Rapport final de projet

Rédaction de tutoriels expliquant le fonctionnement d'un système multi-agent

Réalisé par :
Vincent PRADEILLES et Gaëlle HISLER

Sous la direction de :
M. Fabien MICHEL

Année universitaire 2010-2011

Rapport final de projet

Rédaction de tutoriels expliquant le fonctionnement d'un système multi-agent

Réalisé par :
Vincent PRADEILLES et Gaëlle HISLER

Sous la direction de :
M. Fabien MICHEL

Année universitaire 2010-2011

Remerciements

Avant d'entamer ce rapport, nous souhaitons remercier tout d'abord notre tuteur de projet M. Fabien MICHEL, qui nous a encadrés tout au long de la réalisation de ce projet.

Nous tenons à remercier également nos professeurs de communication, Mme METZ et M. MAHIQUES de nous avoir accompagnés dans notre travail en mettant à notre disposition leurs expériences et leurs compétences principalement en ce qui concerne la rédaction de ce rapport.

Glossaire

Agent : entité autonome qui évolue au sein d'une société hiérarchisée, et est capable d'interagir avec son environnement et les autres membres de sa société.

Modèle organisationnel : structure permettant aux agents de communiquer entre eux. Un modèle organisationnel est établi sur 3 niveaux :

- Les communautés, qui chacun un certain nombre de groupe
- Les groupes qui regroupent les différents rôles
- Les rôles qui sont occupés par un ou plusieurs agents

Encapsuler : méthode de conception qui consiste à donner à un objet, et à lui seul, le droit d'accéder à ses données, autant en écriture qu'en lecture. Dans le cadre de la programmation multi-agent, l'encapsulation permet d'appliquer le principe fondamental qu'est l'interdiction d'accéder aux données d'un agent.

Noyau MadKit : ensemble de classes implémentant les primitives d'un système multi-agent (Par exemple : structure d'un agent, création d'un groupe, envoi d'un message).

Sommaire

1. Introduction

Actuellement en deuxième année de DUT en Informatique à l'Institut Universitaire des Technologies de Montpellier, nous devons réaliser un projet tuteuré pendant une durée de quinze semaines.

Ce projet vise à la rédaction de tutoriels expliquant le fonctionnement d'un système multi-agent.

Ce projet laisse une grande partie d'autonomie et de liberté quand aux choix des tutoriels, tant que ceux-ci mettent en avant les spécificités de la programmation multi-agents.

Au travers de ce projet nous verrons donc en premier lieu la définition exacte d'un système multi-agent, mais aussi une présentation de MadKit (outil contenant les librairies utilisées).

Nous montrerons également la démarche que nous avons suivie, c'est-à-dire tout d'abord la compréhension d'un code déjà existant, puis nos premières modifications, et ensuite nous illustrerons notre projet par la réalisation de quatre exercices différents visant chacun une spécificité de la programmation multi-agents.

2. Organisation du travail

Période	Intitulé
Du 19/10 au 2/11	Compréhension du fonctionnement de MadKit au travers de la démo Ping-pong
Du 2/11 au 16/11	Première modification du programme Ping-pong : modification des messages envoyés par les agents pour permettre la prise en compte des scores. Rédaction du rapport préliminaire.
Du 16/11 au 30/11	Apports de modifications supplémentaires au programme Ping-pong : création d'un mini-tournoi et d'une fenêtre d'affichage des scores.
Du 1/12 au 8/12	Choix des exemples qui permettront de mettre en avant les fonctionnalités essentielles de MadKit
Du 8/12 au 29/12	Ecriture du code correspondant aux exemples choisis. L'accent est mis sur la cohérence des différents codes entre eux ainsi que sur leur clarté.
Du 29/12 au 18/01	Rédaction des tutoriels présentant MadKit et montrant les fonctionnalités principales du système.
Du 18/01 au 29/01	Rédaction du rapport final.

3. MadKit

3.1. Introduction à la programmation multi-agent

La programmation multi-agents permet de concevoir des systèmes qui se composent de plusieurs agents. Un agent étant une entité qui possède plusieurs propriétés, entre autres : être partiellement - voire entièrement – autonome, communiquer avec d'autres agents, posséder des compétences et des objectifs (ou tendances) et posséder des ressources propres.

Un agent peut représenter un processus mais également tout autre type d'entité tel qu'un robot. Il agit dans un environnement composé de groupes d'agents au sein desquels chaque agent joue au moins un rôle prédéfini. Un SMA (Système Multi-Agent) est donc hiérarchisé et les rôles que jouent les agents sont représentables sur un organigramme. Un agent est autonome, c'est-à-dire qu'il est possible de lui demander de réaliser une action, par le biais de l'envoi d'un message, mais il est impossible de « rentrer » dans un agent pour effectuer une action à sa place ; on peut donc dire qu'un agent est encapsulé. Un groupe d'agents à un fonctionnement décentralisé, c'est-à-dire qu'il n'est pas possible pour un agent de diriger tous les autres (si cela était le cas, il ne s'agirait alors plus d'un SMA mais d'un système mono-agent). Un agent est également capable de percevoir son environnement local et d'interagir avec celui-ci. Cette interaction est à double sens puisque l'environnement peut également interagir avec les agents qu'il contient. Cependant un agent ne peut pas avoir une vue d'ensemble de son environnement.

Pour permettre le bon fonctionnement d'un SMA, chaque agent doit posséder un certain nombre de compétences. Premièrement les agents doivent être capables de prise de décisions et de planifications (aussi bien individuelles que collectives), ils doivent être dotés d'un modèle cognitif, c'est-à-dire d'une façon de penser. Un exemple de modèle cognitif est le modèle BDI qui permet à un agent de « penser » en s'appuyant sur des croyances (Beliefs), des objectifs (Desires) et des intentions (Intentions), ces dernières étant obtenues à partir des deux autres. Puisque les agents doivent être capables de communiquer, il est impératif qu'ils « parlent » tous le même langage.

Les SMA possèdent de nombreuses applications pratiques, ils permettent par exemple de modéliser un ensemble d'individu pour réaliser des simulations (évacuation d'un lieu public par exemple), ou bien pour réaliser des actions qui nécessitent l'exécution de nombreuses opérations en parallèle.

3.2. Présentation de MadKit

Pour réaliser des programmes basiques, mettant en avant les principales primitives d'un système multi-agents, nous avons utilisé la plateforme MadKit. Il s'agit d'un projet open source, disponible sur le site http://www.madkit.org/

MadKit est une plateforme multi-agent modulaire et scalable, écrite dans le langage Java. Elle permet la création de SMA en se basant sur le modèle relationnel Agent/Groupe/Rôle. MadKit tire profit de la Programmation Orientée Objet : les fonctionnalités de MadKit sont contenues dans le noyau MadKit*. Ce noyau est un ensemble de classes permettant à l'utilisateur de concevoir un SMA basique de façon simple, mais également, par le biais de l'héritage, de concevoir et d'ajouter de nouvelles fonctionnalités qui seront compatibles avec celles fournies de base.

Un des plus gros avantages de MadKit est que, parce qu'il définit une structure de base pour la représentation d'un agent, d'un groupe, d'un message, etc., il est relativement aisé de faire communiquer ensemble des agents conçus par des programmeurs différents, voire pour des projets différents.

Les classes principales du noyau MadKit sont :

La classe *AbstractAgent* est la super-classe de tout agent crée sous MadKit. Sous MadKit, un agent possède un cycle de vie composé de trois étapes : activation, vie et mort. La classe *AbstractAgent* contient trois méthodes permettant de gérer ce cycle de vie :

- La méthode *activate()* contient les instructions à effectuer lorsque l'agent est activé (via une autre méthode : *launchAgent()*). Cette méthode permet, par exemple d'initialiser les données que possède l'agent, de lui faire faire des requêtes d'admission dans un groupe, etc.)
- La méthode *live()* contient le bloc d'instruction correspondant à la vie d'un agent. Un exemple de contenu d'une méthode *live()* peut être une boucle infinie dans laquelle l'agent répète les instructions : attente d'un message, traitement du message et réponse au message.
- La méthode *end()* contient le bloc d'instructions à exécuter lors de l'arrêt de l'agent. Une méthode *end()* peut, par exemple, contenir les instructions nécessaires à la sauvegarde des données que possède l'agent.

La classe *AbstractAgent* permet également de gérer le modèle relationnel Agent/Groupe/Rôle. Pour cela, elle définit un ensemble de méthodes qui permettent à un agent de rejoindre un groupe, puis d'y demander un rôle particulier. On peut citer les méthodes :

- *createGroup(), createGroupIfAbsent()* permettant de créer un groupe.
- *requestRole()* permettant de demander un rôle au sein d'un groupe particulier, dont le nom est passé en paramètre.

Enfin, puisque les agents agissent au sein d'un modèle relationnel, il est nécessaire qu'ils puissent communiquer entre eux. Pour permettre cette communication, deux classes entrent en jeu : *AgentAdress* et *Message*.

