Guia de programació de Sistemes Multiagent en JADE 3.3

David Isern Alarcón
david.isern@urv.net

David Sánchez Ruenes
david.sanchez@urv.net

ÍNDEX

1 JADE

JADE[1] (*Java Agent Development Enviroment*) és una eina de programació que conté un conjunt de llibreries escrites en JAVA per al desenvolupament de Sistemes Multi-Agent. A més de proporcionar les funcions de manegament d'agents a baix nivell i les interfícies que faciliten la programació, també ens presenta un entorn d'execució on els agents podran executar-se i interactuar.

En les següents seccions es tractaran els diferents aspectes del manegament, execució i programació de Sistemes Multiagent mitjaçant JADE versió 3.3.

1.1 Paquets

JADE està composat per una sèrie de paquets escrits en JAVA. Els principals són els següents:

- `jade.core`: és el nucli del sistema. Proporciona la classe `jade.core.Agent` que és imprescindible per a la creació de SMA. També conté el paquet `jade.core.behaviour` que inclou les classes de comportaments suportats.
- `jade.lang`: conté les classes específiques per suportar un llenguatge de comunicació entre agents. Està format pel paquet `jade.lang.acl`.
- `jade.content`: conté les classes específiques per la definició i utilització d'ontologies.
- `jade.domain`: conté totes les classes per representar la plataforma d'agents i els models del domini, tals com entitats per a l'administració d'agents, llenguatges i ontologies (AMS, DF, ACC ...).
- `jade.proto`: paquet que conté els protocols d'interacció entre agents FIPA.
- `jade.tools`: trobem algunes eines que faciliten el desenvolupament d'aplicacions i l'administració de la plataforma:
    - *Remote Management*: Agent (RMA): actua com una consola gràfica per a l'administració i control del sistema.
    - *Dummy Agent*: és una eina de monitorització i depuració, composta per una interfície gràfica i un agent JADE. Permet crear missatges ACL i enviar-los a altres agents, podent visualitzar-los.
    - *Introspector Agent*: és una eina que permet la monitorització del cicle de vida d'un agent.
    - *Sniffer*: agent que intercepta missatges ACL i els visualitza.
    - *SocketProxyAgent*: agent que actua com un *gateway* bidireccional entre la plataforma JADE i una connexió TCP/IP.

[1] Última versió de JADE: 3.3. Plana web: http://jade.tilab.com/

1.2 Components principals

Per JADE, els agents resideixen en un entorn predefinit anomenat plataforma. Cada plataforma està dividida en contenidors i cadascun d'ells podrà contenir agents (veure Fig. 1). Els contenidors poden entendre's com dominis d'agents. La plataforma s'engega en una màquina determinada, mentre que els agents podran executar-se en qualsevol estació.

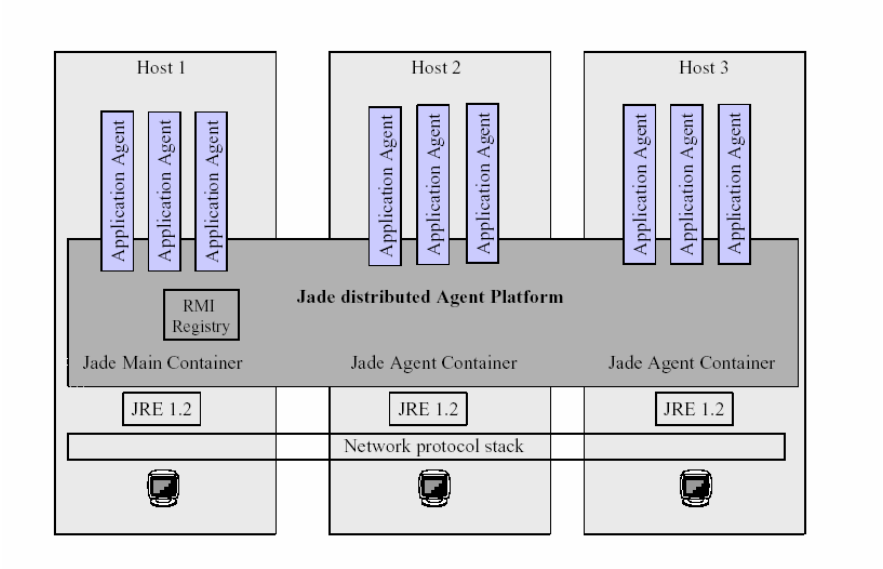

Fig. 1. Plataforma JADE distribuïda en diversos contenidors

Tal com es defineix a la FIPA, en cada plataforma s'inicien dos agents que tenen unes funcions vitals per al funcionament del sistema:

- *Domain Facilitator* (DF): pot ser definit com un servei de "pàgines grogues", és a dir, contindrà tots els serveis que proporcionen els agents que hi estan donats d'alta.
- *Agent Management System* (AMS): es podria considerar com un servei de "pàgines blanques", on es registren les adreces dels agents que s'han donat d'alta.

Gràcies a aquests serveis, es proporciona molta flexibilitat i transparència. D'aquesta manera, si un agent A vol enviar un missatge a un agent B que desenvolupa la tasca T, l'agent A preguntarà al DF quin o quins agents poden fer-se càrrec de la tasca T. Quan es vulgui conèixer l'adreça d'una agent determinat, preguntarà a l'AMS. Per tal que aquest sistema funcioni correctament caldrà que, durant el procés d'inicialització de l'agent, aquest informi al DF de quin o quins serveis proporciona i quin tipus d'agent és (per tal de poder identificar-lo); en fer-ho, l'AMS li donarà una adreça física (depenent d'on s'executi), per tal que altres es puguin posar en contacte amb ell.

1.3 Administració i eines principals

JADE ofereix una interfície gràfica per a l'administració de la plataforma, així com eines per tal de facilitar la depuració i testeig de les aplicacions basades en ell.

Per poder iniciar una sessió i crear els agents, haurem d'escriure la següent comanda[2]:

```
java jade.Boot [opcions] [llista d'agents]
```

Amb això haurem creat la plataforma i un contenidor principal (`Main-Container`) en l'ordinador local, en el que es carregaran els agents definits a la llista a més dels agents bàsics comuns a totes les plataformes (bàsicament, el DF i AMS).

Per tal de poder afegir nous agents a posteriori (un cop la plataforma ja ha estat creada), caldrà definir nous contenidors que s'afegiran a la plataforma principal amb la següent comanda.

```
java jade.Boot –container [opcions] [llista d'agents]
```

Aquesta instrucció suposa que hi ha una plataforma executant-se a l'ordinador local. Altrament, donarà error.

A banda d'aquest model d'execució bàsic, es proporcionen una sèrie de paràmetres dins del camp [opcions] que permeten, entre altres coses, l'execució de plataformes i contenidors remotes (comunicats amb una xarxa amb direccions IP). Les principals opcions són (per més detalls, consultar la guia de l'administrador de JADE):

- "`-host @IP`": especifica el nom (IP) de l'ordinador en el qual s'haurà de crear la plataforma, el contenidor i la llista d'agents. És molt útil quan, per exemple, volem afegir agents mitjançant un nou contenidor a una plataforma que s'executa en una altra màquina.
- "`-port #`": JADE sempre utilitza un port per defecte per fer totes les connexions. Si no s'indica res, es farà servir aquest port. No obstant, amb aquesta opció ens permet definir un port específic. En aquest cas, totes les connexions sobre la mateixa plataforma caldrà haver-les definit sobre el mateix número.
- "`-gui`" : incloent aquesta opció, s'executarà una interfície gràfica que ens permetrà seguir l'execució dels agents i realitzar tasques de debug (més detalls a continuació).

[2] Important: per tal d'evitar problemes, per executar totes les instruccions descrites caldrà haver introduït al CLASSPATH totes les llibreries (*.jar) de JADE (directori jade/lib de la distribució). També caldrà anar en compte amb els *firewalls*, doncs les connexions s'estableixen en ports definits ad-hoc.

Pel que fa a la llista d'agents, serà una cadena de text on cadascun d'ells estarà separat per espais en blanc i seguirà el següent format:

```
<nomAgent>:ClasseJava [parametres]
```

El nom de l'agent haurà de ser un identificador únic per a la instància concreta de l'agent dins de la plataforma actual. La classe java haurà de ser el seu nom sense extensió i amb la ruta corresponent en cas de tractar-se d'un paquet. La llista de paràmetres que, opcionalment, podem passar a l'agent seguirà el mateix format que els paràmetres inicials indicats per qualsevol programa Java.

Quan engeguem l'entorn gràfic, tal com es mostra a la Fig. 2, tot i que no s'inicialitzi cap agent propi, tal com s'ha indicat anteriorment, es creen uns quants automàticament:

- L'agent RMA serà l'encarregat de controlar la interfície.
- El DF, on se subscriuran els serveis dels agents.
- L'AMS on es guardaran les adreces dels agents.

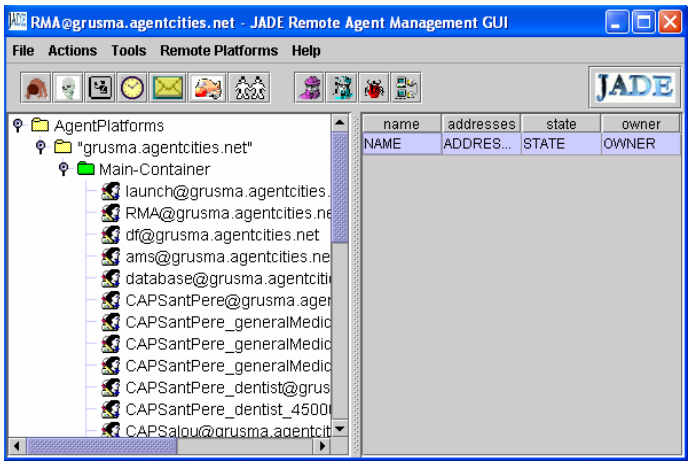

Fig. 2. Interfície gràfica de JADE

A més de la llista d'agents que hi ha al contenidor i la informació sobre el que hem seleccionat, disposem d'una sèrie d'opcions (botons o menú tools) que ens facilitaran el manegament i testeig dels agents:

- Afegir un nou agent: haurem d'indicar el nom, la classe que l'implementa i el contenidor al que pertany.
- Eliminar els agents seleccionats.
- Suspendre els agents seleccionats.
- Continuar els agents seleccionats.
- Enviar un missatge als agents seleccionats: haurem d'omplir una finestra (veure Fig. 3) amb els camps corresponents: qui l'envia, qui el rep, protocol, acte comunicatiu, llenguatge, ontologia, contingut, etc. Per més detalls sobre cóm crear missatges, consulteu les properes seccions.

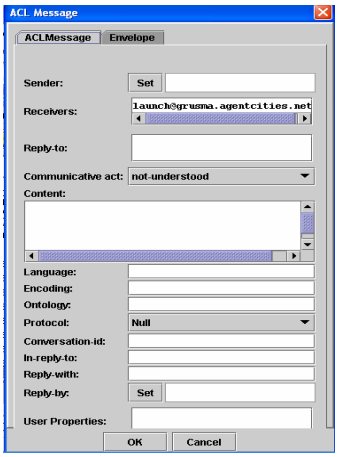

Fig. 3. Finestra per enviar missatges

- Moure els agents seleccionats a un altre contenidor o plataforma.
- Clonar un agent creant-ne un d'igual.
- Executar agent sniffer: és una aplicació creada per rastrejar l'intercanvi de missatges entre els agents JADE. En activar-lo, s'obre una finestra (mostrada a la Fig. 4) on podem observar l'evolució temporal del pas de missatges. Si seleccionem alguna de les fletxes que simbolitzen aquest intercanvi, se'ns mostrarà el missatge amb tots els seus paràmetres. A més podrem seleccionar quins agents volem monitoritzar i guardar els missatges a disc.

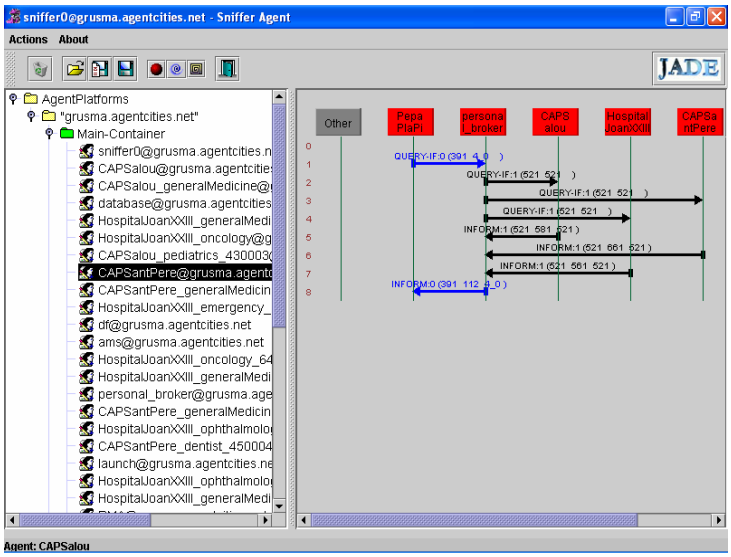

Fig. 4. Finestra de l'agent Sniffer

- Obrir Dummy-Agent: aquest permet enviar missatges ACL entre els agents, rebre'ls i inspeccionar-los i llegir i salvar-los a un fitxer (veure Fig. 5).

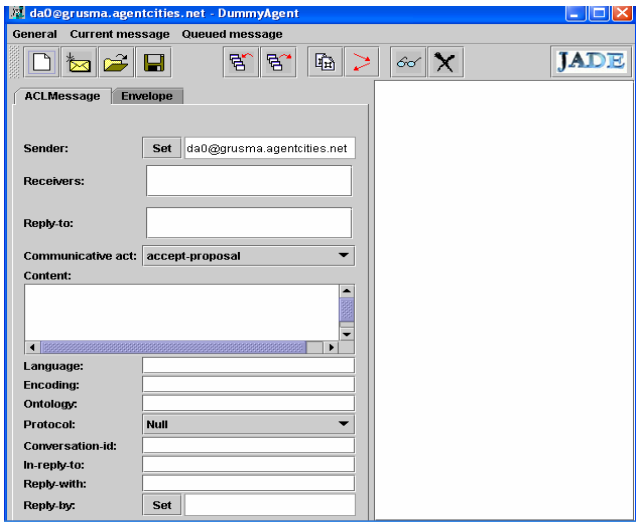

Fig. 5. Finestra de l'agent Dummy

- Engegar l'Intropector Agent que ens permetrà monitoritzar i controlar el cicle de vida de l'agent (estat d'execució, missatges enviats i rebuts) i analitzar els seus comportaments (Behaviours), tal com es mostra a la Fig. 6.

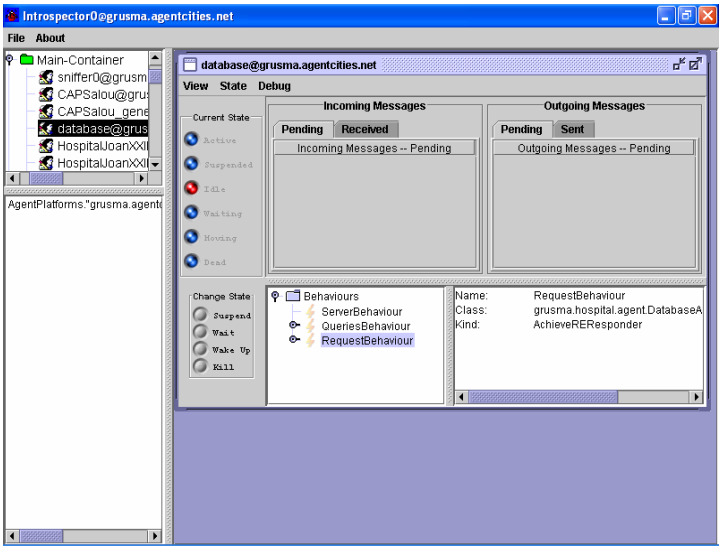

Fig. 6. Finestra de l'agent Introspector

- Afegir una nova plataforma.
- A més, també disposem de la possibilitat d'activar la interfície de l'agent DF (des del menú tools), on podrem observar els agents que s'han registrat i quin serveis ofereixen, tal com es veu a la Fig. 7.

