UNIVERSIDADE FEDERAL DO RIO DE JANEIRO

ESCOLA POLITÉCNICA

DEPARTAMENTO DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO

ANÁLISE, *DESIGN* E IMPLEMENTAÇÃO DE SISTEMAS MULTIAGENTES

Autor: _____

Dermeval da Silva Júnior

Orientador: _____

Prof. Antônio Cláudio Gómez de Sousa

Examinador: _____

Prof. Sérgio Barbosa Villas-Boas

Examinador: _____

Prof. Sérgio Palma da Justa Medeiros

DEL
Dezembro de 2005

Ao meu filho amado,
Raphael,
à minha esposa,
Sanny Mendes,
à minha mãe,
Nilza Silva,
e ao meu saudoso pai,
Dermeval,
que sempre torceu por mim.

Agradecimentos

Ao meu orientador, Prof. Antônio Cláudio Gómez de Sousa, e ao Prof. Carlos José Ribas D'Avila, Coordenador do Curso de Engenharia Eletrônica, que acreditaram em mim.

Aos Professores Sérgio Barbosa Villas-Boas e Sérgio Palma da Justa Medeiros por aceitarem o convite para integrar a banca examinadora.

Aos amigos José Avelino Placca e Paulo Cesar Laureano Zenari pela força.

Leis da Robótica:

1. Um robô não deve machucar um ser humano, ou ainda, por sua não atuação, permitir que um ser humano seja machucado;

2. Um robô deve sempre obedecer às ordens dadas por seres humanos, a não ser que estas violem a primeira lei;

3. Um robô deve proteger sua própria integridade física, a não ser que para isso seja necessário violar a primeira e segunda lei.

(Isaac Asimov)

Resumo

O interesse pela tecnologia de agentes vem crescendo à medida que a humanidade vem demandando soluções mais dinâmicas para problemas cada vez mais complexos tratados no contexto computacional. A reboque dessa tendência, diversas ferramentas e metodologias que suportam o processo de Engenharia de *Software* Orientado a Agentes continuam sendo apresentadas. A proposta deste trabalho é investigar a metodologia Gaia e o *framework* JADE no âmbito do processo de desenvolvimento de Sistemas Multiagentes.

Palavras-chave: Inteligência Artificial Distribuída. Agente de Software. Sistemas Multiagentes. Metodologia Gaia. JADE.

Abstract

The interest about agent technology has been rising at the same time that mankind has been asking for more dynamic solutions for more complex questions at computational context. Along with this trend, a lot of tools and methodologies to Agent Oriented Software Engineering support have been presented every day. The present work proposes to investigate the Gaia methodology and the JADE framework at the scope of Multi-Agent Systems development process.

Keywords: Distributed Artificial Intelligence. Software Agent. Multi-Agent Systems. Gaia Methodology. JADE.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

ACC – *Agent Communication Channel*

ACL – *Agent Communication Language*

AMS – *Agent Management System*

AO – *Agent-Oriented*

AUML – *Agent Unified Modelling Language*

BDI – *Belief, Desire and Intention*

DF – *Directory Facilitator*

DPS – *Distributed Problem Solving*

FIPA – *Foundation for Intelligent Phisical Agents*

IA – Inteligência Artificial

IAD – Inteligência Artificial Distribuída

ICA – Núcleo de Pesquisa em Inteligência Computacional Aplicada da PUC-Rio

IEEE – *Institute of Eletrical and Eletronics Engineering*

JADE – *Java Agent DEvelopment framework*

JDK – *Java Development Kit*

JRE – *Java Runtime Enviroment*

KQML – *Knowledge Query and Manipulation Language*

KSE – *Knowledge Sharing Effort*

LGPL – *Lesser General Public License*

MESSAGE – *Methodology for Engineering Systems of Software Agents*

MTS – *Message Transport System*

OO – *Object-Oriented*

RMA – *Remote Management Agent*

RMI – *Remote Method Invocation*

SMA – Sistemas Multiagentes

TILAB – *Telecom Italy Lab*

Sumário

1. Introdução

1.1. Motivação

Ao passo da evolução humana, novas técnicas e ferramentas surgem no intuito de auxiliar a resolução de problemas complexos. Essas técnicas e ferramentas reduzem a complexidade do problema, aumentando seu grau de abstração e decompondo-o em problemas menores. De outra forma, permitem que os envolvidos com o processo de busca da solução abstraiam de detalhes importantes, porém conhecidos, do próprio processo em si, ou ainda, que a solução seja alcançada por partes, desenvolvidas de forma paralela ou seqüencial.

A abordagem de Sistemas Multiagentes (SMA) tem sido uma das técnicas adotadas para a resolução de alguns dos problemas complexos e distribuídos surgidos no mundo contemporâneo. O conceito principal desta abordagem é o agente, uma unidade autônoma de resolução de problemas (WOOLDRIDGE, 1995). A interação entre agentes viabiliza a construção de uma solução na qual cada um dos agentes resolve parte do problema principal. A abordagem SMA investiga o comportamento de um conjunto de agentes autônomos objetivando a solução de um problema que ultrapassa a capacidade individual de um agente (JENNINGS, 1996). A utilização de SMA permite resolver um problema complexo qualquer onde o algoritmo que o soluciona não é conhecido a priori.

No contexto da Engenharia de *Software* Orientada a Agentes, a disponibilidade de metodologias é um pré-requisito para o desenvolvimento de SMA em escala comercial. Muitas propostas estendem metodologias voltadas ao paradigma orientado a objetos ou baseadas em conhecimento, adicionando conceitos relativos a agentes, como, por exemplo: AUML, MESSAGE e OplA. Outras foram desenvolvidas diretamente a partir de conceitos advindos da teoria de agentes, como a metodologia Gaia. Ambas as categorias fornecem linguagens capazes de descrever as características internas de cada agente individualmente, como também, aspectos relativos ao grupo de agentes e suas relações sociais (MASSONET, 2002).

Existe, ainda, uma grande disponibilidade de ferramentas para implementação de aplicações baseadas em agentes em diversas linguagens de programação. Algumas fornecem, além de um *framework* de implementação, uma plataforma para execução de SMA. Outras provêem, adicionalmente, algum suporte à modelagem. Infelizmente, tais ferramentas diferem nos conceitos básicos suportados, o que dificulta a interoperabilidade entre elas. É possível

classificar tais ferramentas de implementação em duas principais categorias. Na primeira estão as ferramentas baseadas no tripé Crença-Desejo-Intenção, tais como: Jack e Zeus, fundamentadas diretamente na teoria dos agentes. A segunda engloba ferramentas como JADE e FIPA-OS, baseadas no mapeamento Agente-Classe, onde cada agente é implementado através de uma classe do paradigma OO.

1.2. Objetivo

Este trabalho tem como objetivo investigar o desenvolvimento de Sistemas Multiagentes, desde as fases de análise e *design*, utilizando metodologia Gaia, até a sua implementação através do *framework* JADE. A título de ilustração, é apresentado um estudo de caso baseado em uma versão modificada do problema dos "Cinco Filósofos Famintos", proposto, originalmente, por Dijkstra.

1.3. Estrutura do Trabalho

O Capítulo 2 apresenta os principais conceitos relacionados à Teoria de Agentes.

O Capítulo 3 faz um apanhado geral sobre as características dos Sistemas Multiagentes, suas vantagens, aplicações, desafios e, principalmente, quais os aspectos a considerar no desenvolvimento deste tipo de sistema.

O Capítulo 4 introduz a Engenharia de *Software* Orientada a Agentes e descreve a metodologia Gaia.

O Capítulo 5 apresenta o conceito de ontologia, o processo e os motivos para sua construção e introduz a ferramenta *Protégé*.

O Capítulo 6 apresenta conceitos relacionados à implementação de Sistemas Multiagentes e descreve o *framework* JADE.

O Capítulo 7 exibe um estudo de caso contemplando as fases de análise e *design*, construção da ontologia e implementação.

Já o Capítulo 8 apresenta a conclusão e sugestões para futuros trabalhos.

Finalmente, no Apêndice, as principais características técnicas do *framework* JADE são mostradas.

2. Teoria de Agentes

2.1. Introdução

Este capítulo pretende realizar um apanhado geral dos conceitos básicos relacionados à Teoria de Agentes.

2.2. Inteligência Artificial Distribuída (IAD)

O termo "Inteligência Artificial" tem sido motivo de muita confusão a respeito de seu significado. Da mesma forma que diamantes artificiais são, na realidade, falsos diamantes, há quem entenda que uma inteligência dita artificial não seja, verdadeiramente, real. Deste modo, formalmente, diz-se que Inteligência Artificial (IA) é, tão somente, o conjunto de técnicas computacionais para solução de problemas que utilizam processamento estatístico (Redes Bayesianas, por exemplo) ou fundamentam-se na busca de soluções através de uma base de conhecimentos (Sistemas Especialistas). É importante notar que tais técnicas não prevêem aprendizado, relegando ao computador a "simples" tarefa de processar cálculos estatísticos pesados ou buscar a resposta a um problema através de uma base de conhecimentos. O conhecimento é inserido, mas não inferido pelo artefato de *software*. No que se refere às demais técnicas, o termo "Inteligência Computacional", como defendem alguns autores, seria a designação mais correta. Segundo o ICA – Núcleo de Pesquisa em Inteligência Computacional Aplicada, ligado ao Departamento de Engenharia Elétrica da PUC-RJ (ICA, 2005):

> A Inteligência Computacional busca, através de técnicas inspiradas na natureza, o desenvolvimento de sistemas inteligentes que imitem aspectos do comportamento humano, tais como: aprendizado, percepção, raciocínio, evolução e adaptação.

Na Tabela 1, relacionam-se as diversas técnicas computacionais e suas respectivas inspirações na natureza.

Técnica Computacional	Inspiração na Natureza
Agentes Inteligentes	Agentes Humanos
Redes Neurais	Neurônios Biológicos
Computação Evolucionária	Evolução Biológica
Lógica Nebulosa	Processamento Lingüístico
Sistemas Especialistas	Processos de Inferência

Tabela 1 - Técnicas computacionais e suas respectivas inspirações na natureza (ICA, 2005).

A Inteligência Computacional é apenas uma área de pesquisa do universo científico da Computação Natural que envolve, também, Vida Artificial, Sistemas Dinâmicos Não-Lineares e outros. A Figura 1 exibe a hierarquia entre as diversas áreas de pesquisa relativas à Computação Natural.

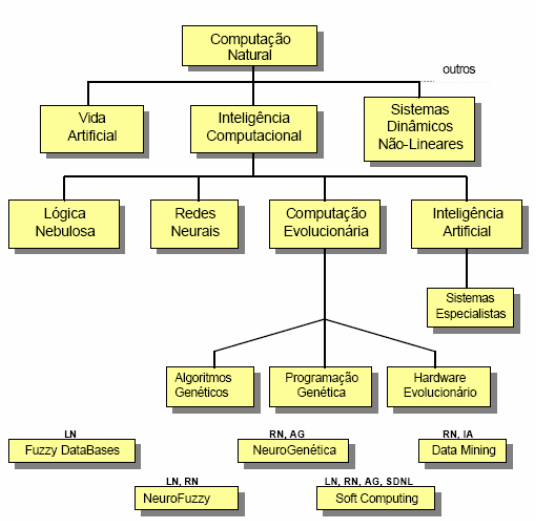

Figura 1 -- Taxonomia das áreas de pesquisa em Computação Natural (ICA, 2005).

Aparentemente, no âmbito da comunidade acadêmica envolvida nos estudos relacionados a agentes, não há nenhuma referência ao termo "Inteligência Computacional", como seria esperado. Ao invés disso, os autores pesquisados referem-se à abordagem de Sistemas Multiagentes como uma sub-área da Inteligência Artificial Distribuída (IAD) que, por seu turno, seria, ela mesma, uma sub-área da Inteligência Artificial.

De modo a manter a consonância com a literatura pesquisada, este trabalho manterá a referência à IAD.

JENNINGS (1996) apud JUCHEM (2001) afirma que:

> O objeto de investigação da IAD são os modelos de conhecimento e as técnicas de comunicação e raciocínio necessárias para que agentes computacionais convivam em sociedades compostas de computadores e pessoas.

A IAD engloba duas principais linhas de pesquisa (JUCHEM, 2001):

- Resolução Distribuída de Problemas (DPS[1]) – particiona a solução de um problema específico entre um número de módulos que cooperam entre si. O processo de coordenação é definido em tempo de projeto. Apresenta uma visão *top-down*.
- Sistemas Multiagentes (SMA) – estuda o comportamento de um conjunto de agentes autônomos cujo objetivo comum é a solução de um problema. Cada agente possui, adicionalmente, capacidades e objetivos individuais. Uma vez agrupados em sociedade, colaboram e interagem no intuito de alcançar o objetivo do sistema. Decisões sobre ações e coordenação destas são de responsabilidade dos próprios agentes.

2.3. Agentes

2.3.1. Conceito

O dicionário (FERREIRA, 2004) define agente, em sua primeira acepção para o termo, como aquele "que opera, agencia, age". Mais adiante, em outra acepção, agente é "quem trata de negócios por conta alheia". Diversas definições sobre o conceito de agente têm sido propostas ao longo do tempo por vários estudiosos no assunto. No entanto, nenhuma das definições propostas é universalmente aceita pela comunidade. WOOLDRIDGE (1995) *apud* FRANKLIN (1996) conceitua agente como um elemento de *hardware* ou *software* que possui autonomia, habilidade social, reatividade e pró-atividade. Na mesma linha de raciocínio, BELLIFEMINE (2003), define agente como "um componente de *software* autônomo, pró-ativo e social". Já a FIPA[2] (2005) define agente (físico ou virtual) como "uma entidade que reside em um ambiente onde interpreta dados [através de sensores] que refletem eventos no ambiente e executam ações que produzem efeitos no ambiente [através de efetuadores]". A Figura 2 mostra um modelo de interação entre o agente e o ambiente.

[1] Do inglês, *Distributed Problem Solving*.
[2] *Foundation for Intelligent Physical Agents.*

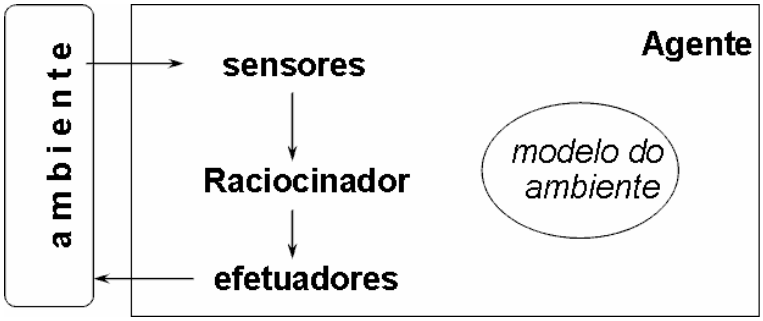

Figura 2 - Modelo de Interação Agente e Ambiente (PLACCA, 2005).

PINTO (2005) define que:

> Um agente é, genericamente, uma unidade computacional independente, com algum grau de inteligência representada pelos estados mentais internos, que possui a capacidade de executar ações conjuntas com outras unidades, numa intenção mútua de atingir um objetivo global em determinado momento.

Para ELAMY (2005) agente é:

> Uma entidade de *software* que processa propriedades inteligentes especiais, como autonomia, raciocínio, mobilidade, sociabilidade, habilidade para aprender, cooperação e negociação. Tais propriedades inteligentes permitem aos agentes iniciarem suas ações sem a direção ou intervenção direta de humanos ou outras entidades, e a interativamente cooperarem e comunicarem-se um com os outros e com o ambiente para efetuar tarefas especiais que não podem ser executadas por *software* convencional.

2.3.2. Agentes e Objetos

Através da analise da história da evolução das linguagens de programação torna-se mais fácil entender as diferenças entre os conceitos de agentes e objetos. Originalmente, a unidade básica de *software* era o programa por completo, controlado inteiramente pelo programador. O estado do programa era de responsabilidade do programador e sua invocação determinada pelo sistema operacional. Nesta fase, o termo modular não era aplicável, pois o comportamento (provido pelo programa) não podia ser invocado como uma unidade de reutilização (ODELL, 2002).

Com o aumento na complexidade dos programas e das capacidades de memória, houve necessidade dos programadores inserirem certo grau de organização em seus códigos.

Surge a Programação Modular. Pequenos trechos de código, totalmente funcionais, poderiam ser inseridos e reutilizados em uma variedade de situações. Laços estruturados e códigos de sub-rotinas eram desenhados para apresentar alto grau de integridade local. Em função do encapsulamento do código da sub-rotina, seu estado, agora, era determinado, externamente, através dos argumentos passados a ela. A invocação ocorria, também, externamente via operação *CALL*. O procedimento (*procedure*) era a unidade primária de decomposição (ODEL, 2002).

A orientação a objetos determinou a internalização do controle do estado pelos objetos. Agora, a manipulação do estado interno poderia ser realizada somente através dos métodos fornecidos pelo próprio objeto. No entanto, a invocação dos métodos continuava sendo feita externamente através de mensagens enviadas por alguma entidade (ODELL, 2002).