La classe *AgentAdress* permet de représenter l'adresse d'un agent, c'est-à-dire l'endroit vers où doivent être expédiés les messages destinés à cet agent. Point important, un agent peut posséder plusieurs adresses. En effet, *l'AgentAdress* représente le triplet <communauté, groupe, rôle>. Comme un agent peut occuper différents rôles au sein d'un ou plusieurs groupes et communautés, il est nécessaire qu'il possède une adresse pour chacun des rôles qu'il tient.

Dans le but de simplifier la communication entre agents conçus par différents développeurs, la classe *Message<T>* définit la structure d'un message :

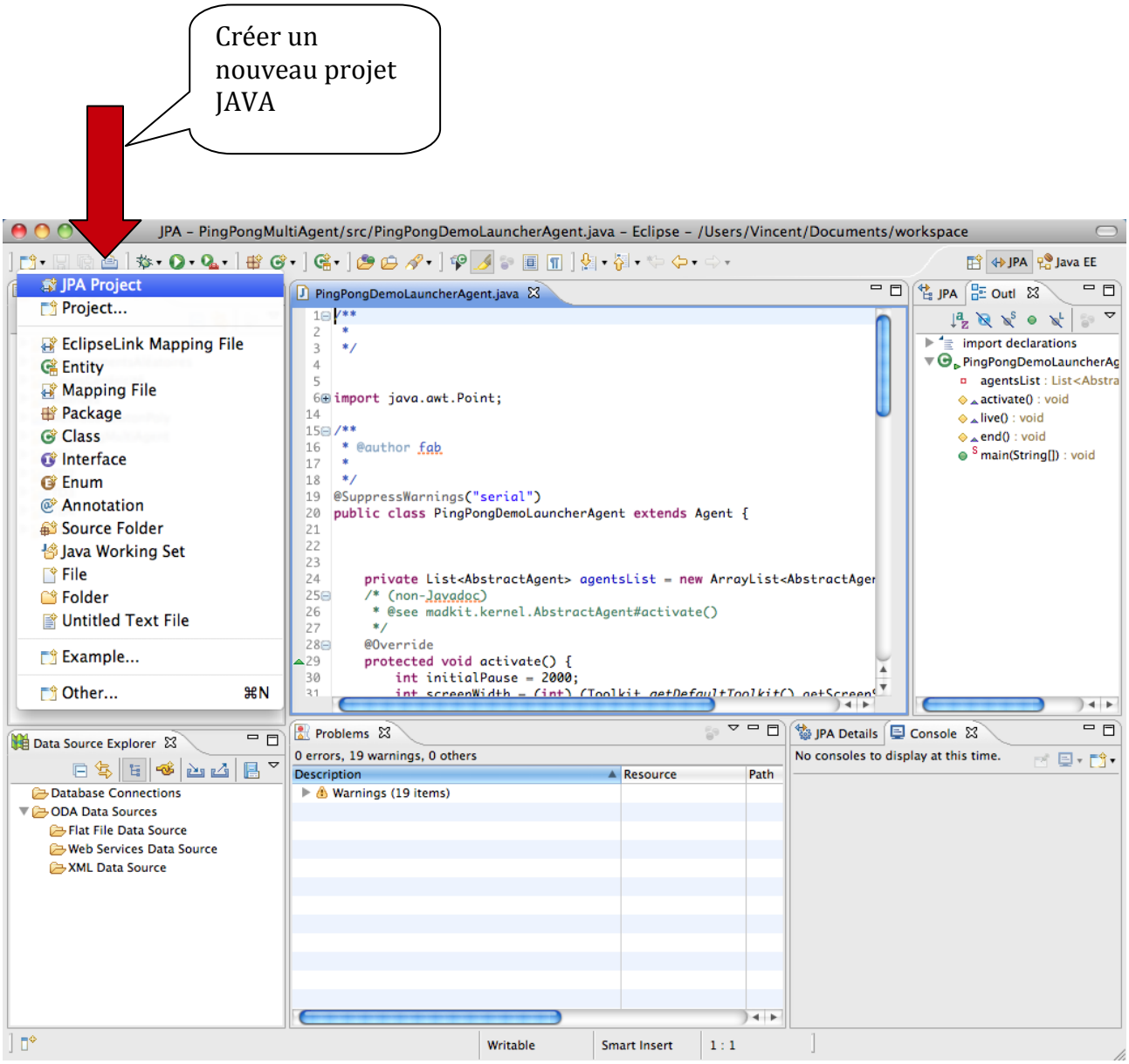
- Comme l'indique le type paramétrisé "*T*", la classe *Message* est une classe générique, il est donc possible pour un agent d'envoyer comme message une instance de n'importe quelle classe, par exemple un *Integer*, une *String*, mais aussi n'importe quelle classe définie par le programmeur.
- La classe *Message* définie également les deux méthodes suivantes : *getReceiver()* et *getSender()*. Comme leurs noms l'indiquent, ces deux méthodes permettent d'obtenir l'adresse de l'expéditeur et du destinataire d'un message, cette adresse étant représentée par une instance de la classe *AgentAdress*.

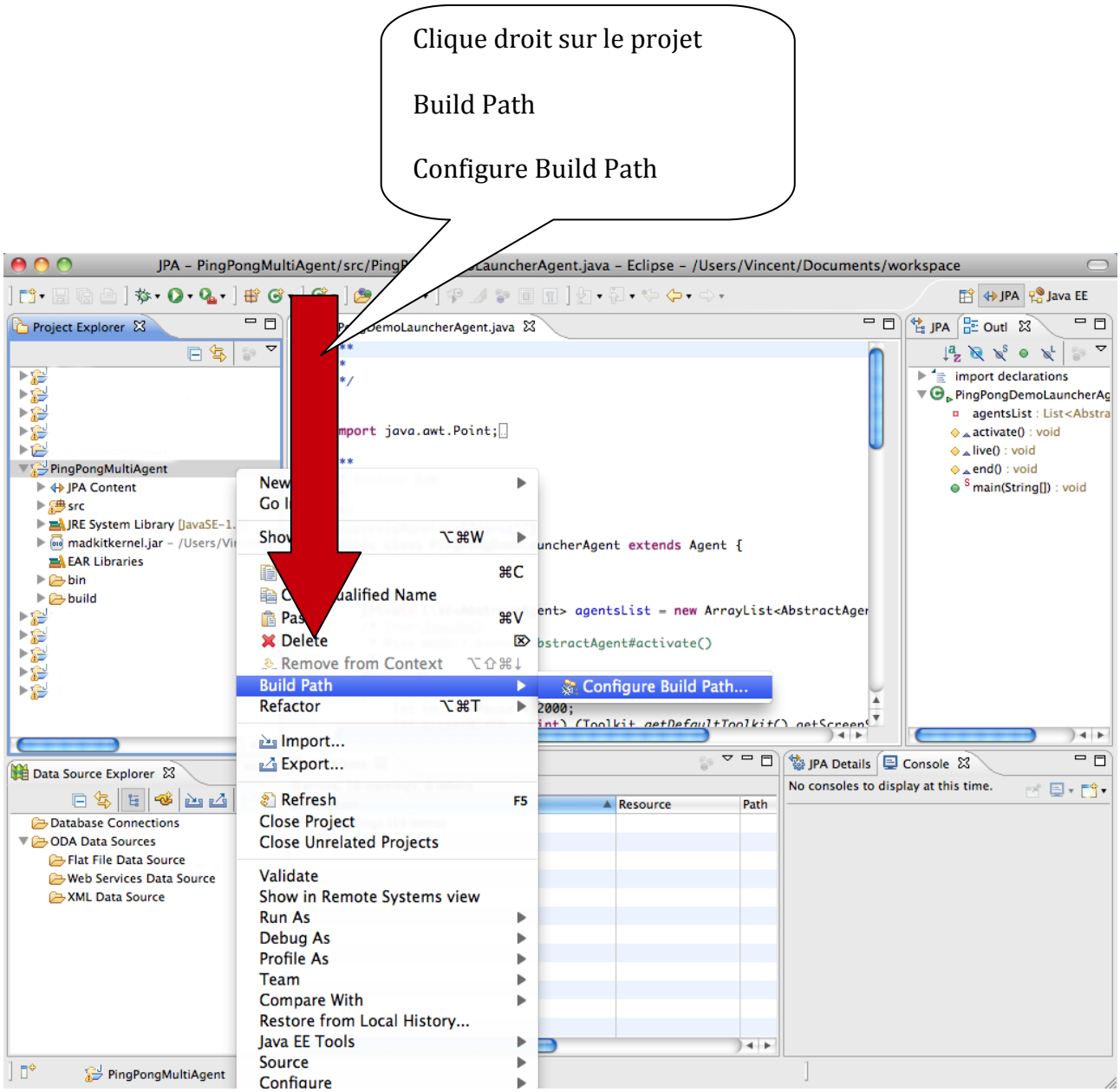
L'envoi des messages sous MadKit est asynchrone, ce qui signifie que l'expéditeur envoie un message au destinataire, mais ne se bloque pas en attendant la réponse. Également, l'expéditeur ne sait pas si le message a effectivement atteint le destinataire.