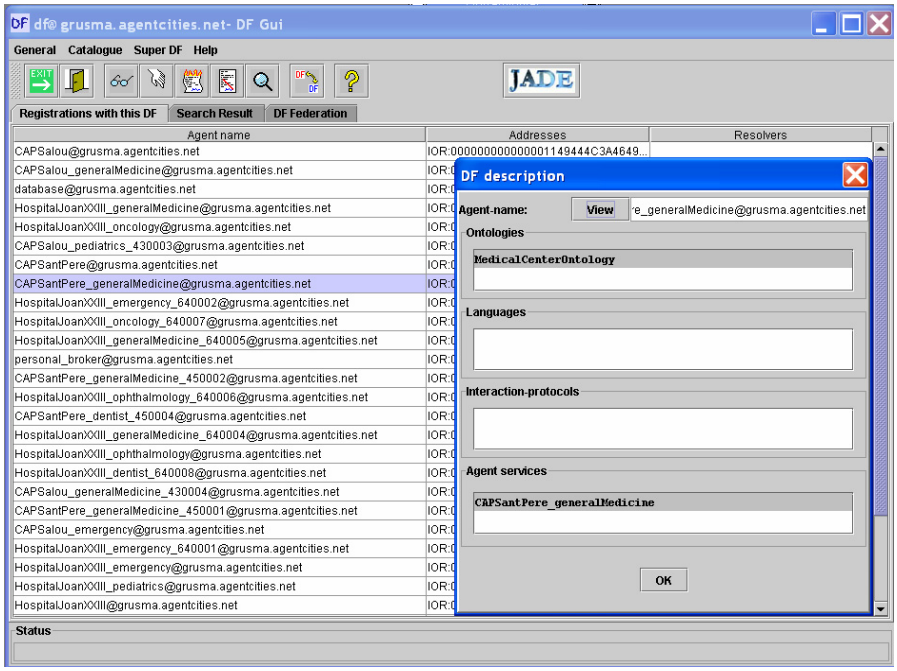

Fig. 7. Interfície gràfica del DF

2 Implementació de Sistemes Multiagent en JADE

El primer pas per tal de crear un Sistema Multiagent en JADE es la definició de l'esquelet bàsic de cadascun dels agents que voldrem definir al nostre sistema en funció del modelatge que hàgim realitzat prèviament.

Des del punt de vista de la programació, un agent es comporta com un procés multifil. El comportament de cadascun d'aquests fils d'execució (que s'utilizaran per modelar cadascuna de les interaccions de l'agent), es defineixen a priori com a "comportaments" i s'inicialitzen durant el procés de carrega de l'agent a la plataforma. D'aquest forma, un cop carregat l'agent, aquest començarà a executar cadascun dels fils d'execució (d'ara en endavant "comportaments") que s'hagin inicialitzat controlats mitjançant l'scheduler inclòs a la plataforma.

Un detall que cal tenir en compte a l'hora de treballar amb JADE són les estructures de dades complexes (llista, vector, etc). Degut a que JADE està pensat per poder-se executar a dispositius mòbils amb recursos limitats (mitjançant el mòdul LEAP que treballa sobre J2ME), els mètodes que proporciona JADE no poden utilitzar estructures de dades dinàmiques. Aquest fet ha provocat que classes complexes disponibles en Java estàndard però que no estan suportades a l'especificació J2ME, hagin estat substituïdes per les equivalents en les llibreries LEAP. Per aquest motiu, moltes de les estructures disponibles a la llibreria `java.util` han estat canviades per les la nova llibreria `jade.util.leap`.

Aquest canvi afecta sobretot a les llistes que són molt utilitzades en el desenvolupament d'agents per crear i llegir els missatges, retornar llistes de resultats, etc. Així, la `java.util.list` i els mètodes que la utilitzen han estat substituïts per del nou paquet `jade.util.leap.list`. No obstant, tant per al cas de les llista com per les altres estructures, el nom i funcionalitat de tots els mètodes clàssics de Java es mantenen idèntics tot i que, internament, tinguin una altra implementació adaptada a l'especificació LEAP.

2.1 Inicialització i finalització d'agents

Entrant més en detall, tot agent JADE haurà de fer sevir (extendre) la classe `Agent` que es troba dins del paquet `jade.core`. Aquesta classe ens proporciona l'estructura bàsica (mètodes) que ha de tenir un agent i la seva forma d'executar-se, de forma que nosaltres només haurem d'anar omplint l'esquelet segons les característiques que volem que tingui l'agent. Els dos mètodes inicials que caldrà que implementem i que seran comuns a tots els agents són:

- `setup`. És el mètode que s'executarà a l'inici (durant la càrrega de l'agent a la plataforma) i és en el que es faran les inicialitzacions pertinents per tal que l'agent es comenci a realitzar les accions per les quals ha estat dissenyat. En general, caldrà que en aquest mètode es realitzi el procés de registre al DF amb el servei que es voldrà proporcionar (el registre al AMS es fa automàticament) i definir almenys un comportament que és el que es començarà a executar just després d'acabar aquesta fase d'inicialització.

- `takeDown`. De forma complementària a l'anterior, aquest mètode s'executa quan l'agent finalitza (de forma controlada, no degut a un error). El seu objectiu es realitzar les accions pertinents per tal que la resta del sistema es continuï executant correctament. Típicament, caldrà que ens desregistrem al DF per tal d'indicar que el servei que oferia l'agent ja no està disponible. Novament, el desregistre a l'AMS es fa automàticament.

```
import jade.core.*;

public class NomAgent extends Agent {

        protected void setup() {
        //inicialitzacions
        }

        protected void takeDown() {
        //finalitzacions
        }
}
```

Pel sol fet d'haver creat una nova classe que estengui `Agent`, se l'hi haurà assignat un identificador únic dins de tot el sistema que farem servir per poder referir-nos a l'agent. Aquest identificador es pot consultar amb el mètode de la pròpia classe `Agent`: `getAID()`. És convenient guardar aquest informació a una variable global (e.g. `myAgent`) doncs ens farà falta per cridar a moltes accions. El nom concret amb que l'agent s'inicialitza (definit pel dissenyador) també es pot consultar amb el mètode `getLocalName()` o `getName()`.

Típicament, un agent estarà dissenyat per realitzar una determinada funció que pot ser sol·licitada per altres agents. Per tal que aquesta funcionalitat estigui publicitada i pugui ser accedida per altres gents, cal que l'agent es registri al DF indicant-la (actua com un servei de pàgines grogues). Tal com s'ha indicat, aquest registre es convenient realitzar-lo durant el procés d'inicialització (setup), tot i que pot ser modificat més endavant. El registre al AMS (servei de pàgines blanques) que permet obtenir les adreces físiques d'execució de cada agent per tal de poder comunicar-se es realitza automàticament just al final del mètode `setup`.

Per tal de realitzar aquest procés de registre, es farà servir la classe `DFAgentDescription` que representa la petició de registre de l'agent al DF, la classe `ServiceDescription` que utilitzarem per definir cadascun dels serveis que prorporcionarà l'agent i la classe `Property` en la que inclourem les característiques associades a cada servei, incloses al paquet `jade.domain.FIPAAgentManagement.`, a més de la classe `DFService`, que ens permet fer peticions al DF, inclosa al paquet `jade.domain`. A continuació es proporciona el codi complet de registre d'un agent que simula un restaurant italià situat a Tarragona. La funció definida s'hauria de cridar en algun moment del procés d'inicialització (`setup`).

```
import jade.domain.FIPAAgentManagement.*;
import jade.domain.*;

void RegistrarDF() {

  //Petició de registre al DF
  DFAgentDescription dfd = new DFAgentDescription();
  //Servei que es proporciona
  ServiceDescription sd = new ServiceDescription();
  //Característiques bàsiques del servei: nom, tipus i organització
  sd.setName("Restaurant");
  sd.setType("Italia");
  sd.setOwnership("GruSMA");
  //Propietat concreta del servei
  Property p = new Property();
  //Nom de la propietat
  p.setName("Ciutat");
  //Valor de la propietat
  p.setValue(new String("Tarragona"));
  //Es poden afegir tantes propietats (parelles clau-valor) com es vul-
gui
  sd.addProperties(p);
  //Es poden afegir tants serveis a la petició de registre com el vul-
gui
  dfd.addServices(sd);
  //Indicar el nom de l'agent que el sol·licita: l'identificador
  dfd.setName(getAID());

  try {
       //Petició de registre
       DFService.register(this, dfd);
       System.out.println("["+getLocalName()+"]:"+" S'HA REGISTRAT AL
DF");
  } catch (jade.domain.FIPAException e) {
       System.out.println("["+getLocalName()+"]:"+" No s'ha pogut re-
gistrar en el DF");
       doDelete();
  }
}
```

De forma complementaria, quan finalitzem l'execució de l'agent, caldrà indicar a la resta del sistema que el seu(s) servei(s) ja no està disponible, amb el corresponent desregistre al DF (a l'AMS es fa automàticament). Aquest procés es pot realitzar de la forma següent (codi que s'hauria de cridar durant en algun moment al mètode `takeDown`)

```
import jade.domain.*;

void DeRegistrarDF() {
  try {
       //Demanem desregistrar-nos
       DFService.deregister(this);
       System.out.println("["+getLocalName()+"]:"+"EL AGENTE HA ESTAT
ELIMINAT DEL DF");
  } catch (Exception e) {
       System.out.println("["+getLocalName()+"]:"+"No s'ha pogut elimi-
nar del DF");
  }
}
```

Un cop els agents s'han registrat al DF/AMS és possible realitzar cerques filtrades per tal de trobar els identificadors (AID) els agents desitjats o que ens poden donar el servei requerit. Les cerques a l'AMS es fan segons el nom que se'ls ha assignat a la plataforma, doncs és un servei de noms (pàgines blanques) mentre que les cerques al DF es fan segons les funcionalitats dels doncs és un registre de serveis (pàgines grogues). Tant en un cas com en l'altre es farà servir el mètode `search` contingut a les classes `AMSService` i `DFService` del paquet `jade.domain`. A continuació s'inclou un exemple de cerca al DF per a l'agent registrat a l'exemple anterior:

```
//Llista on guardarem els agents que compleixen el criteri de cerca
//Cal fer servir una llista especial del paquet jade.util.leap
jade.util.leap.List restaurants = new jade.util.leap.ArrayList();

//De forma semblant al procés de registre, definim les característiques
//dels agents que volem buscar
DFAgentDescription dfd = new DFAgentDescription();
ServiceDescription sd = new ServiceDescription();
//Volem tots els "restaurants" que estiguin a la "Ciutat=Tarragona"
sd.setName("Restaurant");
Property p = new Property();
//Nom de la propietat
p.setName("Ciutat");
//Valor de la propietat
p.setValue(new String("Tarragona"));
//Es poden afegir tantes propietats (parelles clau-valor) com es vulgui
sd.addProperties(p);
dfd.addServices(sd);

try
{
        SearchConstraints c = new SearchConstraints();
        //Fem la cerca indican l'agent que la i els criteris de selecció
        DFAgentDescription[] result = DFService.search(myAgent, dfd, c);
        for (int i=0; i<result.length; i++)
        {
                //Guardem els AIDs dels agents trobats a la llista
                restaurants.add(result.[i].getName());
        }

catch(Exception e)
{
        System.err("Error buscant restaurants");
}
```

A més d'aquestes accions bàsiques, tal com veurem més endavant, haurem de definir i registrar altres característiques de l'agent per tal de poder realitzar processos comunicatius amb altres entitats. Aquests aspectes es descriuran a l'apartat 2.3.

A banda del registre de serveis, una altra possibilitat que resulta interessant és el pas de paràmetres d'inicialització a l'agent. De la mateixa forma que el `main` d'una classe java, quan es crea un agent es pot passar una llista de paràmetres inicials indicats just a continuació del nom de l'agent en la comanda d'inicialització. La lectura d'aquest paràmetres és convenient realitzar-la durant la fase d'inicialització (`setup`) mitjançant el mètode `getArguments` que proporciona per a aquest cas la classe `Agent` i que retorna una taula d'`Object` amb el nombre i valor corresponent dels paràmetres.

2.2 Definició de comportaments (*behaviours*)

Tal com s'ha dit anteriorment, l'execució d'un agent es composa d'una sèrie de comportaments que s'executen de forma concurrent. Típicament, aquests fils d'execució estaran implementats per realitzar cadascuna de les tasques que requereix el servei que modela l'agent (e.g. demanar dades, realitzar càlculs, enviar resultats, etc). Tota la funcionalitat de l'agent ha d'estar inclosa a aquests comportaments i, per tant, es tracta dels components més importants (juntament amb el procés de comunicació) d'un agent. Inicialment (durant el procés de setup), caldrà que hàgim creat almenys un d'aquest comportaments que es qui començarà a realitzar les accions per les quals l'agent ha estat dissenyat. No obstant, nous comportaments es podem anar creant/destruint dinàmicament en altres parts del codi.

Per tal de facilitar la implementació d'aquests comportaments, JADE ens proporciona una sèrie de classes amb la funcionalitat bàsica de diferents tipus de comportaments. Tots ells estan inclosos al paquet `jade.core.behaviours`. De forma semblant al cas de la classe `Agent`, totes les classes que defineixen comportaments comparteixen una sèrie de mètodes bàsics que caldrà implementar per obtenir la funcionalitat desitjada. Aquests mètodes bàsics són:

- `Action`: és el cos del comportament, doncs haurà de contenir les accions a realitzar. Segons el tipus de comportament escollit, aquest mètode s'executarà una sola vegada o de forma cíclica.
- `Reset`: torna el comportament al seu estat inicial d'execució.
- `Block/Restart`: permet parar durant un cert temps i renegar un comportament. El mètode `Block` és especialment útil doncs està sincronitzat amb la cua de recepció de missatges, de forma que es pugui realitzar un processament asíncron, tal i com es descriurà més endavant.
- `Done`: permet comprovar si un comportament ha acabat d'executar-se.

Tot i que existeixen un nombre considerable de comportaments diferents, aquests estan organitzats segons una jerarquia d'herència de classes de forma que els més complexos són extensions dels més senzills. Així, podem trobar dos tipus de comportaments bàsics dels quals en deriven la resta:

- `SimpleBehaviour`: és un comportament simple que s'executa sense interrupcions (atòmic). D'aquest en deriven els següents:
    - `OneShotBehaviour`: executa el contingut del mètode `action` un sol cop. El mètode `done` sempre retorna `true`.
    - `CyclicBehaviour`: executa el contingut del mètode `action` cíclicament. El mètode `done` sempre retorna `false`.
    - `TickerBehaviour`: executa el contingut del mètode `action` cada un cert temps indicat durant la instanciació del comportament.
    - `WakerBehaviour`: comença a executar el mètode `action` quan ha passat un cert temps indicat durant la seva instancicació.

    - `MsgReceiver`: s'executa quan es detecta la recepció d'un missatge que compleixi unes certes condicions (més detalls a la secció de comunicació entre agents).
    - `SimpleAchieveREInitiatior/Responder`: implementa una versió simplificada del protocol de comunicació `FipaRequest` (més detalls a la secció corresponent).