Agentes de *software*, diferentemente, são autônomos. Internalizando, não apenas código e estado, mas também o próprio processo de invocação. Cada agente pode ter regras e objetivos individuais, fazendo-os parecer como "objetos ativos com iniciativa". Como e quando os agentes agem é determinado pelos próprios agentes. Em função de sua autonomia, agentes podem iniciar interações e responder a mensagens da maneira que eles escolherem. Em resumo, agentes podem ser idealizados como objetos que têm capacidade de dizer "Não" (ODELL, 2002).

Segundo PARUNAK (1997) *apud* ODELL (2002):

> O desenvolvedor simplesmente identifica os agentes desejados na aplicação final e os agentes se auto-organizam para executar a funcionalidade determinada.

Agentes são, então, objetos com atitude. Objetos com algum comportamento adicional, como: mobilidade, inferência, etc. Por outro ângulo, objetos são agentes sem estes atributos extras. Algo como agentes degenerados. É possível, então, distinguir artefatos que se comunicam nativamente como orientados a agentes[1] e objetos que encapsulam e simulam capacidades nativas de agentes como baseados em agentes[2].

[1] *Agent-oriented*.
[2] *Agent-based*.

	Programação Monolítica	Programação Modular	Programação Orientada a Objetos	Agentes
Unidade de Comportamento	Não modular	Modular	Modular	Modular
Unidade de Estado	Externa	Externa	Interna	Interna
Invocação	Externa	Externa (via *CALL*)	Externa (via mensagem)	Interna (regras, objetivos)

Tabela 2 - Abordagem da evolução das linguagens de programação (PARUNAK, 1997 apud ODELL, 2002).

2.3.3. Propriedades

Autonomia

Capacidade de executar suas tarefas e iniciar suas ações sem intervenção externa, ou seja, independente de qualquer outra entidade.

Sociabilidade

Habilidade de utilizar um mecanismo específico de comunicação com o objetivo de interagir com outros agentes ou com o ambiente.

Reatividade

Capacidade de perceber e reagir a mudanças no ambiente. Um agente deve ser capaz de adaptar-se a mudanças ambientais com o intuito de levar adiante a funcionalidade pelo qual foi projetado.

Pró-atividade

Habilidade de tomar iniciativa em função de circunstâncias e comportar-se orientado a objetivos.

Continuidade Temporal

Um agente deve funcionar de modo contínuo e nunca cessar suas operações, exceto quando se tornar temporariamente suspenso no intuito de alterar seu estado por razões aceitáveis, como mobilidade.

Capacidade de Aprendizado

Capacidade de aprender, de forma dinâmica, a partir de informações colhidas junto ao ambiente a sua volta ou em função da comunicação com outros agentes. Tais informações devem servir de base para a atualização da sua base de conhecimentos.

Em RUSSEL (1995):

> Sempre que o *designer* tiver um conhecimento incompleto do ambiente no qual o agente irá atuar, o aprendizado é o único caminho que o agente possui para adquirir as informações que necessita saber.

Raciocínio

Trata-se do mecanismo para tomada de decisão com o qual o agente decide agir com base nas informações recebidas e de acordo com seus próprios objetivos e metas.

Racionalidade

Suposição de que um agente somente agirá para cumprir seus objetivos e não o contrário, na medida em que suas crenças assim o permitam (GOMES, 2000).

Veracidade

Suposição de que um agente não comunica, voluntariamente, nenhuma informação falsa (GOMES, 2000).

Mobilidade

Habilidade para migrar, sob certas circunstâncias, de uma máquina para outra em um ambiente de rede heterogêneo a fim de processar suas tarefas localmente naquela máquina (ELAMY, 2005).

Cooperação

Envolve comunicação e interação entre agentes no intuito de alcançar objetivos comuns. LUCK (2001) *apud* ELAMY(2005) afirma que:

> O termo cooperação pode ser utilizado somente quando os envolvidos são autônomos e, ao menos potencialmente, capazes de resistência. Se não são

> autônomos, nem capazes de resistência, então um, simplesmente, engaja-se ao outro.

Alguns autores defendem ainda a inclusão de propriedades relacionadas ao comportamento humano como: **conhecimento**, **crença**, **intenção** e **obrigação**. Conforme SILVA (2003) explica, não há, na realidade, uma unanimidade quanto à importância de cada uma dessas e outras propriedades ou mesmo sua obrigatoriedade na caracterização de agentes. Consensualmente, obtemos apenas a idéia de que um subconjunto qualquer dessas características torna um agente muito diferente de simples programas e objetos.

2.3.4. Categorias

Em função do nível de capacidade de resolução de problemas.

Reativo

Reage a alterações no ambiente ou a mensagens de outros agentes. Não possue capacidade de raciocínio sobre suas intenções, reagindo ao sabor de regras e planos estereotipados. Suas ações se limitam a atualizar a base de conhecimentos e a enviar mensagens para outros agentes ou para o ambiente (JUCHEM, 2001a).

Intencional ou Racional ou Cognitivo ou Deliberativo

Possui a habilidade de raciocínio sobre suas intenções e crenças e sobre criar e executar planos de ações. Considerado como sistema de planejamento, seleciona objetivos – de acordo com suas motivações (JUCEM, 2001a).

Social

Um agente intencional é considerado social quando possui uma representação de outros agentes internamente, a fim de tomar decisões e criar planos (JUCHEM, 2001a).

A Figura 3 apresenta uma representação gráfica das categorias de agentes.

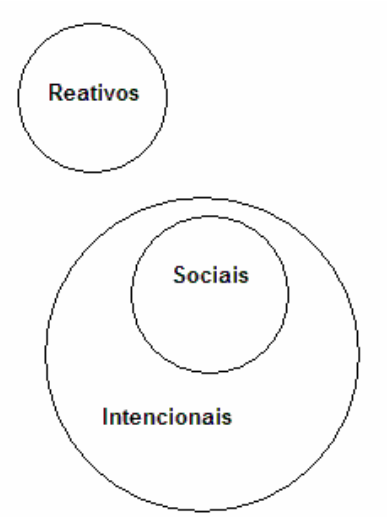

Figura 3 - Categorias de agentes (JUCHEM, 2001a).

2.3.5. Arquiteturas

Uma arquitetura (concreta) de agentes é um mapa de seus detalhes internos, suas estruturas de dados, as operações que pode realizar sobre estas estruturas e o fluxo de controle entre as estruturas. Ou, simplesmente, é a forma de estruturação do agente. No nível mais abstrato são classificadas em dois tipos básicos: cognitivas e reativas.

Cognitiva ou Deliberativa

Possuem uma representação simbólica do mundo. Suas decisões são feitas por meio de um processo baseado em raciocínio lógico (JUCHEM, 2001a).

Processo de deliberação ou decisão:

1. Construção da representação simbólica do mundo (ambiente a sua volta);
2. Utilização de um plano de ação;
3. Utilização de uma função utilidade para a ação.

As arquiteturas de agentes cognitivos são classificadas como funcionais ou baseadas em estados mentais.

Funcional

Tipo de arquitetura cognitiva. Agente é composto por módulos que representam cada uma das funcionalidades necessárias para sua operação (JUCHEM, 2001a).

Conforme STEINER (1996) *apud* JUCHEM (2001a), tal arquitetura é composta por três partes principais:

- Cabeça – controla as ações, portanto engloba as capacidades reativas, racionais e cooperativas;
- Comunicador – implementa as capacidades de comunicação;
- Corpo – encarregado da execução das ações e observação do ambiente.

Baseada em Estados Mentais

Tipo de arquitetura cognitiva. Abordagem de visão psicológica. O estado de um agente é definido com um conjunto de estados mentais, tais como: crenças, capacidades, escolhas e compromissos. A gerência de sua autonomia é baseada, geralmente, nos próprios estados mentais. Em função destes estados mentais é decidida a ação seguinte a executar no intuito de satisfazer algum de seus objetivos, quando e para quem solicitar colaboração, que mudanças ambientais determinam alterações em suas crenças ou objetivos e assim por diante (JUCHEM, 2001a).

***Belief, Desire and Intention* - BDI**

Particularização da arquitetura baseada em estados mentais. Considera apenas três estados: crença, desejo e intenção. Onde o conjunto de crenças é a representação do mundo mantida pelo agente. Intenções são desejos escolhidos para execução. Alterações percebidas pelo agente podem significar modificações em suas crenças e, possivelmente, mudanças em suas intenções, visto que seus desejos podem haver se modificado em função do novo conjunto de crenças (JUCHEM, 2001a). A Figura 4 mostra um diagrama esquemático de uma arquitetura BDI genérica.

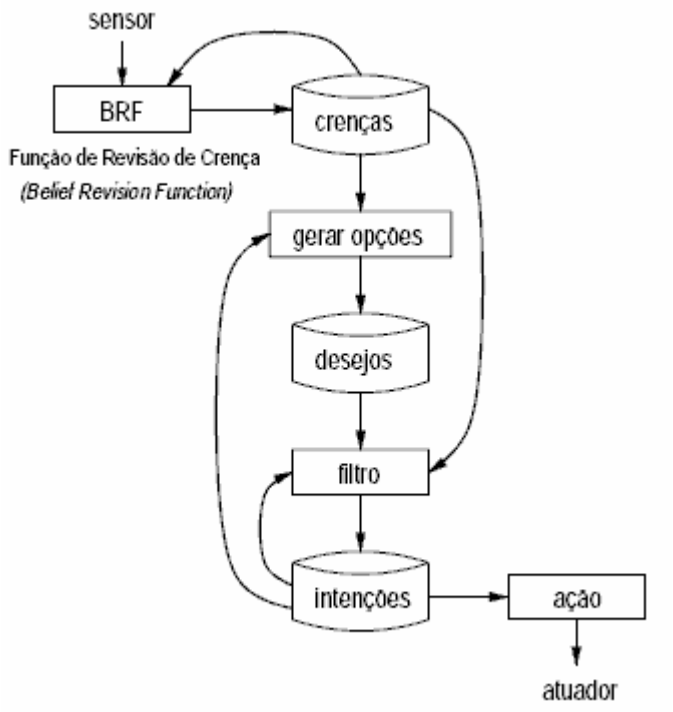

Figura 4 - Diagrama esquemático de uma arquitetura BDI genérica (GOMES, 2000).

Reativa ou Não-Deliberativa

O processo de tomada de decisão de um agente ocorre em tempo real, em resposta a estímulos do ambiente, captados por seus sensores, ou a mensagens enviados por outro agente. O mecanismo de controle é, geralmente, implementado por um conjunto de regras evento-ação (estímulo-resposta) ou por máquinas de estados finitos (autômatos finitos). Não guarda qualquer representação simbólica do mundo e também não utiliza raciocínio simbólico complexo (JUCHEM, 2001a).

Híbridas

São provenientes das deficiências encontradas nas arquiteturas deliberativas e reativas, reunindo propriedades de ambas. Dotadas de capacidade de modificação de seus planos de ação no momento em que a situação passa a divergir dos objetivos iniciais e facilidade em lidar com situações imprevistas que exigem decisões rápidas (JUCHEM, 2001a).

2.3.6. Ambiente

Todo agente está situado em algum ambiente. Entende-se por ambiente tudo que está no entorno do agente. Através do ambiente ocorre a dispersão do controle, dos dados e do conhecimento pela comunidade de agentes.

JUCHEM (2001a) define que:

> O tipo de ambiente no qual o agente está situado pode determinar a maneira pela qual o agente deve visualizar o mundo, ou seja, determinar sobre qual tipo de representação do mundo que o agente deverá trabalhar, bem como sua maneira de atuar e de proceder as alterações no ambiente.

O agente deve focar apenas nas informações pertinentes à sua funcionalidade e aos seus objetivos. Outras informações devem ser descartadas pelos sensores inteligentes.

Propriedades de Ambientes

Acessível × Inacessível

Determina se o aparato sensorial do agente lhe fornece um estado completo do ambiente. Um ambiente é considerado efetivamente acessível se todas as informações pertinentes a escolha de ação são detectáveis pelos sensores. Caso o ambiente seja acessível não há necessidade de o agente manter uma representação interna do mundo.

Determinístico × Não-Determinístico

Um ambiente determinístico é aquele que o estado do ambiente é determinado somente pelo seu estado atual e pela atuação dos agentes nele inseridos.

Episódico × Não-Episódico

Em ambientes episódicos a experiência do agente é dividida em episódios. Cada qual consiste em percepções e ações dos agentes, e a qualidade de cada ação depende somente o episódio em si.

Estático × Dinâmico

Um ambiente dinâmico é aquele que pode sofrer alterações durante a fase de deliberação do agente. Ambiente semi-estáticos são modificados apenas em virtude da atuação de agentes e não temporalmente.

Discreto × Contínuo

Ambiente discreto é aquele que apresenta um número limitado de percepções e ações distintas e claramente definidas.

2.4. Resumo

Neste capítulo vimos:

- A forma de inserção da Teoria de Agentes no contexto da Inteligência Artificial Distribuída e uma discussão teórica sobre a adequação do termo.
- A diferenciação entre objetos e agentes.
- As principais propriedades de agentes.
- As categorias de classificação de agentes.
- Os principais tipos de arquitetura de agentes.
- A descrição de ambiente e suas principais propriedades.

3. Sistemas Multiagentes (SMA)

3.1. Introdução

Neste capítulo serão apresentadas as principais características dos Sistemas Multiagentes, as vantagens dessa abordagem de desenvolvimento de sistemas, suas aplicações e desafios.

3.2. Características

Um SMA pode ser visualizado como um sistema distribuído que incorpora, de modo coerente, diversos solucionadores de problemas (agentes) discretos e independentes que interagem em função de um objetivo comum e cuja capacidade total ultrapassa a simples soma das capacidades individuais. Um SMA pode ser formado por agentes homogêneos ou heterogêneos, colaborativos ou competitivos, cognitivos ou reativos e etc. Seus agentes podem possuir os mesmos objetivos individuais ou não. De forma geral, o projeto dos agentes dependerá da aplicabilidade do sistema.

Segundo GOMES (2000), um ambiente multiagente possui as seguintes características:

- Fornece uma infra-estrutura especificando protocolos de comunicação e interação;
- É, tipicamente, aberto e não possui um criador centralizado;
- Contém agentes que são autônomos e distribuídos, podendo cada um deles apresentar interesse próprio ou agir de forma cooperativa.

3.3. Vantagens

As principais vantagens relacionadas à utilização da ambientes multiagentes são (ELAMY, 2005):

- **Confiabilidade** – os SMA apresentam níveis de tolerância à falha e robustez maiores do que os sistemas convencionais;
- **Modularidade e Escalabilidade** – agentes podem ser adicionados ou retirados de um SMA sem corromper, ou mesmo interromper, o sistema;
- **Adaptabilidade** – agentes possuem forte capacidade adaptativa a novas situações e falhas;

- **Concorrência** – agentes têm capacidade de raciocínio e executam tarefas em paralelo proporcionando maior flexibilidade e velocidade computacional;
- **Dinamismo** – agentes podem colaborar de forma dinâmica a fim de compartilhar recursos e solucionar problemas.

No mundo real, um cenário típico é representado por uma comunidade de entidades discretas com objetivos distintos, comunicando-se e, geralmente, cooperando e/ou negociando no intuito de alcançar benefícios e objetivos comuns. Tal cenário pode ser facilmente modelado através agentes de *software* organizados em um Sistema Multiagentes. Soluções tradicionais para o cenário descrito podem apresentar custos elevados e impraticáveis.

A Figura 5 apresenta uma visão de um Sistema Multiagente.

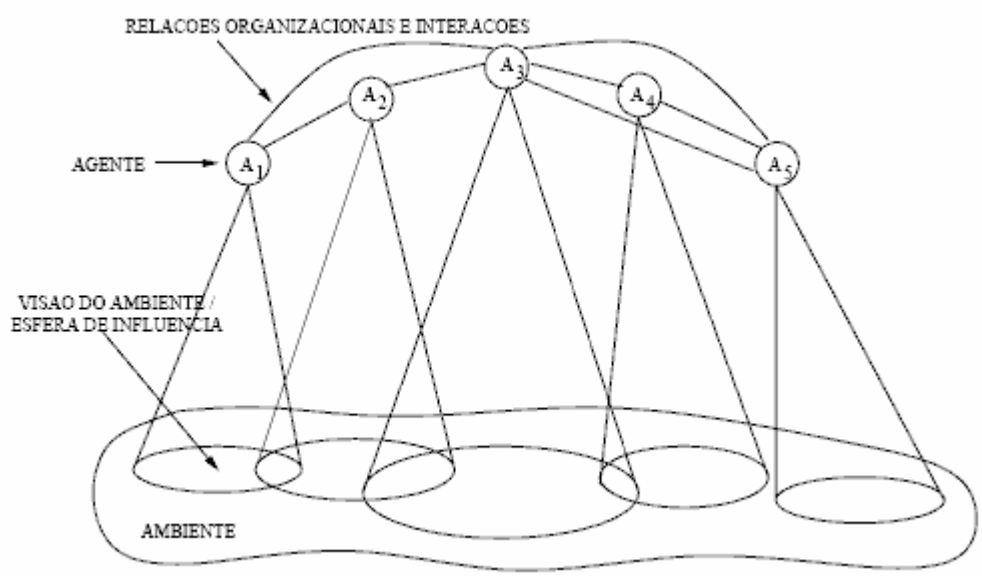

Figura 5 - Visão de um Sistema Multiagente (JUCHEM, 2001).

