Master IHM

Systèmes Multi-Agents
Programmer avec MadKit

Chihab Hanachi, Matthias Mailliard

TP1. Premiers agents, premières organisations

Objectifs
Après une présentation des principes organisationnels de la plate-forme multi-agents MadKit, on cherchera à illustrer à travers trois exercices simples les notions d'agents, de groupes et de rôles, précédemment introduites.

I)Introduction

La plate-forme MadKit a été développé dès 1998 au LIRMM (Montpellier) par Olivier Gutchneck lors de sa thèse et Jacques Ferber son encadrant. Par la suite, toujours avec et encadrer par Jacques Ferber, Fabien Michel reprend le développement de la plate-forme et y intègre, notamment, TurtleKit un StarLogo-like (dédié à la simulation de type automate cellulaire).

1) La plate-forme MadKit

a) caractéristiques

MadKit (Multi-Agent Development Kit) est une plate-forme agent flexible. Parmi ces caractéristiques :

- S'appuie sur le modèle organisationnel AGR
- Pas de pré-requis sur le modèle agent
- Capable de supporter plusieurs modèles de communication simultanément
- Mode distribué transparent
- Interfaces graphiques componentielles et flexibles

b) motivations

La caractéristique principale de la recherche et des applications agents est la grande hétérogénéité du domaine :

- Les agents sont spécifiés et construit avec différents modèles (BDI, réactifs, situés...) et formalismes (programmation objets, logiques diverses, réseaux de Pétri...).
- Les agents utilisent différents protocoles d'interaction (ContractNet, FIPARequest, FishMarket...) et de communication (ACL, KQML...).
- Les Systèmes Multi-Agents (SMA) sont utilisés pour des buts variés et dans divers domaines d'application.

MadKit fait ainsi le pari que tirer simultanément profit de cette diversité d'approches est important pour construire des systèmes complexes, ainsi que de garder cette hétérogénéité gérable à partir de modèles conceptuels *et* d'outils logiciels appropriés.

La création de la boite à outils MadKit fut motivé par le besoin de fournir une plate-forme agent hautement modulable et malléable. Le but fut de construire une couche de base pour divers modèles agent, et de faire que les services de bases soient extensibles et remplaçables.

2) Le modèle conceptuel AGR (ex AALAADIN)

En vue d'être capable de programmer dans MadKit, il est important de comprendre le modèle conceptuel général qui le caractérise.
Nous allons rapidement présenter le modèle Agent/Groupe/Rôle (AGR) et les principales caractéristiques de MadKit à partir du point de vue du programmeur.