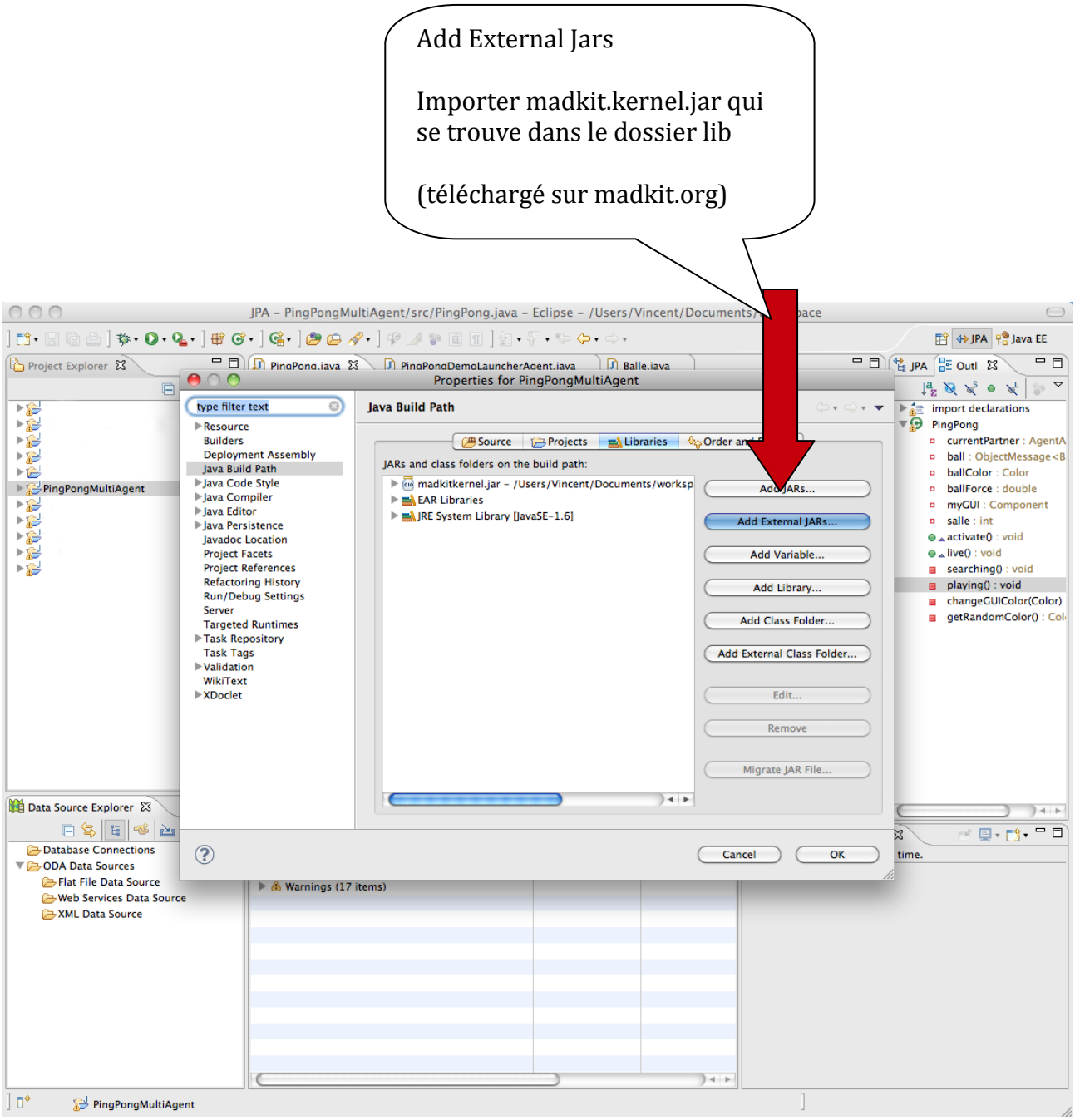
3.3. Compréhension et modification d'une application existante

3.3.1. Utilisation de MadKit sous Eclipse

- Télécharger MadKit 5.0 sur www.madkit.org
- Sous Eclipse :




Clique droit sur le projet
Build Path
Configure Build Path
Project Explorer
PingPongMultiAgent
JPA Content
src
JRE System Library
madkitkernel.jar
EAR Libraries
bin
build
New
Go I
Show
Copy
Copy Qualified Name
Paste
Delete
Remove from Context
Build Path
Refactor
Import...
Export...
Refresh
Close Project
Close Unrelated Projects
Validate
Show in Remote Systems view
Run As
Debug As
Profile As
Team
Compare With
Restore from Local History...
Java EE Tools
Source
Configure
Configure Build Path...
Data Source Explorer
Database Connections
ODA Data Sources
Flat File Data Source
Web Services Data Source
XML Data Source
JPA Details
Console
No consoles to display at this time.
Resource
Path
import java.awt.Point;
import declarations
agentsList : List<Abstra
activate() : void
live() : void
end() : void
main(String[]) : void


Add External Jars
Importer madkit.kernel.jar qui se trouve dans le dossier lib
(téléchargé sur madkit.org)
Properties for PingPongMultiAgent
Java Build Path
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Add External Class Folder...
Edit...
Remove
Migrate JAR File...
Cancel
OK

3.4. Comprendre le fonctionnement de la démo Ping-pong

Dans le but de comprendre comment fonctionne MadKit et plus généralement un Système Multi-Agent, nous avons étudié le code d'une application modélisant le déroulement d'une partie de Ping-pong entre plusieurs agents.

Cette application met en œuvre les bases d'un SMA : modèle relationnel et communication entre agents.

Le programme fonctionne de la façon suivante :

- On lance un petit nombre d'agents (≈ 10)
- Chaque agent demande le rôle "player"
- Chaque agent cherche un partenaire, pour cela il se met en attente de message pendant un certain laps de temps, s'il reçoit un message, alors il considère l'expéditeur du message comme partenaire, sinon, il demande l'adresse d'un agent ayant le rôle "player"
- Les deux agents envoient chacun 10 "balles" à leur partenaire. Une balle étant représentée par un message.
- Lorsque les 10 balles ont été envoyées, l'agent cherche un nouveau partenaire pour effectuer une nouvelle partie.

3.5. Modifier la démo Ping-pong

Nous avons par la suite apporté des modifications à ce programme, afin de nous familiariser avec MadKit.

3.5.1. Gestion des points

Premièrement, nous avons essayé de compter les points lors d'un match. Pour cela, nous avons utilisé la démarche suivante : lorsqu'il envoie une balle, l'agent tire un nombre entre 0 et 1, au hasard, et inclut ce nombre dans le message qu'il envoie. Ce nombre représente la force de la balle. Le partenaire, quant à lui, tire également un nombre aléatoirement entre 0 et 1, ce nombre représente la défense du joueur. Si la défense est supérieure ou égale à l'attaque, le receveur marque un point, sinon il ne marque pas.

```
if (ball.getContent().force() > Math.random()) {
                            res = " I won the point";
                            nbrePoints++;
                        } else
                            res = " I lost the point";
```

Cette portion de code permet au joueur de déterminer s'il a réussi à envoyer la balle.

3.5.2. Système d'élimination de joueurs

Par la suite, nous avons essayé de représenter un tournoi de la manière suivante. Un joueur possède un handicap, initialisé à 5. A la fin d'une partie, si le joueur a un score strictement inférieur à son handicap, il est éliminé, sinon il continue à jouer et son handicap est augmenté d'une unité.

```
private final static int SCORE_ELIMINATOIRE = 5;
```

On définit un score minimal à atteindre pour ne pas être éliminé.

```
private int handicap = 0;
```

Cet attribut permet de prendre en compte le handicap du joueur.

```
if (nbrePoints < (PingPong.SCORE_ELIMINATOIRE + this.handicap))
                killAgent(this);
          else
                this.handicap+
```

Cette portion de code permet de déterminer l'issue d'un match :

- Si le joueur a un score strictement inférieur à la somme du score minimum et de son handicap, il est éliminé
- Sinon, il reste dans la compétition, et son handicap est incrémenté.

3.5.3. Création d'un tableau des scores

Enfin, nous avons voulu tenir un tableau des scores. Pour cela nous avons utilisé la bibliothèque Swing pour concevoir une fenêtre représentant le tableau des scores, ainsi qu'une classe *Arbitre*. Lorsqu'une partie est finie, le joueur envoi son score à l'arbitre, qui se charge d'afficher le score sur le tableau des scores.