3.4. Aplicações

SMA são bastante apropriados em situações que envolvem arquiteturas complexas, numerosos componentes distribuídos remotamente, diferentes *expertises* e interesses conflitantes. A abordagem SMA apresenta maior valor agregado em problemas inerentemente distribuídos. No entanto, o aspecto distributivo não é um fator suficiente para utilizar SMA, requisitos como inteligência e adaptabilidade em sub-processos que envolvam raciocínio explícito sobre o comportamento devem ser observados.

De modo geral, utiliza-se SMA ao lidar com situações onde diversos clientes e organizações com objetivos potencialmente conflitantes e informações distribuídas precisam

interagir uns com os outros visando atingir seus objetivos e executar suas tarefas (ELAMY, 2005).

Situações que qualificam a utilização da abordagem SMA são:

- Aplicações que necessitem da interconexão e interoperação entre diversos sistemas autônomos e com interesses próprios;
- Problemas que envolvam fontes de informações autônomas e distribuídas;
- Problemas que requeiram soluções provenientes de diferentes *expertises* inerentemente distribuídas;
- Problemas que ultrapassam as fronteiras organizacionais nos quais um entendimento das interações entre sociedades e organizações é necessário;
- Problemas caracterizados pela ausência de informações globais sobre todos os agentes, mas que apresentam uma quantidade razoável de agentes com visão local.

3.5. Desafios

Para SYCARA (1998) *apud* ELAMY (2005), alguns dos desafios oriundos do desenvolvimento de SMA são:

- Formular, descrever, decompor problemas e alocar tarefas a agentes individuais;
- Fazer com que agentes heterogêneos sejam capazes de se comunicarem e interagirem;
- Fazer com que agentes individuais ajam coerentemente ao realizar ações e tomar decisões;
- Fazer com que agentes individuais reajam e raciocinem a respeito de ações, planos e conhecimentos oriundos de outros agentes;
- Reconhecer e reconciliar intenções e objetivos conflitantes entre agentes coordenadores;
- Fabricar e projetar sistemas de IAD de modo a empregar metodologias de SMA;
- Resolver questões de comunicação.

3.6. Aspectos a Considerar no Desenvolvimento de um SMA

Uma empresa é uma sociedade composta por empregados (agentes humanos) que unem esforços no intuito de alcançar os objetivos globais da organização. Trata-se de um exemplo de Sistema Multiagente. Certo grau de estruturação e planejamento é necessário para

que os interesses individuais de cada agente se alinhem aos interesses institucionais (da sociedade). Em suma, em um SMA é preciso que sejam definidos critérios que viabilizem e garantam a coerência das ações de cada agente com vistas a alcançar, de modo efetivo, os objetivos globais do sistema (JUCHEM, 2001).

A maior parte das atividades de pesquisa em SMA segue duas perspectivas:

- **Perspectivas do agente** – enfocam características do agente como indivíduo (já abordado no Capítulo 2);
- **Perspectivas de grupo** – reúne aspectos sociais, tais como: organização, coordenação, cooperação, negociação, comportamento coerente, planejamento, comunicação e interação.

No contexto das perspectivas de grupo é possível definir três grandes grupos de aspectos a serem considerados no projeto de um SMA: aspectos fundamentais, arquiteturais e ambientais.

3.6.1. Aspectos Fundamentais

Definem as características que devem ser viabilizadas para a garantia da compatibilidade entre as ações dos agentes que constituem o SMA (JUCHEM, 2001).

Estrutura

Padrão de relações de informação e controle entre agentes, bem como a distribuição das habilidades entre eles. Fornece uma visualização de como os problemas são resolvidos pelo grupo e o papel que cada agente desempenha dentro desta estrutura (JUCHEM, 2001).

Organização

Refere-se ao conjunto de compromissos globais, crenças mútuas e intenções comuns aos agentes quando agem juntos para atingir certo objetivo (MOULIN, 1996 *apud* JUCHEM, 2001).

Coordenação

Capacidade de agentes autônomos trabalharem em conjunto e combinar seus objetivos de maneira a alcançarem o objetivo final do sistema (SILVA, 2003). A coordenação pode ser dividida em negociação e cooperação.

A **negociação** é a coordenação entre agentes antagônicos ou, simplesmente, egoístas (WEISS, 1999 *apud* SILVA, 2003). Está relacionada diretamente a agentes competitivos. De

forma geral, são utilizados protocolos para determinar regras de negociação e são definidos atributos sobre os quais se pretende obter um acordo.

Em oposição à negociação, a coordenação por **cooperação** ocorre entre agentes não antagônicos, ou seja, que possuem objetivos individuais comuns. Neste caso, a coordenação visa conjugar os esforços a fim de promover um trabalho de equipe eficaz e eficiente.

Comunicação

Imprescindível para permitir a colaboração, negociação e cooperação entre agentes. Pode-se dividir em quatro, os tipos de comunicações possíveis: direta, assistida, por difusão ou, ainda, quadro-negro. A **comunicação direta** ocorre quando dois agentes se comunicam diretamente. Nesse caso, é necessário que cada agente tenha conhecimento da existência dos demais agentes do sistema. Na **comunicação assistida** uma estrutura hierárquica de agentes é definida e a troca de mensagens se dá através de agentes especiais designados facilitadores ou mediadores (SILVA, 2003). A **comunicação por difusão de mensagens ou *broadcast*** caracteriza-se pelo envio da mensagem a todos os agentes que compõem o ambiente. Finalmente, a **comunicação por quadro-negro ou *blackboard*** utiliza-se de um repositório para onde agentes remetem e/ou recebem mensagens.

Interação

Segundo SILVA (2003), a linguagem e sua expressividade definem a capacidade de comunicação de cada agente e, portanto, deve ser universal e partilhada por todos, ser concisa e ter um número limitado de primitivas.

A **KQML ou *Knowledge Query and Manipulation Language*** "é uma linguagem e um protocolo de comunicação de alto nível para troca de mensagens independente de conteúdo e da ontologia aplicável" (SILVA, 2003). Foi desenvolvida pelo KSE – *Knowledge Sharing Effort*. Apresenta problemas como ambigüidade e termos vagos, performativas com nomes inadequados e falta de performativas.

A **FIPA-ACL** é uma linguagem, tal qual o KQML, baseada em ações de fala. Sintaticamente é extremamente semelhante ao KQML, mas possui um conjunto de performativas diferenciado. Baseia-se em um conjunto de tipos de mensagens e seus efeitos sobre os agentes remetentes e receptores. Além disso, sua semântica é descrita claramente através de uma linguagem de descrição específica. A Figura 6 apresenta o formato de uma mensagem no padrão FIPA-ACL.

Figura 6 - Formato da Mensagem FIPA-ACL (SILVA, 2003).

3.6.2. Aspectos Arquiteturais

Definem as características que devem ser providas pela arquitetura a ser adotada para viabilização dos aspectos fundamentais dentro do SMA (JUCHEM, 2001).

Uma taxonomia pode ser utilizada para classificar as possíveis formas de dispor os relacionamentos existentes entre os agentes:

Hierárquica

Dispõem os agentes em uma estrutura hierárquica na qual a comunicação ocorre de maneira também hierárquica. Cada agente pode comunicar-se apenas com os agentes por ele supervisionados ou por seu próprio supervisor. Este tipo de estrutura dispensa mecanismos e localização de agentes e reduz, significativamente, a quantidade de comunicação no sistema. No entanto, impede que haja uma organização dinâmica dos agentes (JUCHEM, 2001).

Nivelada

Todos os agentes estão no mesmo nível hierárquico, ou seja, cada agente pode contatar diretamente qualquer dos outros agentes.

Agentes Compostos por Agentes

"Pressupõe a existência de alguns agentes que são componentes de outros agentes" (JUCHEM, 2001).

SMA Compostos por SMAs

Um subconjunto de agentes pertencentes a um determinado SMA formam, por si só, um SMA.

3.6.3. Aspectos Ambientais

Definem as características ambientais nos quais os agentes estão inseridos a fim de que seja possível determinar as técnicas de percepção que devem ser utilizadas pelos agentes. As características do ambiente já foram abordadas no Capítulo 2.

3.7. Resumo

Este capítulo abordou:

- As principais características dos Sistemas Multiagentes;
- As vantagens na utilização deste tipo de abordagem;
- Os tipos de problemas indicados para sua aplicação;
- Os desafios advindos da opção pela abordagem SMA;
- E alguns dos aspectos a considerar no desenvolvimento deste tipo de sistema.

4. Engenharia de *Software* Orientada a Agentes

4.1. Introdução

No contexto da Engenharia de *Software* (PRESSMAN, 1995 *apud* JUCHEM, 2001) afirma que:

> A construção de *software* de alta qualidade de maneira produtiva é viabilizada por um conjunto de métodos, ferramentas e procedimentos. O caminho para evolução no desenvolvimento de *software* passa por uma combinação de métodos abrangentes para todas as fases de desenvolvimento [...], melhores ferramentas para automatizar esses métodos, blocos de construção mais poderosos para a implementação [...], melhores técnicas para a garantia da qualidade do *software* e uma filosofia de coordenação predominante, controle e administração.

A proposta deste capítulo é apresentar ao leitor a Metologia Gaia, uma das principais propostas para modelagem de Sistemas Multiagentes.

4.2. Características de Sistemas Complexos

Problemas encontrados em aplicações industriais, por exemplo, são, geralmente, complexos por natureza. Tais problemas apresentam características que os diferenciam dos problemas comuns tratados por sistemas tradicionais. Dentre essas características podemos listar, entre outras:

- Complexidade hierárquica, onde os sistemas são compostos por sub-sistemas que se inter-relacionam;
- Distribuição espacial das informações;
- Ausência ou número reduzido de informações globais;
- Interesses individuais dos sub-sistemas muitas vezes conflitantes entre si;
- Distribuição espacial de *expertises*;
- Existência de informações que extrapolam as fronteiras organizacionais.

As principais ferramentas existentes para mitigar tal complexidade são:

- Decomposição – técnica básica para resolução de grandes problemas – associada ao jargão "dividir para conquistar" – divisão do problema em problemas menores e, potencialmente, mais fáceis de lidar;
- Abstração – técnica que visa desconsiderar detalhes e propriedades pouco relevantes ao escopo do problema em questão – a idéia é possibilitar a geração de modelos simplificados da realidade;
- Organização – gerência dos inter-relacionamentos entre os componentes de resolução do problema.

A Figura 7 apresenta a visão típica de um sistema complexo.

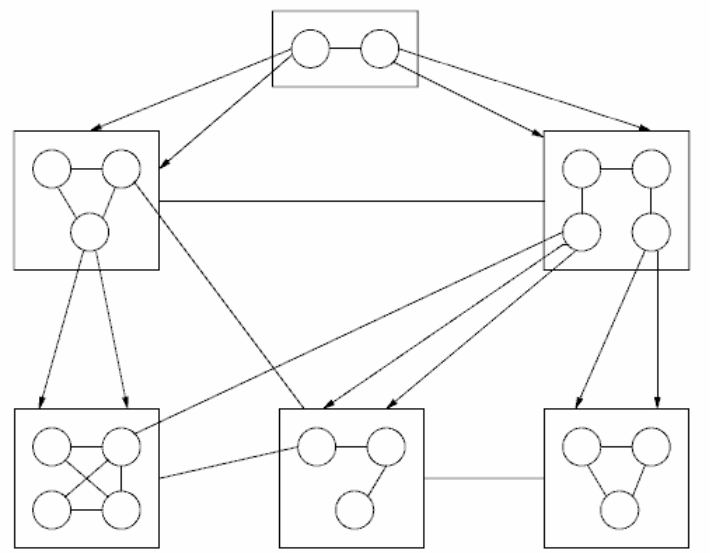

Figura 7 - Visão de um Sistema Complexo (JUCHEM, 2001).

4.3. Software Orientado a Agentes

Ao adotar uma visão de mundo orientada a agentes, é fácil perceber que um único agente é insuficiente para solucionar a grande maioria dos problemas. O envolvimento de múltiplos agentes interativos e autônomos no processo de resolução é imprescindível e permite modelar características inerentes aos problemas complexos. Desta forma, a aplicação da abordagem orientada a agentes significa decompor um problema complexo em múltiplos componentes autônomos com objetivos individuais e que se inter-relacionam.

Neste contexto, as técnicas que adotam uma abordagem orientada a agentes se apresentam bem adaptadas ao desenvolvimento de sistemas complexos. Essa adaptação pode ser comprovada em função das seguintes constatações listadas por JUCHEM (2001a):

- As decomposições da orientação a agentes são um caminho real para particionar a problemática de um sistema complexo;
- As abstrações da orientação a agentes são uma abordagem natural para modelar sistemas complexos;
- A filosofia orientada a agentes para identificar e gerenciar relacionamentos organizacionais é apropriada para a representação das dependências e interações que existem em um sistema complexo.

4.4. Propostas para Modelagem de SMA

As pesquisas relacionadas à construção de uma metodologia para modelagem de SMA têm evoluído em dois principais ramos. A primeira categoria envolve a adaptação de metodologias de Engenharia de *Software* já existentes ao paradigma de agentes. Como exemplo de trabalhos desenvolvidos nesta linha de pesquisa é possível citar:

- *Agent Unified Modelling Language* (AUML), cuja proposta é estender a linguagem de modelagem UML a fim de englobar conceitos relacionados a agentes;
- *Methodology for Engineering Systems of Software Agents* (MESSAGE) – outra metodologia baseada na utilização da linguagem UML. A metodologia inspira-se no estudo de organizações e sociedades humanas como meio para descrever as relações sociais dos agentes e nas pesquisas de Inteligência Artificial e psicologia cognitiva para descrever o comportamento dos agentes como indivíduos.

Já a segunda categoria busca desenvolver uma metodologia diretamente a partir da Teoria de Agentes. A idéia é cobrir, principalmente, as fases de análise e *design* do processo de desenvolvimento de sistemas baseados em agentes. Tipicamente, estas metodologias, propõem uma coleção de modelos para ambas as fases de desenvolvimento. Como exemplo de metodologias desta categoria é possível listar:

- Gaia – proposta originalmente por Wooldridge e outros, engloba dois modelos de análise e três na fase de *design*. Esta metodologia será abordada mais detalhadamente na próxima seção.
- MAS-CommonKADS – trata-se, na realidade, de uma extensão multiagente da principal metodologia estruturada de suporte à engenharia do conhecimento, chamada CommonKADS.

4.5. Metodologia Gaia

A metodologia Gaia é uma metodologia completa e genérica construída, especificamente, para análise e *design* de Sistemas Multiagentes. A metodologia permite tanto a modelagem da estrutura individual de cada agente quanto das relações sociais entre agentes. Trata-se de um avanço em relação às metodologias já existentes, pois mantém neutralidade em relação às linguagens de modelagem e à plataforma de implementação.

Gaia permite que o analista caminhe progressiva e sistematicamente do levantamento de requisitos do sistema para a construção dos modelos de *design*. Tais modelos mostram-se suficientemente detalhados a ponto de permitir a implementação direta dos artefatos de *software*. Ao utilizar a metodologia, o analista caminha de um nível maior para um nível menor de abstração adicionando conceitos concretos (WOOLDRIDGE, 2000).

Segundo o mesmo WOOLDRIDGE (2000):

> Gaia utilizou-se de algumas terminologias e notações oriundas da Análise Orientada a Objetos. No entanto, não é simplesmente uma tentativa de aplicar tais métodos ao desenvolvimento orientado a agentes. Ao invés disso, esta fornece um conjunto de conceitos específicos a agentes através do qual um engenheiro de *software* pode entender e modelar um sistema complexo. Em particular, Gaia encoraja o desenvolvedor a pensar na construção de sistemas baseados em agentes como um processo de *design* organizacional.

De acordo com a Gaia, um SMA é visualizado como sendo composto por um número de agentes autônomos e interativos que vivem em uma sociedade organizada na qual cada indivíduo (agente) desempenha um ou mais papéis.

Os principais conceitos definidos pela metodologia estão dispostos em duas categorias e podem ser visualizados na Tabela 3. Entidades abstratas são aquelas usadas durante a fase de análise para conceitualizar o sistema, mas que não precisam apresentar, obrigatoriamente, qualquer implementação (realização) no contexto do sistema. Por sua vez, entidades concretas são usadas no processo de *design* e apresentarão, tipicamente, contrapartidas diretas no sistema.

A Figura 8 apresenta o relacionamento entre os modelos definidos pela metodologia.

