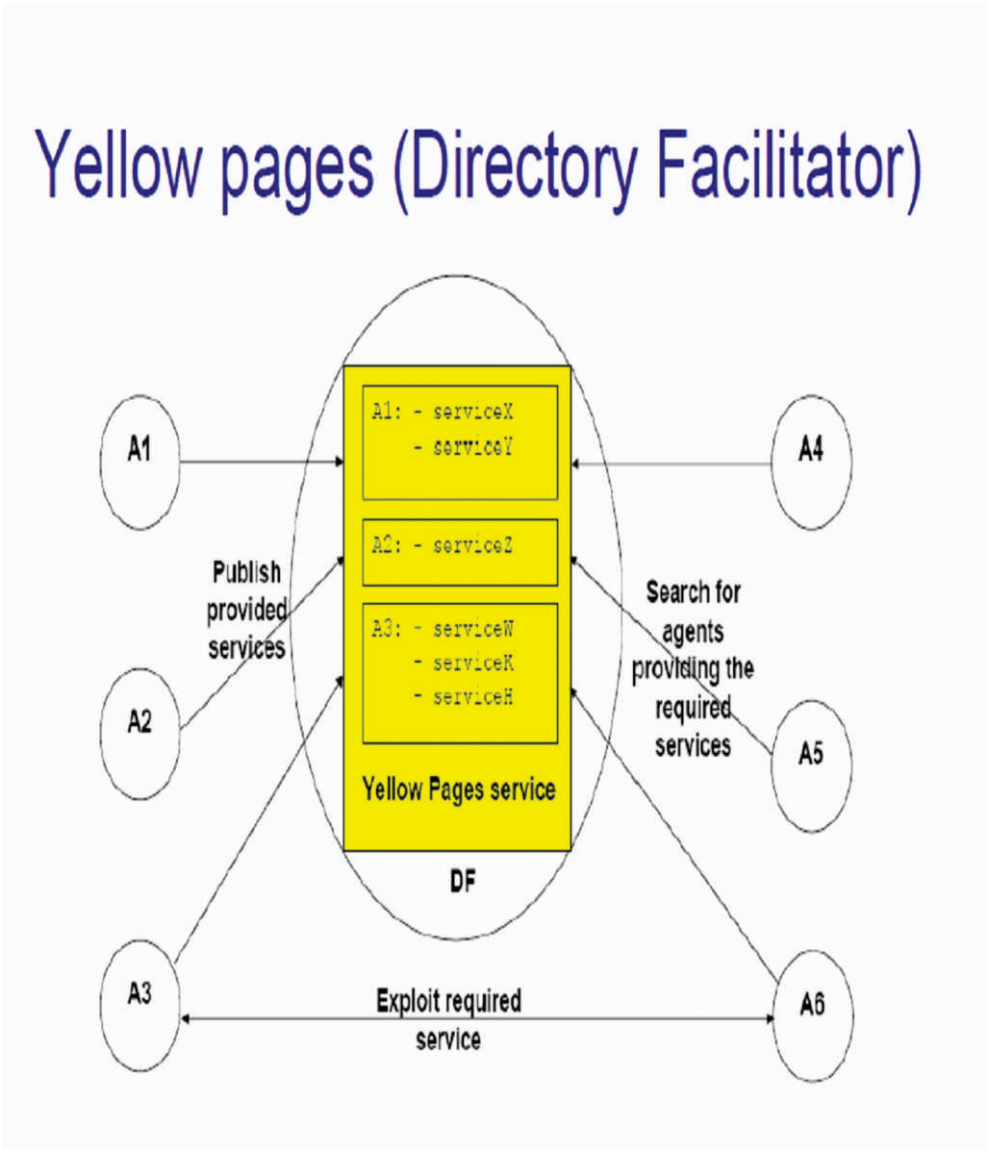
Daniel N. Pop Constantin Bălă Zamfirescu

Mediul de programare multiagent JADE sub ECLIPSE



VISMA®

Presa Universitară Clujeană

Daniel N. Pop **Constantin Bălă Zamfirescu**

Mediul de programare multiagent JADE sub ECLIPSE

Daniel N. Pop **Constantin Bălă Zamfirescu**

Mediul de programare multiagent JADE sub ECLIPSE

Presa Universitară Clujeană

2015

Referenți științifici:

Prof. univ. dr. ing. Florin Leon

Conf. univ. dr. Radu Tiberiu Trîmbițaş

ISBN 978-973-595-874-9

Tehnoredactare computerizată: Daniel N. Pop

Universitatea Babeş-Bolyai
Presa Universitară Clujeană
Director: Codruța Săcelean
Str. Hasdeu nr. 51
400371 Cluj-Napoca, România
Tel./Fax: (+40)-264-597.401
E-mail: editura@editura.ubbcluj.ro
http://www.editura.ubbcluj.ro/

Cuprins

Listă de figuri

Prefaţă

Programarea orientată pe agenţi POA este o paradigmă software relativ nouă care reuneşte idei din domeniile sistemelor distribuite (mai precis middleware), programare orientată pe obiect şi inteligenţă artificială. POA modelează o aplicaţie software sub forma unei colecţii de componente software numite agenţi. Sistemele multiagent (SMA) diferă de mai mulţi factori dintre care cel mai important este aplicabilitatea mediului respectiv.

Putem găsii instrumente dedicate unor categorii de aplicaţii specifice :

1. sisteme de mare complexitate, mai ales militare (*Cougaar, SEAS*),
2. simulare (*JAS, Repast, SeSAm, Swarm, SPADES, ACT-RBot +MRS),*
3. interacţiune socială cu agenti în oraşe virtuale (*FreeWalk/Q),*
4. comunicare/ coordonare în SMA (Sisteme MultiAgent) bazate pe Internet (*TuCSon [1])*

Am ales a prezenta mediul de dezvoltare JADE, fundamentat pe Java, care este distribuit sub forma unui pachet gratuit aceesibil la adresa:

`http://jade.tilab.com`.

JADE (JAVA Agent Development Framework) este o platformă de dezvoltare a agenţilor inteligenţi, cu o arhitectură modulară, bazată pe locaţii în care agenţii pot rula. JADE este un framework software folosit pentru dezvoltarea de S.M.A, ce corespunde specificaţiilor FIPA pentru agenţi inteligenţi.

Framework-ul a fost dezvoltat de către *CSELT (Centro Studi e Laboratori Telecommunicazioni,* cunoscut şi ca *Telecom Italia Lab)* împreună cu *Computer Engineering Group* de la Universitatea din Parma condus de *Giovanni Caire.* Creat iniţial în Java, JADE a fost portat în .NET compilându-se cu ANT (un utilitar de tip MakeFile ce foloseşte ca intrare fişiere XML).

Acesta conţine două produse :

1. o platformă pentru sisteme multiagent,

2. un cadru de dezvoltare al agenţilor Java.

JADE este scris integral în Java şi oferă o serie de funţionalităţi gata implementate precum şi interfeţe abstracte pentru adaptarea la cerinţele domeniilor abordate. JADE garantează conformitatea sintactică, iar acolo unde este posibil, conformitatea semantică cu specificaţiile FIPA pentru sisteme multiagent (FIPA, 2008).

Pentru *Eclipse* există există un plug-in *EJADE (2008)* care permite automatizarea includerii bibliotecilor necesare, pornirea şi oprirea platformei JADE, precum şi facilităţi de lansare şi depanare a agenţilor. Este cel mai facil mod de a lucra cu platforma JADE şi în consecinţă am ales utilizarea acestuia pentru rezolvarea problemelor din aceasta carte. Trebuie menţionat că s-au dezvoltat şi alte platforme de exemplu JADEX in cadrul Universitaţii din Hamburg (2010) [13] sau TROPOS la Universitatea din Lisabona (2008) [18].

De remarcat prezenţa mediului (*Spyse* [20]), bazat pe un limbaj în vădită ascensiune (*Python)* ce oferă proiectanţilor de S.M.A (şi de aplicaţii mai simple bazate pe agenţi) o alternativă de dezvoltare mai aproapiată de paradigma orientării spre agent, odată cu îndepărtarea graduală de paradigma orientării spre obiect, dominată în prezent.

Sistemele multiagent S.M.A se folosesc ca elemente fundamentale în domeniul **Inteligenţei Artificiale**. De exemplu sistemul EUME (Multimedia Ubiquituos Environment for Education) a fost dezvoltat la *Universitatea Santiago de Compostela* din Spania în ideea de a eficientiza metodele de învăţare clasică.

X. Vila, A. Schuster, A.Riera [21] au creat o platforma S.M.A în **JADE-S** folosind un protocolul de comunicaţie FIPA-Query în care agenţii sunt grupati pe trei nivele:

1. **Nivelul Clienţi.** Agenţii din acest nivel au la dispoziţie o interfaţă grafică (GUI) necesară pentru a interacţiona cu sistemul.

2. **Nivelul Servicii.** Care conţine agenţi care furnizează servicii pentru clienţi funcţie de resursele sistemului.

3. **Nivelul Resurse.** Care conţine agenţi ce gestionează resursele sistemului.

Pentru crearea acestui sistem ei ş-au creat propria ontologie pe care au denumit-o **EUMEONTO** bazată pe limbajul descriptiv FIPA_Sl. Totodată au extins modelul de securitate **JAVA_Sanbox(**Sun Developer Network) pe care l-au folosit pentru agenţii din sistem astfel:

- Identificarea clienţilor (USER+PASSWORD),
- Autorizarea acţiunilor cerute de agenţii clienţi,
- Corectitudinea datelor de identificare prin semnatură electronică,
- Criptarea si Compresia datelor.

Cartea este o extensie a lucrarii [24] şi o recomandăm celor interesaţi de a aprofunda tehnici de programare orientată pe agent. Orice observaţii privind aceasta carte sunt binevenite pe adresele de e-mail:
`zbc@acm.org, popdaniel31.@yahoo.com`.

Capitolul 1

Scurtă introducere în teoria sistemelor multiagent

1.1 Sisteme multiagent

Observăm cum tehnologia informaţiei şi a comunicaţiilor (TIC) şi, prin aceasta, ansamblul „Pieţii informaţionale" globale se dezvoltă într-un ritm de un dinamism fără precedent. Una din cele mai creatoare ramuri ale TI(Tehnologia Informaţiei) este inteligenţa artificială distribuită bazată pe agenţi (*Agent-Based Computing* [16]). Încă din anii '90, agenţii au început să se structureze într-o paradigmă generală de dezvoltare a aplicaţiilor informatice care tinde să penetreze întreaga TI. Interesul crescând pentru agenţi s-a manifestat inclusiv în UE prin reţelele de excelenţă în domeniu sau/şi prin orientarea spre conceptul de „inteligenţă ambientală". Domeniul este intens transdisciplinar atât ca surse de concepte, modele şi metode cât şi ca potenţial de aplicare şi inovare. Diversitatea şi amploarea ariei de aplicaţii sunt copleşitoare, mai multe subdomenii ale TI s-au dezvoltat sau chiar au apărut (de exemplu, controlul stigmergic) numai ca urmare a răspândirii agenţilor sau a sistemelor multi-agent.

Următoarele definiţii sunt preluate din lucrarea [16].

1.1.1 Concepte elementare

Definiţie 1 ***Agentul*** *este un proces de calcul care implementează funcţiile de autonomie şi comunicare ale unei aplicaţii. Agenţii comunică printr-un limbaj (ACL: Agent Communication Language). Agentul este actorul principal pe o platformă agent (FIPA, 2004).*

Cea mai mare diferenţă între programarea tradiţională orientată pe **obiect** şi cea bazată pe **agent** este libertatea unui agent de a răspunde la o cerere. Atunci când un obiect primeşte un mesaj, adică una din metodele sale este apelată, fluxul de control se mută în mod automat la acea metodă. Atunci când un agent primeşte un mesaj, acesta poate decide dacă trebuie să acţioneze conform acestuia sau nu.

Definiţie 2 ***Platformă agent (AP: Agent Platform)****este o infrastructură fizică în care agenţii îşi pot desfăşura activitatea. Aceasta cuprinde: echipament, sistem de operare, software accesibile agenţilor, componente FIPA de gestiune a agenţilor şi agenţi (FIPA, 2004).*

De observat că agentul definit ca proces de calcul ("Computational process") poate ieşi din cadrul restrâns al inteligenţei artificiale (IA) pentru a deveni celula de bază a întregii TI deoarece, ca proces fizic, poate modela natural şi eficace orice alt proces din lumea reală. Sub aspect funcţional consecinţele sunt profunde întrucât modelarea unui proces tipic din lumea reală implică :

a) desfăşurare în timp real, în paralel cu alte procese (inclusiv alţi agenţi);

b) situare într-un mediu fizic nedeterminist (inclusiv nedeterminismul generat de utilizatori şi Internet);

c) interacţiune complexă cu mediul (inclusiv prin reacţie diversificată şi promptă la stimuli).

Traduse în arhitectura agentului aceste consecinţe impun:

a) o dimensiune temporală explicită (având primitive cel puţin de tipul celor din programarea concurentă clasică);

b) reactivitate eficace (realizată prin întreruperi şi excepţii);

c) implementarea agentului ca entitate de program dinamică şi persistentă (practic, ca fir de execuţie).

O definiţie asemănătoare a agentului, ceva mai convenţională, se dă în „Foaia de parcurs" în domeniul agenţilor, propusă de *AgentLink III (2005)*: sistem de calcul care este capabil de a acţiona flexibil şi autonom în domenii dinamice, impredictibile, de regulă de tip multi-agent.

Definiţie 3 ***Sistem multiagent (SMA)****este un sistem compus din mai mulţi agenţi, în stare ca împreună să împlinească scopuri care sunt greu de atins de un agent individual sau un sistem monolitic.*

Observaţie 4 *Natura exactă a agenţilor este o temă oarecum controversată, se poate afirma că SMA cuprind şi agenţi umani (organizaţiile umane şi societatea în general pot fi considerate ca exemplu de SMA ce pot manifesta auto-organizare şi comportamente complexe chiar când strategiile individuale ale tuturor agenţilor lor sunt simple).*

În consecinţă, SMA este un grup de agenţi software care funcţionează în legătură unul cu altul, ei pot coopera sau rivaliza (sau o combinaţie de cooperare şi competiţie), dar există o infrastructură comună care face ca grupul să fie un "sistem", spre deosebire de a fi doar o mulţime de agenţi autonomi separaţi (SMA poate fi închis sau deschis, distribuit sau nu) *(Agtivity, 2008)*. Astfel, într-un SMA agenţii cooperează sau sunt în competiţie cu alţii pentru a îndeplini o sarcină individuală sau colectivă.

După cum se poate observa pentru cazul conceptului de SMA nu există o definiţie unanim acceptată (FIPA), evită chiar termenul SMA, preferând deocamdată să se refere la "platforma de agenţi" ca infrastructură pentru o colecţie de agenţi), unde perspectivele sunt destul de divergente, făcând dificil un eventual consens în viitorul apropiat:

1.1.2 Perspectiva echipei de agenţi

Ideea de bază este veche, evidentă şi simplă: „întregul este mai tare decât suma părţilor sale" (după formularea dată de *Aristotel*). Astfel, SMA are funcţionalitate care nu este reductibilă la suma funcţionalităţilor agenţilor componenţi: „un SMA poate avea o proprietate chiar dacă agenţii săi nu o au" (*Denzinger, 2008 [24]*). De obicei agenţii ca entităţi componente ale sistemului sunt introduşi în SMA ca indivizi cu arhitectură relativ complexă şi în număr nu prea mare (cel mult de ordinul zecilor sau sutelor). Dacă agenţii sunt identici, SMA este omogen, altfel este eterogen. Aplicaţii care conţin doar câţiva agenţi sunt numite fie SMA (mai ales de către vânzători), fie „sisteme bazate pe agenţi" (*ABS: Agent-Based Systems [7]*). La sistemele care conţin şi agenţi umani (de exemplu, cele cu agenţi de interfaţă) interacţiunea este obiectiv de bază şi este de obicei, complexă. Din această perspectivă o aplicaţie cu câţiva agenţi (inclusiv un utilizator) poate fi considerată SMA, chiar dacă agenţii software interacţionează slab sau deloc.

1.1.3 Perspectiva holistă

Este diametral opusă, în loc ca sistemul să fie clădit din subsisteme este invers, adică SMA este primordial, iar entităţile sunt foarte multe, foarte simple şi (în principiu) identice.

Fundamentul fizic este tot manifestarea sinergiei dar, sub influenţa unor paradigme bazate pe modele biologice care au în comun acumulări masive de entităţi identice (de exemplu, reţele neurale, algoritmi genetici, automate celulare), abordarea teoretică s-a situat implicit în contextul conceptual al sinergeticii. Perspectiva s-a răspândit odată cu paradigma coordonării stigmergice, punând accent conform evidenţei din modelul biologic al furnicarului pe autoorganizare ca bază a „sintezei emergente" (*Parunak et. Al.,[17]*) (când se consideră că sistemul nu preexistă organizării, se foloseşte şi termenul „autopoieză").

1.1.4 Perspectiva ingineriei programării orientate spre agent

Odată cu dezvoltarea internetului şi a *WWW (World Wide Web)* după 1990, s-a ajuns în situaţia de a avea nevoie de o nouă metaforă pentru „ceea ce face calculatorul": conceptul de prelucrare a informaţiei este înlocuit cu cel de interacţiune. În această metaforă calculul este:

a) ceva ce are loc ca urmare a şi prin comunicarea între entităţi de calcul;

b) o activitate care este mai degrabă intrinsec socială decât individuală, conducând la noi căi de a concepe, proiecta, realiza şi exploata sistemele de calcul. Ca urmare, a apărut „ingineria programării orientate spre agent" (*IPOA, după AOSE: Agent-Oriented Software Engineering*) care transcende IA, cuprinzând întreaga TI.

1.1.5 Discipline conexe

Disciplinele/(sub)domeniile cu care conceptele şi tehnologia SMA se potenţează reciproc sunt (în ordinea abstractizării descrescătoare) *(Luck, M., P. McBurney, C. Priest [16]*):

Filosofie: Filosofia opiniilor şi a intenţiilor, Teoria actelor de vorbire (ca ramură a filosofiei limbajului), Filosofia argumentării (retorică şi alte forme de persuasiune, dialectică, politici), Teoria delegării şi a normelor.

Ştiinţe sociale: Psihologia cognitivă, Etica, Teoria persuasiunii (şi a încrederii), Cognetica, Ergonomia (industrială şi cognitivă), Captologia şi domeniile lor de graniţă se inter-influenţează cu teoriile privind agenţii de interfaţă (de la protoagenţii bazaţi pe caractere biomimetice până la avatari). Trecând de la agenţi la SMA, transdisciplinaritatea se întensifică, implicând din ce în ce mai multe subramuri ale sociologiei (cu problematică variind de la proiectarea organizaţiilor şi marketing, până la luarea deciziilor de grup în condiţii de stress).

Ştiinţe economice:Teoria alocării resurselor, Teoria jocurilor (privind mai ales formalizarea interacţiunilor economice); interacţiunea cu teoriile economice şi cu cercetările operaţionale du-ce la înflorirea ramurii noi a proiectării asistate a licitaţiilor (dezvoltarea coevolutivă a mecanismelor).

Logică: Logica deontică, Logici epistemice şi autoepistemice, Logici dinamice (şi ale proceselor, inclusiv formarea de coaliţii); pe un plan mai general teoriile agentităţii se întrepătrund cu logicile modale, Logicile fuzzy precum şi cu teoriile raţionamentului nemonoton.

Biologie. Metaforele inspirate din biologie precum reţele neurale artificiale, algoritmi genetici (şi programarea evolutivă în general), raţionamentul bazat pe cazuri (întemeiat pe teoria memoriei episodice), automatele celulare, holonii (parţial), coordonarea stigmergică (după modelarea coloniilor de furnici se trece şl la cele mult mai diversificate, cum sunt cele de albine) etc.

1.1.6 Aplicaţii

Aplicaţiile se descriu în ordinea abstractizării descrescătoare, acestea fiind preluate din "Foaia de parcurs" în domeniul acestei tehnologii (*AgentLink III, 2005*).

Agentul în contextul tehnologic actual

Caracteristicile mediilor dinamice şi deschise, cu interacţiuni între sisteme eterogene sugerează că se impun progrese ale modelelor şi paradigmelor de calcul. De aceea, mulţi observatori cred că agenţii reprezintă cea mai importantă paradigmă nouă pentru progresul softului, după orientarea spre obiect" (*AgentLink III, 2005*). Astfel, conceptul de agent a pătruns într-o gamă variată de subdiscipline ale TI, fiind utilizat din trei perspective oarecum diferite în ingineria unor aplicaţii concrete:

S-a introdus conceptele de:

1. **Metaforă de proiectare**, unde agenţii oferă un mod de structurare a aplicaţiilor în jurul unor componente autonome şi comunicante ce permit realizarea unor instrumente şi infrastructuri software adecvate sistemelor complexe în medii deschise.

2. **Sursă de inspiraţie pentru mecanisme şi tehnici** de interacţiune în medii dinamice şi deschise (de exemplu relaţia optimă între proactivitate şi reactivitate,

perceperea altor agenţi din mediu, modelarea utilizatorului, negociere şi cooperare cu alţi agenţi, formarea de coaliţii etc.).

3. **Modele pentru simulare**, SMA oferă modele eficace pentru reprezentarea mediilor dinamice şi deschise din lumea reală (de exemplu, simularea economiilor, societăţilor şi a mediilor biologice) oferind răspunsuri unor probleme complexe, fizice (clădiri inteligente, sisteme de trafic etc.) sau sociale (comerţ electronic,sisteme de informare a conducerii etc.), inobtenabile altfel (de exemplu, modelarea impactului modificărilor climatice asupra populaţiilor biologice sau ale opţiunilor politicilor publice asupra comportamentului social sau economic).

Dezvoltarea rapidă a *WWW* şi a comerţului electronic au dus la dezvoltarea de modele şi tehnici software standardizate destinate sistemelor de calcul distribuit, creând un bogat context pentru dezvoltarea tehnologiilor agent, de la protocoale de comunicaţii de nivel scăzut (*Bluetooth*) la abstracţii de servicii de reţea.

Impactul asupra SMA a fost major şi s-a manifestat pe două căi:

- **Crearea infrastructurilor tehnologice**

Se realizează prin:

a) tehnologii de bază:

- XML(*Extensible Markup Language*) - deşi nu are o semantică interpretabilă la nivel maşină, include tehnici de reprezentare a cunoştinţelor care înlesnesc adnotarea semantică pe *WWW* a documentelor structurate;

- RDF (*Resource Description Format)* - folosit la descrierea şi schimbul de metadate.

b) tehnologii de gestiune a întreprinderilor (*eBusiness*):

- ebXML - ce standardizează specificaţiile de integrare la nivel de întreprindere printr-o infrastructură XML;

- RosettaNet – ce elaborează standarde pentru procese “eBusiness”, oferind o soluţie robustă ce include dicţionare de date, un cadru de implementare şi specificaţii (bazate pe XML) de scheme de mesaje şi de procese de gestiune;

c) Plug & Play universal:

- Jini (o arhitectură deschisă care permite servicii adaptive centrate pe reţea) şi oferă mecanisme simple care permit dispozitivelor să se interconecteze formând o comunitate emergentă în care fiecare dispozitiv oferă servicii altor dispozitive din “comunitate”;

- UPnP (Universal Plug and Play) oferă conectivitate P2P totală de reţea echipamentelor inteligente şi dispozitivelor radio printr-o arhitectură distribuită, de reţea deschisă, pentru a asigura o bună vecinătate în reţea; în plus controlează dispozitivele din reţea şi transferul de date între ele;

d) Servicii de reţea:

- UDDI (Universal Description, Discovery and Integration) elaborează un cadru deschis, independent de platformă, pentru descrierea serviciilor şi identificarea posibilităţilor de afaceri prin Internet;

- SOAP (Simple Object Access Protocol) asigură un mecanism simplu şi convivial de transfer de informaţii structurate şi tipizate între entităţi echivalente, într-un mediu descentralizat şi distribuit, folosind XML.

- WSDL/WS-CDL:

i) WSDL (Web Service Description Language) oferă o gramatică XML pentru descrierea serviciilor de reţea drept colecţii de puncte de comunicaţie finale capabile să facă schimb de mesaje, permiţând astfel automatizarea detaliilor implicate în comunicaţiile aplicaţiilor;

ii) WS-CDL (Web Services Choreo-graphy Description Language) permite definirea unor interfeţe abstracte a serviciilor de reţea, adică nivelul superior al conversaţiilor axate pe un serviciu de reţea.

- Elemente de arhitectură orientată spre agent

Activităţi tipice agenţilor încep să apară în contextul tehnologic descris, inclusiv în demersurile de standardizare ale WWW semantic, ale CORBA şi ale FIPA (de exemplu, sunt definite o seamă de elemente arhitecturale similare celor adoptate acum în specificaţiile W3C pentru arhitectura serviciilor de reţea).

Deşi cercetarea în domeniul tehnologiilor agent are peste un deceniu vechime, numai din 1999 odată cu apariţia tehnologiilor eficiente orientate spre servicii şi calcul omniprezent s-au putut crea şi conecta în reţele sisteme cu adevărat dinamice (ad hoc) fără a face mari investiţii în infrastructura acestora. În speţă, abia odată cu apariţia **calculului grilă** şi a cererilor de soluţii bazate pe servicii de reţea adaptive la scară largă, s-a mărit nevoia de a oferi soluţii viabile problemelor de nivel mai înalt ale comunicării, coordonării şi securităţii.

De remarcat că aplicaţiile făcute posibile de aceste tehnologii devin tot mai agentuale şi tratează probleme tehnice dificile, similare celor vizate de SMA, cum sunt: încredere,

reputaţie, obligaţii, gestiunea contractelor, formare de echipe şi administrarea sistemelor deschise mari.

În general, se vede că dezvoltările tehnologice ample din domeniul calculului distribuit vizează tot mai mult probleme îndelung investigate în cercetările din domeniul agenţilor.

Aici apar două evoluţii:

a) tehnologiile auxiliare se dezvoltă foarte repede; ca urmare, accentul în cercetare s-a mutat de la infrastructură spre aspectele de nivel mai înalt, privind coordonarea şi cooperarea eficace între servicii disparate;

b) un mare număr de sisteme se proiectează folosind aceste infrastructuri noi şi devin tot mai asemănătoare SMA; ca urmare, proiectanţii lor înfruntă aceleaşi probleme conceptuale şi tehnice ca şi în cazul agenţilor.

1.1.7 Subdomenii şi tipuri

Dezvoltarea tehnologiilor agent a avut loc într-un context de orizont mai larg pentru TI. Pe lângă tehnologiile specifice menţionate mai sus, există şi o seamă de tendinţe şi constrângeri care sugerează că agenţii şi tehnologiile agent vor fi vitale. Relevante pentru saltul actual în privinţa folosirii SMA sunt:

WWW semantic (Semantic Web)

Deşi WWW a devenit un mediu esenţial pentru comunicare, cercetare şi comerţ, reţeaua a fost proiectată pentru uz uman, puterea sa fiind limitată de capacitatea de navigare umană. WWW semantic poate fi folosit de calculator adăugând paginilor descrierea conţinutului lor, într-un mod accesibil maşinii.

Printre cerinţele specifice realizării acestui deziderat sunt:

a) descrieri ample ale mediilor şi ale conţinutului pentru a ameliora căutarea şi managementul;

b) descrieri ample ale serviciilor de reţea pentru a permite sau a ameliora identificarea şi asamblarea;

c) interfeţe comune pentru a simplifica integrarea sistemelor disparate;

d) limbaj comun pentru schimbul de informaţii, bogat sub aspect semantic, dintre agenţii software.

Rezultă clar că WWW semantic impune implicare şi efort din partea domeniului calculului bazat pe agenţi, iar cele două domeniii sunt strâns legate. Astfel, WWW semantic oferă un ferment important atât pentru cercetarea fundamentală cât şi pentru o gamă

întreagă de aplicaţii agent care pot fi clădite pe temelia sa.

Servicii WWW şi Calcul orientat spre servicii (*Web Services and Service Oriented Computing*)

Tehnologiile pentru servicii de reţea (*WWW*) asigură un mijloc standard de interoperare între diferite aplicaţii software, care se execută pe o gamă largă de platforme. Specificaţiile acoperă o arie amplă de probleme de interoperabilitate (transmitere de mesaje, securitate şi arhitectură, reunirea serviciilor individual în fluxuri structurate). Standardele (elaborate de organizaţii ca *W3C sau OASIS*) asigură un cadru pentru folosirea serviciilor componente accesibile prin interfeţe HTTP şi XML. Aceste componente pot fi ulterior combinate în aplicaţii cuplate slab care furnizează servicii cu valoare adăugată tot mai complexe.

Într-un sens mai general standardele pentru servicii de reţea servesc drept punct de convergen-ţă potenţială pentru diverse demersuri ca arhitecturi grilă (care se bazează acum tot mai mult pe infrastructuri pentru servicii de reţea), cadre eBusiness (*ebXML, RosettaNet* etc.) şi altele, spre o noţiune mai generală de arhitecturi orientate spre servicii (*SOA: Service-Oriented Architectures*). Aici, sistemele distribuite sunt văzute din ce în ce drept colecţii de componente de prestator sau consumator de servicii, legate între ele prin fluxuri de lucru definite dinamic. De aceea serviciile de reţea pot fi realizate de agenţi care trimit sau primesc mesaje, în timp ce serviciile însele sunt resursele caracterizate de funcţionalitatea oferită. Aşa cum agenţii pot executa sarcini în numele unui utilizator, un serviciu de reţea asigură această funcţionalitate pentru posesorul său (persoană sau organizaţie).

Astfel, serviciile de reţea oferă o infrastructură la cheie, aproape ideală pentru înlesnirea interacţiunilor interagent într-un SMA. Mai mult, această infrastructură este larg acceptată, standardizată şi prevăzută să fie tehnologia de bază dominantă în anii ce vin. Reciproc, o perspectivă orientată spre agent a serviciilor de reţea câştigă teren întrucât mediile (de prestator sau consumator de astfel de servicii) sunt văzute în mod firesc drept sisteme bazate pe agenţi.

P2P (Peer-to-Peer Computing)

P2P acoperă o gamă largă de infrastructuri, tehnologii şi aplicaţii care au în comun o singură caracteristică: sunt proiectate să creeze aplicaţii conectate în reţea unde fiecare nod (sau sistem) este într-un sens echivalent tuturor celorlalte, iar funcţionalitatea aplicaţiei se obţine printr-o interconectare potenţial arbitrară între aceste noduri colegi (peers). În consecinţă lipsa nevoii de a avea componente server centralizate pentru a administra

sistemele P2P le face foarte atractive ca robusteţe, uşurinţă de exploatare, extindere şi întreţinere.

Cele mai cunoscute aplicaţii P2P sunt: aplicaţiile foarte răspândite de partajare a fişierelor (*Gnutella şi Bit Torrent*), cache de conţinut (*Akamai*), lucru în grup (birourile virtuale gen *Groove Networks*) şi aplicaţii de telefonie prin Internet (gen *Skype*). În timp ce majoritatea acestor sisteme se bazează pe protocoale şi platforme private, instrumente complexe ca JXTA (Sun Microsystem) oferă o gamă largă de caracteristici pentru dezvoltarea API-lor P2P (transmitere de mesaje, publicitate pentru servicii, administrarea aplicaţiilor etc.). Tehnologiile P2P sunt în curs de standardizare în cadrul GGF (*Global Grid Forum*), care include un grup de lucru P2P creat de Intel în 2000.

Aplicaţiile P2P au o seamă de caracteristici de agent, aplicând adesea tehnici de autoorganizare pentru a asigura funcţionarea fără întrerupere a reţelei şi bazându-se pe protocoale care stimulează un comportament corect al clienţilor. De exemplu, multe sisteme comerciale de comerţ electronic (de exemplu, *eBay*) au sisteme simple de bonitate care răsplătesc comportamentul benefic sub aspect social. Pe măsură ce sistemele P2P devin mai complexe, devin mai relevante şi tot mai multe tehnologii agentuale. Acestea cuprind, de exemplu: proiectarea mecanismelor de licitaţie pentru a asigura o bază riguroasă de promovare a comportării raţionale a clienţilor din reţelele P2P; tehnici de negociere interagent pentru a ameliora nivelul de automatizare a colegilor în aplicaţiile larg răspândite; abordări tot mai avansate privind încrederea şi reputaţia; aplicarea de norme, reguli şi structuri sociale, precum şi simularea socială, pentru a înţelege mai bine dinamica populaţiilor de agenţi independenţi.

Calcul grilă (Grid Computing)

Grila (the Grid) este infrastructura de calcul de înaltă performanţă pentru demersurile ştiinţifice distribuite de amploare care oferă un mijloc de dezvoltare a aplicaţiilor de *eŞtiinţă* cum sunt, de exemplu cele necesare pentru instalaţia pentru hadroni de la *CERN*, optimizarea proiectării, bioinformatică sau chimie combinatorială. Totuşi, ea oferă şi infrastructura de calcul pentru aplicaţii mai generale care implică prelucrare de informaţii pe scară largă, gestiune de cunoştinţe şi furnizare de servicii. De obicei, sistemele sunt concepute pe diverse paliere ca prelucrare de:

a) date (tratează alocarea resurselor de calcul, planificare şi execuţie);

b) informaţie (tratează reprezentarea, stocarea şi accesul la informaţie);

c) cunoştinţe (tratează modul de achiziţie, regăsire, publicare şi întreţinere a cunoştinţelor).

Infrastructura permite utilizarea integrată, în colaborare, a calculatoarelor de vârf de gamă, reţele, baze de date şi instrumente ştiinţifice gestionate de diverse organizaţii. Aplicaţiile grilă implică adesea cantităţi mari de date şi prelucrare a acestora şi impun adesea o partajare sigură şi trans-organizaţională a resurselor; ca urmare, aplicaţiile nu sunt tratate uşor de infrastructurile de azi ale Internet şi WWW.

Avantajul major al calculului grilă este flexibilitatea sistemul distribuit şi reţeaua pot fi reconfigurate la cerere în diferite moduri după cum se schimbă nevoile, permiţând în principiu o operare TI mai flexibilă şi o utilizare mai eficientă a resurselor de calcul (*Information Age Partnership, 2008*). Se consideră că în timp ce tehnologia se află deja într-o stare în care poate pune în valoare aceste avantaje într-un singur cadru organizaţional, adevărata valoare provine din utilizare trans-organizaţională, prin organizaţii virtuale care impun tratarea problemelor de proprietate, gestiune şi contabilitate într-un cadru de parteneriat de încredere. În termeni economici, astfel de organizaţii virtuale oferă o cale potrivită pentru dezvoltarea de produse şi servicii noi în pieţe cu valoare înaltă; aceasta este în favoarea noţiunii de software centrat pe servicii, care se naşte abia acum din cauza constrângerilor impuse de organizaţiile tradiţionale. După cum se sugerează în *Information Age Partnership (2008)*, viitorul grilei nu constă în asigurarea puterii de calcul ci în asigurarea de informaţie şi de cunoştinţe într-o economie orientată spre servicii. Până la urmă, successul grilei va depinde de standardizarea şi de crearea de produse, iar eforturile în această direcţie sunt deja întreprinse de o serie de producători, printre care Sun, IBM şi HP.

Inteligenţă ambientală (Ambient Intelligence)

Noţiunea de inteligenţă ambientală a apărut în mare măsură prin eforturile Comisiei Europene de identificare a problemelor de cercetare şi dezvoltare în Europa în domeniul IST (*Information Society Technologies*). Având ca ţintă livrarea de servicii şi aplicaţii fără cusur, ideea se bazează pe calculul omniprezent, pe comunicarea omniprezentă şi pe interfeţe utilizator inteligente, ea descrie un mediu de multe dispozitive încuibate şi mobile (sau componente software) care interacţionează pentru realizarea unor scopuri sau activităţi centrate pe utilizator şi sugerează o perspectivă orientată spre componente (o lume în care componentele sunt independente şi distribuite). Se consideră că autonomia, distribuirea, adaptarea, receptivitatea sunt caracteristici cheie ale acestor componente şi în acest sens, ele au caracteristici de agenţi.

Inteligenţa ambientală cere acestor agenţi să poată interacţiona cu mulţi alţi agenţi din mediul ambiant pentru a-şi atinge ţintele. Astfel de interacţiuni au loc între perechi

de agenţi (în relaţie de colaborare sau competiţie unu la unu), între grupuri (pentru a lua decizii consensuale sau pentru a acţiona ca echipă) precum şi între agenţi şi resursele de infrastructură pe care le au mediile lor (de exemplu, depozite de informaţii ample). Interacţiuni ca acestea permit alcătuirea de organizaţii virtuale, unde grupuri de agenţi se reunesc pentru a forma grupuri coerente în stare să atingă obiective generale.

Mediul asigură infrastructura care permite realizarea scenariilor de inteligenţă ambientală. De exemplu, pe de o parte, agenţii care oferă servicii de nivel mai înalt pot fi deosebiţi de infrastructura fizică şi de conectivitatea senzorilor, actuatorilor şi a reţelelor. Pe de altă parte, ei pot fi deosebiţi şi de infrastructura virtuală necesară detectării resurselor, depozitelor ample de informaţii distribuite şi robuste, conectivitatea logică necesară pentru a permite, de exemplu, interacţiuni efective între un număr mare de agenţi şi servicii distribuite.

În privinţa răspândirii este important că expansibilitatea (în speţă, cea a dispozitivelor) sau nevoia de a asigura prezenţa unui număr mare de agenţi şi servicii precum şi eterogenitatea agenţilor şi serviciilor, este facilitată de existenţa unor ontologii adecvate. Tratarea tuturor acestor aspecte va cere eforturi de aflare a soluţiilor de operare, integrare şi vizualizare pentru senzori distribuiţi, servicii ad hoc şi infrastructură de reţea.

Sisteme autonome (*Self Systems and Autonomic Computing [11]).*

Acestea sunt sistemele de calcul capabile să se autoconducă, revenind în atenţie datorită complexităţii crescânde a sistemelor TI avansate precum şi a dependenţei crescânde a societăţii moderne de astfel de sisteme, numite acum auto (self* , pronunţat “self_star*”), asteriscul sugerând că se iau în seamă diverse atribute. Fără a avea încă o definiţie general acceptată, aspecte ale unor astfel de sisteme se referă la proprietăţi ca auto_(referire, organizare, configurare, administrare, diagnoză, modificare, reparare).

Astfel de sisteme abundă în natură, de la nivelul ecosistemelor, trecând prin primatele mari şi până la procesele din celule izolate. Similar, multe sisteme chimice, fizice, economice şi sociale manifestă proprietăţi auto. Astfel, dezvoltarea sistemelor de calcul cu proprietăţi auto se bazează tot mai mult pe cercetare în biologie, ecologie, fizică statistică şi stiinţe sociale. Cercetări recente au încercat să formalizeze unele idei din aceste discipline şi să identifice algoritm şi proceduri care ar putea realiza diverse atribute auto, de pildă în reţele **P2P.**

Sistemele şi reţelele de calcul auto oferă un domeniu de aplicaţie pentru cercetarea şi dezvoltarea tehnologiilor agent precum şi o contribuţie la teoria şi practica agentuală deoarece multe sisteme auto pot fi privite ca implicând interacţiuni între entităţi şi com-

ponente autonome.

Mai concret, ca reacţie la explozia de informaţie, la integrarea tehnologiei în viaţa zilnică şi la problemele complexe de administrare şi operare a sistemelor de calcul asociate acestora, calculul autonom (o abordare specifică a sistemelor auto) se inspiră din funcţia autonomă a sistemului nervos central uman, care controlează funcţii cheie fără o participare conştientă. Propus de IBM în 2003, calculul autonom este o abordare a sistemelor de calcul cu autoadministrare care implică un minim de interferenţă umană. Scopul său este crearea unei reţele de componente complexe de calcul care dau utilizatorilor ceea ce le trebuie, când le trebuie, fără vreun efort conştient mental sau fizic. Printre caracteristicile definitorii ale unui sistem autonom se află:

a) configurarea şi reconfigurarea automată în condiţii diverse (şi impredictibile);

b) optimizarea funcţională, sistemul monitorizându-şi părţile

c) constituente şi reglându-şi procedurile pentru a-şi atinge obiectivele de sistem;

d) detectarea problemele şi să-şi revină din evenimente de rutină sau de excepţie care ar putea genera disfuncţii;

e) acţionarea în armonie cu mediul său actual, adaptându-se pentru a interacţiona optim cu alte sisteme, negociind utilizarea resurselor;

f) funcţionarea într-o lume eterogenă, implementând standarde deschise;

g) acumularea resurselor pentru a reduce discrepanţa dintre scopurile sale (de utilizare) şi îndeplinirea lor fără intervenţia directă a utilizatorului.

Până la urmă, ţinta este realizarea dezideratelor TI: creşterea productivităţii simultan cu minimizarea complexităţii pentru utilizatori. Mesajul de bază care rezultă din această imagine este că împărtăşeşte multe din obiectivele orientării spre agent şi că agenţii oferă o cale de a controla complexitatea sistemelor auto şi autonome (fie ca tehnologie, fie ca model de inspiraţie preluat din lumea SMA naturale).

Sisteme complexe (Complex Systems)

Sistemele software şi tehnologice sunt printre cele mai complexe artefacte umane iar complexitatea lor creşte mereu. Unele din aceste sisteme, cum este Internetul, nu au fost proiectate ci au crescut pur şi simplu organic, fără control centralizat uman sau chiar fără a fi înţelese. Alte sisteme, cum sunt reţelele de comunicaţii globale mobile prin satelit sau sistemele de operare actuale pentru PC, au fost proiectate centralizat dar cuprind atât de multe componente care interacţionează şi atât de multe tipuri de interacţiune încât nicio persoană individuală sau chiar o echipă nu ar putea spera să înţeleagă în detaliu funcţionarea sistemului. Această lipsă de înţelegere ar putea să explice de ce astfel

de sisteme sunt înclinate spre erori, ca de exemplu, în cazul prăbuşirii reţelelor mari de electricitate în America de Nord şi în Italia în 2003.

Mai mult, multe sisteme care ne afectează vieţile implică mai multe decât software. De exemplu, ecosistemul malariei implică entităţi naturale (paraziţi şi ţânţari), oameni, cultură umană şi artefacte tehnologice (medicamente şi tratamente), toate interacţionând în moduri complexe, subtile şi dinamice. Intervenţia într-un astfel de ecosistem, de exemplu, dând o nouă schemă de tratament pentru malarie, poate avea consecinţe neintenţionate şi neprevăzute datorită înţelegerii precare a naturii acestor interacţiuni. Ştiinţa sistemelor adaptive complexe este încă în embrion şi oferă până acum prea puţin în privinţa îndrumării proiectanţilor în ingineria unor sisteme specifice.

Indiferent de faptul că astfel de sisteme adaptive complexe sunt proiectate explicit sau nu, administrarea şi controlul lor au o importanţă vitală pentru societăţile moderne. Tehnologiile agent oferă o cale de a conceptualiza aceste sisteme ca fiind compuse din entităţi autonome în interacţiune fiecare acţionând, învăţând sau e-voluând separat ca răspuns la interacţiuni în mediile lor locale. O astfel de conceptualizare oferă baza pentru simulări realiste ale funcţionării şi comportării sistemelor şi a proiectării proceselor de control şi intervenţie (*Bullock şi Cliff, 2004 [23]*). Pentru sistemele care sunt proiectate centralizat, cum sunt pieţele electronice încărcate pe Internet, tehnologiile agent oferă şi baza pentru proiectarea şi implementarea sistemului însuşi: s-a afirmat că tehnologiile agent oferă o cale importantă pentru a rezolva problema creşterii complexităţii sistemelor software moderne (*Zambonelli şi Omicini, 2004 [23]*), mai ales în ceea ce priveşte calculul omniprezent, inteligenţa ambientală, funcţionarea continuă şi sisteme deschise.

Este natural ca sistemele mari să fie privite din perspectiva serviciilor pe care le oferă şi prin urmare, din perspectiva entităţilor sau a agenţilor prestatori sau consumatori de servicii. Domeniile discutate mai sus reflectă tendinţele şi liniile de forţă pentru aplicaţiile care pot implica de obicei mulţi agenţi şi servicii şi se răspândesc larg într-un mediu distribuit geografic.

Mai important pare că mediile care au fost identificate aici sunt deschise şi dinamice, astfel încât agenţi noi se pot adăuga, iar unii din cei de acum pot pleca. Din această perspectivă, agenţii acţionează în numele posesorilor de servicii, gestionând accesul la servicii şi asigurând îndeplinirea contractelor. Ei acţionează şi în numele consumatorilor de servicii, detectând serviciile, încheind contracte precum şi primind şi prezentând rezultate. În aceste domenii agenţii vor fi ceruţi pentru angajarea în interacţiuni, negocieri, luarea unor decizii proactive în timpul execuţiei (reacţionând totodată la circumstanţele modifi-

cate) precum şi pentru alocarea şi planificarea resurselor către diverse solicitări aflate în competiţie, adresate infrastructurilor şi sistemelor. În speţă, agenţi cu competenţe diferite vor trebui să colaboreze şi să formeze coaliţii de sprijin pentru noi organizaţii virtuale.

Desigur, aceste liniile de forţă nu acoperă toate zonele din domeniul orientării spre agent. De exemplu, există nevoia de sisteme care se pot comporta inteligent şi pot lucra ca parte a unei comunităţi, ajutând sau înlocuind oamenii în medii murdare, neinteresante sau periculoase. Există şi linii de forţă privind interfeţele om-agent, agenţi instruibili, agenţi robotici şi mulţi alţii dar cei identificaţi aici creează un context care va impulsiona probabil întregul domeniu.

1.1.8 Exemple de soluţii comerciale

Vom da în continuare câteva dintre ele:

Acklin BV a elaborat SMA numit KIR care permite confidenţialitatea deosebită necesară în domeniul asigurărilor de maşini prin agenţi care au accesul la sursele de date mediat de agenţii posesorilor acestor date. Pe lângă sporul de productivitate şi de robusteţe, timpul total pentru identificarea clientului şi a reclamaţiei a fost redus de la 6 luni la 2 minute (cea mai mare durată fiind dată de întârzierile din poşta electronică, deoarece din motive de securitate, comunicarea inter-agent se face numai prin acest sistem).

Eurobios a oferit o soluţie de modelare bazată pe agenţi care a permis unei mari firme producătoare de ambalaje să-şi menţină strategia de producţie agilă ("just in time"), reducând cu peste 35% nivelul stocurilor, fără a întârzia livrările şi luând în seamă şi consecinţele modificărilor în structura comenzilor.

NuTech Solutions a elaborat un SMA bazat pe stigmergie (pentru optimizarea căilor de distribuţie) şi algoritmi genetici (pentru optimizarea nivelurilor de producţie) care a dat mari avantaje firmei Air Liquide America.

1.1.9 Instrumente

Pentru că tehnologiile agent sunt vitale pentru inginerie şi pentru conducerea anumitor tipuri de sisteme informatice, cum sunt sistemele grilă şi sistemele pentru inteligenţă ambientală, tehnologiile şi mecanismele descrise mai jos vor fi importante pentru multe aplicaţii, chiar dacă nu sunt etichetate drept sisteme agent. Aceste tehnologii se pot grupa în trei categorii în funcţie de palierul la care se aplică:

1. Organizarea agenţilor care se referă la societăţile de agenţi în ansamblu. Aici sunt cruciale problemele de structură organizaţională, încredere, norme şi obligaţii precum şi autoorganizarea în societăţile de agenţi deschise. Multe din aceste probleme au fost studiate în alte discipline – de pildă, în sociologie, anthropologie sau biologie. Pe rezultatele acestor cercetări se bazează tehnologiile de proiectare, dezvoltare şi conducere a societăţilor de agenţi complexe.

2. Interacţiunea inter-agent care se referă la comunicarea între agenţi – de pildă, tehnologiile privind limbajele, protocoaele de interacţiune şi mecanismele de alocare a resurselor. Multe din problemele rezolvate de aceste tehnologii au fost studiate în alte discipline, cum sunt economia, ştiinţele politice, filosofia sau lingvistica. La fel, pe rezultatele obţinute anterior din astfel de cercetări se bazează dezvoltarea teoriilor de calcul precum şi tehnologiile agent pentru interacţiune, comunicare şi luarea deciziilor.

3. Arhitectura agentului **ce** se referă numai la agenţi individuali de pildă, proceduri pentru raţionament sau învăţare. La acest palier, problemele au fost prima preocupare a inteligenţei artificiale de la început, urmărind crearea de maşini care pot raţiona şi opera autonom. Cercetarea şi dezvoltarea s-au bazat considerabil pe această activitate anterioară astfel încât atenţia se îndreaptă acum spre cele două paliere precedente.Pe lângă tehnologiile la aceste trei paliere, trebuie considerate şi tehnologiile care dau infrastructura şi instrumentele auxiliare pentru sistemele agent, cum sunt limbajele de programare a agenţilor şi metodologiile de ingineria programării.

Problemele au fost prima preocupare a inteligenţei artificiale de la început, urmărind crearea de maşini care pot raţiona şi opera autonom. Cercetarea şi dezvoltarea s-au bazat considerabil pe această activitate, distingem astfel:

1.1.10 Palierul (auto)organizării sistemului

Societăţile de agenţi

Ele sunt organizaţiile de agenţi dinamice care se adaptează pentru a-şi mări avantajele în mediile în care sunt situate evoluează spre societăţi de agenţi dinamice (sau emergente) cum sunt cele sugerate de „Grilă", inteligenţa ambientală şi alte domenii în care agenţi se reunesc pentru a furniza servicii combinate, toate acestea impunând ca agenţii să se poată adapta să funcţioneze eficace în medii incerte sau ostile. S-au iniţiat deja demersuri pentru dezvoltarea unor sisteme care îşi propun această ţintă, fundamentală pentru a pune în valoare puterea paradigmei agentuale; relevanţa sa va rămâne în prim-planul eforturilor de cercetare-dezvoltare până în anii 2015-2020, mai ales în privinţa eforturilor comerciale pentru exploatare. În speţă, crearea de organizaţii de agenţi dinamice (cuprinzând,de exemplu, metode pentru lucrul în echipă, formarea de coaliţii etc.) pentru a trata aspecte ale predicţiilor care se conturează în privinţa Grilei şi a WWW precum şi aspecte ale calculului omniprezent va fi crucială.