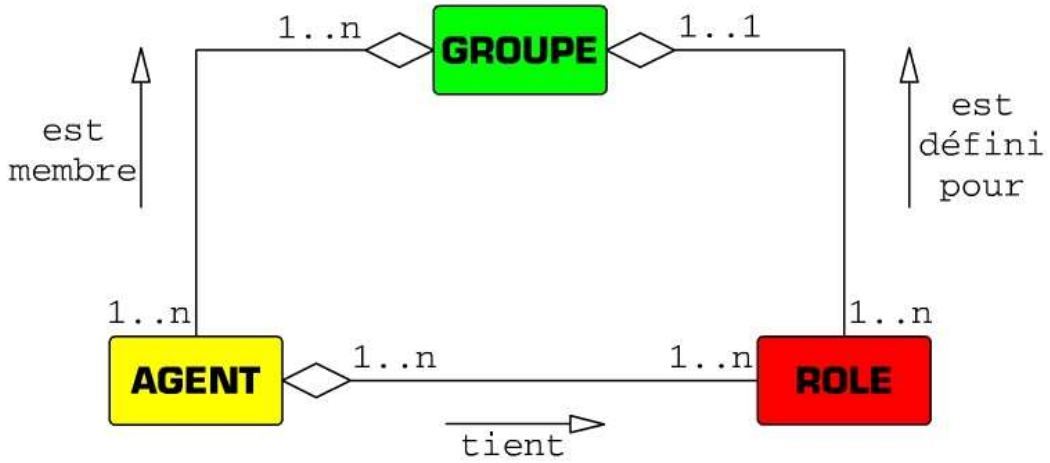


Figure A - le modèle Agent/Groupe/Rôle

a) Agent

Le modèle ne fixe aucune contraintes sur l'architecture interne de l'agent. Un *agent* est spécifié comme une entité active communicante qui joue des *rôles* dans des *groupes*.
Le concepteur de l'agent est responsable de choisir le modèle agent le plus approprié à la structure interne de l'agent suivant l'application envisagée.

b) Groupe

Les *groupes* sont définis comme des ensembles atomiques d'agrégation d'agent. Chaque agent fait partie d'un ou plusieurs groupes. Dans sa forme la plus simple un groupe est un ensemble d'agents. Dans sa forme la plus évolué, en conjonction avec la définition de rôle, il peut représenter n'importe quel système multi-agents usuel.
Les groupes ont les caractéristiques suivante :

- Un agent peut être membres de *n* groupes au même moment
- Les groupes peuvent donc librement se superposer
- Les groupes peuvent être créer par un agent, et un agent peut soumettre une requête d'admission pour n'importe quel groupe
- Grâce à des agents systèmes additionnels, un groupe peut être soit local soit distribué sur plusieurs machines

c) Rôles

Le *rôle* est la représentation abstraite d'une fonction, d'un service ou d'une identification agent dans chaque groupe. Chaque agent put tenir plusieurs rôles, chaque rôle tenu par un agent est local à un groupe. Pour tenir un rôle dans un groupe un agent doit soumettre une requête.

3) Architecture

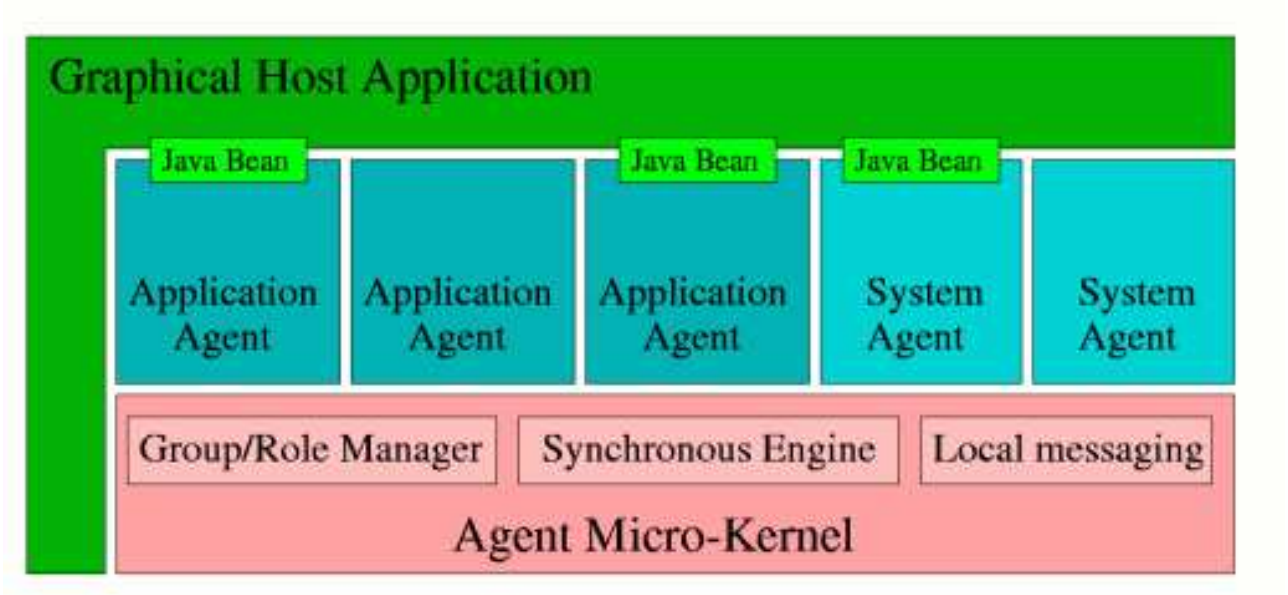


Figure 2- Architecture

La plate-forme MadKit est construite autour du modèle AGR. En plus de ces trois concepts cœur, la plate-forme ajoutes trois principes de design :