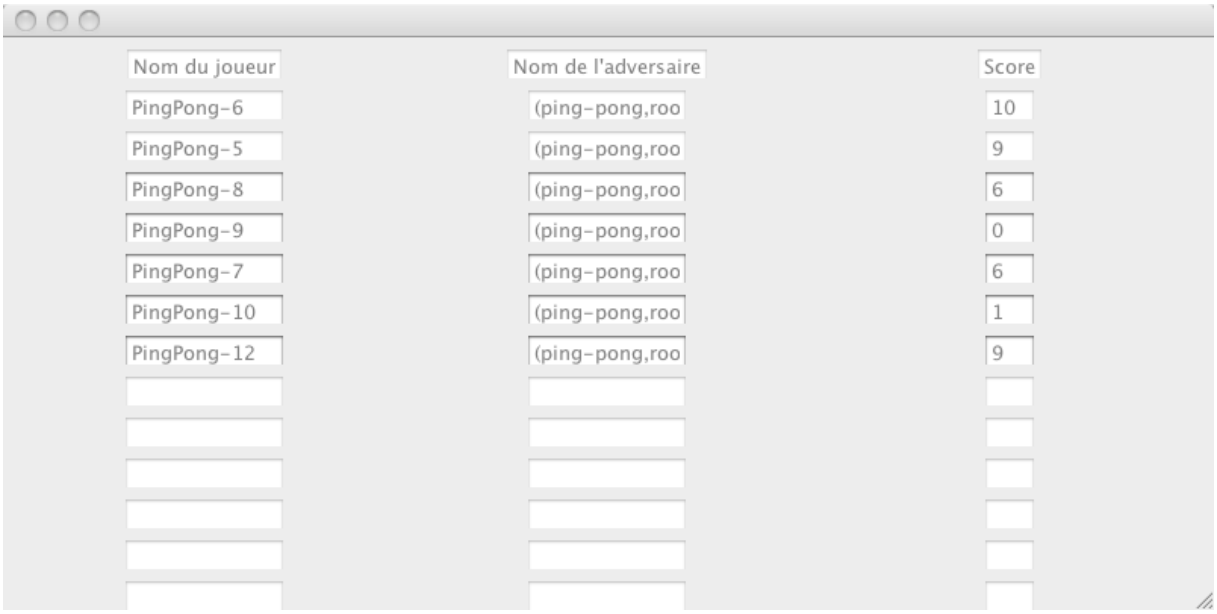


Figure 1 - Tableau des scores

```
sendMessageWithRole("ping-pong","room","arbitre",
                        new ObjectMessage<ResulatMatch>(
new ResulatMatch(this.getName(),currentPartner.toString(),nbrePoints)),"player");
```

Cet extrait de code est exécuté par chaque agent, à chaque fois qu'un match est fini. Il fait usage de la méthode sendMessageWithRole() qui permet d'envoyer un message sans spécifier l'adresse de l'agent destinataire, mais simplement en indiquant le rôle de l'agent que l'on cherche à joindre (ici "arbitre").

4. Réalisation de tutoriels pour apprendre à utiliser MadKit

4.1. Réalisation d'exemples simples mettant en œuvre ces fonctionnalités

Ce projet demande principalement de détailler et d'expliquer tout le travail réalisé de manière à faciliter l'utilisation et la compréhension des utilisateurs futurs.

Nous avons donc décidé de créer quatre exemples simples de difficultés croissantes pour arriver jusqu'à une application de ping-pong.

Ces quatre exemples traitent des fonctions de bases de MadKit et aident à se familiariser avec cette nouvelle programmation.

1-Communication entre agents

Le premier exercice illustre les mécanismes basiques d'envoi et de réception de messages. Notamment, la possibilité de connaître la provenance d'un message.

2-Communication eu sein d'un modèle organisationnel

Le deuxième fait apparaître la notion de groupe et de rôle au sein de celui-ci. Cet exercice montre la communication entre les différents agents de ce groupe malgré le fait qu'ils peuvent avoir un rôle différent.

2bis-Difficultés au sein d'un modèle organisationnel

Un exercice deux bis a également été réalisé il reprend le principe de l'exercice deux. Contrairement au précédent celui-ci crée deux groupes, chacun contenant également deux rôles différents.
Cet exercice a pour but de montrer les limites du modèle organisationnel* : la communication entre deux groupes distincts n'est pas possible.

3-Réagir suivant la provenance d'un message

L'exercice trois est basé sur le fait de connaître la provenance d'un message. En effet deux agents échangent des messages et suivant le lien qui les unit, ils décident de faire telle ou telle chose.

4-Interaction avec l'environnement

Ce quatrième exercice met en avant le fait que l'environnement dans lequel vit un agent peut influencer ses choix. Ainsi à chaque changement d'état de l'environnement, les agents qui évoluent dans celui-ci verront leur état se modifier également.

4.2. Tutoriaux permettent de réaliser 4 exemples de SMA

4.2.1. Exemple 1

But : mettre en œuvre une communication simple entre quelques agents et montrer un exemple simple d'une façon dont un agent peut gérer les données qu'il possède.

Fonctionnalités de MadKit misent en œuvre :

- Envoi de message

Enoncé : on souhaite créer un petit groupe d'agents, ces agents devront adopter envoyer à intervalles réguliers des messages, le contenu de ces messages sera déterminé aléatoirement parmi un ensemble de messages prédéfinis. Un agent possèdera en donnée une liste d'amis, si l'agent reçoit une demande d'ajout à sa liste d'amis, l'agent devra soit ajouter l'expéditeur du message à sa liste d'amis, soit refuser la demande. Si un agent reçoit un refus, il mettra fin à son exécution.

Solution proposée :
Les agents seront représentés par la classe *AgentQuiParle*, qui dérive de la classe *Agent*.

La liste d'amis sera représentée par un objet instance de la classe *ArrayList<AgentAdress>*, déclaré en attribut de la classe *AgentQuiParle*. (L'utilisation d'un autre type de structure de donnée que *l'ArrayList* aurait été tout à fait possible)

Afin de savoir combien d'agents sont actuellement en cours d'exécution, la classe *AgentQuiParle* déclare une variable *nombreAgents*. Cette variable permettra d'éviter à un agent d'envoyer des messages s'il n'y a personne pour les recevoir.

Pour permettre la fonctionnement de nos agents sous MadKit, ont redéfini les trois méthodes qui gèrent le cycle de vie d'un agent :

- La méthode *activate()*

```
public void activate() {
            this.listeAgents = new
ArrayList<AgentAddress>();
            createGroupIfAbsent("amis", "piece");
            requestRole("amis", "piece", "personne",
null);
      }
```

Dans cette méthode, l'agent construit sa liste d'amis, puis utilise les méthodes *createGroupIfAbsent()* et *requestRole()* pour respectivement, créer le groupe "piece" au sein de la communauté "amis", s'il n'existe pas déjà, et demander de jouer le rôle "personne" dans le groupe cité précédemment.

- La méthode l*ive()*

```
public void live() {
            pause(5000);
            while (true) {
                  if(Nombreagents==1)
                  {
                        killAgent(this);
                  }
                  ObjectMessage<String> m = (ObjectMessage<String>) waitNextMessage(3000);
                  if (m != null) {
                        if (m.getContent().equals("prenom?")){
                              sendMessage(m.getSender(), new ObjectMessage<String>(
                              "et toi?"));waitNextMessage(1000);
                              this.listeAgents.add(m.getSender());

                              if (logger != null)
                                    logger.info("je suis " + m.getReceiver());

                        }
                        if (m.getContent().equals("et toi?")) {
                              sendMessage(m.getSender(), new ObjectMessage<String>(
                                          "pas envie de te le
dire"));waitNextMessage(3000);
                              if (logger != null)
                                    logger.info("pas envie de lui qui je suis dire mais lui
il s'apelle" + m.getSender());

                        }
                        if (m.getContent().equals("pas envie de te le dire")) {
                              killAgent(this);
                              Nombreagents--;
                        }

                  }
                  if (Math.random() < 0.7)
                        sendMessageWithRole("amis", "piece", "personne",
                                    new ObjectMessage<String>("prenom?"), "personne");
                  else
                        sendMessageWithRole("amis", "piece", "personne",
                                    new ObjectMessage<String>("et toi?"), "personne");
            }
      }
```

Cette méthode effectue une boucle infinie contenant les instructions suivantes :

1. On attend un certain laps de temps (ici, 3000 millisecondes) la réception d'un message.
2. Si un message est réceptionné pendant ce laps de temps, on le traite.

3. Dans tous les cas, on effectue un tirage aléatoire pour déterminer si le message que l'on va envoyer à un membre du groupe "piece" jouant le rôle "personne".

Remarque : l'instruction "`pause(5000);`" située au début de la méthode *live()* a pour but de permettre à tous les agents d'être lancés avant de commencer la communication.