- `CompositeBehaviour`: es tracta d'un comportament abstracte que permet definir altres comportaments "fills", és a dir, podem associar un o més *subbehaviours* a un comportament complex, que seran gestionats per un *scheduler* independent per a l'agent. Per tal de crear/eliminar aquest nous comportaments (que al seu torn poden ser simples o compostos), s'incorporen dos nous mètodes als descrits anteriorment: `addSubBehaviour` i `removeSubBehaviour`. En aquest cas, es disposa de nous mètodes per implementar la funcionalitat del comportament: `onStart` i `onEnd`. En el primer afegirem els comportaments subordinats. Quan acabi l'execució d'aquest mètode es llençaran els comportaments definits amb el planificador específic. El segon mètode s'executarà quan tots els comportaments hagin acabat per realitzar el processament dels resultats i les finalitzacions pertinents. Segons la forma en què s'executen els comportaments fills (planificador) caldrà utilitzar un d'aquest dos comportaments específics:
    - `ParallelBehaviour`: si s'executen concurrentment de forma paral·lela.
    - `SerialBehaviour`: si ho fan seguint un cert ordre establert (no concurrent). D'aquest últim en deriven dos nous comportaments:
        - `FSMBehaviour`: executa els fills seguint una màquina d'estats finits. D'aquest en deriven altres comportaments complexos utilitzats per implementar les protocols de comunicació (més detalls a la secció corresponent).
        - `SequentialBehaviour`: els fills s'executen un darrera l'altre de forma seqüencial.

De forma anàloga al procés de definició d'un agent, la implementació de comportaments es farà extenent la classe de JADE que proporciona el comportament que millor s'adapta als requeriments d'execució que hàgim definit. Un cop implementat el comportament, podrem crear instàncies concretes d'aquest comportaments i afegir-los al fil d'execució de l'agent. Per fer-ho, caldrà utilitzar el mètode `addBehaviour` de la classe `Agent`, que rep com a paràmetre la instància del comportament concret el qual es començarà a executar immediatament. Tal com s'ha comentat, caldrà fer servir aquesta funció en el procés d'inicialització (`setup`) de l'agent per crear el(s) comportament(s) inicial(s), tot i que també es pot executar des del cos d'execució d'altres comportaments per tal d'afegir-ne de nous dinàmicament.

Per il·lustrar el procés d'implementació, instanciació i execució de comportaments, en el següent exemple definirem un comportament cíclic a base de comportaments simples d'execució puntual.

```
import jade.core.*;
import jade.behaviours.*;

//Farem un comportament ciclic a base de comportaments puntuals
class myCyclicBehaviour extends SimpleBehaviour{

        //Guardarem una instancia de l'agent que ens ha creat per
        //poder crear nous comportaments
        Agent myAgent;

        //Constructor del comportament
        public myCyclicBehaviour (Agent agent) {
           myAgent = agent;
        }

        //El codi del comportament
        public void action () {
           //Imprimim l'hora
           System.out.println("Executant: "+System.currentTimeMillis());
           //Creem una nova instancia del comportament
           myAgent.addBehaviour(new myCyclicBehaviour(myAgent));
        }
}
```

2.3 Comunicació entre agents

La comunicació entre agents es l'aspecte fonamental dels Sistemes Multiagent. Es per això, que l'intercanvi de missatges entre agents en JADE està molt elaborat permetent la definició de complexos protocols de comunicació. Tot i això, per tal que diversos agents heterogenis (e.g. programats per diferents autors) puguin interoperar fàcilment, el procés de comunicació ha d'estar definit sobre una sèrie de components estàndards. Concretament, els aspectes més importants sobre els quals s'ha d'aconseguir el consens són:

- Codificació dels missatge. Concretament, el llenguatge de representació emprat per representar les dades i el propi missatge de forma serialitzada per tal d'enviar-lo per la xarxa. Existeixen diversos estàndards per aquest propòsit que es descriuran a la secció 2.3.4.
- Estructura del missatge. Per tal de permetre una comunicació elaborada, els missatges, a més del contingut a enviar, han de contenir informació extra sobre l'emissor, receptor, tipus de conversa, llenguatge de codificació, etc. Per tal cosa, existeix l'estàndard FIPA-ACL per a la composició de missatges, tal com veurem en aquest apartat.
- Seqüència d'intercanvi de missatges. Quan s'implementen processos de comunicació complexos on intervenen diferents interlocutors i diverses fases de comunicació, cal que tots segueixin el mateix protocol de comunicació. Per tal propòsit, la FIPA ha definit nombrosos protocols estàndards per a Sistemes Multiagent que permeten implementar de forma senzilla comunicacions complexes com negociacions, subscripcions, subhastes, etc. Els protocols bàsics de comunicació emprats típicament es tractaran en aquest capítol.
- Terminologia utilitzada per compondre el contingut dels missatges. Això es defineix mitjançant ontologies, tema que es tractarà a l'apartat 2.3.4.

2.3.1 Missatges

El missatge és el nucli de la comunicació doncs, no només defineix la informació a intercanviar sinó una sèrie de dades extres que permeten identificar els interloculors i realitzar filtratges.

En JADE, la seva estructura segueix l'especificació FIPA-ACL i consisteix en la classe `ACLMessage` que es troba dins el paquet `jade.lang.acl.ACLMessage`. A més del constructor, disposa d'una sèrie de mètodes per llegir/escriure (`setXXX`, `getXXX`) cadascun dels camps que suporta. Així, un missatge ACL està format pels següents camps principals:

- `Preformative`: permet la definició de l'acte comunicatiu i tipus de missatge (e.g. petició, resposta, resultat correcte, error, etc.). Resulta fonamental per seguir el fil d'intercanvi de missatge quan s'utilitzen protocols de comunicació elaborats, tal com es veurà més endavant. Les principals *preformatives* disponibles (com a constants dins del paquet `jade.lang.acl`) són:
    - `Request/Request-when/Request-whenever`: s'utilitzen per demanar la realització d'accions.
    - `Inform/Inform-if/Inform-ref`: indiquen l'enviament dels resultats d'una acció o pregunta sol·licitada.
    - `Query-if/Query-ref`: sol·liciten la contestació d'una pregunta.
    - `CFP`: és una crida a l'enviament de propostes (Call For Proposals) en resposta a la petició de realització d'una determinada acció.
    - `Propose`: envia una proposta de realització d'una acció en resposta a una petició anterior.
    - `Accept-proposal/Reject-proposal`: accepten/refusen una proposta enviada.
    - `Agree/Refuse`: confirmen/refusen una acció sol·licitada.
    - `Not-understood`: indica que no s'ha esta capaç d'entendre el contingut del missatge al qual es demana una resposta.
    - `Cancel`: indica la suspensió de la realització d'una acció demanda anteriorment.
    - `Failure`: indica que l'acció demanada ha fallat i no es pot enviar un resultat.
    - `Subscribe`: l'emissor sol·licita ser informat en el moment que es compleixin unes certes condicions.

  A més d'aquestes, n'existeixen altres que es poden emprar quan es fan servir altres protocols complexos: `Propagate`, `Proxy`, `Unknown`, `Confirm`, `Disconfirm`.
- `Sender`: indica l'emissor del missatge, s'espera que sigui del tipus AID (indentificador d'agent).
- `Receiver`: indica el receptor o receptors del missatge. Novament es tracta d'identificadors d'agent tipus AID. En aquest cas, en lloc dels mètodes `setReceiver` i `getReceiver` disposa dels mètodes `addReceiver` i `getAllReceiver`, doncs permet indicar que el missatge s'enviï a més d'un receptor.

- `ReplyTo`: permet indicar a qui ha d'anar destinat el següent missatge a enviar dins de la conversa. De la mateixa manera, permet indicar diversos receptor mitjançant els mètodes `addReplyTo` i `getAllReplyTo`.
- `Language`: indica el llenguatge de representació emprat per representar el contingut del missatge. Ha de ser el nom (definit com a constant) d'algun dels llenguatge suportats per JADE.
- `Encoding`: indica si s'utiliza alguna codificació especial.
- `Ontology`: indica el nom de l'ontologia emprada per composar el contingut del missatge.
- `Content`: és el contingut del missatge. Normalment estarà expressat mitjançant els objectes definits en una ontologia. Es poden fer servir mètodes alternatius com `set/getContentObject` per enviar objectes serialitzats of `get/setByteSequenceObject` per en contingut com una taula de bytes.
- `Protocol`: indica el protocol de comunicació al qual pertany el present missatge.
- `ConversationId`: permet especificar un identificador a la conversa per tal de realitzar filtratges.
- `ReplyWith`: identificador de missatge que s'utilitza per seguir els diferents passos de la conversa.
- `InReplyTo`: identifica una acció passada a la qual aquest missatge és la resposta.
- `ReplyByDate`: marca de *timeout*: data/hora de caducitat del missatge

No tots els camps descrits són obligatoris, però en tot cas, resulta convenient inicialitzar el major nombre d'ells per tal d'evitar problemes.

Arribat a aquest punt, voldrem enviar el missatge i rebre'l en l'altre interlocutor. L'enviament es fa mitjançant el mètode `send` de la classe `Agent` de forma transparent a través de la plataforma i utilitzant els identificadors AID indicats que són manegats de forma automàtica per l'AMS. L'enviament/recepció es fa de manera asíncrona amb cua de missatges en la recepció, de forma que no es perdi cap missatge. Així, els missatges s'envien al destinatari immediatament i queden guardats en una cua que serà buidada a mesura que el destinatari executi el mètode `receive` de la classe `Agent`. Aquest mètode no és bloquejant, de forma que caldrà assegurar-se si s'ha llegit algun missatge dels que havia disponibles a la cua o bé estava buida i ha retornat `null`.

No obstant, si volem sincronitzar-nos amb la recepció d'un missatge, és a dir, no executar cap codi fins que no estem segurs d'haver rebut un missatge, podem realitzar una espera passiva (sense consumir CPU) mitjançant el mètode `block` que està disponible en tots els comportament i que està sincronitzat amb la cua de recepció de missatges. D'aquesta forma, si executem `receive` i no obtenim cap missatge, podem executar `block` per bloquejar el comportament es qual restarà inactiu. Quan es rebi un missatge a la cua, tots els comportaments de l'agent destinatari que estiguin bloquejats es reiniciaran (tornaran a executar el mètode `action`) de forma que, executant de nou el mètode `receive`, ja puguem obtenir el missatge.

Un exemple d'enviament d'un missatge podria ser el següent:

```
//Instanciació del missatge
ACLMessage msg = new ACLMessage(ACLMessage.INFORM);

//Afegim un receptor (suposem que el seu AID ja l'havíem trobat abans
msg.addReceiver(AID_receiver);
//Ens posem a nosaltres com a emissor
msg.setSender(getAID());
//Indiquem el contingut (String sense estructura)
msg.setContent("Hello world");

//Enviem el missatge
Send(msg);
```

El procés de recepció sincronitzat del missatge quedaria així:

```
public void action() {

   //Recepció no bloquejant d'un missatge
   ACLMessage msg = myAgent.receive();
   //Comprovació del resultat

   if (msg != null) {
        //processament del missatge
        ....
   }else
   {
        //Bloquejar el comportament fins rebre un missatge a la cua
        block();
   }
}
```

Un cop ja sabem com enviar i rebre missatges, el següent pas consisteix en ser capaços de filtrar els possibles missatges que poden arribar a la cua de forma que, en un moment donat, accedim al missatge desitjat. Degut a que les converses entre agents poden ser força complexes, un sol agent pot tenir diverses conversacions obertes alhora entre varis interlocutors i, per tant, seria convenient poder discriminar quin dels possibles missatges rebuts és el que volem processar. Per poder assolir aquesta tasca, JADE ens proporciona la classe `MessageTemplate` del paquet `jade.lang.acl`, sobre la qual es poden definir diverses regles de filtratge sobre els missatges que després es podran aplicar sobre el mètode `receive`. Aquest filtratge es pot realitzar sobre la majoria dels camps addicionals de l'objecte missatge descrits anteriorment. Així per exemple, si tenim dues converses obertes alhora, podem ver servir el camp `ConversationId` per discriminar entre una i una altra. És per aquesta raó que resulta convenient incialitzar el major nombre de camps de l'objecte missatge possible (encara que no siguin obligatoris), doncs ens permeten contextualitzar el missatge concret dins d'una conversa.

A continuació incloem un exemple en el que es discriminarà el missatge mitjançant la preformative utilitzada. Concretament esperarem rebre missatges de tipus `Proposal` (doncs suposadament prèviament havíem enviat un `CFP`).

```
public void action () {
   //Definim un filtre en funció de la preformative
   MessageTemplate mt = MessageTemplate.MatchPerformative (ACLMessage.
PROPOSAL);
   //Rebem els missatges que compleixen el filtre
   ACLMessage msg = myAgent.receive(mt);
   if (msg != null) {
        //Tractem el missatge rebut
        ...
   }else
   {
        block();
   }
}
```

En cas de voler definir filtres més complexos, podem fer servir operadors lògics (AND, OR, NOT) per concatenar diverses plantilles, tal com s'indica al següent exemple (establim restriccions en quant a protocol, ontologia i preformative):

```
MessageTemplate mt = MessageTemplate.and( MessageTemplate.MatchProtocol(
"FIPA-Request" ), MessageTemplate.MatchOntology( "test_ontology" ) );

//Concatenació de templates mitjançant l'operador AND
MessageTemplate mt2 = MessageTemplate.and( mt, MessageTempla-
te.MatchPerformative( ACLMessage.REQUEST ) );
```

2.3.2 Llenguatges de comunicació

Tal com s'ha dit anteriorment, els missatges tenen un camp que els indica el llenguatge en el que està representat el seu contingut (que a la seva banda, tal com es veurà més endavant, pot estar expressat mitjançant una ontologia).

En aquest camp es pot indicar qualsevol llenguatge que es vulgui sempre que les dues bandes de la conversa siguin capaces d'interpretar-lo. Per tal de facilitar aquest aspecte, JADE proporciona de forma nativa o mitjançant plug-ins[3] una sèrie de llenguatges estàndards de comunicació en forma de codecs (llibreries de codificació-decodificació):

- `SL`: un llenguatge especialment creat per intercanviar missatges entre agents. Destaca per la seva simplicitat i eficiència però no es pot considerar un estàndard de caràcter general i, per tant, és difícilment interperable amb altres aplicacions.
- `XML`: el llenguatge de representació estàndard per excel·lència. És interoperable amb una gran quantitat d'aplicacions (com navegadors d'internet, editors d'ontologies i parsers en general) i és fàcilment interpretable al contenir marques semàntiques sobre el contingut. La contrapartida és que resulta molt més estricte que l'SL (compte amb els caràcters estranys dins d'un missatge) i afegeix un gran overhead al missatge en forma de tags.
- `RDF`: una extensió del XML especialment dissenyat per descriure recursos. Té els mateixos avantatges i inconvenients que XML.

[3] Els plug-ins oficials que expandeixen la funcionalitat de JADE es poden trobar a la pròpia pàgina de JADE (http://jade.tilab.com/), a la secció de descàrregues.

- `Bit-efficient`: utilitza una representació binària de les dades per tal d'aconseguir la millor eficiència. No obstant, no és estàndard i tampoc es pot interpretar directament (no es pot "llegir").

La utilització de qualsevol d'aquests llenguatges és transparent al programador en quant, un cop especificat el que es farà servir, automàticament, els missatges s'escriuran i llegiran seguint les regles del llenguatge concret. És important destacar que el llenguatge de representació és independent de la forma en com es defineix el contingut del missatge, ja sigui mitjançant l'enviament d'objectes serialitzats, emprant strings que posteriorment es parsejaran o, tal com es veurà més endavant, a través d'una ontologia definida per tal propòsit.