Factorii sociali în organizarea SMA(Sisteme Multiagent) vor deveni de asemenea tot mai importanţi în următorul deceniu, pe măsură ce se caută căi de structurare a interacţiunilor într-o lume dinamică, conectată la Internet. Aceasta se referă la nevoia de a atribui în mod adecvat agenţilor roluri, puteri (instituţionale), drepturi şi obligaţii pentru a controla securitatea şi aspectele legate de credibilitate ale SMA la nivel semantic, spre deosebire de dezvoltările actuale care le tratează la nivel de infrastructură. Aceşti factori sociali pot oferi baza pentru dezvoltarea, de pildă, a metodelor de control al accesului şi pot să asigure că comportamentul este reglementat şi structurat când se confruntă cu medii dinamice în care tehnicile tradiţionale nu mai sunt viabile. Pe lângă metode şi tehnologii adecvate pentru formarea, conducerea, evaluarea, coordonarea şi dizolvarea echipelor de agenţi, vor fi necesare şi tehnologii astfel încât aceste procese să aibă loc automat la execuţie în medii dinamice.

Sisteme complexe şi autoorganizare

Autoorganizarea se referă la procesul prin care un sistem îşi modifică organizarea internă fără control extern explicit, pentru a se adapta modificărilor în obiectivele sale şi în mediul său. Aceasta poate adesea conduce la un comportament emergent care poate fi sau nu dezirabil. Datorită dinamismului şi deschiderii mediilor de calcul actuale, înţelegerea mecanismelor care pot fi folosite pentru a modela, evalua şi obţine autoorganizare şi emergenţă în SMA este o problemă de interes major: dezvolatrea de

sisteme cu autoorganizare funcţionează prin interacţiuni contextuale locale, fără control central. Componentele ţintesc să îndeplinească sarcini individuale simple dar, din interacţiunile lor mutuale, izvorăşte un comportament colectiv complex. Un astfel de sistem îşi modifică structura şi funcţionalitatea pentru a se adapta la schimbări, la cerinţe şi la mediu pe baza experienţei anterioare. Natura dă exemple de autoorganizare (stigmergia, formarea moleculelor, detectarea anticorpilor). Similar, aplicaţiile software actuale implică interacţiuni sociale (cum sunt negocierile şi tranzacţiile) cu entităţi autonome, în medii intens dinamice. Aplicaţiile industriale care urmăresc robusteţe şi adaptabilitate pe baza principiilor autoorganizării, se bucură de un interes tot mai mare în comunitatea software. Acest interes provine din faptul că actualele aplicaţii software trebuie să rezolve cerinţele şi constrângerile provenite din dinamismul sporit, resursele de control sofisticate, autonomie precum şi din descentralizarea inerentă mediilor economice şi sociale de azi. Majoritatea acestor caracteristici şi constrângeri sunt aceleaşi ca şi cele care se pot observa în sistemele din natură care manifestă autoorganizare.

Mecanismele de autoorganizare oferă maşinile decizionale pe baza cărora componentele sistemului prelucrează intrările de la senzorii software şi de echipament pentru a decide cum, când şi unde să se modifice structura şi funcţionalitatea sistemului. Aceasta permite o mai bună adecvare la cerinţele şi mediile actuale, evitând în acelaşi timp defectări sau eşecuri în prestarea de servicii. De aceea trebuie evidenţiate aplicaţiile în care se pot folosi mecanisme existente, cum ar fi **stigmergia** şi să se dezvolte noi mecanisme generice, independente de un anumit domeniu de aplicaţii.

În unele cazuri, mecanismele de autoorganizare au fost modelate folosind abordări bazate pe reguli sau teoria controlului. De asemenea, în multe ocazii acţiunile de autoorganizare au fost inspirate de procese biologice şi din natură, cum este sistemul nervos uman şi comportamentul observat la speciile de insecte care formează colonii. Deşi astfel de abordări ale autoorganizării au fost eficace în anumite domenii, dinamica ambientală şi complexitatea software au limitat aplicabilitatea lor generală. O cercetare mai amplă în modelarea mecanismelor de autoorganizare şi elaborarea sistematică a unora noi este de aceea necesară. Viitoarele sisteme cu autoorganizare trebuie să prelucreze date multidimensionale de la senzori, să continue să înveţe din experienţe noi şi să profite de acţiunile şi de mecanismele de autoorganizare noi, pe măsură ce devin disponibile.

Un fenomen este considerat emergent, dacă nu a fost (pre)definit exact în prealabil. Un astfel de fenomen poate fi observat la nivelul sistemului (macro) şi este caracterizat în general prin originalitate, coerenţă, ireductibilitate a proprietăţilor de macro- nivel la

proprietăţi de micro-nivel şi de neliniaritate. În SMA fenomene emergente sunt comportamentele globale ale sistemului care sunt rezultate colective provenite de la interacţiuni de agent locale şi comportamente de agent individuale. Comportamentele emergente pot fi dezirabile sau indezirabile; construirea unor sisteme cu capacităţi de comportament emergent dezirabil le poate spori robusteţea, autonomia, deschiderea şi dinamismul.

Pentru a obţine comportamentul global de sistem emergent dorit, comportamentele de agent locale şi interacţiunile trebuie să se conformeze unui cadru comportamental dictat de o **teorie a emergenţei**. Din păcate, prea puţine teorii ale emergenţei sunt disponibile acum iar cele existente trebuie ameliorate. De aceea, noile teorii ale emergenţei ar trebui dezvoltate inspirându-se, de exemplu, de la sistemele din natură sau de la cele sociale.

O problemă importantă încă deschisă privind sistemele cu autoorganizare se referă la modelarea contextului şi mediului aplicaţiei. În această privinţă, un aspect cheie este definirea parametrilor de mediu relevanţi care trebuie consideraţi în determinarea evoluţiei structurii şi functionalităţii softului cu autoorganizare. Alte probleme deschise se referă la: cum se poate capta, prelucra şi exploata contextul pentru ajustarea serviciilor furnizate de aplicaţie într-o situaţie dată; cum se pot sincroniza efectele de autoorganizare care apar din folosirea aplicaţiei în contexte diferite; cum să se modeleze eficace preferinţele şi intenţiile utilizatorului; cantitatea de informaţie istorică ce ar trebui înregistrată de sistem şi luată în seamă la determinarea evoluţiei sale în timp.

Credibilitate şi reputaţie

Multe aplicaţii care implică mulţi indivizi sau organizaţii trebuie să ia în seamă relaţiile (explicite sau implicite) dintre participanţi. În plus, s-ar putea de asemenea, ca agenţii individuali să necesite să fie conştienţi de aceste relaţii pentru a lua decizii adecvate. Domeniul credibilităţii, reputaţiei şi structurii sociale încearcă să prindă sub formă electronică noţiuni umane cum sunt credibilitatea, reputaţia, obligaţiile, autorizările, normele, instituţiile şi alte structuri sociale.

Modelând aceste noţiuni, tehnologii pot împrumuta strategii folosite în mod obişnuit de oameni pentru a rezolva conflictele care apar la realizarea de aplicaţii distribuite, cum sunt reglementarea acţiunilor unor populaţii mari de agenţi recurgând la descurajare financiară în cazul încălcării regulilor sociale sau crearea unor mecanisme de piaţă care sunt sigure contra anumitor tipuri de manipulare răuvoitoare. Teoriile se bazează adesea pe intuiţii din diverse domenii cum sunt economia (abordări bazate pe mecanismele de piaţă), alte ştiinţe sociale (legi sociale, putere socială) sau matematică (teoria jocurilor, mai ales proiectarea mecanismelor de interacţiune).

Aspectul complementar acestei perspective sociale privnd reputaţia şi normele este preocuparea tradiţională în privinţa securităţii. Deşi aplicaţiile agentuale folosite curent asigură adesea o securitate bună, în cazul agenţilor acţionând autonom în numele posesorului lor trebuie luaţi în seamă diverşi factori suplimentari. În particular, orice fel de colaborare, mai ales în situaţii când calculatoarele acţionează în numele unor utilizatori sau organizaţii va avea succes numai dacă există încredere. Asigurarea acestei încrederi cere, de exemplu, recurgerea la: mecanisme pentru reputaţie care să evalueze comportamentul anterior; norme (sau reguli sociale) şi impunerea de sancţiuni; contracte electronice care să reprezinte acorduri.

În timp ce asigurarea/garanţia se referă în primul rând la integritatea sistemului, securitatea priveşte protecţia contra entităţilor răuvoitoare: împiedicarea atacatorilor potenţiali de a exploata mecanismele de autoorganizare care alterează structura şi comportamentul sistemului. În plus, un sistem cu autoorganizare trebuie să-şi protejeze nucleul de atacuri. Diverse mecanisme de securitate bine studiate sunt disponibile în acest scop, cum este în-criptarea tare pentru asigurarea confidenţialităţii şi autenticităţii mesajelor privind autoorganizarea. Totuşi, cadrul în care astfel de mecanisme pot fi aplicate eficace în sisteme cu autoorganizare mai cere o considerabilă cercetare în continuare.

În plus, rezultatele aplicării abordărilor de autoorganizare şi emergenţă pe perioade de timp lungi duc la griji privind confidenţialitatea şi credibilitatea acestor sisteme şi a datelor pe care le deţin. Domeniile securităţii, confidenţialităţii şi credibilităţii sunt componente critice pentru următoarele faze de cercetare şi dezvoltare a sistemelor deschise distribuite şi, prin urmare, a sistemelor cu autoorganizare. Se cer noi abordări pentru a lua în seamă atât aspectele sociale cât şi pe cele tehnice ale acestei probleme pentru a impulsiona proliferarea de software cu autoorganizare într-o gamă largă de domenii de aplicaţie.

1.1.11 Palierul interacţiunii

Coordonarea este definită în multe feluri dar în forma sa cea mai simplă se referă la a asigura că acţiunile unor actori independenţi (agenţi[18]) într-un mediu sunt, într-un anumit sens, coerente. Dificultatea constă în identificarea mecanismelor care permit agenţilor să-şi coordoneze automat acţiunile, fără nevoia unei conduceri umane, o cerinţă care se regăseşte într-o diversitate de aplicaţii reale. La rândul său, cooperarea se referă la coordonare cu intenţia unui obiectiv comun. Cercetările de până acum au identificat o

gamă uriaşă de tipuri diferite de mecanisme de coordonare şi cooperare, de la cooperare emergentă (care poate apărea fără nicio comunicare explicită între agenţi), protocoale de coordonare (care structurează interacţiunile pentru a lua decizii) şi medii de coordonare (sau depozite de date distribuite care permit comunicarea asincronă a scopurilor, obiectivelor sau a altor date utile), până la planificare distribuită (care ia în seamă acţiunile posibile şi probabile ale agenţilor în domeniu.

Negocierea apare atunci când agenţii dirijaţi prin scop într-o societate multi_agent au de regulă scopuri conflictuale; cu alte cuvinte, nu toţi agenţii pot fi în stare să-şi satisfacă simultan scopurile individuale. Aceasta se poate întâmpla, de exemplu, în privinţa resurselor în competiţie sau cu cereri multiple privind timpul şi atenţia unui agent. În astfel de circumstanţe, agenţii vor trebui să intre în negocieri unul cu altul pentru a rezolva conflictele.

Ca urmare, s-au dedicat eforturi considerabile protocoalelor de negociere, metodelor de alocare a resurselor şi procedurilor de repartiţie optimală. Aceste lucrări s-au bazat pe de o parte pe idei din ştiinţa calculatoarelor şi inteligenţă artificială iar pe de altă parte pe idei din ştiinţe socio-economice. De exemplu, un obiectiv tipic al alocării de resurse multi-agent este să se găsească o alocare care e optimală în raport cu o metrică adecvată care depinde, într-un fel sau altul, de preferinţele agenţilor individuali din sistem. Multe concepte studiate în teoria alegerii sociale pot fi utilizate pentru a evalua calitatea alocării de resurse. O importanţă deosebită o au concepte cum sunt lipsa de invidie ("envy-freeness") şi echitabilitatea ("equitability") care pot fi utilizate pentru a modela aspecte de imparţialitate ("fairness") (*Endriss şi Maudet, 2004 [24]*). Aceste concepte sunt relevante pentru o gamă largă de aplicaţii (de pildă, exploatarea echitabilă şi eficientă a resurselor de satelit de observare *GPS*).

În timp ce multe din lucrările recente privind alocarea resurselor s-a concentrat pe abordări centralizate, în speţă licitaţii combinatoriale (*Cramton, Shoham şi Steinberg , 2006 [24]*), multe aplicaţii sunt modelate mai natural cu alte cuvinte autentic distribuite sau sisteme P2P unde alocarea apare ca o consecinţă a unei secvenţe de paşi de negociere locală (*Chevaleyre, 2005 [24]*). Abordarea centralizată are avantajul că necesită numai protocoale de comunicaţie relativ simple. În plus, progresele din anii '90 în privinţa proiectării unor algoritmi puternici pentru licitaţii combinatoriale au avut un impact puternic asupra comunităţii de cercetare. O nouă problemă în domeniul alocării resurselor multi-agent este transferarea acestor tehnici spre structurile de alocare distribuită a resurselor, care sunt importante nu numai în cazurile unde ar putea fi greu

de găsit un agent care ar putea lua rolul de organizator de licitaţii (de exemplu, luând în seamă capacităţile sale de calcul sau credibilitatea sa), ci oferă şi un banc de probă pentru o gamă largă de tehnici bazate pe agenţi. Pentru a-şi atinge întregul potenţial, alocarea distribuită a resurselor cere pe mai departe cercetare fundamentală în domenii ca protocoalele de interacţiune între agenţi, strategii de negociere, proprietăţi formale (de exemplu, de teoria complexităţii) ale structurilor de alocare a resurselor şi ale proiectării algoritmilor distribuiţi precum şi o nouă perspectivă a ceea ce înseamnă "optimal" într-o ambianţă distribuită.

Probabil că şi alte tehnici de negociere vor deveni tot mai răspândite. De exemplu, negocierea unu la unu, cu mai multe atribute sau parametri pentru a stabili înţelegeri la palierul serviciului între furnizorii şi consumatorii de servicii va fi crucială pentru mediile de calcul orientate spre servicii. Pe lângă abordările derivate din economie şi teoria alegerii sociale în ştiinţe politice, demersuri recente în negocierea bazată pe argumentaţie s-au bazat pe idei din filosofia argumentării şi din psihologia persuasiunii. Aceste eforturi oferă potenţial un mijloc de a permite interacţiuni mai adânci între agenţi decât o fac protocoalele relativ mai simple de licitaţie economică şi mecanismele de negociere. Vor fi necesare eforturi considerable de cercetare şi dezvoltare pentru a crea mecanismele şi strategiile de calcul pentru astfel de interacţiuni, ceea ce va fi probabil un important focar al cercetării sistemelor cu agenţi în deceniul viitor.

Comunicarea interagent implică probleme de cercetare dificile. Una din ele este semioza, întrucât sensul precis al unei aserţiuni depinde de: contextul în care a fost exprimată; poziţia sa într-o secvenţă de aserţiuni precedente; natura aserţiunii (de exemplu, propoziţie, angajare la o acţiune, cerere); obiectele referite în aserţiune (obiect din lumea reală, stare mentală, stare viitoare a lumii etc); identitatea vorbitorului şi cea a ascultătorilor avuţi în vedere. Altă dificultate, poate insurmontabilă, este verificarea semantică: cum se verifică ce vrea să spună un agent când face o aserţiune. Într-un sistem deschis, un agent nu este de regulă în stare să vadă codul intern al altui agent pentru a verifica aserţiunea acestuia; chiar dacă ar fi posibil un agent destul de isteţ ar putea simula oricând orice stare mentală dorită când este inspectat de alt agent.

Un aspect cheie în această privinţă este nevoia de a explora teoriile relevante în domeniu şi a elabora un cadru unificator, în particular, este necesară o teorie formală a limbajelor şi protocoalelor pentru agenţi, astfel încât să se poată studia pe larg proprietăţile lor şi să se poată compara riguros unul cu altul. În plus, pentru o adoptare mai largă a rezultatelor cercetării, sunt necesare progrese spre înţelegerea aplicabilităţii diferitelor

limbaje de comunicare inter-agent precum şi a limbajelor şi protocoalelor de conţinut.

1.1.12 Palierul agentului

Capacitatea de raţionament este un atribut crucial pentru agenţi, dar măsura în care este necesară este determinată de context. În timp ce raţionamentul este important în general, în medii deschise apar unele griji legate de eterogenitatea agenţilor, credibilitate şi responsabilitate legală, tratarea eşecului şi recuperarea din el precum şi schimbările din societate. Trebuie lămurită reprezentarea conceptelor de calcul pentru norme, legislaţie, autorităţi, execuţie ş.a.m.d. care pot întări dezvoltarea şi implementarea organizaţiilor electronice dinamice sau a altor SMA deschise. Similar, demersurile actuale în privinţa formării de coaliţii pentru organizaţii virtuale este limitat, organizaţiile fiind în mare măsură statice. Automatizarea formării de coaliţii poate fi mai eficace decât oamenii în aflarea unor coaliţii mai bune în situaţii complexe şi este necesară, de exemplu, pentru aplicaţiile grilă.

Un facilitator pentru aceasta este negocierea, totuşi, deşi există deja progrese semnificative şi aplicaţii în lumea reală, cercetările în privinţa mecanismelor de negociere care sunt mai complexe decât cele de licitaţii sau teoria jocurilor sunt încă în fază embrionară. De exemplu, cercetări privind mecanismele de argumentare şi strategiile adecvate pentru participanţi sunt de asemenea necesare înainte ca tehnicile de argumentare să se răspândească. În plus, multe organizaţii virtuale vor trebui să ia decizii colective, agregând într-un fel preferinţele sau deciziile individuale ale participanţilor. Cercetările privind aplicarea teoriei alegerii sociale din ştiinţele politice şi din sociologie la societăţi de agent este tot relativ nouă şi trebuie dezvoltate considerabil (vezi mai sus).

Cu toate că tehnologia învăţării este evident importantă pentru SMA deschise şi extensibile, ea este încă la început. Deşi s-au făcut progrese în multe domenii, cum sunt abordările evolutive şi învăţarea prin repetiţie *("reinforcement learning"),* acestea n-au trecut încă în aplicaţii din lumea reală. Motivele se pot afla în dificultatea învăţării precum şi în probleme de extensibilitate şi în încrederea utilizatorului în software autodaptabil. Pe termen mai lung, tehnicile de învăţare vor deveni probabil o parte centrală a SMA iar pe termen mai scurt vor oferi prilejuri de aplicare în domenii care nu sunt critice sub aspectul siguranţei, cum este distracţia interactivă.

1.1.13 Infrastructură şi tehnologii auxiliare

Orice infrastructură pentru execuţia aplicaţiilor cu agenţi cum sunt cele aflate în calculul ambiental şi în cel omniprezent trebuie, prin definiţie să fie longevivă şi robustă. În contextul sistemelor cu autoorganizare problema este mai complicată, iar noi abordări privind înlesnirea evoluţiei infrastructurilor şi a dezvoltării şi actualizării lor la execuţie vor fi necesare. Dată fiind mulţimea potenţial vastă de dispozitive, senzori şi aplicaţii personalizate pentru care SMA şi autoorganizarea sunt aplicabile, această problemă a actualizării este sensibil mai complexă decât s-a întâlnit până acum. Mai general, stratul de mijloc ("middleware") sau platformele pentru interoperabilitatea agenţilor precum şi standardele vor fi cruciale pentru dezvoltarea pe termen mediu a SMA.

Interoperabilitatea

În prezent, majoritatea aplicaţiilor agentuale se află în laboratoare universitare sau comerciale dar nu sunt larg disponibile în lumea reală. Ieşirea din laboratoare va avea loc probabil în următorii zece ani dar pentru agenţii de informare este necesar un grad mult mai mare de automatizare în tratarea gestiunii cunoştinţelor decât este disponibil acum. În particular, aceasta cere noi standarde de reţea care să permită descrierea structurală şi semantică a informaţiei precum şi servicii care fac uz de aceste reprezentări semantice pentru acces la informaţie la nivel mai înalt. Crearea de ontologii, tezaure sau baze de cunoştinţe generale are aici un rol central şi merită cercetări în continuare privind descrieri formale ale informaţiei şi, eventual, o arhitectură de referinţă pentru a facilita serviciile de nivel mai înalt menţionate mai sus.

SMA distribuite care se adaptează la mediu trebuie să se adapteze atât prin componentele agent individuale cât şi coordonat între palierele sistemului (adică aplicaţie, prezentare şi stratul de mijloc) şi platforme. Cu alte cuvinte, interoperabilitatea trebuie să fie menţinută între componente agent posibil eterogene în timpul şi după acţiunile şi rezultatele autoorganizării. Mai mult, componentele agent vin probabil de la furnizori diferiţi, astfel încât vor trebui integrate mecanisme de autoorganizare diferite pentru a satisface cerinţele aplicaţiei. Problema este complicată şi mai mult de diversitatea abordărilor de autoorganizare aplicabile la straturi (paliere) de sistem diferite. În multe cazuri, chiar soluţiile din cadrul aceluiaşi strat sunt adesea incompatibile. Ca urmare, sunt necesare instrumente şi metode de integrare a operării componentelor agent între straturile aceluiaşi sistem, între mai multe sisteme de calcul precum şi între cadre de autoorganizare diferite.

Ingineria programării orientată spre agent

În pofida diverselor limbaje, structuri, medii de dezvoltare şi platforme care au fost citate de *Luck, Ashri şi d'Inverno [16],* implementarea SMA a rămas o problemă complexă. În oarecare măsură, pentru a gestiona complexitatea SMA, comunitatea de cercetare a elaborat un număr de metodologii cu scopul de a organiza dezvoltarea agenţilor. Totuşi chiar dacă, dezvoltatorii de SMA urmează astfel de metodologii în timpul etapelor de proiectare, rămân dificultăţi în faza de implementare, în parte din cauza lipsei de evoluţie atât a metodologiilor cât şi a instrumentelor de programare.

În implementare apar dificultăţi şi din următoarele cauze:

a) lipsa unor instrumente de depanare specializate;

b) deprinderile necesare pentru a trece de la analiză şi proiectare la cod;

c) problemele asociate cu înţelegerea specificului diferitelor platforme agent;

d) înţelegerea naturii a ceea ce e o abordare nouă şi distinctă a dezvoltării de sisteme.

În ceea ce priveşte sistemele deschise şi dinamice, sunt necesare noi metodologii pentru o abordare sistematică a autoorganizării. Aceste metodologii trebuie să fie capabile să asigure sprijin pentru toate fazele ciclului de viaţă ale IPOA, permiţând elaboratorului să plece de la analiza cerinţelor, să identifice aspectele problemelor care trebuie rezolvate pe baza autoorganizării precum şi să proiecteze şi să implementeze mecanismele de autoorganizare în comportamentele agenţilor componenţi. Astfel de metodologii trebuie să cuprindă şi tehnici pentru monitorizarea şi controlul aplicaţiei sau sistemului cu autoorganizare instalat.

În general, sprijinul mediilor de dezvoltare integrate (MDI) pentru dezvoltarea sistemelor cu agenţi este destul de slab, iar instrumentele agentuale existente nu oferă acelaşi nivel de utilizabilitate ca MDI moderne orientate spre obiect. Unul din motivele principale pentru aceasta este cuplajul strâns inevitabil mai înainte dintre MDI agent şi platformele agent, care rezultă din varietatea modelelor, platformelor şi a limbajelor de programare pentru agenţi. Această siuaţie se schimbă totuşi acum, cu o tendinţă sporită spre modelare mai degrabă decât spre programare.

Cu instrumentele existente, SMA generează adesea o cantitate uriaşă de informaţie privind starea internă a agenţilor, mesajele trimise şi acţiunile întreprinse dar nu există încă metode adecvate pentru gestionarea acestei informaţii în contextul procesului de dezvoltare. Aceasta influenţează atât controlul informaţiei generate în sistem cât şi obţinerea acestei informaţii fără modificarea proiectării agenţilor din sistem. Platforme ca JADE oferă facilităţi generale de introspecţie pentru starea agenţilor şi pentru mesaje,

dar impun o arhitectură de agent specifică, ce ar putea să nu fie potrivită pentru toate aplicaţiile. Astfel, sunt necesare instrumente pentru inspectarea oricărei arhitecturi de agent, analog instrumentelor de depanare de la distanţă în MDI orientate spre obiect actuale iar unele din aceste instrumente încep să apară (de exemplu *Botía, López-Acosta şi Gómez-Skarmeta, 2004 [24]*). Extinderea acestora la abordarea altor aspecte legate de depanare privind trăsăturile organizaţionale şi pentru a ţine seama de aspecte rezultând din emergenţa în sisteme cu autoorganizare va fi de asemenea importantă pe termen mai lung. Problema este deja considerabilă acum dar va creşte în importanţă pe măsură ce complexitatea sistemelor instalate va creşte pe mai departe.

Complexitatea intrinsecă a aplicaţiilor de agent cere şi o nouă generaţie de instrument de tip **CASE** pentru a asista proiectanţii aplicaţiei în exploatarea marii cantităţi de informaţie implicată. Aceasta cere să se asigure raţionament la niveluri adecvate de abstracţie, automatizând pe cât posibil procesul de proiectare şi implementare şi permiţând calibrarea SMA instalate prin simulare precum şi verificare şi control în timpul rulării.

Mai general, se simte nevoia să se integreze instrumentele existente în **MDI** şi nu să se pornească de la zero. Acum există multe instrumente de cercetare dar puţine care să se integreze cu medii de dezvoltare generice cum este Eclipse; astfel de dezvoltări ar promova dezvoltarea agenţilor şi ar reduce costurile de implementare, deoare-ce dezvoltarea SMA implică acum costuri mai mari decât în cazul folosirii paradigmelor convenţionale, din cauza lipsei metodelor şi instrumentelor de sprijin.

Următoarea generaţie de sisteme de calcul va cere probabil un număr mare de componente care interacţionează, fie că sunt servicii, agenţi sau altceva. Instrumentele actuale lucrează bine cu un număr limitat de agenţi dar, în general, nu sunt încă adecvate pentru dezvoltarea unor sisteme de agenţi de mare anvergură (şi eficiente) şi nici nu oferă facilităţi de dezvoltare, control sau monitorizare în stare să trateze volume mari de informaţie sau să regleze comportamentul sistemului în astfel de cazuri.

De asemenea, sunt necesare metrici pentru softul orientat spre agent: ingineria implică totdeauna o activitate de măsurare iar ingineria programării tradiţională foloseşte deja metode de măsurare larg aplicate pentru a cuantifica aspecte software cum sunt complexitatea, robusteţea şi durata medie între două refuzuri de funcţionare. Cu toate acestea, natura dinamică a sistemelor cu agenţi precum şi comportarea în general nedeterministă a aplicaţiilor agentuale cu autoorganizare condamnă tehnicile tradiţionale de măsurare şi evaluare la inadecvare. Ca urmare, sunt necesare noi măsuri şi tehnici pentru evaluarea şi

clasificarea aplicaţiilor din SMA (cu sau fără autoorganizare), sub aspect atât cantitativ cât şi calitativ.

Limbaje şi medii de programare

Majoritatea cercetărilor în privinţa limbajelor de programare orientate spre agent se bazează pe abordări declarative, mai ales clădite pe logică. Limbajele imperative sunt intrinsec inadecvate pentru a exprima abstracţiile de nivel înalt asociate cu proiectarea sistemelor cu agenţi; totuşi, limbajele de programare orientate spre agent ar trebui să permită integrarea uşoară cu cod (moştenit) scris în limbaje imperative (şi, de fapt, tind spre aceasta). Din perspectivă tehnologică, este importantă şi proiectarea şi realizarea de limbaje orientate spre agent. În prezent, limbajele cu adevărat orientate spre agent (cum sunt cele de tip BDI) sunt limitate şi utilizate mai ales în scopuri de cercetare; cu excepţia unor aplicaţii de nişă, ele rămân neutilizate în practică. Totuşi, în ultimii ani s-a văzut o creştere considerabilă a maturităţii unor astfel de limbaje precum şi îmbunătăţiri majore ale platformelor de dezvoltare şi ale instrumentelor care le sprijină (*Bordini et. al., 2005 [24]*). Cercetările actuale accentuează rolul următoarelor medii de dezvoltare SMA în sprijinirea dezvoltării de SMA complexe; noi principii de programare pentru modelarea şi realizarea trăsăturilor agenţilor; semantica formalizată pentru limbajele de programare pentru a implementa agenţilor comportamente specifice.

Un limbaj de programare pentru SMA ar trebui să respecte principiul separării ariilor de interes (*"separation of concerns"*) şi să ofere construcţii de program dedicate pentru implementarea agenţilor individuali, a organizării, coordonării şi mediului lor. Totuşi, din cauza lipsei de limbaje de programare şi a instrumentelor de dezvoltare pentru agenţi (precum şi din cauza unor preocupări mai fundamentale privind lipsa unei semantici clare pentru agenţi, coordonare etc), construirea SMA este încă o activitate pretenţioasă şi consumatoare de timp.

O problemă cheie în programarea orientată spre agent este definirea şi implementarea unor limbaje cu adevărat orientate spre agent care să integreze concepte din programarea atât declarativă cât şi cea orientată spre obiect, pentru a permite definirea agenţilor în mod declarativ, având însă un sprijin serios ca facilităţi de monitorizare şi depanare. Aceste limbaje trebuie să fie foarte eficiente şi să asigure interfeţe spre limbajele convenţionale existente pentru integrarea uşoară cu cod şi aplicaţii moştenite (*"legacy packages"*). Deşi limbajele pentru agenţi existente tratează deja unele din aceste aspecte, se aşteaptă pe termen scurt alte progrese dar, înainte ca astfel de limbaje să poată fi adoptate de industrie pe termen mediu spre lung, va fi necesară o temeinică verificare practică în situaţii din

lumea reală (mai ales sisteme de mare anvergură).

În afară de limbaje pentru agenţi individuali, sunt necesare şi limbaje pentru programarea de nivel înalt a SMA. În primul rând, nevoia de limbaje expresive, uşor de utilizat şi eficiente pentru coordonarea şi orchestrarea unor componente inteligente eterogene este deja imperativă şi, deşi s-au făcut multe cercetări, dezvoltarea unui limbaj de programare eficace pentru coordonarea unor SMA uriaşe, deschise, extensibile şi dinamice compuse din componente eterogene este o ţintă pe termen mai lung.

Metode formale

Deşi noţiunea unui agent acţionând autonom în lume este simplă intuitiv, analiza formală a sistemelor care conţin mai mulţi agenţi este intrinsec complexă. În speţă, pentru a înţelege proprietăţile sistemelor care conţin mai mulţi actori, sunt necesare tehnici solide de modelare şi de raţionament pentru a prinde posibilele evoluţii ale sistemului. Astfel de tehnici sunt obligatorii, dacă agenţii şi sistemele cu agenţi trebuie modelate şi analizate prin calcul.

Cercetările în domeniul modelelor formale pentru sistemele cu agenţi încearcă să reprezinte şi să înţeleagă proprietăţile sistemelor prin formalisme logice care descriu atât stările mentale ale agenţilor individuali cât şi interacţiunile posibile în sistem. Logicile folosite sunt adesea logici credale sau alte variante modale, împreună cu variante temporale iar astfel de logici necesită algoritmi eficienţi de demonstraţie a teoremelor şi de verificare a modelelor, când se aplică unor probleme de amploare considerabilă. Demersuri recente au folosit formalisme logice pentru a reprezenta proprietăţi sociale cum sunt coaliţiile de agenţi, preferinţe sau proprietăţi legate de teoria jocurilor.

Este evident că tehnici formale cum sunt verificarea modelelor sunt necesare pentru testarea, depanarea şi verificarea proprietăţilor SMA implementate. În pofida progreselor, există încă o nevoie reală de abordare a problemelor care apar din deosebirile dintre sistemele cu agenţi, în raport cu paradigma, limbajele de programare utilizate şi, mai ales, proiectarea comportamentului de autoorganizare şi emergenţă. Pentru aceasta din urmă, ar trebui o paradigmă de programare care să înlesnescă verificarea automată a proprietăţilor sistemului atât funcţionale cât şi non- funcţionale. Aceasta ar duce la nevoia de a certifica componentele agentuale sub aspectul corectitudinii în raport cu specificaţiile (fie alegând componente validate prin tehnici tradiţionale, fie generând automat cod din specificaţii).

În plus, sunt necesare tehnici care să asigure funcţionarea sistemului sub o formă acceptabilă sau sigură în timpul procesului de adaptare, de pildă prin tehnici cum sunt

analiza dependenţelor sau contractelor de nivel înalt.

Simulare

După cum s-a arătat, calculul bazat pe agent oferă o cale de a simula sisteme atât naturale cât şi artificiale, inclusiv sistemele cu agenţi. O astfel de modelare prin simulare se aplică tot mai mult în medicină, politici sociale şi inginerie, ajutând şi la proiectarea, implementarea şi conducerea sistemelor artificiale. Totuşi, pentru a pune plenar în valoare potenţialul modelelor de simulare bazate pe agent (sau pe individ), ar trebui în primul rând să se elaboreze o teorie riguroasă a simulării bazate pe agent. Apar următoarele întrebări fundamentale:

i) Când să se oprească rafinarea unui model de simulare?

ii) Câte iteraţii ale unui model/scenariu de simulare aleatoriu sunt necesare pentru a avea încredere în rezultate?

iii) Cât detaliu trebuie simulat într-un model?

iv) Câtă credibilitate au rezultatele?

v) Cum se poate evita supra-interpretarea rezultatelor cu termeni abstracţi sau vagi?

Întrucât răspunsurile depind probabil de domeniul de aplicaţie, o unică teorie unificată ar putea fi imposibilă. Dar eforturi în acest sens sunt necesare, nu în ultimul rând pentru că în cazul unor decizii importante de politici publice (cum sunt cele care apar din **Protocolul de la Kyoto** privind modificările climatice) se pune tot mai mult bază pe modele de simulare.

Altă problemă majoră se referă la modelele de simulare bazate pe agent care implică agenţi cognitivi şi raţionali. De exemplu, în sistemele economice se ştie de mult că aşteptările unor actori individuali le pot influenţa comportamentul şi astfel proprietăţile globale ale sistemului.

Cum pot fi modelate aceste aspecte anticipative şi reflective ale societăţilor din lumea reală prin modele de simulare bazate pe agent?

Problema devine importantă datorită creşterii rapide a sistemelor cu alocare online de resurse, cum sunt sistemele grilă. Dacă un astfel de sistem cuprinde utilizatori informatici inteligenţi, dintre care mulţi îşi bazează deciziile pe propriile lor modele economice ale operaţiei tip grilă înseşi, atunci sarcina de control se complică imens: afirmaţii şi acţiuni ale adminis-tratorului de sistem pot influenţa opiniile şi intenţiile participanţilor şi pot influenţa astfel operaţiile şi performanţele sistemului. Problema controlului pe această cale a aşteptărilor utilizatorilor este bine ştiută de guvernatorii băncilor centrale, cum este Banca centrală europeană, când încearcă să controleze politica monetară. Teoria şi

practica modelelor de simulare bazate pe agent nu sunt suficient de mature pentru a-i orienta în această sarcină.

Interacţiunea agent-utilizator

În viitoarele sisteme complexe, implicarea umană va deveni probabil mai importantă, deşi aceasta impune explorarea şi înţelegerea mai multor posibilităţi noi cum sunt: autonomia şi improvizaţia (pentru a trata evenimente neprevăzute cum sunt cele cauzate de comportamentul utilizatorilor umani); un limbaj de comunicare interagent standardizat cu o semantică puternică pentru a ghida parţial comportamentul agentului şi a facilita integrarea utilizatorilor umani; modele sociale şi organizaţionale pentru SMA în care programe şi oameni pot interacţiona natural (sisteme hibride). În plus, pe măsură ce softul se adaptează la autoorganizare pentru a se integra în diverse contexte, se creează o nouă clasă de probleme privind interacţiunea cu utilizatorii.

O temă cheie: cum pot interacţiona oamenii cu un software în continuă schimbare?

Alte teme se referă la rostul încercării de a proiecta interacţiuni implicite cu aplicaţii care operează cu intrări indirecte bazate pe senzori şi, în acest caz, cum ar putea migra utilizatorii de la interacţiuni tradiţionale explicite la interacţiuni viitoare implicite. În plus, apar probleme în privinţa autorităţii de decizie, responsibilitate, delegare şi control în cazul sistemelor cu agenţi care acţionează în numele sau în colaborare cu decidenţi umani în sisteme cu iniţiative mixte.

Dacă agenţii sau SMA sunt ele însele responsabile pentru decizii, astfel de probleme devin şi mai dificile.

1.1.14 Tendinţe de asimilare a abordărilor agentuale

În orice domeniu de tehnologie de vârf, sistemele instalate în aplicaţii comerciale sau industriale tind să încorporeze rezultate de cercetare care sunt oarecum în urma vârfului cercetării academice şi industriale. SMA nu fac excepţie, actualele sisteme instalate având trăsături aflate în prototipuri sau în cercetări publicate cu trei până la cinci ani în urmă. Fazele de dezvoltare viitoare ale SMA sunt, inevitabil, doar indicative, întrucât anumite firme vor fi utilizatori de frunte ai tehnologiilor agent împingând aplicaţiile în avans şi exprimă puncte de vedere ale cercetărilor în sensul că aceasta devansează dezvoltarea practică cu până la cinci ani.

Starea actuală

SMA actuale sunt sisteme închise (proiectate de o echipă pentru un mediu de firmă, cu agenţi care au aceleaşi scopuri de nivel înalt într-un singur domeniu). În ciuda încercărilor de standardizare din ultimii ani (în special în cadrul FIPA), limbajele de comunicare şi protocoalele de interacţiune sunt de obicei interne, definite de echipa de proiectare anterior oricăror interacţiuni. Sistemele sunt de regulă expandabile numai în condiţii controlate sau simulate. Abordările de proiectare precum şi platformele de dezvoltare tind să fie ad hoc, inspirate de paradigma agentuală şi nu bazate pe metodologii, instrumente sau limbaje adecvate. Cu toate acestea, acum se pune un accent mai mare pe trecerea metodologiilor din laborator spre mediile de dezvoltare, cu activităţi comerciale având forţă industrială în privinţa tehnicilor şi notaţiilor de dezvoltare. Ca parte a acestui efort, unele platforme vin acum cu propriile lor biblioteci de protocoale şi forţează utilizarea mesajelor standardizate, făcând un pas spre agenda pentru termen scurt.

Pentru viitorul previzibil va rămâne o cerere comercială substanţială pentru SMA închise, din două motive:

a) sunt multe probleme care pot fi rezolvate de SMA fără a recurge la sisteme deschise, ceea ce permite multor firme să facă profit;

b) în probleme care implică mai multe organizaţii, înţelegerile (între persoanele direct interesate) în privinţa obiectivelor sistemelor deschise nu se obţin totdeauna în timp util şi pot apărea şi preocupări în privinţa securităţii unor astfel de sisteme.

Evoluţia pe termen scurt

Sistemele vor fi proiectate din ce în ce mai mult spre depăşirea frontierelor firmei, astfel încât agenţii au mai puţine scopuri în comun, deşi interacţiunile lor se vor referi tot la un domeniu comun iar agenţii vor fi proiectaţi de aceeaşi echipă şi vor partaja acelaşi domeniu de cunoştinţe comun. Se vor folosi din ce în ce mai mult limbaje standard de comunicare inter-agent, cum este FIPA ACL dar protocoalele de interacţiune vor fi mixte (standard şi non-standard).

Aceste sisteme vor putea controla un mare număr de agenţi în medii predeterminate cum sunt cele ale aplicaţiilor tip grilă. Metodologiile, limbajele şi instrumentele de dezvoltare tind să atingă un anumit grad de maturitate iar sistemele sunt deja proiectate peste infrastructuri standard cum sunt, de pildă, serviciile de reţea sau cele tip grilă.

Exemple de obiective de sistem care pot fi abordate includ:

1. planificare automată,

2. coordonarea între departamente diferite ale aceleiaşi firme,

3. grupuri de utilizatori închise privind comerţul electronic,

4. planificarea transportului la scară industrială.

Chiar dacă agenţii care participă în aceste sisteme reprezintă mai multe organizaţii, sistemele şi scheletele de agenţi asociate vor fi dezvoltate de regulă tot de către o firmă dominantă sau de către un consorţiu în numele întregii reţele de afaceri.

Evoluţia pe termen mediu

Pe termen mediu, SMA vor permite participarea agenţilor eterogeni, proiectaţi de echipe diferite. Orice agent va putea participa în aceste sisteme cu condiţia ca comportamentul lor (observabil) să se conformeze unor cerinţe şi standarde afirmate public. Totuşi aceste sisteme deschise vor fi de obicei specifice unor domenii de aplicaţii particulare cum sunt **B2B**, comerţul electronic sau bioinformatica. Limbajele şi protocoalele folosite în aceste sisteme vor fi puse de acord şi standardizate, extrase din biblioteci publice de protocoale alternative care, probabil, vor fi totuşi diferite în funcţie de domeniu. În particular, va fi important ca agenţii şi sistemele să stăpânească această eterogeneitate semantică.

De ajutor va fi folosirea tot mai intensă a unor limbaje de modelare noi, general acceptate : Agent-UML (UML 2.0), care promovează deja folosirea mediilor de dezvoltare integrate şi, cum ar fi de dorit vor lansa un proces de armonizare aşa cum a fost cazul cu UML pentru obiecte. Datorită acestui fapt ne-am focusat în următoarele capitole în prezentarea tuturor aplicaţiilor însoţite de diagrame UML [19] .

Sistemele se vor expanda spre un număr mare de participanţi, deşi de regulă numai în cadrul domeniilor vizate şi folosind tehnici particulare (cum sunt agenţii multidomeniu) pentru a comunica între domenii separate. Dezvoltarea sistemelor se va face prin metodologii orientate spre agent standard, cuprinzând cadre şi modele pentru diferite tipuri de agenţi şi organizaţii. Limbaje de programare şi instrumente orientate spre agent vor fi tot mai folosite, făcând, în oarecare măsură, posibilă utilizarea tehnicilor de verificare formală. Aici au un rol deosebit aspecte semantice privind, de exemplu, coordonarea între agenţi eterogeni, controlul accesului sau încrederea. De asemenea, pentru că aceste sisteme vor fi de regulă sisteme deschise, vor deveni tot mai importante probleme ca robusteţea în faţa unor agenţi răuvoitori sau cu defecte, precum şi aflarea unui compromis adecvat între adaptabilitatea şi predictabilitatea sistemului.

Exemple de astfel de sisteme includ : sisteme de comerţ electronic **B2B** de mare anvergură ce permit participarea oricărui furnizor (în locul unui grup de utilizatori închis) ce

folosesc agenţi care nu se conformează unui anumit cadru. Pentru viitorul mai îndepârtat se prevede dezvoltarea de SMA deschise acoperind mai multe domenii de aplicaţii şi implicând participanţi eterogeni elaboraţi de echipe de proiectare diverse. Agenţii care vor încerca să participe în aceste sisteme vor fi capabili să înveţe comportamentul adecvat pentru participarea în cursul interacţiunii, fără a trebui să-şi dovedească angajamentul înainte de intrare. Selectarea protocoalelor şi mecanismelor de comunicare precum şi a strategiilor se va face automat, fără intervenţie umană. Similar, se vor forma, gestiona şi dizolva automat coaliţii ad hoc de agenţi. Deşi limbajele de comunicare şi protocoalele de interacţiune standard sunt oarecum disponibile de ceva timp, sistemele vor permite acestor mecanisme să emane prin mijloace evolutive din interacţiunile efective ale participanţilor mai degrabă decât să le impună prin proiectare. În plus, agenţii vor putea forma şi re-forma rapid coaliţii dinamice şi organizaţii virtuale şi să urmărească scopuri în continuă schimbare prin mecanisme adecvate de interacţiune pentru cunoaştere distribuită şi acţiune comună. În aceste medii vor apărea probabil fenomene de emergenţă, cu sisteme având proprietăţi (atât bune cât şi rele) care nu au fost imaginate de echipa de proiectare iniţială. SMA vor fi capabile, adaptabile şi competente în faţa unor astfel de medii dinamice, de fapt turbulente, şi vor manifesta multe dintre caracteristicile de conştienţă descrise în vizi-unea calculului autonom. Agenţii şi organizaţiile vor fi considerate componente de nivel înalt ale sistemului, uşor de croit pe măsură şi de antrenat, şi care pot fi combinate pentru a oferi noi componente şi servicii, la fel ca în cazul softului automatizat care se autoasamblează. În această fază, sistemele vor fi expandabile total, adică în sensul că nu vor fi constrânse prin limite arbitrare (privind agenţii, utilizatorii, mecanismele de interacţiune, relaţiile dintre agenţi, complexitatea etc.).

Dificultăţi în asimilarea tehnologiei agentuale

Dezvoltarea deosebită prilejuită de Internet a dus la o nouă înţelegere a naturii calculului, care pune în centrul său interacţiunea. În acest context, paradigma orientării spre agent a căutat să maximizeze adaptabilitatea şi robusteţea sistemelor în medii deschise.

Aici se vede cum o nouă tehnologie poate deveni o putere îngrijorătoare. Confruntându-se cu altă serie de obiective, tehnologiile agent abordează probleme diferite şi aplicaţii diferite de cele ale tehnologiilor obiect. Nu s-au schimbat doar regulile jocului ci este un alt joc. Într-o lume de milioane de calculatoare independente interconectate prin Internet şi, prin aceasta, absorbite în cunoaştere distribuită, o echipă de proiectare de software nu mai consideră că componentele software vor împărtăşi aceleaşi scopuri şi motivaţii sau că obiectivele sistemului vor rămâne invariabile în timp.

De aceea, sistemele trebuie să se poată adapta la medii dinamice, să se configureze, administreze şi întreţine singure şi să înfrunte componente răuvoitoare, impredictibile sau pur şi simplu înclinate spre erori. Puterea paradigmei agentuale este că oferă mijloace de nivel de abstracţie adecvat pentru a concepe, proiecta şi gestiona astfel de sisteme.

În concluzie, problemele generale cu care se confruntă introducerea soluţiilor orientate/bazate spre/pe agenţi provin în special din:

1. lipsa instrumentelor, tehnicilor şi metodologiilor pentru specificarea, dezvoltarea şi administrarea sistemelor cu agenţi;

2. automatizarea activităţilor de specificare, dezvoltare şi administrare a sistemelor cu agenţi - probabil că aspecte cum sunt crearea şi administrarea coaliţiilor de agenţi şi a organizaţiilor virtuale vor fi încă mult timp abordate artizanal, inspirându-se din experienţa specifică domeniului (de pildă, economie, psihologie socială, inteligenţă artificială);

3. integrarea coerentă şi eficientă a componentelor şi trăsăturilor (teorii, tehnologii, infrastructuri);

4. identificarea compromisului adecvat între adaptabilitate şi predictabilitate – soluţionarea problemei emergenţei unor proprietăţi de sistem care nu au fost prevăzute sau dorite, aceasta neffind încă rezolvată nici teoretic şi nici practic;

5. crearea transdisciplinarităţii cu alte ramuri ale ştiinţei calculatoarelor şi cu alte ştiinţe (cum sunt economia, sociologia şi biologia).

La acestea se adaugă nenumărate probleme specifice, dintre care cele mai relevante sunt:

- **Încrederea şi credibilitatea**

Deşi stratul de mijloc (middleware) abordează problema autentificării, tehnicile de verificare şi validare existente nu tratează şi problemele mai dificile de stabilire, observare şi controlare a încrederii într-un sistem dinamic deschis. Sunt necesare tehnici pentru a exprima şi a analiza încrederea şi credibilitatea, la nivel atât individual cât şi social pentru a permite interacţiunea în medii dinamice şi deschise.

- **Formarea şi administrarea organizaţiilor virtuale**

Identificate ca una din contribuţiile cheie ale calculului grilă, organizaţiile virtuale nu dispun încă de proceduri bine definite pentru a decide când să se formeze noi organizaţii virtuale, cum să se admi-nistreze şi cum şi când să se desfiinţeze.

- **Alocarea şi coordonarea resurselor**

Până acum accentul s-a pus pe mecanisme de alocare şi coordonare inspirat din societăţile umane (de exemplu, protocoale de licitaţie) dar puterea de prelucrare şi avantajele de memorie ale dispozitivelor de calcul arată că mecanisme şi protocoale complet noi ar putea fi adecvate pentru interacţiuni automatizate, în particular pentru coordonare şi negociere multi-obiectiv.

- **Negocierea**

Este necesară o bază teoretică solidă teoretică pentru algoritmi şi protocoale de negociere (ce tip de negociere să fie luat considerare şi când). Trebuie şi strategii şi protocoale de negociere eficace, reguli, limbaje pentru exprimarea acordurilor, mecanisme pentru negocierea, implementarea şi analiza acordurilor precum şi încorporarea unor facilităţi pentru argumentare şi pentru exprimarea dezacordurilor.

- **Emergenţă în SMA pe scară largă**

Deşi relativ recente, cercetările privind proprietăţile emergente ale unor astfel de sisteme oferă intuiţii dinspre procesele fizice din lumea reală pentru a înţelege mai bine dinamica unor sisteme artificiale de tot mai mare anvergură care se realizează acum. Această abordare vede SMA pe scară largă ca exemple de sisteme complexe, adaptive, care constitue domeniul noii discipline a ştiinţei complexităţii. Pe măsură ce această ştiinţă se maturizează, focalizarea sa pe macroproprietăţi ale unor entităţi interactante poate influenţa proiectarea, implementarea şi controlul SMA pe scară largă. Abordări din fizică, biologie şi alte zone oferă diverse metode de modelare a sisteme pe scară largă dar nu este clar în ce măsură ele sunt echivalente şi ce oferă fiecare abordare pentru ingineria programării sau controlul sistemului.

