie de mon cursus scolaire au sein de la HEG en tant que camarades, assistants ou professeurs.

Deuxièmement, je remercie spécialement M. Johann SIEVERING de m'avoir suivi tout au long de ce travail et pour tous les conseils professionnels qu'il a pu me donner pendant ce mandat.

Troisièmement, je souhaite également remercier mes amis qui ont fait preuve d'une grande patience envers moi tout au long de ces années scolaires.

Pour terminer, je remercie ma famille et ma compagne pour leur confiance et soutien, et dédie ce mémoire à mon cousin Leandro TAVARES.

Résumé

Ce mémoire fait partie d'un projet découpé en 4 domaines dont le but est de rendre compatible des systèmes hétérogènes sur les plans technologiques, modèle de données et représentation métier.

En effet, il ne suffit pas de faire communiquer les bases de données pour que le système fonctionne. Il faut toujours se positionner d'un point de vue technologique (est-ce que je peux interroger deux bases différentes avec le même programme et surtout les réponses obtenues me satisfont-elles), modèle de données (est-ce que le schéma d'une base A est compatible avec le schéma d'une base B) et métier (le sens de la base A est-il le même que dans la base B). Il faut donc toujours assurer la compatibilité de ces trois couches. Il existe également les notions de droits (accès) et de mise-à-jour

Comme mentionnée ci-dessus, le projet contient 4 domaines : les interfaces agiles, les connecteurs, le système de communication par agent et l'ontologie internationale commune. L'objectif de ce travail concerne le troisième domaine du projet qui est « le système de communication par agent ». En d'autres mots, l'objectif est la mise en place d'un Bus Orienté Agent (BOA) permettant l'échange de données entre les bases de données du patrimoine et des musées.

Le mémoire se compose en 6 parties. La première partie du document présente le domaine de réalisation du projet. La deuxième porte sur « l'état de l'art » du patrimoine et des musées. La troisième explique le mandat qui m'a été confié ainsi que la contribution que je dois apporter au projet. La quatrième est consacrée à l'analyse et la modélisation de ma contribution. La cinquième partie se compose de la réalisation de mon projet. Enfin, dans la dernière partie, je finis par une conclusion portant sur l'évaluation du mandat qui m'a été confié.

Table des matières

Déclaration	i
Remerciements	ii
Résumé.....	iii
Table des matières	iv
Liste des Tableaux.....	vi
Liste des Figures	vi
Introduction.....	1
1. Domaine de réalisation du projet.....	2
2. « Etat de l’art »	3
2.1 Introduction.....	3
2.2 Système d’information	4
2.3 Acteurs	4
2.4 Technologie	4
2.5 Conclusion	5
3. Mandat.....	6
4. Analyse et modélisation du domaine	7
4.1 FIPA	7
4.2 Plateforme JADE	8
4.2.1 Introduction.....	8
4.2.2 Architecture logiciel.....	8
4.2.3 Fonctionnement.....	9
4.2.4 Outils de débogage.....	10
4.3 Les Agents	11
4.4 Agent Communication Language (ACL)	12
5. Réalisation du mandat	13
5.1 Choix de l’IDE (Environnement de Développement Intégré)	13
5.1.1 Mise en place de l’IDE	14
5.1.2 Première utilisation de JADE	15
5.2 Communication	16
5.2.1 Exemple de réception avec filtre	17
5.3 Services et comportements	18
5.3.1 Pages jaunes.....	18
5.3.1.1 Cas pratique d’utilisation d’un DF	20
5.3.2 Comportement (Behaviour).....	22
5.3.3 Librairie d’actes de communication (CAL).....	25
5.3.3.1 Actes de communication FIPA	26
5.4 Ontologie du BOA.....	31
5.4.1 Mise en place et fonctionnement	31
5.4.2 Prototype simulant un BOA.....	32

5.5	Normalisation du projet.....	35
5.6	Intégration.....	38
5.6.1	<i>Interface</i>	38
5.6.2	<i>Connecteur.....</i>	38
5.6.3	<i>Architecture multi-agents.....</i>	38
5.6.4	<i>Agents.....</i>	39
5.6.5	<i>Problèmes et Solutions.....</i>	40
5.6.5.1	Chemin d'accès aux fichiers jar.....	40
5.6.5.2	Déployer un projet Eclipse à la ligne de commande.....	40
5.6.5.3	Drivers OJDBC	41
5.6.5.4	Droit de connexion sur le serveur à Lausanne	41
5.6.5.5	Conclusion.....	42
	Conclusion	43
	Lexique	44
	Bibliographie.....	45
	Annexe 1 Planning du travail de Bachelor	46
	Annexe 2 Prototype SMA – Agents	1
	Annexe 2 Prototype SMA – Connexion BDD	7

Liste des Tableaux

Tableau 1	Message ACL	11
Tableau2	Choix de l'IDE	12

Liste des Figures

Figure 1	Planning du travail de Bachelor.....	6
Figure 2	Agent Management Reference Model	8
Figure 3	Concept de base JADE	10
Figure 4	Cycle de vie d'un agent JADE	11
Figure 5	JADE GUI.....	15
Figure 6	Test du nouvel agent.....	16
Figure 7	Schéma de réception d'un message	16
Figure 8	DFAgentDescription	18
Figure 9	Situation du cas pratique.....	20
Figure 10	Schéma d'interaction.....	21
Figure 11	Cycle d'exécution d'un agent.....	23
Figure 12	Bus Orienté Agent.....	31
Figure 13	Interfaces du prototype	32
Figure 14	Résultat final d'une requête.....	34
Figure 15	Schéma de normalisation.....	35
Figure 16	Schéma d'architecture système multi-agents	37
Figure 17	Arborescence des plateformes.....	38

Introduction

Agent Oriented Programming (AOP) en français : programmation orienté agent, est une façon de programmer un système qui doit être autonome, mobile et distribué. Cette manière de programmer modélise le système comme une collection de composants appelé agents qui sont caractérisés par l'autonomie, la capacité de communiquer, la proactivité et l'introspection. En étant autonome, ces agents peuvent être amenés à gérer des tâches complexes souvent sur du long terme. La proactivité leur permet de prendre la décision de commencer une tâche sans avoir besoin d'un ordre de la part d'un utilisateur. La capacité de communication permet une interaction avec d'autres agents qui permettra ainsi d'atteindre un but commun ou personnel. L'introspection permet à l'agent d'être conscient de son état.

Dans le premier paragraphe il est question d'agent, mais posons nous la question suivante : quelle définition peut-on donner à un « agent informatique » ? Le terme « agent informatique » est utilisé dans différents champs du domaine de l'informatique tels l'intelligence artificielle, les bases de données, les réseaux, etc. En base de données, l'agent informatique est défini comme étant une tâche de fond ou un processus dédié à une tâche précise. Au niveau des réseaux (gestion de réseaux), l'agent est placé sur une interface et permet de récupérer des informations sur différents objets. Pour le projet nous prendrons le sens suivant : processus autonome implémenté en Java possédant des comportements spécifiques, ayant la capacité d'interaction avec son environnement et pouvant échanger des messages.

Ce mémoire est structuré de la façon suivante : le chapitre 1 présente le domaine de réalisation du projet. Le chapitre 2 explique les caractéristiques du système actuellement mis en place. Le chapitre 3 contient les informations se rapportant au mandat confié et les buts devant être atteints. Au chapitre 4, une analyse et une modélisation du domaine sont effectuées. Le chapitre 5 porte sur le développement du travail à effectuer, il contient l'aspect technique et les étapes réalisées pour atteindre l'objectif. Enfin, le mémoire se termine avec une conclusion concernant l'ensemble du projet.

1. Domaine de réalisation du projet

Le mémoire est réalisé dans le cadre d'un projet de l'Association pour le Patrimoine Industriel¹ (API) et de la coopérative Social-IN³.

L'API, fondée en 1979, a pour mission la sauvegarde et la mise en valeur du patrimoine industriel de nos régions. Dans cette optique, elle a édité des ouvrages, organisé des séminaires, des rencontres et des manifestations, mis sur pied un musée pour attirer et favoriser l'intérêt de la conservation de ce patrimoine.

La cohérence voulue par l'API l'a conduite à ériger 4 piliers, à partir desquels toutes les réflexions et pratiques sont possibles :

- **Patrimoine** : la conservation d'objets mobiliers de la tradition des arts graphiques genevoise dans l'ancienne Usine de graisses industrielles Lambercier & Cie, construite en 1895
- **Culture** : la transmission des savoir-faire de la tradition des arts graphiques
- **Art** : la création contemporaine en couplant les techniques d'impression historique aux nouvelles technologies
- **Social** : une plateforme d'insertion sociale et professionnelle pour des demandeurs d'emplois et bénéficiaires de l'aide sociale, qui peuvent ainsi confronter leur métier pratiques

Le directeur de l'API (depuis 1998) est Monsieur Andréas Schweizer.

Actuellement, il est fait un recensement des différents objets, bâtiments et machines du patrimoine suite à la norme ISO 21127 portant sur « Une ontologie de référence pour l'échange d'informations du patrimoine culturel ». C'est sur la base de cette norme que la volonté d'un inventaire informatisé et pouvant donner une liberté d'échange interrégionale sur ces groupes d'items est désiré.

¹ Site internet de l'API : <http://www.patrimoineindustriel.ch/>

2. « Etat de l'art »

Ce chapitre est rédigé d'après l'article « *Semantic approach in the information systems* » de Johann Sievering.

Les systèmes d'informations prennent une importance de plus en plus grande dans les organisations. Actuellement, il existe déjà des solutions permettant l'échange de données et l'intégration de composants hétérogènes. La prochaine génération de solutions doit reposer sur une couche sémantique afin de permettre aux agents rationnels² cognitifs³ d'atteindre les objectifs des utilisateurs de manière autonome et indépendante du matériel ainsi que des infrastructures sous-jacentes.

2.1 Introduction

Les données et les traitements ont progressivement augmenté leur niveau d'abstraction tout comme les systèmes d'informations. Nous observons que le cadre de référence de l'interprétation des éléments se déplace petit à petit de la machine vers le monde réel. Néanmoins, les solutions proposées ne résolvent que les problèmes liés à la technique, comme par exemple les systèmes distribués, l'autonomie, la mobilité, etc.

Lors du développement d'application, on essaye de répondre à un besoin spécifique dans un environnement défini. La sémantique de l'ensemble des composants et les types de données ne sont connus qu'à l'intérieur de cet environnement, ce qui confine alors l'utilisation de cette application uniquement dans son domaine et pour les fonctionnalités prévues explicitement.

Ce qui est recherché pour palier à cette problématique, c'est une définition sémantique de plus haut niveau qui permettrait à des systèmes automatiques de traiter avec plus de finesse les données ainsi qu'une possibilité d'enrichir les comportements afin d'adapter dynamiquement ceux-ci selon le contexte opérationnel.

Dans notre projet, nous souhaitons mettre en place un Bus Orienté Agent qui nous permettra, indépendamment des technologies, la mise en place, d'échanges d'informations dans le domaine du patrimoine et les musées.

² Un agent rationnel possède un état qui va être modifié suite aux perceptions sur son environnement et de ses connaissances- Il va toujours baser son action la plus adapté pour s'approcher de son objectifs

³ Un agent cognitif possède un modèle sur lequel il est capable de raisonner pour accomplir une tâche (le plus classique étant BDI : Beliefs-Desires-Intentions)

2.2 Système d'information

Un système d'information est un ensemble organisé de ressources sur lequel il est possible d'exécuter un certain nombre d'opérations. Les acteurs qui gèrent cet ensemble de ressources, le divisent en « amas⁴ ». Ainsi, chaque acteur participe à l'évolution du système d'information en appliquant ses propres opérations et en utilisant des logiciels hétérogènes d'une manière non déterministe⁵.

2.3 Acteurs

Au sein de notre système d'information, les acteurs sont sémantiquement définis par des concepts et des relations nommées. A chaque intervention, la définition de l'acteur est incluse dans l'expression performative, ce qui permet au système d'adapter son comportement pour chaque contexte. Un performatif, dans le langage ACL⁶, est un type primitif de message. Dans le sens théorique de l'acte de parole, un performatif est un énoncé qui s'enclenche lorsque que le locuteur le dit ou l'affirme.

2.4 Technologie

Pour le développement de cette architecture, nous avons considéré un système d'information asynchrone avec des agents informatiques. Il est donc nécessaire que chaque composant soit autonome et qu'il puisse accomplir ses tâches même dans l'éventualité où la communication serait perdue. La plateforme multi-agent répond parfaitement à ces critères. En effet, elle nous permet de garder le même sens sémantique pour un objet tout au long de son parcours à travers les différentes couches. Il est aussi nécessaire que ces agents aient la capacité d'enregistrer leur profil sémantique dans la base de connaissance. En effet, les agents ont la capacité de s'appeler entre eux pour la recherche de compétences. Afin que cette recherche soit possible, chaque agent inscrit ses compétences dans un annuaire commun telles les pages jaunes.

⁴ Association de plusieurs ressources selon des critères définis

⁵ Pour une situation il y a plusieurs actions possibles

⁶ Langage de communication utilisé entre les agents

Nous avons choisi JADE (Java Agent DEvelopment framework) pour mettre en œuvre la plateforme multi-agents. Cette plateforme est entièrement écrite en Java et elle propose des outils de développement et d'administration. Le protocole de communication http correspond à ce que nous voulons faire dans notre projet. De plus, elle est compatible avec les standards FIPA. Nous pourrions donc utiliser le langage ACL (langage de communication agent). Chaque acte de langage est composé : d'un performatif, de l'expéditeur, du destinataire et du contenu relatif au performatif. D'autres informations complémentaires sont aussi disponibles dépendant de l'utilisation de l'acte de langage souhaitée.