Sempre que es faci servir algun llenguatge estàndard amb un codec associat, caldrà haver-lo registrat durant el procés d'inicialització (`setup`) de l'agent mitjançant el mètode `registerLanguage`, que rep una instància del codec associat al llenguatge (que estarà dins del paquet `jade.content.lang`). Per fer-ho, caldrà fer servir una instància de l'objecte `ContentManager` del paquet `jade.content` que es pot obtenir de la classe Agent mitjançant el mètode `getContentManager`, tal com es mostra a continuació.

```
//Importem les llibreries associades als llenguatges
import jade.content.lang.Codec;
//Els llenguatges SL i XML estan suportats nativament per JADE
import jade.content.lang.sl.SLCodec;
import jade.content.lang.xml.*;
//El llenguatge RDF requereix un plug-in extern
import it.unipr.aot.rdf.RDFCodec;

protected void setup()
{
   try
   {
        //Recuperem una instància del codec SL
        Codec codec = new new SLCodec();
        //Recuperem el nom estàndard del llenguatge
        String codec_name = codec.getName();
       //Registrem el llenguatge SL
       getContentManager().registerLanguage( codec, codec_name );

        //Recuperem una instància del codec XML
        Codec codec = new XMLCodec();
        //Recuperem el nom estàndard del llenguatge
        String codec_name = codec.getName();
       //Registrem el llenguatge XML
       getContentManager().registerLanguage( codec, codec_name );

        //Recuperem una instància del codec RDF
        Codec codec = new RDFCodec();
        //Recuperem el nom estàndard del llenguatge
        String codec_name = RDFCodec.NAME;
       //Registrem el llenguatge RDF
       getContentManager().registerLanguage( codec, codec_name );

   }
   catch( Exception e ){
       System.err.println( "Error registrant els llenguatges );}
}
```

De forma semblant, també caldrà registrar una ontologia, en cas de fer-ne servir alguna, tal com es descriurà més endavant.

Un cop registrat el llenguatge (o llenguatges, doncs se'n poden registrar tants com es vulgui durant el procés de `setup`), només caldrà indicar a cada missatge, el llenguatge concret que es farà servir per construint-lo, indicant el seu nom:

```
ACLMessage msg = new ACLMessage( ACLMessage.CFP );
msg.setLanguage((new SLCodec()).getName());
```

2.3.3 Protocols de comunicació

Tot i que amb les funcions i classes introduïdes fins ara ja podríem ser capaços d'implementar tot tipus de converses complexes (combinant diferents tipus de comportaments, recepcions asíncrones i plantilles de missatges), per tal de facilitar les coses i aconseguir una bona interoperbilitat amb altres Sistemes Multiagent, JADE ofereix una sèrie de protocols estàndards (definits per la FIPA) per tal de modelar els tipus bàsics de negociació: preguntes, peticions d'accions, negociacions, subhastes, etc.

La implementació dels protocols és en forma de behaviours complexos que pengen de la classe `FSMBehaviour` i proporcionen una sèrie d'accions específiques per tractar els diferents passos de la conversa. El fet de ser comportaments pensats específicament per a establir converses fa que el tractament de missatges (recepció/enviament) i les fases de la comunicació es facin de forma transparent al programador.

Els protocols implementats que proporciona JADE en la versió actual són els següents (continguts al paquet `jade.proto`):

- `FIPA_REQUEST`: serveix per demanar peticions d'accions.
- `FIPA_QUERY`: protocol de pregunta-resposta.
- `FIPA_CONTRACT_NET`: s'utilitza per negociar la realització d'accions entre diversos agents.
- `FIPA_BROKERING`: interacció entre agents a través d'un intermediari.
- `FIPA_AUCTION_DUTCH`: per descobrir el preu de mercat d'un ítem fent propostes altes i anant reduint-les (subhasta Holandesa).
- `FIPA_AUCTION_ENGLISH`: Per descobrir el preu de mercat d'un ítem fent propostes baixes i anant augmentant-les (subhasta Anglesa).
- `FIPA_ITERATED_CONTRACT_NET`: Com el `Contract-Net` però amb diverses iteracions (per millorar les propostes).
- `FIPA_PROPOSE`: Per fer una proposta de realització d'una acció.
- `FIPA_RECRUITING`: Per demanar a altres agents per fer d'intermediaris.
- `FIPA_REQUEST_WHEN`: Per demanar a un altre agent la realització d'una acció en el moment en que unes precondicions es compleixen.
- `FIPA_SUBSCRIBE`: Per demanar que se t'informi quan hi hagi un determinant canvi en l'estat d'un objecte

Per a cada conversa, JADE distingeix dos rols: el que la inicia (*initiator*) i el que la segueix (*responder*) i cada agent haurà d'estendre un comportament específic proporcionat per JADE en funció del seu paper. Aquests comportaments especials es creen i es maneguen de la mateixa forma que el comportaments explicats en seccions anteriors.

Tal com veurem més endavant, quan fem servir aquestes classes, no caldrà fer crides explícites als mètodes `send` / `receive` doncs en aquest cas, els missatges a enviar es passaran als constructors o es retornaran de les funcions i els missatges rebuts es rebran per paràmetres de les funcions que s'executaran justament quan es rebi el missatge adient. Així, el procés d'enviament i recepció asíncrona es farà de forma transparent al programador que només haurà d'implementar les accions pertinents per tractar cadascun dels casos (tipus de missatges) possibles.

A continuació s'explicaran amb detalls els protocols més comuns: `FIPA-Request`, `FIPA-Query`, `FIPA-Contract-Net`.

2.3.3.1 Protocols FIPA-Query i FIPA-Request

Donat que la implementació dels dos protocols és la mateixa doncs la seva estructura comunicativa és molt semblant, els hem agrupat al mateix apartat.

FIPA-Request

S'utilitza per demanar la realització d'accions. El comportament que tindran els agents quan utilitzen aquesta especificació és el següent (veure Fig. 8):

- L'agent *initiator* envia un missatge de tipus `request` cap a l'agent amb el que vol interactuar (*responder*) per demanar-li la realització d'una acció.
- El receptor del missatge llegeix el contingut del missatge i obté l'acció que se li demana. L'agent estudiarà la petició i determinarà si la pot fer o no. Si la pot realitzar, enviarà un missatge `agree` a l'agent inicial i començarà a realitzar l'acció. Si per alguna raó no pot realitzar-la, enviarà un missatge `refuse` que informi de la raó d'aquesta negació.
- En qualsevol cas, si l'*initiator* no ha formulat bé el missatge i, per tant, no es pot desxifrar correctament, s'enviarà un missatge `not-understood`.
- En cas d'haver enviat un missatge d'`agree`, el *responder* haurà començat a realitzar l'acció. Quan l'acabi, enviarà un missatge de tipus `inform`. Aquest pot ser de dos tipus: `inform-done` que només notifica que l'acció ha estat finalitzada i `inform-ref`, que retorna un resultat. Si l'acció no s'ha finalitzat correctament, s'enviarà un missatge `failure` indicant la raó.

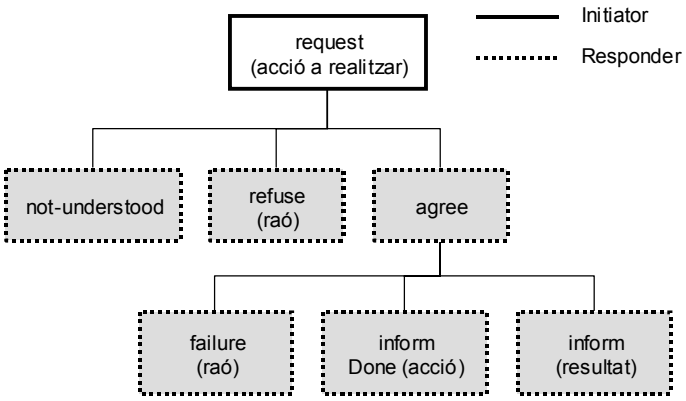

Fig. 8. Protocol FIPA-Request

FIPA-Query

S'utilitza quan es vol demanar (preguntar) algun tipus d'informació. Els passos que se segueixen durant la conversa són els següents (veure Fig. 9):

- L'agent *initiator* formula una pregunta de tipus `query` o `query-ref`.
- L'agent *responder* podrà enviar la informació de la resposta o bé refusar la pregunta.
- Com sempre, si hi ha un error s'enviarà una fallada i si no se sap interpretar el contingut del missatge, un `not-understood`.

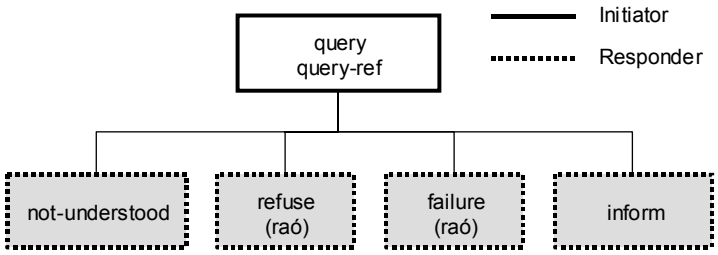

Fig. 9. Protocol FIPA-Query

Implementació homogènia

Tal com s'ha introduït, els protocols de `Query` i `Request` han estat agrupats sota una interfície comuna, tal com es pot veure a la Fig. 10. Per al cas d'un `Request` caldrà utilitzar la seqüència de missatges: `Request-Agree-Inform`, mentre que per a un Query, només serà necessari implementar la parella `Query-Inform`.

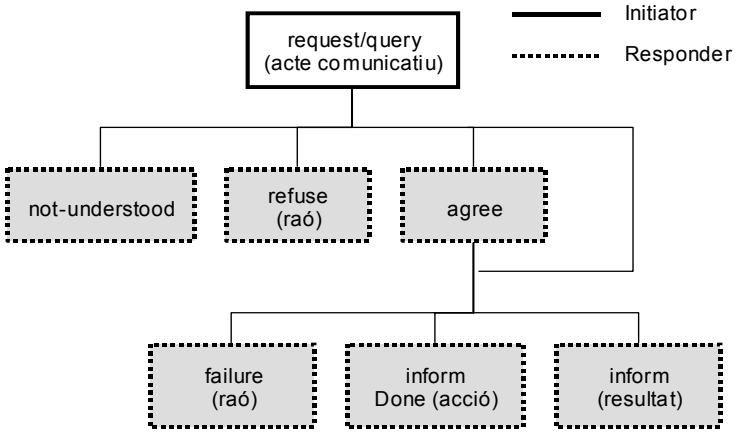

Fig. 10. Definició homogènia de protocols

Les classes que implementen aquests protocols estan disponibles al paquet `jade.proto`:

- `AchieveREInitiator`: fa les vegades d'iniciador de la conversa (realitza la pregunta-petició). S'encarrega de crear el missatge inicial (`Request` o `Query`) i enviar-lo al destí; a continuació esperarà a rebre les respostes.
- `AchieveREResponder`: implementa la banda que respon a les peticions. Si el protocol es de tipus `Request`, caldrà que primer s'enviï una confirmació del missatge (`Agree`) i a continuació el resultat final (`Inform`).

Opcionalment, existeix una parella alternativa de classes que consumeixen menys recursos a costa de reduir la funcionalitat: `SimpleAchieveREInitiator` i `SimpleAchieveREResponder`. La principal restricció afecta al fet de que cada missatge només podrà ser enviat a un receptor.

INITIATOR

El primer pas en la implementació d'algun d'aquests protocols és crear una classe que estengui al comportament `AchieveREInitiator` o `SimpleAchieveREInitiator` i crear un constructor per a ella que rebi una referència a l'agent que l'ha d'executar i el missatge que s'enviarà en el primer pas de la comunicació (`Request` o `Query`). L'enviament d'aquest missatge es farà justament quan s'afegeixi la instància d'aquest comportament a l'agent.

A continuació caldrà implementar els mètodes proporcionats per aquestes interfícies per tal d'aconseguir el comportament desitjat:

- `void handleInform (ACLMessage msg)`: s'executa cada cop que es rep un missatge de tipus `Inform`.
- `void handleNotUnderstood (ACLMessage msg)`: ídem per a missatges de tipus `Not-understood`.

- `void handleRefuse (ACLMessage msg)`: ídem de par missatges de tipus `Refuse`.
- `void handleFailure (ACLMessage msg)`: ídem per a missatges de tipus `Failure`.
- `void handleAgree (ACLMessage msg)`: ídem per a missatges de tipus `Agree`. Aquest mètode només caldrà implementar-lo si volem aconseguir un protocol de tipus `Fipa-Request`, mentre que els mètodes anteriors són comuns al `Request` i al `Query`.

Alternativament, JADE proporciona un parell de mètodes que permet tractar els missatges de forma conjunta (en aquest cas, podríem discriminar manualment segons les preformatives dels missatges):

- `void handleAllResponses (Vector responses)`: rep els missatges esperats en el primer nivell de l'acte comunicatiu: `Agree`, `Refuse` o `Not-understood`.
- `void handleAllResultNotifications (Vector notifications)`: rep els missatges corresponents al segon nivell de la comunicació: `Inform` o `Failure`.

Tenint en compte tot això, l'estructura que caldrà implementar si volem utilitzar un protocol de tipus `FIPA-Query` serà la següent:

```
//Nou comportament que estén la part iniciadora d'un Query
class QueryInitiator extends [Simple]AchieveREInitiator
{
  //El constructor rep una instància de l'agent que l'ha creat
  //i el missatge a enviar inicialment
  public QueryInitiator (Agent myAgent, ACLMessage queryMsg)
  {
       Super (myAgent, queryMsg);
  }

  //Mètodes que caldrà implementar amb les accions pertinents
  protected void handleInform (ACLMessage msg) {...}

  protected void handleNotUnderstood (ACLMessage msg) {...}

  protected void handleRefuse (ACLMessage msg) {...}

  protected void handleFailure (ACLMessage msg) {...}

}
```

En canvi, si el que volem es un protocol de tipus `FIPA-Request` els mètodes que caldrà utilitzar seran els següents (només caldrà afegir un mètode de tractament de l'`Agree`):

```
class RequestInitiator extends [Simple]AchieveREInitiator
{
  public RequestInitiator (Agent myAgent, ACLMessage requestMsg)
  {
       Super (myAgent, requestMsg);
  }
```

```
  protected void handleAgree (ACLMessage msg) {...}

  protected void handleInform (ACLMessage msg) {...}

  protected void handleNotUnderstood (ACLMessage msg) {...}

  protected void handleFailure (ACLMessage msg) {...}

  protected void handleRefuse (ACLMessage msg) {...}
}
```

RESPONDER

Pel que fa a la banda que respon a les peticions, el primer pas serà crear un classe que estengui a `AchiveREReponder` o `SimpleAchieveREReponder`. El constructor d'aquesta classe rebrà un referència a l'agent que l'executa i un `Message-Template` en el que es podran establir els filtres adequats per tal de rebre només els missatges esperats per a aquesta conversa. El comportament es quedarà a l'espera passiva de missatges que compleixin les restriccions just després d'afegir la instància del comportament a l'agent.

Un cop fet això, caldrà implementar el mètodes que tractaran el missatge de petició i crearan les respostes:

- `ACLMessage prepareResponse (ACLMessage peticio)`: s'executa quan es rep un missatge de petició que passa el filtre definit al `MessageTemplate`. Si es vol respondre al `Request`, caldrà que avaluï la petició i respongui retornant un missatge d'`Agree`, `Refuse` o `Not-understood`. Si la petició era un `Query`, es podrà retornar directament la resposta final: `Inform`, `Failure`, `Not-understood` o `Refuse`.
- `ACLMessage prepareResultNotification (ACLMessage request, ACLMessage response)`: només per al cas d'un protocol de tipus `Request` en que s'hagi executat el mètode anterior i s'hagi retornat un missatge d'`Agree`, s'executarà aquest. Rep per paràmetres el missatge de `Request` i la resposta d'`Agree` enviada. Caldrà que executi l'acció demanada i retorni la resposta (`Inform` o `Failure`).