- **Teoria învăţării şi a optimizării**

Cu toate că învăţarea şi adaptarea au o lungă tradiţie a cercetării, contexte particulare ridică noi probleme. În sisteme autonome complicate, agenţii se adaptează continuu mediului altor agenţi şi unul altuia, încălcând ipotezele teoriilor de învăţare a unui singur agent

şi ducând potenţial la instabilităţi. Aici, optimizarea care presupune un mediu staţionar eşuează de asemenea, patologic, iar noi metode trebuie dezvoltate. Mai mult, sunt importante şi probleme cum ar fi ce se înţelege prin învăţare într-un context multi-agent şi ce înseamnă învăţare „bună".

- **Metodologiile de proiectare**

Multe din greutăţile din proiectarea de software provin din natura distribuită, multi-actor, a noilor sisteme software şi în schimbarea de obiective impusă ingineriei programării, care a rezultat. Dezvoltarea de metodologii pentru proiectarea şi gestionarea SMA caută să atace aceste probleme extinzând actualele tehnici de ingineria programării pentru a aborda explicit natura autonomă a componentelor precum şi nevoia de adaptabilitate şi de robusteţe a sistemului. Până acum s-au dezvoltat o gamă largă de metodologii, abordând adesea aspecte diferite ale problemei de modelare sau pornind de la idei diferite, deşi nu sunt mijloace clare de a le combina pentru a culege roadele diferitelor abordări. Similar, metodologiile orientate spre agent mai trebuie încă integrate reuşit cu metodologiile dominante din ingineria programării convenţională, acceptând totodată dezvoltări noi în alte zone acute.

- **Provenienţa**

Mediile distribuite de acum (inclusiv grila, serviciile de reţea şi sistemele bazate pe agenţi) suferă de lipsa unor mecanisme de trasare a rezultatelor şi de lipsa unor infrastructuri pentru crearea unor reţele de încredere. Provenienţa permite utilizatorilor să urmărească modul în care s-a obţinut un anumit rezultat identificând servicii individuale şi agregate care au produs un anumit lucru. Din perspectivă atât de cercetare cât şi industrială, obiectul cercetării este proiectarea, formalizarea şi implementarea unei arhitecturi de provenienţă deschisă. Această arhitectură ar trebui să fie expandabilă, sigură, deschisă şi să promoveze interoperabilita-tea.

- **Arhitectura şi compunerea serviciilor**

Se simte nevoia unor arhitecturi de servicii integrate care să asigure temelii robuste pentru comportament autonom, în vederea sprijinirii serviciilor dinamice şi a unor matriţe importante de negociere, monitorizare şi administrare. Aceasta va ajuta aplicarea şi implementarea tehnologiilor agent în sistemele grilă şi în alte domenii. Deşi tehnologiile

serviciilor de reţea definesc convenţii pentru descrierea interfeţelor serviciilor şi a fluxurilor de lucru, ne trebuie mai multe tehnici puternice care, de exemplu, să descrie, să detecteze, să compună, să monitorizeze, să administreze şi să adapteze dinamic mai multe servicii în sprijinul organizaţiilor virtuale. Aceasta va lua probabil forma unor arhitecturi orientate spre agent bazate pe P2P sau pe alte structuri inovatoare

- **Integrarea semantică**

În sisteme deschise, entităţi diferite vor avea modele informaţionale distincte, impunând dezvoltarea unor tehnici care să treacă peste faliile semantice dintre ele. Sunt necesare progrese în domenii cum sunt definirea ontologiilor, concilierea reprezentărilor şi concilierea semantică. Aici problema constă în dezvoltarea unor modele flexibile pentru descriere şi integrare semantică.

1.1.15 Note şi comentarii

După cum s-a văzut, tehnologiile agent se deosebesc de alte tehnologii de programare prin obiectivele lor diferite. Obiectivele acestor tehnologii sunt crearea de sisteme situate în medii dinamice şi deschise, capabile să se adapteze la aceste medii şi să încorporeze în ele componente autonome şi care au propriile lor interese. De aceea, iuţeala cu care tehnologia agent este adoptată de realizatorii de software va depinde, cel puţin în parte, de cât de multe domenii de aplicaţii solicită sisteme cu aceste caracteristici. Considerând domeniile cărora li se dă atenţie de firme care dezvoltă software agentual cum sunt *Agentis, Magenta, Lost Wax sau Whitestein* (printre altele), în prezent principalele zone sunt:

1. logistica,
2. transportul,
3. gestiunea utilităţilor
4. securitatea naţională.

Ceea ce au în comun multe din aceste domenii sunt mai multe persoane sau organizaţii legate într-o reţea, cum ar fi un lanţ de furnizori, care au cerinţe de prelucrare în timp real cu timp de răspuns critic. Cu alte cuvinte, pentru aceste aplicaţii sunt cerinţe atât funcţionale cât şi tehnice, o falie pe care tehnologiile agent pot să o treacă.

Majoritatea noilor tehnologii software au nevoie de instrumente şi de metodologii. Un obstacol fundamental pentru desprinderea tehnologiei agent este actuala lipsă de metodologii mature pentru dezvoltarea de software pentru sisteme bazate pe agenţi. Evident, principiile de bază ale ingineriei programării şi ingineriei cunoştinţelor trebuie să se aplice la dezvoltarea şi implementarea SMA, ca şi în cazul oricărui software. Aceasta este la fel de valabil pentru problemele de expandabilitate, securitate, gestiunea transacţiilor etc., pentru care sunt deja disponibile soluţii. O dificultate majoră a calculului bazat pe agenţi este amplificarea acestor soluţii existente pentru a satisface cerinţele diferite ale noii paradigme, luând în acelaşi timp cât mai mult posibil din metode verificate. De exemplu, dezvoltarea de software agentual trebuie să se inspire din proiectarea sistemelor economice, a sistemelor sociale şi a sistemelor complexe de control al fabricaţiei. În plus, soluţiile existente pentru stratul de mijloc trebuie amplificate cât mai mult posibil iar acest mesaj a fost înţeles: mai multe firme au creat platforme bazate pe software existent şi standard pentru stratul de mijloc, care este cunoscut şi înţeles în domeniul comercial.

În ceea ce priveşte aplicaţiile, vedem deja realizarea de sisteme de tip agent (în domeniile calculului omniprezent, WWW semantic, reţelelor P2P ş.a.m.d.). Pe termen lung, ne aşteptăm să vedem dezvoltarea industrială de infrastructuri pentru crearea unor aplicaţii intens expandabile utilizând agenţi pre-existenţi care trebuie să fie organizaţi sau orchestraţi. Totuşi, înfăptuirea tranziţiei de la laboratorul de cercetare spre aplicaţii industriale în funcţiune reprezintă o adevărată problemă şi va fi important să apară realizări industriale validate ştiinţific pentru implementări şi descrieri care acţionează ca stimulatori atât pentru adoptare în industrie cât şi pentru cercetare în continuare.

Pentru sisteme comerciale şi industriale, tehnologiile agent trebuie să iasă din laborator cu accent pe probleme economice, calitate şi convergenţă cu tehnologii industriale existente sau în creştere mai degrabă decât cu accent pe inovare. Aici, măsurile pentru siguranţă, robusteţe şi calitatea softului tradiţional sunt la fel de importante şi trebuie abordate toate pentru a obţine o acceptanţă mai largă. În particular, sunt necesare soluţii bazate pe agenţi pentru medii distribuite la scara întreprinderii cu cerinţe de dezvoltare riguroase.

Aceasta se poate obţine prin abordări de tranziţie prin care sisteme existente pot fi extinse printr-o prezenţă din ce în ce mai mare a agenţilor în mod insesizabil. Înglobând sisteme moştenite în cadrul unor agenţi autonomi situaţi într-un SMA mai mare este o abordare care s-a încercat, de pildă, la conectarea unor comutatoare de telecomunicaţii noi cu unele vechi în mod insesizabil, permiţând comutatoarelor moştenite sa fie înlocuite

treptat fără dificultăţi majore pentru sistem în ansamblu.

Mai general, adoptarea tehnologiilor agent în mediile economice depinde de cât de repede şi cât de bine se pot conecta aceste tehnologii cu software şi cu metode software existente şi verificate. Tehnologiile agent ar trebui să ţintească spre acele domenii de aplicaţii pentru care se potrivesc cel mai bine, amplificând tehnicile tradiţionale care ar trebui să fie utilizate când agenţii nu sunt aplicabili sau nu sunt adecvaţi. Până la urmă, atingerea acestui scop cere o angajare din partea comunităţilor atât economice cât şi de cercetare pentru a colabora eficace în sprijinirea unor soluţii mai eficiente pentru toţi. Un astfel de dialog are deja loc, nu numai la nivel mondial ci chiar la nivel local, în Sibiu.

1.2 Dezvoltarea Sistemelor MultiAgent

Paşii care trebuie parcurşi pentru dezvoltarea unui sistem multiagent:

1. Identificarea actorilor (utilizatori, agenti, obiecte, fisiere/baze de date),

2. Identificarea interactiunilor (protocoale si mesaje, tip si continut pentru fiecare mesaj) si intemeierea diagramelor de secventa,

3. Identificarea agentilor si comportamentelor lor. (cel mai simplu este sa adaugati cate un comportament pentru fiecare tip de mesaj primit la fiecare agent in parte).

Din cauza caracteristicilor distribuite, autonome şi de cooperare, proiectarea şi implemetarea algoritmilor într-o platformă multiagent ridică o serie de diferită de probleme faţă de cele generate de cele generate de proiectarea şi implementarea într-un mediu orientat pe obiect.

1.2.1 Ingineria SMA

Necesitatea integrării sistemelor informatice (complexe) cu mediul organizaţional accentuează însă complexiatea de proiectare a acestora. Acest lucru se datorează în principal [3]:

a) mediului în care operează sistemul informatic (dinamic, inaccesibil şi nedeterminist);

b) interacţiunilor care au loc între componentele (subsistemele) sistemului informatic şi mediu;

c) specificaţiilor sistemului care sunt dependente de un spectru larg de probleme ce pot fi întânite în mediul organizaţional.

În evoluţia ingineriei programării se observă totuşi o trecere progresivă de la maşina de calcul la domeniul problematicii abordate, de la limbajele procedurale la orientarea obiect, de la utilizarea componentelor software la folosirea unor modele prestabilite (şabloane) de proiectare. Ultima paradigmă în acest lanţ evolutiv o reprezintă orientarea spre agent (*Jennings şi Bussmann, 2003 [24]*). O analiză comparativă între paradigma OO şi cea a OA este tratată pe larg de Bărbat Boldur [2] .

Implicaţiile paradigmei orientării spre agent asupra proiectării sistemelor informatice pot fi analizate prin intermediul tehnicilor clasice de tratare a complexităţii de proiectare a unui produs software:

a) decompoziţia - descompunerea sistemului în entităţi de dimensiuni mai mici, fiecare controlabilă individual;

b) abstractizarea - definirea unor modele simplificate care să evidenţieze unele detalii sau propretăţi în detrimentul altora; şi

c) organizarea - identificare relaţiilor dintre componentele necesare rezolvării problemei.

Decompoziţia

Proiectarea unui sistem informatic constă în realizarea arhitecturii aplicaţiei, prin definirea entităţilor compozite şi a protocoalelor de interacţiune dintre acestea. În orientarea spre agent unitatea de decompoziţie este agentul, entitate care se poate angaja în interacţiuni flexibile, prin intermediul unui limbaj de comunicare de nivel înalt. Dacă programarea orientată pe obiect presupune decompoziţie funcţională , o abordare „fără precedent în lumea naturală" (*Parunak, [17]*), în orientarea spre agent decompoziţia presupune identificarea entităţilor fizice ce alcătuiesc domeniul problemei. În acest fel se permite o reprezentare firească a problematicii abordată, specifică sistemelor naturale. De remarcat că decompoziţia funcţională presupune implicit existenţa unui element de control unic care, pe masură ce sistemul evoluează, creşte vulnerabilitatea sistemului informatic prin tendinţa de a atrage funcţionalităţi adiţionale (*Parunak [17]*). La aceste probleme se adaugă cele de inconsistenţă ale sistemului informatic realizat, sensibil la modificările mediului, deoarece decompoziţia funcţională implică partajarea variabilelor de stare între diferite funcţionalităţi ale sistemului. Spre deosebire, decompoziţia pe baze etologice din orientarea spre agent impune o separare naturală a variabilelor de stare ale sistemului informatic, variabile ce pot fi controlate de agenţi individuali.

Pe de altă parte, descentralizarea reduce complexitatea de control a sistemului informatic prin diminuarea interdependenţelor dintre componente. Ca entităţi active, agenţii încapsulează elemente de control distribuite (cunoştinţe legate de momentul acţiunilor individuale şi de restricţiile ce influenţează modificarea stării interne), care permit amânarea evaluării unui SMA din faza de proiectare în faza de execuţie (acest principiu se poate regăsi în foarte multe domenii precum: managementul calităţii totale, sistemele de producţie flexibilă, programarea extremă etc.). Cum detalierea specificaţiilor unui sistem informatic este extrem de dificilă în cazul unor aplicaţii nestructurate, descrierea lor prematură conduce de cele mai multe ori la „paralizie prin analiză" (*Parunak [17]*). O consecinţă a faptul că agentul are control asupra propriilor interacţiuni, tratarea incertitudinii devine parte implicită în proiectarea agentului. Astfel, scopul sistemului nu este prescris prin specificaţii, ci este tratat continuu prin întreţinerea obiectivelor de proiectare ale acestuia (*Parunak* [17]).

Abstractizarea

În ceea ce priveşte abstractizarea sistemului informatic, se poate observa corespondenţa dintre noţiunea de subsistem şi aceea de organizare a agenţilor care, din perspectiva interacţiunilor sociale (de exemplu: norme, roluri, reguli), presupune existenţa unei dinamici implicite a relaţiilor existente între subsistemele compozite (de exexemplu, există protocoale de interacţiune pentru formarea/destrămarea grupurilor de agenţi). Aceste forme intermediare de organizare sunt esenţiale în proiectarea unui sistem informatic complex, acestea permiţând dezvoltarea în izolare a agenţilor (sau a grupurilor de agenţi), ce sunt integraţi ulterior în sistem. În acest fel, se asigură uniformitatea functionalităţilor sistemului, unde noţiunea de entitate primitivă de abstractizare poate varia în functie de necesităţile de analiză, de la nivelul agenţilor individuali (nivelul micro) la nivelul SMA (nivelul macro).

Organizarea

În afară de avantajele menţionate anterior, paradigma orientării spre agent amplifică însă problema impredictibilităţii SMA ca rezultat al interacţiunii dintre agenţi. Pentru tratarea acestei probleme, metodologiile de proiectare a SMA au adoptat cu preponderenţă modele organizaţionale (*Zambonelli şi Omicini* [23]) care descriu structura (rolurile agenţilor), interacţiunile (relaţiile dintre roluri) şi regulile (restricţionarea interacţiunilor în funcţie de rolurile agenţilor) din cadrul SMA (vezi Figura 1.1)

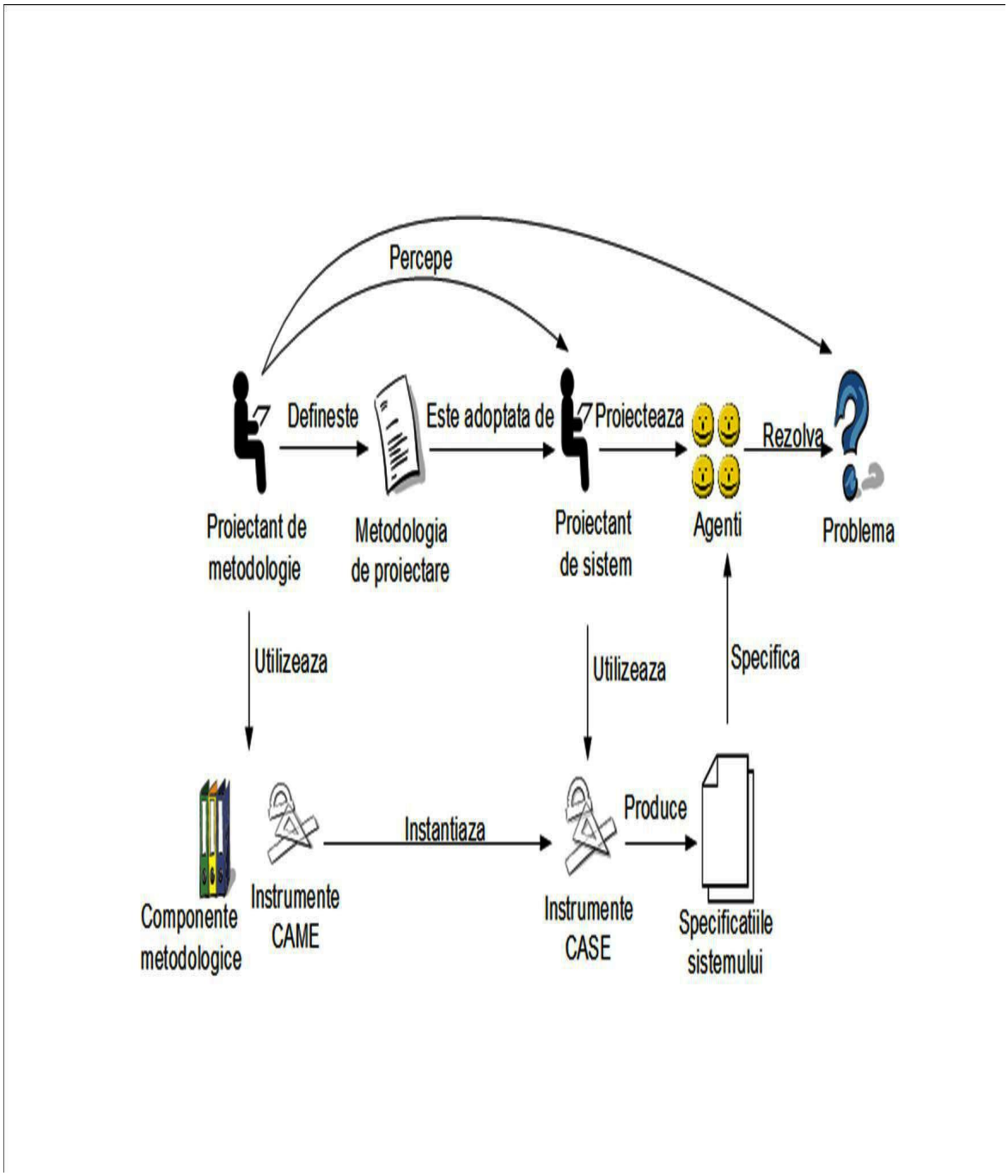


Figura 1.1: Etapele metodologiilor de dezvoltare a unui SMA

1.2.2 Metodologii de dezvoltare

Odată cu apariţia platformelor de dezvoltare a SMA, inedivitabil au apărut şi metodologii specifice care, în mare parte, sunt derivate ale celor utilizate în cazul orientării obiect[23].În practică, analistul/programatorul alege acea medodologie care este cel mai bine sprijinită de instrumentele de analiză, modelare şi dezvoltare disponibile. Fără a intra în detalii descriptive asupra tuturor metodologiilor comentate pe larg în literatura de specialitate, menţionăm cele mai reprezentative [24]:

- GAIA (*Wooldridge, Jennings şi Kinny*, 2000),
- TROPOS (*Tropos*, 2008),
- PROMETHEUS (*Padgham şi Winikoff*, 2004),
- PASSI (*Cossentino*, 2005).

Pentru o perspectivă completă asupra acestora, cititorul interesat poate consulta materialul realizat de *Henderson-Sellers* şi *Giorgini* [10] .

1.2.3 Identificarea agenţilor

Odată descrisă funcţionalitatea unui SMA, pasul următor presupune identificarea agenţilor necesari. În esenţă există patru criterii de identificare a agenţilor care urmează să compună SMA:

1. agenţii trebuie să corespundă entităţilor şi nu funcţiilor pe care sistemul le implementează;
2. agenţii trebuie să fie de dimensiuni reduse;
3. eliminarea controlului descentralizarea prin descentralizarea componentelor sistemului;
4. diversitatea şi generalizarea agenţilor.

Tehnicile de inginerie software clasice s-au bazat pe decompoziţia funcţională, abordare inexistentă în lumea reală.

De remarcat că decompoziţia funcţională conduce la accesarea unor informaţii care fac referinţă la o serie de entităţi distincte ce trebuiesc conduse şi coordonate, modificarea sistemului conducând nemijlocit la reproiectarea acestuia. Pe de altă parte, un sistem format dintr-un număr mare de agenţi specializaţi acoperă un spaţiu mai mare de stări posibile

ale comportamentului sistemului, oferind astfel o funcţionalitate extinsă pe baza comportamentului emergent rezultat în urma interacţiunii dintre aceste componente. În ceea ce priveşte descentralizare ea derivă din evitarea proiectării unui singur punct de dependenţă funcţională, eliminând astfel disfuncţionalitatea generală a sistemului în cazurile critice.

Conform principiilor expuse, se pot contura următoarele specii de agenţi pentru un sistem SMA :

- asistentul personal - care corespunde fiecărui utilizator în parte;

- agentul de resurse - care este ataşat fiecărui program/ instrument colaborativ de implementare a unei acţiuni specifice din compoziţia planului de acţiuni;

- asistentul de planuri - care corespunde planului de acţiuni al fiecărui utilizator implicat într-un anumit proces colaborativ.

Observaţie 5 *Pot exista mai multi asistenţi de planuri pentru fiecare utilizator în parte, deoarece aceştia pot fi implicati la un moment dat în planuri de acţiuni multiple.*

Identificarea speciilor de agenţi (adaptat din *Zamfirescu şi Filip, [25]*):

Entitate	*Agentul*
Decidentul / Utilizatorul	*Asistentul personal* (AA)
Programul / Instrumentul	*Agentul de resurse* (AR)
Planul de acţiuni / Problema	*Asistentul de planuri*(AP)

1.2.4 Descrierea rolurilor agenţilor

După etapa de identificare a agenţilor, următoarea etapă prespune definirea rolurilor pe care fiecare agent îl va avea în cadrul SMA. Această fază intervine în timpul analizei specificaţiilor deoarece ea descrie mai degrabă funcţionalitatea sistemului multi-agent decât structuralitatea acestuia. Rolurile se referă la comportamentul extern observabil al agentului, a.î. orice reprezentare internă a acestuia este aproximativă. Rolul este de asemenea un concept social, astfel încât faza de identificare a rolului este parte a modelului social al agentului. De obicei, identificarea rolurilor se realizează prin utilizarea unor diagrame de secvenţă a celor mai importante scenarii în care agenţii sunt implicaţi.

Asistentul personal e implicat în rolurile de:

1. - *consultant* - sprijină utilizatorului în construirea/elaborarea planului de acţiune (de exemplu, selectarea programelor sau instrumentelor necesare, sugerarea echipei

ideale pentru abordarea problemei în funcţie de expertiza fiecărui utilizator în problematica abordată);

2. - *negociator* - acţionează din partea utilizatorului, ca delegat personal, în stabilirea întâlnirilor şi comunicarea preferinţelor acestuia în utilizarea unor anumite programe sau instrumente;

3. - *observator* - urmăreşte acţiunile utilizatorului în ideea stabilirii şi rafinării profilului acestuia în ceea ce privesc preferinţele şi deprinderile de utilizare a sistemului;

4. - *informator* - anunţă utilizatorul despre modificările asupra elementelor de interes ale acestuia care au survenit între timp.

Pentru **asistentul de planuri** se pot defini două roluri, acela de :

1. - *mediator* – realizează legătura dintre cerinţele problemei (concretizate în planul de acţiuni), profilul utilizatorului şi agentul de resurse pe post de furnizor de instrumente care pot deservi cel mai bine realizarea acţiunilor specifice din planul de acţiune colaborativ;

2. - *coordonator* - coordonează activităţile şi cursul de acţiuni al planului care corespunde unei anumite probleme cu acţiunile celorlalţi utilizatori implicaţi în rezolvarea respectivei probleme.

În ceea ce priveşte **agentul de resurse** acesta va juca rolurile de:

1. - *furnizor* - se ocupă cu gestiunea resursei asociate - în principiu poate fi orice tip de resursă, cum ar fi resurse de comunicare, de gestiune a informaţiei, însă pentru simplificare se referă la programele utilizate în timpul execuţiei unui plan de acţiune, i.e. CAD/CAM, brainstorming, votare, agregare a preferinţelor etc.

2. - *constructor* - va asista unul dintre utilizatori în construirea profilului iniţial al unei resurse pe baza unui model simplificat de reprezentare şi descriere a acesteia (definirea şi reprezentarea caracteristicilor/ funcţionalităţilor programelor/ instrumentelor colaborative ce pot fi utilizate în derularea planului de acţiuni).

1.2.5 Definirea protocoalelor de interacţiune interagent

După etapa de definire a rolurilor agenţilor, urmează definirea protocoalelor de interacţiune interagent. Un protocol de interacţiune interagent descrie cadrul de comunicare, ca o secvenţă de mesaje permisă între agenţi, împreună cu constrângerile asupra contextului de transmisie şi recepţie a acestor mesaje. Protocoalele de interacţiune interagent necesită uneori specificaţii cu o semantică foarte clară a firelor de procesare. În acest sens se pot utiliza diagramele de activitate pentru definirea operaţiilor şi evenimentelor care activează execuţia unor procese interne agentului. Diagramele de activitate diferă de cele de interacţiune prin aceea că oferă o descriere explicită a controlului execuţiei, acestea fiind în special utile pentru protocoale de interacţiune complexe care implică procesare concurentă. Cum diagramele de activitate sunt foarte asemănătoare cu *reţelele Petri (* utilizate în modelare şi simulare procese economice), ele oferă în primul rând o reprezentare grafică a procesului ce facilitează proiectarea comportamentului agentului, iar în al doilea rând, acestea pot reprezenta procese concurente asincrone complexe.

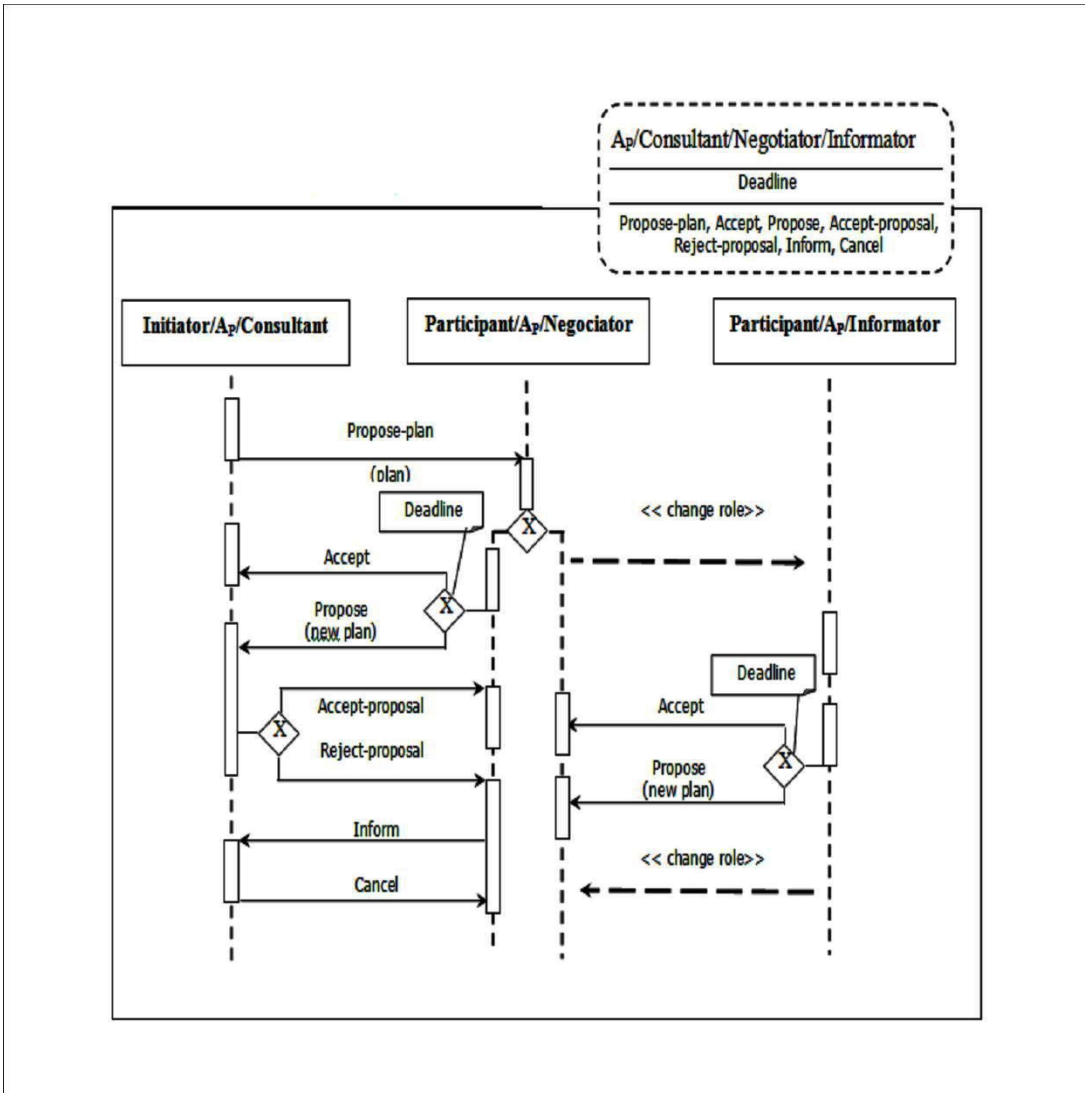


Figura 1.2: Interacţiunea interagent

1.2.6 Definirea comportamentului agenţilor

După definirea protocoalelor de interacţiune interagent urmează detalierea proceselor interne care au loc in fiecare agent în parte prin definirea comportamentului agenţilor. Procesarea internă a agentului este exprimată sub forma unei diagrame de activităţi care verfică agenda de lucru a utilizatorului invitat, verifică competenţele atribuite de către iniţiator, consultă utlizatorul, propune modificări sau nu asupra planului, aşteaptă evaluarea acestora din partea iniţiatorului şi informează iniţiatorul dacă utilizatorul se va implica sau nu în planul de acţiuni.

1.2.7 Limbaje şi medii de dezvoltare

În condiţiile lipsei unor limbaje orientate spre agent, este de dorit ca măcar mediul de programare să fi fost conceput pentru a sprijini dezvoltarea de agenţi. Fiind vorba de un mediu deschis, există posibilitatea eterogenităţii componentelor agent ceea ce sporeşte nevoia flexibilităţii în alegerea mediului de dezvoltare.

Dacă ar fi să facem o comparaţie între cele mai uzitate limbaje de programare, având criterii de comparare generale şi evitând astfel orice referire la programarea agenţilor, trebuie remarcat că acestea:

a) nu au fost proiectate având în vedere agenţii;

b) chiar dacă s-ar fi ţinut seama de agenţi, limbajele sunt prea vechi pentru a reflecta cerinţele de dezvoltare pentru agenţii de azi;

c) criterii privind agenţii vor putea fi considerate numai la compararea mediilor clădite pe aceste limbaje;

d) se evită orice polarizare generată de deosebirile de abordare privind proiectarea SMA.

e) nici sursele de informaţii recente şi bazate pe investigaţii de amploare, utilizate pentru comparare nu focalizează comparaţia pe trăsături legate de agenţi.

De remarcat că între orientarea obiect clasică şi orientarea spre agent există diferenţe fundamentale de execuţie a codului intern. De asemenea, pe lângă diferenţele conceptuale, avem şi diferenţe terminologice între orientarea obiect şi orientarea spre agent.

În ceea ce privesc mediile de programare orientate spre agent, o consultare a enciclopediei Wikipedia (2008) ne-ar sugera următoarele instrumente disponibile: Aglets, BRAHMS, Cougaar, FIPA OS, Jack, JADE (împreună cu o metodologie de utilizare), JavAct, JAS, Jason, LS/TS, MadKit, Repast, SeSAm, Spyse, Swarm, TuCSoN, Visual-

Bots, SPADES, Agent Factory, FreeWalk/Q, SEAS şi ACT-RBot + MRS.

Fără să fie menţionat, multe din aceste instrumente sunt rezultatele unor proiecte de cercetare finalizate de ceva vreme, dezvoltările ulterioare fiind abandonate din varii motive. De asemenea, multe dintre ele au fost special proiectate pentru simularea sistemelor complexe, ele nefiind relevante pentru utilizarea lor în implementări industriale (de exemplu BRAHMS, Swarm, NetLogo, SeSAm) care să respecte standardul FIPA (2008).

Alegerea mediului de programare orientat spre agent diferă de foarte mulţi factori dintre care cel mai important este aplicabilitatea mediului respectiv. Putem găsi instrumente dedicate unor categorii de aplicaţii specifice: sisteme de mare complexitate, mai ales militare (Cougaar, SEAS), simulare (JAS, Repast, SeSAm, Swarm, SPADES,ACT-RBot + MRS), interacţiune socială cu agenţi în oraşe virtuale (FreeWalk/Q), agenţi mobili (Aglets, Agent Factory) şi comunicare/coordonare în SMA bazate pe Internet (TuCSoN). De asemenea foarte multe dintre ele implică familiarizarea cu alte limbaje: AgentSpeak, BRAHMS, JavAct, Jason, LS/TS. Dacă ne-am referi strict la ierarhizarea limbajelor de programare prin indexul TPCI (TIOBE Programming Community Index) care evaluează lunar notorietatea acestora pe baza informaţiilor disponibile pe plan mondial (privind programatorii calificaţi, cursurile şi firmele de software independente care oferă limbajul) este evident că instrumentele cele mai adecvate pentru asimilarea conceptelor de bază ale ingineriei unui SMA trebuie să fie bazate pa Java.

Un alt criteriu a cărui importanţă ar putea conta în viitor este gratuitatea mediului (garantată acum pentru Jade, MadKit şi Spyse) şi conformitatea cu standardele FIPA. La aceste criterii mediul JADE, fundamentat pe Java, rămâne câştigătorul detaşat. De remarcat totuşi prezenţa mediului Spyse, bazat pe un limbaj în vădită ascensiune (Python), ce oferă proiectanţilor de SMA (şi de aplicaţii mai simple bazate pe agenţi) o alternativă de dezvoltare mai apropiată de paradigma orientării spre agent, odată cu îndepărtarea graduală de paradigma orientării spre obiect, dominantă în prezent.

1.3 Exemplificare

Fără a putea trece prin toate aceste etape, majoritatea lor fiind similare celor utilizate în ingineria software orientată obiect, în continuare se vor exemplifica câteva din etapele specifice orientării spre agent:

1. identificarea comportamentului sistemului,
2. identificarea speciilor de agenţi şi a rolurilor acestora,
3. definirea protocoalelor de interacţiune şi specificarea arhitecturii interne a agentilor.

Să prespunem că dorim să realizăm un SMA pentru gestiunea unui plan de acţiuni într-un context multiparticipativ, unde fiecare acţiune a planului implică participarea activă a mai multor utilizatori ce conlucrează în timp real prin utilizarea unui instrument colaborativ (de exemplu, în ingineria concurentă proiectarea unui produs presupune utilizarea succesivă a unor instrumente CAD/CAM de o echipă de proiectare formată din specialişti având competenţe complementare).

Observaţie 6 *Trebuie definită diagrama de caz a unui plan de acţiuni.*

1.3.1 Descrierea comportamentului sistemului

Astfel, descrierea funcţională a comportamentului sistemului se realizează îndeobşte printr-o serie ierarhică de diagrame de caz, în care cea mai abstractă poate servi drept diagramă de context.

În figura următoare [25] este detalită funcţionarea sistemului în ceea ce priveşte crearea unui nou plan de acţiuni care cumprinde toate ativităţile de angajament ale utilizatorilor, activităţile de management al acestuia (adică organizarea, modificarea şi verificarea planului de acţiuni) şi cele de interacţiune cu utilizatorul uman (confirmarea, anunţarea).

Mai departe este detaliat procesul de aprobare al unui plan de către participanţii implicaţi care, la rândul său, cuprinde cele patru activităţi de bază ale oricărui plan de acţiuni. De remarcat că acestea corespund numai fazelor de creare, respectiv de elaborare a unui plan de acţiuni, monitorizarea şi execuţia putând fi descrise în aceeşi manieră.

Reprezentarea este descrisă în **AUML (2008)**, un standard în curs de dezvoltare bazat pe **UML 2.0 (2008)** care încearcă să faciliteze şi să formalizeze ciclul de viaţă al dezvoltării sistemelor bazate pe agenţi. Ca extensii majore în comparaţie cu UML se pot evidenţia:

1. extensiile pentru înlăturarea neajunsurilor semantice ale UML în ceea ce priveşte modelarea obiectelor active autonome, încluzând ambele aspecte (dinamică şi deterministică);

2. extensii pentru reprezentarea protocoalelor de comunicaţie în terminologia actelor de comunicare

În momentul de faţă, propunerile sugerate în AUML se regăsesc (cu unele modificări de reprezentare semantică) şi în standardul **UML 2.0 [19].**

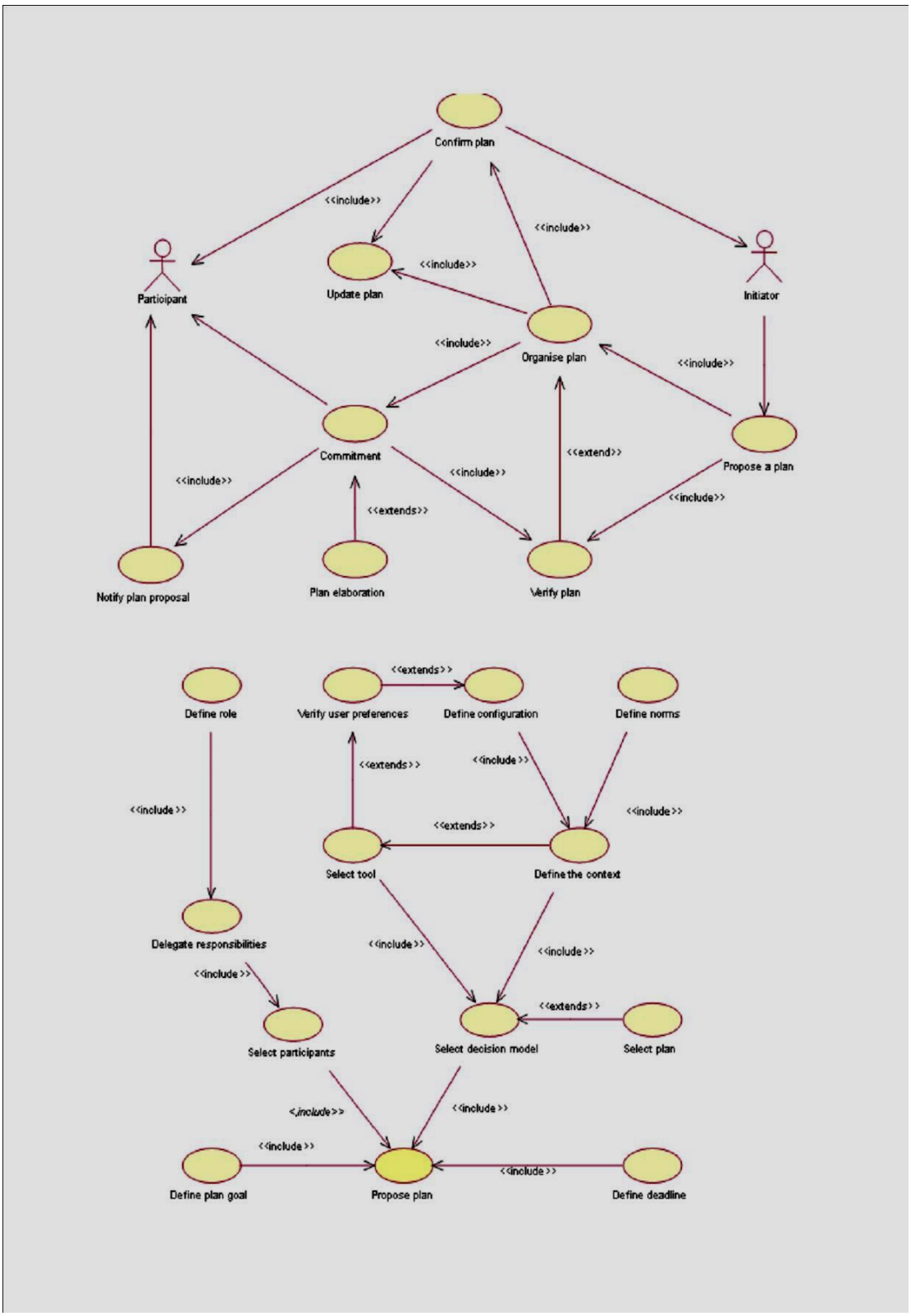


Figura 1.3: Descrierea funcţională a sistemului printr-un set ierarhic de diagrame de caz

1.3.2 Identificarea agenţilor

Odată descrisă funcţionalitatea sistemului, pasul următor presupune identificarea agenţilor necesari. În esenţă există patru criterii de identificare a agenţilor care urmează să compună SMA:

1. agenţii trebuie să corespundă entităţilor şi nu funcţiilor pe care sistemul le implementează;
2. agenţii trebuie să fie de dimensiuni reduse;
3. eliminarea controlului descentralizarea prin descentralizarea componentelor sistemului;
4. diversitatea şi generalizarea agenţilor.

Tehnicile de inginerie software clasice s-au bazat pe decompoziţia funcţională, abordare inexistentă în lumea reală. De remarcat că decompoziţia funcţională conduce la accesarea unor informaţii care fac referinţă la o serie de entităţi distincte ce trebuiesc conduse şi coordonate, modificarea sistemului conducând nemijlocit la reproiectarea acestuia. Pe de altă parte, un sistem format dintr-un număr mare de agenţi specializaţi acoperă un spaţiu mai mare de stări posibile ale comportamentului sistemului, oferind astfel o funcţionalitate extinsă pe baza comportamentului emergent rezultat în urma interacţiunii dintre aceste componente. În ceea ce priveşte descentralizare ea derivă din evitarea proiectării unui singur punct de dependenţă funcţională, eliminând astfel disfuncţionalitatea generală a sistemului în cazurile critice.

Conform principiilor expuse, se pot contura următoarele specii de agenţi pentru exemplul SMA de gestiune a unui plan de acţiuni colaborative :

- - `asistentul personal` - care corespunde fiecărui utilizator în parte;
- - `agentul de resurse` – care este ataşat fiecărui program/ instrument colaborativ de implementare a unei acţiuni specifice din compoziţia planului de acţiuni;
- - `asistentul de planuri` – care corespunde planului de acţiuni al fiecărui utilizator implicat într-un anumit proces colaborativ.

Observaţie 7 *Pot exista mai mulţi asistenţi de planuri pentru fiecare utilizator în parte, deoarece aceştia pot fi implicaţi la un moment dat în planuri de acţiuni multiple.*

1.3.3 Identificarea speciilor de agenţi

Vom prezenta in continuare urmatorul tabel sugestiv [25]:

Entitate	*Agent*
Decidentul /Utilizatorul	*Asistentul personal* (AA)
Programul /Instrumentul	*Agentul de resurse* (AR)
Planul de actiuni /Problema	*Asistentul de planuri*(AP)

Schema AUML a sistemului de gestiune a planului de acţiuni orientat spre agent este prezentată în figura următoare:

Sistemul de gestiune a planului de acţiuni colaborative poate fi considerat în aceste condiţii o compoziţie din toţi agenţii **AA, AR şi AP** care rulează în sistem la un moment dat. În consecinţă, între aceşti agenţi vor interveni schimburi de mesaje ce aparţin protocoalelor de comunicare interagent care servesc la schimbul de informaţii necesare realizării scopului specific fiecărui dintre ei.

1.3.4 Descrierea rolurilor agenţilor

După etapa de identificare a agenţilor, următoarea etapă presupune definirea rolurilor pe care fiecare agent îl va avea în cadrul SMA. Această fază intervine în timpul analizei specificaţiilor deoarece ea descrie mai degrabă funcţionalitatea sistemului multiagent decât structuralitatea acestuia. Rolurile se referă la comportamentul extern observabil al agentului, astfel încât orice reprezentare internă a acestuia este aproximativă. Rolul este de asemenea un concept social, astfel încât faza de identificare a rolului este parte a modelului social al agentului. De obicei, identificarea rolurilor se realizează prin utilizarea unor diagrame de secvenţă a celor mai importante scenarii în care agenţii sunt implicaţi. După o astfel de analiză pentru exemplul sugerat se pot defini următoarele:

Asistentul personal e implicat în rolurile de:

- `consultant` - sprijină utilizatorului în construirea/elaborarea planului de acţiune (de exemplu, selectarea programelor sau instrumentelor necesare, sugerarea echipei ideale pentru abordarea problemei în funcţie de expertiza fiecărui utilizator în problematica abordată);

- `negociator` - acţionează din partea utilizatorului, ca delegat personal, în stabilirea întâlnirilor şi comunicarea preferinţelor acestuia în utilizarea unor anumite programe sau instrumente;

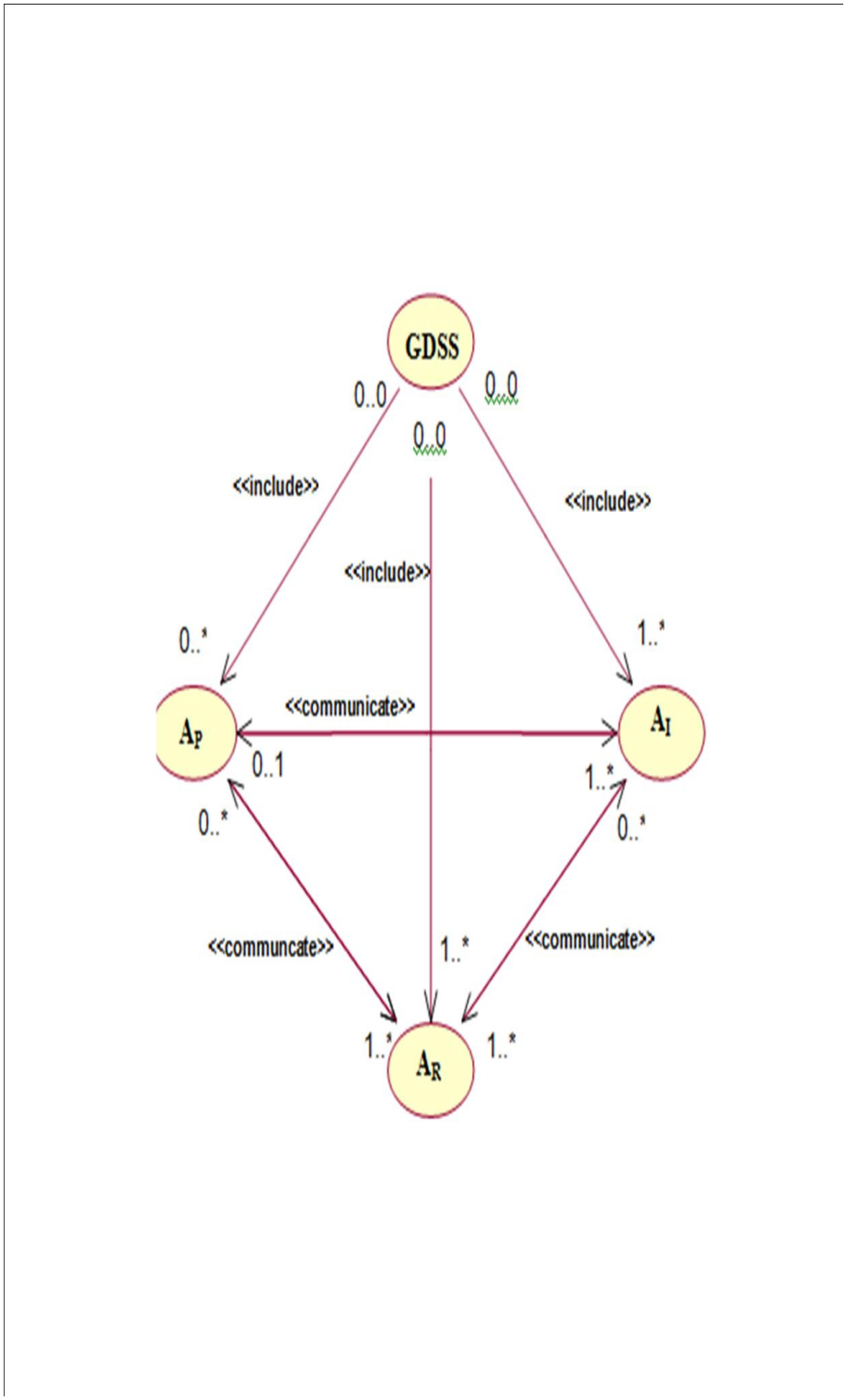


Figura 1.4: Schema AUML a sistemului de gestiune a planului de acţiuni orientat spre agent

- `observator` - urmăreşte acţiunile utilizatorului în ideea stabilirii şi rafinării profilului acestuia în ceea ce privesc preferinţele şi deprinderile de utilizare a sistemului;

- `informator` - anunţă utilizatorul despre modificările asupra elementelor de interes ale acestuia care au survenit între timp.

Pentru **asistentul de planuri** se pot defini două roluri, acela de :

- `mediator` – realizează legătura dintre cerinţele problemei (concretizate în planul de acţiuni), profilul utilizatorului şi agentul de resurse pe post de furnizor de instrumente care pot deservi cel mai bine realizarea acţiunilor specifice din planul de acţiune colaborativ;

- `coordonator` – coordonează activităţile şi cursul de acţiuni al planului care corespunde unei anumite probleme cu acţiunile celorlalţi utilizatori implicaţi în rezolvarea respectivei probleme.