2.5 Conclusion

Un système d'informations générique permet de faire communiquer des systèmes d'informations spécifiques hétérogènes.. Chacun peut être développé individuellement, mais l'élément clé est la base de connaissance qui doit être en mesure de fédérer les mondes logiques avec le monde physique. La prochaine étape est de mettre en place une architecture orienté agent permettant l'échange d'information à travers des requêtes sémantiques au sein du patrimoine et des musées.

3. Mandat

Le mandat qui m'a été confié est le suivant : mettre en place un système de Bus Orienté Agent (BOA) afin de rendre des systèmes hétérogènes compatibles.

Pour réaliser ce mandat, je suis encadré par Monsieur Johann Sievering. La durée de mon travail de Bachelor est fixée à 16 semaines à raison de 20 heures par semaine répartis selon le planning mis en place par Monsieur Sievering et moi-même.

Figure 1
Planning du travail de Bachelor

Planning hebdomadaire

Lundi	Mardi	Mercredi	Jeudi	Vendredi	Samedi	Dimanche
Cours d'algorithmie et programmation	Bachelor (4h)	Bachelor (4h)	Professionnel	Professionnel	OFF	Professionnel
Bachelor (4h)	Cours de Recherche Opérationnelle	TP	Bachelor (4h)	Bachelor (4h)	OFF	Professionnel

Début du projet : 24. octobre 2012

Fin du projet : 21. janvier 2013

Soutenance : 30. janvier 2013

Afin de structurer ces 16 semaines⁷, divers objectifs ont été fixés afin d'atteindre le résultat recherché par le mandat.

Les semaines 1 et 2 sont consacrées à la maîtrise de la plateforme JADE (littérature et tutoriels) avec une présentation le lundi 8 octobre 2012 aux membres de l'équipe de travail. Lors de la semaine 3, je dois acquérir les compétences concernant le protocole de communication de la FIPA. Lors de la semaine 4, je prends en main la programmation d'Agent Informatique en langage JAVA. La semaine 5 est dédiée à l'avancement de la rédaction de mon mémoire. Les semaines 6 et 7 sont consacrées à la compréhension du protocole « Musée » et à l'appropriation du langage ACL. Les semaines 8 et 9 ont pour but de comprendre l'ontologie du bus. Les semaines 10 et 11 doivent mettre en place les requêtes sémantiques afin de permettre la communication entre les différents agents. En semaine 12 et 13, les objectifs sont respectivement la normalisation du domaine et son intégration dans le projet global. Les semaines 14 et 15 sont consacrées à la continuité de la rédaction du mémoire. Enfin, la semaine 16 est pour la mise au net des documents et le rendu final du projet.

⁷ Décrit dans l'annexe « Planning »

4. Analyse et modélisation du domaine

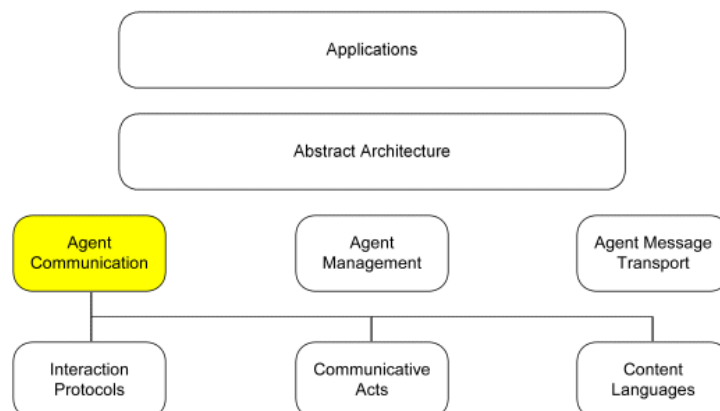
4.1 FIPA

FIPA (Foundation for Intelligent Physical Agents) en français : fondation pour l'intelligence physique des agents, est une organisation à but non lucratif. Créée en 1996, son but est de produire des standards pour l'interopération d'agents logiciels hétérogènes. Grâce à la combinaison d'actes de langages, de logiques des prédicats et d'ontologies publiques, cette organisation veut mettre à disposition des moyens standardisés qui permettent d'interpréter les communications entre agents de telle façon que l'on respecte leur sens initial. Les différents domaines où l'on peut avoir des standards donné par la FIPA sont :

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

Ces standards sont toujours mis à jour. La FIPA ne veut pas créer une plateforme de construction multi-agents, elle cherche à normaliser la plateforme de sorte que lors de l'exécution, on bénéficie d'un système interopérable.

Actuellement, le nombre de membre de la FIPA s'élève à 60⁸, ce qui représente environ 17 pays à travers le monde.



⁸ Source : <http://www.fipa.org/about/joining.html> (11.10.2012)

4.2 Plateforme JADE

4.2.1 Introduction

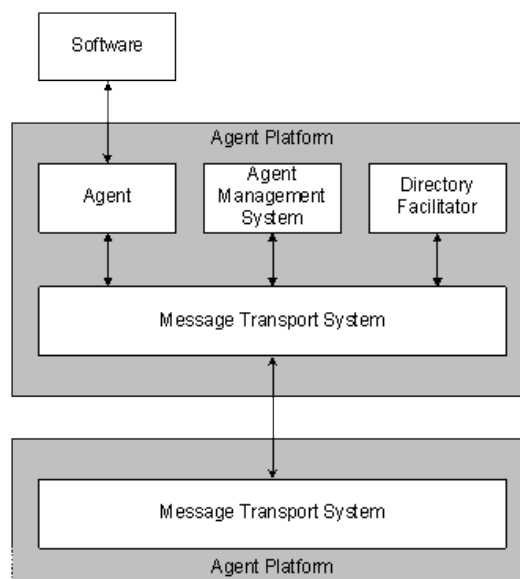
JADE (Java Agent DEvelopment Framework) est une plateforme entièrement implémentée en Java qui permet la mise en place de Système Multi-Agent (SMA). JADE est une instance de la FIPA (Foundation for Intelligence Physical Agent) et fournit une multitude de classes qui implémentent le comportement des agents qu'elle crée.

Pour utiliser cette plateforme nous avons le choix de la lancer à la ligne de commande et ensuite de créer les classes java depuis un éditeur de code source (EditPlus, NotePad++, etc.) ou d'utiliser un environnement de développement intégré (IDE). Afin de réaliser ce projet, l'option de l'IDE a été retenue.

4.2.2 Architecture logiciel

JADE se base sur le modèle d'architecture fourni par la FIPA, l'Agent Management Reference Model. Il établit le modèle de référence logique pour la création, l'enregistrement, la localisation, la communication, la migration et l'arrêt des agents. Dans la figure ci-dessous, nous pouvons voir plusieurs éléments que l'on va appeler services. Les services dont on dispose à la base sont le Directory Facilitator (DF) et l'Agent Management System (AMS). Il est bien sûr possible de demander le chargement au démarrage du Message Transport System (MTS) pour permettre la communication entre plusieurs plates-formes, mais le modèle de base ne le fait pas afin de garder par défaut les fonctionnalités adaptées à tout type d'utilisation du modèle.

Figure 2
Agent Management Reference Model



L'agent a un rôle essentiel sur la plateforme, il est identifié grâce à un Agent Identifier (AID) de manière unique.

Le DF fournit un service de « pages jaunes » afin de pouvoir mettre en relation un agent avec les compétences qui lui incombent. Un agent peut faire appel en tout temps au DF afin d'enregistrer ses comportements ou d'obtenir des informations sur d'autres comportements d'agents. L'AMS quant à lui contrôle l'accès et l'utilisation de la plateforme. Ce composant possède et maintient un répertoire contenant les adresses de transport des agents disponibles sur la plateforme. On appellera ce service les « pages blanches », il effectuera le lien entre un agent et un AID.

Lorsqu'un agent est créé, il doit s'enregistrer sur un AMS afin d'obtenir un AID. Il ne peut y avoir qu'un AMS par plateforme. Le MTS (service de transport de message) est une méthode implantée par défaut afin de fournir un moyen de communication entre deux plateformes. L'Agent Platform (AP) est la structure physique sur laquelle les agents peuvent vivre.

4.2.3 Fonctionnement

JADE contient 3 éléments essentiels à son fonctionnement :

- Un runtime Environment
- Une librairie de classes
- Une suite d'outils graphique

Le runtime Environment représente l'environnement où les agents sont exécutés et gérés. Il est essentiel que ce composant soit activé pour que les agents puissent être créés et fonctionner.

La librairie de classes servira aux développeurs à créer et développer leurs agents.

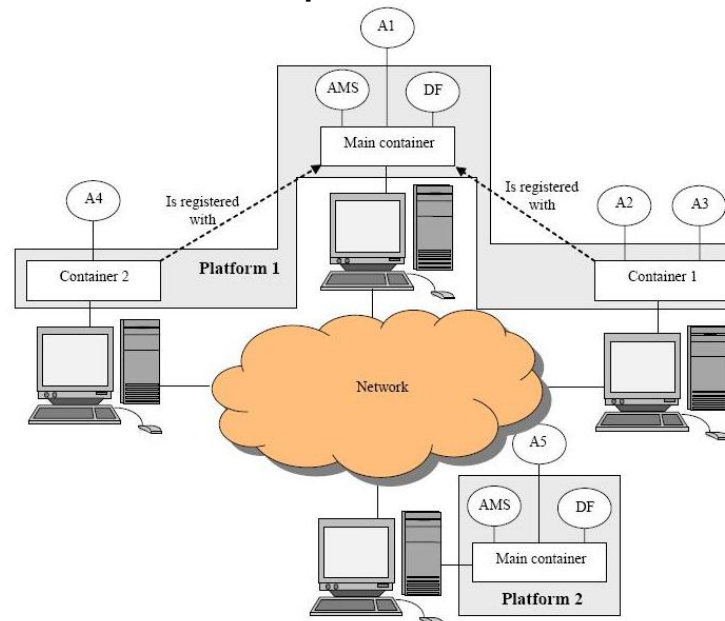
La suite d'outils graphique facilitera la gestion et la supervision de la plateforme des agents.

Chaque instance de JADE est appelé un « Container », et peut contenir plusieurs agents. Plusieurs « Container » ensemble prennent la dénomination de « Plateforme ». Chaque « Plateforme » doit avoir un « Main-Container » auprès duquel iront s'enregistrer les autres « Containers » dès leur lancement.

Le « Main-Container » est le seul conteneur qui possède toujours deux agents spéciaux nommés AMS et DF. Ils sont toujours lancés automatiquement au démarrage du « Main-Container ».

La figure ci-dessous illustre les propos évoqués durant ce sous-chapitre.

Figure 3
Concept de base JADE



4.2.4 Outils de débogage

Afin de déboguer la plateforme, JADE met à disposition trois outils :

- Dummy Agent
- Sniffer Agent
- Introspector Agent

Le « Dummy Agent » va nous servir dans deux contextes. Le premier, pour visualiser les messages pour d'un agent. Le second pour envoyer des messages aux agents présents sur la plateforme et réceptionner leur réponse.

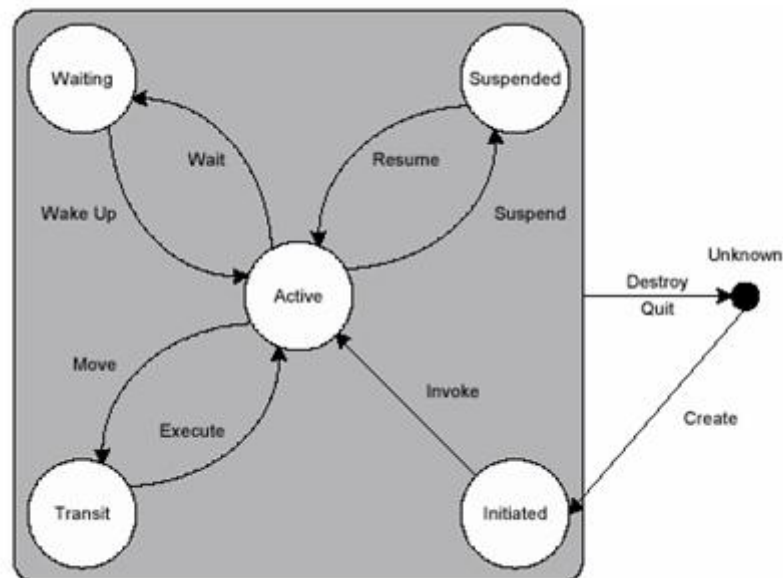
Le « Sniffer Agent » permet la visualisation de l'enchaînement de messages et la vérification interactive de la correction des protocoles.

Enfin, grâce à l' « Introspector Agent », nous pourrions contrôler l'état de des agents, de visualiser les comportements actifs et non-actifs de ceux-ci et de visualiser les messages envoyés et reçus entre plusieurs agents.

4.3 Les Agents

Un Agent JADE est un thread en JAVA. Il possède des comportements implémentant des services ou des fonctionnalités et suit un cycle de vie suivant :

Figure 4
Cycle de vie d'un agent JADE



Un Agent JADE hérite de la classe Agent. Il a la possibilité de s'inscrire ou de rechercher un service.

Un service peut être

- Une action enregistrée et délivrée par la plateforme
- Un comportement d'un ou plusieurs agents répondant à la demande
- Gérer par des « pages jaunes » (DF)
- Semblable à la notion de Web Services

L'Agent JADE possède une méthode *setup()* qui est invoquée dès la création de l'agent et une méthode *takedown()* appelée avant qu'un agent ne quitte la plateforme (passe à l'état « détruit »).