Així, per al cas d'un protocol de tipus `Query`, els mètodes que caldrà implementar seran els següents:

```
class QueryResponder extends [Simple]AchiveREResponder
{
  //El constructor rep l'agent que l'ha creat
  //i una plantilla amb els filtres per als missatges entrants
  public QueryResponder (Agent myAgent, MessageTemplate mt)
  {
       Super (myAgent, mt);
  }

  protected ACLMessage prepareResponse (ACLMessage msg) {...}

}
```

En canvi, si es tracta d'un `Request` l'estructura a implementar serà la següent:

```
class RequestResponder extends [Simple]AchieveREResponder
{

   public RequestResponder (Agent myAgent, MessageTemplate mt)
   {
        Super (myAgent, mt);
   }

   protected ACLMessage prepareResponse(ACLMessage request) {...}

   //Aquest mètode s'executarà si anteriorment s'ha enviat un Agree
   protected ACLMessage prepareResultNotification (ACLMessage request,
   ACLMessage reponse) {...}

}
```

2.3.3.2 Protocol FIPA-Contract-Net

Aquest protocol s'utilitza quan es vol realitzar una negociació entre diversos agents per trobar aquell qui la pot realitzar millor segons els nostres interessos. Els passos que se segueixen són els següents (veure Fig. 11):

- L'agent *initiator* envia una petició de proposta d'acció passant-li unes precondicions a complir.
- L'agent *responder* avaluarà les premisses rebudes i, en funció de quines siguin proposarà realitzar o no l'acció. En el primer cas, enviarà una proposta d'acció amb unes condicions determinades; en el segon, un missatge de refús. Per tal que l'agent inicial sàpiga quan ha acabat de rebre totes les propostes a la seva petició, es disposa d'un *time-out* màxim.
- Si s'ha enviat una proposta, l'agent inicial l'avaluarà, i podrà o no acceptar-la enviant a l'agent *responder* el missatge corresponent.
- Finalment, l'agent *responder* començarà a realitzar l'acció, enviant un missatge de finalització (`inform`), una fallada (`failure`) en cas de produir-se algun error o una possible cancel·lació abans que acabi la tasca (`cancel`).

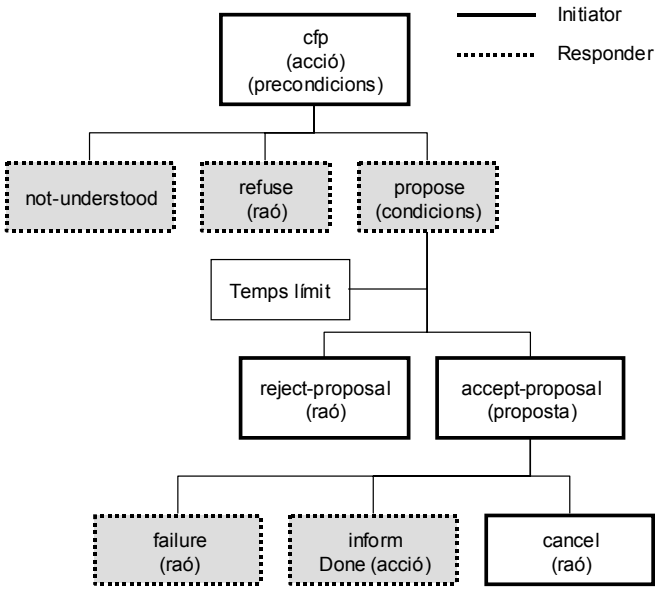

Fig. 11. Protocol FIPA-Contract-Net

Implementació

Al paquet `jade.proto`, trobem les classes `ContractNetInitiator` i `ContractNetResponder` que defineixen les dues bandes de la conversa.

INTIATOR

El primer pas per a implementar el rol que inicia la conversa és crear una classe que estengui a `ContractNetInitiator`. El constructor corresponent rebrà una referència a l'agent que l'executa i el missatge que s'enviarà a l'inici (`CFP`). Aquest missatge s'enviarà just després d'afegir la instància del comportament a l'agent.

Els mètodes que caldrà implementar per tractar la resposta a la petició són:

- `void handlePropose (ACLMessage propose, Vector acceptances)`: s'executa cada cop que es rep un missatge de `Propose`. Cal que s'afegeixi la resposta corresponent a aquest missatge al vector de respostes (indicant si s'accepta o no).
- `void handleNotUnderstood (ACLMessage msg)`: ídem per a missatges `Not-understood`.
- `void handleRefuse (ACLMessage msg)`: ídem per a missatges `Refuse`.

Alternativament, es disposa del següent mètodes que agrupa la funcionalitat de tots els anteriors:

- `void handleAllResponses (Vector responses, Vector results)`: s'executa quan totes les respostes a les peticions s'han rebut. Per a les afirmatives (`Propose`) caldrà que creem una resposta adequada (acceptant-la o no) que serà afegida i retornada al vector del segon paràmetre.

Per tal de manegar els resultats de les accions disposem dels següents mètodes:

- `void handleInform (ACLMessage msg)`: s'executa cada cop que es rep un missatge d'`Inform`.
- `void handleFailure (ACLMessage msg)`: ídem per a missatges de tipus `Failure`.

Alternativament, disposem d'un mètode que agrupa als dos anteriors:

- `void handleAllResultNotifications (Vector result)`: s'executa quan totes les respostes de les accions s'han rebut (`Failure` o `Inform`).

Així, els mètodes que caldrà implementar per a aquest protocols són:

```
//Nova classe que estén el comportament iniciador
class CNInitiator extends ContractNetInitiator
{
   //El constructor rep una referència a l'agent que l'ha creat
   //i el primer missatge (CFP) a enviar
   public CNInitiator (Agent myAgent, ACLMessage CfpMsg)
   {
        Super (myAgent, CfpMsg);
   }

   //Comportaments a implementar per manegar els missatges rebuts
   protected void handlePropose (ACLMessage propose, Vector acceptances)
   {...}

   protected void handleNotUnderstood (ACLMessage msg) {...}

   protected void handleRefuse (ACLMessage msg) {...}

   protected void handleInform (ACLMessage msg) {...}

   protected void handleFailure (ACLMessage msg) {...}

}
```

RESPONDER

El primer pas serà definir la classe que estengui a `ContractNetResponder`. El constructor rebrà una referència a l'agent que l'executa i un `MessageTemplate` amb les restriccions que s'aplicaran als missatges rebuts. El comportament es quedarà a l'espera de missatges que compleixin les restriccions establertes just després la instància del comportament a l'agent.

Els mètodes que caldrà implementar en aquesta banda de la conversa són:

- `ACLMessage prepareResponse (ACLMessage msg)`: s'executa quan es rep un missatge `CFP` i caldrà que retorni la resposta corresponent (`not-understood`, `refuse` o `propose`).
- `ACLMessage prepareResultNotification (ACLMessage cfp, ACLMessage propose, ACLMessage accept)`: s'executa quan es rep un missatge `Accept-proposal`. També rep per paràmetres el CFP inicial i la resposta (`Propose`). Cal que executi l'acció corresponent i retorni el resultat (`Inform` o `Failure`).
- `void handleRejectPorposal (ACLMessage cfp, ACLMessage propose, ACLMessage reject)`: manega la recepció de missatges `Reject-proposal`.

Seguint aquestes indicacions, l'estructura de mètodes a implementar per a aquest protocol és:

```
class CNResponder extends ContractNetResponder
{
  //El constructor rep la instància de l'agent que l'ha creat
  //i la plantilla amb els filtres per als missatges entrants
  public CNResponder (Agent myAgent, MessageTemplate mt)
  {
       Super (myAgent, mt);
  }

  protected ACLMessage prepareResponse (ACLMessage msg) {...}

  protected ACLMessage prepareResultNotification (ACLMessage cfp, ACL-
  Message propose, ACLMessage accept) {...}

  protected void handleRejectProposal (ACLMessage cfp, ACLMessage pro-
  pose, ACLMessage reject) {...}

}
```

2.3.4 Ontologies

Tal com s'ha indicat anteriorment, la definició estructurada de la informació que intercanvien els agents durant la comunicació es fa mitjançant una ontologia. Tot i que aquesta no és imprescindible enviar informació, sí que es convenient per aconseguir bona interoperabilitat i facilitar el processament de les dades (sobretot en sistemes grans i complexos).

Una ontologia permet definir els següents punts:

- *Frames*: objectes sobre els quals es podrà parlar i que, per tant, estaran disponibles per omplir el missatge.
- *Slots*: camps d'informació en cada objecte o *frame*.
- Etiqueta (amb la que s'accedirà), tipus de dades (`String`, `long`, `float`, compost...) que conté cada *slot* i altres característiques com són l'opcionalitat o la cardinalitat de cada element.

Amb tots aquest elements, podrem crear totes les entitats que formaran la nostra ontologia i que es poden classificar en tres grans grups:

- *Conceptes*: defineixen els objectes amb els que tractem (ex: hospital, metge...).
- *Predicats*: defineixen una sintaxi lògica que permet demanar peticions en base a un conjunt de restriccions (ex: "Obtenir pacient amb ID=1111").
- *Accions*: es refereixen als serveis que poden oferir-se els agents (ex: "Donar d'alta un pacient").

Tots els elements relacionats amb les ontologies es poden trobar en els paquets `jade.content` i `jade.domain`. Internament, JADE implementa més ontologies com per exemple, `BasicOntology`, `FIPAAgentManagement`, que també es podran usar en els nostres agents/sistemes per realitzar tasques més avançades com la creació i gestió dinàmica d'agents en temps d'execució.

En general, els passos que cal seguir per crear i utilitzar una ontologia són: el a) creació del *fitxer de definició*, b) implementació de les *classes* associades als conceptes, predicats i accions; c) *instanciació* de l'ontologia des de programa i d) *generació de missatges* que la utilitzin.

2.3.4.1 Fitxer de definició

Una ontologia té un fitxer principal que defineix tots els elements que es poden usar, el format i els paràmetres permesos. Tots els conceptes, predicats i accions estan implementats finalment en llurs classes Java. Com que aquesta feina pot ser força tediosa, hi hagut investigadors que han proporcionat una eina per a facilitar-ne tot el desenvolupament i generació de codi font. Aquesta eina s'anomena BeanGenerator[4] i és un plugin de Protégé[5].

En la Fig. 12 es pot veure una ontologia d'exemple definida usant Protégé i compatible BeanGenerator. Per poder exportar correctament el codi font de JADE, cal incloure un projecte proporcionat per BeanGenerator anomenat `SimpleJADEAbstractOntology`. Aquesta ontologia defineix diferents ítems importants com accions (anomenada `AgentAction`), conceptes (anomenats `Concept`), i predicats (anomentats `Predicate`). Un cop inclòs correctament aquesta ontologia, definirem la nostra ontologia, com per exemple, la classe `Employee` és un concepte, `NASA-MissionSpecialist` és una subclasse de `Employee` (i n'hereta tots els seus atributs), i `is-shuttle` és un predicat. De la mateixa forma, crearíem la jerarquia de classes que necessitéssim pel nostre cas.

[4] El BeanGenerator té diferents versions per a generar el codi font compatible amb diferents versions de Jade. En aquest document s'ha usat la darrera versió disponible, v.3.0 disponible a l'adreça http://acklin.nl/page.php?id=34 i també de forma local a http://www.etse.urv.es/recerca/banzai/toni/MAS/material.html.

[5] Protégé és una eina de manejament de coneixement que permet de definir de forma gràfica ontologies. S'ha usat la darrera versió disponible a dia d'avui, la 3.0 i disponible a http://protege.stanford.edu/index.html.

Internament, cada element definit pot contenir atributs o *slots*. Per exemple, com es mostra a la figura, el concepte `Shuttle` té nou atributs de diferents tipus i cardinalitats. A la Fig. 13 es mostra la finestra de diàleg que permet de configurar cadascun d'aquests atributs. Un atribut pot ser de tipus `any` (objecte qualsevol), `boolean`, `class` (no definit en JADE), `float`, `instance` (usat per a definir instàncies de classes de la nostra jerarquia), `integer`, `string`, `symbol` (no suportat en JADE).

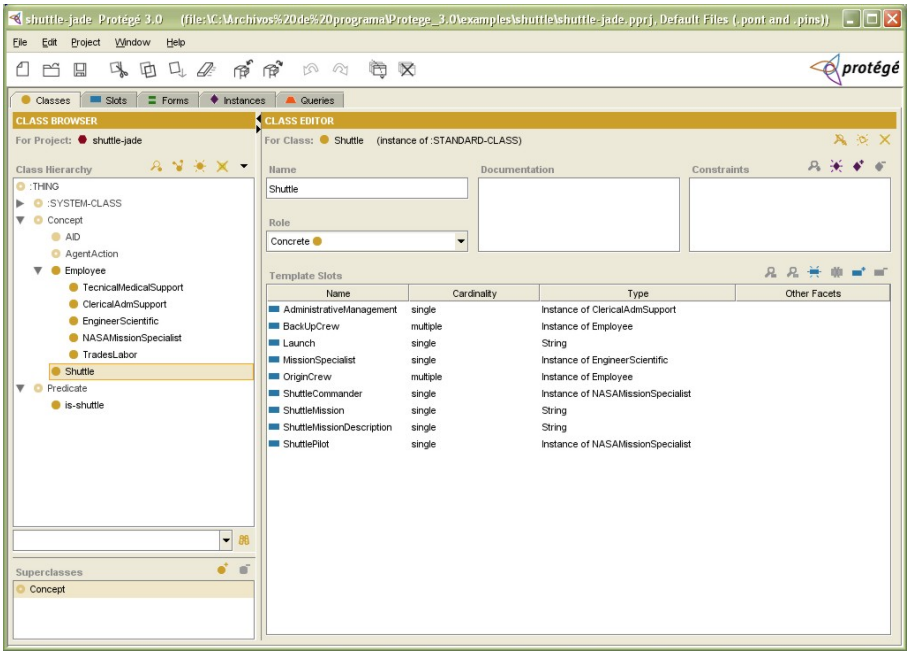

Fig. 12. Captura de pantalla de la ontologia anomenada `ShuttleOntology` usant Protégé 3.0 com a editor.

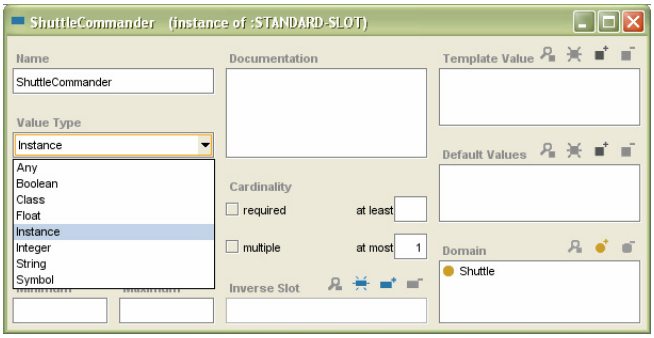

Fig. 13. Diàleg que permet la definició d'un atribut o *slot*. Cal determinar-ne el tipus i la cardinalitat. Opcionalment es pot donar una descripció.

L'ús de BeanGenerator per automatitzar la creació i implementació d'ontologies, permet la creació de l'esquelet i la jerarquia de classes, però després es poden editar tots aquests fitxers per acabar de perfilar alguns petits canvis. Per exemple, s'ha comentat que hi ha uns tipus de dades bàsics suportats per l'editor de Protégé, però no són exactament els mateixos que els implementats per JADE, definits a `jade.content.onto.BasicOntology`: `boolean`, `date`, `byte_sequence`, `float`, `integer`, `string` o qualsevol instància d'un objecte definit a la ontologia.

2.3.4.2 Jerarquia de Classes

Un cop creada la ontologia que usarem en el nostre SMA, es pot generar tot el codi font a través del BeanGenerator. En la Fig. 14 es mostra una execució d'aquesta eina sobre la ontologia de exemple que estem desenvolupant. Com s'ha dit, aquesta generació dependrà de la versió de JADE que usem i per a cadascuna hi ha una versió del BeanGenerator disponible.