- Architecture du *micro-kernel*
- Agentification des services
- Interface graphique pour des modèles composants

MadKit lui-même est un ensemble de packages de classe Java qui implémentent le micro-kernel agent, ainsi que diverses bibliothèques de messages et d'agents.

Communautés

Les communautés sont des groupes de noyaux MadKit.

Modèle graphique

L'interface graphique MadKit est basée sur les spécifications Java Bean. Chaque agent est responsable de sa propre interface graphique dans chaque aspect (affichage, traitement des événements, actions...).

3) Programmer les agents (classe *AbstractAgent*)

a) Cycle de vie

Activation

Quand un agent est créé le kernel appelle sa méthode *activate()*. Cette méthode peut être vue comme une sorte de constructeur dans le mode agent. Toutes les initialisations doivent être faites dans la méthode ***activate()***.

Vie (uniquement pour la classe thread Agent sous classe de AbstractAgent)

La méthode ***live()*** est appelé par le kernel après la méthode *activate()*. Elle sert à gérer la vie de l'agent. A la sortie de la méthode, la méthode *end()* est appelée.

Mort

A la fin de la vie de l'agent (*killAgent()*, ou fin de *live()*) la méthode ***end()*** est appelée.

Lancer un agent

Pour lancer un agent à partir d'un autre agent il faut en plus de l'instancier le faire enregistrer dans le contexte sociale (tables de groupes et rôles) gérer par le kernel en utilisant la méthode ***launchAgent()***.

Tuer un agent

Un agent peut aussi être tuer en utilisant la méthode ***killAgent()***. Seul le lanceur de l'agent, ou lui même, peut le détruire.

b) Communications

Les agents peuvent envoyer des messages (classe ***Message*** et classes filles *XMLMessage, ActMessage, ACLMessage*...) à d'autres agents grâce à leur identifiant unique de type ***AgentAddress***.
Les agents communiquent de façon asynchrone par le biais des primitives :

- *sendMessage(AgentAddress other, Message m)*
- *BroadcastMessage(Groupe g, Role r, Message m)*
- *BroadcastMessage(Community c, Groupe g, Role r, Message m)*

Pour accéder à ses message un agent utilise :

- *isMessageBoxEmpty()*
- *nextMessage()*

c) gestion des groupes et rôles

	Actions	Requêtes
Groupes	createGroup(..), leaveGroup(..)	getMyGroups(..), getExistingGroups(..), getAvailableCommunities(..), isCommunity(..), isGroup(..)
Rôles	requestRole(..), leaveRole(..)	getMyRoles(..), getExistingRoles(..), getAgentsWithRole(..), getAgentWithRole(..), isRole(..)

II) Manipuler Madkit : exécuter, observer et interagir avec un SMA.

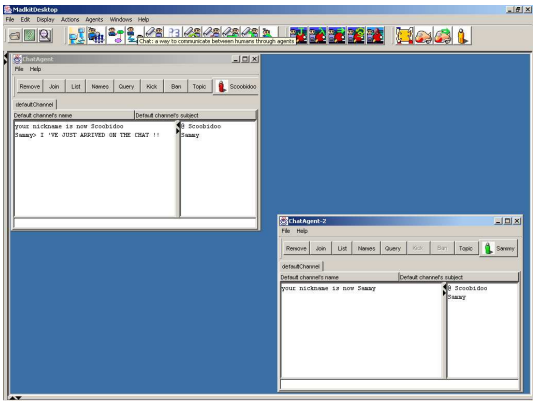
Objectifs : Comprendre la plate-forme multi agents Madkit et les concepts sociaux des SMA par le jeu. L'étudiant est amené à manipuler les concepts de groupe, de rôle, d'agent, et de protocole d'interaction. La manipulation se fait via Madkit

1) Créez une équipe !