Remarques générales :

- On a utilisé ici une grande partie des primitives de MadKit permettant la communication entre agents, cependant de nombreuses variantes existent. Il est en effet possible d'envoyer un message à un agent particulier (grâce à l'*AgentAdress)* mais également d'effectuer un envoi de message avec attente de réponse, par le biais de la méthode *sendMessageAndWaitForReply*() de la classe *Agent*.
- La méthode *live()* de cette classe est un algorithme stochastique, c'est-à-dire basé sur le hasard, il est donc impossible de prévoir le résultat de cet algorithme à chaque nouvelle exécution, par opposition à un algorithme déterministe pour lequel une telle prédiction serait possible.

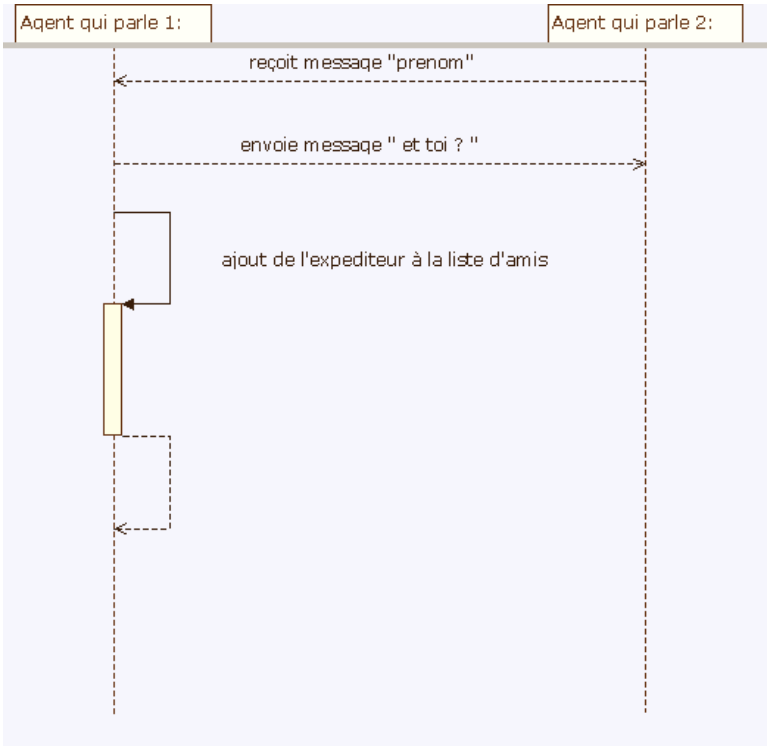


Diagramme de séquence de la méthode live() de la classe AgentQuiParle après réception d'un message « prenom » .

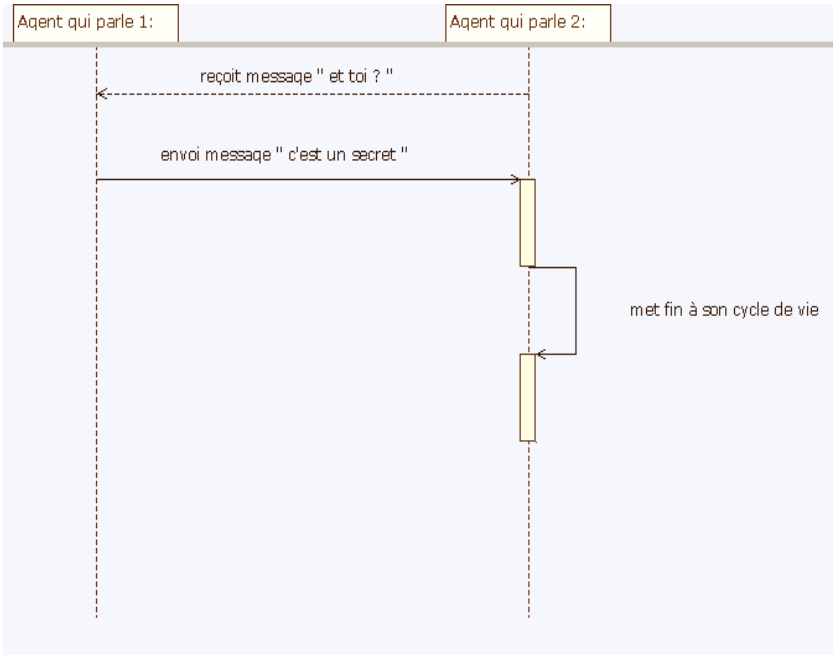


Diagramme de séquence de la méthode live() de la classe AgentQuiParle apres réception d'un message « et toi ? ».

4.2.2. Exemple 2

But : mettre en œuvre la communication au sein d'un même groupe composé d'agents ayant des rôles différents.

Fonctionnalités de MadKit misent en œuvre :

- Création d'un groupe
- Requête d'un rôle au sein d'un groupe
- Envoi de message au sein du groupe à un agent d'un rôle différent

Enoncé : on souhaite créer un petit groupe d'agents qui peuvent avoir deux rôles : professeur ou élève. Un professeur pose une question, par la suite chaque agent élève lui répond, le contenu de message sera déterminé aléatoirement parmi un ensemble de messages prédéfinis. Une fois les réponses reçues le professeur répond de manière positive ou négative aux agents. Si un agent reçoit une réponse négative, il met fin à son cycle de vie.

Solution proposée :
Les agents seront représentés par les classes *Professeur* et *Élèves*, qui toutes deux dérivent de la classe *Agent.*

La liste d'agents sera représentée par un objet instance de la classe *ArrayList<AbstractAgent>,* déclaré en attribut de la classe *Lanceur,* afin de pouvoir connaître à tout moment la liste des agents encore en cours d'exécution.

Afin de savoir combien d'agents ont déjà mis fin à leur cycle de vie, la classe *Professeur* déclare une variable *nombreErreurs*. Cette variable permettre d'éviter à un agent Professeur d'envoyer des messages s'il n'y a personne pour les recevoir.

Pour pouvoir programmer le cycle de vie de l'agent, on redéfinit les méthodes a*ctivate() et* l*ive() :*

- La méthode *activate()*

```
Dans la classe Eleve:

public void activate() {
            createGroupIfAbsent("salle K029", "lycee");
            requestRole("salle K029", "lycee", "eleve", null);
      }

Dans la classe Professeur:

public void activate() {
            createGroupIfAbsent("salle K029", "lycee");
            requestRole("salle K029", "lycee", "prof", null);
      }
```

Dans cette méthode, l'agent utilise les méthodes *createGroupIfAbsent()* et *requestRole()* pour respectivement, créer le groupe "lycee" au sein de la communauté "salle K029", s'il n'existe pas déjà, et demander de jouer le rôle "eleve" ou « prof » dans le groupe cité précédemment.

- La méthode *live()* :

```
Dans la classe Eleve:

public void live() {
            pause(5000);
            while (true) {
                  ObjectMessage<String> m = (ObjectMessage<String>)
waitNextMessage(1000);
                  if (m != null) {
                        if (m.getContent().equals("Perdu")) {
                              if (logger != null)
                                    logger.info("Je me suis trompe...");
                              killAgent(this);

                        } else {
                              if (logger != null)
                                    logger.info("J'ai gagne");

                        }

                  }
                  if (Math.random() < 0.70)
                        sendMessageWithRole("salle K029", "lycee", "prof",
                                    new ObjectMessage<String>("4"), "eleve");
                  else
                        sendMessageWithRole("salle K029", "lycee", "prof",
                                    new ObjectMessage<String>("2"), "eleve");
            }
                                          }
```

Cette méthode effectue une boucle infinie contenant les instructions suivantes :

1. On attend un certain laps de temps (ici, 1000 millisecondes) la réception d'un message.
2. Si un message est réceptionné pendant ce laps de temps, on le traite et suivant celui-ci l'agent met fin ou non à son cycle de vie.
3. Dans tous les cas, on effectue un tirage aléatoire pour déterminer le message que l'on va envoyer à un membre du groupe "lycee" jouant le rôle "prof".

Remarque : l'instruction "`pause(5000);`" située au début de la méthode *live()* a pour but de permettre à tout les agents d'être lancés avant de commencer la communication limitant ainsi les erreurs d'envoi de message envers un agent qui n'existe pas encore.

```
Dans la classe Professeur:

public void live() {
            pause(5000);
            while (true) {
                  if (this.nbreErreurs == 10) {
                        if (logger != null)
                              logger.info("Je n'ai plus d'eleve...");
                  } else {
                        if (Math.random() > 0.95) {
                              if (logger != null)
                                    logger.info("Je pose une question");
                        sendMessageWithRole("salle K029", "lycee", "eleve",
                                          new ObjectMessage<String>("2+2"), "prof");
                        }
                  }

                  ObjectMessage<String> m = (ObjectMessage<String>)
waitNextMessage(1000);
                  if (m != null) {
                        if (m.getContent().equals("2")) {
                              this.nbreErreurs++;
                              sendMessage(m.getSender(), new ObjectMessage<String>(
                                          "Perdu"));
                        }

                        if (m.getContent().equals("4")) {
                              sendMessage(m.getSender(), new ObjectMessage<String>(
                                          "Bravo"));
                              waitNextMessage(1000);
                              if (logger != null)
                                    logger.info("Bravo" + m.getSender());

                        }

                  }

            }
      }
```

Cette méthode effectue une boucle infinie contenant les instructions suivantes :

1. Si le nombre d'erreurs est de 10 , l'agent n'a plus d'élève et stoppe la communication , sinon il pose une question aux membres du groupe « lycee » jouant le rôle « eleve »
2. On attend un certain laps de temps (ici, 1000 millisecondes) la réception d'un message.
3. Si un message est réceptionné pendant ce laps de temps, on le traite et suivant celui-ci l'agent envoi un message a son expéditeur.