Dins el directori seleccionat per a la generació del codi trobarem el conjunt de classes Java que defineixen la nostra ontologia. El fitxer principal té el prefix del domini seguit de la paraula *Ontology,* en el nostre cas, *ShuttleOntology.java*. El fitxer de definició consta de tres parts: inicialitzacions, definicions i especificacions. En la primera part, es determinen les classes a importar, el constructor de la classe i les variables públiques per a permetre l'accés remot. En la segona part, es defineixen tots els elements que s'usaran en la ontologia: classes, atributs, predicats, i accions; només podrem usar el que definim en aquesta part. Finalment, en la tercera part, s'instancia la ontologia pròpiament dita dins l'entorn JADE, afegint cada objecte a l'entorn amb els tipus i cardinalitat definits prèviament.

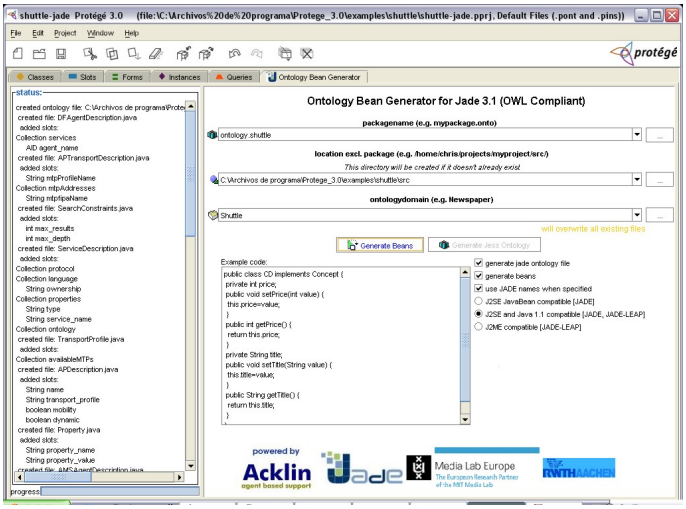

Fig. 14. Generació de codi font per JADE 3.1 (compatible amb 3.3) usant BeanGenerator 3.

Cadascun dels conceptes, accions i predicats definits en el fitxer de definició tenen associada una classe Java. Dins de cada classe trobarem la implementació dels atributs o *slots* que s'han definit en el projecte Protégé. Per cadascun d'ells s'hauran definit una sèrie de mètodes que permeten la seva lectura i escriptura tenint en compte la cardinalitat i tipus. Per exemple, veiem un fragment del concepte *Shuttle* que defineix dos atributs, un de múltiple i un de senzill:

```
package ontology.shuttle;
import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

public class Shuttle implements Concept {

   /**
   * Protege name: BackUpCrew
   */
   private List backUpCrew = new ArrayList();
   public void addBackUpCrew(Employee elem) {
     backUpCrew.add(elem);
   }
   public boolean removeBackUpCrew(Employee elem) {

     boolean result = backUpCrew.remove(elem);
     return result;
   }
   public void clearAllBackUpCrew() {

     backUpCrew.clear();
   }
   public Iterator getAllBackUpCrew() {return backUpCrew.iterator(); }
   public List getBackUpCrew() {return backUpCrew; }
   public void setBackUpCrew(List l) {backUpCrew = l; }

   /**
   * Protege name: MissionSpecialist
   */
   private EngineerScientific missionSpecialist;
   public void setMissionSpecialist(EngineerScientific value) {
    this.missionSpecialist=value;
   }
   public EngineerScientific getMissionSpecialist() {
     return this.missionSpecialist;
   }
...
```

Com es veu en aquest fragment, el que fem es definir una llista d'elements o un element concret segons el cas. Per a cada atribut, es defineixen els mètodes per a consultar (get) i modificar (set) el seu valor, que varien segons sigui una llista o no. Més enllà, en el fitxer de definició es determina la opcionalitat o obligatorietat d'aquests atributs, és a dir, si en un missatge són obligatoris o no. En cas que siguin opcionals, permetrà una major flexibilitat a l'hora d'enviar un missatge, però caldran més comprovacions del contingut en el destí, donat que no se sap *a priori* si hi ha o no un valor associat per l'atribut.

En aquest fragment es veu com la classe Java extèn la classe de JADE `jade.content.Concept`. Les altres classes poden ser `AgentAction` o `Predicate`.

2.3.4.3 Instanciació de la ontologia

De forma anàloga al registre del llenguatge de comunicació, en el procediment de `setup` d'un agent, aquest ha de determinar quina o quines ontologies pot interpretar. Un cop assignades en aquest punt, ja no se'n podran afegir més. Per exemple, si un agent interpretarà missatges d'una ontologia anomenada *ShuttleOntology*, caldria afegir aquesta línia en el mètode setup:

```
getContentManger().registerOntology(ShuttleOntology.getInstance());
```

2.3.4.4 Ús de la ontologia

Un cop ja hem incialitzat l'agent, registrant l'ontologia i el llenguatge de codificació ja estem en disposició de crear missatges que utilitzin aquesta estructura.

A més dels objectes bàsics definits per nosaltres en la ontologia, també podem disposar d'alguns objectes heretats de la ontologia `BasicOntology`, i definits en la classe `SLVocabulary`, que fan referència a certs predicats bàsics que indiquen afirmació, negació, pregunta, etc, com per exemple, `all`, `any`, `iota`, `and`, `done`, `equals`, `exist`, `not`, `forall`, `or`, `result`, `set`, `true`, o `false`, entre d'altres. Aquests predicats, amplien el vocabulari i permeten la composició d'expressions més complexes[6]. Per a mostrar un exemple, veiem com es crearia un pregunta que demana, fent servir l'ontologia creada i els predicats bàsics, tots els elements que compleixen alguna de les condicions donades. Com a resultat, l'agent donarà una llista (set) amb els elements resultats, o una llista buida, si no n'ha trobat cap.

```
private ACLMessage createQueryShuttle(
   jade.util.leap.List receivers,
   String shuttle_wanted) {

   ACLMessage dummy=new ACLMessage(ACLMessage.QUERY_REF );
   dummy.setSender((AID)getAID());
   dummy.setLanguage(this.codec.getName());
   dummy.setOntology(ShuttleOntology.ONTOLOGY_NAME);
   dummy.setProtocol("FIPA-Query");
   dummy.clearAllReceiver();
   for(int i=0; i<receivers.size(); i++)
    dummy.addReceiver((AID)receivers.get(i));

   try {
    AbsConcept abs =
     new AbsConcept(ShuttleOntology.SHUTTLE);
```

[6] Les diferents formes i significats es poden trobar al document *FIPA SL Content Language Specification*, document número SC00008I

```
   abs.set(ShuttleOntology.SHUTTLE_SHUTTLEMISSION,
    shuttle_wanted);
   abs.set(ShuttleOntology.SHUTTLE_SHUTTLEMISSION
    DESCRIPTION, shuttle_wanted);

   AbsPredicate absIsShuttle =
    new AbsPredicate(ShuttleOntology.IS_SHUTTLE);
   absIsShuttle.set(ShuttleOntology.IS_SHUTTLE_ITEM, abs);

   AbsVariable var = new AbsVariable("x", ShuttleOntology.SHUTTLE);

   AbsIRE allPred = new AbsIRE(SLVocabulary.ALL);
   allPred.setVariable(var);
   allPred.setProposition(absIsShuttle);
   this.getContentManager().fillContent(dummy,allPred);

 }catch( Exception e ) {
   System.err.println(getLocalName()+" ERROR: "+e.toString());
 }
 return dummy;
} //endof createQueryShuttle
```

En l'altra banda de la conversa, un cop rebut el missatge, l'agent realitzaria el procés invers per tal d'obtenir tots els objectes que formen el missatge en base a l'ontologia que s'ha fet servir per crear-ne el contingut.

```
protected ACLMessage prepareResponse(ACLMessage query) {
   ACLMessage reply = query.createReply();
   reply.setPerformative(ACLMessage.INFORM);

   try {
        showMessage("Extracting content: "+query.getOntology());
        AbsContentElement            allCE            =            pa-
rent.getContentManager().extractAbsContent(query);
        AbsIRE allPred = (AbsIRE)allCE;

        if(allPred.getTypeName().equals(SLVocabulary.ALL)) {

          showMessage("Extracting  content:  Predicate  ALL  identified
...");
          AbsPredicate pred = (AbsPredicate)allPred.getProposition();
          showMessage("Extracting content: Reading constraints ...");
          showMessage(pred.toString());

          AbsConcept             c             =             (AbsCon-
cept)pred.getAbsTerm(ShuttleOntology.IS_SHUTTLE_ITEM);
          showMessage(c.toString());

          String                          str                          =
c.getString(ShuttleOntology.SHUTTLE_SHUTTLEMISSION);
          showMessage(str);

         //Preparació de la resposta
         //.....................
     } //endif

   } catch (Exception e) {
     e.printStackTrace();
   }
   return reply;
} //endof prepareResponse
```

Com a exemple, veiem com serien els continguts d'un missatge de pregunta i la seva resposta per part d'un altre agent. En aquest cas, l'agent client, es pregunta per si hi ha alguna missió aquest any. L'agent que té tota la informació mira si dins la seva base de coneixement hi ha algun element enguany, i veu que si, que hi ha previst un llançament el proper estiu, en aquest cas, la llista resultant només conté un element.

```
((all ?x
    (is-shuttle
      (Shuttle
        :ShuttleMissionDescription "2005"
        :ShuttleMission "2005"))))

((=
    (all ?x
      (is-shuttle
        (Shuttle
          :ShuttleMissionDescription "2005"
          :ShuttleMission "2005")))
    (set
      (Shuttle
        :BackUpCrew
          (sequence
            (NASAMissionSpecialist
              :name "Rick Husband")
            (NASAMissionSpecialist
              :name "Willie McCool")
            (NASAMissionSpecialist
              :name "Dave Brown")
            (NASAMissionSpecialist
              :name "Laurel Clark")
            (NASAMissionSpecialist
              :name "Kalpana Chawla")
            (NASAMissionSpecialist
              :name "Soichi Noguchi"))
        :MissionSpecialist
          (EngineerScientific
            :name "Michael Anderson")
        :OriginCrew
          (sequence
            (NASAMissionSpecialist
              :name "Rick Husband")
            (NASAMissionSpecialist
              :name "Willie McCool")
            (NASAMissionSpecialist
              :name "Dave Brown")
            (NASAMissionSpecialist
              :name "Laurel Clark")
            (NASAMissionSpecialist
              :name "Kalpana Chawla")
            (NASAMissionSpecialist
              :name "Soichi Noguchi"))
        :ShuttleMissionDescription "STS114-S-002 (2005). The STS-114 crewmembers
will deliver supplies to the International Space Station, but the major focus of
their mission will be testing and evaluating new Space Shuttle flight safety,
which includes new inspection and repair techniques."
        :ShuttlePilot
          (NASAMissionSpecialist
            :name "Willie McCool")
        :Launch "July 13, 2005 - July 31, 2005"
        :ShuttleCommander
          (NASAMissionSpecialist
            :name "Rick Husband")
        :AdministrativeManagement
          (ClericalAdmSupport
            :name "Bryan Lunney"
            :identifier JSC2002-E-04739
            :position "Flight Director")
        :ShuttleMission "STS 114-S-002"))))
```

3 Exemple complet

A continuació es detalla un exemple concret de Sistema Multiagent format per dos agents que intercanvien dades fent servir les eines que ens proporciona JADE descrites abans (comportaments, ontologies, protocols, etc). En aquest cas, l'agent tipus NASAAgent espera rebre dos tipus de missatges: query o request. En el cas del primer, mirarà si té algun cas dins la seva base de casos que concordi amb el demanat; en el cas del request, acaba la seva execució. L'altre agent, de tipus ClientAgent, demana informació de missions d'algun any concret, o amb algun tripulant concret. En el cas de l'exemple, es demana per alguna missió de l'any 2005.

L'agent NASAAgent carrega els casos en el setup i no es poden modificar més. Aquesta càrrega és independent de la funció d'agent i es podria fer des d'una base de dades, o algun fitxer exterior.

3.1 Agents

ClientAgent

```
package agents;

import java.io.*;
import jade.util.leap.*;
import java.util.Vector;
import jade.content.lang.Codec;
import jade.content.*;
import jade.content.abs.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.sl.*;
import jade.lang.acl.*;
import jade.core.*;
import jade.core.behaviours.*;
import jade.proto.*;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.*;
import ontology.shuttle.*;

/**
 * <p>Title: Shuttle MAS</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Universitat Rovira i Virgili (URV)</p>
 * @author GruSMA
 * @version 1.0
 */
public class ClientAgent extends Agent {

  private Agent myagent;
  private Codec codec;
  private Ontology ontology;


  /**
   * Busquem els agents d'un tipus o un nom donats
```

```
   * @param type Tipus d'agent buscat; null si no desitjem aquest crite-
ri
   * @param name Nom d'un agent buscat; null si no desitjem aquest cri-
teri
   * @return llista amb els AIDs dels agents trobats
   */
  private jade.util.leap.List buscarAgents(String type,String name) {
    jade.util.leap.List results = new jade.util.leap.ArrayList();
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    if(type!=null) sd.setType(type);
    if(name!=null) sd.setName(name);
    dfd.addServices(sd);
    try {
      SearchConstraints c = new SearchConstraints();
      c.setMaxResults(new Long(-1));
      DFAgentDescription[] DFAgents = DFService.search(this,dfd,c);
      int i=0;
      while ((DFAgents != null) && (i<DFAgents.length)) {
        DFAgentDescription agent = DFAgents[i];
        i++;
        Iterator services = agent.getAllServices();
        boolean found = false;
        ServiceDescription service = null;
        while (services.hasNext() && !found) {
          service = (ServiceDescription)services.next();
          found    =    (service.getType().equals(type)    ||    servi-
ce.getName().equals(name));
        }
        if (found) {
          results.add((AID)agent.getName());
          //System.out.println(agent.getName()+"\n");
        }
      }
    } catch (FIPAException e) {
      System.out.println("ERROR: "+e.toString());
    }
    return results;
  }


  /**
   * Mètode per a registrar l'agent al DF. Cal aportar tota la informa-
ció
   * disponible o que es vulgui donar per a que qualsevol altre agent el
   * pugui trobar si el busca
   */
  private void RegistrarDF() {
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setName(this.getClass().toString());
    sd.setType("InformationAgent");
    sd.setOwnership("GruSMA");
    dfd.addServices(sd);
    dfd.setName(getAID());
    try {
      DFService.register(this, dfd);
      System.out.println("Registration           successful           of
"+this.getLocalName());
    } catch (jade.domain.FIPAException e) {
      System.err.println("Registration  ("+this.getLocalName()+")  ERROR.
Reason "+e.toString());
      doDelete();
```

```
    }
  } //endof RegistrarDF

  /**
    * Registrar la/es ontologia/es permeses per l'agent
    */
   private void registrarOntologia() {
     this.codec = new SLCodec();
     this.ontology = ShuttleOntology.getInstance();
     getContentManager().registerLanguage(codec);
     getContentManager().registerOntology(ontology);
   }


   /**
    * Mètode per a deregistrar l'agent del DF
    */
   private void DeRegistrarDF() {
     try {
       DFService.deregister(this);
       System.out.println("Registration          successful          of
"+this.getLocalName());
     } catch (Exception e) {
       System.out.println("Deregistration      ("+this.getLocalName()+")
ERROR. Reason "+e.toString());
     }
   } //endof DeRegistrarDF

   public void takeDown(){
     DeRegistrarDF();
   }

  public void setup() {
    RegistrarDF();
    registrarOntologia();
    showMessage("Initialised successful");

    iniciarPregunta("2005");
    showMessage("Finished");
  } //endof setup

  public void showMessage(String mss) {
    System.out.println(getLocalName()+": "+mss);
  }

  private void iniciarPregunta(String condition) {
    jade.util.leap.List            r             =            buscarA-
gents("ShuttleInformationAgent",null);
    if(r.size()==0) {
      showMessage("No <ShuttleInformationAgent> present");
      return;
    }
    showMessage("("+r.size()+") ShuttleInformationAgent present");

    ACLMessage query = createQueryShuttle(r,condition);
    this.addBehaviour(new QueryInitiator(this,query));
  } //endof iniciarPregunta


  /**
   * Crear una pregunta
   * @param receivers Set of receivers of that message
```

```
   * @param shuttle_wanted Condition or set of conditions to consider in
the
   * search
   * @return QUERY-REF created to sent
   */
  private ACLMessage createQueryShuttle(jade.util.leap.List receivers,
                                        String shuttle_wanted) {
      ACLMessage dummy = new ACLMessage( ACLMessage.QUERY_REF );
      dummy.setSender((AID)getAID());
      dummy.setLanguage(this.codec.getName());
      dummy.setOntology(ShuttleOntology.ONTOLOGY_NAME);
      dummy.setProtocol("FIPA-Query");
      dummy.clearAllReceiver();
      for(int i=0; i<receivers.size(); i++)
        dummy.addReceiver((AID)receivers.get(i));


      try {

        AbsConcept abs = new AbsConcept(ShuttleOntology.SHUTTLE);
        abs.set(ShuttleOntology.SHUTTLE_SHUTTLEMISSION, shuttle_wanted);
        abs.set(ShuttleOntology.SHUTTLE_SHUTTLEMISSIONDESCRIPTION,
shuttle_wanted);

        AbsPredicate      absIsShuttle      =      new      AbsPredica-
te(ShuttleOntology.IS_SHUTTLE);
        absIsShuttle.set(ShuttleOntology.IS_SHUTTLE_ITEM, abs);

        AbsVariable var = new AbsVariable("x", ShuttleOntology.SHUTTLE);

        AbsIRE allPred = new AbsIRE(SLVocabulary.ALL);
        allPred.setVariable(var);
        allPred.setProposition(absIsShuttle);

        this.getContentManager().fillContent(dummy,allPred);

      }
      catch( Exception e ) {
        System.err.println(getLocalName()+" ERROR: "+e.toString());
      }

      return dummy;
  } //endof createQueryShuttle


  /**
   *
   * <p>Title: Shuttle MAS</p>
   * <p>Description: </p>
   * <p>Copyright: Copyright (c) 2005</p>
   * <p>Company: Universitat Rovira i Virgili (URV)</p>
   * @author GruSMA
   * @version 1.0
   */
  class QueryInitiator extends AchieveREInitiator {

    private Agent parent;

    public QueryInitiator(Agent myAgent,ACLMessage m){
      super(myAgent,m);
      this.parent = myAgent;
    }
```

```
    protected void handleAllResultNotifications(Vector responses) {
      showMessage("Has received "+ responses.size() +" messages");
      /** aqui manegariem el contingut de l'exemple */
    }

    protected void handleNotUnderstood(ACLMessage msg) {
      showMessage("Has received a "+
                  ACLMessage.getPerformative(msg.getPerformative())+
                  " message from "+msg.getSender().getLocalName());
    }

    protected void handleRefuse(ACLMessage msg) {
      showMessage("Has received a "+
                  ACLMessage.getPerformative(msg.getPerformative())+
                  " message from "+msg.getSender().getLocalName());
    }

  } //endof class QueryInitiator


} //endof class ClientAgent
```