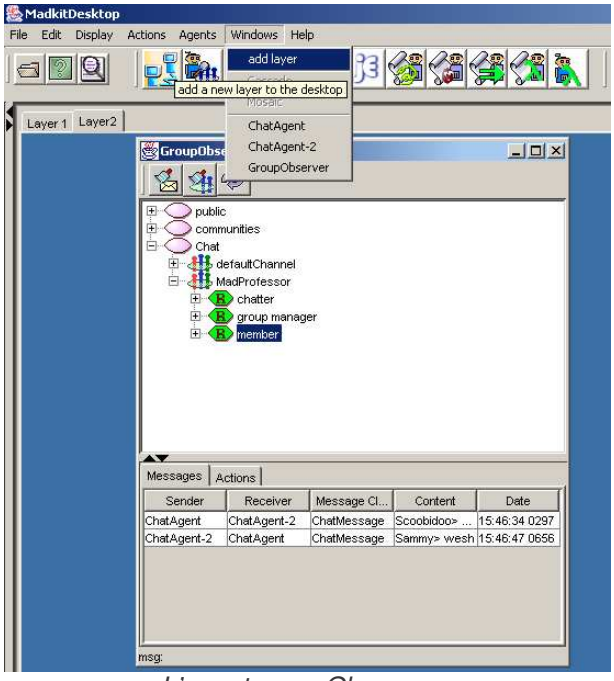
- Les *équipes* sont composées de deux paires d'*étudiants* désignées.
- Les équipes choisissent un nom comme « madProfessor » ou « choKKo».

2) Lancez Madkit et des agents !

- Chaque binôme exécute Madkit.
- Chaque étudiant
  - exécute un *agent Chat*,
  - lui donner un nom.



Le bureau de Madkit avec deux agents Chat



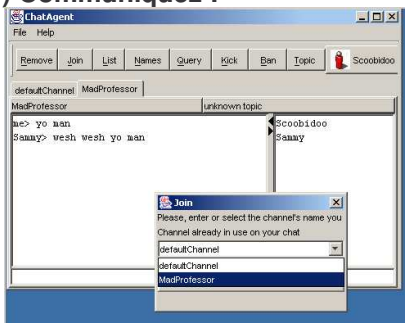
L'agent groupObserver

3) Observez les l'organisation et les agents !

On aimerait bien observer l'organisation de la *communauté Chat*.

- Pour éviter une surcharge de l'affichage, on décide de créer un deuxième panneau de visualisation avec la fonction "add layer" du menu "Windows".
- Chaque binôme lance un agent *groupObserver*, spécialiste des observations des groupes, rôles, messages et actions.
- Chacun ordonne à son agent de joindre un groupe de discussion au nom de son équipe.
- On peut observer que le groupe a bien été créé.

4) Communiquez !