Remarque : l'instruction "`pause(5000);`" située au début de la méthode *live()* a pour but de permettre à tout les agents d'être lancés avant de commencer la communication limitant ainsi les erreurs d'envoi de message envers un agent qui n'existe pas encore.

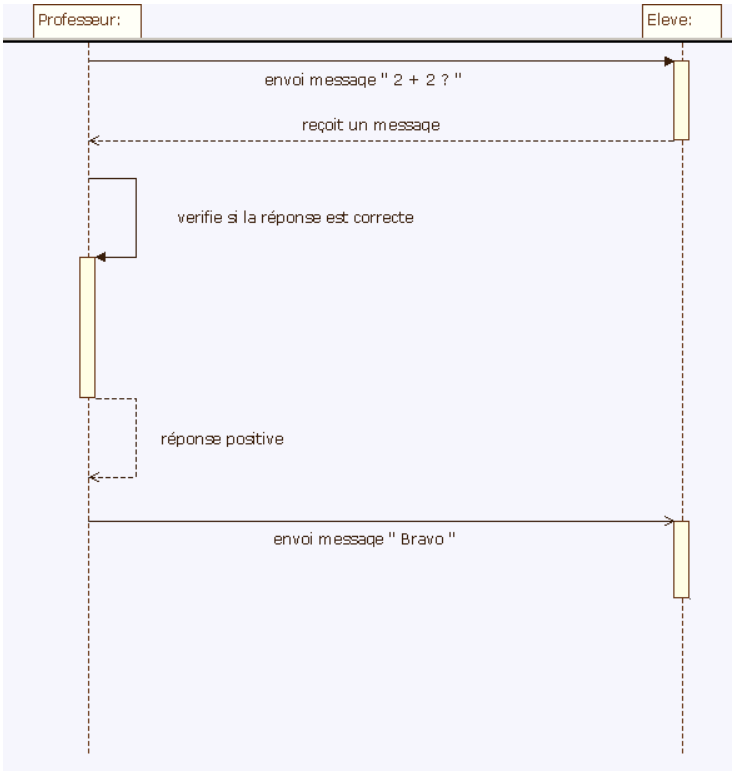


Diagramme de séquence des méthodes *live()* des classes Eleve et Professeur lorsqu'une question est envoyée et que la réponse reçue est correcte

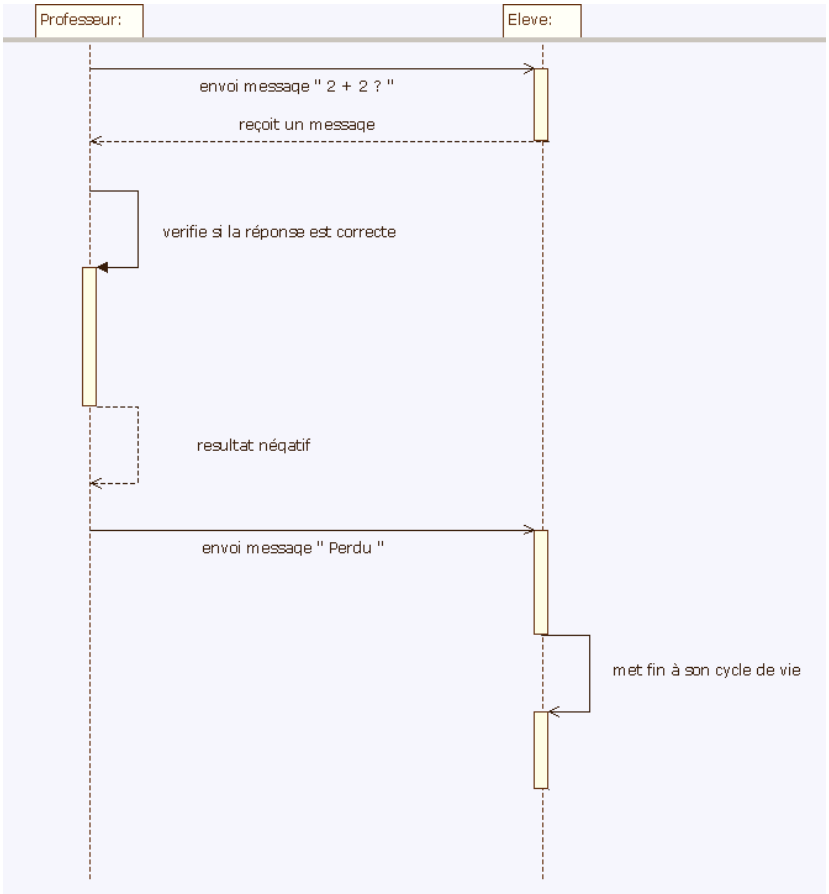


Diagramme de séquence des méthodes *live()* des classes Eleve et Professeur lorsqu'une question est envoyée et que la réponse reçue est incorrecte

4.2.3. Exemple 2bis

But : reprendre l'exercice précédent afin de montrer que la communication entre deux groupes différents n'est pas possible. Pour communiquer deux agents doivent nécessairement appartenir au même groupe.

Fonctionnalités de MadKit misent en œuvre :

- Création d'un groupe
- Requête d'un rôle au sein d'un groupe
- Envoi de message au sein du groupe à un agent d'un rôle différent

Enoncé : on souhaite créer un petit groupe d'agents qui peuvent avoir deux rôles : professeur ou élève. Un professeur pose une question, par la suite chaque agent élève lui répond, le contenu de message sera déterminé aléatoirement parmi un ensemble de messages prédéfinis. Une fois les réponses reçues le professeur répond de manière positive ou négative aux agents. Si un agent reçoit une réponse négative, il met fin à son cycle de vie.

Solution proposée :
Les agents seront représentés par les classes *ProfesseurCollege* , *ÉlèveCollege*, *ProfesseurLycee* et *EleveLycee* qui dérivent de la classe *Agent.*

La seule différence avec l'exercice précédent apparaît dans la méthode live().
En effet pour montrer que la communication n'est pas possible entre ces deux groupes , un agent de la classe *EleveCollege* essaye d'envoyer un message à un agent de la classe *ProfesseurLycee.*

```
public void live() {
            pause(5000);
            while (true) {
                  ObjectMessage<String> m = (ObjectMessage<String>)
waitNextMessage(1000);
                  if (m != null) {
                        if (m.getContent().equals("Perdu")) {
                              if (logger != null)
                                    logger.info("Je me suis trompe...");
                              killAgent(this);

                        } else {
                              if (logger != null)
                              logger.info("J'ai gagne");

                        }

                  }
                  if (Math.random() < 0.70)
                        sendMessageWithRole("personne", "lycee", "prof",
                                    new ObjectMessage<String>("4"),
"eleve");
                  else
                        sendMessageWithRole("personne", "lycee", "prof",
                                    new ObjectMessage<String>("2"),
"eleve");
            }
      }
```

Remarques générales :

Lors de l'exécution il est tout à fait normal de voir apparaître des messages d'erreur de ce type :

```
sendMessageWarning: Envoi de message échoué : Je suis pas dans ce groupe: Groupe
<personne,lycee>
      at Exercice2bis.ElevesCollege.live(ElevesCollege.java:32)
```

4.2.4. Exemple 3

But : montrer de façon simple les différentes réactions des agents en fonction des messages qu'ils reçoivent et en utilisant les données qu'ils possèdent.

Fonctionnalités de MadKit misent en œuvre :

- Envoi de messages
- Exceptions pouvant être lancées lors de l'envoi d'un message
- Représentations des données d'un agent

Enoncé : on souhaite représenter une communauté d'agents de petite taille (≈ 50 agents) qui peuvent, par le biais de l'envoi de messages, demander à un autre agent d'être son ami ou bien de partir (ici, mettre fin à son exécution) avec lui, si le destinataire du message est ami avec le receveur il accepte, sinon il refuse.

Solution proposée :
On définit une classe *AgentAmis* qui dérive de la classe *Agent.*

Dans cette classe, on déclare en attribut un objet instance de la classe *ArrayList<AgentAdress>* qui permettra à un agent de stocker l'adresse de ses amis.

On choisit, volontairement, de ne pas comptabiliser le nombre d'agents en cours d'exécution. Ainsi, un agent n'a pas de moyen de savoir s'il existe un destinataire au message qu'il envoie.