NASAAgent

```
package agents;

import java.io.*;
import jade.util.leap.*;
import java.util.Vector;
import jade.content.lang.Codec;
import jade.content.*;
import jade.content.abs.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import jade.content.lang.sl.*;
import jade.lang.acl.*;
import jade.core.*;
import jade.core.behaviours.*;
import jade.proto.*;
import jade.domain.FIPAAgentManagement.*;
import jade.domain.*;
import ontology.shuttle.*;
import agents.IO;

/**
 * <p>Title: Shuttle MAS</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Universitat Rovira i Virgili (URV)</p>
 * @author GruSMA
 * @version 1.0
 */
public class NASAAgent extends Agent {

  private Agent myagent = this;
  private Codec codec;
  private Ontology ontology;
  ontology.shuttle.Shuttle availableMissions[] = null;

  /**
   * Carregar les dades disponibles per l'agent. Aquesta càrrega pot ser
```

```
   * externa i feta abans d'iniciar-se o abans de rebre cap missatge.
   */
  private void carregarDades() {
    this.availableMissions = new ontology.shuttle.Shuttle[2];
    this.availableMissions = IO.getMissions();
  }


  /**
   * Mètode per a registrar l'agent al DF. Cal aportar tota la informa-
ció
   * disponible o que es vulgui donar per a que qualsevol altre agent el
   * pugui trobar si el busca
   */
  private void RegistrarDF() {
    DFAgentDescription dfd = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setName(this.getClass().toString());
    sd.setType("ShuttleInformationAgent");
    sd.setOwnership("GruSMA");

    Property prop = new Property(); /* optional */
    prop.setName("NASA");
    prop.setValue("shuttle");
    sd.addProperties(prop);

    dfd.addServices(sd);
    dfd.setName(getAID());
    try {
      DFService.register(this, dfd);
      showMessage("Registration OK!");
    } catch (jade.domain.FIPAException e) {
      showMessage("Registration error("+
                  this.getLocalName()+
                  ") ERROR. Reason "+e.toString());
      doDelete();
    }

  } //endof RegistrarDF

  /**
   * Registrar la/es ontologia/es permeses per l'agent
   */
  private void registrarOntologia() {
    this.codec = new SLCodec();
    this.ontology = ShuttleOntology.getInstance();
    getContentManager().registerLanguage(codec);
    getContentManager().registerOntology(ontology);
  }

  /**
   * Mètode per a deregistrar l'agent del DF
   */
  private void DeRegistrarDF() {
    try {
      DFService.deregister(this);
      System.out.println("Registration           successful           of
"+this.getLocalName());
    } catch (Exception e) {
      System.out.println("Deregistration       ("+this.getLocalName()+")
ERROR. Reason "+e.toString());
    }
  } //endof DeRegistrarDF
```

```
  public void takeDown(){
    DeRegistrarDF();
  }


  /**
   * Dóna d'alta el behaviour per escoltar els missatges entrants
   */
  public void setup() {
    carregarDades();
    RegistrarDF();
    registrarOntologia();

    AchieveREResponder b1 =
        new  QueryResponder(this,  MessageTemplate.MatchProtocol("FIPA-
Query"));
    addBehaviour(b1);
    AchieveREResponder b = new RequestResponder(this,
        MessageTemplate.MatchPerformative(ACLMessage.REQUEST));
    addBehaviour(b);

    showMessage("All behaviours added");
  }

  public void showMessage(String mss) {
    System.out.println(getLocalName()+": "+mss);
  }

  /**
   * <p>Title: Shuttle MAS</p>
   * <p>Description: </p>
   * <p>Copyright: Copyright (c) 2005</p>
   * <p>Company: Universitat Rovira i Virgili (URV)</p>
   * @author GruSMA
   * @version 1.0
   */
  private class QueryResponder extends AchieveREResponder {

    private NASAAgent parent = null;

    public QueryResponder(NASAAgent myAgent, MessageTemplate mt){
      super(myAgent, mt);
      this.parent = myAgent;
      showMessage("Ready to receive QUERIES ...");
    }

    private jade.util.leap.List cercaInformacio(String condition) {
      jade.util.leap.List r = new jade.util.leap.ArrayList();
      for(int i=0; i<parent.availableMissions.length; i++) {

if(parent.availableMissions[i].getShuttleMission().indexOf(condition)!=-
1) {
          r.add(parent.availableMissions[i]);
        }                                                    else
if(parent.availableMissions[i].getShuttleMissionDescription().indexOf(co
ndition)!=-1) {
          r.add(parent.availableMissions[i]);
        }
      }
      return r;
    }
```

```
    protected ACLMessage prepareResponse(ACLMessage query) {
      showMessage("Received a "+
                  ACLMessage.getPerformative(query.getPerformative())+
                  " message from "+query.getSender().getLocalName());

      ACLMessage reply = query.createReply();
      reply.setPerformative(ACLMessage.INFORM);

      try {
        showMessage("Extracting content: "+query.getOntology());
//        showMessage("Content: <"+query.getContent()+">");
        AbsContentElement            allCE            =            pa-
rent.getContentManager().extractAbsContent(query);
        AbsIRE allPred = (AbsIRE)allCE;

        if(allPred.getTypeName().equals(SLVocabulary.ALL)) {

          showMessage("Extracting  content:  Predicate  ALL  identified
...");
          AbsPredicate pred = (AbsPredicate)allPred.getProposition();

          showMessage("Extracting content: Reading constraints ...");
          showMessage(pred.toString());

          AbsConcept             c             =             (AbsCon-
cept)pred.getAbsTerm(ShuttleOntology.IS_SHUTTLE_ITEM);
          showMessage(c.toString());

          String                           str                          =
c.getString(ShuttleOntology.SHUTTLE_SHUTTLEMISSION);
          showMessage(str);

          List results = new ArrayList();
          results = cercaInformacio((String)str);
          showMessage("Extracting      content:      Results      made
("+results.size()+")");

          /** Based upon the example showed in FIPA-SL Specification,
page 11. */
          /** result: ((= (all ...) (set <list>))) */

          AbsAggregate absSet = new AbsAggregate(BasicOntology.SET);
          for (int k = 0; k < results.size(); k++) {
            Shuttle si = (Shuttle)results.get(k);
            AbsConcept elem = (AbsConcept)ontology.fromObject(si);
            absSet.add(elem);
          }

          AbsPredicate      equalPred      =      new      AbsPredica-
te(SLVocabulary.EQUALS);
          equalPred.set(SLVocabulary.EQUALS_LEFT,allPred);
          equalPred.set(SLVocabulary.EQUALS_RIGHT,absSet);

          parent.getContentManager().fillContent(reply, equalPred);
        } //endif

      } catch (OntologyException oe) {
        oe.printStackTrace();
      } catch (Codec.CodecException ce) {
        ce.printStackTrace();
      } catch (Exception e) {
        e.printStackTrace();
      }
```

```
      return reply;
    } //endof prepareResponse


  } //endof class QueryResponder



  /**
   * <p>Title: Shuttle MAS</p>
   * <p>Description: </p>
   * <p>Copyright: Copyright (c) 2005</p>
   * <p>Company: Universitat Rovira i Virgili (URV)</p>
   * @author GruSMA
   * @version 1.0
   */
  private class RequestResponder extends AchieveREResponder {

    private Agent parent = null;

    public RequestResponder(Agent myAgent, MessageTemplate mt){
      super(myAgent, mt);
      this.parent = myAgent;
      System.out.println(getLocalName()+" is ready to receive REQUESTS
...");
    }

    protected ACLMessage prepareResponse(ACLMessage request) {
      System.out.println(this.parent.getLocalName()+
                         " has received a "+
                         ACLMessa-
ge.getPerformative(request.getPerformative())+
                         "                    message                 from
"+request.getSender().getLocalName());
      ACLMessage response = request.createReply();
      response.setPerformative(ACLMessage.AGREE);
      return response;
    }

    protected ACLMessage prepareResultNotification (ACLMessage request,
ACLMessage reponse) {
      System.out.println(this.parent.getLocalName()+
                         " has received a "+
                         ACLMessa-
ge.getPerformative(reponse.getPerformative())+
                         "                    message                 from
"+reponse.getSender().getLocalName());
      DeRegistrarDF();
      ACLMessage reply = request.createReply();
      reply.setPerformative(ACLMessage.INFORM);
      return reply;
    }
  } //endof class RequestResponder

} //endof class NASAAgent
```

IO

```
package agents;

import ontology.shuttle.*;
import jade.util.leap.*;


/**
 * <p>Title: Shuttle MAS</p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: Universitat Rovira i Virgili (URV)</p>
 * @author GruSMA
 * @version 1.0
 */

public class IO {

  public ontology.shuttle.Shuttle s = new ontology.shuttle.Shuttle();



  public static ontology.shuttle.Shuttle[] getMissions() {

    ontology.shuttle.Shuttle[]      mission      =      new      ontolo-
gy.shuttle.Shuttle[2];

    /** mission(0) STS 114-S-002 */
    mission[0] = new ontology.shuttle.Shuttle();
    mission[0].setLaunch("July 13, 2005 - July 31, 2005");
    mission[0].setShuttleMissionDescription("STS114-S-002  (2005).  The
STS-114 crewmembers will deliver supplies to the International Space
Station, but the major focus of their mission will be testing and evalu-
ating new Space Shuttle flight safety, which includes new inspection and
repair techniques.");
    mission[0].setShuttleMission("STS 114-S-002");
    ontology.shuttle.ClericalAdmSupport c = new ClericalAdmSupport();
    c.setName("Bryan Lunney");
    c.setPosition("Flight Director");
    c.setIdentifier("JSC2002-E-04739");
    mission[0].setAdministrativeManagement(c);

    ontology.shuttle.NASAMissionSpecialist commander = new NASAMission-
Specialist();
    commander.setName("Eileen M. Collins");
    mission[0].setShuttleCommander(commander);

    ontology.shuttle.EngineerScientific specialist = new EngineerScien-
tific();
    specialist.setName("Wendy B. Lawrence");
    mission[0].setMissionSpecialist(specialist);

    ontology.shuttle.NASAMissionSpecialist pilot = new NASAMissionSpeci-
alist();
    pilot.setName("James M. Kelly");
    mission[0].setShuttlePilot(pilot);

    ontology.shuttle.NASAMissionSpecialist astronaut1 = new NASAMission-
Specialist();
    astronaut1.setName("Stephen K. Robinson");
```

```
    ontology.shuttle.NASAMissionSpecialist astronaut2 = new NASAMission-
Specialist();
    astronaut2.setName("Andrew S. W. Thomas");

    ontology.shuttle.NASAMissionSpecialist astronaut3 = new NASAMission-
Specialist();
    astronaut3.setName("Charles J. Camarda");

    ontology.shuttle.NASAMissionSpecialist astronaut4 = new NASAMission-
Specialist();
    astronaut4.setName("Soichi Noguchi");

    List crew = new ArrayList();
    crew.add(commander);
    crew.add(pilot);
    crew.add(astronaut1);
    crew.add(astronaut2);
    crew.add(astronaut3);
    crew.add(astronaut4);
    mission[0].setOriginCrew(crew);
    mission[0].setBackUpCrew(crew);


    /** mission(1) - Last mission, the Columbia disaster */
    mission[1] = new ontology.shuttle.Shuttle();
    mission[1].setLaunch("Jan. 16, 2003 9:39 a.m. CST");
    mission[1].setShuttleMissionDescription("Columbia broke up during
re-entry over Texas en route to landing at Kennedy Space Center, Fla.
More info at http://spaceflight.nasa.gov/shuttle/archives/sts-
107/index.html");
    mission[1].setShuttleMission("STS 107");
    c = new ClericalAdmSupport();
    c.setName("Ilan Ramon");
    c.setPosition("");
    c.setIdentifier("");
    mission[1].setAdministrativeManagement(c);

    commander = new NASAMissionSpecialist();
    commander.setName("Rick Husband");
    mission[0].setShuttleCommander(commander);

    specialist = new EngineerScientific();
    specialist.setName("Michael Anderson");
    mission[0].setMissionSpecialist(specialist);

    pilot = new NASAMissionSpecialist();
    pilot.setName("Willie McCool");
    mission[0].setShuttlePilot(pilot);

    astronaut1 = new NASAMissionSpecialist();
    astronaut1.setName("Dave Brown");

    astronaut2 = new NASAMissionSpecialist();
    astronaut2.setName("Laurel Clark");

    astronaut3 = new NASAMissionSpecialist();
    astronaut3.setName("Kalpana Chawla");

    astronaut4 = new NASAMissionSpecialist();
    astronaut4.setName("Soichi Noguchi");

    crew = new ArrayList();
    crew.add(commander);
```

```
    crew.add(pilot);
    crew.add(astronaut1);
    crew.add(astronaut2);
    crew.add(astronaut3);
    crew.add(astronaut4);
    mission[0].setOriginCrew(crew);
    mission[0].setBackUpCrew(crew);

    return mission;
  }


}
```