Conceitos Abstratos	Conceitos Concretos
Papéis	Tipos de Agentes
Permissões	Serviços
Responsabilidades	Conhecimentos
Protocolos	
Atividades	
Propriedades de Sobrevivência[1]	
Propriedades de Segurança[2]	

Tabela 3 - Conceitos concretos e abstratos em Gaia (WOOLDRIDGE, 2000).

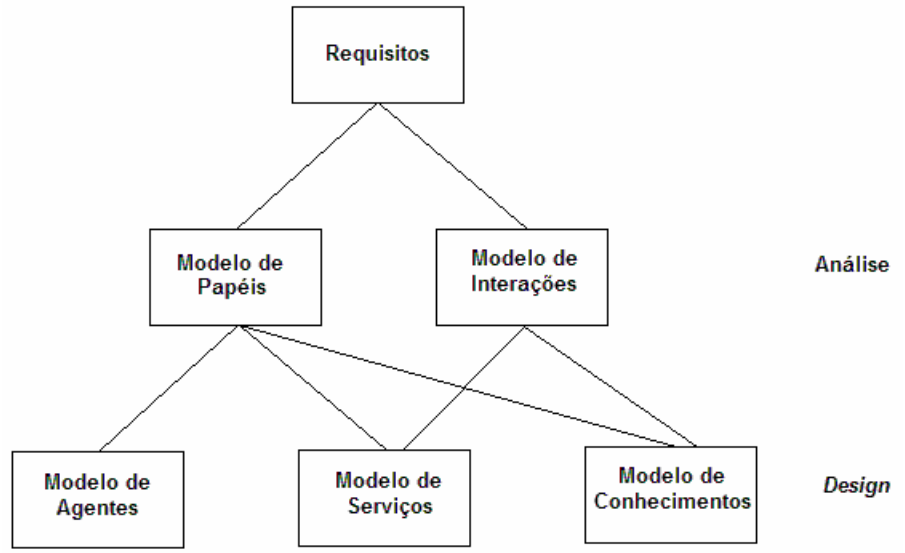

Figura 8 - Relacionamentos entre os modelos da metodologia Gaia (WOOLDRIDGE, 2000).

4.5.1. Fase de Análise

O objetivo da fase de análise é permitir o entendimento do sistema e sua estrutura, sem a referência a nenhum detalhe específico de implementação. No caso da Gaia, esse entendimento é obtido a partir da organização do sistema. A metodologia visualiza o sistema como uma coleção de papéis que sustentam relacionamentos uns com os outros e que tomam parte em padrões de interações sistemáticas e institucionalizadas com outros papéis (WOOLDRIDGE, 2000).

A entidade de nível mais abstrato na hierarquia de conceitos é o **sistema**. O termo sistema é usado, no contexto de agentes, para representar uma sociedade ou organização. De outra forma, um sistema baseado em agentes é uma sociedade ou organização artificial. Uma sociedade, a partir desta visão, é formada por um conjunto de **papéis**. Utilizando uma

[1] Tradução livre do autor para o termo "*liveness properties*".
[2] Tradução livre do autor para o termo "s*afety properties*".

analogia à uma forma de organização humana, pode-se afirmar que, em uma sala de aula típica existem dois papéis: o professor e o aluno. É importante notar que papel é um conceito abstrato. Em uma realização concreta de uma sala de aula tais papéis serão instanciados através de indivíduos que desempenharão os papéis de professor e aluno. A correspondência papel versus indivíduo não é estática, ou seja, muitos indivíduos podem desempenhar o papel de professor, inclusive aquele que já desempenhou o papel de aluno e vice-versa. Também não se trata de uma relação um-para-um. Um indivíduo pode desempenhar ao mesmo tempo mais de um papel.

Um papel é definido em termos de atributos: responsabilidades, permissões, atividades e protocolos.

Responsabilidades determinam a funcionalidade e é o atributo principal de um papel. Um exemplo de responsabilidade relacionada ao papel professor poderia ser ministrar aula. As responsabilidades são divididas em dois tipos: propriedades de sobrevivência e propriedades de segurança. As **propriedades de sobrevivência** representam o que o papel precisa produzir em função das condições ambientais. Voltando ao exemplo da sala de aula, ministrar aula, assim como lançar nota, são exemplos de propriedades de sobrevivência. Já as **propriedades de segurança** representam condições mínimas aceitáveis a serem mantidas durante todos os estados de execução. Em uma sala de aula, o aluno deve manter a média final em cada disciplina igual ou acima do valor mínimo de aprovação. Em um reator nuclear é preciso garantir que a temperatura do núcleo se mantenha dentro de certa faixa de temperatura.

No intuito de implementar responsabilidades, um papel possui uma gama de **permissões**. Permissões são, na realidade, os direitos associados ao papel. Identificam os recursos que estão disponíveis ao papel no intuito de permiti-lo cumprir suas responsabilidades. Na grande maioria dos sistemas implementados, permissões tendem a ser recursos informacionais. Um professor tem acesso à prova de cada aluno e, após a correção da mesma, lançará uma nota representando o desempenho do aluno naquela avaliação. No exemplo, prova e nota são permissões.

Finalmente, um papel possui, ainda, um conjunto de atividades e protocolos. **Atividades** são processos a serem executados por um indivíduo que implemente um determinado papel que não possua interação com outros papéis. Já **protocolos** são processos associados a um papel que englobam interação com outros papéis. O papel professor tem como exemplo de atividade preparar a aula. Apresentar uma dúvida ao professor pode ser caracterizado como um protocolo relacionado ao papel aluno.

A Figura 9 apresenta os conceitos da fase de análise e seus relacionamentos.

Desta forma, o modelo organizacional da metodologia Gaia é composto por dois modelos: o **Modelo de Papéis** e o **Modelo de Interações**.

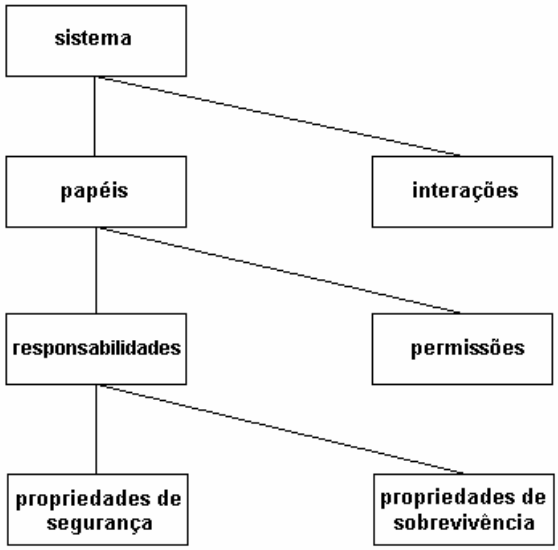

Figura 9 - Conceitos da fase de análise (WOOLDRIDGE, 2000).

Modelo de Papéis

Identifica os principais papéis de um sistema, suas atividades, protocolos, responsabilidades e permissões. A Tabela 4 apresenta um *template* para o modelo de papéis.

Papel	**Nome do papel**
Descrição	Pequena descrição do papel.
Protocolos e Atividades	Protocolos e atividades no qual o papel toma parte. As atividades devem ser apresentadas com o nome sublinhado.
Permissões	Recursos associados ao papel. Devem ser assinalados como os recursos são utilizados, se somente são consultados ou se alguma modificação é realizada.
Responsabilidades	
Sobrevivência	NOME_DO_PAPEL = Atividades e Protocolos conjugados com operadores.
Segurança	• Propriedade(s) de segurança listadas através de marcadores.

Tabela 4 - *Template* do Modelo de Papéis (WOOLDRIDGE, 2000).

Os operadores utilizados nas expressões de sobrevivência estão listados na Tabela 5.

Operadores	Interpretação
x.y	x seguido de y
x|y	x ou y ocorre
x*	x ocorre 0 ou mais vezes
x+	x ocorre 1 ou mais vezes
x^{∞}	x ocorre eternamente
[x]	x é opcional
x||y	x e y ocorrem de forma intercala

Tabela 5 - Operadores para expressões de sobrevivência (WOOLDRIDGE, 2000).

Modelo de Interações

Existem inevitáveis dependências e relacionamentos entre os vários papéis no contexto de uma organização multiagente. De fato, tal inter-relacionamento é ponto central para o funcionamento do sistema. Em função deste fato, interações precisam ser capturadas e representadas durante a fase de análise. Na Gaia, tais ligações entre papéis são modeladas no Modelo de Interações. O modelo engloba um conjunto de definições de protocolos, um para cada tipo de interação entre papéis. Um protocolo, neste contexto, pode ser visualizado como um padrão de interação formalmente definido, institucionalizado e independente do número de etapas que o compõe. Tal abordagem significa que uma única definição de protocolo pode representar uma variedade de mensagens intercambiadas através do sistema em tempo de execução. Na sala de aula, o protocolo que representa o diálogo entre aluno e professor no intuito de sanar alguma dúvida, provavelmente, envolverá uma coleção de perguntas e respostas até que o aluno tenha sua dúvida sanada (ou o professor desista de explicar o assunto!) (WOOLDRIDGE, 2000).

Uma definição de protocolo engloba os seguintes atributos:

- Propósito – descrição textual resumida da natureza da interação;
- Iniciador – papel responsável por iniciar a interação;
- Contraparte – papel com o qual o iniciador interage;
- Entrada – informação utilizada pelo papel iniciador enquanto executa o protocolo;
- Saída – informação fornecida pela ou para a contraparte durante o curso da interação;
- Descrição – descrição textual resumida de qualquer processamento que o protocolo iniciador executa durante o curso da interação.

A Tabela 6 apresenta um *template* para o Modelo de Interações.

Nome do Protocolo:		
Iniciador:	Contraparte:	Entrada:
Descrição:		Saída:

Tabela 6 - *Template* para o Modelo de Interações (WOOLDRIDGE, 2000).

Para WOOLDRIDGE (2000), o processo de análise da Gaia pode ser resumido da seguinte forma:

1. Identificar os papéis no sistema. Papéis tipicamente correspondem a:
   - Indivíduos, dentro de uma organização ou mesmo agindo independentemente;
   - Departamentos da organização;
   - A própria organização.

   Saída: um protótipo do Modelo de Papéis – uma lista de papéis-chave que ocorrem no sistema com uma descrição informal e não elaborada.
2. Para cada papel, identificar e documentar os protocolos associados.

   Saída: um Modelo de Interações que captura os padrões recorrentes de interações entre papéis.
3. Com base no Modelo de Interações, elaborar o Modelo de Papéis.

   Saída: um Modelo de Papéis totalmente refinado que documenta os papéis-chave de um sistema, suas permissões e responsabilidades, aliados a protocolos e atividades nos quais participam.
4. Repetir estágios 1 a 3.

4.5.2. Fase de *Design*

O objetivo tradicional da fase de *design* é transformar os modelos abstratos oriundos da fase de análise em modelos em um nível de abstração tal que seja possível implementá-los facilmente. Isto não ocorre no *design* orientado a agentes. O objetivo do *design* em Gaia é transformar os modelos de análise em modelos em um nível de abstração tal, que seja possível aplicar técnicas tradicionais de *design* (incluídas as orientadas a objeto) e a partir daí, implementar os agentes. De outra forma, Gaia preocupa-se em determinar quais os padrões de cooperação inter-agentes necessários para o atingimento dos objetivos do sistema e o que é necessário para que cada agente tenha capacidade de fazê-lo. Na verdade, como o agente

executa o serviço está além do escopo da Gaia e dependerá do domínio da aplicação (WOOLDRIDGE, 2000).

O modelo de *design* envolve três modelos: o **Modelo de Agentes**, o **Modelo de Serviços** e o **Modelo de Conhecimentos**[1].

Modelo de Agentes

Identifica os tipos de agentes (papéis) que irão compor o sistema e as instâncias (agentes) que irão ser instanciadas a partir destes tipos. A Tabela 7 apresenta os qualificadores de instância.

Qualificador	Significado
n	exatamente n instâncias
m..n	entre m e n instâncias
*	0 ou mais instâncias
+	1 ou mais instâncias

Tabela 7 - Qualificadores de instâncias (WOOLDRIDGE, 2000).

A Figura 10 apresenta um *template* para o Modelo de Agentes.

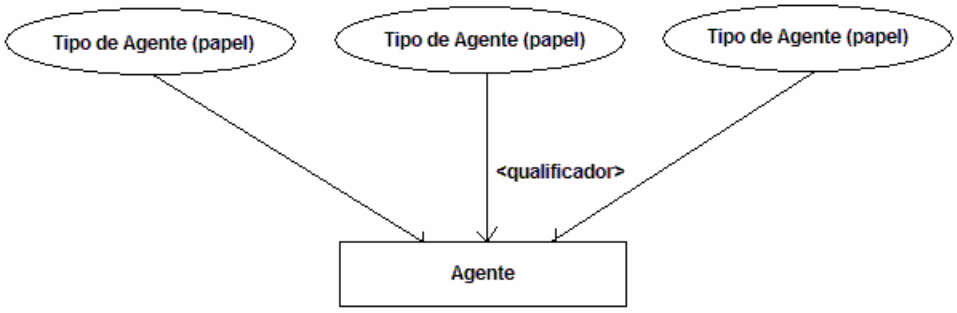

Figura 10 - *Template* do Modelo de Agentes.

Modelo de Serviços

Apresenta os serviços disponibilizados por cada papel e especifica as propriedades principais desses serviços. Por serviço, entenda-se função do agente. Para cada serviço a ser executado pelo agente é necessário documentar suas entradas, saídas, pré-condições, pós-condições. Entrada e saídas são derivadas do Modelo de Interações. Pré e pós-condições são restrições aos serviços e são derivadas das propriedades de segurança do Modelo de Papéis. Para cada papel será obrigatório apresentar pelo menos um serviço. De maneira geral, a definição dos serviços executados por um agente são derivados da lista de protocolos, atividades, responsabilidades e propriedades de sobrevivência (WOOLDRIDGE, 2000).

[1] Tradução livre do autor para o termo “acquaintance model”.

A Tabela 8 apresenta um *template* para o Modelo de Serviços.

Serviço	Entradas	Saídas	Pré-condições	Pós-condições

Tabela 8 - *Template* para o Modelo de Serviços.

Modelo de Conhecimentos

Define as ligações de comunicação que existem entre agentes. O modelo não define o quê ou quando as mensagens são enviadas, mostra apenas que a comunicação existe. O propósito principal do modelo é identificar potenciais gargalos na comunicação entre agentes que poderão vir a se tornar um problema em tempo de execução. É uma boa prática garantir que os sistemas apresentem baixo acoplamento e o modelo ajuda nesta identificação. O modelo é composto, simplesmente, por um grafo direcionado onde os nós são tipos de agentes e os arcos correspondem a caminhos de comunicação. O fato de existir um caminho de comunicação de a para b não significa, obrigatoriamente, que exista um caminho no sentido contrário. Os Modelos de Papéis, Interações e Agentes são as fontes naturais deste modelo (WOOLDRIDGE, 2000).

A Figura 11 apresenta um *template* para o Modelo de Conhecimentos.

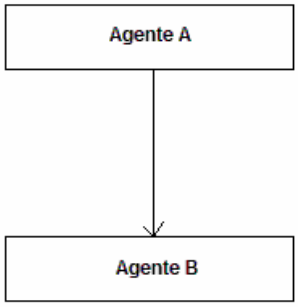

Figura 11 - *Template* do Modelo de Conhecimentos.

Para WOOLDRIDGE (2000), o processo de *design* da Gaia pode ser resumido da seguinte forma:

1. Criar o Modelo de Agentes;
    - Agregar papéis aos tipos de agentes e refinar o modelo a fim de hierarquizar tipos de agentes;
    - Documentar instâncias de cada tipo de agente usando anotações.
2. Desenvolver o Modelo de Serviços a partir do exame de atividades, protocolos e das propriedades de sobrevivência e segurança dos papéis.

3. Desenvolver o Modelo de Conhecimentos a partir do Modelo de Interações e do Modelo de Agentes.

4.6. Resumo

Este capítulo abordou:

- As características de um problema complexo;
- A adequação da abordagem SMA à solução de problemas complexos;
- Um apanhado das características principais de algumas propostas para modelagem de SMA;
- Um resumo descritivo da Metodologia Gaia, *templates* para todos os seus modelos e os processos de análise e desenvolvimento.

5. Ontologia

5.1. Introdução

Este capítulo apresentará o conceito de ontologia, sua aplicação, seu processo de desenvolvimento e as características da ferramenta Protégé para edição de ontologias.

5.2. Conceito

Uma ontologia é uma descrição formal e explicita de conceitos e seus relacionamentos em um domínio do discurso. Define um vocabulário comum para pesquisadores que necessitem trocar informações a respeito deste domínio. Inclui definições interpretáveis por máquinas dos conceitos básicos de um domínio e suas relações (NOY, 2005).