Pour pouvoir programmer le cycle de vie de l'agent, on redéfinit les méthodes *activate() et live() :*

```
public void activate() {
            this.listeAmis = new ArrayList<AgentAddress>();
            createGroupIfAbsent("amis", "piece");
            requestRole("amis", "piece", "personne", null);
                                    }
```

On retrouve dans cette méthode les mêmes mécanismes que dans les exercices précédents.

```
	public void live() {
		pause(5000);
		while (true) {
			ObjectMessage<String> m = (ObjectMessage<String>)
waitNextMessage(1000);
			if (m != null) {
				if (m.getContent().equals("ami?")) {
					this.listeAmis.add(m.getSender());
					if (logger != null)
						logger.info("Je suis amis avec " +
m.getSender());
				}
				if (m.getContent().equals("partir?")) {
					if (listeAmis.contains(m.getSender())) {
						sendMessage(m.getSender(), new
ObjectMessage<String>(
								"on y va"));
						if (logger != null)
							logger.info("Je part avec " +
m.getSender());
						killAgent(this);
					}
				}
				if (m.getContent().equals("on y va"))
					killAgent(this);
			}
				if (Math.random() < 0.95)
				sendMessageWithRole("amis", "piece", "personne",
						new ObjectMessage<String>("ami?"),
"personne");
			else
				sendMessageWithRole("amis", "piece", "personne",
						new ObjectMessage<String>("partir?"),
"personne");
		}
	}
```

Dans cette méthode, on répète indéfiniment les opérations suivantes :

1. On atteint la réception d'un message pendant un laps de temps prédéfini (ici, 1000 millisecondes).
2. Si un message est reçu alors : s'il contient une demande d'ajout à la liste d'amis, alors l'agent ajoute l'*AgentAdress* de l'expéditeur à sa liste d'amis, s'il contient une invitation à partir, alors si l'AgentAdress de l'éxpéditeur est contenue dans la liste d'amis, l'agent envoi une confirmation puis "part" (appel de la méthode *killAgent()* sur lui-même). On peut remarquer que l'agent n'informe pas le reste de son groupe de son départ.

3. Enfin, l'agent effectue un tirage aléatoire : il y a 95% de chances qu'il envoie une demande d'ajout à la liste d'amis et 5% de chances qu'il envoie une invitation à partir.

Remarques générales :

- Lors de l'exécution, il est très probable d'avoir sur la console le message :

```
sendMessageWarning:  TODO : write message  invalid
                     address
   at Ex3.AgentAmis.live(AgentAmis.java:35)
```

Comme l'envoi des messadges est asynchrone, il peut arriver qu'un agent A demande à un agent B de partir, puis que l'agent A envoie une autre invitation de départ à un agent C. Puisque l'agent B aura mis du temps à lui répondre, à ce moment là, il est possible que l'agent A reçoive la confirmation de départ de l'agent B, il met alors fin à son exécution. *L'AgentAdress* que possède l'agent C pour joindre l'agent A n'est alors plus valide puisque l'agent A n'existe plus.

```
                    WARNING :
sendMessageWarning:  TODO : write message  no body found
     at Ex3.AgentAmis.live(AgentAmis.java:46)
```

Puisque nous avons, volontairement, choisi de ne pas comptabiliser le nombre d'agents en cours d'exécution, il peut arriver qu'un agent se retrouve seul, les messages qu'il envoie n'auront alors pas de destinataire.

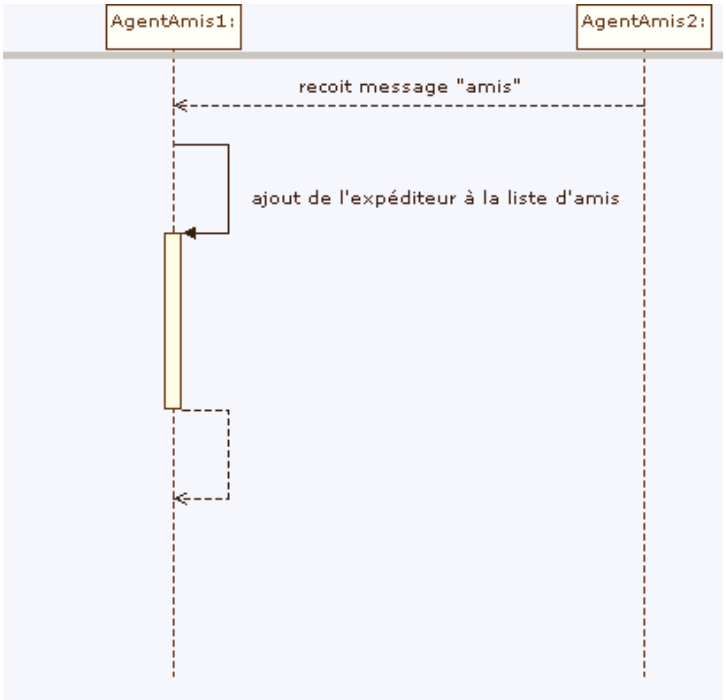


Diagramme de séquence de la méthode *live()* lorsqu'une demande d'amis est reçue

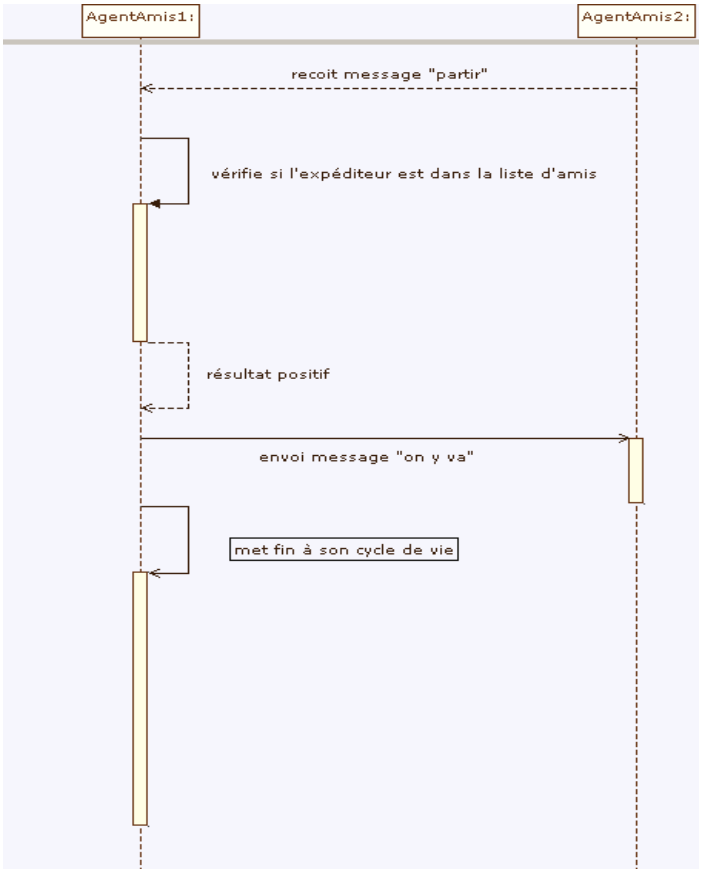


Diagramme de séquence de la méthode *live()* lorsqu'une invitation à partir est reçue, et que l'agent accepte l'invitation.

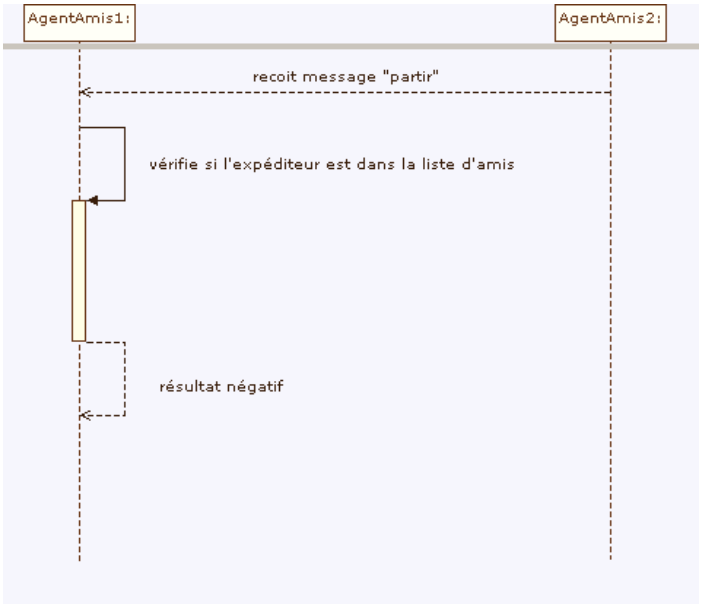


Diagramme de séquence de la méthode *live()* lorsqu'une invitation à partir est reçue, et que l'agent n'accepte pas l'invitation.

4.2.5. Exemple 4

But : montrer, au travers d'un exemple simple, comment un agent peut utiliser les données de son environnement pour décider du comportement qu'il va adopter.

Fonctionnalités de MadKit misent en œuvre :

- Représentation de l'environnement d'un agent.