În ceea ce priveşte **agentul de resurse** acesta va juca rolurile de:

- `furnizor` - se ocupă cu gestiunea resursei asociate – în principiu poate fi orice tip de resursă, cum ar fi resurse de comunicare, de gestiune a informaţiei, însă pentru simplificare se referă la programele utilizate în timpul execuţiei unui plan de acţiune, i.e. CAD/CAM, brainstorming, votare, agregare a preferinţelor etc.

- `constructor` – va asista unul dintre utilizatori în construirea profilului iniţial al unei resurse pe baza unui model simplificat de reprezentare şi descriere a acesteia (definirea şi reprezentarea caracteristicilor/ funcţionalităţilor programelor/ instrumentelor colaborative ce pot fi utilizate în derularea planului de acţiuni).

Asistentul personal	*Asistentul de planuri*	*Agentul de resurse*
consultant		
Negociator	*Mediator*	*Furnizor*
Observator	*Coordonator*	*Constructor*
informator		

1.3.5 Definirea protocoalelor de interacţiune inter-agent

După etapa de definire a rolurilor agenţilor, urmează definirea protocoalelor de interacţiune inter-agent. Un protocol de interacţiune inter-agent descrie cadrul de comunicare, ca o secvenţă de mesaje permisă între agenţi, împreună cu constrângerile asupra contextului de transmisie şi recepţie a acestor mesaje.

Atunci când este invocat, **agentul personal**, cu rol de consultant al iniţiatorului, trimite o propunerea cu planul de acţiuni către **agentul personal** cu rol de negociator

al unui potenţial participant la acesta. Asistentul personal al celui din urmă poate alege să răspundă înaintea unui termen limită prestabilit de iniţiatorul planului:

1. fie să-şi asume întreaga responsabilitate decizională pe baza instrucţiunilor generale primite de la utilizator,

2. fie să-şi întrebe în timp real utilizatorul pe care-l deserveşte.

Dacă trebuie să-şi consulte în mod explicit utilizatorul, acesta îşi schimbă rolul din negociator în informator, după care are loc dialogul cu utilizatorul (**decizii de tipul XOR**). Dacă este propus un nou plan de acţiune, initiatorul are şansa fie de a accepta propunerea respectivă, fie de a o respinge.

Atunci când participantul primeşte o acceptare a propunerii, acesta va informa iniţiatorul despre maniera de execuţie a propunerii. În ceea ce priveşte semnificaţia acestor mesaje în contextul sistemului pentru gestiunea planului de acţiuni, termenul limită se referă la momentul în care execuţia respectivului plan trebuie să intre în acţiune. Responsabilitatea asistentului personal se va rezuma numai la verificarea constrângerilor temporale ale agendei participantului şi eventual a programelor sau instrumentelor utilizate. De asemenea, acceptarea unei propuneri se referă la angajamentul participantului în executarea planului respectiv. Astfel de digrame de interacţiune intre-agent sunt definite pentru toţi agenţii şi pentru toate interacţiunile în care sunt implicaţi.

Din perspectiva ingineriei software a SMA, protocolul de interacţiune inter-agent poate fi considerat o entitate de sine stătătoare ce permite agregarea conceptuală a unei secvenţe de interacţiune. În al doilea rând, protocolul de interacţiune poate fi tratat ca un model care poate fi personalizat pentru probleme sau contexte similare.

De remarcat că protocolul nu stipulează nici procedura utilizată de iniţiator pentru a realiza producerea unui nou plan de acţiuni, nici procedura utilizată de participant pentru a răspunde acestei propuneri. În continuare vom prezenta interacţiunea inter-agent în cadrul sistemului de management a planului de acţiuni.

Se observă că protocoalele de interacţiune inter-agent necesită uneori specificaţii cu o semantică foarte clară a firelor de procesare. În acest sens se pot utiliza diagramele de activitate pentru definirea operaţiilor şi evenimentelor care activează execuţia unor procese interne agentului. Diagramele de activitate diferă de cele de interacţiune prin aceea că oferă o descriere explicită a controlului execuţiei, acestea fiind în special utile pentru protocoale de interacţiune complexe care implică procesare concurentă. Cum diagramele de activitate sunt foarte asemănătoare cu reţelele Petri, ele oferă în primul rând

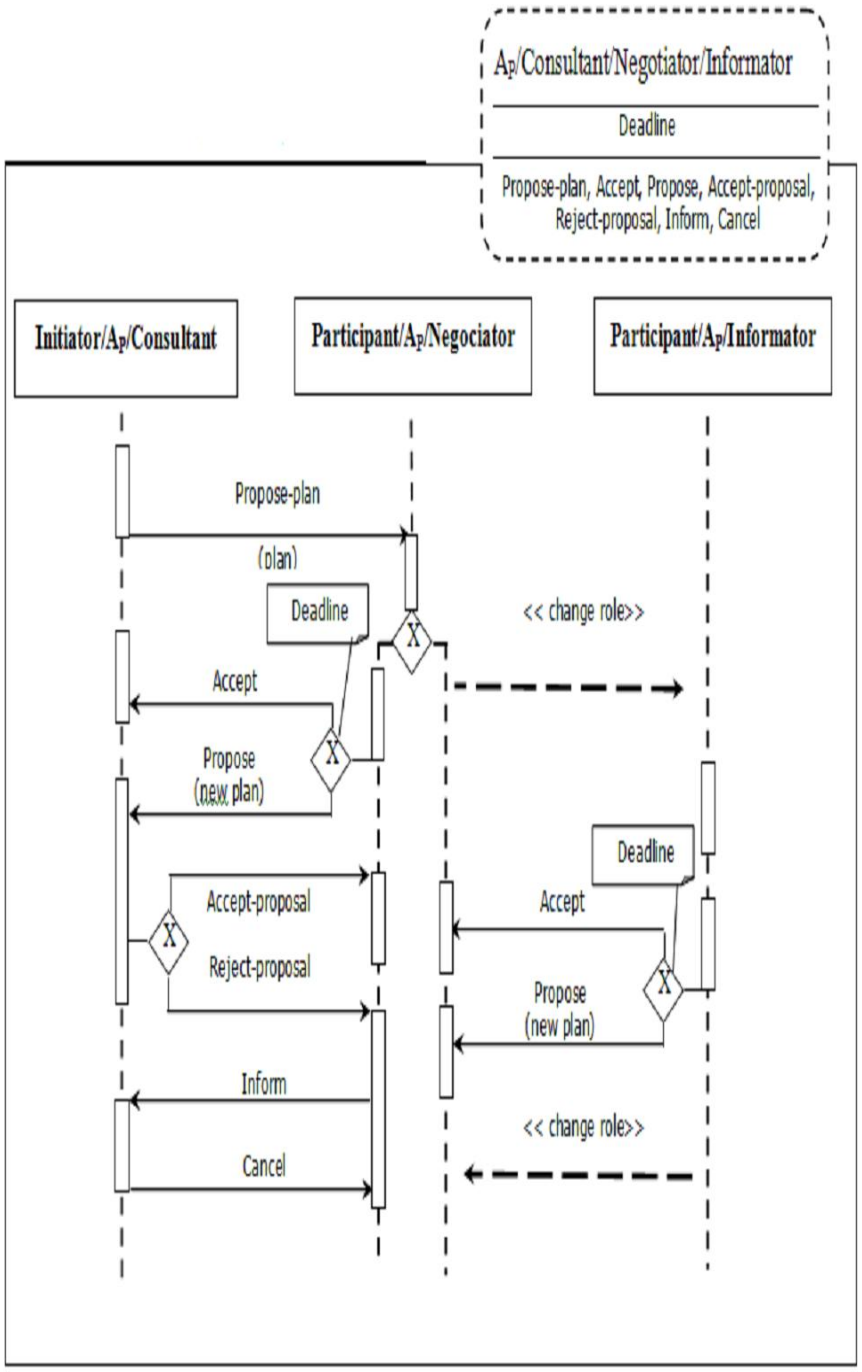


Figura 1.5: Interacţiunea inter-agent în cadrul sistemului de management a planului de acţiuni

o reprezentare grafică a procesului ce facilitează proiectarea comportamentului agentului, iar în al doilea rând, acestea pot reprezenta procese concurente asincrone complexe. În figura următoare vom prezenta comportamentul asistentului personal, având rolul de negociator, în faza de iniţiere a unui plan de acţiuni.

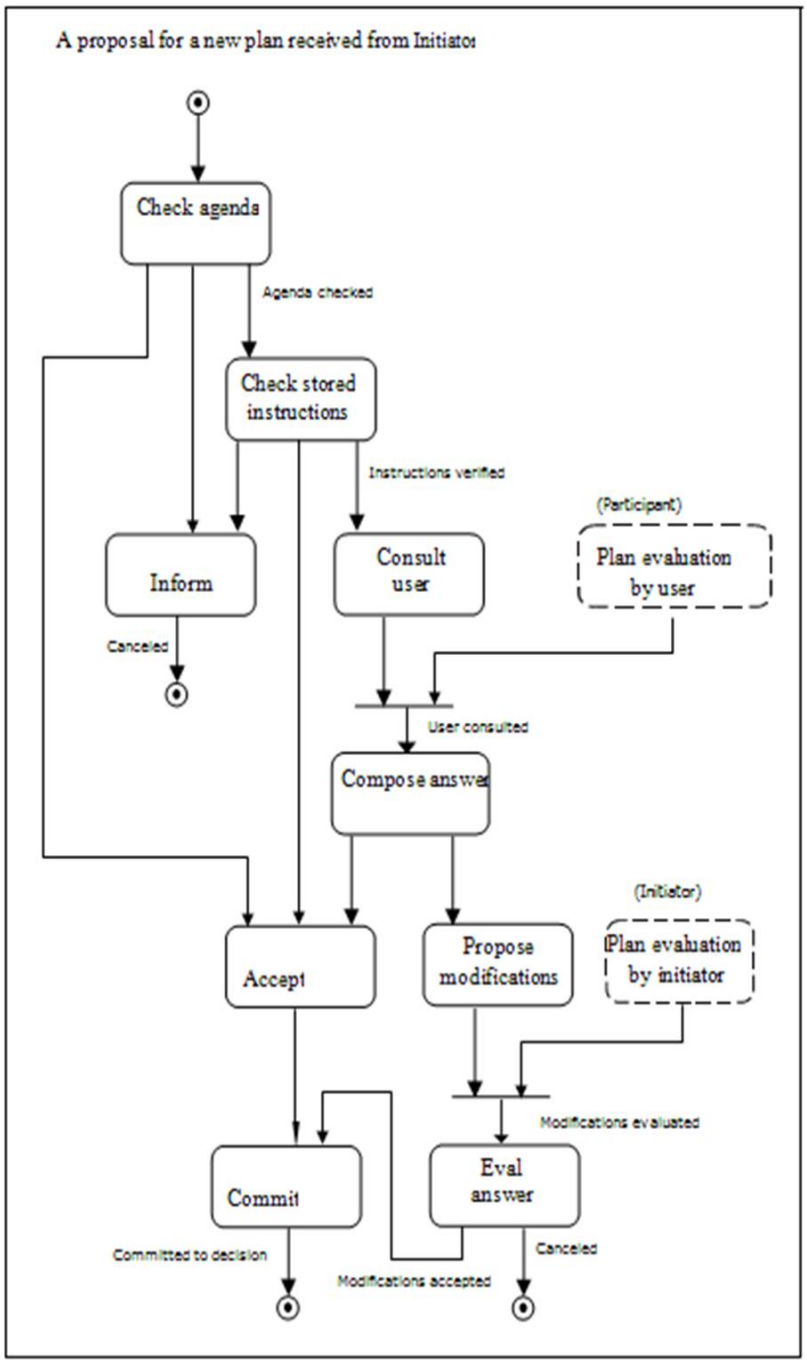


Figura 1.6: Comportamentul asistentului personal, având rolul de negociator, în faza de iniţiere a unui plan de acţiuni

1.3.6 Definirea comportamentului agenţilor

După definirea protocoalelor de interacţiune inter-agent urmează detalierea proceselor interne care au loc în fiecare agent în parte prin definirea comportamentului agenţilor. În figura anterioară sunt descrise procesele care executate de asistentul personal pe post de consultant în faza de iniţiere a unui plan de acţiune. Acest proces este lansat la apariţia unei propuneri de stabilire a unui plan de acţiuni colaborative iniţiat de către asistentul personal al iniţiatorului planului respectiv şi se finalizează fie cu angajarea utilizatorului, fie cu derogarea implicării în planul de acţiuni comune.

Procesarea internă a agentului este exprimată sub forma unei diagrame de activităţi care verfică:

1. agenda de lucru a utilizatorului invitat,
2. verifică competenţele atribuite de către iniţiator,
3. consultă utlizatorul,
4. propune modificări sau nu asupra planului,
5. aşteaptă evaluarea acestora din partea iniţiatorului,
6. informează iniţiatorul dacă utilizatorul se va implica sau nu în planul de acţiuni.

Capitolul 2

Agenţi JADE

JADE este un proiect cu o dinamică extraordinară şi suferă de probleme pe care orice proiect le are la început (cod sursă mult şi nedocumentat, clase care nu conţin cod, surse neactualizate pe site, instrucţiuni care nu se potrivesc pentru toate versiunile, add-on-uri ce nu sunt compatibile cu anumite versiuni, etc.). Pentru evitarea pierderii inutile de timp cei ce întâmpină greutăţi în compilare, dezvoltare sau rulare sunt îndemnaţi să caute soluţii ale acestor probleme inerente cu ajutorul celor care lucrează la acel proiect prin mailing-list-ul developer-ilor JADE / LEAP (înregistrarea se face pe site-ul

`http://jade.tilab.com` secţiunea `Community & developers).`

În prezent s-au dezvoltat în platforma Java:

Agenţi inteligenţi care păstrează proprietăţile agentilor autonomi şi în plus, prezintă un comportament *flexibil* caracterizat prin:

- **reactivitate:** capacitatea de a percepe propriul mediu şi de a răspunde în timp util la schimbările care apar în acesta;
- **proactivitate** : capacitatea de a expune un comportament orientat spre scop prin preluarea iniţiativei;
- **capacitatea socială :** de a interacţiona cu alţi agenţi şi eventual, cu utilizatori umani.

Agenţi deliberativi sunt agenţi care dispun de un sistem format din două componente principale:

- un set de cunoştinţe (memoria de date, similară unei baze de date)

- un set de comportamente (descrise prin reguli de tip *if -then).*

În acest sens s-a dezvoltat **Jess [14]** care este un motor de reguli pentru platforma Java, dezvoltat de *Ernest Friedman-Hill* în *Laboratoarele Naţionale Sandia din Livemore, Canada.* Pe baza acestui motor, se pot realiza aplicaţii ce au capacitatea de a "raţiona" folosind algoritmi structuraţi sub formă de reguli declarative ce sunt aplicaţi continuu setului de date din memorie. La fiecare actualizare a setului de cunoştinţe, regulile sunt evaluate (componenta *if* a acestora) după un algoritm special *Rete* (*Jess, 2008 [14]*), urmând ca în situaţiile de îndeplinire a condiţiilor să fie executate acţiunile asociate (componenta *then*).

Jess şi alte motoare de reguli îşi dovedesc eficienţa în cazul sistemelor ce conţin sute ori mii de reguli ce într-o abordare normală ar implica o complexitate operaţională de tip exponenţial. Dezvoltarea iniţială a **Jess** a vizat aplicarea sa în sistemele expert, acesta depăşindu-şi însă sfera de aplicabilitate. În sistemele multiagent, poate fi folosit drept o componentă decizională, de calcul simbolic, implementată într-un mod declarativ la nivelul agentului.

Ce trebuie să ştim despre agenţi pentru a putea lucra cu ei (strictul necesar):

- Există o platforma de agenţi (in cazul nostru JADE).
- Se scrie o clasă care extinde clasa Agent.
- Se rulează platforma şi se pornesc agenţii scrişi de noi.

Paşii pentru dezvoltarea unui sistem multiagent:

1. *Identificarea actorilor (utilizatori, agenţi, obiecte, fişiere/baze de date),*
2. *Identificarea interacţiunilor (protocoale şi mesaje, tip şi conţinut pentru fiecare mesaj) şi a diagramelor de secvenţă,*
3. *Identificarea agenţilor şi comportamentelor lor (cel mai simplu este să adăugăm câte un comportament pentru fiecare tip de mesaj primit la fiecare agent în parte).*

2.1 Instalarea platformei JADE în Eclipse

Paşi:[8]

1. Se descarcă Eclipse v.3.5 (Luna, Juno, Kepler, Mars,etc) care se găseste free pe Internet.

2. Se descarcă versiunea 3.5 a platformei Jade (`http://jade.tilab.com`.), ce conţine plug-inul **it.fbk.sra.ejade_0.8.0**, ce trebuie copiat în folderul Eclipse/Plugins.

3. Dacă s-au efectuat corect paşi anteriori, la pornirea mediului de programare Eclipse apare în bara de meniuri EJADE.

2.2 Instalare UML 2.0 sub Eclipse

Paşi: [19]

1. Pornim Eclipse şi din bara de meniuri selectam **Help,**

2. Selectăm **Install New Software,**

3. Acţionăm butonul **Add,**

4. În **Add Repository** la **Name:** tastăm **ObjectAID UML Explorer,**
   iar la **Locations: http://ObjectAid.net/update,**

5. Selectăm **ObjectAid class Diagram (free)**

6. Next

7. I accept the terms of the LICENSE AGREEMEMT +Finish.

8. Apare o eroare de atenţionare, tastăm O.K.

9. Restartăm Eclipse

2.3 Crearea unui pachet Java sub Eclipse

1. Lansăm Eclipse selectăm **File→New Project→Java Project,**
2. Click dreapta pe numele proiectului şi selectăm **Properties,**
3. Selectăm **Java Build Path** la **Libraries** selectăm **Add External Jar,**
4. Din Eclipse/Plugins selectăm *it.fbk.sra.ejade_0.8.0*, iar în folderul Lib/Libjade găsim:

   - **jade.jar**: reprezintă platforma în sine. Conţine clasele abstracte pentru definirea agenţilor (jade.core.Agent), a comportamentelor acestora, protocoalelor de interacţiune, ontologiilor, etc; este singurul jar imperativ necesar pentru a putea porni platforma.

   - **jadeTools.jar** : conţine agenţii utilitari (*Introspector, DummyAgent* etc.) precum şi componente grafice.

Dacă paşii 1-4 au fost executati corect, atunci la lansarea în executie a unui agent (click dreapta)->*EJADE->Deploy-Agent* apare framework-ul de *JADE.* Portul implicit folosit de JADE este 1099, dacă dintr-un anumit motiv nu putem folosi acest port se defineşte profilul specificând toate argumentele (hostul, portul şi id-ul platformei) astfel:

```
Profile p = new ProfileImpl("192.168.123.147",10000,
"192.168.123.147:10000/JADE");
```

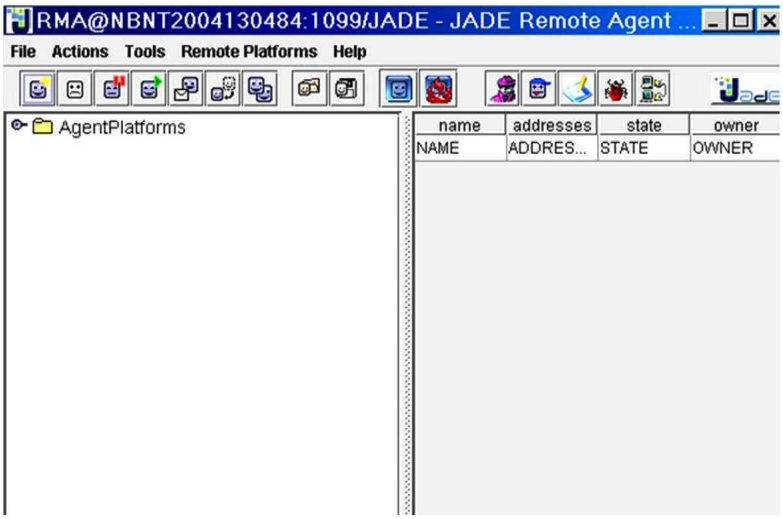


Figura 2.1: Platforma Jade

Capitolul 3

Componentele platformei JADE

Platforma JADE a devenit unul dintre cele mai populare software middleware orientate agent. Este un sistem complet distribuit cu o infrastructură flexibilă, având scopul de a facilita dezvoltarea de aplicaţii complet bazate pe agent prin intermediul unui mediu runtime ce implementează caracteristicile care stau la baza ciclului de viaţă solicitate de agenţi, precum şi logica de bază a agenţilor înşişi.

FIPA (Foundation for Intelligent Physical Agents) [9]
este o organizaţie având standardele IEEE Computer Society, care promovează tehnologia bazată pe agent şi interoperabilitatea propriilor standarde cu alte tehnologii. Standardul FIPA se bazează pe principiul că ar trebui specificat numai comportamentul extern al componentelor sistemului, lăsând arhitectura internă şi detaliile de implementare dezvoltatorilor de platforme individuale. Acest lucru asigură interoperarea între platforme conforme.

Platforma JADE oferă compatibilitate completă cu specificaţiile FIPA (comunicare, management şi de arhitectură), care oferă cadrul în care agenţii pot exista, opera şi comunica, în timp ce adoptă o arhitectura internă unică şi implementarea serviciilor de agent cheie.

JADE conţine, pe lângă biblioteca de clase pentru dezvoltarea şi manipularea agenţilor, un *runtime environment* ce trebuie activat pentru a putea lansa agenţii în execuţie. Fiecare instanţă a acestui *runtime environment* este numită *Container.* Fiecare container poate conţine mai mulţi agenţi. Toate containerele active formează o platformă, în care trebuie să existe obligatoriu un singur container special numit *MainContainer,* cu doi agenţi [4] :

1. **AMS(Agent Management Sistem) -** agentul care deţine dreptul de supervizare al accesului şi utilizării platformei multiagent. Există un singur agent **AMS** în plat-

formă. Acesta deţine şi gestionează registrul agenţilor aflaţi în platformă (registrul de AID-uri) precum şi informaţiile legate de starea acestora.

2. **DF(DirectoryFacilitator) -** agentul care furnizează serviciul de *yellow pages* pentru agenţi şi serviciile oferite de ei.

Un agent cu statut special, dar facultativ este agentul **RMA(Remote Monitoring Agent).** Acesta ofera o interfaţa pentru platforma JADE.

Trebuie reţinut că:

1. Un Agent poate rula numai într-un Container (fie el principal – MainContainer – sau într-un Container);
2. Un Container poate să nu conţină nici un Agent, un Agent sau mai mulţi agenţi;
3. Un Container poate aparţine unui singure platforme ;
4. Fiecare Platformă trebuie să conţină exact un container principal (MainContainer);
5. Un Computer poate să găzduiască una sau mai multe Platforme.

La acestea se pot adăuga containere adiţionale atât de pe calculatorul propriu cât şi de pe alte calculatoare din reţea.

Agenţi sunt identificaţi printr-un **Agent IDentifier(AID**) unic pe platformă care se setează la rularea sa de către utilizatori. Ei au o metoda *setup()* care se rulează o singură dată la pornirea agentului, aceasta trebuie suprascrisă, deoarece aici se fac initializari etc...

Un agent este doar o clasă container pentru comportamente lui, nu face altceva decât să le instanţieze urmând ca acestea să conţină codul. *Comportamentele (Behaviours)* sunt clase ale căror metode *action()* sunt rulate în mod repetat, până când sunt oprite explicit prin cod. Metoda *action()* trebuie suprascrisă, comportamentul trebuie adăugat la agent cu:

```
myAgent.addBehaviour(new ComportamentulMeu());
```

//dacă vă aflaţi intr-un alt comportament al agentului

sau

```
addBehaviour(new ComportamentulMeu());
```

// dacă vă aflati în metoda setup() a agentului

În JADE fiecare agent din platformă rulează pe un **thread** separat. Comportamentele împart acest thread nepreemtiv, deci nu ruleaza în paralel, cu alte cuvinte în momentul în care un comportament începe să ruleze nu va fi întrerupt până când nu eliberează de buna voie thread-ul invocand *metoda block().* Deci nu se foloseste metoda *sleep()* într-un comportament pentru că se va opri thread-ul agentului.

Pentru a crea propriul agent se procedează astfel:

1. se crează o clasa care extinde clasa agent,
2. se suprascrie *metoda setup()* făcând initializări şi adaugând comportamentele agentului,
3. se crează comportamentele extinzând comportamentul dorit.

Capitolul 4

Limbaje de comunicare

În JADE, comunicarea inter-agenţi se realizează prin schimb de mesaje (clasa ACLMessage) [16]. Sistemul este similar serviciilor consacrate de mesagerie *(tip Messenger)*, în sensul că oferă posibilitatea agenţilor de a schimba mesaje în timp ce sunt activi, dar şi de a dispune de o listă de mesaje (*offline messages*) în care sunt salvate mesajele primite şi încă necitite. Spre deosebire de *Messenger* mesajele nu sunt liniarizate, ci corespund unei anumite conversaţii identificate printr-un ID special.

4.1 Transmiterea mesajelor

Clasa ACLMessage prezintă următoarea structură:

Câmpuri	*Clasă – Tip*	*Metoda de setare*	*Oblig.*
Expeditor	*AID*	*setSender (...)*	*
Set de destinatari	*AID*	*addReceiver (...)*	*
Id. de tip al mesajului	*int*	*setPerformative(...)*	*
Continut	*String*	*setContent (...)*	*
PROTOCOL	*String*	*setProtocol (...)*	
Id.conversatie	*String*	*setConversationId (...)*	
ONTOLOGIE	*String*	*setOntology (...)*	

şi presupune două etape:

- crearea şi completarea mesajului *ACL*;

- trimiterea mesajului prin metoda *send (. . .)* din clasa de baza *Agent.*

Studiind clasa **jade.core**. **Agent** observăm definiţiile următoarelor metode ce lucreaza cu mesaje *ACL*, semnificând faptul că setul de funcţionalităţi necesare transmiterii/recepţionării de mesaje este disponibil fiecărui agent, prin extinderea acestei clase de bază:

```
public final void send(ACLMessage aCLMessage) { }
public final ACLMessage receive() {
return null;}
public final ACLMessage receive
(MessageTemplate messageTemplate){
return null;}
public final ACLMessage blockingReceive()
{ return null;}
public final ACLMessage blockingReceive(long _long)
{return null;}
public final ACLMessage blockingReceive
(MessageTemplate msgTemplate)
{ return null;}
public final ACLMessage blockingReceive
(MessageTemplate msgTemplate,long _long) {
return null;}
public final void
putBack(ACLMessage aCLMessage) { }
public final void
postMessage(ACLMessage aCLMessage) { }
```

Pentru a vedea continutul mesajului ACL procedăm astfel : pornim Sniffer-ul din platforma JADE şi dăm dublu click pe săgeata care apare (ea semnifică faptul că agentul este "viu", adică se face comunicarea interagent), iar în fereastra content apare conţinutul.

4.2 Filtrarea mesajelor

Frecvent este necesară o filtrare a mesajelor ce doresc a fi preluate, pentru că doar celor relevante să le fie dedicate atenţie (să fie procesate). Omul are de asemeni un impresio-

nant mecanism de filtrare al stimulilor, în special audio, astfel încât intelectul său poate fi concentrat asupra raţionamentelor esenţiale, introspecţiilor etc. (înlăturând bruiajele externe).

JADE pune la dispoziţie *clasa MessageTemplate*, cu ajutorul căreia se poate realize filtrarea mesajelor, astfel încât un agent va recepţiona doar mesaje cu anumite caracteristici dorite. Filtrul se poate aplica pe orice câmp din ACLMessage, putându-se realiza combinări de tip AND/OR/NOT pentru a forma anumite condiţii complexe. Astfel, un agent poate primi doar mesaje de tipul dorit(performative) de la un anumit agent, se poate axa doar pe o anumită conversaţie conversationId. De exemplu un agent Iniţiator poate fi modificat pentru a putea realiza filtrarea mesajelor, primind doar cereri de tip QUERY_IF sau QUERY_REF, iar agentul Responder va aştepta răspunsul agentului Iniţiator (filtrând mesajele).

4.3 Aplicaţie

Vom prezenta în cele ce urmează urmatoarea aplicatie [24] :

A) agentul Buyer îl intreabă pe agentul Seller dacă deţine o anumită carte.

B) agentul Seller va folosi filtrarea mesajelor pentru a primi doar cereri valide, iar agentul Buyer va aştepta răspunsul acestuia.

Cumparator: Buyer

```
import jade.core.*;}
import jade.lang.acl.*;
public class Buyer extends Agent {
public Buyer() {}
}
public void setup() {
 //1.creere mesaj (de tip QUERY_IF)
ACLMessage msg = msg.setSender(this.getAID());
AID receiverAID = new AID( (String) getArguments()[0], AID.ISLOCALNAME);
msg.addReceiver(receiverAID);
msg.setContent("Idiotul");
 //2.trimitere mesaj
this.send(msg);
//3.blocare pentru asteptarea mesajului doar de la vanzator
```

```
 MessageTemplate mt = MessageTemplate.MatchSender(receiverAID);
ACLMessage msgResponse = blockingReceive(mt);
//4. prelucrare raspuns vanzator
 if (msg.getContent() == "true") {}
else if (msg.getContent() == "false") {} } }
```

****

Vanzator: Seller

****

```
import jade.core.Agent;
import jade.lang.acl.*;
import java.util.Hashtable;
public class Seller extends Agent {
private Hashtable catalogue;
public Seller() {
catalogue = new Hashtable();
catalogue.put("Idiotul", "30");
catalogue.put("\Invierea", "10");
catalogue.put("Crima si Pedeapsa", "50"); }
public void setup() {
 while (true) {
 ***********************************************
//1.creere filtru MessageTemplate folosit pentru
//receptionarea doar a mesajelor de tip QUERY_IF
//ori QUERY_REF
***********************************************
MessageTemplate mt =
MessageTemplate.or(MessageTemplate.MatchPerformative
( ACLMessage.QUERY_IF),MessageTemplate.MatchPerformative
(ACLMessage.QUERY_REF));
}
//2.blocarea agentului pana la receptionarea unui
// mesaj conform filtrului
ACLMessage msg = blockingReceive(mt);
```

```
System.out.println(getLocalName() +
": am primit un mesaj de la "
 + msg.getSender().getLocalName());
//3.prelucrarea mesajului ACLMessage
msgReply = msg.createReply();
msg.setPerformative(ACLMessage.INFORM);
Object catalogueVal = catalogue.get(msg.getContent());
if (msg.getPerformative() == ACLMessage.QUERY_IF) {
 msgReply.setContent(catalogueVal != null ?
"true" : "false");}
else { msgReply.setContent
(catalogueVal != null ? catalogue.toString() : "");}
 //4.trimitere raspuns
send(msgReply);}}
```

4.4 Probleme propuse

1. *Doi prieteni comunică astfel:*

a) *Primul un iniţiator (Pavel) doreşte să cunoască cum poate să cucerească inima unei fete (Adina, Ligia, Natalia, Paula,Ioana).*

b) *Celălalt responder (Petru) îi oferă detalii (flori, cărţi, filme).*

2. *Să se scrie o aplicaţie JADE care să conţină următoarea conversaţie:*

a) *Agenţii studenţi îl întreabă pe agentul profesor dacă deţine plugin de UML sub ECLIPSE,*

b) *Agentul profesor va folosii filtrarea mesajelor pentru a primii doar cereri valide, iar agentul student X va aştepta răspunsul acestuia.*

3. *Scrieţi o aplicaţie JADE în care să existe un agent ce executa un comportament tip FSMBehaviour, didactic [24]. Folosiţi un DummyAgent pentru a dialoga cu agentul , corespunzător protocolului de interacţiune descris (codat) în comportamentul său.*

4. *Să se scrie un exemplu de agregare recursivă a comportamentelor compuse. Acestea conţin comportamente secvenţiale (SequentialBehaviour) şi simple, OneShotBehaviour.*

Capitolul 5

Comportamentele agenţilor

Sarcina sau sarcinile concrete pe care un agent le are de îndeplinit sunt realizate prin comportamente.

Definiţie 8 *Un comportament reprezintă o sarcină pe care un agent trebuie să o îndeplinească şi este implementat ca un obiect al unei clase ce extinde* ***jade.core.behaviour.Behaviour.***

Autonomia este caracteristica principală a unui agent FIPA. Pentru asigurarea acesteia, un agent nu trebuie să poată fi controlat direct de către program, decât într-o măsură limitată (prin interfaţarea agentului). Pentru a face ca un agent să execute o sarcină implementată de un obiect `Behaviour,` acesta trebuie să fie adăugat la clasa agentului prin metoda `addBehaviour().`

```
public class AgentNou extends Agent
{@Override
public void setup()
{ addBehaviour(new Asistent (this,300));
addBehaviour(new Asistent (this,800));
} }
```

Pentru simplitate, agenţii JADE dispun de un singur fir de execuţie, prin intermediul căruia sunt executate instanţe de comportamente, pe baza unui algoritm de planificare de tip **round-robin** nepreemptiv (se execută *metoda action ()* a primei instanţe din listă, apoi a următoarei etc. iar când se ajunge la capătul listei se reia cu prima instanţă). Fiecare agent dispune de două liste de comportamente: una pentru cele active, cealaltă

pentru comportamentele blocate (funcţionalitate necesară comportamentelor consumatoare de timp CPU şi care de fapt doar aşteaptă un mesaj). Blocarea unui comportament se realizeaza prin apelul *metodei block(),* care îl scoate din lista comportamentelor active şi-l introduce în cea a comportamentelor blocate, imediat după finalizarea *metodei action().*

Fiecare comportament, extinzând clasa abstracta Behaviour, trebuie să definească metodele:

1. *public void action()*{ }*:* conţinând codul comportamentului;
2. *public boolean done()*{ }*:* reprezentând condiţia pentru terminarea comportamentului.

Comportamentul va rămâne activ până când *metoda done()* va returna **true**. Execuţia comportamentului poate fi:

1. blocată prin *metoda block( )* - blocarea se va realiza efectiv la finalul *metodei action( )*;
2. reluată prin *metoda restart( )*- ori dacă a intervenit un timeout pentru blocare, ori s-a primit un mesaj (caz in care toate comportamentele sunt reluate).
3. apelul *metodei reset()*- reactivează comportamentul, resetându-l însă.

Structurare clasei Behaviour permite definirea unor tipuri predefinite de comportamente. Acestea pot fi:

- simple extinzând clasa abstractă SimpleBehaviour,
- compuse, extinzând clasa abstractă CompositeBehaviour.

Algoritm general pentru implementarea *metodei action() :*

```
ACLMessage msg = myAgent.receive
(MessageTemplate.MatchConversationId("idConversatie"));
//este nevoie sa folosim MessageTemplate ca
//sa trieze mesajele automat
if (msg != null) {
// cate odata coada de mesaje este plina
```

```
// de exemplu cand acest comportament ruleaza
// daca dorim sa executam altceva pina se primeste
// mesajul
else {
block();
//Se blocheaza firul pina cand apare un nou mesaj in coada
}
```

5.1 Comportamente simple

1. Clasa **SimpleBehaviour:** reprezintă o clasă abstractă ce modelează un comportament atomic (execuţia acestuia nu poate fi întreruptă);

2. Clasa **OneShotBehaviour**: clasă abstractă ce modelează un comportament atomic ce este executat o singura dată (*done()* returnează implicit true);

3. Clasa **WakerBehaviour**: clasă abstractă ce modelează un comportament atomic ce va fi executat o singură dată, după un anumit interval de timp;

4. Clasa **CyclicBehaviour**: clasă abstractă ce modelează un comportament atomic ce va fi executat la nesfârşit (*done()* returnează implicit false);

5. Clasa **TickerBehaviour**: clasă abstractă ce modelează un comportament atomic ce va fi executat continuu periodic (la un anumit interval de timp specificat);

5.2 Aplicaţii

1.Un agent ce execută două comportamente, unul simplu şi unul ciclic [24]:

```
public class TestAgent extends Agent{
private long actionCount = 0;
public TestAgent() { }
public void setup() {
//adaugarea comportamentului simplu tip OneShot
addBehaviour(new OneShotBehaviour(this){
public void action() {
```

```
System.out.println(getAID().getLocalName()+"
// executa comportamentul simplu
(" + actionCount + ")"); actionCount++; } });
//adaugarea comportamentului simplu ciclic
addBehaviour(new CyclicBehaviour()
public void action() {
System.out.println(getAID().getLocalName()+
" executa comportamentul ciclic
(" + actionCount + ")");
actionCount++;} );
}
```

Comportamentele au fost executate secvenţial. Primul, cel de tip **OneShotBehaviour** a efectuat prima incrementare a contorului **actionCount**.

La terminarea metodei sale *action()*, comportamentul ciclic a fost executat, cel **OneShot** fiind scos din lista de comportamente active.

Astfel a rămas în execuţie doar comportamentul ciclic, care va incrementa continuu variabila **actionCount.**

Segmentul de cod:

```
addBehaviour(new OneShotBehaviour(this) {
public void action(){...}
public boolean done(){...}
}
```

are semnificaţia creării unei noi clase de comportament ce extinde clasa **OneShotBehaviour** şi adăugării directe a unei instanţe a acesteia în lista de comportamente.

Apelul **new OneShotBehaviour(this)** reprezintă instanţierea acestei clase nou definite. Această modalitate este o variantă prescurtată pentru definirea separată a unei clase pentru comportament (intern clasei agentului sau într-un fişier separat), crearea unei instanţe a acesteia şi adăugarea în lista de comportamente.

Clasa definită nu este accesabilă sau instanţiabilă din altă regiune a programului, vizibilitatea ei fiind restransă doar la acest segment de cod.

2. Să se scrie un agent **Time Agent** care instanţiază două comportamente dependente de timp: un **TickerBehaviour** care se va executa continuu, şi un **WakerBehaviour (** combinaţie între **TickerBehaviour şi OneShotBehaviour)** [5].

```
import jade.core.Agent;
import jade.core.behaviours.WakerBehaviour;
import jade.core.behaviours.TickerBehaviour;
public class TimeAgent extends Agent {
protected void setup() {
 System.out.println("Agent "+getLocalName()+" started.");
 // Add the TickerBehaviour (period 1 sec)
addBehaviour(new TickerBehaviour(this, 1000) {
protected void onTick() {
 System.out.println("Agent "+
myAgent.getLocalName()+":
tick="+getTickCount());}
});
 // Add the WakerBehaviour (wakeup-time 10 secs)
addBehaviour(new WakerBehaviour(this,10000){
protected void handleElapsedTimeout(){
 System.out.println("Agent "+myAgent.getLocalName()+":
It's wakeup-time. Bye...");
myAgent.doDelete();}
});
```

În urma executiei va rezulta la consola:

Agent TimeAgent started.

Agent TimeAgent: tick=1

Agent TimeAgent: tick=2

Agent TimeAgent: tick=3

Agent TimeAgent: tick=4

Agent TimeAgent: tick=5

Agent TimeAgent: tick=6

Agent TimeAgent: tick=7

Agent TimeAgent: tick=8

Agent TimeAgent: tick=9

Agent TimeAgent: It's wakeup-time. Bye...

3.Un agent ce execută un comportament etapizat simulând prepararea unei cafele.

```
public class TestAgent extends Agent
{private long actionCount = 0;
public TestAgent() {}
public void setup() {
MyBehaviour behaviour = new MyBehaviour();
addBehaviour(behaviour); }
private class MyBehaviour extends SimpleBehaviour
{ int step = 1;
public void action() {
switch (step) {
case 1: System.out.println("Step " + step + ":Punem apa la fiert.");
 break;
case 2: System.out.println("Step " + step +
":Punem doua lingurite de cafea.");
break;
case 3: System.out.println("Step " + step +
": Lasam sa fiarba timp de 10 minute, amestecand bine .");
break;
case 4: System.out.println("Step " + step + ":Cafeaua este gata.
Aprindem o tigara");
break;}
step++; }
 public boolean done() {
return step == 5;}
}}
```

Comportamentele etapizate sunt folosite atât în dezvoltarea protocoalelor de interacţiune, când pentru fiecare etapă agentul ştie că trebuie să trimită un anumit mesaj ori să efectueze o anumită acţiune. Astfel de comportamente se situează între **OneShotBehaviour** şi **CyclicBehaviour.**

Observaţie 9 *Când se extinde clasa SimpleBehaviour va trebui să suprascriem metodele* action() *şi* done().

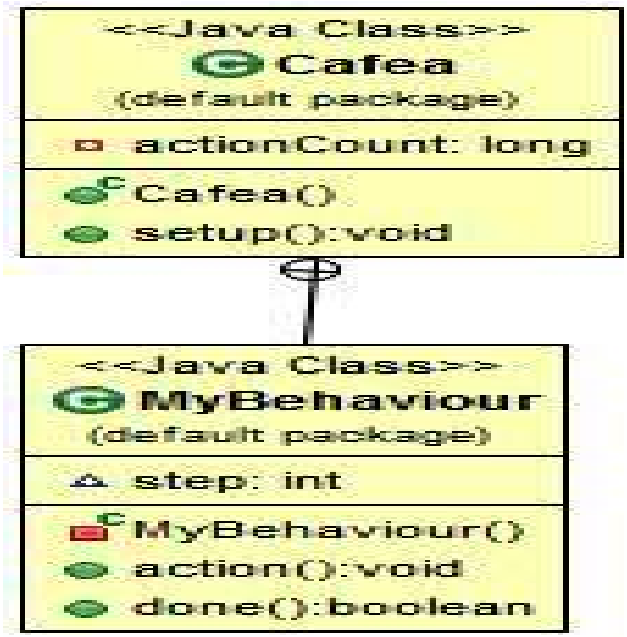


Figura 5.1: Diagrama UML Cafea

Observaţie 10 *Comportamentele care extind TickerBehaviour nu trebuie să suprascrie decât metoda* onTick().

Aceasta metodă va fi invocată cu perioadă specificată în constructor: `TickerBehaviour(Agent a,long period)`.

Observaţie 11 *Analog WakerBehaviour are* metoda onWake().

5.3 Comportamente compuse

JADE furnizează clase pentru implementarea de comportamente compozite. Astfel, este posibilă definirea de ierarhii de comportamente, ce vor conţine subcomportamente.

În pachetul **jade.core.behaviours** găsim clasele: **FSMBehaviour, SequentialBehaviour** si **ParallelBehaviour** (de tip CompositeBehaviour).

Aceste trei clase au definită politica de planificare a subcomportamentelor, de aceea se obişnuieste folosirea directă a acestora:

1. **Clasa CompositeBehaviour:** reprezintă o clasă abstractă ce modelează un comportament compus (conţine subcomportamente), astfel încât acţiunile propriu zise

nu sunt definite în acesta ci în metodele *action()* ale subcomportamentelor, comportamentul părinte fiind responsabil de planificarea pentru execuţie a acestora conform unei politici stabilite. **Clasa CompositeBehaviour** nu are implementată nici o astfel de politică, acestea fiind implementate în clasele specializate SequentialBehaviour, FSMBehaviour, ParallelBehaviour;

2. **Clasa SequentialBehaviour:** comportament compozit ce planifică subcomportamentele pentru execuţie în mod secvenţial; clasă utilă pentru implementarea protocoalelor de interacţiune.

3. **Clasa ParallelBehaviour:** comportament compozit ce execută subcomportamentele în mod concurent. Execuţia acestuia se finalizează fie când toate subcomportamentele, unul singur ori un număr specificat din acestea şi-au finalizat execuţia (metoda *done()* returnează true).

4. **Clasa FSMBehaviour:** comportament compozit ce foloseşte modelul unui automat cu stări în care tranziţiile sunt definite de programator, iar stările reprezintă subcomportamentele.

Dacă se ajunge într-o stare finală, comportamentul îşi termină execuţia după ce subcomportamentul a fost executat.

5.4 Aplicaţie

Un agent ce execută un comportament tip **FSMBehaviour,** didactic [5].

```
import jade.core.Agent;
import jade.core.behaviours.OneShotBehaviour;
import jade.core.behaviours.FSMBehaviour;
public class FSMAgent extends Agent {
 // Numele starilor
private static final String STATE_A = "A";
private static final String STATE_B = "B";
private static final String STATE_C = "C";
private static final String STATE_D = "D";
private static final String STATE_E = "E";
```

```
private static final String STATE_F = "F";
protected void setup() {
FSMBehaviour fsm = new FSMBehaviour(this) {
public int onEnd() {
System.out.println("FSM behaviour completed.");
 myAgent.doDelete();
 return super.onEnd();}
};
// Inregistrarea  starii A (first state)
 fsm.registerFirstState(new NamePrinter(),STATE_A);
// Inregistrarea  starii B
 fsm.registerState(new NamePrinter(),STATE_B);
// Inregistrarea  starii C
 fsm.registerState(new RandomGenerator(3),STATE_C);
 // Inregistrarea starii D
 fsm.registerState(new NamePrinter(),STATE_D);
 // Inregistrarea starii E
 fsm.registerState(new RandomGenerator(4,STATE_E);
 // Inregistrarea starii F (final state)
fsm.registerLastState(new NamePrinter(),STATE_F);
// Inregistraea tranzitiilor
fsm.registerDefaultTransition(STATE_A,STATE_B);
fsm.registerDefaultTransition(STATE_B,STATE_C);
fsm.registerTransition(STATE_C,STATE_C,0);
fsm.registerTransition(STATE_C,STATE_D,1);
fsm.registerTransition(STATE_C,STATE_A, 2);
fsm.registerDefaultTransition(STATE_D,STATE_E);}
fsm.registerTransition(STATE_E,STATE_F,3);
fsm.registerDefaultTransition(STATE_E,STATE_B);
addBehaviour(fsm);
private class NamePrinter extends OneShotBehaviour {
 public void action() {
 System.out.println("Executing behaviour"+getBehaviourName());}}
 private class RandomGenerator extends NamePrinter {
```

```
 private int maxExitValue;
 private int exitValue;
 private RandomGenerator(int max) {
super();
 maxExitValue = max;}
public void action() {
System.out.println("Executing behaviour "+getBehaviourName());
 exitValue = (int) (Math.random() * maxExitValue);
System.out.println("Exit value is "+exitValue);}
 public int onEnd() {
 exitValue;
 }}}
```

La consola va aparea rezultatul:

```
Executing behaviour A
Executing behaviour B
Executing behaviour C
Exit value is 1
Executing behaviour D
Executing behaviour E
Exit value is 3
Executing behaviour F
FSM behaviour completed.
```

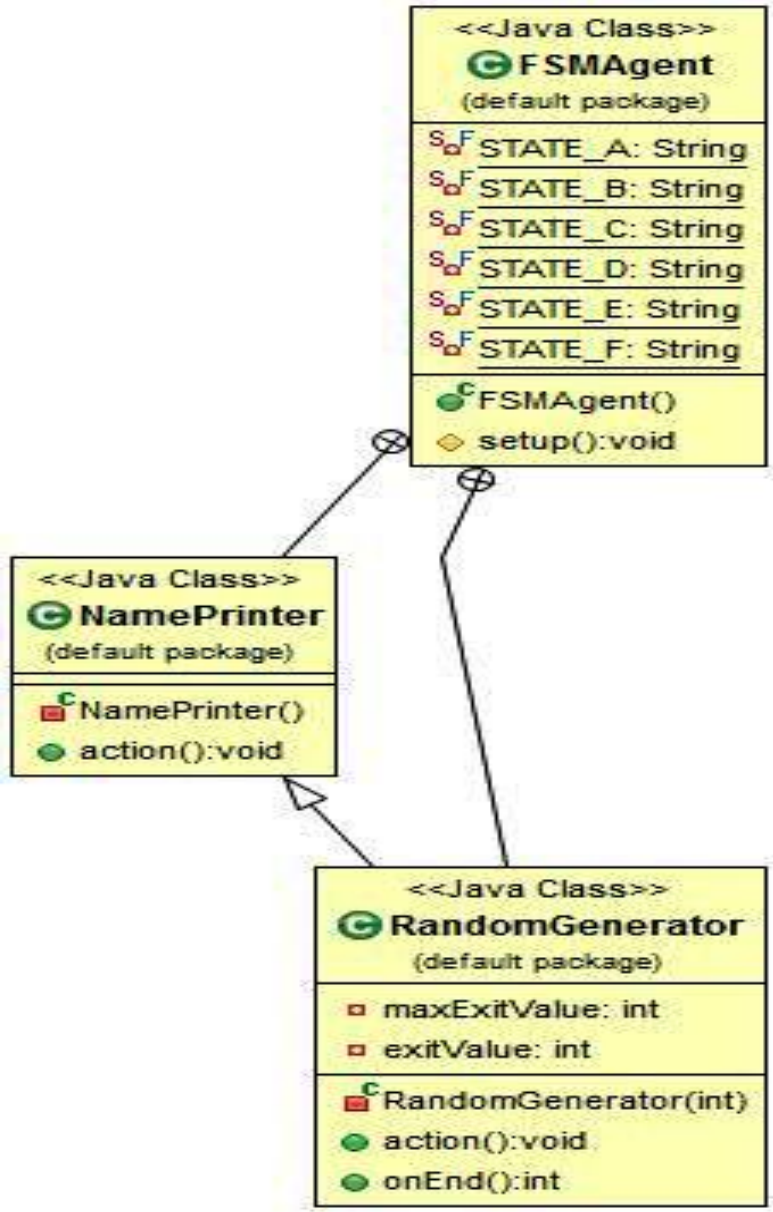


Figura 5.2: Diagrama UML-FSM

5.5 Comunicare interagent

Comunicarea şi interacţiunea dintre agenţi sunt caracteristici fundamentale ale sistemelor multiagent. Acestea sunt realizate prin intermediul schimbului de mesaje. Este deci foarte important ca agenţii să folosească pentru aceste mesaje un format şi o semantică compatibile. Deoarece JADE foloseşte standardele FIPA, în mod normal agenţii JADE ar trebui să poată comunica cu alţi agenţi care rulează pe alte platforme.