O desenvolvimento de uma ontologia difere do *design* de classes e relações do paradigma OO. Este centraliza o desenvolvimento em torno de métodos de classes – um programador OO toma decisões baseado em propriedades operacionais da classe, enquanto que o *designer* de ontologias toma decisões baseado em propriedades estruturais de uma classe. Como resultado, a estrutura de classe e o relacionamento entre classe em uma ontologia são diferentes em relação a um domínio similar na programação OO (NOY, 2005).

Uma ontologia é formada por **classes** (ou conceitos), por ***slots*** de cada conceito (ou propriedades ou papéis) e restrições dos *slots* (***facets*** ou restrições). Uma ontologia, em conjunto com uma coleção de instâncias individuais de classes, constitui uma base de conhecimentos.

Classes descrevem conceitos de um domínio. Uma classe Aluno, por exemplo, representa todos os alunos. Alunos específicos, o João, o Pedro e a Maria, são instâncias da classe Aluno. Classes podem apresentar subclasses que representam conceitos mais específicos que a superclasse. A subclasse AlunoUFRJ representa todos os alunos que estudam na UFRJ, por exemplo.

Slots descrevem propriedades das classes e suas instâncias. A classe Aluno tem nome, data de nascimento, tipo sanguíneo e etc. Todas as instâncias das classes apresentarão valores para os *slots* especificados na classe. A classe Aluno pode ter um *slot* chamado professores que referenciará todas as instâncias da classe Professor (ou seja, todos os professores) que ministram aula para uma determinada instância da classe Aluno.

A Figura 12 exibe classes, instâncias e seus relacionamentos.

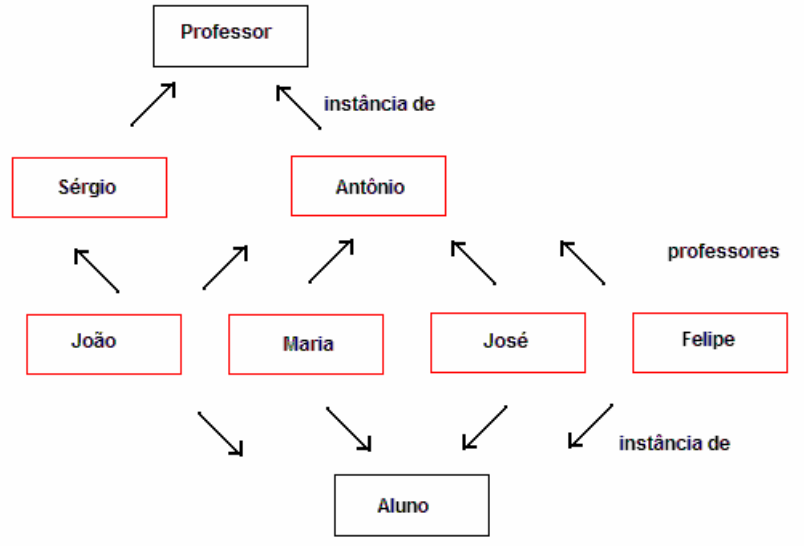

Figura 12 - Classes, instâncias e seus relacionamentos.

A Figura 13 mostra detalhes da classe Aluno.

Class: Aluno

Documentation: Alunos.

Superclasses
:THING

Subclasses
None

Types
:STANDARD-CLASS

Template Slots

	Slot Name	Documentation	Type	Cardinality
	data de nascimento	Data de nascimento.	String	1:1
	nome	Nome.	String	1:1
	professores	Professores do aluno.	Class	0:1
	tipo sangüíneo	Tipo de sangue.	String	1:1

Figura 13 - Classe Aluno.

5.3. Motivação

Uma ontologia, segundo NOY (2005) pode ser desenvolvida por vários motivos, entre eles:

- Disponibilizar um entendimento comum sobre uma estrutura de informação entre pessoas e agentes de *software*;
- Habilitar o reuso de um domínio do conhecimento;
- Tornar suposições públicas;
- Separar domínio do conhecimento do domínio operacional;

- Analisar um domínio do conhecimento.

5.4. Processo de Desenvolvimento de Ontologias

Antes de apresentar o processo de desenvolvimento propriamente dito, algumas regras fundamentais no desenvolvimento de ontologias devem ser enfatizadas (NOY, 2005):

- Não há um único meio de modelar um domínio e sim várias alternativas possíveis. A melhor solução depende da aplicação e das extensões previstas;
- O processo de desenvolvimento é necessariamente iterativo;
- Conceitos na ontologia devem ser objetos (físicos e lógicos) e relacionamentos do domínio de interesse. Geralmente são substantivos (objetos) ou verbos (relacionamentos) em sentenças que descrevem seu domínio.

O processo de desenvolvimento proposto por NOY (2005) contém sete passos, a saber:

1. Determinar o domínio e o escopo da ontologia;
2. Considerar o reuso de ontologias já existentes;
3. Enumerar termos importantes para a ontologia;
4. Definir as classes e sua hierarquia;
5. Definir as propriedades das classes – *slots*;
6. Definir *facets* (restrições) dos *slots*.
7. Criar instâncias.

5.5. A Ferramenta Protégé

O Protégé é uma ferramenta de *software* integrada usada por desenvolvedores de sistemas e especialistas de domínio para construção de bases de conhecimento (as ontologias adicionadas às instâncias de suas classes). A ferramenta é um projeto da *Stanford University*. Maiores detalhes pode ser obtidos no endereço *http://protege.stanford.edu*.

A Figura 14 mostra a interface gráfica da ferramenta.

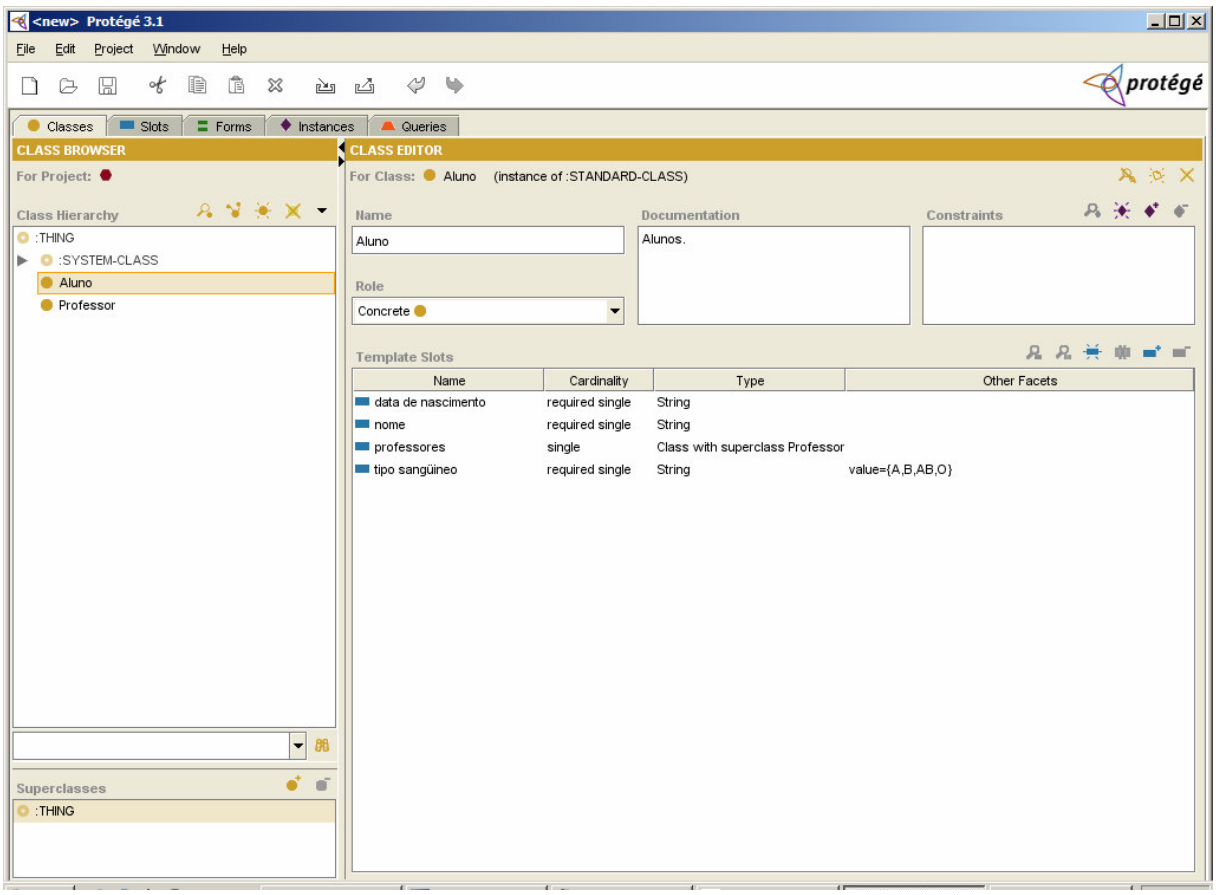

Figura 14 - A ferramenta Protégé.

5.6. Resumo

Este capítulo apresentou:

- O conceito de ontologia;
- A motivação para construção de ontologias;
- Um processo de desenvolvimento dessas ontologias;
- Uma ferramenta voltada para construção de ontologias.

6. *Implementação de Sistemas Multiagentes Utilizando JADE*

6.1. Introdução

Este capítulo abordará aspectos da fase de implementação de SMA através da utilização do *framework* JADE. Antes, porém, alguns conceitos básicos e descrições serão apresentados.

6.2. Middleware

BELLIFEMINE (2003) explicita que o termo *middleware* descreve um conjunto de bibliotecas de alto nível que permitem facilitar e tornar mais eficiente o desenvolvimento de aplicações através do fornecimento de serviços genéricos úteis, tais como: comunicação, acesso a dados, codificação, gerenciamento de recursos e etc. De forma geral, a idéia está em fornecer serviços básicos independentes em relação ao sistema operacional adotado. A Figura 15 mostra um diagrama de blocos no intuito de permitir a visualização do papel do m*iddleware*.

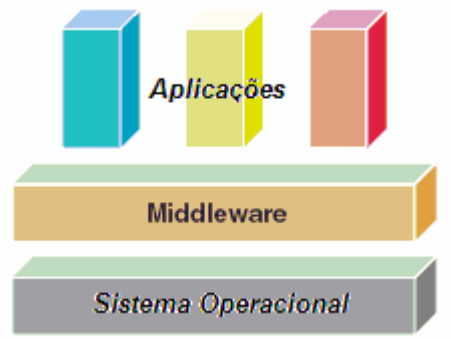

Figura 15 - Papel do *Middleware* (BELLIFEMINE, 2003).

6.3. Framework

Segundo JOHNSON (1997 *apud* OLIVEIRA, 2001), um *framework* é "um esqueleto de uma aplicação que deve ser parametrizado pelo desenvolvedor de aplicações". Para LARMAN (2002) um *framework* é "um conjunto extensível de objetos voltado para uma determinada função". Sobre *frameworks* LARMAN (2002) define, adicionalmente, como "um conjunto coeso de interfaces e classes que colaboram para fornecer serviços para o núcleo invariável de um determinado artefato de software".

Como um esqueleto, ou seja, um artefato de software genérico e "inacabado", um *framework* precisa ser adequado a uma determinada realidade, ou seja, instanciado. O processo de instanciação de um *framework* consiste em adaptar o código genérico, idealizado para atender um conjunto limitado de aplicações, a uma necessidade específica.

6.4. FIPA

A *Foundation for Intelligent Physical*[1] *Agents* (FIPA) foi fundada em 1996 com a missão de promover tecnologias e especificações de interoperabilidade que busquem facilitar a cooperação fim a fim de sistemas de agentes inteligentes relacionados ao cenário comercial e industrial moderno (FIPA, 2005). De outro modo, trata-se de uma associação internacional sem fins lucrativos que reúne empresas e organizações no intuito de produzir especificações relacionadas à tecnologia de agentes. Atualmente, a FIPA está em vias de se tornar um Comitê do *Institute of Eletrical and Eletronics Engineering* (IEEE).

A Figura 16 apresenta o Modelo de Comunicação da FIPA.

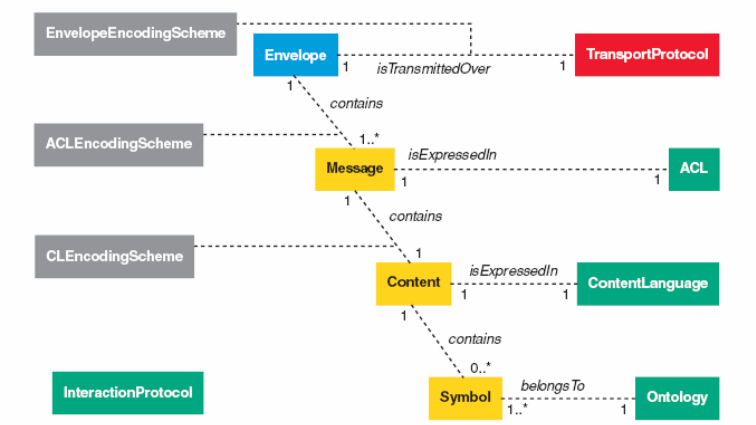

Figura 16 - Padrão FIPA: Modelo de Comunicação (BELLIFEMINE, 2003).

6.5. Java

Criada pela *Sun MicroSystems* em 1995 para desenvolvimento de programas em ambientes heterogêneos ligados em rede. Objetivava, principalmente, a utilização em sistemas isolados com quantidade mínima de memória, tipicamente em dispositivos eletrônicos para o consumidor final como geladeira, microondas, televisor e outros.

Trata-se de uma linguagem derivada do C++. No entanto, algumas características da linguagem original foram retiradas ou modificadas no intuito de simplificar o desenvolvimento em Java, reduzindo as possibilidades de erros durante a codificação. Suporta

[1] A palavra *physical* de motivos históricos. Inicialmente, a FIPA pretendia padronizar agentes robóticos também.

o desenvolvimento de software conforme o paradigma Orientado a Objetos (OO) e possui um conjunto de funcionalidades que fornece suporte ao desenvolvimento de sistemas heterogêneos, distribuídos e *multithread*[1]. Os programas construídos em Java são, de forma geral, compilados em *bytecodes*. Tais *bytecodes* podem ser interpretados em qualquer plataforma que possua uma máquina virtual Java (JVM). Opcionalmente, é possível traduzir *bytecodes* em código nativo de uma determinada plataforma, aumentando o desempenho final do programa construído.

6.6. RMI

A *Remote Method Invocation*, segundo SILVA (2003), é:

> Um conjunto de classes e interfaces em Java que encapsulam vários mecanismos de troca de dados, a fim de simplificar a execução de chamadas de métodos remotamente localizados em espaços de endereçamentos diferentes. [...] Um objeto RMI é um objeto remoto em Java cujos métodos podem ser chamados de outra JVM, normalmente usando a rede.

O RMI foi incorporado ao Java a partir de 1997 como parte da biblioteca padrão da versão 1.1 do *Java Development Kit* (JDK 1.1).

6.7. JADE

O *Java Agent DEvelopment framework* (JADE) é um *middleware* e um *framework* totalmente escrito em Java para desenvolvimento de Sistemas Multiagentes aderentes ao padrão FIPA. O JADE foi desenvolvido e é mantido pelo *Telecom Italia Lab* (TILAB) em parceria com a Universidade de Parma. É distribuído sobre a *GNU Lesser General Public License* (LGPL) e, portanto, trata-se de um projeto *open-source*.

A Figura 17 apresenta a logomarca da ferramenta.

[1] Propriedade que define a execução de múltiplas linhas de código concorrentemente via utilização de múltiplos *threads*.

Figura 17 - Logomarca da Ferramenta JADE (JADE, 2005).

6.7.1. Principais Características do JADE

Plataforma Distribuída de Agentes – JADE pode ser distribuído por várias máquinas, desde que uma conexão RMI possa ser mantida entre elas. Apenas uma aplicação Java e uma JVM são executadas em cada máquina. Os agentes são implementados via *threads* e inseridos dentro de repositórios chamados *containers*. Estes provêm todo o suporte necessário à execução do agente.

Interface Gráfica – Permite o gerenciamento, inclusive remoto, de agentes e *containers*.

Ferramentas de Depuração – Permitem a depuração do código construído e, desta forma, ajudam no processo de desenvolvimento.

Suporte a Comportamentos – Suporte a execução de múltiplas, paralelas e concorrentes atividades de agentes através dos modelos de comportamentos (*behaviors*).

Aderente ao Padrão FIPA – O JADE é totalmente aderente às especificações da FIPA. JADE foi aprovado no primeiro teste de interoperabilidade da FIPA realizado em Seul em janeiro de 1999 e no segundo teste realizado em Londres em abril de 2001.

Suporte à Mobilidade no Contexto da Plataforma – Suporta a mobilidade de agentes no contexto da plataforma JADE incluindo a transferência tanto de código como de estado, quando necessário.

Transporte de Mensagens – Transporte de mensagens no formato FIPA-ACL no contexto da plataforma.

Biblioteca de Protocolos FIPA - Disponibiliza uma biblioteca de protocolos prontos para interação entre agentes.

Automação de Registros – Registro e cancelamento automático de agentes. Permite ao desenvolvedor abstrair desse processo.