4.4 Agent Communication Language (ACL)

Le langage de communication utilisé par JADE est le FIPA-ACL (agent communication language). Le nombre de paramètres que l'on va mettre dans le message peut varier selon la situation dans laquelle se trouve l'agent, le seul paramètre obligatoire est celui de la requête *performative* (actes de langages). Wikipédia nous dit que la notion d'acte de langage⁹, est un moyen mis en œuvre par un locuteur afin d'agir sur son environnement par ses mots, cherchant à informer, inciter, demander, convaincre, promettre, etc. son ou ses interlocuteurs par ce moyen. Dans le cas où un agent ne comprend pas le message, il peut répondre avec un message *not-understood* (pas compris).

A chaque envoi de message, l'agent crée un nouvel objet de type `ACLMessage`. Ensuite pour procéder à son envoi, il faut utiliser la méthode `send()`, en français : envoyer. Si un agent veut recevoir un message, il doit alors utiliser la méthode `receive()`, en français : recevoir ou au contraire le bloquer avec la méthode `blockingreceive()` en français : bloquer réception. Afin de rechercher ou modifier les champs de l'objet `ACLMessage`, l'agent utilise les méthodes `set()` et `get()` en français respectivement *modifier* et *obtenir*. La communication des messages se fait de manière asynchrone (la communication ne se passe pas à la même vitesse pour tous les agents).

Le tableau ci-dessous présente les champs disponibles dans un message ACL.

Tableau 1
Message ACL

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

⁹ Source : http://fr.wikipedia.org/wiki/Acte_de_langage

5. Réalisation du mandat

5.1 Choix de l'IDE (Environnement de Développement Intégré)

Parmi les IDE connus, pour ce projet les choix possibles sont NetBeans et Eclipse qui sont les plus utilisés au sein de l'équipe projet.

Tableau 2
Choix de l'IDE

<i>Solutions</i>	Critères				
	<i>Plugin</i> Coef. : 0.5	<i>Intégration</i> Coef.: 0.8	<i>Développement</i> Coef.: 1	<i>Interface</i> Coef.: 0.5	<i>Langage</i> Coef.: 0.5
ECLIPSE	1	2	2	1	2
NETBEANS	2	1	1	2	2

Analyse de la matrice de choix

Au niveau des plugins, avec Eclipse, il faut entrer un site distant afin d'obtenir le plugin et cela peut être assez long au niveau du temps de chargement. Pour NetBeans, c'est plus simple, il faut télécharger le package contenant les librairies de JADE et ensuite il faut ajouter les librairies dans la configuration NetBeans.

Pour l'intégration, les deux IDE fonctionnent correctement avec JADE. Concernant le développement, il s'agit plutôt d'une préférence du développeur selon l'environnement qu'il connaît le mieux.

Concernant l'interface, NetBeans est plus léger qu'Eclipse. Il est moins encombrant et plus facile pour la création d'interface.

Enfin pour le critère du langage, les deux IDE supportent les langages connus et surtout le JAVA qui sera nécessaire pour la réalisation du mandat.

En conclusion, NetBeans ou Eclipse sont semblables dans l'ensemble et ne présentent pas de grandes différences entre eux, c'est pourquoi il tient au développeur de choisir l'IDE avec lequel il possède une plus grande aisance et maîtrise.

Pour ma part, après avoir utilisé le système des pondérations, le résultat pour Eclipse est de 5.6 et NetBeans obtient 4.8. Je vais donc opter pour l'IDE d'Eclipse.

5.1.1 Mise en place de l'IDE

Ce sous-chapitre va expliquer comment mettre en place le plugin JADE sous Eclipse Indigo Sr2.

Pré-requis :

- [Eclipse Indigo Sr2](#)
- JDK 1.6.0_14
- Connexion internet

Voici la marche à suivre détaillé :

1. Démarrer Eclipse
2. Menu « Help » puis cliquez sur « Install New Software »
3. Saisir l'adresse du lien de plugin :
<http://dit.unitn.it/~dnguyen/ejade/update>
4. Attendre le chargement et cocher le plugin EJADE puis cliquer sur « Next »
5. Vérifier que la coche concernant EJADE est cochée et cliquez sur « Finish »

Une fois le plugin installé, vous verrez apparaître dans votre barre de menu les deux icônes concernant JADE. Nous pouvons dès à présent utiliser JADE sous Eclipse.

Pour créer un nouveau projet JADE, on doit créer un nouveau projet JAVA. Une fois que cela est fait, faites un clic droit sur le projet puis sélectionner « Toggle JADE/JADEX agent nature ». Avec cette manipulation, vous additionnez les librairies JADE et avez maintenant un projet JADE prêt à être exécuté.

Maintenant, il faut ajouter une variable d'environnement afin de permettre, lors du démarrage de la plateforme depuis une invite de commandes, de retrouver les fichiers jar (contenant les classes compilées) de jade nécessaire à son lancement.

Pour rajouter une variable d'environnement, il faut faire les choses suivantes :

Clic droit sur le poste de travail → Propriétés → Paramètres systèmes avancés
→ Variable d'environnement

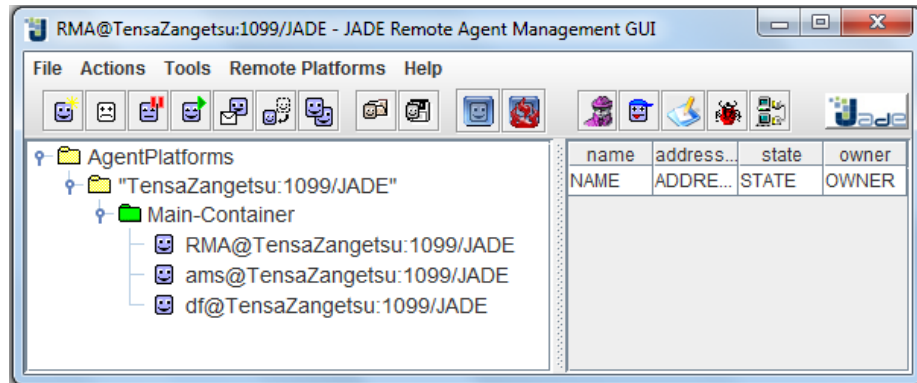
Ensuite si vous avez déjà une variable avec le nom CLASSPATH, mettez-la à jour en introduisant les lignes suivantes :

```
C:\Program Files\jade\lib\jade.jar;    C:\Program Files\jade\lib\jadeTools.jar;  
C:\Program Files\jade\lib\http.jar;    C:\Program Files\jade\lib\iiop.jar;  
C:\Program Files\jade\lib\commons-codec\commons-codec-1.3.jar;
```

5.1.2 Première utilisation de JADE

Premièrement, voici comment la plateforme JADE, une fois lancée, se présente.

Figure 5
JADE GUI



Grâce à elle, nous pouvons effectuer diverses actions sur les agents ou la plateforme. Nous pouvons aussi mettre en place un « sniffer » et observer les échanges de messages entre agents. Nous disposons de l'arborescence de la plateforme. Le « Main-Container » contient les agents spéciaux AMS et DF comme évoqué précédemment.

Pour créer un agent, nous avons soit la possibilité de choisir une classe générique dans la librairie des classes ou créer notre propre classe agent. Pour cette démonstration, je vais créer une classe agent qui sera en charge d'écrire dans le log « Bonjour ! Je suis nomAgent ».

Pour ce faire voici la classe créé¹⁰ :

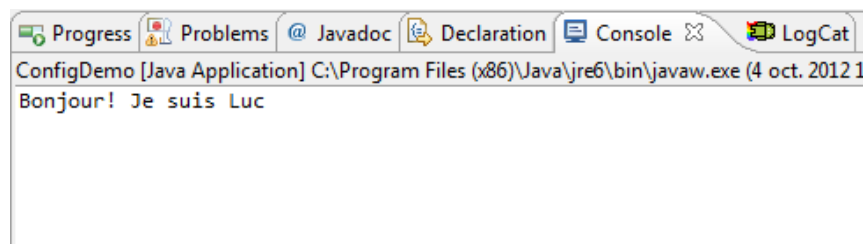
```
public class HelloAgent extends Agent {  
  
    // Première méthode exécutée lors du lancement de l'agent  
    protected void setup() {  
        System.out.println("Bonjour! Je suis " + getLocalName());  
        doDelete();  
    } // setup fin  
  
} // HelloAgent fin
```

¹⁰ Les classes se trouvent toutes dans le répertoire « src » du projet.

Maintenant depuis l'interface JADE, il faut sélectionner le démarrage d'un nouvel agent. Depuis la nouvelle fenêtre, il faut cliquer sur les « ... » afin d'ouvrir le repertoire des classes que JADE possède et choisir la nouvelle classe créée portant le nom de « HelloWorldAgent.HelloAgent ». Puis nous choisissons un nom à donner à notre agent ici pour l'exemple « Luc » et enfin valider le tout en appuyant sur « Ok ».

L'agent « Luc » est crée et sur le log d'Eclipse nous pouvons lire la phrase « Bonjour ! Je suis Luc ».

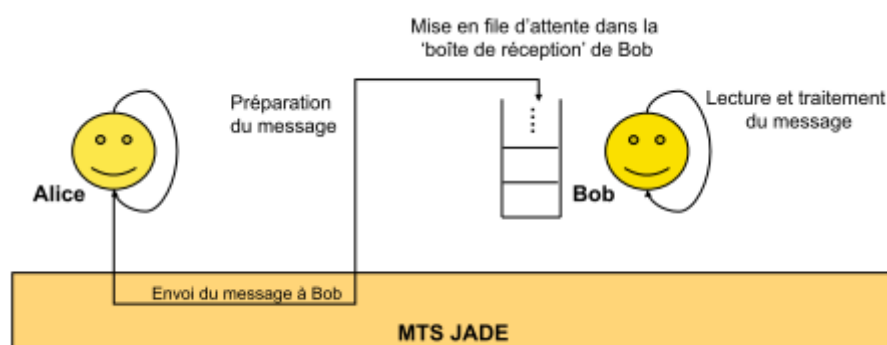
Figure 6
Test du nouvel agent



5.2 Communication

Après avoir créé notre premier agent, voyons comment il peut recevoir un message. Pour illustrer le schéma de réception d'un message¹¹ observons la figure ci-dessous.

Figure 7
Schéma de réception d'un message



¹¹ Source : <http://www.emse.fr/~boissier/enseignement/maop11/courses/jade-prog-4pp.pdf>

Lors de la réception d'un message, nous avons la possibilité d'avoir deux types de réceptions, la réception bloquante et la réception non-bloquante.

Pour avoir une réception bloquante, il faut utiliser la méthode :

- `ACLMessage message = blockingReceive()`

Cette méthode a pour effet de mettre l'agent dans l'état *Waiting*. Attention, il faut observer l'arrêt de tous les comportements jusqu'à la réception d'un nouveau message.

Pour avoir une réception non-bloquante, on utilise la méthode :

- `ACLMessage message = receive()`

Cette méthode nous retourne *null* si la file de message est vide. L'agent reste dans son état *Active*. Son comportement va se dérouler normalement même s'il n'a pas reçu de message. Afin de suspendre le comportement, on utilisera la méthode *block()* jusqu'à la réception d'un message. L'agent ne changera pas d'état.

Par défaut, les méthodes *blockingReceive()* et *block()* récupèrent l'ensemble des messages présents dans la file d'attente. On a la possibilité de créer des filtres avec la classe *MessageTemplate*.

5.2.1 Exemple de réception avec filtre

```
MessageTemplate m = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
```

```
MessageTemplate protmt = MessageTemplate.MatchProtocol("ping");
```

```
MessageTemplate template = MessageTemplate.and(m, protmt);
```

```
ACLMessage receive = receive(template);
```

```
if (receive != null){
```

```
    ACLMessage inform = receive.createReply();
```

```
    inform.setPerformative(ACLMessage.INFORM);
```

```
    inform.setContent("pong");
```

```
    send(inform);
```

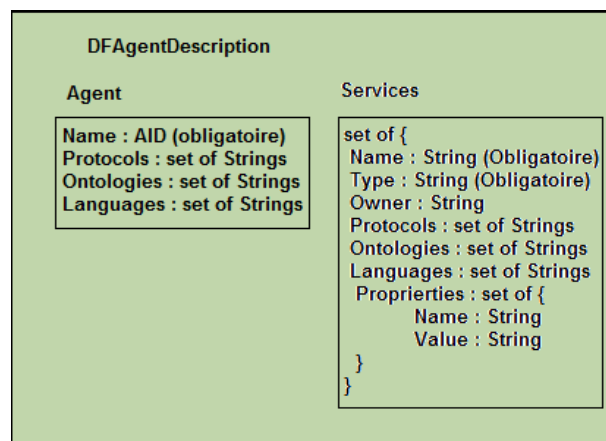
```
} else {block();}
```

5.3 Services et comportements

5.3.1 Pages jaunes

Comme expliqué précédemment, le DF est un agent qui gère l'annuaire de pages jaunes. « Les pages jaunes » sont un registre centralisé qui va lier l'ID d'un agent à ses services. Afin de créer et de gérer les différentes entrées, il faut utiliser l'objet `DFAgentDescription`. Pour enregistrer un service, il suffit de donner une description du service ainsi que l'AID de l'agent qui le fournit. Pour rechercher un service, il suffit de donner la description du service que l'on désire.