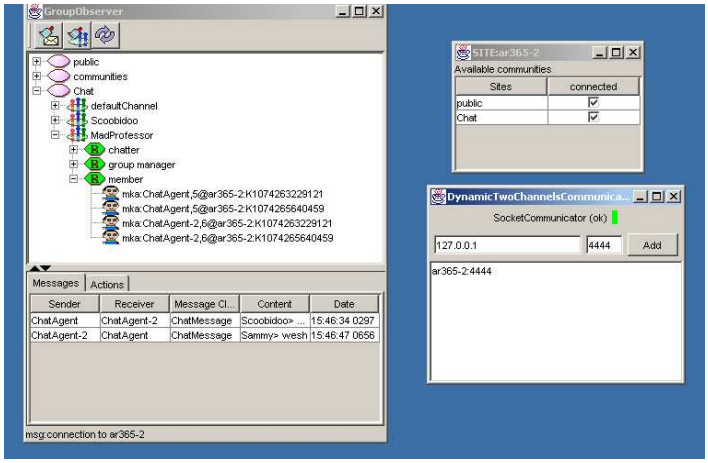
Joindre un agent à un groupe

- On peut envoyer des messages à partir de l'agent Chat,
- Observer ce qu'il se passe dans son groupe, dans le groupe public, et avec l'agent observateur.

5) Communiquez à distance !

On désire maintenant disposer d'un réseau de communication pour interagir avec les autres binômes et notamment celui de son équipe.

- Chaque binôme va lancer un agent *Communicator* afin de créer un réseau de plates-formes Madkit. Indiquez simplement au Communicator le nom de la machine du prof pour que toutes les plates-formes puissent communiquer.
- Vous pouvez observer la nouvelle composition de votre groupe, et y apercevoir les agents du deuxième binôme de votre équipe.
- Vous pouvez également observer les groupes de discussion des autres équipes.
- Vous n'avez pas encore le droit d'ordonner à votre agent Chat de les rejoindre (le prof à lui aussi un agent GroupObserver actif et pourra repérer les tricheurs).



Une nouvelle vue des organisations de la communauté Chat après l'interconnexion effectuée par l'agent Communicator.

6) Mettez vos casquettes de courtiers : que la meilleure équipe gagne !

Chaque étudiant dispose de 100 euros et d'un ensemble d'items qu'il peut vendre (téléphones portables, parapluies troués, pin's parlant…). Chacun peut décider de jouer le rôle d'*acheteur* ou/et de *vendeur* d'un certain nombre d'objets. Pour garantir l'anonymat des acheteurs et des vendeurs chaque requête d'achat ou demande de publication se fait dans le groupe *DESS_IHM* gérer par un agent du prof qui joue le rôle *maître du jeu*. Le rôle du *maître du jeu* est d'assister le prof à contrôler le respect des règles du jeu. Il centralise l'information sur les enchères et les parieurs et retransmet l'état des enchères à tous les agents de la communauté *Marché*.

Vente :

On choisit un produit, une quantité et un prix unitaire on publie auprès du maître du jeu et on lui confit les items sous forme de lot.
Si un lot intéresse :

- Personne : le vendeur doit alors baisser le prix.
- Plus d'une personne : le vendeur doit alors augmenter le prix.
- Une personne : alors le maître du jeu assure le bon échange du lot et de l'argent.

Achat :

Si une enchère intéresse un acheteur il peut faire une mise sur l'enchère au maître du jeu.

- Il peut récupérer sa mise lorsque l'enchère est finie.
- Si il remporte l'enchère il reçoit le lot en jeu.

III) Exercices

Exercice 1. Premier agent : l'agent "HelloWorld"

Objectifs : a) Maîtriser le cycle de vie d'un agent
b) Compiler les sources d'un agent
c) Lancer un agent dans l'environnement MadKit

a) Spécification d'un agent

Maintenant que nous sommes familiers des concepts ayant trait au cycle de vie d'un agent, nous allons créer notre premier agent.

- Prenons d'abord soin d'ouvrir notre éditeur Java préféré.