Pe lângă conţinut, un mesaj trebuie sa aibă o listă cu destinatari, un expeditor, un format, un tip de mesaj etc. În JADE mesajele folosesc standardul ACL (Agent Communication Language),standard ce permite mai multe tipuri de codare a conţinutului mesajului. De asemenea permite conţinutului mesajului să reprezinte un obiect serializat.

Comunicarea dintre agenţi se bazează pe trimiterea de mesaje asincrone. Fiecare agent are o „căsuţă poştală" (coada de mesaje), unde JADE plasează mesaje trimise de alţi agenţi şi agentul care a primit mesajul este notificat.

Trimiterea de mesaje spre alţi agenţi se realizează completând câmpurile unui obiect de tip ACLMessage şi apoi apelând metoda *send()* a agentului cu parametrul reprezentat de mesajul completat. Un agent poate selecta mesaje din coada sa de mesaje executând metoda *receive()* care returnează primul mesaj din coada sau null dacă nu sunt mesaje.

Codul Java este:

```
//trimiterea unui mesaj: creeaza mesajul ACL
// si seteaza destinatar, continut
ACLMessage message = new ACLMessage();
AID receiverAID ="Agent1", AID.ISLOCALNAME);
// In acelasi container
message.addReceiver(receiverAID);
message.setContent("Salut");
myAgent.send(message);
//receptionarea unui mesaj:
// cere un mesaj din coada sa interna
ACLMessage message = myAgent.receive();
//afiseaza continutul la consola
if (message != null)
{
string s = message.getContent() + " from " + message.getSender().
```

```
getLocalName();
System.Console.WriteLine(s);
}
```

5.6 Aplicaţie

Enunţ

Să se scrie o aplicaţie care să conţină un agent Manager, un număr de agenţi PING-PONG şi un număr de agenţi care ştiu să oprească un agent PING-PONG [12].

Avem:

- agenţi Ping-Pong care ştiu să răspundă la mesaje PING cu PONG si să se închidă la primirea unui mesaj STOP .
- agenţi care ştiu să oprească agenţi PING-PONG aleşi aleatoriu la intervale regulate.
- un agent Manager care va raporta periodic care agenţi PING-PONG sunt în "viaţă "prin efectuarea unui PING către aceştia.

Comunicarea dintre agenţi se bazează pe trimiterea de mesaje asincrone. Fiecare agent PING are o „căsuţă poştală" (coada de mesaje), unde JADE plasează mesaje trimise de alţi agenţi (PONG) şi agentul care a primit mesajul este notificat. Trimiterea de mesaje spre alţi agenţi se realizează completând câmpurile unui obiect de tip ACLMessage şi apoi apelând metoda *send()* a agentului cu parametrul reprezentat de mesajul completat. Un agent poate selecta mesaje din coada sa de mesaje executând metoda *receive()* care returnează primul mesaj din coadă sau **null,** dacă nu sunt mesaje.

Implementarea aplicaţiei

Aplicaţia conţine următorii agenţi:

1. AgentPingPong
2. AgentManager
3. AgentStop

Agenţii PingPong

Aceştia au implementat un `CyclicBehaviour`, adică comportamente ce se execută în mod repetat. Funcţia *block()* pune comportamentul în **stand bye** pâna la receptia urmatorului mesaj. La receptie, toate comportamentele vor fi activate, iar funcţiile lor *action()* vor fi executate pe rând, *block()* nu opreşte execuţia, ci face tranziţia spre următoarea execuţie. Dacă nu se apeleaza *block(),* comportamentele de tip `CyclicBehavior` vor rula într-o buclă. Agenţii PingPong, în momentul în care au primit mesajul "Ping" răspund cu "Pong", iar dacă au primit "Stop" se opresc.

Codul care trebuie implementat este:

```
addBehaviour(new CyclicBehaviour( this) {
public void action()
{ ACLMessage msg = receive();
if (msg!=null)
{if(msg.getContent().compareTo("Ping")==0)
{ System.out.println( myAgent.getLocalName() + "
received: " + msg.getContent() );
ACLMessage reply = msg.createReply();
reply.setPerformative( ACLMessage.INFORM );
reply.setContent("Pong");
System.out.println( myAgent.getLocalName() + "
send: Pong" );
send(reply);}
else if(msg.getContent().compareTo("Stop")==0)
{System.out.println( myAgent.getLocalName() + "
received:" + msg.getContent() );
try
{ DFService.deregister(myAgent);}
catch(FIPAException e){ }
myAgent.doDelete();
//stergerea agentilor din coada de asteptare
}}
else block(); }
```

AgentManager

Acesta are implementat un comportament TickerBehaviour, comportament care se

execută periodic, la un interval specificat . Operaţiile executate se specifică suprascriind metoda *onTick()*.

La intervalul de timp specificat, acesta trimite mesajul "Ping" către un agent **Ping-Pong** ales în mod aleator.

Codul este:

```
addBehaviour(new TickerBehaviour(this,1000){
protected void onTick(){
ACLMessage message =
new ACLMessage(ACLMessage.INFORM);
message.setContent("Ping");
try
{theReceiver = get_random_agent();
message.addReceiver(theReceiver);
System.out.println
( myAgent.getLocalName() + "send:Ping");
send(message);}
catch(Exception e)
{}}
```

AgentStop

Acesta are implementat tot un comportament `TickerBehaviour`. Rolul lui este de a trimite mesajul "Stop" către un agent PingPong ales în mod aleator, la intervalul de timp specificat. În momentul în care AgentPingPong va primii mesajul, acesta se va închide.

Codul este:

```
addBehaviour(new TickerBehaviour(this,4000){
protected void onTick(){
try
{ACLMessage message =
new ACLMessage(ACLMessage.INFORM);
message.setContent( "Stop" );
message.addReceiver( get_random_agent() );
System.out.println
( myAgent.getLocalName() + " send: Stop" );
send(message);}
```

```
catch(Exception e)
{}
}
```

Modul în care comunica aceşti agenţi poate fi observat în figura următoare.

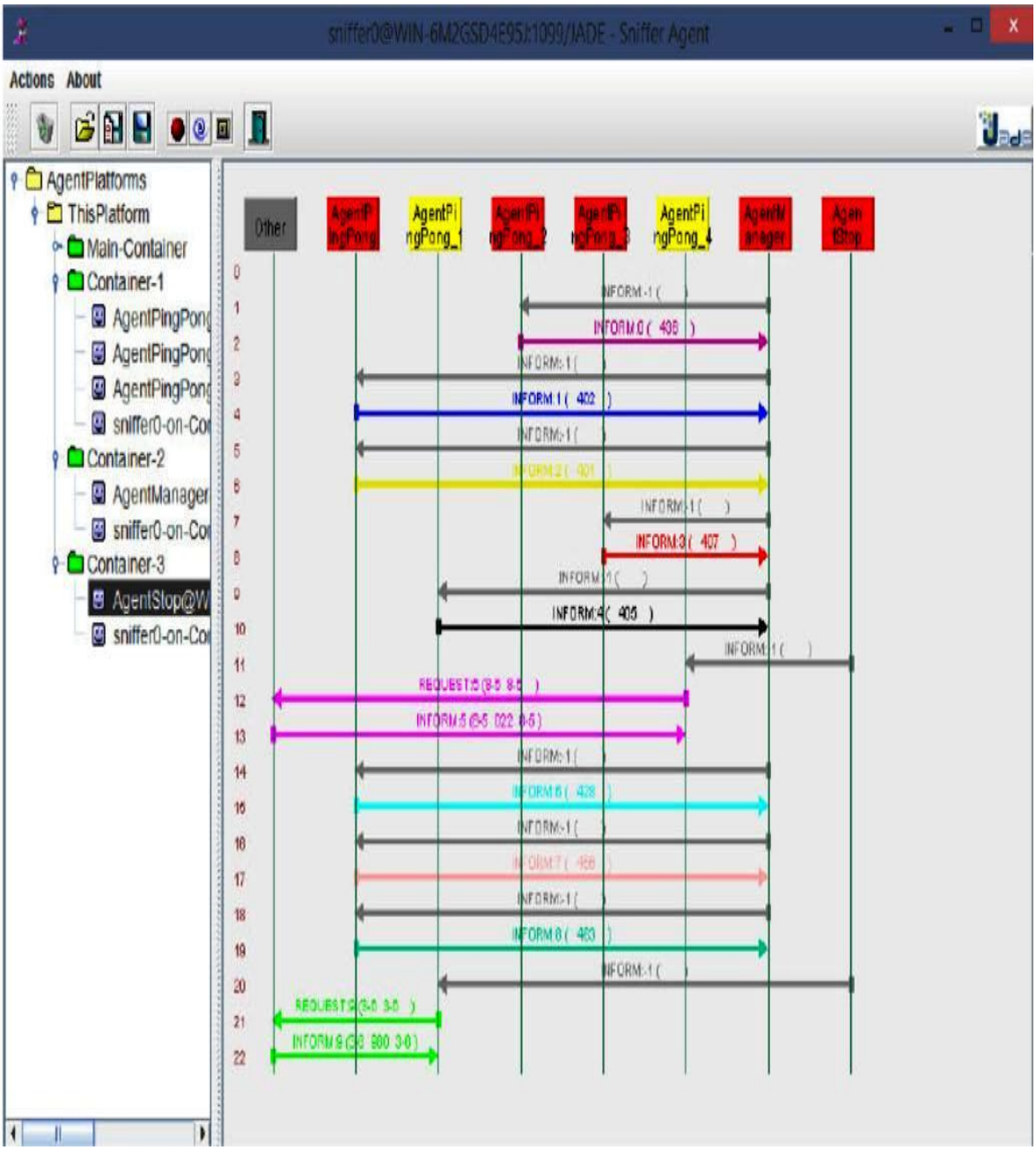


Figura 5.3: Sniffer

Se poate observa cum Agenţii PingPong răspund Agentului Manager în momentul în care au primit "Ping" de la acesta. Agenţii PingPong cu galben sunt acei agenţi ce au primit mesajul "Stop" de la AgentStop şi s-au inchis. Am folosit agentul DF(Directory Facilitator) ce pune la dispoziţie un serviciu de genul Pagini Aurii, prin care poate găsi agenţi care implementează un anumit tip de servicii. Agenţii PingPong sunt înregistrati, iar agenţii Manager şi Stop folosesc acest serviciu pentru a găsi agenţii cărora le trimit respectivele mesaje.

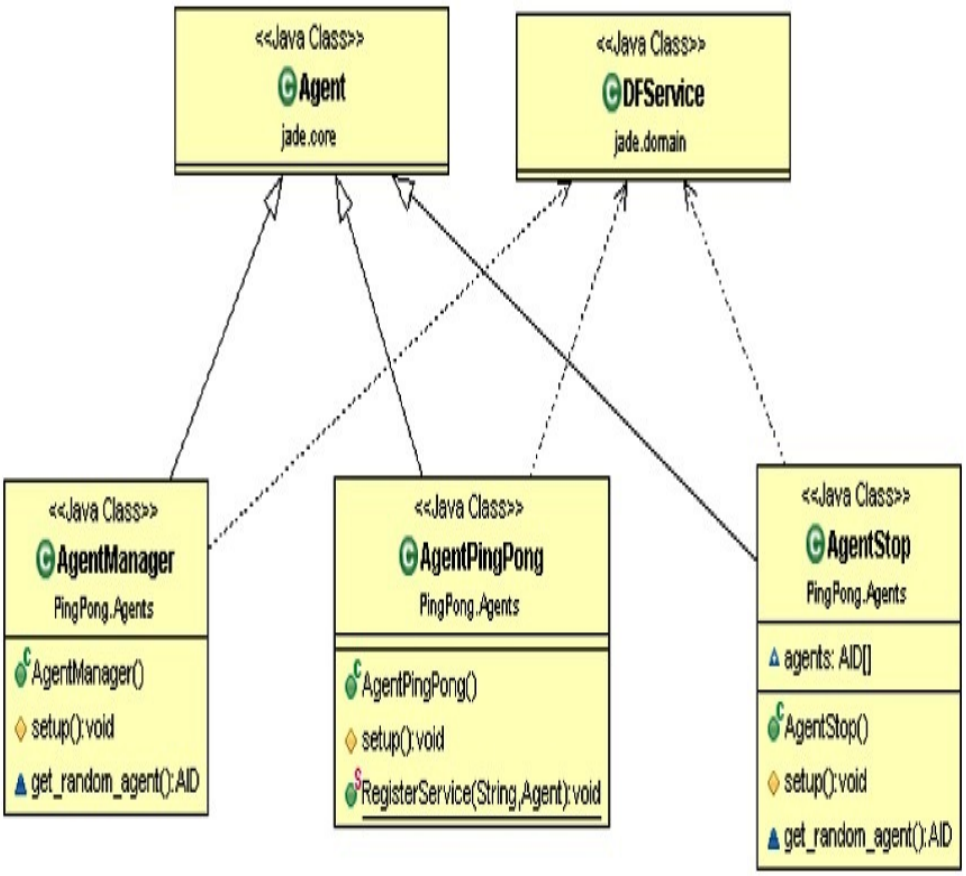


Figura 5.4: UML Ping-Pong

5.7 Probleme propuse

1. *Să se implementeze un sistem multiagent care gestionează traficul rutier pe Bulevardul Coposu din Sibiu. În sistem vor exista mai mulţi agenţi care vor gestiona datele astfel încât să nu se creeze ambuteiaje, iar timpul de aşteptare total, la toate semafoarele să nu depăşească 3 minute. Se va ţine cont de numărul de maşini ce se află la semafor şi de distanţa de deplasare între semafoare. O maşină are în general o lungime între 3-5 metrii(medie 4.2 plus spaţiul dintre ele, maxim 0.5 metrii). Pentru fiecare nouă maşină ce apare în intersecţie se crează un nou agent care reţine datele maşinii şi direcţia de mers (stânga,dreapta,înainte). Aceste date vor fi ananlizate de către un agent, un calculator care va analiza atât datele traficului cât şi a maşinilor ce merg în direcţia respectivă şi va lua decizia corectă ca timp de stat la semafor. Se va considera cel puţin trei intersecţii şi se va ţine cont de faptul că o maşină se poate deplasa la un alt semafor sau că pot părăsii Bulevardul Coposu, în timp ce alte maşini intră pe bulevard. Se poate crea un agent care generează maşini noi.*

2. *Să se implementeze o aplicaţie S.M.A în JADE pentru echilibrarea încărcării într-un sistem distribuit [4]. Sistemul va prelucra date de mari dimensiuni. Fiecare componentă a sistemului distribuit va fi reprezentată printr-un agent* ***Processor Agent.*** *În sistem vor exista încă doi agenţi* ***DistributorAgents*** *care vor obţine informaţii despre datele de intrare şi va repartiza aleator operaţiile ce trebuie efectuate către fiecare* ***Processor Agent.*** *Evident aceasta va conduce la un dezechilibru, unii* ***Processor Agents*** *având mai mult de lucru decât alţii. După distribuirea operaţiilor,* ***Processor Agents*** *vor trimite informaţii legate de propria încărcare a unui nou agent* ***DispatcherAgent.*** *Pentru a contracara dezechilibrul generat* ***DispatcherAgent*** *va putea genera* **HelperAgents (*agenţi fii*)** *care pot efectua un număr fix, prestabilit de operaţii. Funcţie de dezechilibrul existent,* ***HelperAgents*** *vor putea prelua o parte din sarcinile distribuite* ***Processor Agents.*** *Trebuie generaţi suficienţi* **HelperAgents** *pentru ca încărcarea* ***Processor Agent*** *să se situeze sub un prag (threshold) stabilit de* ***DistributorAgents.*** *Fiecare agent, în afară de* **HelperAgents** *va afişa în ferestre informaţii complecte legate de propria stare, datele prelucrate, agenţii generaţi, operaţiile efectuate după caz.*

3. *Să se creeze un chat automat la care să participe 5 agenţi: Agent1, Agent2, ...Agent5. Fiecare agent va trimite periodic câte un mesaj către un agent ales aleatoriu de genul "AgentX întreabă Ce oră e acum ?", "Agent Y răspunde ora este 20:43".*

Capitolul 6

Protocoale de interacţiune

Pentru îndeplinirea anumitor obiective, individuale ori colective, este necesară respectarea anumitor etape, reguli.

Definiţie 12 *Un protocol de interacţiune detaliază şi stabileşte secvenţa interacţiunilor între doi sau mai mulţi agenţi pentru atingerea obiectivelor acestora.*

Necesitatea protocoalelor de interacţiune derivă nu numai din nevoia de a clarifica şi etapiza anumite interacţiuni complexe ci şi pentru a compensa caracterul autonom al agenţilor, care le permite acestora să răspundă sau nu unor anumite mesaje (cereri). Astfel, prin folosirea unui protocol de interacţiune, un agent va ştii sigur dacă interlocutorul său a primit mesajul sau/şi felul în care a reacţionat, deoarece va fi înţtiinţat de acesta. Dacă cei doi nu ar fi respectat un protocol, destinatarul ar fi putut ignora mesajul, iar expeditorul ar fi rămas în incertitudine în legătură cu îndeplinirea cererii sale sau cu recepţionarea acestuia.

Protocoale FIPA:

FIPA a specificat un set minim de tipuri de protocoale standard, care reuşesc să atingă marea majoritate a nevoilor de interacţiune în sistem:

FIPA-Request, FIPA-Request-When, FIPA-Query, FIPA-Contract-Net, FIPA-Contract-Net-Interaction Protocol, FIPA-Brokering, FIPA-Recruiting, FIPA-Subscribe, FIPA-Propose, FIPA-English Auction, FIPA- Dutch Auction. JADE, fiind o platformă conformă FIPA, implementează aceste protocoale, punând la dispoziţie pachetul **jade.proto** ce conţine clasele abstracte corespunzatoare. Programatorul defineşte doar metodele ce tratează recepţia de mesaje.

6.1 Protocolul FIPA-Request

Acest protocol permite unui agent să ceară altui agent să execute anumite acţiuni. Agentul care participă la conversaţie (cu rolul de Participant) procesează cererea şi stabileşte, dacă acceptă sau refuză sarcina. Dacă refuză, atunci trebuie să înştiinteze iniţiatorul printr-un mesaj REFUSE. Dacă este de acord, trimite (opţional) un mesaj AGREE. Îndată ce Participantul s-a decis să fie de acord, va trimite unul dintre urmatoarele mesaje:

- EROARE (FAILURE), în caz că nu reuşeşte să îndeplinească sarcina;
- INFORM-DONE, în caz că a îndeplinit sarcina cu succes;
- INFORM-RESULT, în caz că a indeplinit sarcina cu succes şi doreşte să furnizeze rezultatele.

Orice interacţiune care foloseşte acest protocol, este identificată printr-un identificator global unic, dat de catre Iniţiator. Agenţii implicaţi în această conversaţie (interacţiune) trebuie să adauge mesajelor acest identificator. Acesta se dovedeşte foarte util, atunci când un agent trebuie să-şi aleagă strategiile de comunicare sau atunci când trebuie să indentifice anumite conversaţii, având la dispoziţie o "istorie a acestora". Agentul care recepţionează un mesaj, poate trimite în orice moment un mesaj "**not_understood**", în cazul în care nu a înţeles ce i s-a comunicat. Comunicarea unui astfel de mesaj, poate avea ca şi consecinţe, terminarea întregii conversaţii. În orice moment al conversaţiei, iniţiatorul acesteia poate anula conversaţia printr-un mesaj "**cancel**". Participantul poate răspunde cu unul dintre următoarele mesaje: INFORM-DONE, interacţiunea s-a terminat sau FAILURE, în cazul unei erori în anularea interacţiunii.

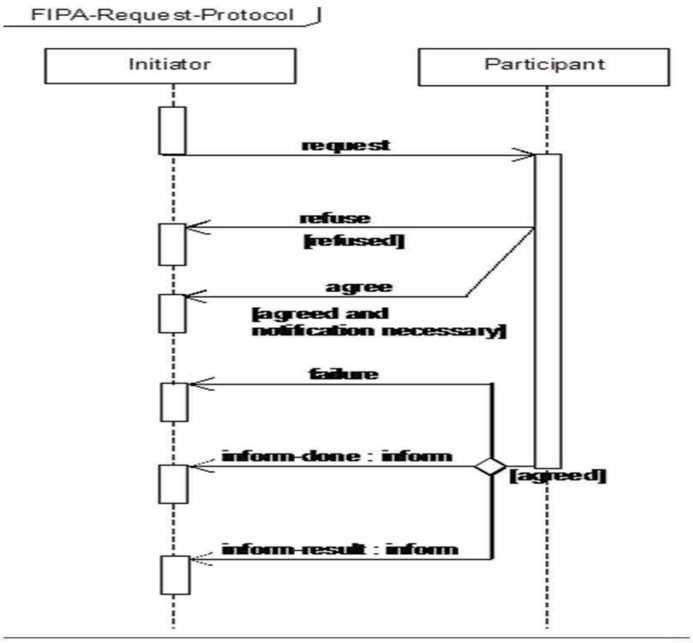


Figura 6.1: Diagrama FIPA-Request

6.2 Aplicaţie

Copilul Iniţiator va incerca sa obţină maşina din partea tatălui Responder. Va reuşi oare?

În acest exemplu vom descrie doi agenţi: Copil si Tată.

A) Agent Copil

```
import jade.core.behaviours.*;
import jade.core.Agent;
import jade.core.AID;
import jade.lang.acl.*;
// Acest agent il reprezinta pe fiul
//care ii propune tatalui sa-i imprumute masina
public class Fiu extends Agent {
public Fiu() { }
public void setup() {
System.out.println(getLocalName() +
```

```
"Tata te rog frumos sa-mi imprumuti masina!");
 addBehaviour(new MyProposeInitiator
(new AID("Tata", AID.ISLOCALNAME)));}
 protected class MyProposeInitiator
 extends SimpleBehaviour {
 int step = 0;
 AID receiverAID;
 public MyProposeInitiator(AID receiverAID){
 this.receiverAID = receiverAID;}
 public void action() {
 switch (step) {
 case 0: //formularea si trimiterea propunerii
 ACLMessage msg =
new ACLMessage(ACLMessage.PROPOSE);
msg.setSender(this.myAgent.getAID());
msg.addReceiver(receiverAID);
msg.setContent("Tata, pot sa te ...ajut,
conducand eu masina?");
System.out.println(getLocalName() +
 ": Imi incerc norocul, poate azi conduc masina?");
 send(msg);
 step = 1;
 break;
 case 1:
//a primit raspunsul de la tatal (receiverAID)?
 ACLMessage msgResponse =
receive(MessageTemplate.MatchSender(receiverAID));
 if (msgResponse != null) {
msg = msgResponse.createReply();
msg.setPerformative(ACLMessage.INFORM);
 if (msgResponse.getPerformative() =
= ACLMessage.ACCEPT_PROPOSAL) {
 msg.setContent(
 "Multumesc mult tata! Ai facut alegerea cea buna !");
```

```
}
 else if (msgResponse.getPerformative() =
= ACLMessage.REJECT_PROPOSAL) {
msg.setContent(" Data viitoare sper sa colaboram mai bine!");
}
 else {
 msg.setContent("Nu te inteleg");
 System.out.println(getLocalName()+":
"+msg.getContent());
}
 send(msg);
 step = 2;
 block();
//comportamentul e blocat pentru a nu mai ocupa CPU
}
public boolean done() {
 return step == 2;
 }}}
**************************************************
```

B) Agent Tata

```
import jade.core.*;
import jade.core.behaviours.*;
import jade.lang.acl.*;
**************************************************
// Acest agent il reprezinta pe tatal,care va ... decide,
// daca sa-i dea sau nu masina fiului
**************************************************
public class Tata extends Agent {
 double qTrustInChild =Math.random() * 1000) % 5;
 public Tata() { }
 public void setup() {
 System.out.println(getLocalName()+
":Nu se poate am nevoie de masina azi");
```

```
addBehaviour(new MyChildProposeResponder
(new AID("Copil", AID.ISLOCALNAME)));}
protected class MyChildProposeResponder extends
CyclicBehaviour { AID childAID;
public MyChildProposeResponder(AID childAID){
 this.childAID = childAID;}
public void action() {
 ACLMessage msg = receive();
 MessageTemplate.MatchSender(childAID)
 if (msg != null) {
 switch (msg.getPerformative()){
 case ACLMessage.PROPOSE:
handlePropose(msg);}
 else{
 block();}}
protected void handlePropose(ACLMessage msg) {
 ACLMessage msgReply = msg.createReply();
 System.out.println(getLocalName()
 + " " + childAID.getLocalName() +
 " imi propune: " + msg.getContent() +
"sa-i dau masina");
 double proposalWeight =(Math.random() * 1000) % 5;
 double qGoodMoodFactor = (Math.random() * 1000) % 2;
 if (qTrustInChild -proposalWeight+qGoodMoodFactor<0){
 msgReply.setPerformative
(ACLMessage.REJECT_PROPOSAL);
 msgReply.setContent
("Dragul meu copil, poate data viitoare.");}
else{
msgReply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
msgReply.setContent("Cheile sunt pe birou.Poti sa dai jos capota!");}
System.out.println(getLocalName()+":
"+msgReply.getContent());
 send(msgReply);
```

```
}}}
```

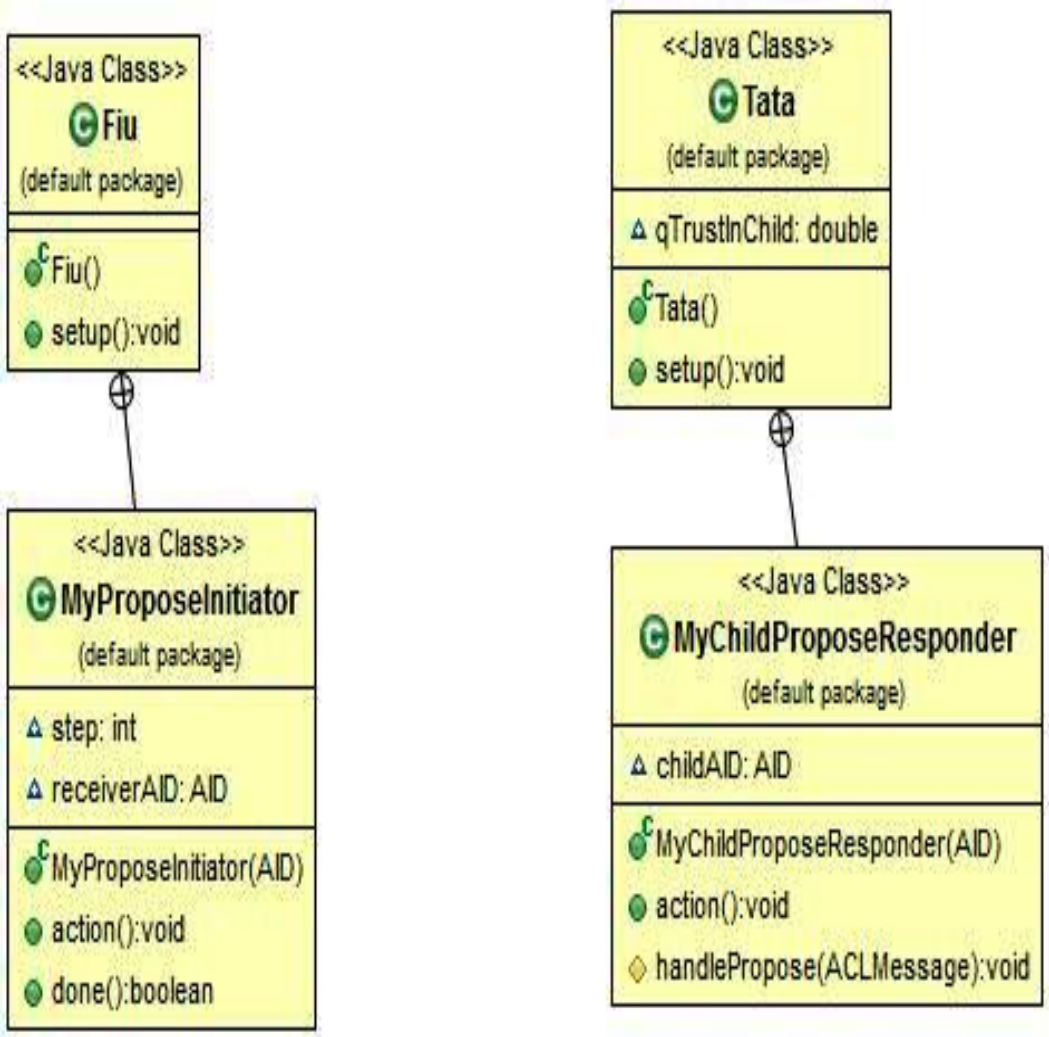


Figura 6.2: Diagrama UML Tata-Fiu

6.3 Protocolul FIPA-Contract-Net

Se foloseşte atunci când un agent doreşte să ceară unuia sau mai multor agenţi un anumit serviciu. Este utilizat în special în cazul proceselor de identificare a unui ofertant optim pentru o anumită cerere care este adresată unui set de agenţi. Agentul cu rol de iniţiator preia sarcina unui manager, care solicită **m** propuneri printr-un mesaj CFP (CALL FOR PROPOSAL) , în care specifică sarcina şi condiţiile de execuţie a sarcinii.

Dintre participanţii care primesc aceste mesaje, **n** dintre ei „nu doresc" să trimită înapoi un răspuns. Răspunsurile vor conţine un anumit număr de refuzuri şi un anumit număr de oferte. Ofertele pot conţine informaţii despre cost, timpul de execuţie a sarcinii etc. După ce a trecut termenul limită de depunere a ofertelor, iniţiatorul evaluează ofertele primite şi selectează pe baza lor un agent, mai mulţi sau nici un agent.

Agentul/agenţii acceptaţi vor primi un mesaj ACCEPT_PROPOSAL, iar restul un mesaj REJECT_PROPOSAL. Îndată ce un agent a primit mesajul de acceptare, el îşi ia angajamentul şi obligaţia de a executa sarcina. Termenul limită este esenţial pentru buna funcţionare a protocolului. În lipsa lui, iniţiatorul care aşteaptă mesaje de răspuns de la toţi participanţii, ar aştepta la infinit (blocare), dacă spre exemplu unul dintre agenţi ar eşua în a trimite un mesaj de ofertă/refuz. Acest protocol este de tip *"broadcast" (1:N)*. Agentul care recepţionează un mesaj, poate trimite în orice moment un mesaj NOT_UNDERSTOOD, în cazul în care nu a înţeles ce i s-a comunicat. Comunicarea unui astfel de mesaj, poate avea ca şi consecinţe, terminarea întregii conversaţii.

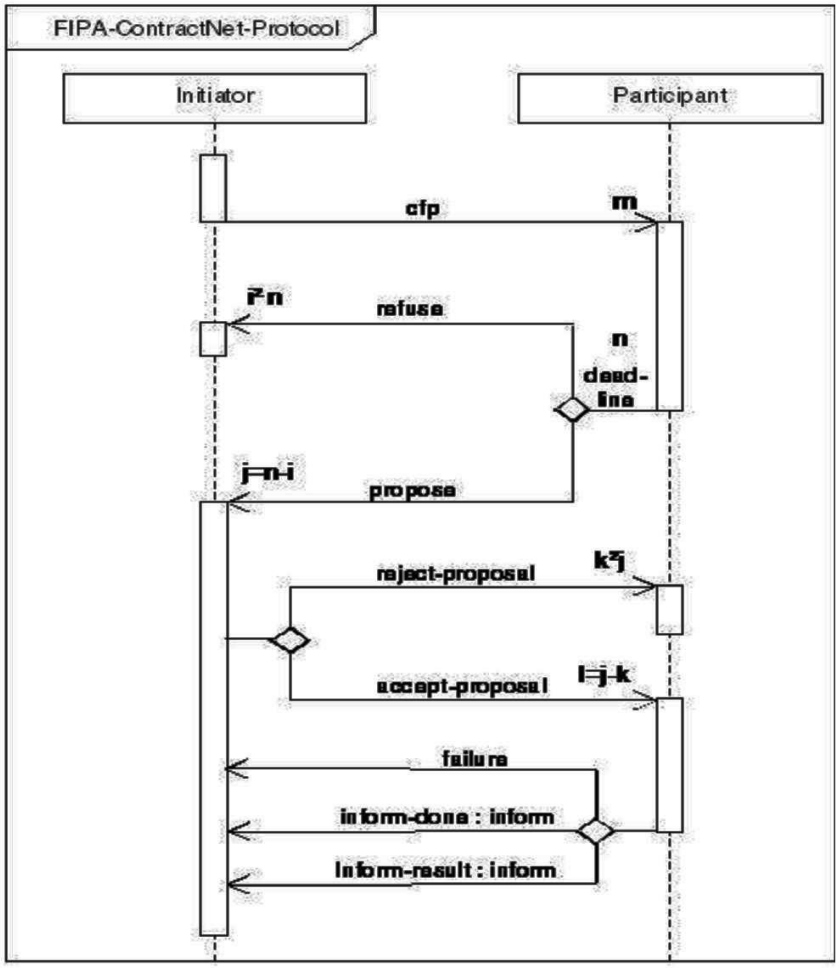


Figura 6.3: FIPA-Contract-Net

În exemplele JADE se găseşte un pachet, **examples.bookTrading [12]** ce pune la dispoziţie un sistem multi-agent ce simulează procesul de vânzare-cumpărare a unei cărţi.

Există două tipuri de agenţi care joacă rolurile de:

1. **Iniţiator (BookBuyerAgent):** un agent delegat de utilizator pentru a identifica şi achiziţiona un articol la preţul cel mai mic,

2. **Participant (BookSellerAgent):** agent tip broker ce gestionează o bază de date

cu articole pe care le poate „vinde", în funcţie de cerere. Acesta deţine o interfaţă prin care pot fi adăugate noi articole în baza sa de date.

Interacţiunea dintre cele două tipuri de agenţi corespunde protocolului FIPA Contract-Net, agentul cu rol de **iniţiator** putând astfel identifica vânzatorul cu preţ minim pentru o anumită carte.

BookBuyerAgent: primeşte drept parametru în linia de comandă numele unei cărţi pe care trebuie sa o achiziţioneze.În metoda *setup()* porneţte un comportament care la un interval de 60 de secunde va actualiza lista de agenţi vânzători `sellerAgents` şi va instanţia comportamentul `RequestPerformer` ce implementează protocolul FIPA Contract-Net şi prin care va încerca să realizeze tranzacţia.

BookSellerAgent: în metoda *setup()* se înregistrează cu DF, specificând tipul şi numele serviciului (book-trading, respectiv JADE-book-trading) cu ajutorul cărora va putea fi identificat de către cumpărători. Acesta, porneşte două comportamente tip server:

1. `OfferRequestsServer` prin care va răspunde cererilor de cărţi (mesaje tip **cfp** – **call for proposa**l),

2. `PurchaseOrdersServer` prin care se realizează tranzacţia, agentul răspunzând mesajelor tip ACCEPT_PROPOSAL.

De asemenea, agentul dispune de o interfaţă prin care este actualizată dinamic baza sa de date (catalogul de cărţi). Interfaţa primeste ca parametru în constructor o referinţă către agent, astfel încât din ea se apelează direct metoda *updateCatalog()* din interiorul agentului ce realizează această operaţie.

6.4 FIPA-Contract-Net-Interaction Protocol

6.4.1 Aspecte generale

FIPA Contract Net Interaction Protocol (IP) este o variantă puţin modificată a versiunii iniţiale a algoritmului Contract Net, în sensul că adaugă acţiunile de respingere şi confirmare în cadrul procesului de comunicare. În cadrul Contract Net IP, un agent Iniţiator va prelua rolul de manager care doreşte să delege sarcini către unul sau mai mulţi agenţi participanţi. Ininiţiatorul doreşte să optimizeze execuţia sarcinii respective, în sensul de a o efectua cu costuri cât mai scăzute. Acest aspect este de obicei exprimat ca preţ,

timp de finalizare, distribuirea echitabilă a sarcinilor etc. Pentru o sarcină dată, la solicitarea iniţiatorului, orice agent participant poate răspunde cu o propunere sau poate refuza. Negocierile vor continua cu participanţii care au trimis o propunere. În sistem pot fi mai mulţi agenţi din ambele categorii, însă între fiecare pereche iniţiator-participant comunicarea se desfaşoară, astfel:

- (a) - Agenţii iniţiatori trimit participanţilor mesaje de tip **Call for Proposals,** unde solicită de la aceştia soluţii de rezolvare a unor probleme.

- (b) - Agenţii participanţi pot răspunde cu o propunere sau pot refuza acest lucru (caz în care comunicarea se încheie). De exemplu, într-un sistem tranzacţional, participanţii propun sau refuză trimiterea de propuneri dupa cum se pot sau nu conforma criteriilor impuse în **Call for Proposals** (ex. nu pot furniza anumite bunuri, sau nu le pot vinde sub un anumit preţ etc.).

- (c) - Iniţiatorii evalueaza propunerile primite şi decid care din acestea sunt acceptabile. Criteriile conform cărora propunerile corespund nevoilor iniţiatorilor variază de la caz la caz. Iniţiatorii trimit participanţilor mesaje de acceptare sau respingere a propunerii, după caz.

- (d) - Patricipantii cărora le-au fost acceptate propunerile trebuie acum să ducă la îndeplinire sarcinile aferente. Ei vor răspunde iniţiatorilor cu mesaje de confirmare a reuşitei sau eşecului ducerii la bun-sfarşit a propunerii, dupa caz.

Interacţiunea poate fi identificata în mod unic printr-un parametru **conversationId** asignat de iniţiator. Acest lucru permite fiecarui agent să gestioneze comunicarea: de exemplu, permite unui agent sa identifice conversaţiile cu un anumit agent initiator.

6.4.2 Excepţii de la execuţia normală a protocolului

În orice moment al comunicării, receptorul mesajului poate informa expeditorul că nu a înţeles ce a fost comunicat. Acest lucru este realizat prin returnarea unui mesaj `not_understood`. Comunicarea faptului că nu s-a înţeles ce s-a recepţionat în cadrul unui protocol de interacţiune poate implica faptul că orice angajamente asumate în timpul interacţiunii nu sunt valabile. Acest lucru poate duce la rezilierea întregului protocol sau numai a comunicării cu agentul de la care s-a primit mesajul `not_understood`.

6.5 Alte protocoale de interacţiune

6.5.1 FIPA Query

Iniţiatorul cere Participantului să realizeze o acţiune de tipul `inform` utilizând unul din cele două performative: `query-if` sau `query-ref.` Performativa `query-if` este utilizată când Iniţiatorul doreşte să afle dacă o anumită propoziţie este adevărată sau falsă, iar performativa `query-ref` când iniţiatorul doreşte să afle ceva despre un anumit obiect, trimis ca referinţă în conţinutul mesajului `query-ref.` Participantul procesează mesajul `query-if` sau `query-ref` şi ia decizia dacă acceptă sau nu cererea respectivă. Dacă condiţiile presupun necesitatea unui acord explicit atunci participantul comunică un mesaj de tipul `agree` (acesta poate fi opţional în funcţie de circumstanţe). Dacă răspunsul este favorabil, participantul răspunde cu unul din cele două alternative pentru performativa inform: `inform-t/f` ca răspuns la un query-if când conţinutul mesajului `inform-t/f` este evaluat ca fiind adevărat sau fals sau `inform-result` ca răspuns la un `query-ref`.

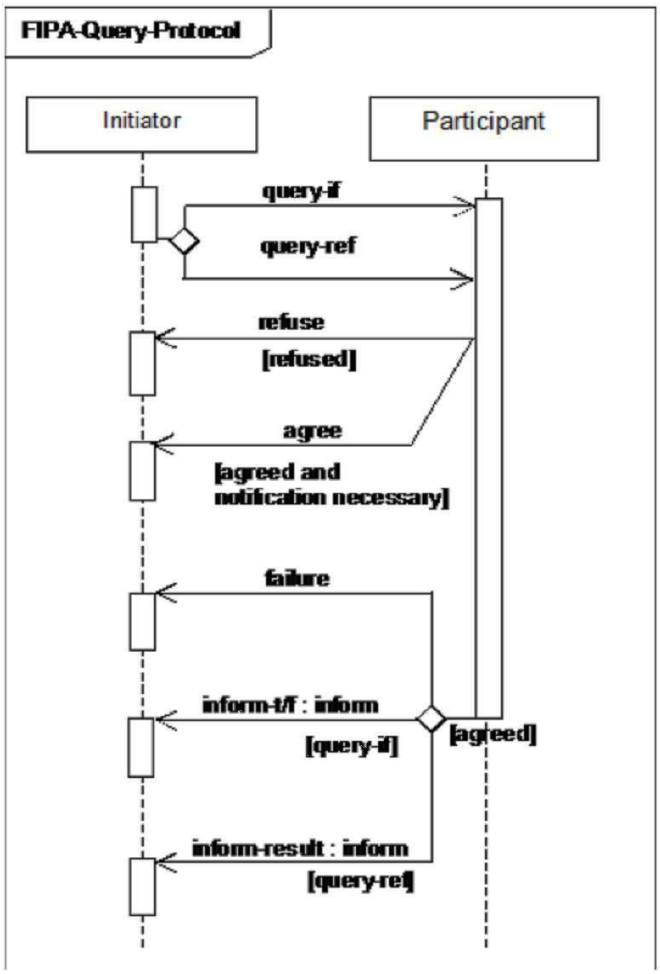


Figura 6.4: FIPA Query

6.5.2 FIPA Request-When

Iniţiatorul utilizează performativa `request-when` să ceară Participantului realizarea unei acţiuni când sunt îndeplinite anumite precondiţii. Dacă cererea este înţeleasă şi nu a fost refuzată în prealabil va răspunde cu un mesaj de tipul **agree**, aşteptând ca precondiţia invocată să fie îndeplinită pentru a executa acţiunea dorită de iniţiator şi a-l informa de rezultatul execuţiei acesteia (`inform-done`, `inform-result`). Dacă participantul nu mai este capabil să realizeze acţiunea va trimite un mesaj de tipul `failure`.

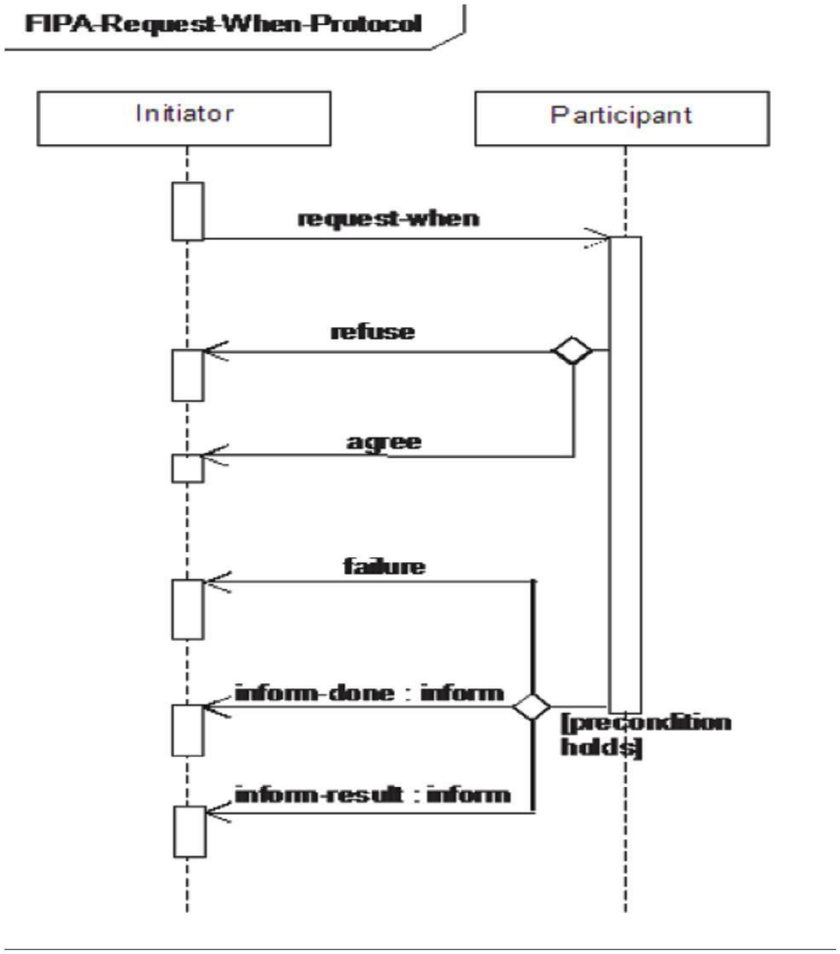


Figura 6.5: FIPA RequestWhen

6.5.3 FIPA-Propose

Iniţiatorul trimite un mesaj propose participanţilor indicând că doreşte ca cei din urmă să execute o anumită acţiune dacă aceştia sunt de accord. Răspunsul participanţilor poate fi de acceptare sau de refuz: `acceptproposal` sau `reject-proposal.` Finalizarea acestui protocol de interacţiune cu o performativă `accept-proposal` va fi urmată în mod normal de evaluarea acţiunii de catre iniţiator şi înştiinţarea participantului de rezultatul acesteia.

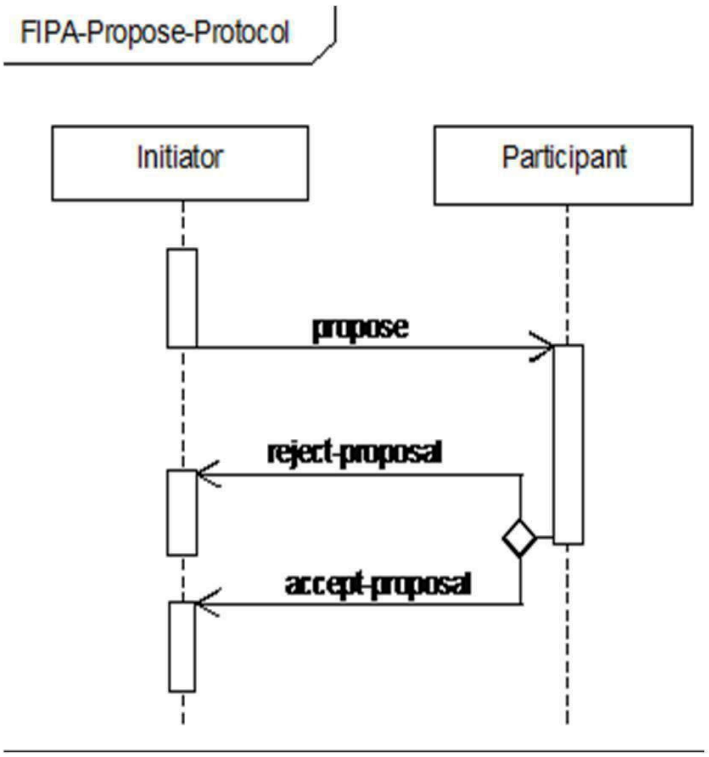


Figura 6.6: FIPA-Propose

6.5.4 FIPA English Auction

În acest protocol vânzătorul încearcă să găsească preţul de piaţă pentru un bun propunând iniţial un preţ sub cel presupus după care îl creşte în mod gradual similar licitaţiilor curente. De fiecare dată când preţul este anunţat, vânzătorul aşteaptă să vadă dacă există cumpărători care doresc să cumpere la preţul propus. Imediat ce un cumpărător va accepta preţul, vânzătorul trimite o nouă propunere cu un nou preţ de strigare. Licitaţia continuă până în momentul în care nu mai există nici un cumpărător dispus să plătească preţul cerut.

Dacă ultimul preţ care a fost acceptat de un cumpărător este mai mare decât preţul acceptabil de vânzare (cunoscut doar de vânzător), bunul este vândut cumpărătorului la preţul stabilit, dacă nu, bunul nu va mai fi vândut. În mod normal, după finalizarea acestui protocol vânzătorul va intra într-un protocol de `tipul FIPA-Request` pentru a finaliza tranzacţia de cumpărare.

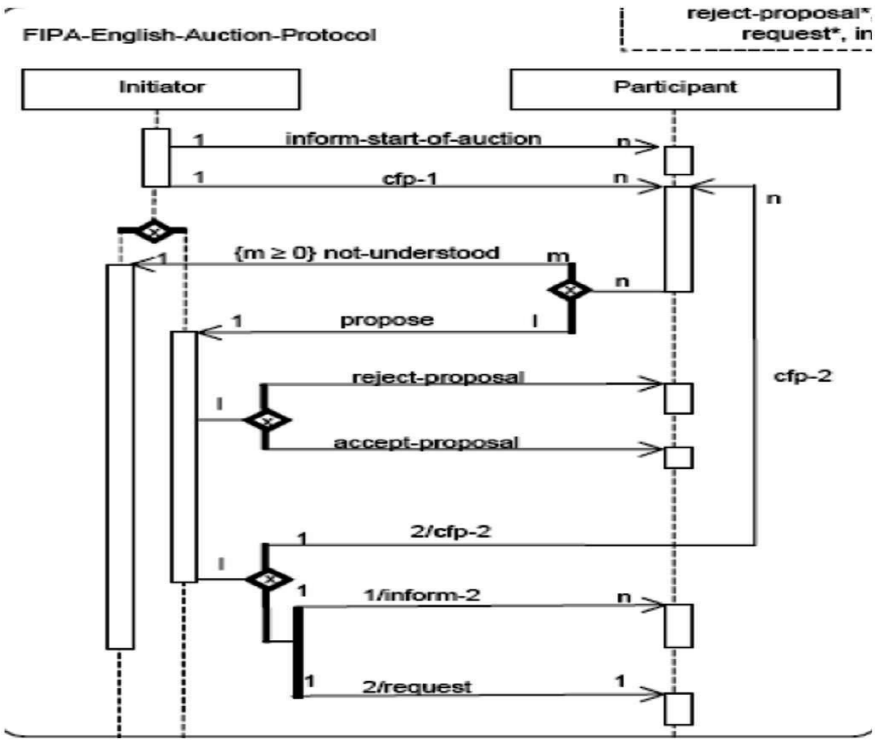


Figura 6.7: FIPA-English-Auction

6.5.5 FIPA Brokering

Este un macro-protocol de interacţiune deoarece utilizează performative de tipul `proxy` incluse în mesajele de intermediere ce structurează restul interacţiunii. Iniţiatorul începe interacţiunea cu un mesaj de tip **proxy** care conţine: o expresie ce face referinţă la agenţii vizaţi la care `Broker-ul` ar trebui să transmită mai departe mesajul, performativa care ar trebui trimisă şi un set de condiţii (de exemplu numărul maxim de agenţi la care trebui trimis mesajul). Broker-ul procesează cererea şi ia o decizie dacă este de acord sau nu cu executarea cererii (un răspuns de refuz finalizează interacţiunea). Odată ce Broker-ul a fost de acord să joace rolul de `proxy`, identifică agenţii în funcţie de descrierea primită în conţinutul mesajul `proxy.` Dacă nici un astfel de agent nu poate fi identificat, Broker-ul returnează `failure-no-match`, iar interacţiunea este terminată. În caz contrar, Broker-ul poate modifica lista agenţilor cărora le transmite mesajul în funcţie de condiţiile stabilite în parametrul `proxy-condition`. În acest moment Broker-ul începe m interacţiuni cu n agenţi din lista rezultată anterior, fiecare din aceste interacţiuni executându-se în propriul sub-protocol. De observat că natura sub-protocolului şi a rezultatelor sunt în funcţie specificaţia dată în performativa mesajului `proxy`. Pe măsură ce sub-protocolul progresează, Broker-ul trimite răspunsurile primite Iniţiatorului, iar când sub-protocoalele sunt finalizate, trimite un mesaj final te tipul `reply-message` şi interacţiunea se termină.