Figure 8
DFAgentDescription



Au niveau du code, pour enregistrer un service il faut écrire :

```
protected void setup(){
//Création d'une instance dans le DF puis l'agent qui fournit le service
    DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());

// Création d'un service
    ServiceDescription sd = new ServiceDescription();

// On définit un type et un nom de service puis on l'ajoute aux pages jaunes
    sd.setType("Type"); sd.setName("Service 1");
    dfd.addServices(sd);

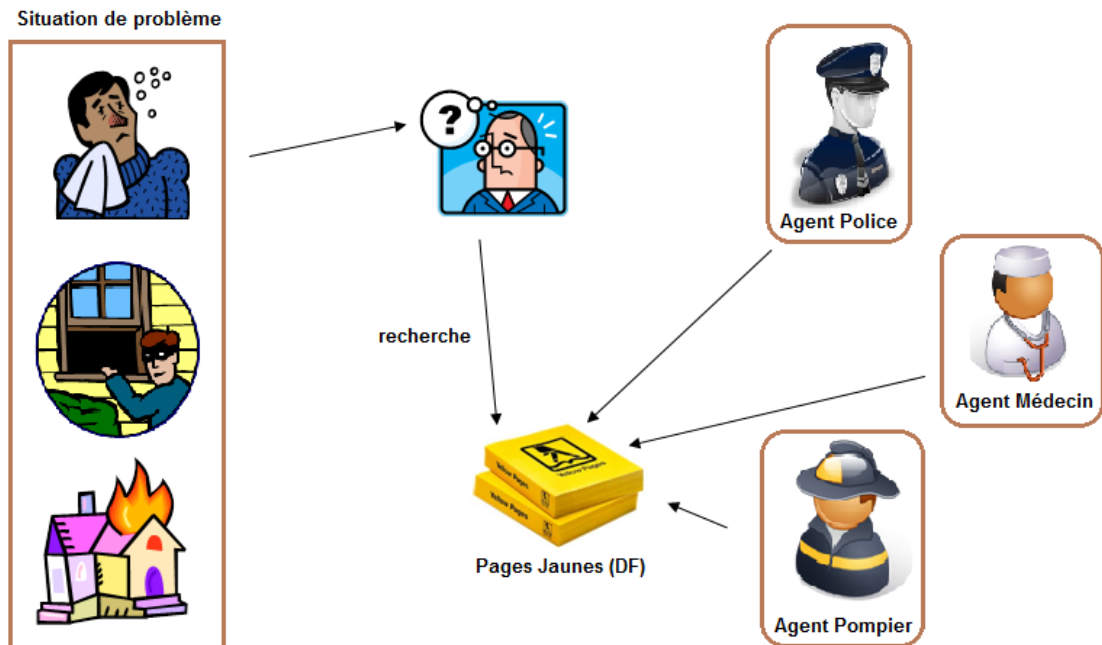
//Enregistrement de l'agent dans le DF
    try {
// Agent qui fournit le service avec la description
        DFService.register(this, dfd);
    } catch (Exception e) {
        e.printStackTrace();
    }
} // fin setup()
```

La structure de la figure 7 peut paraître complexe mais en réalité elle est très facile à manipuler étant donné que seuls quelques champs sont obligatoires. Pour chaque service publié dans l'annuaire, il est obligatoire de fournir l'AID de l'agent, le type et le nom du service. Les méthodes disponibles afin de manipuler cet objet sont les suivantes :

- `register()` enregistre les services d'un agent dans le DF;
- `deregister()` élimine les services d'agent du DF;
- les services sont définis avec les méthodes de la classe *ServiceDescription*
 - `setName()` défini le nom du service;
 - `setOwnership()` défini le propriétaire du service;
 - `setType()` défini le type du service;
 - `addLanguage()` ajoute des langues au service;
 - `addOntologie()` ajoute des ontologies au service;
 - `addProtocols()` ajoute des protocoles au service;
 - `addProperties()` ajoute des propriétés au service;
- la description de l'agent qui fournit le service s'effectue avec les méthodes de la classe *DFAgentDescription*
 - `setName()` défini le nom de l'agent;
 - `addServices()` ajoute des services;
 - `removeServices()` supprime des services;
 - `addLanguages()` ajoute des langues que l'agent comprend;
 - `addProtocols()` ajoute des protocoles;
 - `addOntologies()` ajoute des ontologies que l'agent manipule.

5.3.1.1 Cas pratique d'utilisation d'un DF

Figure 9
Situation du cas pratique



Comme illustré ci-dessus, considérons un agent Observateur qui va être attentif à la « situation de problème » dans son environnement tel qu'un vol, un incendie ou une personne malade. Cet agent recherche alors d'autres agents capables de l'aider selon la compétence, dans notre exemple, l'Agent Police, l'Agent Médecin et l'Agent Pompier. Dans l'optique Multi-Agent, l'agent Observateur va rechercher dans le DF de la plateforme des agents qui offre un service déterminé. Ces agents là doivent, lorsqu'ils arrivent sur la plateforme, enregistrer leur service dans le DF.

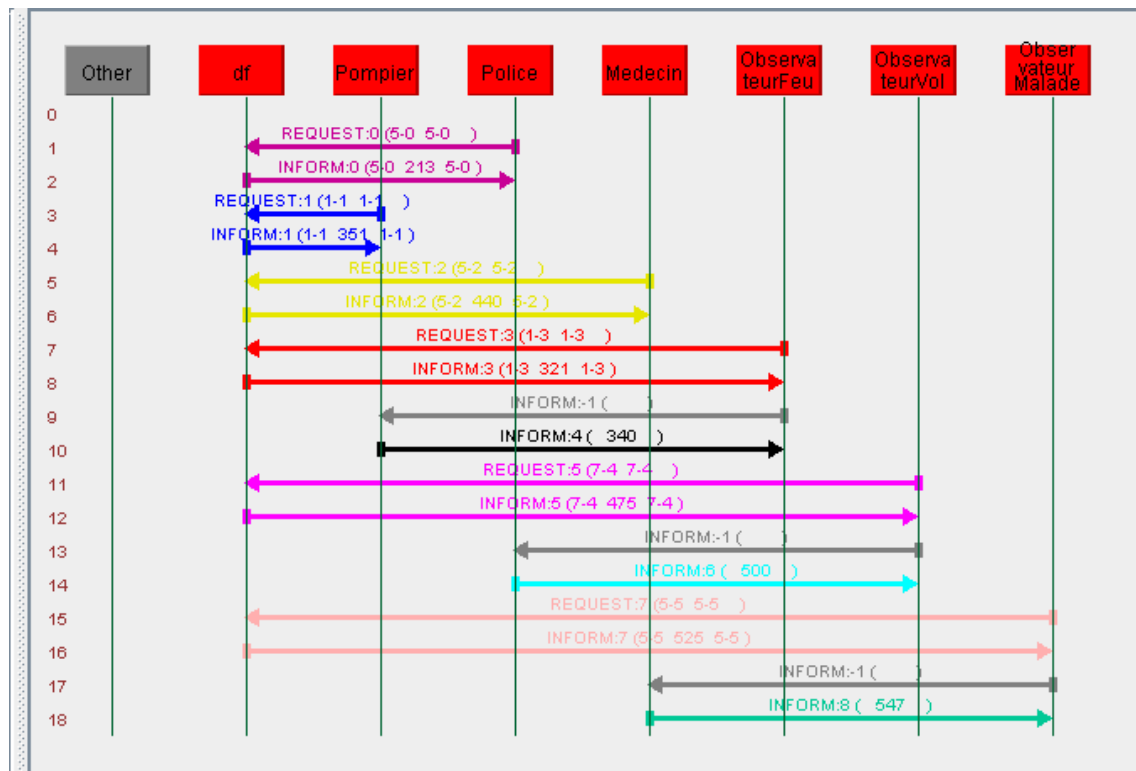
Les objectifs de cette démonstration sont :

- Enregistrement de services dans le DF ;
- Recherche d'un service par rapport à la compétence ;
- Communication simple entre agent ;

Afin de réaliser cet exemple, il faut créer les classes agents Observateur, Police, Médecin et Pompier. Le code spécifique à ces classes se trouve en **annexe**.

Après exécution de l'exemple nous obtenons grâce à l'interface *Agent Sniffer* (cette interface permet de tracer les interactions ayant lieu sur la plateforme) le schéma illustré ci-dessous.

Figure 10
Schéma d'interaction



Lors du démarrage des agents Police, Pompier et Médecin, ils envoient une requête afin d'enregistrer leurs services auprès du DF. Une fois enregistré le DF les informe. Lors de l'arrivée de l'ObservateurFeu, il envoie une requête au DF avec l'argument « feu » afin de trouver l'agent qui fournit le service correspondant à son argument. Le DF (si le service est enregistré) indique à l'ObservateurFeu les coordonnées de l'agent qui fournit le service. Celui-ci entre en contact et informe l'agent Pompier qui répondra avec un message de confirmation, ici « Je vais éteindre l'incendie ». Le même schéma se produit entre les autres agents observateurs.

5.3.2 Comportement (Behaviour)

Chaque action réalisée par un agent est représenté comme un comportement de l'agent. Pour définir un nouveau comportement, il faut créer une nouvelle classe qui hérite de la classe *jade.core.behaviours.Behaviour*. Une fois le comportement défini, il faut l'appeler depuis la classe de l'agent (dans la méthode *setup()*) qui est censé le réaliser. Pour ajouter le comportement, la méthode à utiliser est *addBehaviour()*. Comme exemple, voici un code représentant l'implantation d'un comportement au sein d'une classe Agent.

```
public class MonAgent extends Agent{
    protected void setup(){
        addBehaviour(new MonComportement(this));
    } // fin setup()
} //fin MonAgent
```

Une classe désignant un comportement doit toujours avoir les méthodes suivantes :

- Action() – code du comportement qui sera exécuté par l'agent ;
- Done() – retourne un booléen pour indiquer si le comportement est terminé ou non.

Intéressons nous à la classe *MonComportement* ci-dessous.

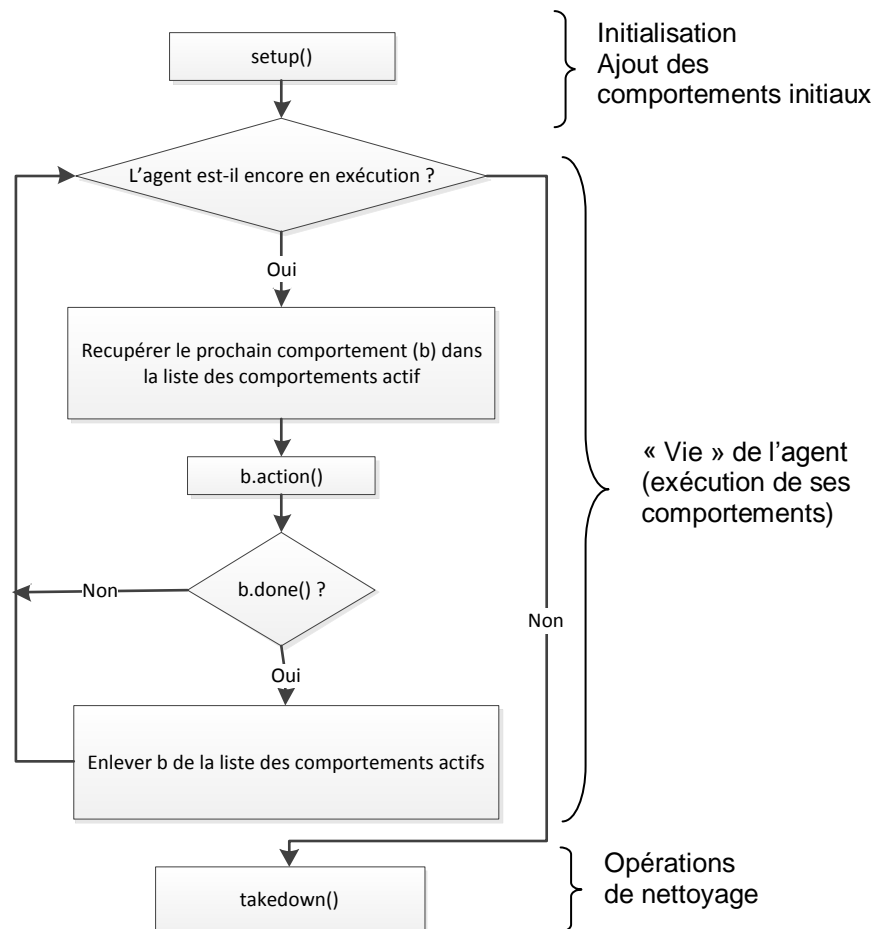
```
public class MonComportement extends Behaviour {
    int i = 0;
    // Constructeur
    public MonComportement(Agent a){super(a);}

    @Override
    public void action() {
        System.out.println("Bonjour à tous, mon nom est " +
            myAgent.getLocalName());
        i+=1;
    } // fin action()

    @Override
    public boolean done() {return i > 4;}
}
```

Le comportement défini imprimera 5 phrases dans le log. Dès que la variable *i* est plus grande que 4, la méthode *done()* passe à vrai et le comportement se termine. Dans le constructeur de la classe *MonComportement*, la méthode *super(a)* est appelée afin d'initialiser la variable *myAgent* utilisée dans la mise en place du comportement.

Figure 11
Cycle d'exécution d'un agent



Un agent peut exécuter plusieurs comportements. Un comportement est exécuté jusqu'à ce que l'agent arrive à la fin de la méthode *action()*. Toutefois, c'est le programmeur qui définit quand il faut passer d'un comportement à un autre. Sur la figure représentant le cycle de vie d'un agent¹², il est démontré clairement qu'il existe 3 niveaux :

1. Initialisation – exécution de la méthode *action()* et ajout des comportements initiaux de l'agent ;
2. « Vie » de l'agent – exécution des comportements de l'agent ;
3. Opération de nettoyage – exécution de la méthode *takedown()* pour la finalisation de l'agent.

¹² Source : *Developing multi-agents systems with JADE*

Dans le chapitre des comportements, il existe des comportements dans JADE que l'on appellera « comportement prédéfini ». Ces comportements sont au nombre de trois :

Les comportements « one-shot » s'exécutent de manière quasi instantanée et une seule fois. La classe **jade.core.behaviours.OneShotBehaviour** implémente déjà la méthode *done()* qui retourne true (vrai).

```
public class MonComportementOneShot extends OneShotBehaviour{

    @Override
    public void action() {

        // executer opération X

    } // fin action()
} // fin MonComportementOneShot
```

Ici l'opération X n'est exécutée qu'une fois.