- Importons le package **madkit.kernel** :
  import madkit.kernel. ;*

- Créons une classe qui hérite de la classe **Agent** (un agent thread) :
  *public class HelloWorldAgent extends **Agent**{*

- A l'**activation** de l'agent : annoncer la naissance de l'agent

```
public void activate(){
        println("Hello : vive la vie !");
}
```

- Faisons maintenant **vivre** notre agent :

```
public void live(){
        for (int n=0 ; n<3 ; n++){
                if (n==0) println("4 pattes : je me déplace !");
                if (n==1) println("2 pattes : vive la course !");
                if (n==2) println("3 pattes : une canne...");
                pause(1000);
        }
}
```

- Hélas chaque cycle de vie à une **fin**... :

```
public void end(){
        println("Je me meurs, arghhhh..");
}
}
```

b) Compilation

- Compilons les sources avec la commande **javac** suivante où le chemin d'installation de MadKit est ***%MADKIT_DIRECTORY%*** :
  - ***javac** -classpath **%MADKIT_DIRECTORY%**\libs\madkit\madkit.jar\ *.java*

- Créons un package (jarpack) avec la commande **jar** et **copions** le fichier .jar dans le répertoire **autoload** de MadKit :
  - ***jar** cf helloworld.jar *.class*
  - ***copy** helloworld.jar %MADKIT_DIRECTORY%**autoload***

c) Lancement de l'agent

Nous pouvons maintenant lancer l'agent que nous avons spécifié :

- Lançons MadKit.
- Dans l'arborescence, ouvrons autoload puis HelloWorld.jar.
- Lançons l'agent HelloWorldAgent.

Exercice 2. Première organisation : les agents pongistes

Objectifs : a) Maîtriser les concepts organisationnels de MadKits (Groupe, Rôle)
b) Utiliser la communication entre agents sur une seule plate-forme
c) Utiliser la communication entre agents sur deux seule plate-formes, et la documentation de l'API

a) groupe ping-pong et rôle joueur

- importez tous d'abbord les packages suivant :
  import madkit.kernel.;*
  import madkit.lib.messages.;*

- Créez ensuite une classe PingPong qui hérite de la classe Agent.

- Ajoutez un attribut pour identifier **l'addresse** de votre partenaire :
  AgentAddredd *partenaire = null;*

- A **l'activation** de l'agent :
  - vérifier l'existence du groupe ping-pong...
    println("Agent Ping-Pong activé");
    priintln("je cherche un groupe ping-pong...");
    if (isGroup("ping-pong"))
    println("Yeah ! Je joins le groupe");
  - ...s'il n'existe pas, créer le groupe...
    else{
    println("Pas de groupe ping-pong. J'en crée un !");
    createGroup(true, "ping-pong", null, null);
    }
  - ...puis prendre le rôle joueur :
    requestRole("ping-pong", "joueur", null);

- Pendant la **vie** de l'agent :
  - ...Chercher la présence d'un autre agent ayant un rôle joueur dans le groupe ping-pong...
    println("A la recherche d'un partenaire...");
    do{
    exitImmediatlyOnKill();//être sûr d'arrêter le thread
    pause(100);
    AgentAddress[] lesJoueurs = getAgentsWithRole("ping-pong", "joueur");
    for (int i=0 ; i<lesJoueurs.length ; i++){
    AgentAddress agent = lesJoueurs[i];
    if (!agent.equals(getAddress()))
    partenaire=agent;
    }
    }
    while (partenaire == null);
    println("J'ai trouvé un partenaire : "+partenaire);
  - ...puis si je ne suis pas créateur du groupe (rôle 'group manager' par défaut)...
    boolean createur=false;
    String[] mesRoles = getMyRoles("ping-pong");
    for (int i=0;i<mesRoles.length;i++)
    if (mesRoles[i]=="group manager")
    createur=true;

b) communication locale

- ...envoyer la balle (un message) à mon partenaire...
  if(! createur)
  sendMessage(partenaire, new StringMessage("balle"));

- ...enfin, sur 5 tours, attendre la réception de la balle et la renvoyer :
  for (int i=5;i>0;i--){
  Message m=waitNextMessage();
  println("Yep ! c'est mon tour. ("+i+")");
  pause(1000);
  sendMessage(partenaire, (StringMessage)m);
  }

- Vous pouvez maintenant :
  - compiler votre fichier source
  - créer un jarpack
  - copier le jarpack dans le répertoire autoload de MadKit
  - lancer MadKit
  - lancer deux agents PingPong

c) communication distante

Nous allons maintenant utiliser le concept de communauté pour faire communiquer nos pongistes d'une plate-forme MadKit à une autre. Le concept de communauté vient du concept de groupe appliqué aux noyaux MadKit. ainsi les groupes, rôles et agents appartenant à une communauté ouverte (l'accès peut être privé) sont visibles de façon transparente pour tous les agents situés sur un réseaux de noyaux connectés.