Integração – Possibilita que aplicações externas carreguem agentes autônomos JADE.

6.7.2. Plataforma de Agentes

Uma plataforma de agentes como descrita pela FIPA é representada conforme a Figura 18.

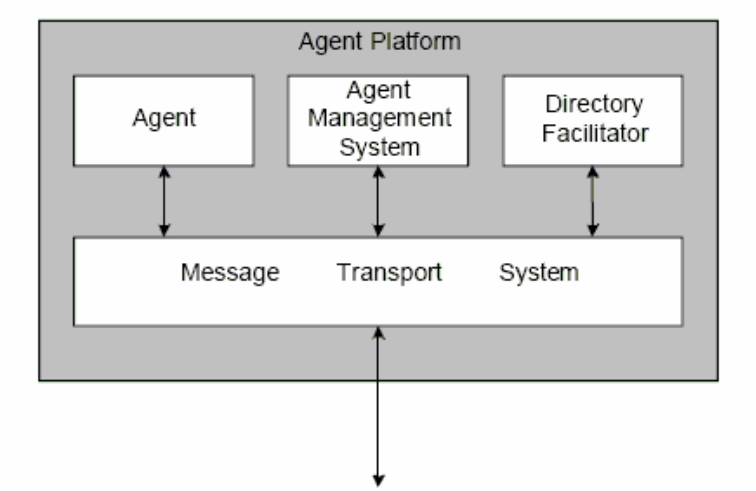

Figura 18 - Modelo da Plataforma de Agentes (BELLIFEMINE, 2005).

Agent Management System

O *Agent Management System* (AMS) é o agente responsável pelo controle de acesso e uso da plataforma de agentes. Somente um único AMS existirá em uma determinada plataforma. O AMS fornece os serviços de *white-page*, controle do ciclo de vida, manutenção de diretório de identificadores de agentes (*Agent ID – AID*) e de estados dos agentes. Cada agende deve se registrar junto ao AMS no intuito de adquirir um AID válido (BELLIFEMINE, 2005).

Directory Facilitator

O *Directory Facilitator* (DF) é o agente que fornece o serviço padrão de *yellow-page* na plataforma. Isto é, divulga os agentes da plataforma e seus serviços.

Message Transport System

O *Message Transport System*, também chamado de *Agent Communication Channel* (ACC), é o componente de software responsável por controlar todas as trocas de mensagens no contexto da plataforma e entre plataformas.

6.7.3. JADE e a Plataforma de Agentes

O JADE é totalmente aderente a esta especificação. Os agentes especiais AMS e DF são automaticamente criados no momento da abertura da plataforma. Neste momento o módulo ACC é configurado para permitir a comunicação por mensagens.

A plataforma pode ser distribuída entre vários equipamentos. Somente uma aplicação Java e uma JVM podem existir em cada equipamento. Cada JVM é, basicamente, um *container* que fornece um ambiente completo para execução de agentes. Além disso, o *container* permite a execução concorrente de diversos agentes em um mesmo equipamento. O *container* principal (*main*) ou *front-end* é aquele onde residem o AMS e o DF. Outros *containers* de uma mesma plataforma precisam se conectar ao principal (BELLIFEMINE, 2005). A Figura 19 mostra uma plataforma de agentes distribuída em JADE.

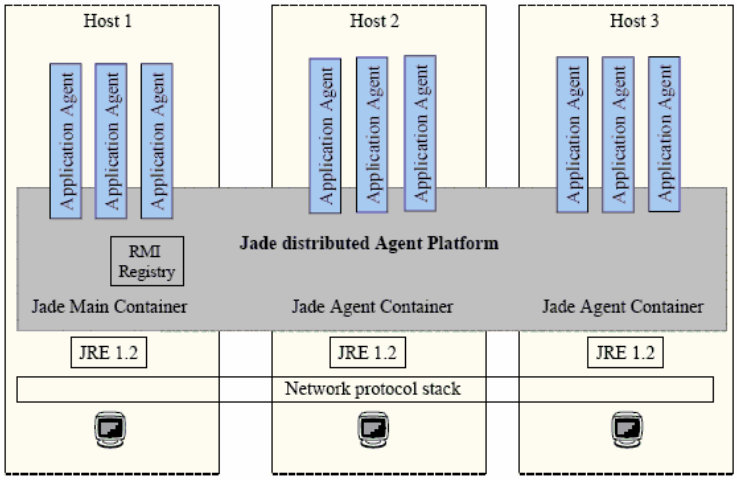

Figura 19 - Plataforma de Agentes Distribuída (BELLIFEMINE, 2005).

6.7.4. Agentes em JADE

A FIPA nada especifica sobre as estruturas internas dos agentes. Um agente JADE é implementado como uma classe Java chamada *Agent* que é definido internamente na ferramenta. Essa classe *Agent* representa uma classe básica comum para definir agentes. De outra forma, funciona como uma superclasse para a criação de agentes de software definidos pelo usuário (SILVA, 2003). Isto implica que são herdadas funcionalidades básicas para interação com a plataforma de agentes, tais como: registro, configuração, gerenciamento remoto e outras. Um conjunto básico de métodos para implementação de comportamentos definidos pelo próprio usuário, como enviar e receber mensagens, utilizar protocolos de

interação padronizados ou registro em muitos domínios, também são herdados (BELLIFEMINE, 2005).

O modelo computacional de um agente é multitarefa, onde tarefas ou comportamentos (*behaviors*) são executados concorrentemente. Cada funcionalidade fornecida por um agente pode ser implementada por um ou mais comportamentos. Um escalonador, interno à classe base *Agent*, automaticamente gerencia o escalonamento dos comportamentos.

A Figura 20 apresenta a arquitetura do agente em JADE.

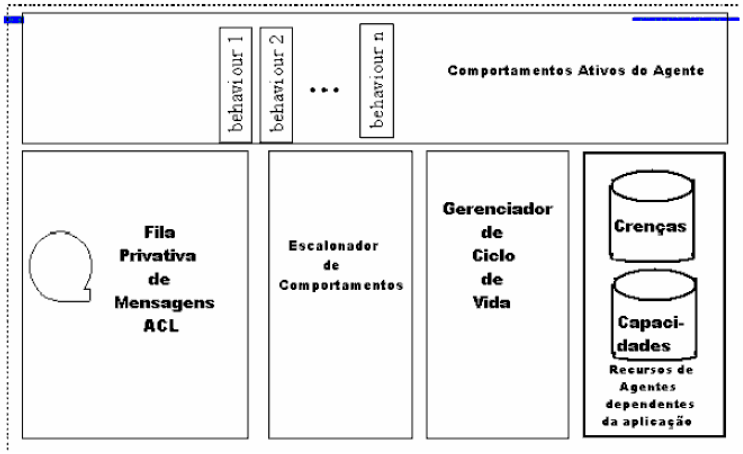

Figura 20 - Arquitetura do Agente JADE (SILVA, 2003).

Ciclo de Vida do Agente

A Figura 21 mostra o ciclo de vida do agente, enquanto a Tabela 9 descreve os estados deste ciclo de vida.

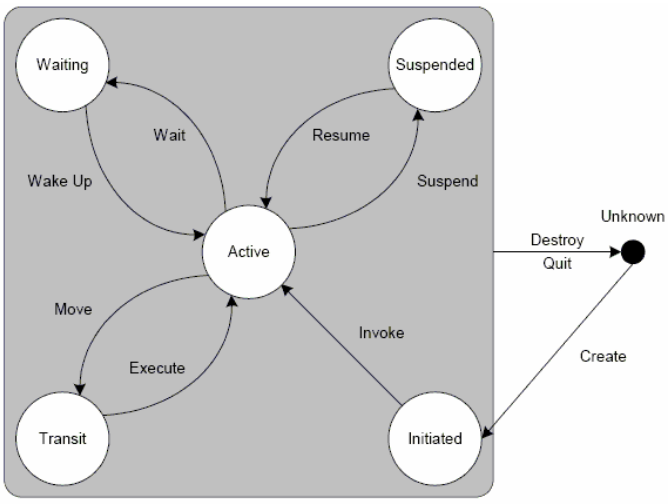

Figura 21 - Ciclo de Vida do Agente (BELLIFEMINE, 2005).

Estado	Descrição
INITIATED	Está construído, mas ainda não se registrou junto ao AMS, não tem nome nem endereço e não pode se comunicar com outros agentes.
ACTIVE	Está registrado, tem nome e endereço e pode acessar as diversas funcionalidades do JADE.
SUSPENDED	Está suspenso no momento. Sua *thread* interna está suspensa e nenhum comportamento (*behavior*) está em execução.
DELETED	Está definitivamente "morto". A *thread* interna terminou sua execução e não está mais registrado junto ao AMS.
WAITING	Está bloqueado, esperando por algo. Sua *thread* interna está em estado de espera no monitor Java e irá acordar quando alguma condição for alcançada, tipicamente quando uma mensagem chega.
TRANSIT	Um agente móvel entra nesse estado quando está em processo de migração para outra localização. O sistema continua a armazenar as mensagens que serão enviadas para a nova localização.

Tabela 9 - Estados do ciclo de vida do agente (BELLIFEMINE, 2005).

Alguns métodos são fornecidos pelo JADE na intenção de permitir a transição entre os diversos estados. A Tabela 10 apresenta os métodos fornecidos pelo JADE para transição de estados.

Assinatura do Método	Descrição
public void doActivate ()	SUSPENDED → ACTIVE ou WAITING
public void doSuspend ()	ACTIVE ou WAITING → SUSPENDED
public void doWait ()	ACTIVE → WAITING
public void doWake ()	WAITING → ACTIVE
public void doDelete ()	ACTIVE ou SUSPENDED ou WAITING → "deleted"
public void doMove (...)	ACTIVE → TRANSIT
public void doClone (...)	ACTIVE → "cloned"

Tabela 10 - Métodos para transição de estados.

Outros Métodos Importantes da Classe *Agent*

A Tabela 11 apresenta outros métodos da classe *Agent*.

Assinatura do Método	Descrição
protected void setup ()	Executa procedimentos de inicialização.
public void addBehavior (...)	Adiciona um novo comportamento.
public final void send (...)	Envia mensagem ACL para outro agente.
public final ACLMessage receive ()	Recebe mensagem ACL da fila de mensagens.

Tabela 11 - Outros Métodos da Classe Agent.

6.7.5. Comportamentos

Comportamentos ou *behaviors* são as ações que um agente desempenha dentro de um Sistema Multiagente. A implementação desses comportamentos é realizada através de subclasses herdadas da classe *Behavior*. A Tabela 12 lista alguns métodos da classe *Behavior*.

Assinatura do Método	Descrição
public void action ()	Aqui são implementadas as ações que o comportamento irá tomar.
public boolean done ()	Indica se o comportamento foi terminado ou não.
public void block (...)	Bloqueia comportamento.
public void restart ()	Reinicia comportamento bloqueado.
public void reset ()	Restaura estado inicial do comportamento.
public boolean isRunnable ()	Índica se o comportamento está bloqueado ou não.

Tabela 12 - Métodos da Classe Behavior.

Adicionalmente, o atributo *myAgent* informa o agente a qual pertence o comportamento.

Cada comportamento criado deve, então, ser adicionado ao agente alvo (uma subclasse personalizada da classe *Agent*). De forma geral, os comportamentos podem ser simples ou complexos. Comportamentos simples são atômicos, ou seja, são monolíticos e não podem ser interrompidos. Já os comportamentos complexos ou compostos, são formados por um ou mais comportamentos simples. A Figura 22 exibe a hierarquia de comportamentos.

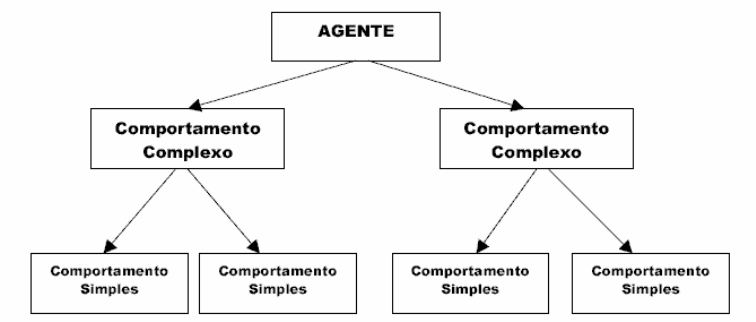

Figura 22 - Hierarquia de Comportamentos (SILVA, 2003).

Classes de Comportamento Prontas

JADE disponibiliza algumas classes de comportamento prontas. Para utilizar tais classes o desenvolvedor necessitará apenas adequá-las as necessidades específicas do agente. As Figuras 23, 24 e 25 apresentam as classes de comportamento fornecidas pelo JADE.

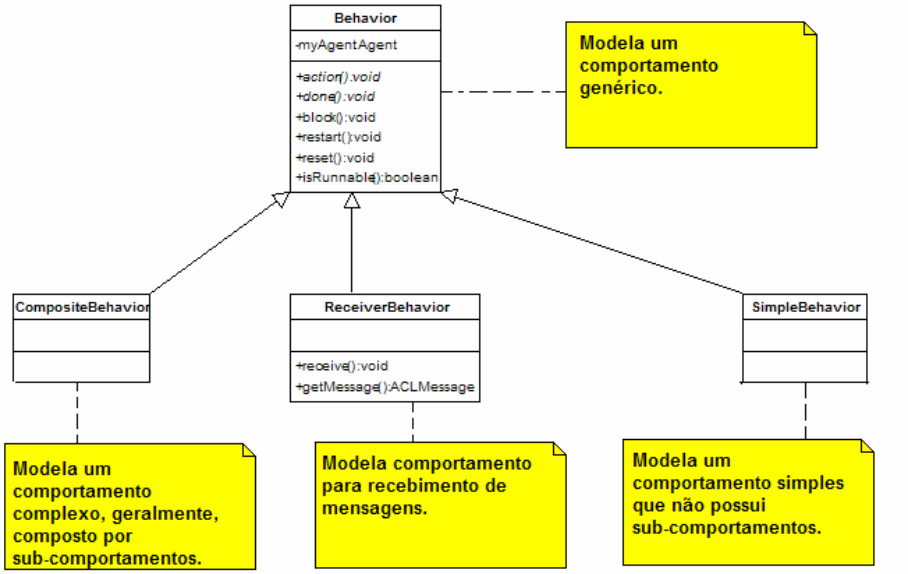

Figura 23 - Classes de Comportamento - Visão Geral.

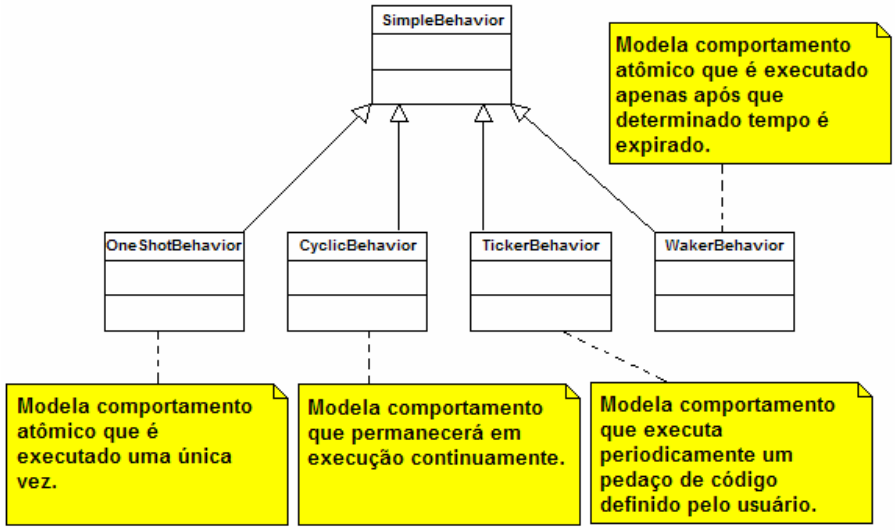

Figura 24 - Classes de Comportamento Simples.

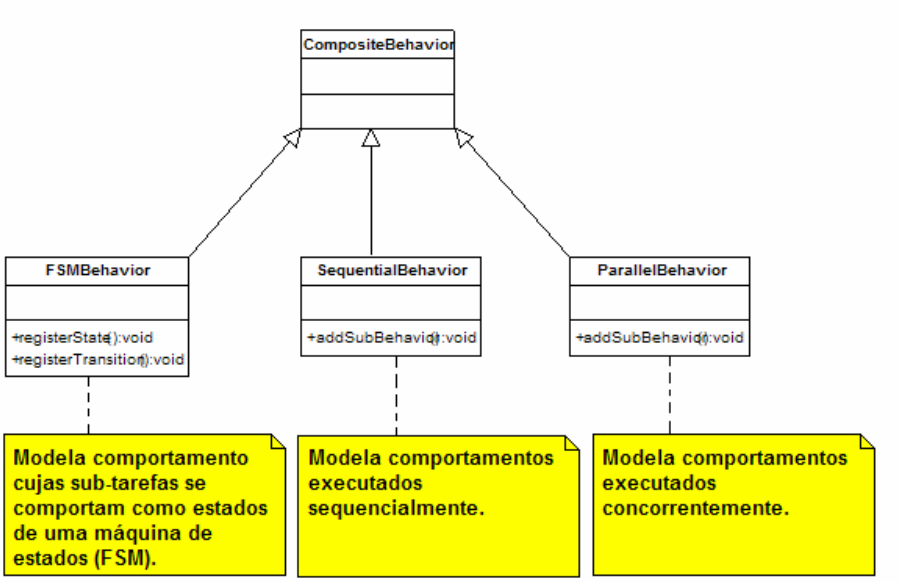

Figura 25 - Classes de Comportamento Complexas.