Les comportements « cyclic » sont conçus pour ne jamais se terminer ; leur méthode *action()* exécute la même opération chaque fois qu'elle est appelée. La classe **jade.core.behaviours.CyclicBehaviour** implémente déjà la méthode *done()* qui retourne false (faux).

```
public class MonComportementCyclique extends CyclicBehaviour {

    @Override
    public void action() {

        // exécution de l'opération Y

    } // fin action()
} // fin MonComportement Cyclique
```

Ici l'opération Y est exécutée en continue jusqu'à ce que l'agent qui déclenche le comportement se termine.

Les comportements génériques intègrent la notion de trigger et exécutent différentes opérations selon l'état de la valeur.

```
public class MonComportementGenerique extends Behaviour{
    int step = 0;

    @Override
    public void action() {
        switch (step) {
            case 0:
                // opération X
                step++;
                break;
            case 1:
                // opération Y
                step++;
                break;
            case 2:
                // opération Z
                step++;
                break;
        }
    }

    @Override
    public boolean done() {
        return step == 3;
    }
}
```

Dans cet exemple, la variable *step* sert à représenter le statut du comportement. Les opérations X, Y et Z sont exécutées séquentiellement ensuite le comportement se termine.

5.3.3 Librairie d'actes de communication (CAL)

Au niveau de la communication entre agents, la FIPA a édicté des standards reposant sur des actes de communication. Les objectifs de cette standardisation sont les suivants :

- Assurer une interopérabilité en fournissant un set de composants dérivés des actes de communications primitifs de la FIPA,
- Faciliter la réutilisation des composants,
- Fournir un processus bien défini pour le maintien d'un ensemble d'actes de communications utilisés dans le protocole de communication FIPA ACL.

5.3.3.1 Actes de communication FIPA

La liste des actes de communication n'est pas exhaustive, elle ne contient que ceux qui sont importants dans un premier temps pour le projet. Les références sont tirées du document : « *FIPA Communicative Act Library Specification* ».

Nom : Accept Proposal (accepter la proposition)

Description : Accepter une proposition faite par un autre agent. L'agent qui envoie l'acceptation informe l'agent récepteur qu'il effectuera (à un certain moment dans le futur) l'action, une fois que la condition fixée passe à vrai.

Exemple : L'agent *i* informe *j* qu'il a accepté l'offre de *j* de diffuser un fichier multimédia sur la chaîne 19 quand le client sera prêt. L'agent *i* informera *j* lorsque cela sera le cas.

```
(accept-proposal
:sender (agent-identifier :name i)
:receiver (set (agent-identifier :name j))
:in-reply-to bid089
:content
  "((action (agent-identifier :name j)
    (stream-content movie1234 19))
    (B (agent-identifier :name j)
      (ready customer78)))"
:language fipa-sl)
```

Nom : Agree (se mettre d'accord)

Description : Cet acte de langage est un accord d'usage général à une demande qui se finalise par l'accomplissement d'une action. L'agent qui envoie l'accord, informe le récepteur qu'il a l'intention d'exécuter l'action lorsque sa condition de réalisation sera vrai. Cet acte peut être utilisé pour informer le récepteur lorsque l'agent exécutera l'action demandée.

Exemple : L'agent *i* demande à *j* de délivrer une boîte à un certain endroit ; *j* répond qu'il est d'accord avec la requête mais que la priorité est basse.

```
(request
:sender (agent-identifier :name i)
:receiver (set (agent-identifier :name j))
:content
  "((action (agent-identifier :name j)
    (deliver box017 (loc 12 19))))"
:protocol fipa-request
:language fipa-sl
:reply-with order567) (agree
:sender (agent-identifier :name j)
:receiver (set (agent-identifier :name i))
:content
  "((action (agent-identifier :name j)
    (deliver box017 (loc 12 19)))
    (priority order567 low))"
:in-reply-to order567
:protocol fipa-request
:language fipa-sl)
```

Nom : Cancel (annuler)

Description : Cancel permet à un agent *i* d'informer un agent *j* que *i* ne désire plus que *j* exécute l'action demandé par la requête envoyée précédemment.

Exemple : L'agent *j* demande à l'agent *i* d'annuler une précédente requête en citant à chaque fois l'action.

```
(cancel
  :sender (agent-identifier :name j)
  :receiver (set (agent-identifier :name i))
  :content
    "((action (agent-identifier :name j)
      (request-whenver
        :sender (agent-identifier :name j)
        :receiver (set (agent-identifier :name i))
        :content2
        \"((action (agent-identifier :name i)
          (inform-ref
            :sender (agent-identifier :name i)
            :receiver (set (agent-identifier :name j))
            :content3
            \"((iota ?x
              (= (price widget) ?x))\\")
              (> (price widget) 50))\"
              ...)))\"
        :langage fipa-sl
        ...)
```

Nom : Call for proposal ou cfp (appel de proposition)

Description : cfp est utilisé pour débiter un processus de négociations en faisant un appel à propositions pour effectuer l'action donnée. Les agents connaissent le protocole de négociations soit par un accord préalable soit il est explicitement indiqué dans le paramètre du protocole de message.

Exemple : L'agent *j* demande à l'agent *i* de soumettre sa proposition de vendre 50 boîtes de prunes.

```
(cfp
  :sender (agent-identifier :name j)
  :receiver (set (agent-identifier :name i))
  :content
    "((action (agent-identifier :name i)
      (sell plum 50))
      (any ?x (and (= (price plum) ?x) (< ?x 10))))\"
  :ontology fruit-market
  :language fipa-sl)
```

Nom : Not Understood (pas compris)

Description : L'expéditeur de l'acte *not-understood* a reçu un acte de communication qu'il n'a pas compris. Il peut exister plusieurs raisons à ça : l'agent n'a pas été désigné pour effectuer certains actes ou alors il attend un message différent.

Exemple : L'agent *i* n'a pas compris un message de l'acte de communication *requête-si* parce qu'il n'a pas reconnu l'ontologie.

```
(not-understood
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "((action (agent-identifier :name j)
      (query-if
        :sender (agent-identifier :name j)
        :receiver (set (agent-identifier :name i))
        :content
          \"<fipa-ccl content expression>\"
        :ontology www
        :language fipa-ccl))
      (unknown (ontology \"www\"))))\"
  :language fipa-sl)
```

Nom : Propagate (propager)

Description : Cet acte de communication est une composition de deux actions :

- L'agent expéditeur accomplit l'acte de communication embarqué dans le message directement sur le receveur du message.
- L'expéditeur veut que le destinataire identifie les agents désignés dans la description et qu'il leur envoie ensuite le message.

Exemple : L'agent *i* demande à l'agent *j* et aux agents correspondant à la description de fournir un service de vidéo à la demande pour la catégorie « SF ».

```
(propagate
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "((any ?x (registered
      (agent-description
        :name ?x
        :services (set
          (service-description
            :name agent-brokerage))))
      (action (agent-identifier :name i)
```

```

(proxy
:sender (agent-identifier :name i)
:receiver (set (agent-identifier :name j))
:content
  \"((all ?y (registered
    (agent-description
      :name ?y
      :services (set
        (service-description
          :name video-on-demand))))))
    (action (agent-identifier :name j)
      (request
        :sender (agent-identifier :name j)
        :content
          \"((action ?z5
            (send-program (category \"SF\"))))\"
        :ontology vod-server-ontology
        :protocol fipa-request ...))
    true)\")
:ontology brokerage-agent-ontology
:conversation-id vod-brokering-2
:protocol fipa-brokering ...)
(< (hop-count) 5))\"
:ontology brokerage-agent-ontology
...)
```

Nom : Propose (proposition)

Description : Cet acte de communication sert à faire des propositions ou à répondre à un processus de négociations en cours en proposant d'accomplir une action aux pré-conditions étant vraies.

Exemple : L'agent *j* propose à l'agent *i* de vendre une boîte de prunes pour 5\$.

```

(propose
:sender (agent-identifier :name j)
:receiver (set (agent-identifier :name i))
:content
  \"((action j (sell plum 50))
    (= (any ?x (and (= (price plum) ?x) (< ?x 10)))) 5)\"
:ontology fruit-market
:in-reply-to proposal2
:language fipa-sl)
```

Nom : Request (requête)

Description : L'expéditeur demande au destinataire d'exécuter une action. Ce protocole se décline en 2 autres protocoles de requêtes qui sont : *request-when* et *request-whenever*. Pour le premier, il attend que la proposition devienne vraie (une fois), et dans le deuxième, il exécutera la requête chaque fois que la proposition devient vraie (plusieurs fois).

Exemple : L'agent *i* demande à *j* d'ouvrir un fichier.

```
(request
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "open \"db.txt\" for input"
  :language vb)
```

Nom : Subscribe (souscription)

Description : Cet acte permet d'envoyer par référence un objet auquel on souhaite se souscrire et d'être notifié à chaque changement de la part de l'objet.

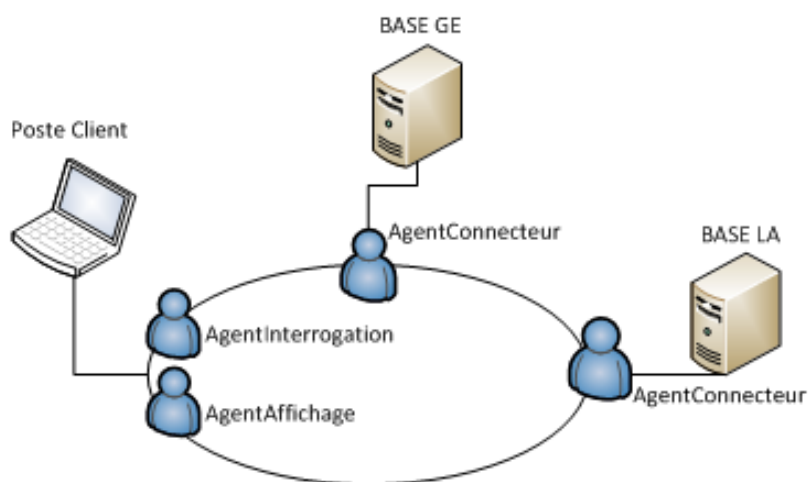
Exemple : L'agent *i* souhaite être mis à jour sur le taux de change FFR - \$ et fait une demande de souscription auprès de *j*.

```
(subscribe
  :sender (agent-identifier :name i)
  :receiver (set (agent-identifier :name j))
  :content
    "((iota ?x (= ?x (xch-rate FFR USD))))")
```

5.4 Ontologie du BOA

5.4.1 Mise en place et fonctionnement

Figure 12
Bus Orienté Agent



Sur le schéma nous pouvons observer les éléments suivants : un poste client, deux bases de données (ici celle de Genève et Lausanne), trois types d'agent (interrogation, connecteur, affichage) ainsi qu'un bus de communication (plate-forme JADE).

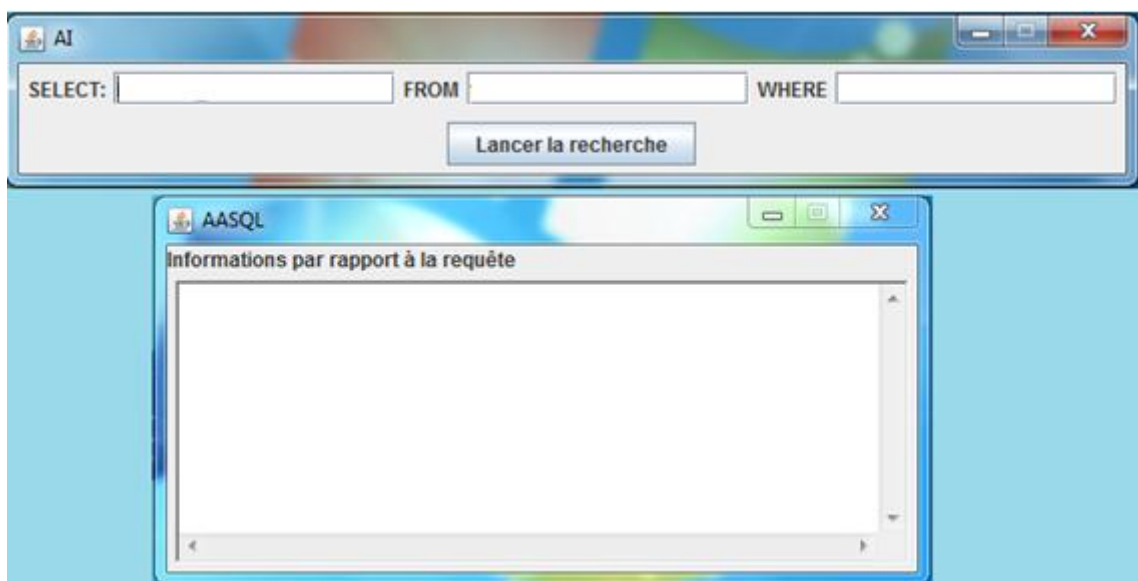
Parlons maintenant de la partie concernant les agents. Dans un premier temps, il a été défini trois types d'agents ; les agents dit *AgentInterrogation*, *AgentConnecteur* et *AgentAffichage*. L'*AgentInterrogation* est en charge de récupérer les informations saisies sur le poste client et de transmettre la requête sémantique à l'*AgentConnecteur*. De son côté, ce dernier sera chargé de contenir un module de traduction SPARQL/SQL et d'envoyer la requête dans la base de donnée. Cet agent se trouve à deux endroits sur le schéma car il faut autant d'*AgentConnecteur* que de base. Après avoir reçu une réponse, il transmettra le résultat de la requête à l'*AgentAffichage* qui sera responsable, comme son nom l'indique, d'afficher.