Mai multe detalii despre aceste protocoale se pot găsii [24, pp. 140-152].

6.5.6 FIPA Dutch Auction

În acest protocol vânzătorul încearcă să identifice preţul de piaţă al unui bun prin iniţializarea licitaţiei la un preţ mai mare decât cel presupus, după care îl reduce succesiv până când un cumpărător acceptă preţul propus. Marja de reducere a preţului este la latitudinea vânzătorului însă de obicei aceştia au o limită inferioară sub care tranzacţia nu mai are loc.

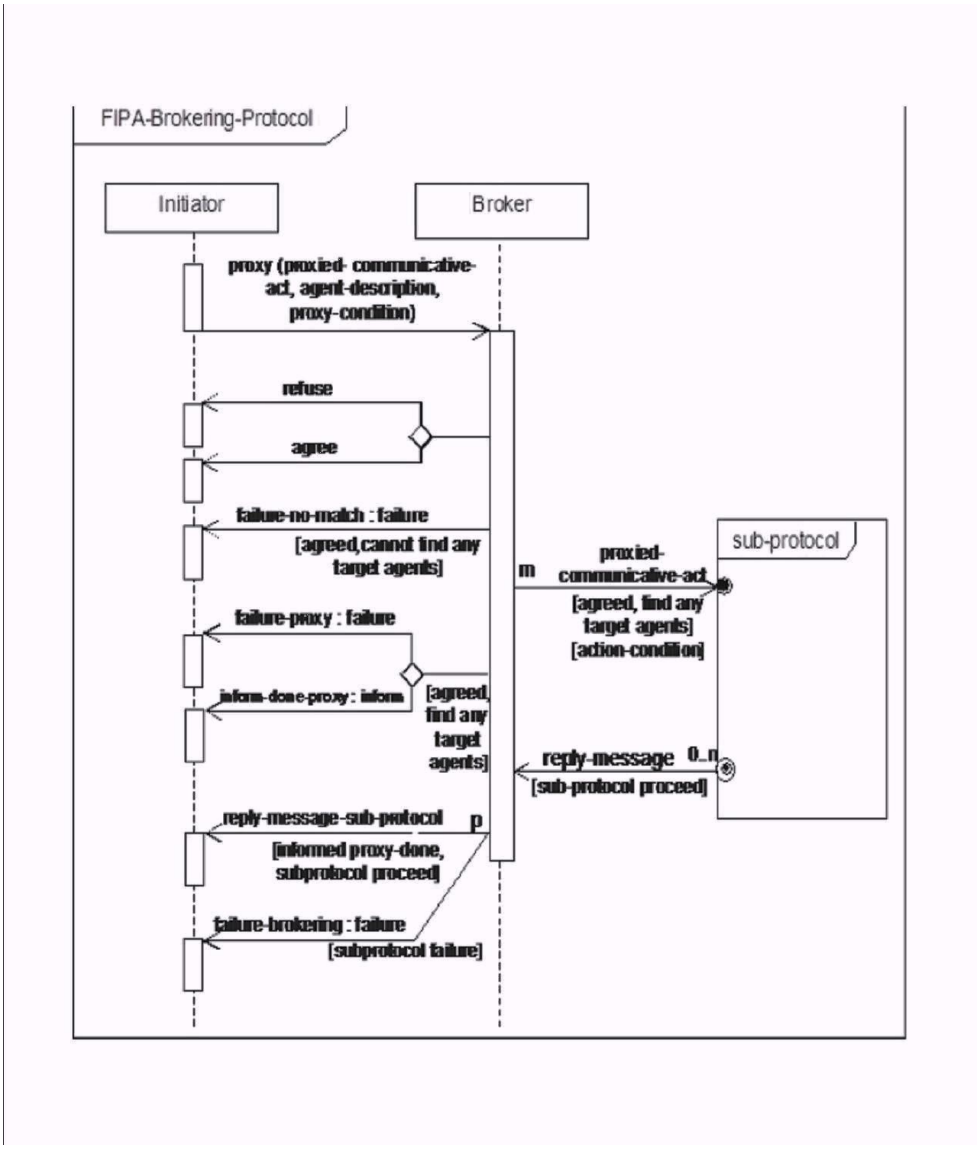


Figura 6.8: FIPA-Brokering

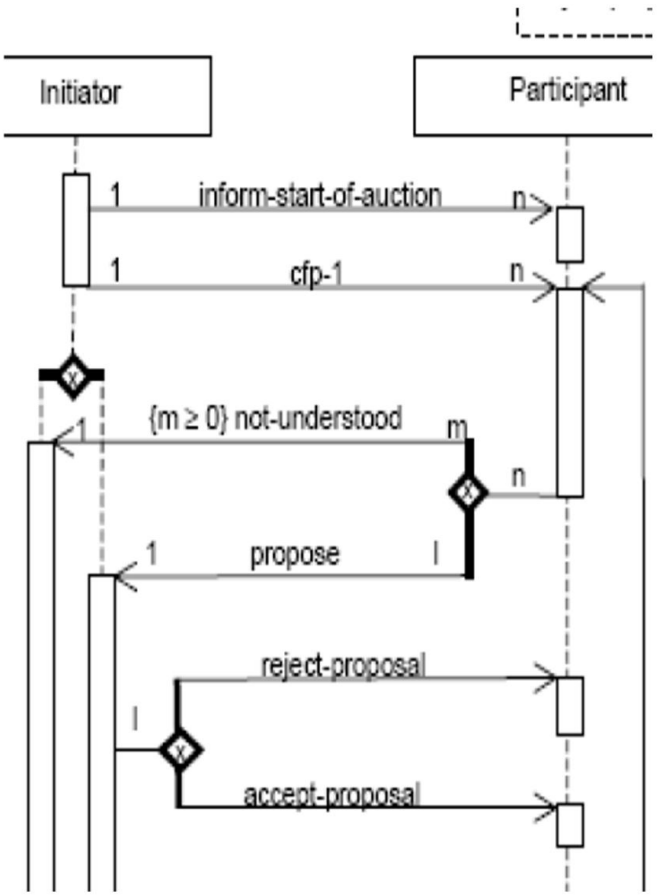


Figura 6.9: FIPA-Dutch-Action

6.5.7 FIPA- Subscribe

Iniţiatorul începe interacţiunea cu un mesaj de subscriere ce conţine referinţa la obiectul în care este interesat. Participantul procesează mesajul `subscribe` şi ia o decizie de a accepta sau nu cererea. Dacă condiţiile protocolului implică un acord explicit ("notification necessary" este true), atunci participantul comunică cu un mesaj de `agree`. Dacă cererea este acceptată, participantul răspunde cu un mesaj `inform-result` având în conţinut o expresie ce face referinţă la obiectul subscris. Participantul continuă cu trimiterea unui mesaj `inform-result`, dacă obiectul trimis ca referinţă s-a modificat între timp. Dacă la un moment dat, după ce Participantul a fost de acord cu cererea respectivă, nu mai este capabil (din varii motive) să proceseze cererea, atunci trimite un mesaj `failure` care implică automat şi finalizarea interacţiunii. Altfel protocolul poate fi încheiat de Iniţiator utilizând un `meta-protocol de tip cancel`.

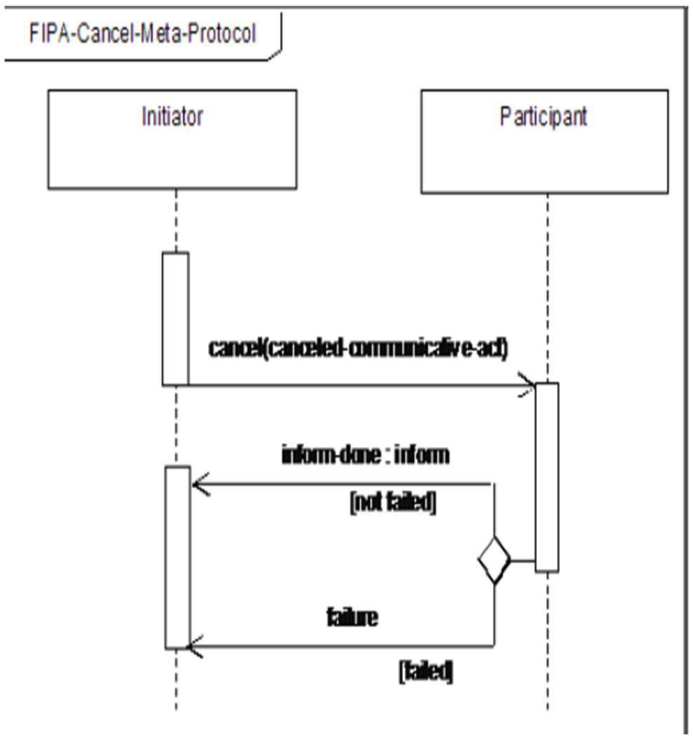


Figura 6.10: FIPA-Subscribe

6.5.8 FIPA Recruiting

La fel ca protocolul anterior, FIPA Recruiting este un macro-protocol deoarece utilizează performative proxy pentru a transmite protocolul de interacţiune real în conţinutul mesajelor. Iniţiatorul protocolului începe interacţiunea cu un mesaj de tip proxy care conţine: agenţii cărora urmează să le fie transmis mesajul împreună cu setul de condiţii care filtrează numărul şi caracteristicile acestora şi performativa. Recruiter-ul procesează cererea şi decide dacă acceptă sau nu cererea venită din partea Iniţiatorului (refuzul acesteia încheie interacţiunea). Odată ce Recruiter-ul este de acord să servească pe post de proxy cererea Iniţiatorului, va localiza agenţii în funcţie de condiţiile stabilite în conţinutul mesajului iniţial de cerere. Dacă nici un astfel de agent nu poate fi identificat Recruiter-ul va returna un mesaj de tipul failure-no-match şi interacţiunea încetează. În caz contrar, Recruiter-ul poate modifica lista iniţială a agenţilor, în funcţie de parametrul proxy-condition, după care startează m sub-protocoale de interacţiune cu fiecare din cei n agenţi din listă. Dacă Recruiter-ul a primit un mesaj cu parametrul designated-receiver de la Iniţiator el trebui să starteze fiecare sub-protocol cu un mesaj având pentru parametrul reply-to identificatorul agentului Receptor Desemnat şi pentru parametrul conversation-id valoarea identificatorului conversaţiei iniţiale.

De remarcat că sub-protocolul de interacţiune dintre Recruiter şi agenţi este specificat în mesajul iniţial de tip proxy. Pe măsură ce sub-protocoalele avansează, Recruiter-ul retransmite mesajele primite fie Receptorului Desemnat, fie Iniţiatorului în funcţie de valoarea parametrului reply-to din mesajul iniţial de tip proxy. Aceste mesaje sunt de tipul reply-message-sub-protocol. Când sub-protocoalele de interacţiune sunt finalizate, Recruiter-ul trimite un mesaj final de tipul reply-message-sub-protocol. În cazul unor interacţiuni 1:N, în cazul protocoalelor sau sub-protocoalelor, Iniţiatorul este liber să decidă dacă va trebui să fie utilizat acelaşi atribut pentru conversation-id sau va genera unul nou. În plus, mesajele pot specifica alte informaţii legate de protocolul de interacţiune precum termenul limită în parametrul reply-by.

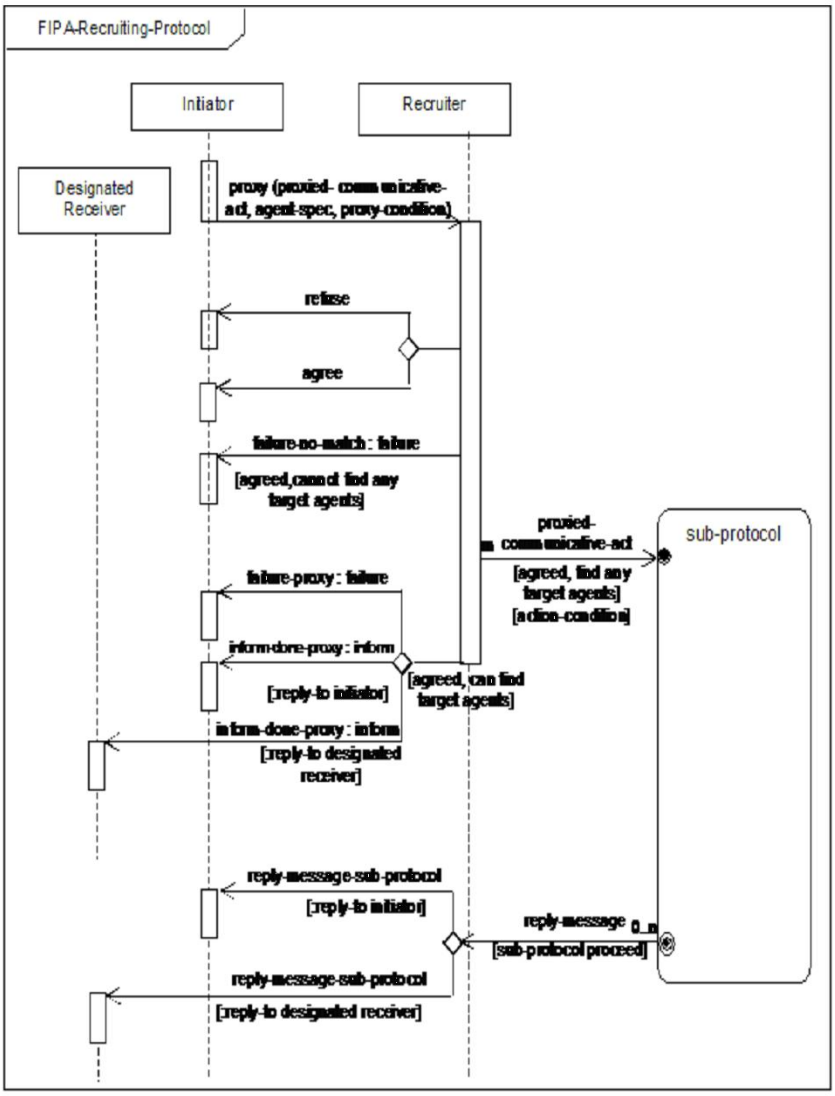


Figura 6.11: FIPA-Recruiting

6.6 Probleme propuse

1. Executaţi o tranzacţie pe Internet, având cel puţin doi agenţi tip `BookSellerAgent` porniţi. Se va folosii un agent tip `DummyAgent` pentru `Iniţiator`.

2. Implementaţi o statistică la nivelul clienţilor unei firme cu vânzări produse cosmetice on-line, astfel încât pentru cei fideli să puteţi oferi reduceri (calculate eventual în funcţie de anumite valori de prag).

Capitolul 7

Ontologii

7.1 Noţiuni teoretice

Interacţiunea între agenţi presupune transmiterea unor anumite informaţii `I`, prin intermediul unui mesaj `ACL`, de la expeditor către destinatar. În interiorul mesajului, informaţia `I` este reprezentată conform unui limbaj descriptiv (ex. `SL` sau `LEAP`) şi codificată într-un anumit format (ex. şir de caractere).

Modalitatea de reprezentare a informaţiei `I` în interiorul agenţilor poate diferi de la un agent la altul, însă semantica şi sensul acesteia trebuie să fie acelaşi de ambele părţi. Pentru manipularea internă a informaţiei `I` se folosesc clase şi obiecte, iar pentru vehicularea externă, este necesară serializarea lor (transformarea în şir de caractere) şi transmiterea prin intermediul mesajelor ACL.

De fiecare dată când un agent `A` trimite o informaţie `I` unui alt agent `B`:

- 1) agentul A trebuie să transforme informaţia I, din reprezentarea sa internă (obiectuală) în expresia ACL corespunzătoare, iar agentul B trebuie să realizeze operaţia inversă;
- 2) agentul B trebuie să efectueze un şir de operaţii de validare a conţinutului informaţiei I (de exemplu frecvenţa să fie un număr valid). Validarea se realizează prin verificarea unor anumite reguli, stipulate în ontologia folosită de ambii agenţi în cadrul conversaţiei.

JADE permite definirea de ontologii şi limbaje descriptive (`content languages`), realizând automat cele două operaţii prezentate anterior. Astfel, programatorul va lucra

obiectual, fiind degrevat de necesitatea implementării etapei de codificarea/decodificarea la nivelul mesajelor `ACL`.

Aceste două operaţii sunt realizate de un obiect, un manager de conţinut (content manager), instanţă a clasei **ContentManager,** din pachetul jade.content. Aceasta furnizează metode de conversie a obiectelor în şiruri de caractere, de includere a acestora în slotul de conţinut al mesajului `ACL` şi vice-versa. Managerul pune la dispozitie doar interfeţe pentru realizarea acestor operaţii, execuţia realizându-se prin delegarea către o ontologie şi un codec a acestor sarcini. Ontologia validează din punct de vedere semantic informaţia ce trebuie convertită în timp ce `Codec-ul` o transformă în şiruri de caractere, conform gramaticii limbajului descriptiv folosit.

Fiecare agent are un astfel de manager de conţinut, accesabil prin metoda *getContentManager()* a clasei **jade.core.Agent.**

7.2 Atomi lexicali

În gramatica unui limbaj descriptiv se identifică următoarele elemente lexicale:

- **Predicate**: expresii descriptive cu valori de adevăr booleane (true sau false).

Exemplu:(Prezent (Persoana :nume Student)
(Laborator: nume SMA)) declară faptul că persoana Student a fost prezentă la cursul SMA. Acestea sunt folositoare în mesaje tip QUERY-IF, INFORM, dar nu în REQUEST;

- **Termeni**: expresii ce identifică entităţi (concrete sau abstracte) ce există în sistem şi despre care agenţii "vorbesc" sau raţionează. Acestea se clasifica în:

1) **Concepte**: expresii ce descriu entităţi complexe.

Exemplu: (Persoana :nume Pop :prenume:Daniel) . Acestea sunt folosite ca parte a dialogului (în predicate, query-uri etc.)

Exemplu:(Carte: titlu "Învierea": autor (Persoana :nume Lev Tolstoi)).

2) **Acţiuni:** concepte speciale ce denotă acţiuni ce pot fi îndeplinite de agenţi. Acestea sunt utile în cazul mesajelor tip REQUEST.

Exemplu:(Vânzare (Carte: titlu Crimă şi pedeapsă, Persoana: nume N.F. Dostoievski)).

3) **Primitive:** expresii ce identifică entităţi atomice (întregi, şiruri de caractere etc.).

4) **Agregate:** expresii ce identifică entităţi ce conţin grupuri de entităţi.

Exemplu: (echipa (Persoana: nume Jucator_1) (Persoana: nume Jucator_2)).

5) **IRE (Identifying Referential Expressions):** expresii ce identifică entităţile pentru care un anumit predicat este adevărat.

Exemplu: toţi ? x (Prezent ? x (Laborator :nume SMA)) va identifica toate elementele x pentru care predicatul (Prezent x (Laborator :nume SMA)) este adevărat, deci va identifica toţi studenţii care au fost prezenţi la laboratorul de SMA. Sunt utilizate în mesaje tip QUERY_REF şi necesită folosirea variabilelor.

6) **Variabile:** expresii ce identifică un element generic, necunoscut apriori (folosite de obicei în query-uri).

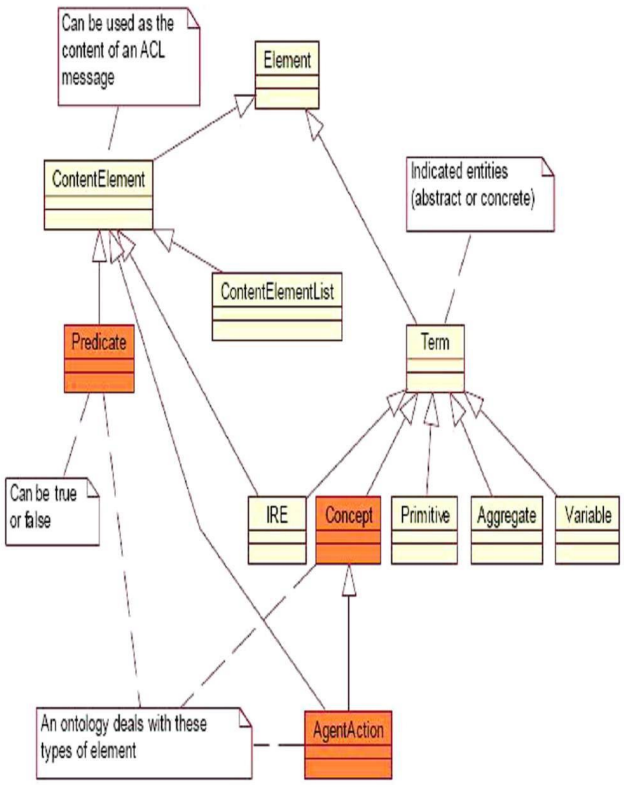


Figura 7.1: Modelul de referinţă al conţinutului

Un limbaj descriptiv complet trebuie să poată reprezenta şi distinge toţi aceşti atomi lexicali.

O ontologie pentru un anumit domeniu conţine un set de definiţii reprezentând structura predicatelor, acţiunilor si conceptelor ce aparţin acestuia.

7.3 Implementare JADE

Folosirea limbajelor descriptive şi a ontologiilor în cadrul platformei JADE, pentru a permite agenţilor să comunice şi raţioneze asupra unor fapte ori obiecte aparţinând unui anumit domeniu necesită următoarele:

- a) definirea unei ontologii pentru domeniul abordat, conţinând definiţiile predicatelor, acţiunilor şi conceptelor corespunzătoare acestuia;
- b) realizarea claselor ce vor descrie aceste predicate, acţiuni, concepte;
- c) selectarea unui limbaj descriptiv compatibil din cadrul celor oferite de JADE. Se pot defini noi limbaje (prin extinderea celor deja existente), dar în majoritatea cazurilor nu va fi necesar;
- d) înregistrarea la nivelul agentului a ontologiei definite şi a limbajului descriptiv selectat;
- e) crearea şi utilizarea expresiilor descriptive sub forma obiectelor Java ce sunt instanţe ale claselor definite în la punctul b) şi folosirea suportului JADE pentru transformarea acestora în şiruri de caractere.

7.4 Definirea unei ontologii

În JADE, o ontologie este o instanţă a clasei **jade.content.onto.Ontology**, căreia i-au fost adăugate definiţii (scheme) pentru predicatele, acţiunile şi conceptele relevante domeniului abordat. Acestea sunt instanţe ale claselor *PredicateSchema, ActionSchema, ConceptSchema* aflate în pachetul:
jade.content.schema.

Conceptul de AID, necesar identificării agenţilor la nivelul ontologiei, este deja definit în ontologia de baza `BasicOntology` pe care noile ontologii o extind.

Aceasta include scheme pentru:

- a) tipurile primitive (integer, long, etc);
- b) tipul agregat;
- c) câteva elemente generice (printre care şi conceptul de AID).

Reţinem:

1. Ontologia are o denumire şi un vocabular pe baza căruia se va configura structura (schema) sa de predicate, acţiuni şi concepte.
2. Un element din conţinutul unei scheme poate fi obligatoriu (MANDATORY), caz în care nu poate fi NULL. De asemenea, se poate defini cardinalitatea acestuia (De exemplu o carte poate avea un autor sau mai mulţi).
3. Pentru a degreva ontologia de complexitatea vocabularului aferent, se foloseşte un design-pattern prin care vocabularul este definit la nivelul unor interfeţe pe care ontologiile le implementează.

7.5 Dezvoltarea claselor ontologice

În dezvoltarea acestor clase, vom avea în vedere corespondenţa dintre acestea şi schemele descrise anterior. Astfel, eventuala ierarhizare a schemelor (prin extindere) trebuie să se reflecte şi în ierarhia claselor descriptive.Se vor respecta urmatoarele reguli:

- a) în funcţie de tipul schemei, clasa trebuie să implementeze o anumită interfaţă (de exemplu AgentAction pentru o acţiune);
- b) ierarhia schemelor trebuie să se regăsească şi la nivelul claselor descriptive;
- c) fiecare clasă trebuie să aibă membri şi metode de acces corespunzătoare schemei descrise (membrii claselor vor avea numele şi tipul conform structurii:

1. `conceptului,`
2. `acţiunii,`
3. `predicatului implementat;`

În cazul elementelor având cardinalitate > 1, clasa trebuie să mai includă metodele de accesare:

```
public void setNnn(jade.util.leap.List l);
public jade.util.leap.List getNnn();
```

7.6 Selectarea unui limbaj descriptiv

Pachetul **jade.content** conţine codecuri pentru limbajele descriptive: SL şi LEAP. Un codec pentru un limbaj descriptiv L este un obiect Java ce lucrează cu expresii descriptive scrise în limbajul L. În majoritatea cazurilor, aceste două codec-uri sunt suficiente pentru susţinerea procesului de comunicare, dezvoltatorul nefiind obligat să lucreze şi asupra acestei secţiuni a sistemului.

Limbajul `SL` este un limbaj codificat în mod text, inteligibil utilizatorului uman şi este probabil cel mai folosit limbaj descriptiv folosit la nivelul comunităţii ştiinţifice. Este preferat a fi folosit în cadrul sistemelor multi-agent deschise, în care agenţii interacţionează cu agenţi externi sistemului, de o natura poate diferită.

De asemenea acesta facilitează procesele de testare şi depanare, datorită codificării sale în mod text. Conţine un set de operatori utili: binari (AND, OR, NOT), modali (BELIEF, INTENTION, UNCERTAINTY). SL este orientat spre acţiunile agenţilor, de aceea toate aceste acţiuni trebuie inserate în constructul ACTION (definit în `BasicOntology`) prin care o acţiune este asociată cu AID-ul agentului care o va efectua.

Limbajul `LEAP` este un limbaj codificat binar, neinteligibil din punct de vedere uman şi a fost dezvoltat special pentru JADE, deci nu va putea fi folosit decât de agenţi ai platformei JADE. Este preferat a fi folosit în cazuri de constrângeri asupra memoriei necesare (codec-ul LEAP fiind mai compact decat cel SL) ori în cazul în care este necesară transmiterea unui şir de byte-s, ce nu poate fi realizată prin SL.

7.7 Înregistrarea ontologiilor şi limbajelor descriptive la nivelul agentului

Premergător folosirii de către agent a unei ontologii ori limbaj descriptiv, acestea trebuie sa fie înregistrate în managerul de conţinut (**ContentManager-ul**) al agentului. Această operaţie se face de obicei în cadrul fazei de activare, în metoda *setup()*. Odată ce au fost înregistrate, **ContentManager-ul** va asocia codec-ul şi ontologia string-urilor returnate de metodele lor *getName()*.

7.8 Folosirea limbajelor descriptive şi a ontologiilor

7.8.1 Combinarea ontologiilor

Similar operaţiilor ce se pot efectua asupra claselor şi ontologiile pot fi combinate, extinse, fiind posibilă astfel reutilizarea codului. O ontologie poate extinde una sau mai multe ontologii, prin specificarea acestora ca parametri in contructorul acesteia. Spre exemplu, ontologia MusicShopOntology poate extinde o alta ontologie, CDOntology, în care să fie definite:

- concepte,
- predicate,
- acţiuni generice.

7.8.2 Descriptori abstracţi

Deşi manipularea informaţiilor vehiculate este facilitată prin reprezentarea acestora cu ajutorul obiectelor Java, există situaţii când acestea reprezintă un impediment:

- a) în cazul unei ontologii ce include mii de elemente, este necesară definirea a mii de clase ontologice. Chiar dacă acestea s-ar genera automat, există contexte cu anumite restricţii (limitări de memorie pe anumite echipamente portabile etc.) pentru care sistemul nu ar fi fezabil;

- b) conceptele unei ontologii pot fi definite prin moştenire multiplă, ceea ce ar impune realizarea moştenirii multiple şi la nivelul claselor ontologice însă în Java nu este posibilă aceasta.

Query-urile nu pot fi reprezentate datorită ambiguităţilor. În consecinţă, JADE furnizează o alta metodă, generică, de reprezentare a expresiilor descriptive: fiecare element poate fi definit printr-un descriptor abstract ce conţine un nume de tip (numele tipul elementului) şi un set de atribute. Spre exemplu, conceptul:

(F1Driver :nume Alonso: team BMV)

poate fi reprezentat printr-o instanţa a clasei *AbsConcept* (un descriptor abstract de tip concept) având numele tipului F1Driver, atributul nume setat cu “Alonso”, iar atributul team cu BMV. Fiecare tip de element din modelul de referinţa are definit un descriptor abstract, aceştia fiind practic folosiţi pentru reprezentarea tuturor elementelor (deci si insţantele claselor ontologice sunt convertite în descriptori abstracţi, utilizandu-se astfel o

metoda uniformă de reprezentare). Pachetul **jade.content.abs** are implementările acestor descriptori abstracţi.

Observaţie 13 *a) Se poate folosi obiectul ontology pentru a obţine descriptorul abstract al AID;*

b) se poate lucra combinat, folosind atât clase ontologice cât şi descriptori abstracţi ce sunt necesari în special în cazul query-urilor.

7.8.3 Fazele conversiei

Chiar dacă la nivelul codului se utilizează doar clase ontologice, JADE foloseşte descriptori abstracţi când procesează expresii descriptive. Astfel:

- a) Codec-ul specificat transformă în (din) instanţe ale AbsContentElement (descriptori abstracţi);
- b) Elementul AbsContentElement este validat corespunzător definiţiei sale (din ontologie);
- c) Ontologia specificată transformă elementul `AbsContentElement` într-un (dintr-un) obiect Java, instanţă a clasei ontologice ce implementează interfaţa corespunzatoare:
  tip `ContentElement`.

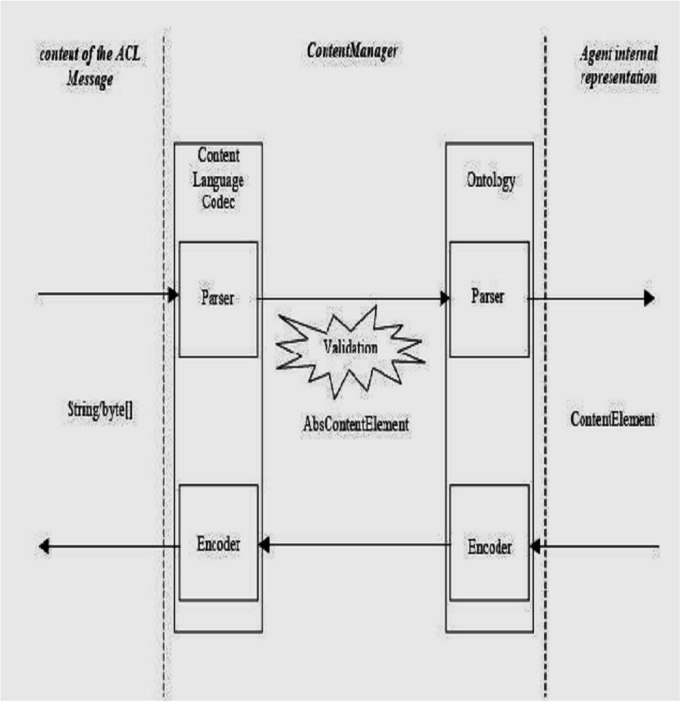


Figura 7.2: Fazele conversaţiei informaţiei

7.8.4 Operatori ai limbajelor descriptive

În general, un limbaj descriptiv defineşte atât o sintaxă pentru expresiile descriptive, cât şi un set de operatori (logici etc.). Similar elementelor ontologiei, un operator poate fi definit construind o schemă ce specifică structura tuturor expresiilor bazate pe acest operator. Corespunzător semanticii operatorului, schema poate fi de tip *PredicateSchema, ConceptSchema* etc. Astfel, întregul set de operatori poate fi definit la nivelul unei ontologii aparţinând codec-ului ce poate fi accesat prin *metoda getInnerOntology().*

Limbajul SL deţine un set complex de operatori, plecând de la cei logici (`AND, OR, NOT`), până la operatori modali şi de acţiune (`ACTION, ;, |`), fiecare având o definiţie proprie în interiorul ontologiei codec-ului. Doar operatorii folosiţi în limbajul SL0 au clase ontologice definite, structurarea fiind realizată la nivelul ontologiei `BasicOntology`. Aceşti operatori sunt folosiţi la crearea expresiilor ce sunt referite în definiţia semanticii `ACL`, pentru ceilalţi operatori este necesară folosirea descriptorilor abstracţi.

Întregul set de denumiri pentru operatori este definit sub formă de constante în interfaţa:
jade.content.lang.sl.SLVocabulary.

7.8.5 Crearea query-urilor

Utilizarea descriptorilor abstracţi este necesară în cazul query-urilor. În special se folosesc:

- a) AbsIRE: un descriptor abstract reprezentând un IRE;
- b) AbsVariable: un descriptor abstract reprezentâd o variabilă.

Un IRE include întotdeauna un predicat şi o variabilă, de aceea clasa AbsIRE are metode de accesare specificate pentru aceştia.

7.8.6 Anularea verificărilor semantice

După cum a fost menţionat anterior, transformarea conţinutului unui mesaj ACL în setul de obiecte ontologice corespunzătoare este precedată de o etapă de validare a acestuia, bazată pe schemele definite în ontologie. Se asigură astfel consistenţa informaţiilor recepţionate de un agent, raportate la ontologia specificată. Această etapă este necesară în cazul sistemelor deschise, în care agenţii realizaţi de diferiţi dezvoltatori, având structuri diferite, interacţionează. Pentru sistemele închise, în care toţi agenţii sunt de tip JADE iar reprezentarea internă a informaţiilor este unitară, validarea informatiilor nu mai este imperativă şi poate constitui doar o întârziere în prelucrarea datelor.

Din aceste considerente, JADE permite anularea efectuării acestor verificari prin metoda *setValidationMode(...)* a clasei `ContentManager`.

7.8.7 Limbaje descriptive definite de programator

În JADE, un codec pentru un limbaj descriptiv este o instanţă a unei clase ce extinde clasa abstracta **jade.content.lang.Codec**, implementând metodele *decode()* şi *encode()* pentru:

- a) a trimite conţinutul unui mesaj ACL în vederea obţinerii obiectului AbsContentElement;

- b) completarea conţinutului unui mesaj ACL cu o instanţă AbsContentElement serializată corespunzător sintaxei limbajului şi codificării utilizate.

Într-un mesaj ACL, o expresie descriptivă poate fi reprezentată ca un String sau şir de byte (byte[]). Clasa Codec este specializată prin clasele `StringCodec` şi `ByteArrayCodec` care au prototipurile metodelor *encode()* şi *decode()* corespunzătoare parametrilor String sau byte[]. Pentru exemplificare, codec-ul SL extinde `StringCodec`, iar codec-ul LEAP extinde **ByteArrayCodec.**

Pentru a vizualiza conţinutul mesajelor dintre agenţi se procedează astfel :

1. din mediul de programare *Eclipse,* meniul *Ejade* selectăm *Start Ejade RMA* pentru a lansa în execuţie platforma SMA,

2. se înscriu în platformă agenţi astfel: *click dreapta* pe numele agentului şi apoi se alege opţiunea *Ejade ->Deploy Agents,*

3. agenţi activi vor fi plasaţi într-un container, apoi se porneşte din bara de meniuri *Snifferul* dând click pe numele agentului,

4. în fereastra de dialog dăm click pe vârful săgeţii care marchează tipul dialogului (Inform, Request, ..)

7.8.8 Introspectori

Pentru decuplarea ontologiei de clasele ontologice, clasa Ontology nu se ocupă direct cu transformarea descriptorilor abstracţi în obiecte Java. Aceasta deleagă atribuţiile unui obiect ce implementează interfaţa **Introspector** aflată în pachetul **jade.content.onto**, ce conţine metodele:

- a) *externalize(...)*: transformă un obiect Java într-un descriptor abstract corespunzător,

- b) *internalize(...)*: transformă un descriptor abstract într-un obiect Java;

- c) *checkClass(...)*: verifică dacă o clasă (ontologică) asociată unei scheme are structura corespunzatoare acesteia şi în consecinţă metodele externalize() şi internalize() vor putea fi executate asupra instanţelor acesteia.

Observaţie 14 *Când se defineşte o nouă ontologie se poate specifica în constructorul acesteia obiectul Introspector ce se doreşte a fi folosit, standard se foloseşte drept introspector o instanţă a clasei ReflectiveIntrospector (foloseste Java Reflection).*

7.9 Aplicaţie

1. Construiţi un S.M.A(sistem multiagent) ce simulează angajarea unor persoane prin colaborarea dintre doi agenţi:

- *RequesterAgent* are rolul de a prelua specificaţiile clientului pentru angajaţii doriţi
- *EngagerAgent*, care realizeaza efectiv angajarea, în colaborare cu *Requester Agent.*

Condiţiile de angajare:

a) Cunoaşterea unei platforme multiagent: Jade, Jadex, Tropos, ..

b) Vârsta maximă 35 ani.

> **Observaţie 15** *În scrierea codului pentru* RequesterAgent *ne-am inspirat din lucrarea [6].*

Pentru a înţelege corect cum a fost concepută această aplicaţie vom prezenta diagrama UML aferentă:

A) Conceptele ontologiei le vom definii în clasele Address si Company, prima va conţine date despre persoana care doreste să se angajeze, iar a doua date despre firmă:

```
//1.
import jade.content.Concept;
public class Address implements Concept{
private String _street;//Numele strazii
private Long _number; // Nr. Casei
private String _city;//Oras
public void setStreet(String street) {
 _street=street;}
public String getStreet(){
 return _street;}
public void setNumber(Long number) {
 _number=number;}
 public Long getNumber(){
return _number;}
public void setCity(String city){
_city=city;}
public String getCity() {
 return _city;}
public boolean equals(Address a){
if (!_street.equalsIgnoreCase(a.getStreet()))
 return false;
 if (_number.longValue() !=
a.getNumber().longValue())
 return false;
 if (!_city.equalsIgnoreCase(a.getCity()))
 return false;
 return true;
}}
//2.
import jade.content.Concept;
public class Company implements Concept {
```

```
private String  _name;// Firma
private Address _address;//Adresa
public void setName(String name) {
 name=name;}
 public String getName() {
return _name;}
 public void setAddress(Address address) {
_address=address;}
 public Address getAddress() {
 return _address;}
 public boolean equals(Company c){
return (_name.equalsIgnoreCase
(c.getName()));}}
```

B) Predicatele vor fi descrise in clasa Person si in clasa WorksFor:

```
//1.
import jade.content.Predicate;
 public class Person implements Predicate {
 private String _name;//Numele persoanei
 private Long _age;//Virsta
 private Address _address; //Adresa
// Metode necesare identificare persoana
public void setName(String name) {
 name=name;
public String getName() {
 return name;}
public void setAge(Long age) {
 age=age;}
public Long getAge() {
return age;}
public void setAddress(Address address) {
 address=address;}
public Address getAddress() {
 return address;}
```

```
public boolean equals(Person p){
if (!name.equalsIgnoreCase(p.getName()))
return false;
 if (age != null && p.getAge() != null)
if (age.longValue()!= p.getAge().longValue())
return false;
 if (address != null && p.getAddress()!= null)
if (!address.equals(p.getAddress()))
return false;
 return true;}}
//2.
import jade.content.Predicate;
public class WorksFor implements Predicate {
 private Company _company; //Angajatorul
 private Person\qquad _person; //Persoana angajata
 //Metode utilizate de JADE-framework
 public void setPerson(Person person) {
 _person=person; }
 public Person getPerson() {
 return _person; }
 public void setCompany(Company company) {
 _company=company; }
 public Company getCompany() {
 return _company; }}

**********************************
```

C) Ontologia utilizata

```
import jade.content.onto.*;
import jade.content.schema.*;
import java.util.*;
public class EmploymentOntology extends Ontology {
 // Constante simbolice, continute in ontologie
 public static final String NAME = "employment-ontology";
```

```
 // VOCABULARUL, CONCEPTE
 public static final String ADDRESS ="ADRESA";
 public static final String ADDRESS_NAME ="STRADA";
 public static final String ADDRESS_NUMBER ="NUMAR";
 public static final String ADDRESS_CITY ="ORAS";
 public static final String PERSON ="PERSOANA";
 public static final String PERSON_NAME ="NUME";
 public static final String PERSON_AGE = "VIRSTA";
 public static final String PERSON_ADDRESS ="ADRESA PERSOANEI";
 public static final String COMPANY ="COMPANIA";
 public static final String COMPANY_NAME ="NUME";
 public static final String COMPANY_ADDRESS ="ADRESA";
 // ACTIUNI
public static final String ENGAGE ="ANGAJATOR";
public static final String ENGAGE_PERSON ="PERSOANA";
public static final String ENGAGE_COMPANY ="COMPANIE";
 // PREDICATE
public static final String WORKS_FOR ="Lucreaza pentru";
public static final String WORKS_FOR_PERSON ="PERSOANA";
public static final String WORKS_FOR_COMPANY ="COMPANIE";
public static final String ENGAGEMENT_ERROR ="ENGAGEMENT-ERROR";
public static final String PERSON_TOO_OLD =
"PERSOANA CU VIRSTA > 35 ANI";
private static Ontology theInstance =
new EmploymentOntology();
 public static Ontology getInstance() {
return theInstance; }
 private EmploymentOntology() {
super(NAME, BasicOntology.getInstance());
try {
 add(new ConceptSchema(ADDRESS),Address.class);
 add(new ConceptSchema(PERSON),Person.class);
 add(new ConceptSchema(COMPANY),Company.class);
 add(new PredicateSchema(WORKS_FOR),WorksFor.class);
```

```
 add(new PredicateSchema(PERSON_TOO_OLD),
PersonTooOld.class);
 add(new PredicateSchema(ENGAGEMENT_ERROR),
EngagementError.class);
 add(new AgentActionSchema(ENGAGE),Engage.class);
 ConceptSchema cs =(ConceptSchema)getSchema(ADDRESS);
 cs.add(ADDRESS_NAME,
(PrimitiveSchema)getSchema(BasicOntology.STRING));
 cs.add(ADDRESS_NUMBER,
(PrimitiveSchema)getSchema(BasicOntology.INTEGER),
ObjectSchema.OPTIONAL);
 cs.add(ADDRESS_CITY, (PrimitiveSchema)
getSchema(BasicOntology.STRING),
 ObjectSchema.OPTIONAL);
 cs = (ConceptSchema)getSchema(PERSON);
cs.add(PERSON_NAME, (PrimitiveSchema)
getSchema(BasicOntology.STRING));
 cs.add(PERSON_AGE, (PrimitiveSchema)
getSchema(BasicOntology.INTEGER),
ObjectSchema.OPTIONAL);
 cs.add(PERSON_ADDRESS, (ConceptSchema)
getSchema(ADDRESS), ObjectSchema.OPTIONAL);
 cs = (ConceptSchema)getSchema(COMPANY);
 cs.add(COMPANY_NAME,
(PrimitiveSchema)getSchema(BasicOntology.STRING));
cs.add(COMPANY_ADDRESS,
(ConceptSchema)getSchema(ADDRESS),
ObjectSchema.OPTIONAL);
 PredicateSchema ps =
(PredicateSchema)getSchema(WORKS_FOR);
 ps.add(WORKS_FOR_PERSON,
(ConceptSchema)getSchema(PERSON));
 ps.add(WORKS_FOR_COMPANY,
(ConceptSchema)getSchema(COMPANY));
```

```
 AgentActionSchema as =
(AgentActionSchema)getSchema(ENGAGE);
as.add(ENGAGE_PERSON,
(ConceptSchema)getSchema(PERSON));
as.add(ENGAGE_COMPANY,
(ConceptSchema)getSchema(COMPANY)); }
catch(OntologyException oe) {
oe.printStackTrace();}}}
```

D) Agenţii utizaţi în platformă:

```
import jade.content.Concept;
import jade.core.AID;
*****************************************
//Engage este un agent ce realizeaza efectiv
// angajarea,in colaborare cu un requestAgent
*****************************************
public class Engage implements Concept {
private Company\qquad _company;//Director personal
private Person person;//Persoana care se angajeaza
private AID actor;//Metode utilizate in Framework
public void setPerson(Person person)
person =person;}
public Person getPerson() {
return person;}
public void setCompany(Company company)
{company=company;}
public Company getCompany()
{return company;}}
```

G. Caire, J.Cabanillas în lucrarea [5] sugerează folosirea protocolului FIPA-QUERY. Vom definii în continuare comportamentele agentului **Engager Agent** care realizează efectiv angajarea, în colaborare cu Requester Agent, folosind sursele din **jade.proto.FipaQueryResponderBehaviour.**

```
import jade.proto.SimpleAchieveREResponder;
```

```
import jade.core.*;
import jade.core.behaviours.*;
import jade.domain.*;
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.ACLMessage;
import jade.lang.acl.MessageTemplate;
import jade.content.onto.basic.*;
import jade.util.leap.*;
public class EngagerAgent extends Agent {
// Comportamentele Agentului
class HandleEnganementQueriesBehaviour
extends SimpleAchieveREResponder
public HandleEnganementQueriesBehaviour(Agent myAgent){
super(myAgent, MessageTemplate.and
(MessageTemplate.MatchProtocol
(FIPANames.InteractionProtocol.FIPA_QUERY),
MessageTemplate.MatchOntology(EmploymentOntology.NAME)));}
*************************************************
// Aceasta metoda este apelata cand un mesaj
//de tipul QUERY-IF sau QUERY-REF este receptionat.
*************************************************
public ACLMessage prepareResponse(ACLMessage msg){
ACLMessage reply = msg.createReply();
*************************************************
// Daca QUERY message este QUERY-REF
// atunci se va raspunde cu performativa NOT_UNDERSTOOD.
*************************************************
if (msg.getPerformative() != ACLMessage.QUERY_IF){
reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
```

```
String content = "(" + msg.toString() + ")";
reply.setContent(content);
 return(reply); }
try {
Predicate pred =(Predicate)myAgent.getContentManager().
extractContent(msg);
if (!(pred instanceof WorksFor)) {
*********************************************************
// Daca predicatul,impus de WORKS_FOR are valoare de false,
// atunci se raspunde cu NOT_UNDERSTOOD
*********************************************************
reply.setPerformative(ACLMessage.NOT_UNDERSTOOD);
String content = "(" + msg.toString() + ")";
reply.setContent(content);
return(reply);
reply.setPerformative(ACLMessage.INFORM);
WorksFor wf = (WorksFor)pred;
Person p = wf.getPerson();
Company c = wf.getCompany();
if (((EngagerAgent)myAgent).isWorking(p, c))
reply.setContent(msg.getContent());
 else {
*******************************************
// crearea unui obiect pentru care predicatul
// WORKS_FOR returneaza false
*******************************************
Ontology o = getContentManager()
.lookupOntology(EmploymentOntology.NAME);
AbsPredicate not =
new AbsPredicate(SLVocabulary.NOT);
not.set(SLVocabulary.NOT_WHAT, o.fromObject(wf));
myAgent.getContentManager().
fillContent(reply, not);}
catch (Codec.CodecException fe) {
```

```
 System.err.println(myAgent.getLocalName()+
" Fill/extract content unsucceeded. Reason:
" + fe.getMessage());
catch (OntologyException oe){
System.err.println(myAgent.getLocalName()+
" getRoleName() unsucceeded.
 Reason:" + oe.getMessage());
return (reply);
*****************************************************
// Acest comportament se ocupa de actiuni
// care au fost solicitate de protocolul FIPA-Request
*****************************************************
class HandleEngageBehaviour extends SimpleAchieveREResponder {
public HandleEngageBehaviour(Agent myAgent){
super(myAgent, MessageTemplate.MatchProtocol
(FIPANames.InteractionProtocol.FIPA_REQUEST));
 public ACLMessage prepareResultNotification
(ACLMessage request, ACLMessage response) {
*****************************************************
// Pregatirea unui mesaj ACL utilizat pentru crearea
//continutul tuturor mesajelor receptionate
*****************************************************
ACLMessage msg = request.createReply();
try{
Action a = (Action)myAgent.getContentManager().extractContent(request);
Engage e = (Engage) a.getAction();
Person p = e.getPerson();
Company c = e.getCompany();
*****************************************************
// Daca varsta persoanei <35-->se accepta (AGREE),
// altel REFUSE si EXIT.
// Se porneste actiunea de angajare
*****************************************************
int result =((EngagerAgent) myAgent).doEngage(p,c);
```

```
// Raspunsuri corespunzatoare functie de cerinte
 if (result >0){
// OK --> INFORM action done
Done d = new Done();
d.setAction(a);
myAgent.getContentManager().fillContent(msg, d);
msg.setPerformative(ACLMessage.INFORM);}
else{
// NOT OK --> FAILURE
ContentElementList l = new ContentElementList();
l.add(a);
myAgent.getContentManager().fillContent(msg, l);
msg.setPerformative(ACLMessage.FAILURE);
catch (Exception fe){
System.out.println(myAgent.getName() +
": Error handling the engagement action.");
System.out.println(fe.getMessage().toString());}
System.out.println(msg);
 return msg;}
public ACLMessage prepareResponse(ACLMessage request){
************************************************************
// Pregatirea agentului dummy ACLMessage
// Utilizat pentru crearea continutului tuturor mesajelor de
// raspuns
************************************************************
 ACLMessage temp = request.createReply();
try{
Action a = (Action)getContentManager().extractContent(request);
Engage e = (Engage) a.getAction();
Person p = e.getPerson();
Company c = e.getCompany();
if (p.getAge().intValue() < 35){
ContentElementList l = new ContentElementList();
l.add(a);
```

```
l.add(new TrueProposition());
getContentManager().fillContent(temp, l);
temp.setPerformative(ACLMessage.AGREE);
else {
ContentElementList l = new ContentElementList();
l.add(a);
getContentManager().fillContent(temp, l);
temp.setPerformative(ACLMessage.REFUSE);
catch (Exception fe){
 fe.printStackTrace();
System.out.println(getName() + ":
 Error handling the engagement action.");
System.out.println(fe.getMessage().toString());
return temp;}}
**********************************************
// Variabile locale
**********************************************
private Company represented Company;
private List employees;//Lista angajatilor companiei
**********************************************
// CONSTRUCTORUL AGENTULUI
**********************************************
public EngagerAgent(){
super();
representedCompany = new Company();
representedCompany.setName("Trident");
Address a = new Address();
a.setStreet("\"Hipodromului\"");
a.setNumber(new Long(274));
a.setCity("Sibiu");
representedCompany.setAddress(a);
employees = new ArrayList();}
protected void setup() {
System.out.println("Acest angajat reprezinta compania"
```

```
+representedCompany.getName());
// codec-ul pentru limbajul SL0
getContentManager().registerLanguage(new SLCodec(),
FIPANames.ContentLanguage.FIPA_SL0);
// Definirea ontologiei
getContentManager().registerOntology
(EmploymentOntology.getInstance());
addBehaviour
(new HandleEnganementQueriesBehaviour(this));
{MessageTemplate mt = MessageTemplate.and
(MessageTemplate.MatchProtocol
(FIPANames.InteractionProtocol.FIPA_REQUEST),
MessageTemplate.MatchOntology
(EmploymentOntology.NAME));
HandleEnganementQueriesBehaviour b =
new HandleEnganementQueriesBehaviour(this);
HandleEngageBehaviour c =
new HandleEngageBehaviour(this);
addBehaviour(b);
addBehaviour(c);
```