6.7.6. Troca de Mensagens

O esquema de troca de mensagens do JADE se baseia na utilização de métodos próprios e com o uso de instâncias da classe *ACLMessage*. Tal classe implementa uma coleção de atributos em conformidade com a especificação FIPA, ou seja, implementa a linguagem FIPA-ACL. Um agente deve instanciar a classe *ACLMessage*, preencher as informações necessárias e chamar o método *send ()* da classe *Agent*. Para receber mensagens deve-se utilizar os métodos *receive()* ou *blockingReceive ()*. Outra forma de enviar ou receber mensagens se dá através do uso das classes *SenderBehavior* e *ReceiverBehavior*. Tal procedimento permite que as trocas de mensagens possam ser escalonadas como atividades independentes do agente (SILVA, 2003).

De forma geral, os atributos da classe *ACLMessage* são acessíveis via métodos *get* e *set*. A Tabela 13 lista alguns dos métodos da classe *ACLMessage*.

Assinatura do Método	Descrição
public void addReceiver (...)	Adiciona destinatário da mensagem (agente).
public void removeReceiver (...)	Remove destinatário.
public ACLMessage createReply ()	Cria nova mensagem em resposta à determinada mensagem.
public String getContent ()	Retorna conteúdo da mensagem.
public void setContent (...)	Define conteúdo da mensagem a ser enviada.
public void setOntology (...)	Define ontologia da mensagem.

Tabela 13 - Métodos da Classe ACLMessage.

6.7.7. Interoperabilidade

O mecanismo de comunicação utilizado pelo JADE é selecionado de acordo com a localização de remetentes e receptores (SILVA, 2003). A Tabela 14 resume as situações e os respectivos mecanismos de comunicação utilizados pela ferramenta.

Situação	Mecanismo
Mesmo container.	Eventos Java (nenhuma invocação remota).
Mesma plataforma JADE.	Java RMI.
Diferentes plataformas JADE.	Protocolo IIOP.

Tabela 14 - Mecanismos de Interoperabilidade.

6.7.8. Ontologias, Linguagens e Protocolos

De modo a tornar possível a comunicação entre agentes é necessário que estes compartilhem um mesmo vocabulário (ontologia), linguagem e protocolos. As ontologias já foram abordadas no Capítulo 5 deste trabalho. A linguagem é a forma de expressão do vocabulário para fins de transmissão da mensagem. Protocolos são conjuntos predefinidos de atos de comunicação utilizados para obtenção de certo objetivo.

Em JADE, a implementação de uma ontologia é realizada através de uma extensão da classe predefinida *Ontology* e de um conjunto de *schemas* descrevendo a estrutura de conceitos, ações e predicados que podem ser utilizados como conteúdo de uma mensagem. Também é possível estender as classes *BasicOntology* e *ACLOntology* que, por seu turno, estendem a classe *Ontology*.

Conceitos

Expressões que indicam entidades de estrutura complexa que podem ser definidos em termos de *slots*. O conceito Aluno, por exemplo, engloba os atributos (*slots*) de nome e idade.

Predicados

São expressões que comunicam algo sobre o estado de coisas. Podem assumir os valores verdadeiro ou falso. O predicado AssisteAula, por exemplo, define que o Aluno João assiste aula do Professor (conceito) Alberto.

Ações

Tipo especial de conceito cujo objetivo é indicar que ação pode ser executada por um agente.

Schemas são instâncias das classes predefinidas *ConceptSchema*, *PredicateSchema* e *AgentActionSchema* incluídas no pacote *jade.content.schema*. Cada *schema*, então, é associado a uma classe ou interface Java.

- *Schema* do tipo *ConceptSchema* → associado a uma classe que implementa a interface *Concept* (direta ou indiretamente);
- *AgentActionSchema* → classe que implementa *AgentAction*;
- *PredicateSchema* → classe que implementa *Predicate*.

A Tabela 15 exibe o código para a classe Aluno (ontologia) em Java que implementa o conceito Aluno.

```
public class Aluno implements Concept {
        private String nome;
        private int idade;
        …
        public String getNome() { return nome; }
        public void setNome(String nome) {this.nome=nome;}
        public int getIdade() { return idade;}
        public void setIdade(int idade) {this.idade=idade;}
        ...
}
```

Tabela 15 - Classe Aluno em Java.

O conteúdo de uma mensagem no padrão FIPA-ACL nada mais é do que uma cadeia de caracteres, ou seja, uma *string*. Para que dois agentes troquem uma mensagem que envolva informações de Aluno, por exemplo, será necessário transformar um objeto da classe Java Aluno em uma cadeia de caracteres. Uma alternativa seria, simplesmente, serializar o objeto através do método *setContentObject(<objeto Aluno>)* fornecido pelo JADE e extrair o objeto da *string* via *getContentObject()*. O problema dessa abordagem é que destino e o remetente precisam, necessariamente, acordar, a priori, que tipo de informação está sendo enviada e recebida.

Uma forma de comunicação mais genérica é fornecida através da utilização de linguagens de conteúdo. A linguagem determina o formato do conteúdo da mensagem em função da informação que se deseja transmitir. O pacote *jade.content* provê codificadores e decodificadores para duas linguagens de conteúdo: SL e LEAP.

A SL é uma linguagem de conteúdo legível ao ser humano e é, provavelmente, a mais difundida na comunidade científica no contexto de agentes inteligentes. A Tabela 16 mostra um objeto da classe Aluno em SL.

(Aluno :nome "João" :idade 15)

Tabela 16 - Objeto da classe Aluno em SL.

Para preencher o conteúdo de uma mensagem utilizando a linguagem SL é necessário utilizar o método *fillContent(<mensagem>,<objeto>)*. Para extrair o objeto da mensagem existe o método *extractContent(<mensagem>)*. A Tabela 17 mostra os métodos para manipulação do conteúdo da mensagem FIPA-ACL em função dos tipos de conteúdo.

Tipo de Conteúdo	Get	Set
String	getContent(...)	setContent(...)
Objeto Java	getContentObject(...)	setContentObject(...)
Ontologia	extractContent(...)	fillContent(...)

Tabela 17 - Métodos para manipulação de conteúdo.

Os métodos *setOntology(<ontologia>)* e *setLanguage(<linguagem>)* são invocados no intuito de configurar a ontologia e a linguagem, respectivamente, a serem utilizados em uma determinada mensagem.

6.8. Resumo

Este capítulo apresentou:

- Alguns conceitos básicos como *framework*, *middleware* e RMI, necessários ao entendimento da natureza do JADE;
- As principais características do *framework* JADE;
- Conceitos de agentes, comportamentos, mensagens e interoperabilidade em JADE.
- A implementação de ontologias e linguagens de conteúdo em JADE.

7. Estudo de Caso

7.1. Introdução

Este capítulo apresentará a implementação de um estudo de caso baseado em uma versão modificada do Problema dos Cinco Filósofos Famintos.

7.2. Descrição Original do Problema

Formulado originalmente por DIJKSTRA (1971), em seu trabalho *Hierarchical Ordering of Sequential Processes*, o problema consiste na existência de cinco filósofos que pensam (trabalham) e sentem fome periodicamente. Ao sentirem fome os filósofos se encaminham a uma mesa em cujo centro encontra-se uma travessa de comida. Na periferia da mesa existem cinco pratos e um talher para cada filósofo. Infelizmente, são necessários dois talheres para que um filósofo consiga se alimentar. Desta forma, se cada filósofo se apossar de um talher, nenhum deles poderá comer e saciar sua fome e todos morrerão de inanição. Assim, apenas dois filósofos podem comer simultaneamente. A Figura 26 apresenta uma representação visual do problema.

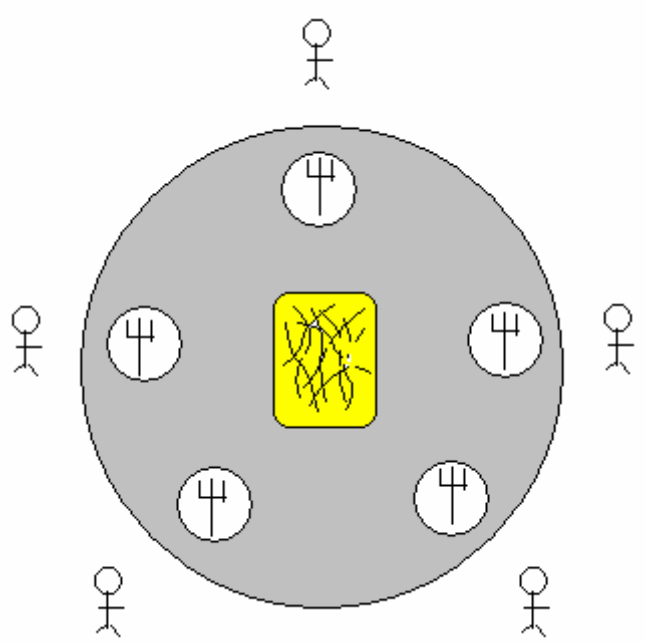

Figura 26 - Cinco Filósofos Famintos - versão original.

7.3. Descrição Modificada do Problema

No contexto deste trabalho o problema original foi ligeiramente modificado. Ao estilo do documento produzido por KIMLAYCHUK (2003), a idéia dos cinco pratos e cinco talheres foi descartada. Ao invés disso, a mesa tem uma restrição: apenas dois filósofos podem comer simultaneamente. Adicionalmente, um filósofo muito faminto que não obteve acesso à mesa poderá ir ao hospital. O hospital também apresenta uma restrição, pois tem capacidade para tratar, apenas, um filósofo por vez. O filósofo que saciar sua fome poderá levantar da mesa permitindo, assim, que outro tenha acesso à comida. Da mesma forma, o filósofo hospitalizado poderá ser liberado assim que estiver restabelecido. O pensador faminto que não conseguir comer e não tiver acesso ao hospital morrerá de inanição. A Figura 27 mostra uma representação visual da versão modificada.

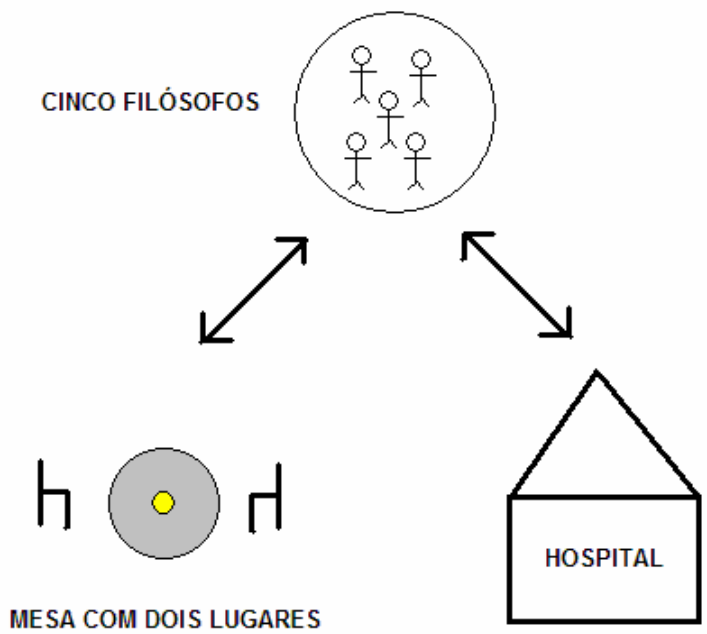

Figura 27 - Cinco Filósofos Famintos – versão modificada.

7.3.1. Adequação do Problema Proposto à Abordagem Baseada em Agentes

O problema proposto apresenta algumas características relevantes à aplicação da abordagem baseada em agentes. Algumas das justificativas são listadas abaixo:

- O problema envolve fontes de informações distribuídas e egoístas (os Filósofos, a Mesa e o Hospital);

- A modelagem do problema é inerentemente orientada a agentes, afinal, os Filósofos são entidades tipicamente autônomas;
- Os Filósofos não apresentam uma visão global. Informações quanto à vacância da Mesa ou do Hospital ou, ainda, sobre o estado de outros Filósofos não são conhecidas pelos Filósofos. Logo, trata-se de um problema caracterizado pela ausência de visão global por parte dos agentes em contraste a existência de uma coleção de agentes com visão local.

7.4. Processo de Desenvolvimento

7.4.1. Construção da Ontologia

Com o objetivo de construir a ontologia referente ao problema proposto foi utilizado o *software* Protégé. O processo de construção descrito no Capítulo 5 foi seguido passo a passo. As Tabelas 18 a 26 exibem o resultado final deste processo de construção.

Conceito: Filósofo

Nome	Tipo	Cardinalidade	Observações
id	AID[1]	1 (obrigatório)	
nome	*String*	0 (opcional)	
energia	*Integer*	1 (obrigatório)	mínimo=0
estado	*Classe Estado*	1 (obrigatório)	

Tabela 18 – Estudo de Caso – Descrição do conceito Filósofo.

Conceito: Estado

Nome	Tipo	Cardinalidade	Observações
nomeEstado	*String*	1 (obrigatório)	

Tabela 19 – Estudo de Caso – Descrição do conceito Estado.

Conceito: Necessidade

Nome	Tipo	Cardinalidade	Observações
nomeNecessidade	*String*	1 (obrigatório)	

Tabela 20 – Estudo de Caso – Descrição do conceito Necessidade.

[1] *Agent IDentification.*

Conceito: Hospital

Nome	Tipo	Cardinalidade	Observações
vacância	*Integer*	1 (obrigatório)	
camas	*Classe Cama*	1:1 (obrigatório)	

Tabela 21 – Estudo de Caso – Descrição do conceito Hospital.

Conceito: Cama

Nome	Tipo	Cardinalidade	Observações
filósofo	*Classe Filósofo*	1 (obrigatório)	
requisiçãoId[1]	*String*	1 (obrigatório)	

Tabela 22 – Estudo de Caso – Descrição do conceito Cama.

Conceito: Mesa

Nome	Tipo	Cardinalidade	Observações
vacância	*Integer*	1 (obrigatório)	
lugares	*Classe Lugar*	1:2 (obrigatório)	

Tabela 23 – Estudo de Caso – Descrição do conceito Mesa.

Conceito: Lugar

Nome	Tipo	Cardinalidade	Observações
filósofo	*Classe Filósofo*	1 (obrigatório)	
requisiçãoId	*String*	1 (obrigatório)	

Tabela 24 – Estudo de Caso – Descrição do conceito Lugar.

Ação: Informar

Nome	Tipo	Cardinalidade	Observações
filósofo	*Classe Filósofo*	1 (obrigatório)	

Tabela 25 – Estudo de Caso – Descrição da ação Informar.

Ação: Requerer

Nome	Tipo	Cardinalidade	Observações
filósofo	*Classe Filósofo*	1 (obrigatório)	
necessidade	*Classe Necessidade*	1 (obrigatório)	
requisiçãoId	*String*	1 (obrigatório)	

Tabela 26 – Estudo de Caso – Descrição da ação Requerer.

A Figura 28 apresenta a tela do *Protégé* que define a ação Requerer.

[1] Identificação da requisição.