La partie traitant de l'interface et du module de traduction sont développés par deux autres étudiants étant également dans le projet.

5.4.2 Prototype simulant un BOA

Dans le but de me rapprocher du système à mettre en place, j'ai développé un prototype mettant en œuvre ce qui a été expliqué dans le précédent chapitre. Afin de réaliser cette première version, je vais utiliser une interface simple qui me permettra de saisir les informations que je souhaite récupérer (cf. figure ci-dessous et représentant l'*AgentInterrogation* (AI)) et une interface d'affichage des résultats (simulée par l'*AgentAffichage* (AASQL)).

Figure 13
Interfaces du prototype



Le rôle de chaque agent :

AgentInterrogation – A chaque clique sur le bouton « Lancer la recherche », récupérer les données des champs, les structurer sous forme de requête SQL et transmettre la requête à l'*AgentConnecteur* ;

AgentConnecteur – Ouvrir une connexion avec la base de données, puis exécuter la requête reçue dans la base de données (base SQL avec Oracle). Par la suite, cet agent enverra la réponse reçue à l'*AgentAffichage*.

AgentAffichage – Affiche dans un panel les résultats de la requête.

Intéressons nous dès à présent à la partie développement. Afin de comprendre comment se construit chaque agent et les liens entre eux, je vais expliquer le code java qu'ils emploient.

Premièrement, parlons des classes¹³ *AgentInterrogationGUI* et *AgentInterrogation*. Elles représentent l'agent et l'interface qu'il utilise.

Dans la classe *AgentInterrogationGUI* nous avons un constructeur (lieu où on définit la mise en place des éléments) et deux méthodes : *showGUI()* qui démarre l'affichage de l'interface et *validateForm()* qui s'assure que l'on ne lance pas la recherche avec des champs considérés comme vide (ne contenant aucun caractère) dans le formulaire. En étudiant la classe *AgentInterrogation* on voit qu'elle contient trois méthodes : *setup()* qui se lance à la création de l'agent, *takeDown()* s'exécutant lorsque l'agent se termine et *updateString()* s'occupant dans un premier temps de mettre à jour la requête selon les champs du formulaire et par la suite de lancer le comportement de l'agent *SendRequest()* qui enverra la requête.

Deuxièmement, voyons la classe *AgentConnecteurSQL*. Elle simule l'agent qui s'occupe de la connexion et du traitement de la requête dans la base. Cette classe contient la méthode *setup()* où l'on a l'ouverture de la connexion à la base de données, l'inscription de son service dans le DF et l'ajout de son comportement. Concernant son comportement *ExecuteQuery* (exécuter la requête), il utilise le contenu du message afin de le passer en paramètre dans la méthode *cnx.select()* qui se trouve dans le package « base » implémentée dans la classe *ConnexionBase.java*.

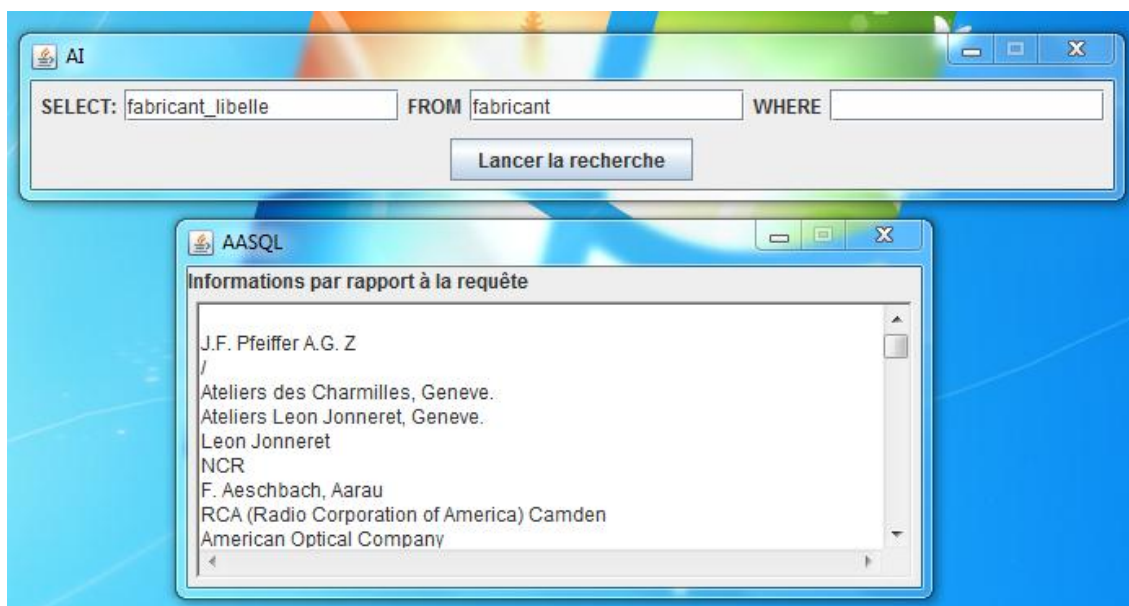
Troisièmement, étudions les classes *AgentAffichageSQL* et *AgentAffichageSQLGUI* qui tout comme les deux premières classes expliquées au début représentent l'agent et l'interface qu'il utilise.

Dans la première, nous avons la méthode *setup()* ayant un code similaire aux autres agents et implémentant le comportement *ShowResults()* (afficher les résultats) qui fera appel à la classe *AgentAffichageSQLGUI* pour exécuter le travail d'affichage. Dans cette dernière, nous trouvons la mise en place du layout (mise en page) du composant graphique et la méthode *setResults()* qui a comme responsabilité afficher le(s) résultat(s) dans le panel.

¹³ Le code complet de ces classes se trouve en annexe

Dernièrement, étudions la classe *ConnexionBase()* se trouvant le package « base ». Elle possède un constructeur et deux méthodes. Dans le constructeur se trouve les éléments permettant d'établir le driver et la connexion à la base. La première méthode, *getConnection()* nous permet de retourner une nouvelle connexion à la base. La deuxième méthode, *select(String rqt)*, nous permet d'exécuter la requête reçu en paramètre dans la base. Elle retournera une réponse sous forme de « string » afin de pouvoir l'afficher dans la zone prévu du panel.

Figure 14
Résultat final d'une requête

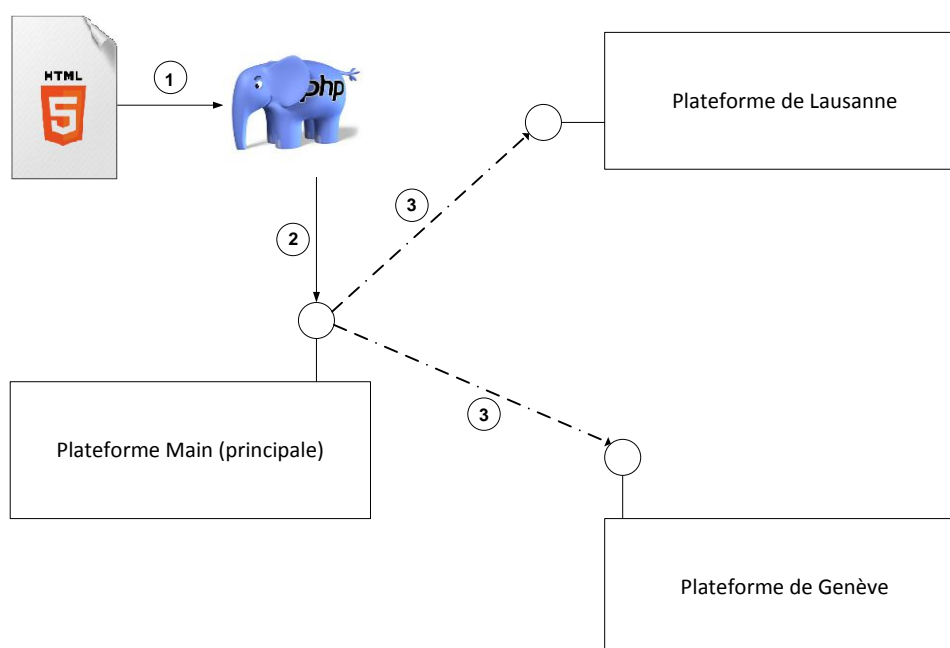


5.5 Normalisation du projet

La partie normalisation du projet consiste à mettre en place un plan permettant de rassembler les différentes parties du projet.

Pour cette dernière étape du projet, je vais travailler en collaboration avec M. David Hassine. Il m'apportera l'expérience informatique que je ne possède pas encore et de mon côté, je l'introduirais à la plateforme JADE et aux agents qui y vivent.

Figure 15
Schéma de normalisation



Le schéma que vous pouvez observer ci-dessus, est l'architecture souhaitée pour arriver à joindre les différentes parties du projet (interface et connecteurs). Dans le point 1, nous transmettons les données fournies par l'utilisateur d'une page HTML5 à un serveur PHP. Sur la « Plateforme Main », nous avons un agent qui est à l'écoute et qui attend les informations depuis le serveur PHP, point 2. Au point 3, l'agent de la plateforme principale transmet les données aux agents se trouvant sur les différentes plateformes éloignées (ici à Lausanne et à Genève). C'est l'agent se trouvant sur ces différentes plateformes qui contient le connecteur permettant la traduction de la requête afin de pouvoir interroger la base de données en SQL. Ces agents feront ensuite le chemin inverse pour fournir une réponse à l'utilisateur via l'interface dynamique en HTML5.

Pour la partie du projet me concernant, voici les éléments à mettre en place :

- 3 Plateformes distantes
- 1 agent connecteur par base de données
- 1 agent de recherche pour la détection du musée
- 2 agents de liaison avec le serveur PHP

Ensuite pour chaque plateforme nous avons les éléments suivants :

Platform-Principale

Dans le "Main container", on a les agents AMS, DF et RMA (nécessaire à la gestion de la plateforme)

Dans le "Container Agents", on y trouvera 1) l'agent du côté interface en entrée (contient la requête à exécuter), 2) l'agent côté interface en sortie (contient la réponse à afficher) et 3) l'agent s'occupant de la recherche des agents connecteurs sur les plateformes distantes.

Platform- Lausanne

Dans le "Main container", on a les agents AMS, DF et RMA (nécessaire à la gestion de la plateforme)

Dans le "Container Agents", on y trouve 1) l'agent contenant le « connecteur » et qui exécutera la requête dans la base de données pour ensuite retourner la réponse obtenue.

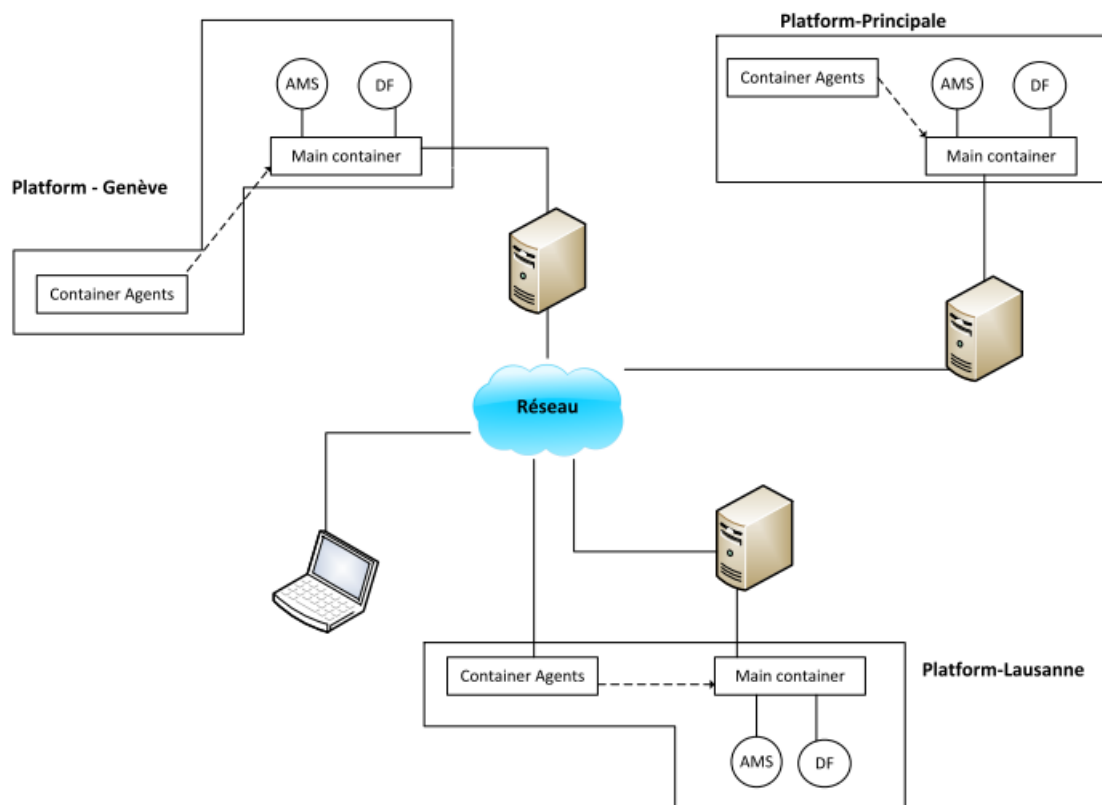
Platform-Genève

Dans le "Main container", on a les agents AMS, DF et RMA (nécessaire à la gestion de la plateforme)

Dans le "Container Agents", on y trouve 1) l'agent contenant le « connecteur » et qui exécutera la requête dans la base de données pour ensuite retourner la réponse obtenue.