- Copiez votre fichier source PingPong.java et renommez la copie en SuperPingPong.java !
- Ouvrez le fichier SuperPingPong.java et renommer la classe PingPong en SuperPingPong.
- Ouvrez la documentation de l'API : %MADKIT_DIRECTORY%\doc\api\index.html .
- Regarder la classe Agent.
- Avec l'aide de la documentation modifiez les primitives organisationnelles (isGroup, createGroup, requestRole, getAgentsWithRole, getMyRoles) que vous avez utilisées dans votre fichier source par les celles prenant comme attribut supplémentaire un nom de communauté. Choisissez 'jeux' comme nom de communauté.
- Compilez SuperPingPong.java, créez votre jarpack et copiez le dans le répertoire %MADKIT_DIRECTORY%\autoload.
- Avec l'un de vos camarades et sur des machines distinctes, lancez chacun une plate-forme MadKit.
- Lancez ensuite l'agent de service communicator (cet agent permet deconnecter des plate-formes MadKit ensembles) :
  - Entrez l'addresse IP ou le nom de la machine de votre partenaire.
  - Ajoutez l'IP.
  - Vos plate-formes sont désormais connectées!!!
  - (Vous pouvez voir une communauté 'public')

Lancez chacun un agent SuperPingPong (une communauté "jeux" est apparue).

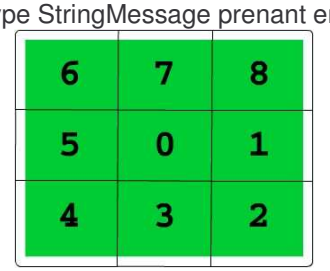
Exercice 3. Tournois de ping-pong

Objectifs : Créer une l'interface graphique

Le but de l'exercice est de donner à l'utilisateur de l'agent un moyen d'interagir avec celui-ci pour renvoyer la balle.

Les règles du jeu sont les suivantes :

- un agent peut envoyer la balle dans 9 positions de la table de l'adversaire
  (utiliser un simple message de type StringMessage prenant en argument un chiffre de 0 à 8)



- le renvoie de balle est effectué par l'utilisateur (un événement click de préférence)
- l'utilisateur a un temps de réponse de 300 ms (*pause(300)* dans la boucle *while* de la méthode *life()*).
- La destination de la balle est assuré par l'agent (pas de stratégie élaborée : un simple random suffit).
- Créer le type de composant qui vous semble le plus adéquat (Java3D, boutons, impact de la balle, affichage d'un score…)
- Dans la méthode **initGUI()** de l'agent initialiser votre composant et activé le avec **setGUI(monComposant)**.

Questions relevées

On souhaite améliorer l'organisation multi-agent en ajoutant un agent pour gérer un tournois de ping-pong (lancer une partie, tirage au sort…).

- Proposez les agents, groupes, rôles et protocoles nécessaire à la mise en oeuvre d'un tournois. Utilisez des diagrammes de séquence UML pour représenter les protocoles d'interaction.
- Décrivez, de préférence à l'aide de diagramme de classe, les IHM nécessaires au contrôle de l'organisation. On peut, également, considérer une IHM comme un agent et décrire les interactions humain-IHM-agent avec des diagrammes de séquence.

On ne veut pas forcément d'implémentation, mais s'il reste du temps une implémentation collective sera envisagée à partir des idées les plus originale