E) Metodele AGENTULUI

```
boolean isWorking(Person p, Company c){
boolean isAnEmployee = false;
Iterator i = employees.iterator();
while (i.hasNext()){
Person empl = (Person) i.next();
if (p.equals(empl))
isAnEmployee = true;}}
if (c.equals(representedCompany)){
return isAnEmployee;
else {
return !isAnEmployee;}
int doEngage(Person p, Company c){
```

```
 if (!c.equals(representedCompany)){
return (-1);
else
employees.add(p);
return (1);
************************************************
//RequesterAgent are rolul de a prelua
//specificatiile clentului pentru angajatii doriti
************************************************
import jade.lang.acl.ACLMessage;
import jade.core.*;
import jade.core.behaviours.*;
import jade.domain.FIPAException;
import jade.domain.FIPANames;
import jade.proto.SimpleAchieveREInitiator;
import jade.content.lang.Codec;
import jade.content.lang.sl.*;
import jade.domain.*;
import jade.content.*;
import jade.content.abs.*;
import jade.content.onto.*;
import jade.content.onto.basic.*;
import examples.ontology.employment.*;
import java.util.Vector;
import jade.util.leap.*;
import java.io.*;
public class RequesterAgent extends Agent {
// Comportamentul Agentului executat ciclic
class HandleEngagementBehaviour extends SequentialBehaviour
// Variabile locale
Behaviour queryBehaviour = null;
Behaviour requestBehaviour = null;
// Constructor
public HandleEngagementBehaviour(Agent myAgent){
```

```
super(myAgent);}
public void onStart(){
*****************************************
// Obtinerea de informatii despre persoana
// care solicita angajarea.
*****************************************
BufferedReader buff = new BufferedReader
(new InputStreamReader(System.in));
Person p = new Person();
Address a = new Address();
System.out.println ("Introduceti detalii despre persoana:");
System.out.print(" Numele persoanei --> ");
p.setName(buff.readLine());
System.out.print(" Varsta ---> ");
p.setAge(new Long(buff.readLine()));
System.out.println(" Adresa --->");
System.out.print(" Strada -----> ");
a.setStreet(buff.readLine());
System.out.print(" Numarul -----> ");
a.setNumber(new Long(buff.readLine()));
System.out.print(" Orasul -----> ");
 a.setCity(buff.readLine());
p.setAddress(a);
WorksFor wf = new WorksFor();
wf.setPerson(p);
wf.setCompany(((RequesterAgent) myAgent).c);
Ontology o = myAgent.getContentManager()
.lookupOntology(EmploymentOntology.NAME);
// Crearea unui mesaj ACL  de tip QUERY_IF
ACLMessage queryMsg =
new ACLMessage(ACLMessage.QUERY_IF);
queryMsg.addReceiver
(((RequesterAgent) myAgent).engager);
queryMsg.setLanguage
```

```
(FIPANames.ContentLanguage.FIPA_SL0);
queryMsg.setOntology(EmploymentOntology.NAME);
try {
myAgent.getContentManager().fillContent(queryMsg,wf);
 catch (Exception e){
e.printStackTrace();}
*****************************************************
// Crearea si adaugarea unui comportament
// pentru engager agent respectand protocolul FIPA-Query
*****************************************************
queryBehaviour =
new CheckAlreadyWorkingBehaviour(myAgent, queryMsg);
addSubBehaviour(queryBehaviour);}
catch (IOException ioe) {
System.err.println("I/O error: " + ioe.getMessage());}
// Metoda care va fi executata la terminarea comportamentului
public int onEnd(){
try{
BufferedReader buff = new BufferedReader
(new InputStreamReader(System.in));
System.out.println("Doriti sa continuati ?[y/n] ");
String stop = buff.readLine();
if (stop.equalsIgnoreCase("y"))
reset(); }
myAgent.addBehaviour(this);}
 else
myAgent.doDelete(); // Exit
catch (IOException ioe) {
System.err.println("I/O error:" + ioe.getMessage());}
 return 0;
public void reset(){
if (queryBehaviour != null){
removeSubBehaviour(queryBehaviour);
queryBehaviour = null;
```

```
if (requestBehaviour != null){
removeSubBehaviour(requestBehaviour);
requestBehaviour = null;
super.reset();
class CheckAlreadyWorkingBehaviour extends SimpleAchieveREInitiator{
 // Constructorul
public CheckAlreadyWorkingBehaviour
(Agent myAgent, ACLMessage queryMsg){
super(myAgent, queryMsg);
queryMsg.setProtocol
(FIPANames.InteractionProtocol.FIPA_QUERY);}
protected void handleInform(ACLMessage msg) {}
try{
AbsPredicate cs =(AbsPredicate)myAgent.getContentManager().
extractAbsContent(msg);
Ontology o = myAgent.getContentManager()
.lookupOntology(EmploymentOntology.NAME);
if (cs.getTypeName(.equals(EmploymentOntology.WORKS_FOR))
{// Informarea utilizatorului
WorksFor wf = (WorksFor)o.toObject((AbsObject)cs);
Person p = (Person) wf.getPerson();
Company c = (Company) wf.getCompany();
System.out.println("Persoana " + p.getName() +
" este angajata " + c.getName());
else {
if (cs.getTypeName().equals(SLVocabulary.NOT)){
WorksFor wf = WorksFor o.toObject(cs.getAbsObject
(SLVocabulary.NOT_WHAT));}
Person p = (Person)wf.getPerson();
Company c = (Company)wf.getCompany();
Engage e = new Engage();
e.setPerson(p);
e.setCompany(c);
Action a = new Action();
```

```
a.setActor(((RequesterAgent) myAgent).engager);
a.setAction(e);
ACLMessage requestMsg = new ACLMessage (ACLMessage.REQUEST);
requestMsg.addReceiver
(((RequesterAgent) myAgent).engager);
requestMsg.setLanguage
(FIPANames.ContentLanguage.FIPA_SL0);
requestMsg.setOntology(EmploymentOntology.NAME);
try{
myAgent.getContentManager().
fillContent(requestMsg,a);}
catch (Exception pe) {
((HandleEngagementBehaviour) parent).requestBehaviour =
new RequestEngagementBehaviour(myAgent, requestMsg);
((SequentialBehaviour) parent).addSubBehaviour
(((HandleEngagementBehaviour)parent).
requestBehaviour);
else{
System.out.println ("Asteptare oferta de servici de la Angajator");}
catch (Codec.CodecException fe){
System.err.println(
"FIPAException in fill/extract Msgcontent:"
 + fe.getMessage());
catch (OntologyException fe) {
System.err.println("OntologyException in getRoleName:
" + fe.getMessage());}
class RequestEngagementBehaviour
extends SimpleAchieveREInitiator{
// Constructor
public RequestEngagementBehaviour(Agent myAgent,
 ACLMessage requestMsg){
super(myAgent, requestMsg);
requestMsg.setProtocol(FIPANames.InteractionProtocol.
FIPA_REQUEST);
```

```
protected void handleAgree(ACLMessage msg) {
System.out.println("Angajare reusita.
Asteptam notificarea d-voastra");
protected void handleInform(ACLMessage msg){
System.out.println("A-ti fost angajat");
protected void handleNotUnderstood
(ACLMessage msg){
System.out.println("Nu ati complectat toate datele");}
protected void handleFailure
(ACLMessage msg){
System.out.println("Angajare esuata");
try{
AbsPredicate absPred =
(AbsPredicate)myAgent.getContentManager().
extractContent(msg);
System.out.println("Motivatia este:
" + absPred.getTypeName());
catch (Codec.CodecException fe){
System.err.println("FIPAException reading failure reason:
" + fe.getMessage());
catch (OntologyException oe){
System.err.println("OntologyException reading failure reason:
" + oe.getMessage());
protected void handleRefuse(ACLMessage msg)
System.out.println("Angajare refuzata");
try{
AbsContentElementList list =
(AbsContentElementList)myAgent.getContentManager().
extractAbsContent(msg);
AbsPredicate absPred =
(AbsPredicate)list.get(1);
System.out.println("Motivatia este: " +
absPred.getTypeName());
catch (Codec.CodecException fe){
```

```
System.err.println("FIPAException reading
refusal reason: "+ fe.getMessage());}
catch (OntologyException oe){
System.err.println("OntologyException reading
refusal reason: " + oe.getMessage());
// Variabilele locale ale agentului
AID engager;
Company c;
protected void setup() {
// Inregistrarea codec-lui pentru limbajul  descriptiv SL0
getContentManager().registerLanguage(new SLCodec(),
FIPANames.ContentLanguage.FIPA_SL0);
// Inregistrarea ontologiei utilizata in aceasta aplicatie
getContentManager().registerOntology(EmploymentOntology.
getInstance());
try {
BufferedReader buff =
new BufferedReader(new InputStreamReader(System.in));
System.out.print("Introduceti numele pentru agentul --> ");
String name = buff.readLine();
engager = new AID(name, AID.ISLOCALNAME);
// Se furnizeaza informatii despre firma angajatoare
c = new Company();
Address a = new Address();
System.out.println("Introduceti detalii despre firma angajatoare");
System.out.print(" Numele Companiei --> ");
c.setName(buff.readLine());
System.out.println(" Adresa companiei");
System.out.print(" Strada ------> ");
a.setStreet(buff.readLine());
System.out.print(" Numar ------> ");
a.setNumber(new Long(buff.readLine()));
System.out.print(" Oras ------> ");
a.setCity(buff.readLine());
```

```
c.setAddress(a);}
catch (IOException ioe) {
System.err.println("I/O error: "+ ioe.getMessage());
addBehaviour(new HandleEngagementBehaviour(this));
}
```

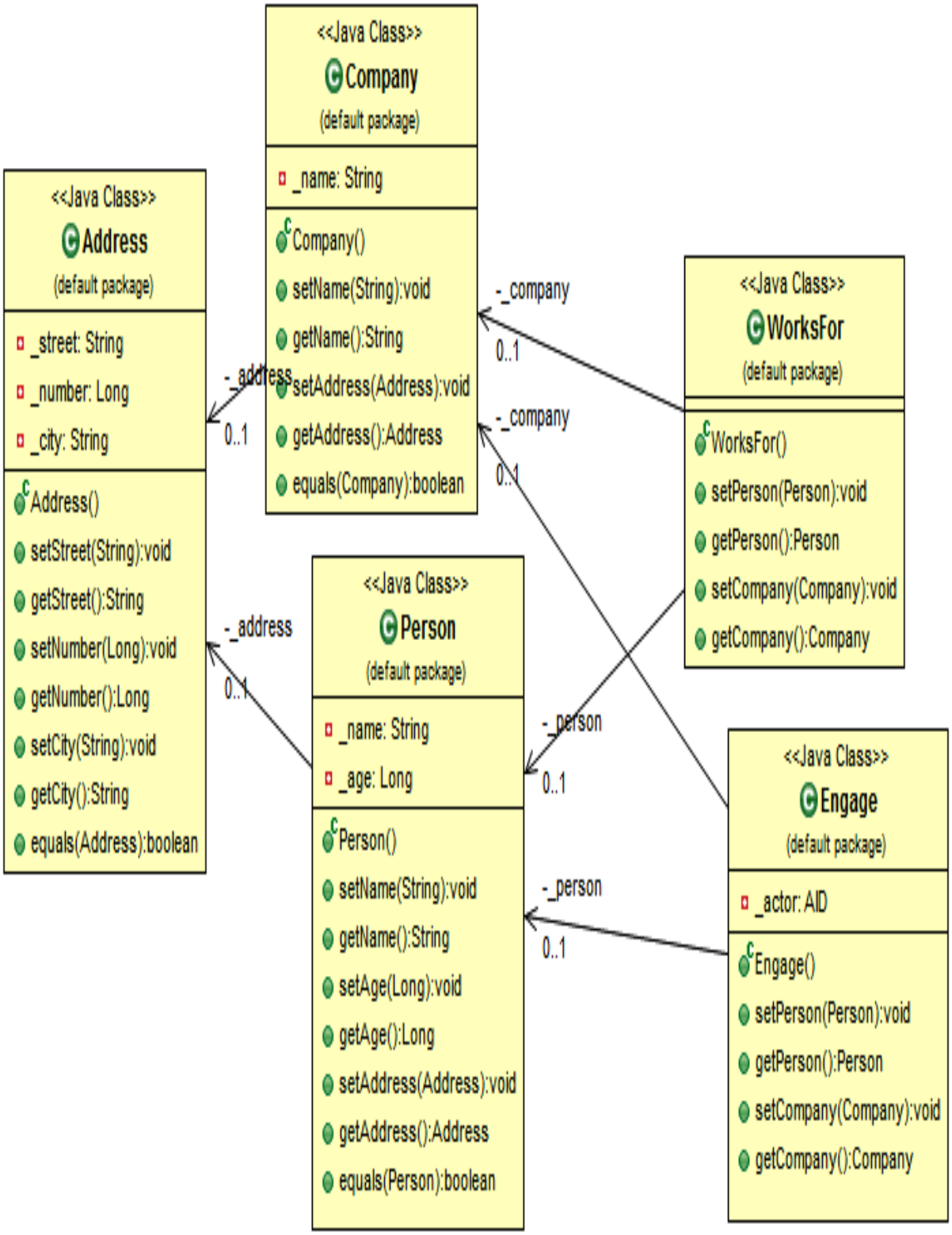


Figura 7.3: Diagrama UML Angajare-online

Capitolul 8

Serviciul Yellow-Pages

8.1 Directory Facilitator

Fiecare `maincontainer` conţine un agent **DF (Directory Facilitator)** ce pune la dispoziţie un serviciu de **Pagini Aurii**, prin care se pot găsi agenţi, care implementează un anumit tip de servicii. Modul de lucru este prezentat în figura următoare [7] :

- DF este an agent care comunică cu ceilati agenţti folosind limbajul ACL,
- Ontologiile si protocoalele sunt standarde FIPA,
- Este posibila înregistrarea şi căutarea de servicii şi pe alte platforme.

Biblioteca **jade.domain.DFService** conţine clase cu metode care uşurează interacţiunea cu serviciile oferite de DF:

– *register()*

– *modify()*

– *deregister()*

– *search()*

Când un agent se înregistrează cu ajutorul agentului DF el trebuie să furnizeze o descriere a serviciului pe care-l pune la dispoziţie

(**jade.domain.FIPAAgentManagement.DFAgentDescription**).

Descrierea serviciului trebuie să conţină:

– AID-ul agentului

– o descriere detaliată a seviciului (**class ServiceDescription**)

Descrierea serviciului este compusă din:

• tip (exemplu “Buletin de stiri”)

• numele serviciului (exemplu. “Meteo”)

• Limbajul descriptiv folosit, ontologiile şi protocoalele de interacţiune utilizate (este indicat să se lucreze cu FIPA Contract-net protocol).

8.2 Aplicaţie

Enunţ

Să se creeze un sistem de piaţă virtuală alcătuit dintr-un număr de agenţi cumpăratori şi un număr de agenţi vânzatori, cu următoarele caracteristici:

- vânzătorii, înregistraţi în „Pagini Aurii”, au o listă de produse cu anumite preţuri;

- cumpăratorii caută vânzătorii în „Pagini Aurii” şi cer informaţii despre preţul unui produs;

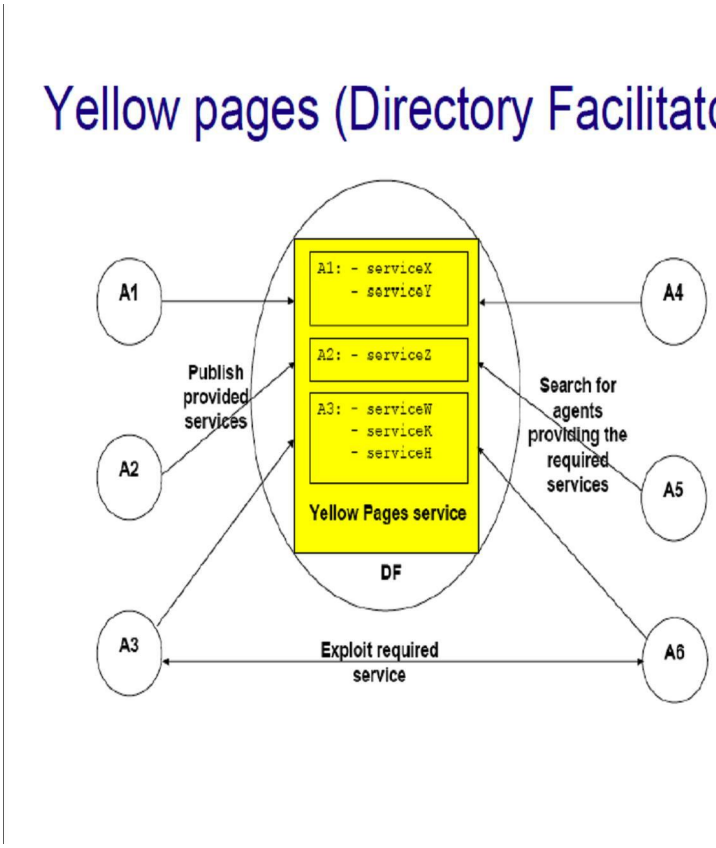


Figura 8.1: Directory Facilator

- cumpăratorii au o listă de produse din care vor alege aleatoriu un produs pe care doresc să-l achiziţioneze şi vor căuta preţul minim pentru acest produs în piaţa virtuală.

Pentru fiecare cumpărător, să se afişeze oferta de preţuri găsită şi preţul minim.

Pentru fiecare vânzător, să se afişeze oferta lui şi interogările la care răspunde.

Implementare

Există următoarele clase:

-**clasa DFBuyerAgent** care mosteneste clasa Agent şi conţine:

-Membrii: myGUI, nrCereri

-Metode: setup(), care caută Produs(final ProductRequest request)

-Inner Classes: RequestSolver extends SimpleBehaviour,

-class DFBuyerAgentGUI extends JFrame care mosteneste clasa JFrame

-Membrii: cumparariAgent, txtNumeProdus

-Metode: jbInit(), show()

-**clasa DFSellerAgent** care mosteneste clasa Agent şi conţine:

-Membrii: produse, vanzatorGUI, numarProduse

-Metode: setup(), înregistrareServiciu(), adaugaProdus(final Product produs)

-**clasa DFSellerAgentGUI** care mosteneste clasa JFrame

-Membrii: vanzariAgent, nameField, priceField

-Metode: guiInit(), show()

-**clasa DFSubscribeAgent** mosteneste clasa Agent

- Metode: setup()

-**clasa Service**

-Membrii: static String BUY_SELL_SERVICE

-**clasa** Product

-Membrii: nume, pret

-**clasa** ProductRequest

-Membrii: numeProdus, pret, agentAID

Aplicaţia constă în urmatoarele acţiuni principale:

-se doreşte vânzarea/cumpârarea de diferite produse

-Agenţii vânzatori işi înregistreaza serviciul. Aceştia au o listă de produse disponibile pe care doresc sâ le vândă cumpărătorilor.

Aşteaptă cereri din partea cumpărătorilor, iar dacă ambele parţi sunt de accord, vânzătorul îi vinde produsul dorit cumpărătorului.

-Agenţii cumpărători verifică ce servicii au fost inregistrate în Yellow Pages şi trimite tuturor o cerere ce ofertează: în care specifica numele produsului cautat.

În funcţie de ofertele primite, cumpărătorul selectează doar pe aceea care are preţul minim şi îi trimite acelui vânzător un mesaj de acceptare a ofertei.

Există următoarele tipuri de agenţi:

1.**DFSellerAgent(vânzatorul)**

-Acesta îşi înregistreaza serviciul în Yellow Pages. De asemenea, are o lista de produse disponibile, la care le poate modifica preţul şi la care poate adauga altele noi.

-Apoi aşteaptă să primească cereri de oferte
de la agenţii cumpărători(DFBuyerAgent).

-Odată primită cererea de ofertă, DFSellerAgenţii trimite cumpărătorului propunerea de preţ pentru respectivul produs.

-Dacă cumpărătorul hotărăşte că acela e cel mai bun preţ dintre cele existente, agentul DFSellerAgent curent va primi o acceptare de propunere, iar ceilalţi agenţi vânzatori vor primi un refuz de propunere.

Dupa confirmarea şi a cumpărătorului, va avea loc vânzarea produsului.

2.**DFBuyerAgent (cumpărătorul)**

-Acesta doreste sa cumpere un produs, la cel mai mic pret dintre cele existente.

-Verifică lista de agenţi care şi-au înregistrat serviciile în Yellow Pages, după care le trimite tuturor o cerere de ofertă.

-Primeşte înapoi preţurile produsului dorit de la agenţii vânzători.

-Determină ce agent are produsul cu preţul cel mai mic şi îi trimite acestuia o acceptare de propunere, iar celorlalţi agenţi le trimite un mesaj prin care le refuză oferta.

-Apoi confirmă tranzacţia agentului vânzător, iar produsul este vândut.

3.**DFSubscribeAgent**

Acesta primeşte notificări cu privire la serviciile înregistrate de agenţii DFSellerAgent.

Protocolul utilizat este FIPA-Contract-Net. Diagrama UML este prezentată în figura următoare:

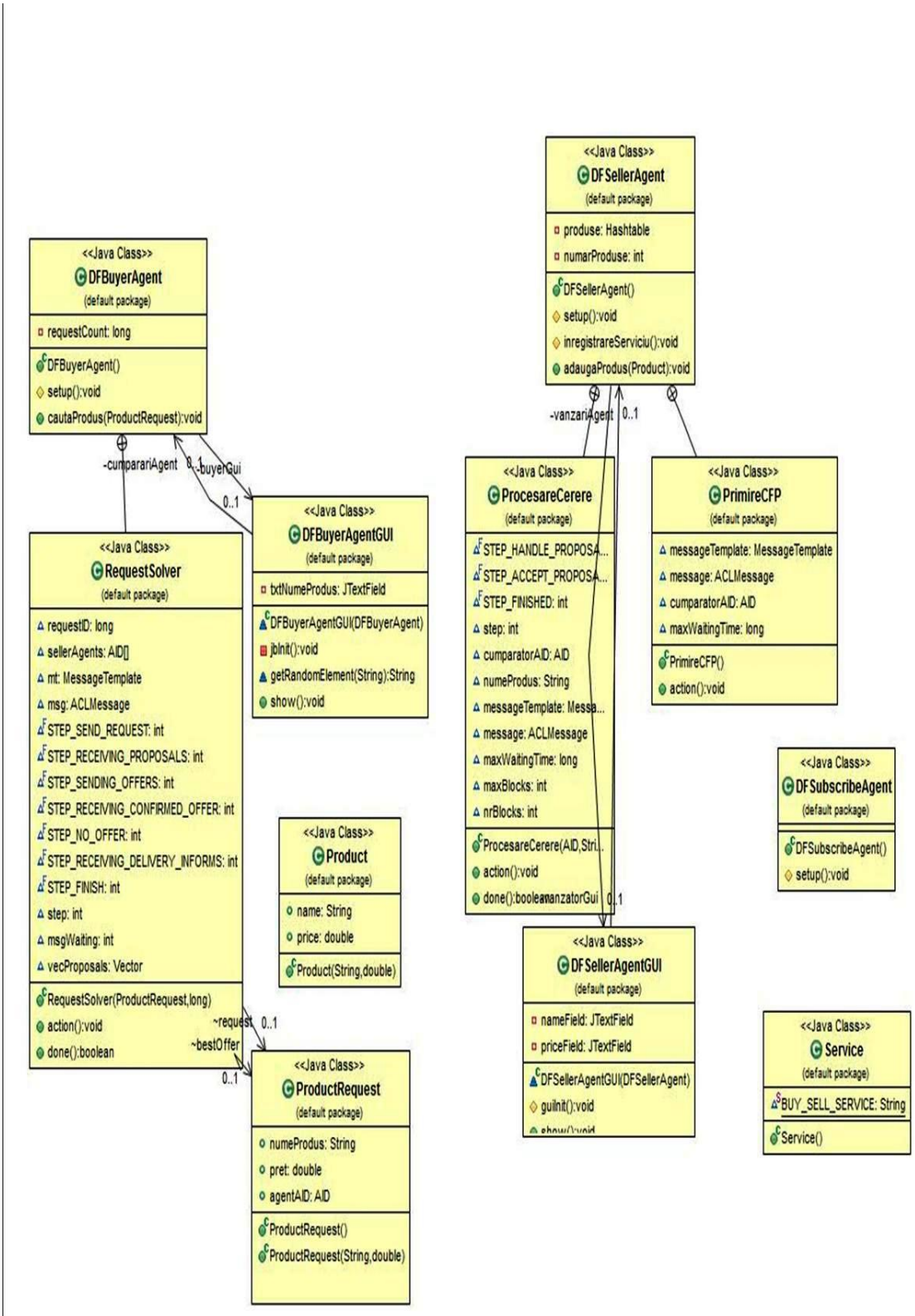


Figura 8.2: Diagrama UML-Yellow-Page

Codul sursă: Clasa DFSellerAgent:

```
 public class DFSellerAgent extends Agent {
 private Hashtable produse;
 private DFSellerAgentGUI vanzatorGui;
 privateintnumarProduse = 0;
 // initializare agent vanzari
 protected void setup() {
 produse = newHashtable();
 vanzatorGui = new DFSellerAgentGUI(this);
 vanzatorGui.show();
 inregistrareServiciu();
 System.out.println("Agentul de vanzare " + getAID().getLocalName()
 + " s-a conectat");
 addBehaviour(new PrimireCFP());}
 protectedvoid inregistrareServiciu(){
 if (numarProduse != 0){
 try {
 DFService.deregister(this);
catch (FIPAException fe) {
 fe.printStackTrace();}}
 // inregistrare serviciu in Yellow Pages
 DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());
 ServiceDescription sd = new ServiceDescription();
 sd.setType("product-selling");
 sd.setName("product");
 sd.addOntologies(Service.BUY_SELL_SERVICE);
 dfd.addServices(sd);
try {
 DFService.register(this, dfd);
 System.out.println("Inregistrare serviciu de catre "
 + getAID().getLocalName());}
 catch (FIPAException fe) {
fe.printStackTrace();}}
```

```
*****************************************
//functie prin care sunt adaugate produsele
// cu ajutorul interfetei GUI
*****************************************
public void adaugaProdus(final Product produs) {
 addBehaviour(newOneShotBehaviour() {
 public void action(){
 // cautam produsul in lista pentru a vedea daca exista deja
 Product produsExistent =
(Product) produse.get(produs.name);
 if (produsExistent != null)
 System.out.println(getLocalName() + ": Produsul "
 + produs.name + " si-a schimbat pretul");
 else {
 System.out.println(getLocalName()
 + ": Produs nou cu numele: " + produs.name);
 numarProduse++;}
 System.out.println("Pret: " + produs.price);
 produse.put(produs.name, produs);}
*************************************************************
// comportament ciclic ce raspunde tuturor cererilor CFP de la
//cumparatori si trimite informatii despre produsele existente
*************************************************************
 public class PrimireCFPextends CyclicBehaviour {
 MessageTemplate messageTemplate;
 ACLMessage message;
 AID cumparatorAID;
 longmaxWaitingTime = 60000;
 public PrimireCFP() {
 System.out.println(getAID().getLocalName()
 + ": Sunt gata sa primesc cereri!");}
 @Override
 public void action() {
 // TODO Auto-generated method stub
```

```
 messageTemplate =
MessageTemplate.MatchPerformative(ACLMessage.CFP);
 message = myAgent.receive(messageTemplate);
if (message != null) {
// a primit un mesaj de tip Call For Proposal
// de la cumparator
 cumparatorAID = message.getSender();
 String content = message.getContent();
// continut mesaj: "nume Produs"
 String numeProdus = content;
// vanzatorul ii trimite raspuns cu
// PROPOSE sau REFUSE
 ACLMessage reply = message.createReply();
Product produs = (Product) produse.get(numeProdus);
// cautare produs in lista
 if (produs != null) {
// daca produsul exista in lista trimite PROPOSE
 reply.setPerformative(ACLMessage.PROPOSE);
 reply.setContent(numeProdus + ":" + produs.price);
 reply.setReplyByDate
(new Date(System.currentTimeMillis()
 + maxWaitingTime));
**********************************************************
// adauga un nou comportament care va modela in continuare
// interactiunea intre cumparator si vanzator
**********************************************************
 addBehaviour(new ProcesareCerere
(cumparatorAID, numeProdus));
 else {
// produsul nu e in lista
reply.setPerformative(ACLMessage.REFUSE);
reply.setContent(numeProdus + " nu e in stoc!");}
myAgent.send(reply);}
 else {
```

```
 block();}}}
*********************************************************
// Comportament pentru cererile vizand articolele
// existente in lista
*********************************************************
 private class ProcesareCerere extends SimpleBehaviour{
 finalintSTEP_HANDLE_PROPOSAL = 1;
 finalintSTEP_ACCEPT_PROPOSAL = 2;
finalintSTEP_FINISHED = 3;
 intstep = STEP_HANDLE_PROPOSAL;
AID cumparatorAID = null;
 String numeProdus = null;
 MessageTemplate messageTemplate;
 ACLMessage message;
 longmaxWaitingTime = 60000;
 intmaxBlocks = 2;
 intnrBlocks = 0;
 public ProcesareCerere
(AID clientAID, String numeProdus)
{
 this.cumparatorAID = clientAID;
 this.numeProdus = numeProdus;
 messageTemplate = MessageTemplate.MatchSender(clientAID);
}
 @Override
public void action() {
 // TODO Auto-generated method stub
 switch (step) {
 case STEP_HANDLE_PROPOSAL:
 System.out.println(getAID().getLocalName()
 + ": Am inceput negocierile cu: "
 + cumparatorAID.getLocalName());
*********************************************************
 // acceptare doar a mesajelor provenind de la client
```

```
 // avand tipurile ACCEPT_PROPOSAL/REJECT_PROPOSAL
*********************************************************
 MessageTemplate mt = MessageTemplate.and(messageTemplate,
 MessageTemplate.or(
 MessageTemplate .MatchPerformative(ACLMessage.ACCEPT_PROPOSAL),
 MessageTemplate .MatchPerformative(ACLMessage.REJECT_PROPOSAL)));
 message = myAgent.receive(mt);
 if (message == null) {
 block(maxWaitingTime);
 System.out.println(getAID().getLocalName()
+ ": Astept dupa receptie ACCEPT/REJECT_PROPOSAL
de la "+ cumparatorAID.getLocalName() + " pentru "
 + maxBlocks + " perioade...");
 nrBlocks++;
 if (nrBlocks>maxBlocks) {
 step = STEP_FINISHED;
 System.out.println(getAID().getLocalName()
 + ": Am abandonat tranzactia cu "
 + cumparatorAID.getLocalName());
 break;}
 else {
 if (message.getPerformative() = ACLMessage.ACCEPT_PROPOSAL) {
 String content = message.getContent();
*****************************
// Continut: "nume Produs"
*****************************
 numeProdus = content;
 Product produs =
(Product) produse.get(numeProdus);
 ACLMessage msgReply = message.createReply();
if (produs != null) {
*********************************************
// daca produsul exista, ii trimite inapoi
// cumparatorului un mesaj de informare (INFORM)
```

```
**********************************************
 msgReply.setPerformative(ACLMessage.INFORM);
 msgReply.setContent(produs.name + ":"+ produs.price);
 step = STEP_ACCEPT_PROPOSAL;
 System.out.println(getAID().getLocalName()+ ": Am retinut produsul "
+ produs.name + "pentru " + cumparatorAID.getLocalName());}
else {
**************************************************
// daca produsul nu exista in lista,
// ii trimite inapoi cumparatorului mesajul FAILURE
**************************************************
 msgReply.setPerformative(ACLMessage.FAILURE);
 msgReply.setContent("Produsul " + produs.name
 + " nu este in stoc.");
step = STEP_FINISHED;}
 myAgent.send(msgReply);
 block(maxWaitingTime);
 nrBlocks = 0;
// reseteaza numarul de asteptari
 else {
// inseamna ca a primit REFUSE_PROPOSAL
System.out.println("Negociere esuata");
step = STEP_FINISHED;}}
break;
case STEP_ACCEPT_PROPOSAL:
**********************************************************
// agentul asteapta mesaj de CONFIRM sau DISCONFIRM pentru
// produsul retinut daca va reception DISCONFIRM,
// va raspunde CANCEL
**********************************************************
 mt = MessageTemplate.and(messageTemplate,
MessageTemplate.or(
 MessageTemplate.MatchPerformative
(ACLMessage.CONFIRM),
```

```
 MessageTemplate.MatchPerformative
(ACLMessage.DISCONFIRM)));
message = myAgent.receive(mt);
if (message == null) {
block(maxWaitingTime);
System.out.println(getAID().getLocalName()
+ ": Astept dupa CONFIRM/DISCONFIRM din
partea cumparatorului "
+ cumparatorAID.getLocalName() + " pentru "
+ maxBlocks + " perioade...");
nrBlocks++;
 // limitare a numarului de asteptari
if (nrBlocks>maxBlocks) {
step = STEP_FINISHED;
System.out.println(getAID().getLocalName()
+ ": Am abandonat tranzactia " + cumparatorAID.getLocalName());
 break;
 return;}
 if (message.getPerformative() ==
ACLMessage.CONFIRM) {
**************************************************
// daca receptioneaza CONFIRM,va raspunde prin INFORM
// adica va livra produsul
**************************************************
Product produs = (Product)produse.get(numeProdus);
System.out.println(getAID().getLocalName() +
": Am livrat "+ numeProdus + " cumparatorului "
+ cumparatorAID.getLocalName() + "("
+ new Date(System.currentTimeMillis()).
toString()+ ").");
ACLMessage msgReply = message.createReply();
msgReply.setPerformative(ACLMessage.INFORM);
msgReply.setContent(numeProdus + " livrat!");
send(msgReply);
```

```
 else {
****************************************
// mesajul e DISCONFIRM trimite inapoi
// cumparatorului CANCEL
****************************************
System.out.println(
"Tranzactie anulata prin DISCONFIRM din partea
cumparatorului!");
 ACLMessage msgReply = message.createReply();
 msgReply.setPerformative(ACLMessage.CANCEL);
 msgReply.setContent("Tranzactie anulata!");
 send(msgReply);}
 step = STEP_FINISHED;
 break;}}
 public boolean done() {
return step == STEP_FINISHED;}}}
//Clasa DFBuyerAgent:
public classDFBuyerAgent extends Agent {
// GUI prin care cumparatorul poate realiza cereri
private DFBuyerAgentGUI buyerGui;
//nr. cereri
 private longrequestCount = 0;
 protected void setup() {
System.out.println
("Agentul de cumparare " + getLocalName() + " s-a conectat");
System.out.println
(getLocalName() + ": Sunt gata sa cumpar produse!");
// Creaza si afiseaza GUI
buyerGui = new DFBuyerAgentGUI(this);
buyerGui.show();}
public void cautaProdus(final ProductRequest request){
addBehaviour(newOneShotBehaviour(){
public void action(){
requestCount++;
```

```
System.out.println("Cerere nr. " + requestCount + ": "
+ request.numeProdus);
addBehaviour(new RequestSolver(request,requestCount));
// proceseaza cererea
});}
class RequestSolver extends SimpleBehaviour {
ProductRequest request = new ProductRequest();
longrequestID = 0;
AID[] sellerAgents = null;
MessageTemplate mt;
ACLMessagemsg = null;
finalintSTEP_SEND_REQUEST = 0;
finalintSTEP_RECEIVING_PROPOSALS = 1;
finalintSTEP_SENDING_OFFERS = 2;
finalintSTEP_RECEIVING_CONFIRMED_OFFER = 3;
finalintSTEP_NO_OFFER = 4;
finalintSTEP_RECEIVING_DELIVERY_INFORMS = 5;
finalintSTEP_FINISH = 6;
intstep = STEP_SEND_REQUEST;
intmsgWaiting = 0;
public RequestSolver(ProductRequest request,
longrequestID) {
this.request = request;
this.requestID = requestID;}
VectorvecProposals = newVector();
// aici salvam toate ofertele
ProductRequest bestOffer =
new ProductRequest(request.numeProdus, -1);
public void action() {
switch (step) {
 case
STEP_SEND_REQUEST:
**************************************************
//1.identifica toti agentii care furnizeaza serviciul
```

```
//2.trimite cererea catre toti agentii din lista
**************************************************
//1.
DFAgentDescription template =
new DFAgentDescription();
 ServiceDescription sd =
new ServiceDescription();
 sd.setType(Service.BUY_SELL_SERVICE);
template.addServices(sd);
try{
DFAgentDescription[] result =
DFService.search(myAgent,template);
System.out.println("Am gasit urmatorii vanzatori:");
sellerAgents =
new AID[result.length];
for (inti = 0; i<result.length; ++i) {
sellerAgents[i] = result[i].getName();
System.out.println(sellerAgents[i].getName());}
 catch (FIPAException fe) {
 fe.printStackTrace();}
// 2.
msg = newACLMessage(ACLMessage.CFP);}
msg.setContent(request.numeProdus);
//trimite numele articolului
msg.setConversationId(String.valueOf
(requestID));
msg.setSender(myAgent.getAID());
for (inti = 0; i<sellerAgents.length; i++){
msg.addReceiver(sellerAgents[i]);}
send(msg);
// trimite cererea catre toti vanzatorii
msgWaiting = sellerAgents.length;
step = STEP_RECEIVING_PROPOSALS;
//asteapta mesaje de la to\U{163}i vanzatorii
```

```
break;
case STEP_RECEIVING_PROPOSALS:
 // 1.asteapta mesaje de tip PROPOSE sau REFUSE
 mt = MessageTemplate.or(
 MessageTemplate.MatchPerformative(ACLMessage.PROPOSE),
 MessageTemplate.MatchPerformative(ACLMessage.REFUSE));
 msg = receive(mt);
 if (msg == null) {
 block();
 return;}
****************************************
//2.Daca este PROPOSE,verifica continutul
//(nume produs: pret)
****************************************
if (msg.getPerformative() == ACLMessage.PROPOSE){
String content = msg.getContent();
String numeProdus = content.substring(0,
content.indexOf(":"));
doublepretOferit =
Double.parseDouble(content.substring
(content.indexOf(":") + 1));
ProductRequest productRequest = new ProductRequest(
numeProdus,pretOferit);
productRequest.agentAID = msg.getSender();
System.out.println(myAgent.getLocalName() + ": "
 + msg.getSender().getLocalName()
 + " a trimis o oferta de " + pret Oferit
 + " pentru produsul " + numeProdus);
 vecProposals.add(productRequest);
****************************************************
 // daca oferta curenta e cea mai buna
//(cel mai mic pret),atunci devine cea mai buna oferta
****************************************************
 if (bestOffer.pret == -1
```

```
|| bestOffer.pret>productRequest.pret)
 bestOffer = productRequest?}
 msgWaiting--;
 if (msgWaiting == 0){
************************************
// au fost receptionate toate mesajele,
// deci se pot procesa ofertele
************************************
if (bestOffer.pret != -1) {
 step = STEP_SENDING_OFFERS;
// procesam ofertele
 else {
 step = STEP_NO_OFFER;
// nici un vanzator nu are respectivul produs
 break;
 case STEP_NO_OFFER:
System.out.println(myAgent.getAID().getLocalName()+
 " : " + request.numeProdus + " nu este in oferta vreunui vanzator!");
step = STEP_FINISH;
 break;
case STEP_SENDING_OFFERS:
 doubletotalQuantity = 0;
 doubletotalPrice = 0;
//1.trimit accept_proposal la vanzatorul care are cel mai mic pret
 msg = newACLMessage(ACLMessage.ACCEPT_PROPOSAL);
 msg.setSender(myAgent.getAID());
 msg.setContent(request.numeProdus);
 msg.addReceiver(bestOffer.agentAID);
 send(msg);
 // 2. trimit refuse_proposal la ceilalti
 msg = newACLMessage(ACLMessage.REJECT_PROPOSAL);
 msg.setSender(myAgent.getAID());
 msg.setContent(request.numeProdus);
 for (inti = 0; i<vecProposals.size(); i++) {
```

```
 ProductRequest pr = (ProductRequest) vecProposals.get(i);
if (pr.agentAID != bestOffer.agentAID)
msg.addReceiver(pr.agentAID);}
 send(msg);
 step = STEP_RECEIVING_CONFIRMED_OFFER;
 break;
case STEP_RECEIVING_CONFIRMED_OFFER:
*****************************************
// asteapta mesaj de tip inform,
// pentru a confirma finalizarea tranzactiei
*****************************************
 mt = MessageTemplate.
MatchPerformative(ACLMessage.INFORM);
 msg = receive(mt);
if (msg == null) {
 block();
 return;}
 String continutMesaj = msg.getContent();
 ACLMessagemsgRaspuns = msg.createReply();
 // trimite confirm pentru finalizarea tranzactiei
msgRaspuns.setPerformative(ACLMessage.CONFIRM);
 send(msgRaspuns);
step = STEP_RECEIVING_DELIVERY_INFORMS;
 break;
 case STEP_RECEIVING_DELIVERY_INFORMS:
*************************************
// asteapta mesaj de tip inform,
// pentru a confirma primirea comenzii
*************************************
 mt = MessageTemplate.MatchPerformative(ACLMessage.INFORM);
 msg = receive(mt);
if (msg == null) {
 block();
 return;}
```

```
 AID vanzatorAID = msg.getSender();
 System.out.println(myAgent.getLocalName() +
": Am obtinut "\qquad + request.numeProdus + "la un pret de
" + bestOffer.pret + " de la "
+ vanzatorAID.getLocalName());
 step = STEP_FINISH;
 break;}}
public boolean done() {
booleanfinished = step = = STEP_FINISH;
if (finished) {
 System.out.println(getLocalName() +
":Tranzactie finalizata!");}
 returnfinished;}}}
//Clasa DFSubscribeAgent:
public class DFSubscribeAgent extends Agent{
protectedvoid setup(){
 DFAgentDescription template =
new DFAgentDescription();
 ServiceDescription templateSd =
new ServiceDescription();
 templateSd.setType("product-selling");
templateSd.addOntologies(Service.BUY_SELL_SERVICE);
 template.addServices(templateSd);
SearchConstraints searchConstrains = new SearchConstraints();
// primeste cel mult 15 servicii
 searchConstrains.setMaxResults(new Long(15));
 addBehaviour(newSubscriptionInitiator
(this, DFService.createSubscriptionMessage
(this, getDefaultDF(), template, searchConstrains))
protectedvoid handleInform(ACLMessage inform) {
 System.out.println("Agent "+ getLocalName() +
": Notificare primita de la DF");
 try {
DFAgentDescription[] results =
```

```
DFService.decodeNotification(inform.getContent());
 if (results.length> 0) {
 for (inti = 0; i<results.length; ++i){
 DFAgentDescription dfd = results[i];
 AID provider = dfd.getName();
 Iterator it = dfd.getAllServices();
while (it.has Next()) {
ServiceDescription sd =
(ServiceDescription) it.next();
if (sd.getType().equals(Service.BUY_SELL_SERVICE)){
System.out.println("Serviciul product-selling gasit :");
System.out.println("- Serviciul
\""+sd.getName()+"\" oferit de catre " + provider.getName());
System.out.println();
catch (FIPAException fe) {
 fe.printStackTrace();}
```

Prezentăm in continuare rezultatele **execuţiei programului**:

1. Crearea agenţilor de tip DFSellerAgent(agenţi responsabili cu vânzarea produselor):

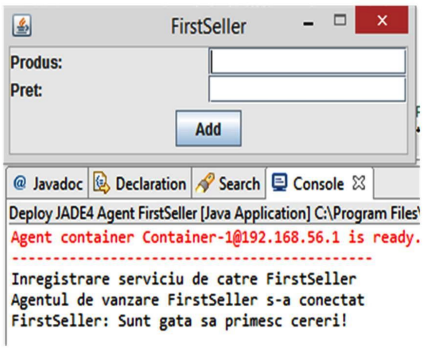


Figura 8.3: Crearea agenţilor DFSeller

2. Adăugarea ofertelor: La consolă apare mesajul: Înregistrare serviciu de către First-

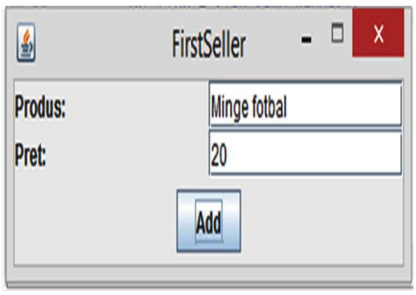


Figura 8.4: Adaugarea ofertelor

Seller
Agentul FirstSeller s-a conectat
FirstSeller: Sunt gata să primesc cereri!
FirstSeller: Produs nou cu numele: Minge fotbal
preţ: 20.0

3. Crearea agenţilor de tip DFBuyerAgent(agenţi responsabili cu cumpărarea produselor):

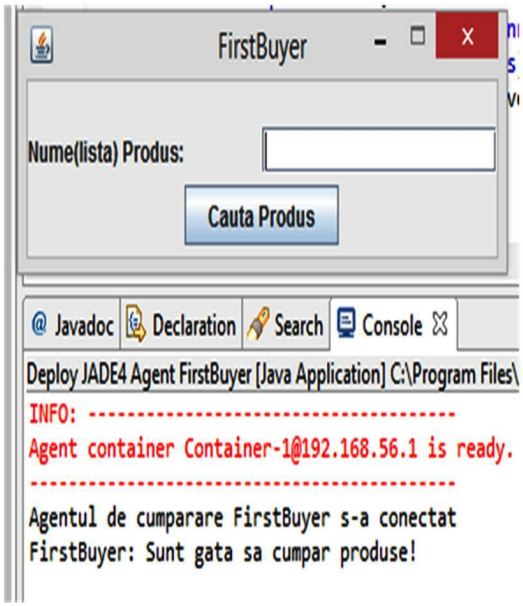


Figura 8.5: Crearea agenţilor DFBuyerAgent

4. Căutarea unui produs şi cumpărarea celui mai ieftin din ofertele disponibile La con-

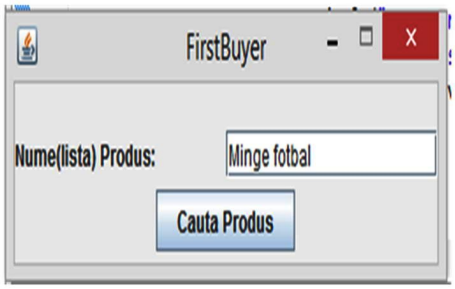


Figura 8.6: Cautare produs

solă apare:Agentul de cumpărare FirstBuyer s-a conectat
FirstBuyer: Sunt gata sa cumpăr produse!
Minge fotbal
Cerere nr.1: Minge fotbal
Am găsit urmatorii vanzatori
FirstSeller@192.168.56.1:1099/JADE
FirstBuyer: FirstSeller a trimis o oferta de 20.0 pentru produsul Minge fotbal
FirstBuyer: Am obţinut Minge fotbal la pret de 20.0 de la FirstSeller
FirstBuyer: Tranzacţie finalizată!
****
Inregistrare serviciu de catre FirstSeller
Agentul de vânzare FirstSeller s-a conectat
FirstSeller: Sunt gata să primesc cereri!
FirstSeller: Produs nou cu numele: Minge fotbal

Pret:20
FirstSeller: Am început negocierile cu FirstBuyer
FirstSeller: Aştept după recepţie ACCEPT-REJECT-PROPOSAL de la FirstBuyer pentru 2 perioade ...
FirstSeller: Am început negocierile cu FirstBuyer
FirstSeller: Am reţinut produsul Minge fotbal pentru FirstBuyer
FirstSeller: Am livrat Minge fotbal cumpărătorului FirstBuyer (Sat Jan 24 13:38:45 EET 2015).