Enoncé : on souhaite concevoir un agent dont l'environnement est constitué d'un entier, représentant la température extérieure (en °C). L'agent doit pouvoir lire la température et réagir en fonction de celle-ci, en choisissant une activité adaptée.

Solution proposée :
On représente l'agent à l'aide de la classe *AgentMeteo* qui hérite de la classe *Agent*.

La classe *AgentMeteo* possède en variable de classe (attribut "static") un entier représentant la température extérieure.

La classe *AgentMeteo* gère le cycle de vie d'un agent en redéfinissant les méthodes :

```
public void activate() {

	}
```

Comme on ne gère qu'un seul agent, il est inutile de se servir du modèle Communauté/Groupe/Rôle.

```
public void live() {
		pause(5000);
		while (true) {
			if (temperature <= 0) {
				if (logger != null)
					logger.info("On va faire du ski!");
			}
			 if (temperature >= 30) {
				if (logger != null)
					logger.info("On va à la plage!");
			} else {
				if (logger != null)
					logger.info("On fait rien de special
:(");
			}
			pause(500);
		}
	}
```

L'agent effectue un aiguillage logique en fonction de la température extérieure, puis effectue une pause de 500 millisecondes.

On peut remarquer que l'agent ne fait que lire le contenu de la variable représentant la température. Il est donc nécessaire de modifier régulièrement cette valeur. Pour cela, on définit une classe *GestionnaireMeteo* qui hérite de la classe *AgentMeteo.* Le rôle de cette classe est de mettre à jour, à intervalle de temps régulier, la valeur de la température extérieure, et ce de façon aléatoire. Pour cela, on redéfinit dans la classe agent la méthode *live()* - (les méthodes *activate()* et *end()* ne sont pas redéfinies) :

```
public void live() {
            while (true) {
           temperature = (int) (Math.random() * 50 - 15);
                  pause(1000);
            }
      }
```

Le *GestionnaireMeteo* effectue une boucle infinie sur les instructions suivantes : mettre à jour la température avec une valeur aléatoire, tirée entre -15 et 49, puis se mettre en pause pendant 1000 millisecondes.

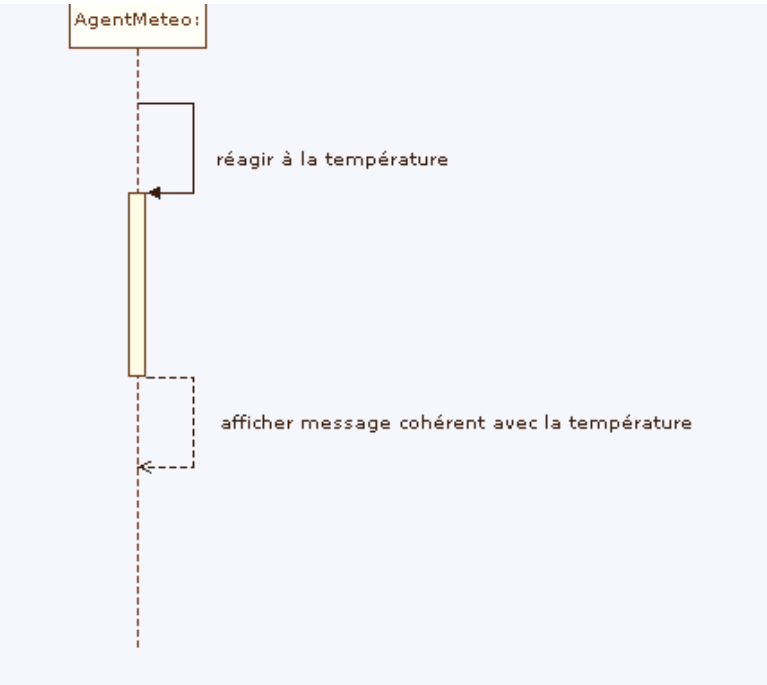


Diagramme de séquence d'une itération de la méthode *live()* pour la classe *AgentMeteo*

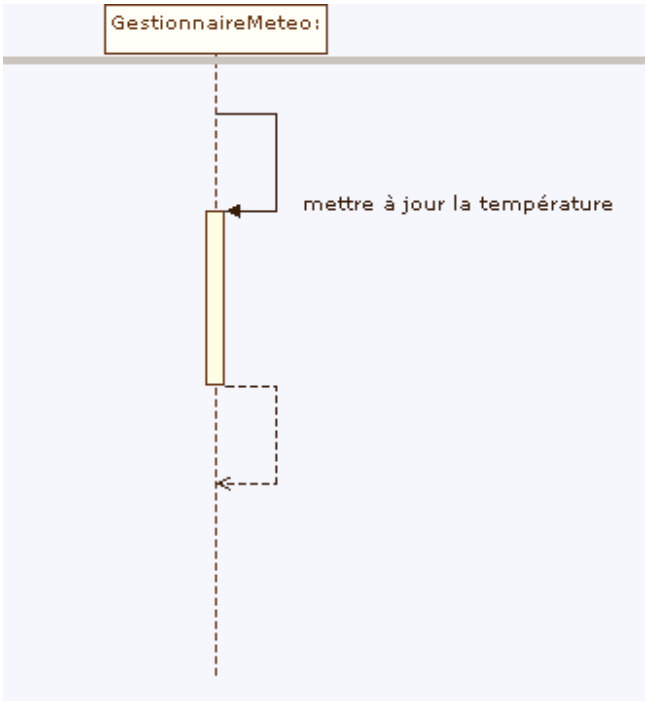


Diagramme de séquence d'une itération de la méthode *live()* pour la classe *GestionnaireMeteo*

5. Conclusion

L'objectif de notre projet était de comprendre comment un système multi-agent fonctionne et ce qu'il permet de réaliser. La finalité étant la rédaction de tutoriels afin de proposer aux futurs utilisateurs de MadKit une introduction au fonctionnement de ce système.

La méthode employée pour atteindre cet objectif parait la plus appropriée. En effet, le fait de nous être appuyés sur une application existante au début de notre projet nous a permis de comprendre comment fonctionne MadKit. Le fait d'implémenter progressivement de nouvelles fonctionnalités à cette application nous a permis de voir quels sont les points qui peuvent porter à confusion lorsque l'on programme sous MadKit. Nous avons ainsi pu mettre ces points en avant dans nos tutoriels.

Les tutoriels que nous avons rédigés se limitent aux fonctionnalités de base de MadKit, et d'un SMA en général, c'est-à-dire représentation d'un modèle organisationnel, la gestion de la communication entre agents et l'interaction des agents avec leur environnement. Il est toutefois important de noter que ces fonctionnalités à elles seules permettent la création de SMA relativement complexes.

En conclusion à ce projet nous avons énormément appris à propos d'un sujet que nous ne connaissions pas puisque celui-ci n'a jamais été abordé dans le cadre de nos études.
Nous nous somme ainsi rendu compte de la difficulté à expliquer et qu'il existe donc une grande frontière entre savoir faire et savoir expliquer.
Nous avons également appris beaucoup en ce qui concerne l'organisation et la répartition du travail puisque cette dernière s'est faite de manière naturelle et équitable.
Nous avons ainsi pu avoir un léger aperçu de ce qui nous attend prochainement dans le monde du travail.

Cependant le travail que nous avons effectué pourra être repris et approfondi, en concevant, par exemple, un SMA assez conséquent, et en détaillent et documentant chaque étape de la conception.

Résumé

En programmation, il est parfois nécessaire de réaliser un grand nombre d'opérations différentes en parallèle, par exemple lorsque l'on souhaite réaliser des simulations ou effectuer les calculs complexes. Les systèmes multi-agents permettent de modéliser un ensemble d'entités, appelés agents, qui évoluent au sein d'une société hiérarchisée et possèdent la capacité de communiquer entre eux. Il est alors relativement aisé de concevoir des modèles complexes où un grand nombre d'agents effectuent un nombre important d'opérations, tout en communiquant des informations entre eux. MadKit est une plateforme implémentant les principes de fonctionnement d'un système multi-agent, permettant ainsi de développer un SMA en faisant abstraction de la gestion de l'exécution en parallèle des agents et des problèmes de synchronisation des processus. Nous présentons dans ce rapport comment nous avons appris à concevoir des SMA sous MadKit, suivis de quelques tutoriels expliquant les étapes fondamentales de la création de la plupart des SMA.

Summary

When programming, it is sometimes necessary to perform a great number of different operation at the same time, for instance when performing simulations or complex computations. Multi agent systems allow the modelling of a set of entities, called agents, which evolve inside a hierarchized society and have the ability to communicate with one another. It is then relatively easy to design complex models inside of which an important number of agents perform various operations while exchanging information with one another. MadKit is a platform implementing the working principles of a multi agent system, allowing the programming of a MAS without the need of taking care of the issues that arise when multithreading. This rapport will explains how we learned to design MAS with MadKit, followed by a few tutorials explaining the fundamental steps of the creation of most MAS.