Dans le schéma illustrant les propos ci-dessus, il faut savoir que les plateformes distantes (Genève et Lausanne) sont ajoutées à la plateforme principale et que dans chaque plateforme, les containers avec les agents sont enregistrés auprès du container principal contenant les agents de gestion de la plateforme.

Figure 16
Schéma d'architecture système multi-agents



Grâce à cette architecture, avec un ordinateur on va pouvoir se connecter à une page HTML5, qui elle est en liaison avec notre plateforme principale (dans le container agent). L'agent en entrée de l'interface va s'occuper de recevoir la requête de l'utilisateur et demandera à l'agent de recherche de la transmettre aux différents agents connecteurs. Une fois la requête exécutée, la réponse sera envoyé à l'agent en sortie de l'interface qui fournira les réponses à afficher (selon format défini par l'algorithme du connecteur). Il faut noter que l'agent connecteur sera en charge dans un premier temps d'établir une connexion avec sa base de données, dans un deuxième temps d'exécuter la requête et pour finir retourner la réponse.

5.6 Intégration

Dans ce dernier chapitre, nous allons aborder la mise en place de chaque partie du projet et comment les mettre en relation. On va parler de la mise en place de l'architecture, des agents créés, des différents problèmes lors de la mise en place et les solutions envisagées pour les résoudre. Les configurations expliquées dans ce chapitre concernant l'architecture, ont été testées en réseaux local avec 3 ordinateurs.

5.6.1 Interface

L'interface a pour but de se composer dynamiquement selon les données reçues ou l'utilisateur qui l'emploie. C'est lui qui fournira les données nécessaires pour créer la requête permettant d'interroger la base de données. Afin d'interagir avec les agents, il faudra mettre en place un serveur PHP qui gèrera les interactions utilisateurs sur notre interface.

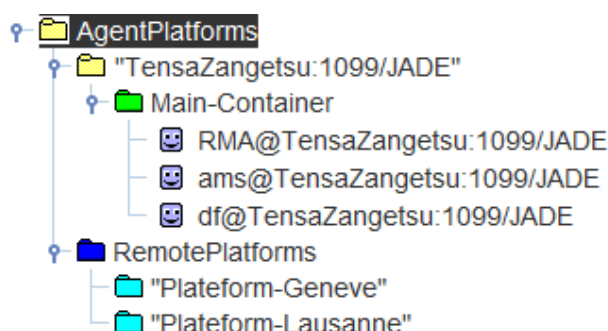
5.6.2 Connecteur

Ce qu'on appelle connecteur, est un module contenant un algorithme de traduction SPARQL/SQL en entrée et pouvant produire un fichier XML en sortie afin que l'interface se compose dynamiquement selon les données reçues. Ce module sera intégré dans un agent.

5.6.3 Architecture multi-agents

L'architecture mise en place est celle expliquée au chapitre de la normalisation. Pour rappel, nous avons 3 plateformes représentant le point de départ de la communication et les points d'arrivées des messages. Lorsque nous sommes sur notre plateforme principale et que l'on y ajoute nos plateformes distantes, nous obtenons l'arborescence illustrée sur la figure ci-dessous.

Figure 17
Arborescence des plateformes



Afin de démarrer les différentes plateformes, nous devons lancer 3 invites de commandes et y saisir respectivement les instructions suivantes :

- *java jade.Boot -gui -platform-id Plateform-Principale*
- *java jade.Boot -gui -port 1098 -host localhost -platform-id Plateform-Lausanne*
- *java jade.Boot -gui -port 1097 -host localhost -platform-id Plateform-Geneve*

Ensuite pour ajouter une plateforme, il faut aller sur la plateforme principale dans le menu « *Remote Platforms* », puis choisir « *Add Platform via AMS AID* ». Une fenêtre va alors apparaître et il faudra remplir le champ « *NAME* » avec le nom de l'AMS de la plateforme que l'on désire ajouter, ensuite il faut saisir l'adresse de la plateforme distante dans notre exemple, <http://TensaZangetsu:XXXXX/acc>.

L'étape suivante est l'ajout de nouveaux containers. Elle se fait dans une nouvelle invite de commandes, avec l'instruction :

- *java jade.Boot -container*
- *java jade.Boot -container -host localhost -port 1098*
- *java jade.Boot -container -host localhost -port 1097*

Nous avons maintenant notre architecture en place et prête à recevoir les agents.

5.6.4 Agents

Dans un premier temps, voici la liste des agents nécessaires à notre projet :

- AgentRecherche
- AgentConnecteur
- AgentInterfaceIn
- AgentInterfaceOut

Dans un deuxième temps regardons plus précisément les responsabilités de chaque agent.

L'**AgentRecherche** est responsable de trouver les différents connecteurs sur les plateformes distantes. Afin de trouver les agents connecteurs, il faut qu'il fasse une recherche dans le DF pour trouver le service correspondant. Il remplit un tableau d'agent avec toutes les réponses qu'il a obtenu. Il ajoute ensuite chaque agent de ce tableau comme destinataire du message. Il définit le contenu du message avec la requête reçue de la part de l'AgentInterfaceIn et finit son comportement par l'envoi du message.

L'**AgentConnecteur** est garant de la partie en relation avec la base de données. Son comportement va suivre les étapes suivantes ; établir une connexion avec la base de données, traiter la requête avec le « module connecteur », exécuter la requête, introduire la réponse dans « module connecteur » et retourner une réponse à l'**AgentInterfaceOut**. On va donc utiliser un comportement générique qui peut se composer de plusieurs étapes (cf. 5.2.2).

L'**AgentInterfaceIn** est celui qui fait le lien entre l'interface et la plateforme multi-agents. Son rôle va être de récupérer les informations saisies par l'utilisateur dans l'interface et de forger une requête correspondant aux paramètres demandés par le « module connecteur ». Une fois qu'il a terminé cette tâche, il passe le relais à l'**AgentRecherche**.

L'**AgentInterfaceOut** est l'agent responsable de transmettre la réponse à l'interface afin de pouvoir afficher les informations obtenues.

5.6.5 Problèmes et Solutions

5.6.5.1 Chemin d'accès aux fichiers jar

Lors de la mise en place de la plateforme JADE sur le serveur à Lausanne, nous avons eu un problème de *classpath* (variable système ayant le chemin direct sur les différents fichiers jar de jade).

Pour résoudre ce problème, j'ai redéployé sur une machine propre (n'ayant jamais installé JADE) l'arborescence nécessaire au bon fonctionnement de la plateforme. J'ai donc pu par la suite tester la mise en place et l'exécution de mon projet de simulation sans rencontrer à nouveau ce problème. J'ai donc transmis à nos collaborateurs à Lausanne la marche à suivre pour déployer la plateforme selon la nouvelle arborescence.

5.6.5.2 Déployer un projet Eclipse à la ligne de commande

Afin de tester mon environnement, j'ai voulu mettre en place mon prototype de simulation parlé au chapitre 5.3.2. Dans un premier temps, j'ai copié le dossier du projet Eclipse directement dans le dossier des exemples proposés par JADE. Lors du lancement de la commande de compilation (`javac -classpath lib\jade.jar -d classes src\examples\simulation*.java`), je me suis retrouvé avec des erreurs de package et d'accès aux méthodes dans les classes.

J'ai résolu ce problème en m'aidant des projets exemples de JADE et j'ai remarqué que le nom des packages ne correspondaient pas à ceux que j'avais sous Eclipse. Alors j'ai ouvert un éditeur de texte (EditPlus 3) et j'ai changé les noms des package afin de les faire correspondre avec le dossier dans lequel j'ai mis mon projet. J'ai ensuite lancé la commande de compilation et je n'ai plus eu de problèmes.

5.6.5.3 Drivers OJDBC

Dans mon prototype, je mets en œuvre une connexion à une base de données sous ORACLE. J'ai placé le driver dans le dossier des jars de la plateforme et j'ai essayé de lancé mon agent connecteur mais lorsque qu'il arrive sur la plateforme, il me dit que le driver est introuvable et par conséquent la base non plus.

Après avoir fait des recherches sur des forums professionnels, j'ai pu remarquer les choses suivantes ; Je n'avais pas ajouté à ma variable *classpath* le chemin jusqu'au fichier jar de mon driver et le « listener » (écoute les événements) de ma base de donnée ORACLE était arrêter. J'ai donc changé ma variable en lui rajoutant le chemin de mon fichier et démarré le « listener » ORACLE. J'ai alors pu trouver mon driver et me connecter à la base de données.

5.6.5.4 Droit de connexion sur le serveur à Lausanne

Dans le but d'implanter la plateforme multi-agents à Lausanne, j'ai rencontré M. Jean-Claude Genoud (chef de projets documentaires et patrimoniaux à la Ville de Lausanne) et M. Damiano Cereghetti (chef de projet informatique). Lors de la mise en place de la plateforme nous avons constaté que la plateforme n'était pas joignable dû à un timeout. M. Cereghetti nous a alors expliqué que très probablement, le proxy mis en place sur le serveur n'autorisait pas la plateforme à recevoir des messages externes.

Dans le but de résoudre ce problème, il faudra demander l'ouverture du port sur lequel la plateforme est déployée afin de pouvoir communiquer avec celle-ci. Nos interlocuteurs nous ont précisé que cela ne dépend pas d'eux et qu'ils ne peuvent pas nous donner un ordre de temps concernant la résolution de ce problème.

5.6.5.5 Conclusion

N'ayant pas à l'heure actuelle les éléments nécessaires pour finir l'intégration avec les autres parties du projet, je n'ai pas rencontré d'autres problèmes. Pour le projet des interfaces et pour celui des connecteurs, j'ai moi-même fait en sorte de simuler les différents messages possibles sur la plateforme afin de tester mon architecture.

Sur un plan personnel, les différents problèmes rencontrés m'ont permis de mieux comprendre le fonctionnement de la plateforme multi-agents JADE.

Conclusion

La notion de sémantique est de plus en plus présente dans les systèmes d'informations. Avec cette notion, ajouter une couche qui nous permet d'avoir un niveau d'abstraction plus élevé dans notre environnement et de permettre à des processus automatiques un traitement plus fin et plus habile des données. Grâce à elle, on va pouvoir mettre en place de nouvelles technologies permettant de communiquer et agir par le « sens » des données et non pas simplement par leur syntaxe. Afin de transmettre des messages entre ces couches sémantiques, la FIPA a mis en place des standards de communication. Ce mémoire orienté architecture multi-agents, a démontré que l'implantation d'une plateforme (compatible FIPA) permet d'envoyer et de recevoir des messages entre différentes plateformes (locales ou distantes).

La technologie agent nous donne la possibilité de créer des systèmes autonomes et pouvant réagir selon le contexte dans lequel se trouve l'environnement. Le fait de pouvoir donner des comportements simples ou complexes à un agent nous permet d'intégrer des comportements issus du domaine de l'« intelligence artificielle ».

D'un point de vue personnel, ce travail m'a permis de développer les notions acquises au sein de l'Haute Ecole de Gestion. Bien que la technologie multi-agents et les outils pour sa mise en place m'étaient inconnus, j'ai accepté d'intégrer ce projet car il m'a semblé important que j'en apprenne plus sur la notion de sémantique. Je pense que dans un avenir proche, cette notion sera importante et de plus en plus utilisée dans les systèmes d'informations.

Concernant le planning mis en place, il a été respecté dans son ensemble car bien que des fois dans un ordre différent, j'ai pu parcourir et appliquer tous les objectifs fixés lors de l'établissement du planning du projet. Je regrette que l'objectif « Intégration » n'ait pas pu être réalisé dans son ensemble. Pour finir, je tiens à soulever l'aspect humain de ce projet. Les différentes interactions que j'ai eu tout au long du projet m'ont fait grandir culturellement et scientifiquement. Les explications données par M. Johann Sievering m'ont été précieuses dans la réalisation de mon mandat. Les interventions en fin de projet de M. Hassine m'ont permis d'appréhender les problèmes sous d'autres angles.

Lexique

FIPA :	Fondation pour l'Intelligence Physique des Agents
ACL	Langage de Communication Agent
JADE	Java Agent DEvelopment framework est une plate-forme multi-agent créer par le laboratoire TILAB
CAL	Libraires d'Actes de Communication
IDE	Environnement Intégrer de Développement
BOA	Bus Orienté Agent
API	Association pour le Patrimoine Industriel
AMS	Système Manager des Agents
DF	Pages Jaunes (Directory Facilitator)
MTS	Système de Transport de Message

Bibliographie

Ouvrages consultés

BELLIFEMINE, Fabio, CAIRE, Giovanni, et GREENWOOD Dominic, *developing multi-agents systems with JADE*, 2007, John Wiley & Sons, Ltd, 303 p, ISBN 978-0-470-05747-6.

DE MORAES BATISTA, André Filipe, *Desenvolvendo sistemas multiagentes na plataforma JADE*, 2008, Santos André, 79 p., Manuel complémentaire au projet de recherche : *Sistemas Multiagentes na Construção de um Middleware para Suporte a Ambientes Computacionais*.

SIEVERING, Johann, *Semantic approach in the information systems*, 2003, Genève.