3.2 Ontologia

ShuttleOntology

```
// file: ShuttleOntology.java generated by ontology bean generator.  DO
NOT EDIT, UNLESS YOU ARE REALLY SURE WHAT YOU ARE DOING!
package ontology.shuttle;

import jade.content.onto.*;
import jade.content.schema.*;
import jade.util.leap.HashMap;
import jade.content.lang.Codec;
import jade.core.CaseInsensitiveString;

/** file: ShuttleOntology.java
 * @author ontology bean generator
 * @version 2005/05/27, 11:11:34
 */
public class ShuttleOntology extends jade.content.onto.Ontology imple-
ments ProtegeTools.ProtegeOntology {
   /**
    * These hashmap store a mapping from jade names to either protege
names of SlotHolder
    * containing the protege names.  And vice versa
    */
   private HashMap jadeToProtege;

  //NAME
  public static final String ONTOLOGY_NAME = "Shuttle";
  // The singleton instance of this ontology
  private static ProtegeIntrospector introspect = new ProtegeIntrospec-
tor();
  private static Ontology theInstance = new ShuttleOntology();
  public static Ontology getInstance() {
     return theInstance;
  }

   // ProtegeOntology methods
   public SlotHolder getSlotNameFromJADEName(SlotHolder jadeSlot) {
     return (SlotHolder) jadeToProtege.get(jadeSlot);
   }


   // storing the information
   private void storeSlotName(String jadeName, String javaClassName,
String slotName){
```

```
       jadeToProtege.put(new  SlotHolder(javaClassName,  jadeName),  new
SlotHolder(javaClassName, slotName));
   }


   // VOCABULARY
    public static final String IS_SHUTTLE_ITEM="item";
    public static final String IS_SHUTTLE="is-shuttle";
    public static final String EMPLOYEE_DEPARTMENT="department";
    public static final String EMPLOYEE_IDENTIFIER="identifier";
    public static final String EMPLOYEE_NAME="name";
    public static final String EMPLOYEE="Employee";
    public static final String TRADESLABOR="TradesLabor";
    public static final String CLERICALADMSUPPORT_POSITION="position";
    public static final String CLERICALADMSUPPORT="ClericalAdmSupport";
    public static final String ENGINEERSCIENTIFIC="EngineerScientific";
    public               static               final               String
TECNICALMEDICALSUPPORT_SPECIALITY="speciality";
    public               static               final               String
TECNICALMEDICALSUPPORT="TecnicalMedicalSupport";
    public static final String SHUTTLE_SHUTTLEMISSION="ShuttleMission";
    public               static               final               String
SHUTTLE_ADMINISTRATIVEMANAGEMENT="AdministrativeManagement";
    public               static               final               String
SHUTTLE_SHUTTLECOMMANDER="ShuttleCommander";
    public static final String SHUTTLE_LAUNCH="Launch";
    public static final String SHUTTLE_SHUTTLEPILOT="ShuttlePilot";
    public               static               final               String
SHUTTLE_SHUTTLEMISSIONDESCRIPTION="ShuttleMissionDescription";
    public static final String SHUTTLE_ORIGINCREW="OriginCrew";
    public               static               final               String
SHUTTLE_MISSIONSPECIALIST="MissionSpecialist";
    public static final String SHUTTLE_BACKUPCREW="BackUpCrew";
    public static final String SHUTTLE="Shuttle";
    public               static               final               String
NASAMISSIONSPECIALIST="NASAMissionSpecialist";

  /**
   * Constructor
  */
  private ShuttleOntology(){
    super(ONTOLOGY_NAME, BasicOntology.getInstance());
    introspect.setOntology(this);
    jadeToProtege = new HashMap();
    try {

    // adding Concept(s)
    ConceptSchema   nasaMissionSpecialistSchema   =   new   ConceptSche-
ma(NASAMISSIONSPECIALIST);
    add(nasaMissionSpecialistSchema,                                ontolo-
gy.shuttle.NASAMissionSpecialist.class);
    ConceptSchema shuttleSchema = new ConceptSchema(SHUTTLE);
    add(shuttleSchema, ontology.shuttle.Shuttle.class);
    ConceptSchema   tecnicalMedicalSupportSchema   =   new   ConceptSche-
ma(TECNICALMEDICALSUPPORT);
    add(tecnicalMedicalSupportSchema,                               ontolo-
gy.shuttle.TecnicalMedicalSupport.class);
    ConceptSchema    engineerScientificSchema    =    new    ConceptSche-
ma(ENGINEERSCIENTIFIC);
    add(engineerScientificSchema,                                   ontolo-
gy.shuttle.EngineerScientific.class);
    ConceptSchema    clericalAdmSupportSchema    =    new    ConceptSche-
ma(CLERICALADMSUPPORT);
```

```
    add(clericalAdmSupportSchema,                                ontolo-
gy.shuttle.ClericalAdmSupport.class);
    ConceptSchema tradesLaborSchema = new ConceptSchema(TRADESLABOR);
    add(tradesLaborSchema, ontology.shuttle.TradesLabor.class);
    ConceptSchema employeeSchema = new ConceptSchema(EMPLOYEE);
    add(employeeSchema, ontology.shuttle.Employee.class);

    // adding AgentAction(s)

    // adding AID(s)

    // adding Predicate(s)
    PredicateSchema is_shuttleSchema = new PredicateSchema(IS_SHUTTLE);
    add(is_shuttleSchema, ontology.shuttle.Is_shuttle.class);


    // adding fields
    shuttleSchema.add(SHUTTLE_BACKUPCREW, employeeSchema, 0, ObjectSche-
ma.UNLIMITED);
    shuttleSchema.add(SHUTTLE_MISSIONSPECIALIST, engineerScientificSche-
ma, ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_ORIGINCREW, employeeSchema, 0, ObjectSche-
ma.UNLIMITED);
    shuttleSchema.add(SHUTTLE_SHUTTLEMISSIONDESCRIPTION,      (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_SHUTTLEPILOT, nasaMissionSpecialistSchema,
ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_LAUNCH,                            (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_SHUTTLECOMMANDER,       nasaMissionSpecia-
listSchema, ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_ADMINISTRATIVEMANAGEMENT,  clericalAdmSup-
portSchema, ObjectSchema.OPTIONAL);
    shuttleSchema.add(SHUTTLE_SHUTTLEMISSION,                    (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    tecnicalMedicalSupportSchema.add(TECNICALMEDICALSUPPORT_SPECIALITY,
(TermSchema)getSchema(BasicOntology.STRING), 0, ObjectSchema.UNLIMITED);
    clericalAdmSupportSchema.add(CLERICALADMSUPPORT_POSITION, (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    employeeSchema.add(EMPLOYEE_NAME,                            (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.MANDATORY);
    employeeSchema.add(EMPLOYEE_IDENTIFIER,                      (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    employeeSchema.add(EMPLOYEE_DEPARTMENT,                      (TermSche-
ma)getSchema(BasicOntology.STRING), ObjectSchema.OPTIONAL);
    is_shuttleSchema.add(IS_SHUTTLE_ITEM,   shuttleSchema,   ObjectSche-
ma.MANDATORY);

    // adding name mappings
    storeSlotName("BackUpCrew",   "ontology.shuttle.Shuttle",   "BackUp-
Crew");
    storeSlotName("MissionSpecialist", "ontology.shuttle.Shuttle", "Mis-
sionSpecialist");
    storeSlotName("OriginCrew",   "ontology.shuttle.Shuttle",   "Origin-
Crew");
    storeSlotName("ShuttleMissionDescription",                   "ontolo-
gy.shuttle.Shuttle", "ShuttleMissionDescription");
    storeSlotName("ShuttlePilot",  "ontology.shuttle.Shuttle",  "Shuttle-
Pilot");
    storeSlotName("Launch", "ontology.shuttle.Shuttle", "Launch");
    storeSlotName("ShuttleCommander",        "ontology.shuttle.Shuttle",
"ShuttleCommander");
```

```
    storeSlotName("AdministrativeManagement",                    "ontolo-
gy.shuttle.Shuttle", "AdministrativeManagement");
    storeSlotName("ShuttleMission",          "ontology.shuttle.Shuttle",
"ShuttleMission");
    storeSlotName("speciality",                                  "ontolo-
gy.shuttle.TecnicalMedicalSupport", "speciality");
    storeSlotName("position",     "ontology.shuttle.ClericalAdmSupport",
"position");
    storeSlotName("name", "ontology.shuttle.Employee", "name");
    storeSlotName("identifier", "ontology.shuttle.Employee", "identifi-
er");
    storeSlotName("department",  "ontology.shuttle.Employee",  "depart-
ment");
    storeSlotName("item", "ontology.shuttle.Is_shuttle", "item");

    // adding inheritance
    nasaMissionSpecialistSchema.addSuperSchema(employeeSchema);
    tecnicalMedicalSupportSchema.addSuperSchema(employeeSchema);
    engineerScientificSchema.addSuperSchema(employeeSchema);
    clericalAdmSupportSchema.addSuperSchema(employeeSchema);
    tradesLaborSchema.addSuperSchema(employeeSchema);

   }catch (java.lang.Exception e) {e.printStackTrace();}
  }
  }
```

ClericalAdmSupport

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: ClericalAdmSupport
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class ClericalAdmSupport extends Employee{

   /**
   * Protege name: position
   */
   private String position;
   public void setPosition(String value) {
    this.position=value;
   }
   public String getPosition() {
     return this.position;
   }

}
```

Employee

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
```

```
import jade.core.*;

/**
   * Protege name: Employee
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class Employee implements Concept {

   /**
   * Protege name: name
   */
   private String name;
   public void setName(String value) {
    this.name=value;
   }
   public String getName() {
     return this.name;
   }

   /**
   * Protege name: identifier
   */
   private String identifier;
   public void setIdentifier(String value) {
    this.identifier=value;
   }
   public String getIdentifier() {
     return this.identifier;
   }

   /**
   * Protege name: department
   */
   private String department;
   public void setDepartment(String value) {
    this.department=value;
   }
   public String getDepartment() {
     return this.department;
   }

}
```

EngineerScientific

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: EngineerScientific
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class EngineerScientific extends Employee{

}
```

Is-shuttle

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: is-shuttle
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class Is_shuttle implements Predicate {

   /**
   * Protege name: item
   */
   private Shuttle item;
   public void setItem(Shuttle value) {
    this.item=value;
   }
   public Shuttle getItem() {
     return this.item;
   }

}
```

NASAMissionSpecialist

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: NASAMissionSpecialist
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class NASAMissionSpecialist extends Employee{

}
```

Shuttle

```
import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: Shuttle
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class Shuttle implements Concept {
```

```
/**
* Protege name: BackUpCrew
*/
private List backUpCrew = new ArrayList();
public void addBackUpCrew(Employee elem) {
  backUpCrew.add(elem);
}
public boolean removeBackUpCrew(Employee elem) {

  boolean result = backUpCrew.remove(elem);
  return result;
}
public void clearAllBackUpCrew() {

  backUpCrew.clear();
}
public Iterator getAllBackUpCrew() {return backUpCrew.iterator(); }
public List getBackUpCrew() {return backUpCrew; }
public void setBackUpCrew(List l) {backUpCrew = l; }

/**
* Protege name: MissionSpecialist
*/
private EngineerScientific missionSpecialist;
public void setMissionSpecialist(EngineerScientific value) {
 this.missionSpecialist=value;
}
public EngineerScientific getMissionSpecialist() {
  return this.missionSpecialist;
}

/**
* Protege name: OriginCrew
*/
private List originCrew = new ArrayList();
public void addOriginCrew(Employee elem) {
  originCrew.add(elem);
}
public boolean removeOriginCrew(Employee elem) {

  boolean result = originCrew.remove(elem);
  return result;
}
public void clearAllOriginCrew() {

  originCrew.clear();
}
public Iterator getAllOriginCrew() {return originCrew.iterator(); }
public List getOriginCrew() {return originCrew; }
public void setOriginCrew(List l) {originCrew = l; }

/**
* Protege name: ShuttleMissionDescription
*/
private String shuttleMissionDescription;
public void setShuttleMissionDescription(String value) {
 this.shuttleMissionDescription=value;
}
public String getShuttleMissionDescription() {
  return this.shuttleMissionDescription;
}

/**
```

```
   * Protege name: ShuttlePilot
   */
   private NASAMissionSpecialist shuttlePilot;
   public void setShuttlePilot(NASAMissionSpecialist value) {
    this.shuttlePilot=value;
   }
   public NASAMissionSpecialist getShuttlePilot() {
     return this.shuttlePilot;
   }

   /**
   * Protege name: Launch
   */
   private String launch;
   public void setLaunch(String value) {
    this.launch=value;
   }
   public String getLaunch() {
     return this.launch;
   }

   /**
   * Protege name: ShuttleCommander
   */
   private NASAMissionSpecialist shuttleCommander;
   public void setShuttleCommander(NASAMissionSpecialist value) {
    this.shuttleCommander=value;
   }
   public NASAMissionSpecialist getShuttleCommander() {
     return this.shuttleCommander;
   }

   /**
   * Protege name: AdministrativeManagement
   */
   private ClericalAdmSupport administrativeManagement;
   public void setAdministrativeManagement(ClericalAdmSupport value) {
    this.administrativeManagement=value;
   }
   public ClericalAdmSupport getAdministrativeManagement() {
     return this.administrativeManagement;
   }

   /**
   * Protege name: ShuttleMission
   */
   private String shuttleMission;
   public void setShuttleMission(String value) {
    this.shuttleMission=value;
   }
   public String getShuttleMission() {
     return this.shuttleMission;
   }

}
```

TecnicalMedicalSupport

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
```

```
import jade.core.*;

/**
   * Protege name: TecnicalMedicalSupport
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class TecnicalMedicalSupport extends Employee{

   /**
   * Protege name: speciality
   */
   private List speciality = new ArrayList();
   public void addSpeciality(String elem) {
     speciality.add(elem);
   }
   public boolean removeSpeciality(String elem) {

     boolean result = speciality.remove(elem);
     return result;
   }
   public void clearAllSpeciality() {

     speciality.clear();
   }
   public Iterator getAllSpeciality() {return speciality.iterator(); }
   public List getSpeciality() {return speciality; }
   public void setSpeciality(List l) {speciality = l; }

}
```

TradesLabor

```
package ontology.shuttle;

import jade.content.*;
import jade.util.leap.*;
import jade.core.*;

/**
   * Protege name: TradesLabor
* @author ontology bean generator
* @version 2005/05/27, 11:11:34
*/
public class TradesLabor extends Employee{

}
```

Agraïments

Aquest treball ha estat realitzat amb el suport del "Departament d'Universitats, Recerca i Societat de la Informació" de Catalunya.

Bibliografia complementària

Tutorials de JADE:
Application-defined content languages and ontologies.
Administrator's guide.
Jade Programmer's guide.

Documents de la FIPA:
FIPA SL Content Language Specification. SC00008L.
FIPA Agent Management Specification. SC00023K.
FIPA Communicative Act Library Specification. SC00037J.

GruSMA. http://www.etse.urv.es/recerca/banzai/toni/MAS/
Projectes Final de Carrera entregats.
Material de treball (cursos, tutorials, guies, transparències, etc).