8.3 Probleme propuse

1.*Să se creeze un sistem S.M.A de piaţă virtuală alcătuit dintr-un număr de agenţi cumpărători şi un număr de de agenţi vânzători cu următoarele caracteristici:*

a) *vânzătorii înregistraţi în "Pagini Aurii" au o listă de produse cu anumite preţuri*

b) *cumpărătorii caută vânzătorii în "Pagini Aurii" şi cer informaţii despre preţul unui produs. De asemenea ei au o listă de produse din care vor alege aleatoriu un produs pe care doresc să-l achiţioneze şi vor căuta preţul minim pentru acest produs.*

Cerinţe:

i) *pentru fiecare cumpărător să se afişeze oferta de preţuri găsită şi preţul minim.*

ii) *pentru fiecare vânzător, să se afişeze oferta lui şi înrebările la care răspunde*

2. *Folosind serviciul "Pagini Aurii" să se implementeze un sistem S.M.A pentru găsirea unui pachet de vacanţă în acord cu dorinţele utilizatorului[15]. În sistem vor exista mai mulţi agenţi specializaţi în oferirea de servicii: hoteliere, transport, atracţii turistice, evenimente. Utilizatorul are un agent personal căruia îi cere să-i găsească oferze de cazare sau activităţi recreative. Cerinţe:*

a) *Căutare transport şi cazare:*

i) *destinaţie: oraşul,ţara,*

ii) *perioada (datele pot fi flexibile de exemplu 25.08.2015.-30.08.2015 +/- 3 zile),*

iii) *numărul de camere şi de persoane în cameră,*

iv) *categoria hotelului (numărul de stele)*

v) *un interval de costuri.*

b) *Căutare activităţi recreative*

i) *destinaţie: oraşul,ţara,*

ii) *perioada,*

iii) *tipul de activitate: punctul de atracţie turistică, concerte, activităţi turistice, etc.*

Agentul asistent personal va discuta cu ceilalţi agenţi şi va crea un pachet complet de vacanţă care va cuprinde o listă de variante de cazare şi transport şi va oferii posibile activităţi recreative, cu un cost acceptabil de către utilizator.

Capitolul 9

Mobilitatea agenţilor

În contextul sistemelor multi-agent, mobilitatea reprezintă acea capacitate a agenţilor de a migra sau realiza clone pe una sau mai multe staţii conectate. Mobilitatea poate fi restransă la cadrul aceleiaşi platforme ori poate face referinţă la spaţiul mai multor platforme, accesibile.

JADE pune la dispozitie serviciul `Agent-mobility,` activ implicit, pe baza căruia agenţii pot realiza aceste operaţii (intra-platformă), fie prin înaintarea unei cereri către AMS, fie prin apelul unor metode aparţinând clasei părinte **jade.core.Agent**.

Domeniul mobilităţii agenţilor este descris în cadrul ontologiei MobilityOntology, ce conţine vocabularul specific precum şi conceptele şi predicatele corespunzatoare migrarii şi clonării agenţilor.

9.1 Migrarea agenţilor

Un agent poate migra dintr-un container în altul, aparţinând aceleaşi platforme ori a unei platforme diferite. Fiecare container este perceput drept o locaţie, astfel încât pentru migrarea ori clonarea unui agent trebuie întâi specificată viitoarea locaţie a agentului (ori a clonei sale). În JADE, interfaţa Location identifică o astfel de destinaţie. Fiind interfaţă, nu pot fi instanţiate obiecte de tip Location, acestea obţinându-se în două modalităţi:

- a) de la AMS, prin trimiterea unei cereri utilizând ontologia MobilityOntology, ce conţine o acţiune de tip **WhereIsAgentAction** ori **QueryPlatformLocations-Action**;

prin acţiunea **WhereIsAgentAction** se va obţine locaţia în care se află un anumit agent, iar prin **QueryPlatformLocationsAction** se vor obţine toate locaţiile din sistem.

- b) prin utilizarea clasei **jade.core.ContainerID**

ce implementeaza interfaţa **Location** şi are rol de descriptor pentru containere.

Un exemplu al folosirii acestei clase se regaseşte în codul agentului RMA **(jade.tools.rma)**. Din interfaţa grafică a acestuia se poate specifica numele container-ului destinaţie, agentul RMA trimitând apoi o cerere catre AMS pentru efectuarea operaţiei în care foloseşte o instanţa ContainerID pe post de Location.

Metodele clasei jade.core.Agent asociate migrarii agentului sunt:

- a) *beforeMove()*: metoda apelata înaintede migrare, folosită pentru eliberarea resurselor alocate ori pentru alte operaţii;
- b) *doMove():* realizează efectiv migrarea agentului din locaţia curentă în cea destinaţie;
- c) *afterMove():* metodă apelată imediat după finalizarea migrării, când agentul se află în containerul destinaţie;

Când un agent migrează, nu toate variabilele sale de stare sunt mutate odată cu el. Un exemplu îl constitue parametrii dependenţi de platformă (container) cum ar fi ontologia şi limbajul descriptiv. De aceea, dupa ce agentul a fost mutat, acesta trebuie să le reînregistreze. Procesul de mutare al unui agent din containerul curent într-un container din aceeaşi platformă se realizează printr-un simplu apel al funcţiei *doMove()*, ce aparţine clasei **Agent**. JADE pune la dispoziţie două metode ce pot fi suprascrise pentru a realiza o serie de funcţii necesare operaţiilor pe care agentul le are de făcut înainte de a părăsi containerul curent, prin metoda *beforeMove()* şi a operaţiilor imediat următoare mutării, când agentul se află deja în noul container, prin metoda *afterMove()*. Aceste două metode sunt apelate automat de către JADE, înainte, respectiv după ce agentul a trecut din starea ACTIVE în starea TRANSIT, specifică ciclului de viaţă al agentului mobil.

9.2 Clonarea agenţilor

Procesul clonării unui agent este similar celui de migrare. În afara locaţiei destinaţie trebuie specificat şi numele clonei căruia îi trebuie asigurată unicitatea la nivelul platformei.

Se pot folosi metode de prefixare ori sufixare a numelui agentului clonat pentru formarea de nume unice pentru agenţii clonă.

Metodele clasei **jade.core.Agent** asociate clonării agentului sunt:

- a) *beforeClone():* metoda apelată înainte de clonare;
- b) *doMove(Location l, String name):* realizează efectiv clonarea în locaţia destinaţie ;
- c) *afterClone()*: metoda apelată imediat după finalizarea clonării, când agentul clonă se află în containerul destinaţie;

9.3 Utilizarea ontologiei MobilityOntology

Deşi operaţiile de migrare ori clonare a unui agent pot fi iniţiate autonom de către acesta, logica din spatele deciziei putând fi complet decuplată de factorii externi; în marea majoritate a cazurilor însă, aceste operaţii sunt urmări ale unor interacţiuni pe care agentul le are cu alţi agenţi, ale unor cerinţe directe pentru realizarea acestora [23, pag 187] .

Domeniul mobilităţii agenţilor este descris în cadrul ontologiei **MobilityOntology**, pe baza căreia agenţii pot raţiona şi comunica folosind aceleaşi concepte, predicate acţiuni.

Pachetul jade.domain.mobility conţine definiţia ontologiei precum şi a vocabularului, conceptelor, acţiunilor, predicatelor aferente. Ontologia conţine:

- a) 2 acţiuni: migrare (MoveAction) şi clonare (CloneAction);
- b) 5 concepte: identificate prin clasele MobileAgentDescription, MobileAgentProfile; MobileAgentSystem etc.;

Ontologia va fi folosită în cazul interacţiunii standardizate cu agentul AMS prin care i se va cere efectuarea unei operaţii de mutare/clonare a unui agent ori în cazul interacţiunii între doi agenţi ce implementează un protocol de interacţiune propriu orientat spre realizarea acestor operaţii (de eemplu dacă un agent îi va cere celuilalt să migreze în alt container, cei doi trebuie să se "inţeleagă" printr-un protocol tip FIPA-Request).

Pentru specificarea tipului de operaţie, se vor folosi instanţe ale **claselor MoveAction sau CloneAction**. Acestea necesită un parametru de tip **MobileAgentDescription** ce conţine sloturi pentru locaţia destinaţie si AID-ul agentului la care se referă operaţia. Când un agent va primi un mesaj cu o astfel de acţiune, va extrage din obiectul **MobileAgentDescription** locaţia destinaţie şi va apela metoda *doMove(...) ori doClone(...)* după caz.

9.4 Mobilitatea indirectă

Mutarea unui agent poate fi făcută şi din exteriorul agentului ce urmează a fi mutat, folosind serviciul AMS (Agent Management Service) din containerul în care se afla agentul (prin intermediul clasei **AMSService**). Acestuia i se pot trimite mesaje cu un limbaj standardizat (FIPA-SL0) prin care să se ceară mutarea unui agent în alt container. Această procedură este simplificată prin acţiunea move-agent (clasa MoveAction), pentru care se vor specifica numele agentului ce trebuie mutati destinatia (ContainerID sau PlatformID). În JADE există mai multe clase de **tipAction** ce comunică în diverse scopuri cu serviciul AMS, astfel că se recomandă folosirea metodei următoare pentru a stabili ontologia şi tipul comunicaţiei cu acesta, indiferent de acţiunea pe care dorim să o executăm [15]:

```
public class MigrareIndirecta
{ // Trimite o cerere de raspuns de la  AMS
public static void sendRequest
(Agent caller, Action action)
// inregistrarea ontologiei si a limbajului de comunicare cu AMS
caller.getContentManager().registerLanguage
(new SLCodec());
caller.getContentManager().
registerOntology(MobilityOntology.getInstance());
// Trimite o cerere la AMS
ACLMessage request =
new ACLMessage(ACLMessage.REQUEST);
request.setLanguage(new SLCodec().getName());
request.setOntology(MobilityOntology.
getInstance().getName());
try{
caller.getContentManager().fillContent
(request, action);
request.addReceiver(action.getActor());
caller.send(request);}
catch (Exception ex){
System.out.println(ex.StackTrace);
}}}
```

Observaţie 16 *Putem să folosim metoda sendRequest() pentru a trimite cererea de mutare a unui agent.*

```
public class MobilityServiceAgent extends Agent
{
private AID agentToMove;
private ContainerID newContainer;
public MobilityServiceAgent
(AID AgentToMove,ContainerID NewContainer)
{
agentToMove = AgentToMove;
newContainer = NewContainer;
}
@Override
public void setup()
{ super();
addBehaviour(new RequestToMoveAgentBehaviour
(agentToMove, newContainer));
takeDown();}}
public class RequestToMoveAgentBehaviour
extends OneShotBehaviour
{
private AID agentToMove;
private ContainerID newContainer;
public RequestToMoveAgentBehaviour
(AID AgentToMove, ContainerID NewContainer)
{
agentToMove = AgentToMove;
newContainer = NewContainer;}
@Override
public void action()
{
MoveAction moveAction = new MoveAction();
MobileAgentDescription describeMovement
= new MobileAgentDescription();
```

```
describeMovement.setName(agentToMove);
describeMovement.setDestination(newContainer);
moveAction.setMobileAgentDescription(describeMovement);
Common.sendRequest(myAgent,new Action(myAgent.getAMS(),
moveAction));}}
```

9.5 Aplicaţie

Să se scrie o aplicaţie Jade în care un agent (iniţiator), să migreze în containerul Main, apoi agentul (responder) să fie clonat în toate containerele active din platformă. Se va folosii ontologia MobilityOntology, ce conţine vocabularul specific precum şi conceptele şi predicatele corespunzătoare migrarii şi clonarii agenţilor [15].

```
import java.util.*;
import jade.content.*;
import jade.content.lang.sl.*;
import jade.content.onto.basic.*;
import jade.core.*;
import jade.domain.JADEAgentManagement.*;
import jade.domain.mobility.*;
import jade.lang.acl.*;
**************************************************************
//Procesul clonarii unui agent este similar celui de migrare
//In afara locatiei destinatie trebuie specificat
//si numele clonei caruia ii trebuie asigurata
//unicitatea la nivelul platformei
**************************************************************
public class Clonare extends Agent{
private Hashtable locations =
new Hashtable();
 public Clonare() {}
 public void setup() {
//1.inregistreaza codecul si ontologia specifica
getContentManager()
.registerLanguage(new SLCodec());
```

```
getContentManager().registerOntology
(MobilityOntology.getInstance()); }
}
 @SuppressWarnings("unchecked")
void sendRequest(Action action) {
ACLMessage request =
new ACLMessage(ACLMessage.REQUEST);
request.setLanguage(new SLCodec().getName());
request.setOntology
(MobilityOntology.getInstance().getName());
//2.Cere locatiile platformei de la AMS
sendRequest(new Action(getAMS(),
new QueryPlatformLocationsAction()));
//3.Receive response from AMS
MessageTemplate mt =
MessageTemplate.and( MessageTemplate.
MatchSender(getAMS()),
MessageTemplate.MatchPerformative(ACLMessage.INFORM));
 ACLMessage resp = blockingReceive(mt);
try {
//4.extrage raspunsul din mesaj..
// Raspunsul contine o colectie de locatii.
ContentElement ce = getContentManager().
extractContent(resp);
Result result = (Result) ce;
jade.util.leap.Iterator it = result.getItems().iterator();
while (it.hasNext()) {
Location loc = (Location) it.next();
locations.put(loc.getName(), loc);
//5. se cloneaza agentul in locatia Specificata
doClone(loc, "" + getLocalName());}}
// se va da doar numele agentului care va
// fi clonat intr-un alt container
catch (Exception ce) {
```

```
ce.printStackTrace();}}
public void beforeClone() {
System.out.println(getLocalName() +
": Cloning...cloning...");}
public void afterClone() {
 System.out.println(getLocalName() +
": It's so good to be cloned!"); }}}

**************************************************
import java.util.*;
import jade.content.*;
import jade.content.lang.sl.*;
import jade.content.onto.basic.*;
import jade.core.*;
import jade.domain.JADEAgentManagement.*;
import jade.domain.mobility.*;
import jade.lang.acl.*;
public class MobileAgent extends Agent {
private Hashtable locations = new Hashtable();
public MobileAgent() { }
 public void setup() {
//1.inregistreaza codecul si ontologia specifica
getContentManager().registerLanguage(new SLCodec());
getContentManager().registerOntology
(MobilityOntology.getInstance());
**********************************************
//2.cere locatia containerului pe care
// se afla AMS, deci a containerului principal
**********************************************
 WhereIsAgentAction requestAction =
new WhereIsAgentAction();
requestAction.setAgentIdentifier(getAMS());
 sendRequest(new Action(getAMS(),requestAction));
**********************************************
```

```
//3.asteapta raspunsul AMS-ului
*********************************************
 MessageTemplate mt = MessageTemplate.and(
 MessageTemplate.MatchSender(getAMS()),
 MessageTemplate.MatchPerformative
(ACLMessage.INFORM));
ACLMessage resp = blockingReceive(mt);
 try {
*********************************************
//4.extrage raspunsul din mesaj
// Raspunsul este un obiect de tip Location.
*********************************************
 ContentElement ce = getContentManager().extractContent(resp);
 Result result = (Result) ce;
if (result.getValue() != null) {
//5. migreaza catre noua locatie
 doMove( (Location) result.getValue()); } }
 catch (Exception ex) {
System.out.println("Operatia nu s-a putut efectua!
\nErr:" + ex.getMessage()); } }
*****************************************************
// Trimite un mesaj de cerere continand actiunea
// folosind codecul SL si ontologia
*****************************************************
 void sendRequest(Action action) {
 ACLMessage request = new ACLMessage(ACLMessage.REQUEST);
request.setLanguage(new SLCodec().getName());
 request.setOntology
(MobilityOntology.getInstance().{getName());
try {
 getContentManager().fillContent(request,action);
 request.addReceiver(action.getActor());
 send(request);}
 catch (Exception ex){
```

```
 ex.printStackTrace(); }}
 public void beforeMove(){
 System.out.println(getLocalName() +
": Moving...moving...");}
 public void afterMove(){
 System.out.println(getLocalName() +
": It's so good to touch ground!"); }
public void beforeClone() {
System.out.println(getLocalName() +
": Cloning...cloning..."); }
 public void afterClone() {
 System.out.println(getLocalName() + ":
It's so good to be cloned!");
}
```

Observaţie 17 *Pentru Iniţiator şi Responder vom folosi exemplul prezentat* ***în capitolul 7, paragraf 7.2 /pag 61.***

La terminarea executiei aplicaţiei în frameworkul de Jade în maincontainer găsim agentul Iniţiator, iar în toate celelalte containere se găseste agentul Responder.

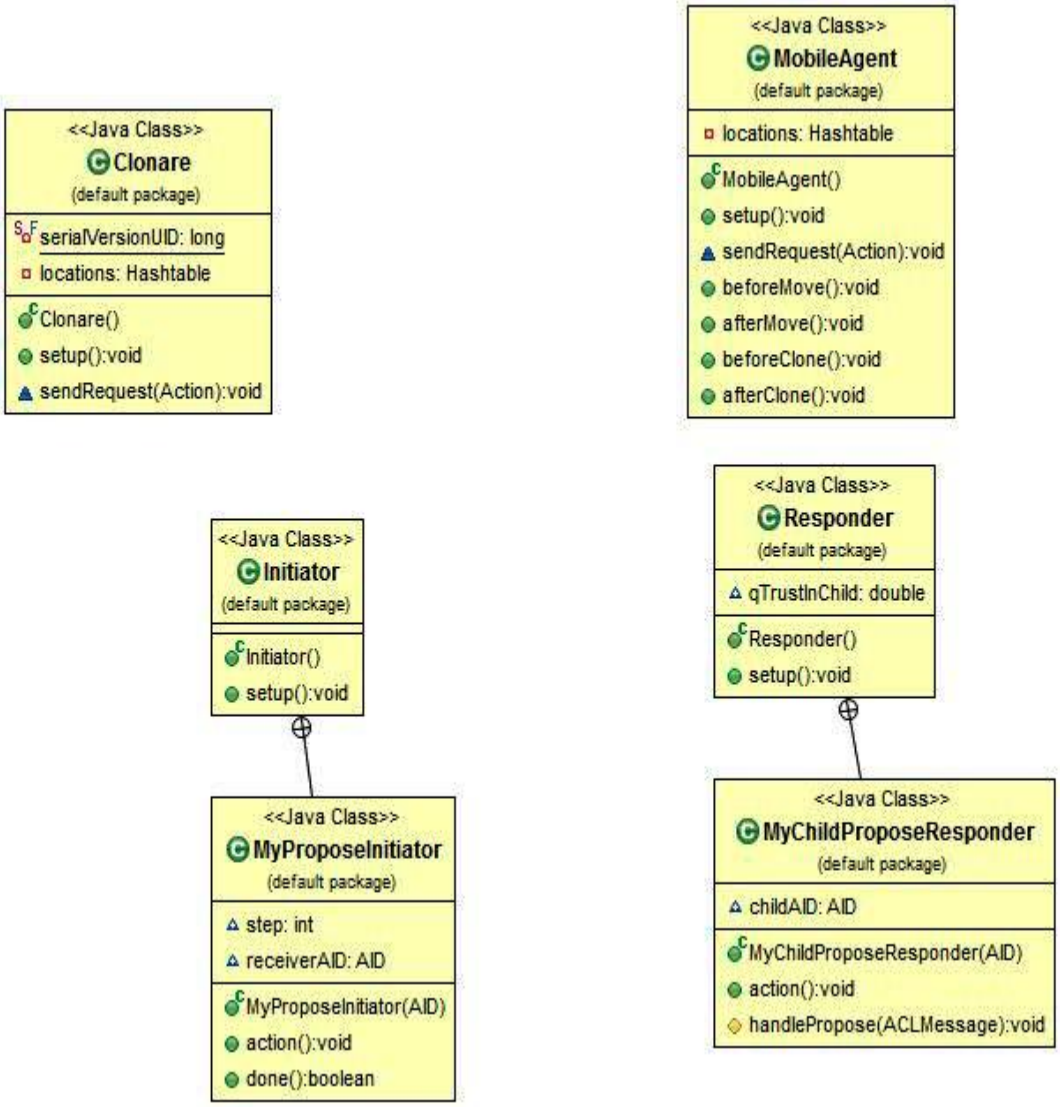


Figura 9.1: Diagrama UML-Migrare

9.6 Probleme propuse

1. *Să se creeze două containere, primul (principal) va conţine:*

a) *un agent* **DisplayAgent** *în containerul principal care afişează informaţiile primite de la alţi agenţi într-o fereastră.*

b) *un agent mobil* **ReadFileMobileAgent** *în containerul principal care citeşte o listă de preţuri de produse dintr-un fişier şi le transmite apoi la agentul* **DisplayAgent.**

Contanerul doi (secundar) să conţină un agent **FilePathInfoAgent** *care ştie din ce fişier trebuie citite informaţiile cu lista de produse.*

Cerinţe: Să se implementeze următorul comportament: Agentul **ReadFileMobileAgent** *se va muta în containerul secundar şi va cere prin intermediul mesajelor agentului* **FilePathInfoAgent** *locaţia în care se află fişierul cu lista de produse şi va trimite această listă la agentul* **DisplayAgent.**

2. *[23] a)Creaţi un sistem multi-agent compus din cel putin două containere: un Main-Container reprezentând nava StarTrek, ce orbitează în jurul stelei Antares, supergigantul galaxiei, simulată printr-un container simplu.*

b) Formaţi o echipă de agenţi la bordul navei ce va coborâ pe stea pentru a prelua informaţii esenţiale omenirii, de la conducătorul acesteia, pe baza unui protocol de comunicaţie securizat. Echipa trebuie să aibă un conducator.

c) Trimiteţi echipa de agenţi la sol (prin migrare).

d) La semnalul conducătorului echipei, după preluarea informaţiilor, echipa trebuie să se teleporteze înapoi la bord.

3. *Să se scrie o aplicaţie JADE în care un agent (Iniţiator) să migreze în maincontainer, iar agentul (Responder) să fie clonat în toate containerele active din platformă. Se va folosii ontologia* **MobilityOntology,** *ce conţine vocabularul specific şi conceptele, predicatele corespunzătoare migrării şi clonării agenţilor.Limbajul descriptiv folosit va fi* **SL.**

Capitolul 10

Licitaţii

10.1 Aspecte generale

Licitaţiile au loc între un agent numit licitator şi mai mulţi agenţi numiţi cumpărători. Scopul unei licitaţii este ca licitatorul să aloce un bun unuia dintre cumpărători. De obicei interesul licitatorului este să maximizeze preţul, în timp ce al cumpărătorului este să-l minimizeze [15] .

În practică, există mai multe tipuri de licitaţii. Printre cele mai frecvent întâlnite se numără:

1. licitaţia englezească,
2. licitaţia olandeză şi
3. tipul de licitare închis.

10.2 Descrierea licitatiilor

10.2.1 Tipuri de evaluare a bunului licitat

- - valori private: valoarea bunului depinde doar de preferinţele personale ale cumpărătorului; acesta se presupune că ştie valoarea bunului;
- - valori comune: valoarea unui produs pentru un cumpărător depinde de valoarea acelui produs pentru alţi cumpărători;

- - valori corelate: valoarea unui produs pentru un cumpărător depinde parţial de preferinţele sale personale şi parţial de valoarea acelui produs pentru alţii.

10.2.2 Strategii în licitaţii

Strategia unui cumpărător în licitaţii constă în modul în care va licita şi în acţiunile pe care le va întreprinde cu scopul de a obţine profit cât mai mare. Strategia dominantă este cea care generează cele mai bune rezultate din punctul de vedere al profitului. Cumpărătorul va obţine profitul maxim folosind strategia dominantă.

10.2.3 Tipuri de licitaţie

English (first-price open cry)

Aceasta este un tip de licitaţie frecvent întâlnit, în cadrul căruia cel care licitează începe la preţul de rezervă al obiectului şi oferta înaintează progresiv până când nici un participant nu emite o ofertă mai mare. În această licitaţie, fiecare ofertă ţine cont de celelalte oferte citate.

- - licitaţia porneşte de la un preţ stabilit;
- - cumpărătorii pot anunţa deschis preţul pe care îl licitează, care trebuie să fie mai mare decât preţul licitat până în acel moment;
- - câştigă cumpărătorul care a oferit cel mai mare preţ.

Observaţie 18 - *Strategie dominantă: cumpărătorul oferă puţin mai mult decât ultimul preţ şi se opreşte când ajunge la valoarea reală a bunului.*

- Winner's curse câştigătorul licitaţiei obţine bunul licitat la un preţ exagerat în cazul incertitudinii referitoare la valoarea reală a bunului licitat

- Licitatorul poate plasa cumpărători falşi pentru a creşte artificial preţul.

First-price sealed-bid

La acest tip de licitaţie, cumpărătorii emit oferte în mod secret. Nici un ofertant nu cunoaşte cota unei alte oferte. Cel mai mare ofertat este premiat cu obiectul corespunzător preţului ofertei sale.

1. - o singură rundă de negociere,

2. - fiecare cumpărător anunţă preţul în plic închis;

3. - câştigă cel cu preţ maxim.

Observaţie 19 *Strategie dominantă: cumpărătorul face o ofertă egală cel mult cu valoarea privată a bunului. Nu există o soluţie generală pentru a determina cât de mică ar trebui să fie această ofertă.*

Dutch (descending)

În acest caz, licitatorul porneşte de la un preţ ridicat şi solicită un set de oferte descendente până când un cumpărător acceptă bunul la preţul curent. O ofertă suficient de mică propusă chiar la începutul licitaţiei asigură faptul că bunul va fi vândut în cele din urmă, dar există posibilitatea ca licitatorul să nu obţină profit.

1. - licitaţia porneşte de la un preţ stabilit,

2. - licitatorul scade preţul până când unul din participanţi acceptă preţul.

Observaţie 20 - *Susceptibilă la winner's curse (situaţie în care câştigătorul ajunge să plătească mai mult decât valorează obiectul pe care l-a câştigat prin licitaţie)*

- *Nu există strategie dominantă.*

Vickrey (second-price sealed-bid) auction

Cumpărătorii fac oferte în mod secret. Câştigă autorul celei mai mari oferte, care va primi bunul corespunzător la preţul dat de a doua ofertă. La această licitaţie, cel mai mare ofertant obţine întotdeauna profit, dacă oferta acestuia este raţională.

1. - o singură rundă de negociere;

2. - ofertele sunt secrete faţă de ceilalţi cumpărători

3. - câştigă cel care a făcut cea mai mare ofertă, dar plăteşte al doilea preţ.

Observaţie 21 - *Strategia dominantă: se licitează valoarea reală a bunului.*

- *Avantajul acestui tip de licitaţie este faptul că încurajează cumpărătorii să fie sinceri*

- *Câştigătorul licitaţiei nu poate şti cât a licitat cel care a pierdut (locul al doilea), deci nu ştie dacă preţul plătit (al doilea preţ) este corect. O soluţie ar fi folosirea de semnături digitale sau a unei terţe părţi de încredere.*

- Se poate dezvolta un comportament antisocial : dacă un cumpărător ştie că va pierde, poate creşte intenţionat preţul pentru a-i face pe alţii să plătească mai mult.

10.2.4 Coaliţii (collusion)

Ori de câte ori la o licitaţie ofertanţii cunosc identitatea celorlaltor participanţi, există riscul ca aceştia să se asocieze în scopul manipulării rezultatului licitaţiei, o practică cunoscută sub numele de `collusion (întelegere secretă, coaliţie).`

1. - Nici un tip de licitaţie nu este imun la coaliţii,
2. - Se pot forma coaliţii care să menţină preţul scăzut,
3. - Câştigătorul obiectului (care trebuie să facă parte din coaliţie) împarte profitul cu ceilalţi.

O altă practică ilegală prin care se poate a creşte preţul unui bun licitat este următoarea: proprietarul bunului participă la licitaţie, crescând astfel concurenţa, licitând fictiv şi ieşind din licitaţie chiar înainte de valoarea licitată finală.

În cazul licitaţiilor cu plic închis (oferte secrete), dacă un cumpărător reuşeşte să afle ofertele celorlalţi participanţi la licitaţie, el poate:

1. - să liciteze direct un preţ care îl va face să câştige;
2. - să se retragă, ştiind de la început că nu poate câştiga.

10.3 Aplicaţie

Următoarea aplicaţie **Licitaţie** implementează o variantă simplificată de licitaţie **English.**

Se presupune că există următoarele bunuri de licitat:

1. "Ceas Tissot",
2. "Oglinda Venetiana",
3. "Tablou de Picasso",
4. "Colecţie timbre",

5. ”Vază de cristal Ludovic al XIV-lea”.

```
import jade.core.AID;
import jade.core.Agent;
import jade.domain.FIPAException;
import java.util.LinkedList;
import java.util.List;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;
@SuppressWarnings("serial")
public class Licitatie extends Agent {
 public List<AID> participantList;
 public String[] productList =
{"Ceas Tissot", "Oglinda-Venetiana", "Tablou de Picasso",
"Colectie timbre", "Vaza de cristal Ludovic al XIV-lea "};
 public String currentProduct;
 public int currentPrice = 0;
 public AID currentWinner = null;
 int maxStartingPrice = 100;
 int auctionCallInterval = 3000;
 @Override
public void setup() {
 participantList = new LinkedList();
currentProduct =
productList[(new Random()).
nextInt(productList.length)];
currentPrice =
new Random().nextInt(maxStartingPrice);
 try {
YellowPages.RegisterService("Licitatie",
 this);}
catch (FIPAException ex) {
 Logger.getLogger
(AgentAuction.class.getName()).log
```

```
(Level.SEVERE, null, ex);}
 try { Thread.sleep(100);}
catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();}
System.out.println(this.getName() +
" : Serviciul 'Licitatie' a fost inregistrat.");
addBehaviour(new ManagerReceive(this));
addBehaviour(new ManagerSend(this,
auctionCallInterval));}}
```

Liciataţia este gestionată de un **managerAgent** care înregistrează serviciul **YelowPage** corespunzător licitaţiei:

```
import jade.core.AID;
import jade.core.Agent;
import jade.domain.DFService;
import jade.domain.FIPAAgentManagement.DFAgentDescription;
import jade.domain.FIPAAgentManagement.ServiceDescription;
import jade.domain.FIPAException;
public class YellowPages {
 // returneaza AID-urile furnizorilor
public static AID FindService
(String serviceName,Agent myAgent,int timeOut)
throws FIPAException {
 AID providerAID = null;
 boolean found = false;
 double t1 =
System.currentTimeMillis();
 while (!found) {
if (System.currentTimeMillis() - t1 > timeOut){
 break;}
// cauta un furnizor
DFAgentDescription template =
new DFAgentDescription();
```

```
 ServiceDescription sd =
new ServiceDescription();
 sd.setType(serviceName);
 template.addServices(sd);
 DFAgentDescription[] result =
DFService.search(myAgent,template);
 if (result != null && result.length > 0) {
 providerAID = result[0].getName();
 found = true; } }
 return providerAID; }
// Inregistreaza serviciul solicitat de un agent
public static void RegisterService
(String serviceName, Agent myAgent)
throws FIPAException {
DFAgentDescription dfd =
new DFAgentDescription();
 dfd.setName(myAgent.getAID());
ServiceDescription sd = new ServiceDescription();
 sd.setType(serviceName);
 sd.setName(serviceName);
 dfd.addServices(sd);
 DFService.register(myAgent, dfd); }
 // Elimina serviciul solicitat de un anumit agent
 public static void DeregisterService(Agent myAgent)
throws FIPAException {
 DFService.deregister(myAgent);}
 public static void {
DeregisterServiceOnTakeDown(Agent myAgent)
throws FIPAException {
 DFService.deregister(myAgent); }}
*****************************************************
// Se asteapta solicitari de inregistrare si oferte de
// pret de la agentii cumparatori in
//cadrul comportamentului de tip ciclic ManagerReceive.
```

```
*****************************************************
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
public class ManagerReceive extends CyclicBehaviour {
private AgentAuction myAgent;
public ManagerReceive(AgentAuction a){
 super(a); myAgent = a;}
 @Override
 public void action() {
 ACLMessage m = myAgent.receive();
if (m != null) {
 String received = m.getContent();
 AID sender = m.getSender();
 if(received.contains("registering")) {
 //Un participant la licitatie doreste sa se inregistreze
 myAgent.participantList.add(sender);
***************************************
//raspunde agentului de tip initiator
//ca inregistrarea s-a facut cu succes
***************************************
 ACLMessage reply = new ACLMessage(ACLMessage.INFORM);
 reply.setContent("registered");
reply.addReceiver(sender);
 myAgent.send(reply);
 System.out.println(myAgent.getName() +
 : a aparut un nou participant: " +
sender.getLocalName());
 if (received.contains("offer")) {
 //asteapta o noua oferta a agentului
 int receivedPrice =
Integer.parseInt(received.substring(5));
System.out.println(myAgent.getName() + " : a primit oferta " +
receivedPrice + " de la:" + sender.getLocalName());
```

```
 if (receivedPrice > myAgent.currentPrice){
myAgent.currentPrice = receivedPrice;
 myAgent.currentWinner = sender?}}
 if (received.contains("retreat"))
*************************************************************
// agentul se retrage din licitatie?
//daca da, se trimite mesajul organizatorului licitatiei ca
// s-a retras din licitatie
*************************************************************
{ myAgent.participantList.remove(sender);
 System.out.println(myAgent.getName() + " : "
+ sender.getLocalName() + "
s-a retras din licitatie.");}}
else {
 block(); } }}
```

În sistem apar agenţii cumpărători care solicită **manager agentului** să-i înregistrareze la licitaţie prin comportamentul **BuyerRegister** de tip **oneshotbehaviour:**

```
import jade.core.AID;
import jade.core.behaviours.OneShotBehaviour;
import jade.lang.acl.ACLMessage;
************************************
// agentul care solicita inscrierea
// in licitatie acceptind conditiile
************************************
class BuyerRegister extends OneShotBehaviour{
 private BuyerAgent myAgent;
 public BuyerRegister(BuyerAgent a)
super(a);
myAgent = a;}
 @Override
public void action(){
```

```
//informeaza organizatorului Licitatiei(manager agent)
// ca doreste sa se inscrie la licitatie
ACLMessage m = new ACLMessage(ACLMessage.INFORM);
AID receiverAID = new AID("AuctionManager", AID.ISLOCALNAME);
m.addReceiver(receiverAID);
m.setContent("registering");
myAgent.send(m);
System.out.println(myAgent.getName() + " :
se inscrie la licitatie ... ");
}}}
```

Manager-agentul adaugă o structură de date proprie AID-urile agenţilor cumpărători care solicită înregistrarea şi le confirmă faptul că au fost înregistraţi cu succes,(în comportamentul **ManagerReceive**):

```
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
public class ManagerReceive extends CyclicBehaviour {
 private AgentAuction myAgent;
 public ManagerReceive(AgentAuction a)
 super(a);
 myAgent = a; }
 @Override
public void action(){
 ACLMessage m = myAgent.receive();
if (m != null)
 String received = m.getContent();
 AID sender = m.getSender();
if(received.contains("registering"))
// agentul cumparator solicita inscrierea in licitatie
 myAgent.participantList.add(sender);
**************************************************
// managerAgent ii comunica ca
//inregistrarea s-a efectuat cu succes
```

```
*****************************************************
ACLMessage reply = new ACLMessage(ACLMessage.INFORM);
 reply.setContent("registered");
reply.addReceiver(sender);
 myAgent.send(reply);
 System.out.println(myAgent.getName() +
" : a aparut un nou participant: " +
sender.getLocalName());}
if (received.contains("offer")){
//se receptioneaza o noua oferta de pret pentru agent
 int receivedPrice =
Integer.parseInt(received.substring(5));
 System.out.println(myAgent.getName() + " :
a primit oferta " + receivedPrice +
" de la: " + sender.getLocalName());
 if (receivedPrice > myAgent.currentPrice) {
myAgent.currentPrice = receivedPrice;
myAgent.currentWinner = sender;} }
if (received.contains("retreat")){
*********************************************************
// daca agentul se retrage?
//se comunica platformei retragerea agentului din licitatie
*********************************************************
myAgent.participantList.remove(sender);
System.out.println(myAgent.getName() + " : "
+ sender.getLocalName() + " s-a retras din licitatie."); } }
 else {
 block(); } }}
```

Licitaţia se desfaşoară în runde, care sunt iniţiate (încheiate periodic) de către **Manager Agentul** în comportamentul **ManagerSend** de tip **TickerBehaviour.** O rundă constă în primirea şi evaluarea ofertelor de preţ de la agenţi cumpărători. La inceput **ManagerAgentu-ul** trimite cumpărătorilor un preţ iniţial al produsului, pentru ca apoi în fiecare rundă să le trimita cel mai mare preţ primit până în acel moment. La primirea informaţiilor legate de preţ (in cadrul comportamentului **BuyerReceive)** agenţii

cumpărători decid, cu o anumită probabilitate, dacă fac o ofertă de preţ mai mare sau dacă se retrag din licitaţie. Ei trimit **ManagerAgent-ului** răspunsul corespunzător:

```
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;
import java.util.Random;
public class BuyerReceive extends CyclicBehaviour {
 private BuyerAgent myAgent;
 public BuyerReceive(BuyerAgent a) {
 super(a);myAgent = a;}
 @Override
 public void action() {
ACLMessage m = myAgent.receive();
if (m != null) {
 String received = m.getContent();
if (received.contains("registered")) {
//managerul imi inregistreaza actiunea
System.out.println(myAgent.getName() + " :
a fost inscris cu succes in licitatie!");}
if (received.contains("current_price")) {
int currentPrice = Integer.parseInt(received.substring(13));
System.out.println(myAgent.getName() + "
: a primit pretul curent:" + currentPrice);
myAgent.probabilityToBid *=
myAgent.decreaseFactor;
ACLMessage toSend =
new ACLMessage(ACLMessage.INFORM);
toSend.addReceiver(m.getSender());
if (myAgent.probabilityToBid > Math.random()) {
int addedValueToCurrentPrice = 1 +
{new Random().nextInt(myAgent.maxOfferIncrease);
int newPrice = currentPrice + addedValueToCurrentPrice;
toSend.setContent("offer" + newPrice);
System.out.println(myAgent.getName() +
" : a trimis noua oferta: " + newPrice);}
```

```
else {
toSend.setContent("retreat");
System.out.println(myAgent.getName() + "
: s-a retras din licitatie. ");}
myAgent.send(toSend);}
else {block();}}}
```

Licitaţia se încheie când fie rămâne un singur cumpărător (care va fi câştigatorul), fie se retrag toţi cumpăratorii (comportamentul **ManagerSend).**

```
import jade.core.AID;
import jade.core.behaviours.TickerBehaviour;
import jade.lang.acl.ACLMessage;
public class ManagerSend extends TickerBehaviour {
private AgentAuction myAgent;
public ManagerSend(AgentAuction a,long period)
 {super(a, period);
 myAgent = a;}
 @Override
 public void onTick()
{ int noOfParticipants =
myAgent.participantList.size();
}
 if (noOfParticipants > 1){
**********************************************************
// Daca au ramas la licitatie mai mult decit un agent
// Se incepe o noua runda si se trimite participantilor
// ultimul pret si le cere o noua oferta de pret
**********************************************************
ACLMessage toSend = new ACLMessage(ACLMessage.INFORM);
 toSend.setContent("current_price" + myAgent.currentPrice);
 for (AID a : myAgent.participantList) {
 toSend.addReceiver(a); }
 System.out.println(myAgent.getName() + " :
a inceput o noua runda! Produs lictat:
```

```
 " + myAgent.currentProduct + "
Pretul initial: " + myAgent.currentPrice);
 myAgent.send(toSend);}
 else if (noOfParticipants = = 1)
 { if(myAgent.currentWinner != null)
 AID winner = myAgent.currentWinner;
 System.out.println(myAgent.getName() +
" : licitatia s-a incheiat.
Castigatorul pentru "+ myAgent.currentProduct + " este " +
winner.getLocalName() + " cu pretul: " +
myAgent.currentPrice);}
 else
System.out.println(myAgent.getName() + "
: licitatia s-a incheiat deoarece are prea putini participanti!");}
 this.stop(); }}
 else {
System.out.println(myAgent.getName() +
" : licitatia s-a incheiat deoarece
toti participantii s-au retras.");
 this.stop();}}}
```

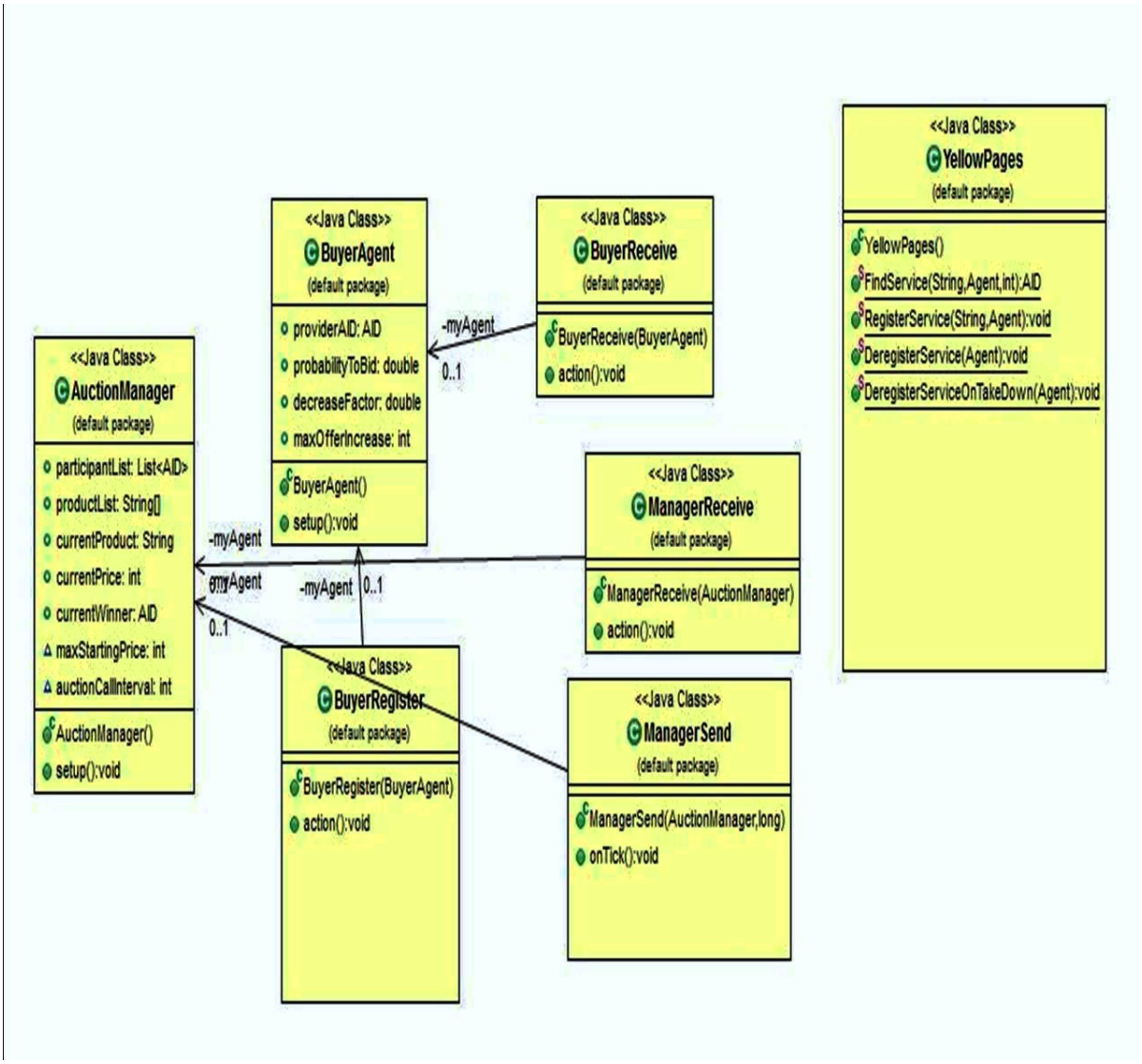


Figura 10.1: Diagrama UML-Licitaţie

10.4 Probleme propuse

1. *Să se implementeze un protocol de licitaţie [15]. Caracteristici:*

a) *licitatorul va avea o listă de produse ,*

b) *cumpărătorii vor avea o un preţ (valoare privată) care poate varia +/- 10 % de la un agent la altul pentru fiecare produs,*

c) *cumpărătorii se vor înregistra ca participanţi la licitaţie,*

d) *licitatorul va comunica startul, informând asupra obiectului scos la licitaţie: produs, caracteristici, perioda de aşteptare a ofertelor,*

e) *cumpărătorii vor trimite câte o ofertă de preţ,*

f) *după* ce *perioada de aşteptare a ofertelor a expirat, licitatorul va analiza ofertele primite şi va trimite mesaje celor care vor pierde licitaţia. Acestea vor conţine informaţii despre preţul dat de câştigătorul licitaţiei precum şi preţul care îl va plăti acesta pentru produs*

g) *un mesaj câştigătorului licitaţiei ce va conţine informaţii despre preţul oferit de acesta şi preţul ce va trebui plătit.*

2. *Să se creeze un market virtual alcătuit dintr-un număr de agenţi cumpărători şi un număr de agenţi vânzători [3]. Să se implementeze un S.M.A cu următoarele cerinţe:*

i) *vânzătorii vor avea o listă de produse cu anumite cantităţi şi preţuri ce se vor înregistra într-un serviciu de tip "Yellow Pages" ,*

ii) *vânzătorii vor avea un preţ maximal şi unul minimal precum şi un algoritm de discounturi (de exemplu pentru mai mult de cinci produse de un anumit tip o reducere de 2% , dar nu mai puţin decât preţul minim),*

iii) *după vânzare stocurile aferente respectivului produs vor scădea,*

iv) *vânzătorii vor accepta cererea doar dacă pot să o satisfacă (de exemplu au cantitatea respectivă în stoc), altfel o vor refuza.*

v) *cumpărătorii vor putea face anunţul despre produs şi cantitatea dorită, căutând vânzătorii în "Yellow Pages",*

vi) *cumpărătorii vor avea o listă de produse din care vor alege aleator un produs şi cantitatea pe care o doresc să o achiziţioneze şi vor face periodic câte un anunţ de achiziţie. Un anunţ se va putea face doar după ce achiziţia în curs s-a finalizat.*

vii) *criteriul de stabilire a câştigătorului licitaţiei va fi cel mai mic preţ.*

Bibliografie

[1] Athanasiu I. –Utilizarea RMI in Java, *PC Report and BYTE, pp 45-49, februarie 1999.*

[2] Bărbat B. –*Sisteme inteligente orientate spre agent*, Ed. Academiei Române, 2002.

[3] d'Inverno M.,Luck M. –*Understanding Agent Systems (Springer Series on Agent Technology)*, Second Edition, Springer, 2010.

[4] Bellifemine F.,Caire G.,Greenwood D. –*Developing multiagent systems with JADE,* Wiley Series in Agent Technology, 2007.

[5] Caire G.,Cabanillas J. –*Application-defined content languages and ontologies,* `http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf`, 2006.

[6] Caire G. –`http://www.iro.umontreal.ca/~dift6802/jade/src/example/ontology`

[7] Dalpiaz F. –*Yellow pages and Interaction Protocols,* Agent-Oriented Software Engineering (AOSE), 2009-2010.

[8] EJADE –*Eclipse IDE Plugin for Java Agent Development Environment,* `http://dit.unitn.it/~dnguyen/ejade/`, 2008.

[9] FIPA –*The Foundation for Intelligent Physical Agent.*, `http://www.fipa.org/`, 2008.

[10] Henderson B., Giorgi P., Bresciani P.,–*The gaya methodology for agent-oriented analysis and design,* 2002.

[11] Horn P.– *Manifesto. Autonomic Computing:* IBM's Perspective on the State of IT, October 2001.

[12] JADE –`http://jade.tilab.com`.

[13] JADEX Active Components– www.activecomponents.org/bin/view/About/New+Home.

[14] JESS –*Jess,the Rule Engine for Java Platform,* http://herzberg.ca.sandia.gov/, 2008.

[15] Leon F.–*Modelarea şi analiza sistemelor multi-agent*, http://florinleon.byethost24.com, 2010.

[16] Luck M.,Ashri J.–*Agent-Based Software Development,* Artech House, 2004.

[17] Parunak V.,Wilensky U., Rand W.,–An introduction to agent based modeling : natural, social and engineered complex sistem – *The MIT Press, Cambridge, MA, 2015. 504 pp. Type: Book (978-0-262731-89-8), Reviews: (1 of 2)*

[18] Tropos –http://www.troposproject.org/, 2014.

[19] UML –*Object Management Group,* http://www.uml.org, 2008.

[20] http://sourceforge.net/projects/spyse/

[21] Villa X.,Schuster A.,Riera A., –Security for a Multi-Agent System based on JADE , *Elsevier Computers & Security pp: 391-400, 2007.*

[22] Wooldridge M. –*An introduction to multi-agent system,* John Wiley & Sons, 2002.

[23] Zambonelli F.,Omicini A. –*Autonomous Agents and Multi-Agent,* Kluver Academic PubliShers, 2004.

[24] Zamfirescu B.C,Gheţiu T. –*Tehnologia aplicatiilor multiagent. Mediul de dezvoltare JADE,* Editura Univ. Lucian Blaga Sibiu, 2009.

[25] Zamfirescu C.B., Filip F.G. – Supporting Self-Facilitation in Distributed Group Decisions. Proceedings 13th International Workshop on Database and Expert Systems Applications - *2-6 September, Aix-en-Provence, France, IEEE Computer Society Press, p. 321-325, DEXA 2002.*

ISBN: 978-973-595-874-9