Sites internet consultés :

ASSOCIATION POUR LE PATRIMOINE INDUSTRIEL, *Site de l'association pour le patrimoine industriel* [en ligne]. <http://www.patrimoineindustriel.ch/> (consulté le 30 octobre 2012)

DEVELOPPEZ.COM. *Créez votre premier agent avec JADE et ECLIPSE* [en ligne]. <http://djug.developpez.com/java/jade/creation-agent/> (consulté le 24 octobre 2012)

DEVELOP123, *Netbeans IDE preparation for first JADE project* [en ligne]. <http://agents.develop123.com/index.php/getting-started-with-jade/48-netbeans-ide-preparation-for-first-jade-project> (consulté le 25 octobre 2012)

EJADE. *Eclipse IDE Plug-in for Java Agent Development Environment* [en ligne]. <http://selab.fbk.eu/dnguyen/ejade/index.html> (consulté le 25 octobre 2012)

FIPA, *FIPA Agent Management Specification* [en ligne]. <http://www.fipa.org/specs/fipa00023/XC00023H.html> (consulté le 11 novembre 2012)

JADE, *JADE Administration Tutorial* [en ligne]. <http://jade.tilab.com/doc/tutorials/JADEAdmin/index.html> (consulté du 30 octobre 2012 au 15 janvier 2013)

Annexe 1

Planning du travail de Bachelor

Semaine 1		24.09.2012	25.09.2012	26.09.2012	27.09.2012	28.09.2012	29.09.2012	30.09.2012
Objectifs :		Maîtrise de la plateforme de développement JADE		Acquisition de documents concernant les plates-formes JADE (Eclipse) multi-agent	Lecture des documents + installation de			
		Rendez-vous avec Johann Sievering pour démarrer le projet						
		Rédaction d'un planning de projet						
		Rédaction du travail de Bachelor - Partie obligatoire						
Semaine 2		01.10.2012	02.10.2012	03.10.2012	04.10.2012	05.10.2012	06.10.2012	07.10.2012
Objectifs :		Maîtrise de la plateforme de développement JADE						
Semaine 3		08.10.2012	09.10.2012	10.10.2012	11.10.2012	12.10.2012	13.10.2012	14.10.2012
Objectifs :		Maîtriser le protocole de communication FIPA						
Semaine 4		15.10.2012	16.10.2012	17.10.2012	18.10.2012	19.10.2012	20.10.2012	21.10.2012
Objectifs :		Programmation de l'Agent Informatique						
Semaine 6		29.10.2012	30.10.2012	31.10.2012	01.11.2012	02.11.2012	03.11.2012	04.11.2012
Objectifs :		Compréhension du protocole "Musée" (communication entre les agents sur un même référentiel)						
		S'approprier le langage ACL						

Semaine 7

05.11.2012	06.11.2012	07.11.2012	08.11.2012	09.11.2012	10.11.2012	11.11.2012
Objectifs : Compréhension du protocole "Musée" (communication entre les agents sur un même référentiel) S'approprier le langage ACL						

Semaine 8

12.11.2012	13.11.2012	14.11.2012	15.11.2012	16.11.2012	17.11.2012	18.11.2012
Objectifs : Comprendre l'ontologie du Bus						

Semaine 9

19.11.2012	20.11.2012	21.11.2012	22.11.2012	23.11.2012	24.11.2012	25.11.2012
Objectifs : Comprendre l'ontologie du Bus						

Semaine 10

26.11.2012	27.11.2012	28.11.2012	29.11.2012	30.11.2012	01.12.2012	02.12.2012
Objectifs : Requêtes sémantiques --> logiques descriptives						

Semaine 11

03.12.2012	04.12.2012	05.12.2012	06.12.2012	07.12.2012	08.12.2012	09.12.2012
Objectifs : Requêtes sémantiques --> logiques descriptives						

Semaine 12

10.12.2012	11.12.2012	12.12.2012	13.12.2012	14.12.2012	15.12.2012	16.12.2012
Objectifs : Normalisation avec les autres parties du projet						

Semaine 13

17.12.2012	18.12.2012	19.12.2012	20.12.2012	21.12.2012	22.12.2012	23.12.2012
Objectifs : Intégration						

Semaine 14

07.01.2013	08.01.2013	09.01.2013	10.01.2013	11.01.2013	12.01.2013	13.01.2013
Objectifs : Documentation - mis à jour du mémoire						

Semaine 15

14.01.2013	15.01.2013	16.01.2013	17.01.2013	18.01.2013	19.01.2013	20.01.2013
Objectifs : Examen oral d'algo - prog						

Semaine 16

21.01.2013	22.01.2013	23.01.2013	24.01.2013	25.01.2013	26.01.2013	27.01.2013
Objectifs : Rendu du travail de Bachelor Préparation de la présentation de soutenance						

Annexe 2

Prototype SMA – Agents

AgentAffichageSQL.java

```
package agents;
import jade.core.AID;

/**
 * @author Miguel Tavares Dos Santos - HEG Genève
 * @version 1.0
 */
public class AgentAffichageSQL extends Agent {
    private AgentAffichageSQLGUI myGui;

    protected void setup() {
        // Create and show the GUI
        myGui = new AgentAffichageSQLGUI(this);
        myGui.showGui();

        // Register the interrogation service in DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType("bdd-affichage");
        sd.setName("JADE-bdd-simulation");
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        addBehaviour(new ShowResults());
    } // setup

    // Put agent clean-up operations here
    protected void takeDown() {
        // Deregister from the yellow pages
        try {
            DFService.deregister(this);
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        // Close the GUI
        myGui.dispose();
        // Printout a dismissal message
        System.out.println("Interrogation-Agent "+getAID().getName()+" terminating.");
    }

    private class ShowResults extends CyclicBehaviour {
        @Override
        public void action() {
            ACLMessage msg = myAgent.receive();
            if (msg != null) {
                System.out.println("J'ai reçu le message " + msg.getContent().toString());
                myGui.setResults(msg.getContent().toString());
            } else {
                block();
            }
        }
    } // EnvoyerRequete
} // AgentAffichageSQL
```

```

package agents;
import java.awt.BorderLayout;

public class AgentAffichageSQLGUI extends JFrame {
    private AgentAffichageSQL myAgent;
    TextArea can;

    AgentAffichageSQLGUI(AgentAffichageSQL a){
        super(a.getLocalName());
        myAgent = a;

        JPanel panelHaut = new JPanel();
        panelHaut.setLayout(new BorderLayout());
        panelHaut.add(new JLabel("Informations par rapport à la requête"),BorderLayout.NORTH);

        JPanel panelBas = new JPanel();
        can = new TextArea();
        can.setSize(380, 280);
        can.setBackground(Color.WHITE);
        panelBas.add(can, BorderLayout.SOUTH);

        getContentPane().add(panelHaut, BorderLayout.NORTH);
        getContentPane().add(panelBas, BorderLayout.SOUTH);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                myAgent.doDelete();
            }
        });
        setResizable(false);
    } // Constructeur AgentAffichageSQLGUI

    public void showGui() {
        pack();
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
        int centerX = (int)screenSize.getWidth() / 2;
        int centerY = (int)screenSize.getHeight() / 2;
        setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
        super.setVisible(true);
    } // showGUI

    public void setResults(String str){
        can.setText(str);
    } // setResults
} // AgentAffichageSQLGUI

```

```

package agents;

import java.sql.ResultSet;

/**
 * @author Miguel Tavares Dos Santos - HEG Genève
 * @version 1.0
 */
public class AgentConnecteurSQL extends Agent {
    ResultSet rs = null;
    ConnexionBase cnx = null;

    protected void setup(){
        cnx = ConnexionBase.getConnection();
        // Register the interrogation service in DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType("bdd-connect");
        sd.setName("JADE-bdd-simulation");
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        addBehaviour(new ExecuteQuery());
    } // setup

    private class ExecuteQuery extends CyclicBehaviour {
        public void action() {
            ACLMessage msgReponse = new ACLMessage(ACLMessage.REQUEST);
            ACLMessage msg = receive();
            if (msg!=null) {
                System.out.println("J'ai reçu le message " + msg.getContent().toString());
                msgReponse.setContent(cnx.select(msg.getContent()));
                msgReponse.addReceiver(new AID("AASQL", AID.ISLOCALNAME));
                send(msgReponse);
                //System.out.println("J'ai envoyé le message " +
                msgReponse.getContent().toString());
            }else {
                System.out.println("je bloque !");
                block();
            }
        } // action
    } //EnvoyerRequete
} // AgentConnecteurSQL

```

```

package agents;
import jade.core.AID;

/**
 * @author Miguel Tavares Dos Santos - HEG Genève
 * @version 1.0
 */
public class AgentInterrogation extends Agent {
    private AgentInterrogationGUI myGui;
    private String strRequest;

    protected void setup() {
        // Create and show the GUI
        myGui = new AgentInterrogationGUI(this);
        myGui.showGui();
        // Register the interrogation service in DF
        DFAgentDescription dfd = new DFAgentDescription();
        dfd.setName(getAID());
        ServiceDescription sd = new ServiceDescription();
        sd.setType("bdd-request");
        sd.setName("JADE-bdd-simulation");
        dfd.addServices(sd);
        try {
            DFService.register(this, dfd);
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        //addBehaviour(new EnvoyerRequete());
    } // setup

    protected void takeDown() {
        // Deregister from the yellow pages
        try {
            DFService.deregister(this);
        }
        catch (FIPAException fe) {
            fe.printStackTrace();
        }
        // Close the GUI
        myGui.dispose();
        // Printout a dismissal message
        System.out.println("Interrogation-Agent "+getAID().getName()+" terminating.");
    }

    /**
     * This is invoked by the GUI when the user ask another query
     */
    public void updateString(final String str) {
        addBehaviour(new OneShotBehaviour() {
            public void action() {
                strRequest = str;
                addBehaviour(new SendRequest());
            }
        });
    } // updateString

    private class SendRequest extends OneShotBehaviour {
        @Override
        public void action() {
            if (strRequest.equals("")) {
                block();
            }
            else {
                ACLMessage msg = new ACLMessage(ACLMessage.REQUEST);
                msg.setContent(strRequest);
                msg.addReceiver(new AID("ACSQL", AID.ISLOCALNAME));
                send(msg);
                System.out.println("J'ai envoyé le message " + msg.getContent().toString());
            }
        }
    } //SendRequest
} // AgentInterrogation

```

```

package agents;

import java.awt.*;

public class AgentInterrogationGUI extends JFrame {

    private AgentInterrogation myAgent;
    private JTextField selectField, fromField, whereField;

    AgentInterrogationGUI(AgentInterrogation a) {
        super(a.getLocalName());
        myAgent = a;
        JPanel p = new JPanel();
        p.setLayout(new FlowLayout());
        p.add(new JLabel("SELECT:"));
        selectField = new JTextField(15);
        p.add(selectField);
        selectField.setText("fabricant_libelle");
        p.add(new JLabel("FROM"));
        fromField = new JTextField(15);
        p.add(fromField);
        fromField.setText("fabricant");
        p.add(new JLabel("WHERE"));
        whereField = new JTextField(15);
        p.add(whereField);
        getContentPane().add(p, BorderLayout.CENTER);

        JButton addButton = new JButton("Lancer la recherche");
        addButton.addActionListener( new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                try {
                    if (validateForm()) {
                        if (whereField.getText().trim().isEmpty()) {
                            String str = "SELECT " + selectField.getText().trim() + " FROM " +
fromField.getText().trim();
                            myAgent.updateString(str);
                        }else{
                            String str = "SELECT " + selectField.getText().trim() + " FROM " +
fromField.getText().trim() + " WHERE " + whereField.getText().trim();
                            myAgent.updateString(str);
                        }
                    }else{
                        JFrame popup = new JFrame();
                        JOptionPane.showMessageDialog(popup, "Les champs ne sont pas
correctement renseignés!!!", "Erreur de champs", JOptionPane.ERROR_MESSAGE);
                    }
                }
                catch (Exception e) {
                    JOptionPane.showMessageDialog(AgentInterrogationGUI.this, "Invalid values.
"+e.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
                }
            }
        });
        p = new JPanel();
        p.add(addButton);
        getContentPane().add(p, BorderLayout.SOUTH);

        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                myAgent.doDelete();
            }
        });

        setResizable(false);
    } //AgentInterrogationGUI
}

```

```

public void showGui() {
    pack();
    Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
    int centerX = (int)screenSize.getWidth() / 2;
    int centerY = (int)screenSize.getHeight() / 2;
    setLocation(centerX - getWidth() / 2, centerY - getHeight() / 2);
    super.setVisible(true);
} // showGui

private boolean validateForm(){
    if ((selectField.getText().trim().isEmpty()) ||
(fromField.getText().trim().isEmpty())) {
        return false;
    }else{
        return true;
    }
} // validateForm
} //AgentInterrogationGUI

```

Annexe 2

Prototype SMA – Connexion BDD

ConnexionBase.java

```
package base;
import java.sql.*;

/**
 * @author Miguel Tavares Dos Santos - HEG Genève
 * @version 1.0
 */
public class ConnexionBase {
    private static ConnexionBase myConnexion = null;
    Connection cnx = null;
    Statement stmt = null;
    ResultSet rs = null; ResultSet resultats ;

    private ConnexionBase(){
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            System.out.println("Le driver est établi");
        } catch (Exception e) {
            System.out.println("Erreur : driver introuvable");
        }
        try {
            String url = "jdbc:oracle:thin:@//localhost/xe"; String user = "system"; String
            passwd = "mds";
            cnx = DriverManager.getConnection(url,user,passwd);
            System.out.println("Connexion avec la base API établie");
        } catch (Exception e) {
            System.out.println("Erreur : base introuvable");
        }
    } // constructeur privé de notre ConnexionBase()

    public static ConnexionBase getConnection(){
        if (myConnexion == null) {
            return myConnexion = new ConnexionBase();
        } else {
            return myConnexion;
        }
    } // getConnexion()

    public String select(String rqt){
        String rep = "";
        try {
            Statement stmt = cnx.createStatement();
            rs = stmt.executeQuery(rqt);
            while (rs.next()) {
                rep = rep + "\n" + rs.getString(1);
            }
            stmt.close();
            return rep;
        } catch (Exception e) {
            System.out.println("Erreur : anomalie avec la requête");
        }
        return rep;
    }
} // ConnexionBase
```