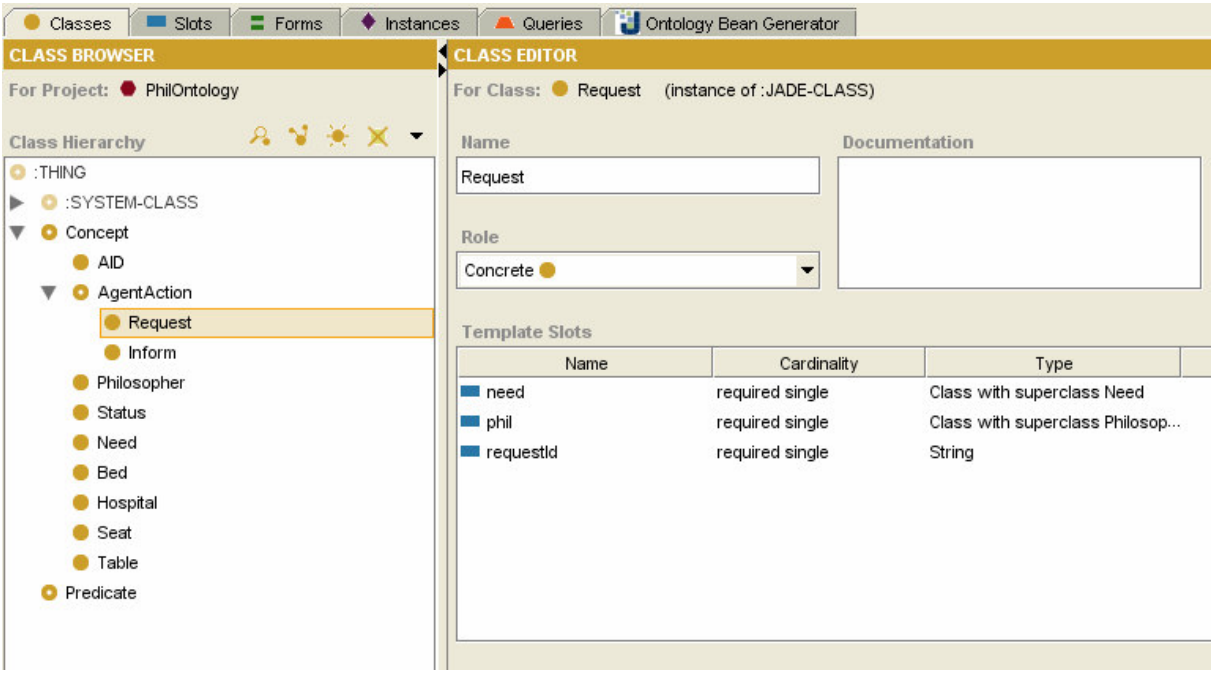

Figura 28 – Estudo de Caso – Tela do *Protégé* para definição da ação Requerer.

7.4.2. Metodologia Gaia – Construção dos Modelos da Fase de Análise

Utilizou-se um editor de texto comum, já que não foi encontrada nenhuma ferramenta que suportasse a notação da metodologia.

Modelo de Papéis

As Tabelas 27 a 30 descrevem, respectivamente, os modelos referentes aos papéis Filósofo, Mesa, Hospital e Gerente.

Papel	**Filósofo**		
Descrição	Pensa, come, adoece e morre.		
Protocolos e Atividades	Pensa, Come, Trata, InformaEnergia, SolicitaAcessoMesa, RespondeAcessoMesa, SolicitaSaidaMesa, RespondeSaidaMesa, SolicitaAcessoHospital, RespondeAcessoHospital, SolicitaSaidaHospital, RespondeSaidaHospital		
Permissões	lê	energia	// quantidade de energia
	fornecido	autoMesa	// autorização acesso à mesa
	fornecido	autoHospital	// autorização acesso ao hospital
	fornecido	autoSaiMesa	// autorização saída da mesa
	fornecido	autoSaiHospital	// autorização saída do hospital
	gera	energia	// quantidade de energia
Responsabilidades			
Sobrevivência	FILÓSOFO	=	(VIDA || InformaEnergia)$^{+}$
	VIDA	=	NO_TRABALHO | NA_MESA | NO_HOSPITAL
	NO_TRABALHO	=	Pensa. [(SolicitaAcessoMesa. [RespondeAcessoMesa]) | (SolicitaAcessoHospital. [RespondeAcessoHospital])
	NA_MESA	=	Come. [SolicitaSaidaMesa. [RespondeSaidaMesa]]
	NO_HOSPITAL	=	Trata. [SolicitaSaidaHospital. [RespondeSaidaHospital]]
Segurança	• energia > 0		

Tabela 27 - Estudo de Caso - Modelo de Papéis - Filósofo.

Papel	**Mesa**		
Descrição	Controla ocupação da mesa.		
Protocolos e Atividades	SolicitaEstadoMesa, RespondeEstadoMesa, OcupaVagaMesa, DesocupaVagaMesa		
Permissões	lê	qtdVagaMesa	// quantidade de vagas na mesa
	gera	qtdVagaMesa	// quantidade de vagas na mesa
Responsabilidades			
Sobrevivência	MESA	=	(PEDIDO || OCUPACAO)$^{\infty}$
	PEDIDO	=	SolicitaEstadoMesa. RespondeEstadoMesa
	OCUPACAO	=	OcupaVagaMesa | DesocupaVagaMesa
Segurança	• verdadeiro		

Tabela 28 - Estudo de Caso - Modelo de Papéis - Mesa.

Papel	**Hospital**		
Descrição	Controla ocupação do hospital.		
Protocolos e Atividades	SolicitaEstadoHospital, RespondeEstadoHospital, OcupaVagaHospital, DesocupaVagaHospital		
Permissões	lê	qtdVagaHospital	// quantidade de vagas na hospital
	gera	qtdVagaHospital	// quantidade de vagas na hospital
Responsabilidades			
Sobrevivência	HOSPITAL	=	(PEDIDO ‖ OCUPACAO)$^{\infty}$
	PEDIDO	=	SolicitaEstadoHospital. RespondeEstadoHospital
	OCUPACAO	=	OcupaVagaHospital ǀ DesocupaVagaHospital
Segurança	• verdadeiro		

Tabela 29 - Estudo de Caso - Modelo de Papéis - Hospital.

Papel	**Gerente**			
Descrição	Otimiza o bem-estar dos filósofos.			
Protocolos e Atividades	SolicitaAcessoMesa, SolicitaEstadoMesa, RespondeEstadoMesa, AvaliaAcessoMesa , RespondeAcessoMesa, OcupaVagaMesa, SolicitaSaidaMesa, AvaliaSaidaMesa, RespondeSaidaMesa, DesocupaVagaMesa, SolicitaAcessoHospital, SolicitaEstadoHospital, RespondeEstadoHospital, AvaliaAcessoHospital, RespondeAcessoHospital, OcupaVagaHospital, SolicitaSaidaHospital, AvaliaSaidaHospital, RespondeSaidaHospital, DesocupaVagaHospital, ExibeEstadosFilosofos, InformaEnergia			
Permissões	lê	fornecido	energia	// quantidade de energia
		fornecido	qtdVagaMesa	// quantidade de vagas na mesa
		fornecido	qtdVagaHospital	// quantidade de vagas no hospital
	gera		autoMesa	// autorização acesso à mesa
			autoHospital	// autorização acesso ao hospital
			autoSaiMesa	// autorização saída da mesa
			autoSaiHospital	// autorização saída do hospital
Responsabilidades				
Sobrevivência	GERENTE		=	(InformaEnergia ‖ ExibeEstadosFilosofos ‖ [PEDIDO])$^{\infty}$
	PEDIDO		=	PARA_MESA ǀ PARA_HOSPITAL ǀ DA_MESA ǀ DO_HOSPITAL
	PARA_MESA		=	SolicitaAcessoMesa. SolicitaEstadoMesa. RespondeEstadoMesa. AvaliaAcessoMesa. [OcupaVagaMesa]. RespondeAcessoMesa
	PARA_HOSPITAL		=	SolicitaAcessoHospital. SolicitaEstadoHospital. RespondeEstadoHospital. AvaliaAcessoHospital. [OcupaVagaHospital]. RespondeAcessoHospital
	DA_MESA		=	SolicitaSaidaMesa. SolicitaEstadoMesa. RespondeEstadoMesa. AvaliaSaidaMesa. [DesocupaVagaMesa]. RespondeSaidaMesa
	DO_HOSPITAL		=	SolicitaSaidaHospital. SolicitaEstadoHospital. RespondeEstadoHospital. AvaliaSaidaHospital. [DesocupaVagaHospital]. RespondeSaidaHospital.
Segurança	• verdadeiro			

Tabela 30 - Estudo de Caso - Modelo de Papéis - Gerente.

Modelo de Interações

As Tabelas 31 a 41 descrevem o Modelo de Interações para o estudo de caso.

Nome do Protocolo: **InformaEnergia**		
Iniciador: **Filósofo**	Contraparte: **Gerente**	Entrada: **nível atual de energia**
Descrição: Informa ao Gerente o nível atual de energia do Filósofo		Saída: --

Tabela 31 - Estudo de Caso - Modelo de Interações - InformaEnergia.

Nome do Protocolo: **SolicitaAcessoMesa**		
Iniciador: **Filósofo**	Contraparte: **Gerente**	Entrada: **nível atual de energia** **estado do Filósofo**
Descrição: Quando o nível de energia cai para valores abaixo do nível de fome, solicita acesso à mesa ("pede acesso à comida")		Saída: **RespondeAcessoMesa**

Tabela 32 - Estudo de Caso - Modelo de Interações - SolicitaAcessoMesa.

Nome do Protocolo: **SolicitaSaidaMesa**		
Iniciador: **Filósofo**	Contraparte: **Gerente**	Entrada: **nível de energia atual** **estado do Filósofo**
Descrição: Quando o nível de energia retorna a valores acima da metade da distância entre o nível de doença e o nível de fome em função da ingestão de alimentos (acesso à mesa concedido), solicita saída da mesa		Saída: **RespondeSaidaMesa**

Tabela 33 - Estudo de Caso - Modelo de Interações - SolicitaSaidaMesa.

Nome do Protocolo: **SolicitaAcessoHospital**		
Iniciador: **Filósofo**	Contraparte: **Gerente**	Entrada: **nível de energia atual** **estado do Filósofo**
Descrição: Quando o nível de energia cai para valores abaixo do nível de doença, solicita acesso ao hospital ("pede para ser tratado")		Saída: **RespondeAcessoHospital**

Tabela 34 - Estudo de Caso - Modelo de Interações - SolicitaAcessoHospital.

Nome do Protocolo: **SolicitaSaidaHospital**		
Iniciador: **Filósofo**	Contraparte: **Gerente**	Entrada: **nível de energia atual** **estado do Filósofo**
Descrição: Quando o nível de energia retorna a valores acima do nível de doença em função do tratamento médico (acesso ao hospital concedido), solicita saída do hospital		Saída: **RespondeSaidaHospital**

Tabela 35 - Estudo de Caso - Modelo de Interações - SolicitaSaidaHospital.

Nome do Protocolo: **SolicitaEstadoMesa**		
Iniciador: **Gerente**	Contraparte: **Mesa**	Entrada: **SolicitaAcessoMesa ou SolicitaSaidaMesa**
Descrição: Em função de um pedido de acesso ou de saída, solicita estado de ocupação da mesa		Saída: **RespondeEstadoMesa**

Tabela 36 - Estudo de Caso - Modelo de Interações - SolicitaEstadoMesa.

Nome do Protocolo: **OcupaVagaMesa**		
Iniciador: **Gerente**	Contraparte: **Mesa**	Entrada: **SolicitaAcessoMesa** **existência de vaga na mesa** **decisão de permitir acesso à mesa**
Descrição: Em função de solicitação de acesso, de existência de vaga e por outros critérios de decisão, ocupa vaga na mesa		Saída: **--**

Tabela 37 - Estudo de Caso - Modelo de Interações - OcupaVagaMesa.

Nome do Protocolo: **DesocupaVagaMesa**		
Iniciador: **Gerente**	Contraparte: **Mesa**	Entrada: **SolicitaSaidaMesa** **decisão de permitir saída da mesa**
Descrição: Em função de solicitação de saída e de outros critérios de decisão, desocupa vaga na mesa		Saída: **--**

Tabela 38 - Estudo de Caso - Modelo de Interações - DesocupaVagaMesa.

Nome do Protocolo: **SolicitaEstadoHospital**		
Iniciador: **Gerente**	Contraparte: **Hospital**	Entrada: **SolicitaAcessoHospital ou SolicitaSaidaHospital**
Descrição: Em função de um pedido de acesso ou de saída, solicita estado de ocupação do hospital		Saída: **RespondeEstadoHospital**

Tabela 39 - Estudo de Caso - Modelo de Interações - SolicitaEstadoHospital.

Nome do Protocolo: **OcupaVagaHospital**		
Iniciador: **Gerente**	Contraparte: **Hospital**	Entrada: **SolicitaAcessoHospital** **existência de vaga no hospital** **decisão de permitir acesso ao hospital**
Descrição: Em função de solicitação de acesso, de existência de vaga e por outros critérios de decisão, ocupa vaga no hospital		Saída: **--**

Tabela 40 - Estudo de Caso - Modelo de Interações - OcupaVagaHospital.

Nome do Protocolo: **DesocupaVagaHospital**		
Iniciador: **Gerente**	Contraparte: **Hospital**	Entrada: **SolicitaSaidaHospital** **decisão de permitir saída do hospital**
Descrição: Em função de solicitação de saída e de outros critérios de decisão, desocupa vaga no hospital		Saída: **--**

Tabela 41 - Estudo de Caso - Modelo de Interações - DesocupaVagaHospital.

7.4.3. Metodologia Gaia – Construção dos Modelos da Fase de *Design*

Modelo de Agentes

A Figura 29 exibe o Modelo de Agentes para o estudo de caso.

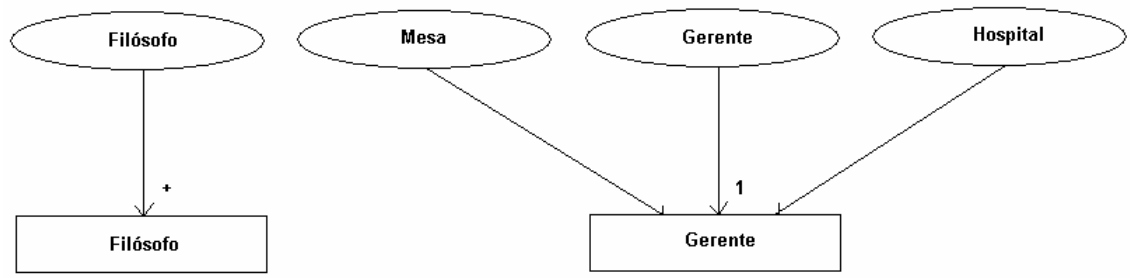

Figura 29 – Estudo de Caso – Modelo de Agentes.

Modelo de Serviços

A Tabela 42 apresenta o Modelo de Serviços para o estudo de caso.

Modelo de Conhecimentos

A Figura 30 exibe o Modelo de Conhecimentos para o estudo de caso.

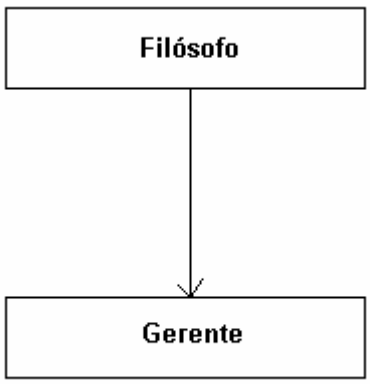

Figura 30 – Estudo de Caso – Modelo de Conhecimentos.

Serviço	Entradas	Saídas	Pré-condições	Pós-condições
Avaliar acesso à mesa	energias, estados e ocupação da mesa	autoMesa	solicitação de acesso à mesa	autoMesa ≠ nulo
Avaliar acesso ao hospital	energias, estados e ocupação do hospital	autoHospital	solicitação de acesso ao hospital	autoHospital ≠ nulo
Avaliar saída da mesa	energias, estados e ocupação da mesa	autoSaiMesa	solicitação de saída da mesa	autoSaiMesa ≠ nulo
Avaliar saída do hospital	energias, estados e ocupação do hospital	autoSaiHospital	solicitação de saída do hospital	autoSaiHospital ≠ nulo
Comer	energia	energia	energia > nivelDeDoenca	energia = energia + 1
Desocupar vaga na mesa	--	--	autoSaiMesa = verdadeiro	qtdVagaMesa = qtdVagaMesa + 1
Desocupar vaga no hospital	--	--	autoSaiHospital = verdadeiro	qtdVagaHospital = qtdVagaHospital + 1
Exibir estados dos filósofos	energias e estados	--	verdadeiro	verdadeiro
Informar nível de energia	energia	--	energia ≥ 0	verdadeiro
Ocupar vaga na mesa	--	--	autoMesa = verdadeiro	qtdVagaMesa = qtdVagaMesa – 1
Ocupar vaga no hospital	--	--	autoHospital = verdadeiro	qtdVagaHospital = qtdVagaHospital – 1
Pensar	energia	energia	energia > 0	energia = energia – 1
Solicitar acesso à mesa	energia	autoMesa	energia ≤ nivelDeFome	autoMesa ≠ nulo
Solicitar acesso ao hospital	energia	autoHospital	energia ≤ nivelDeDoenca	autoHospital ≠ nulo
Solicitar estado da mesa	--	qtdVagaMesa	solicitação de acesso ou saída da mesa	qtdVagaMesa ≠ nulo
Solicitar estado do hospital	--	qtdVagaHospital	solicitação de acesso ou saída do hospital	qtdVagaHospital ≠ nulo
Solicitar saída da mesa	energia	autoSaiMesa	energia > nivelInterm	autoSaiMesa ≠ nulo
Solicitar saída do hospital	energia	autoSaiHospital	energia > nivelDeDoenca	autoSaiHospital ≠ nulo
Tratar	energia	energia	energia > 0	energia = energia + 1

Tabela 42 – Estudo de Caso – Modelo de Serviços.

7.4.4. Implementação

Todo o estudo de caso foi implementado com o auxílio do *framework* JADE.

Para implementação dos agentes do tipo Filósofo foi considerada a máquina de estados apresentada na Figura 31.

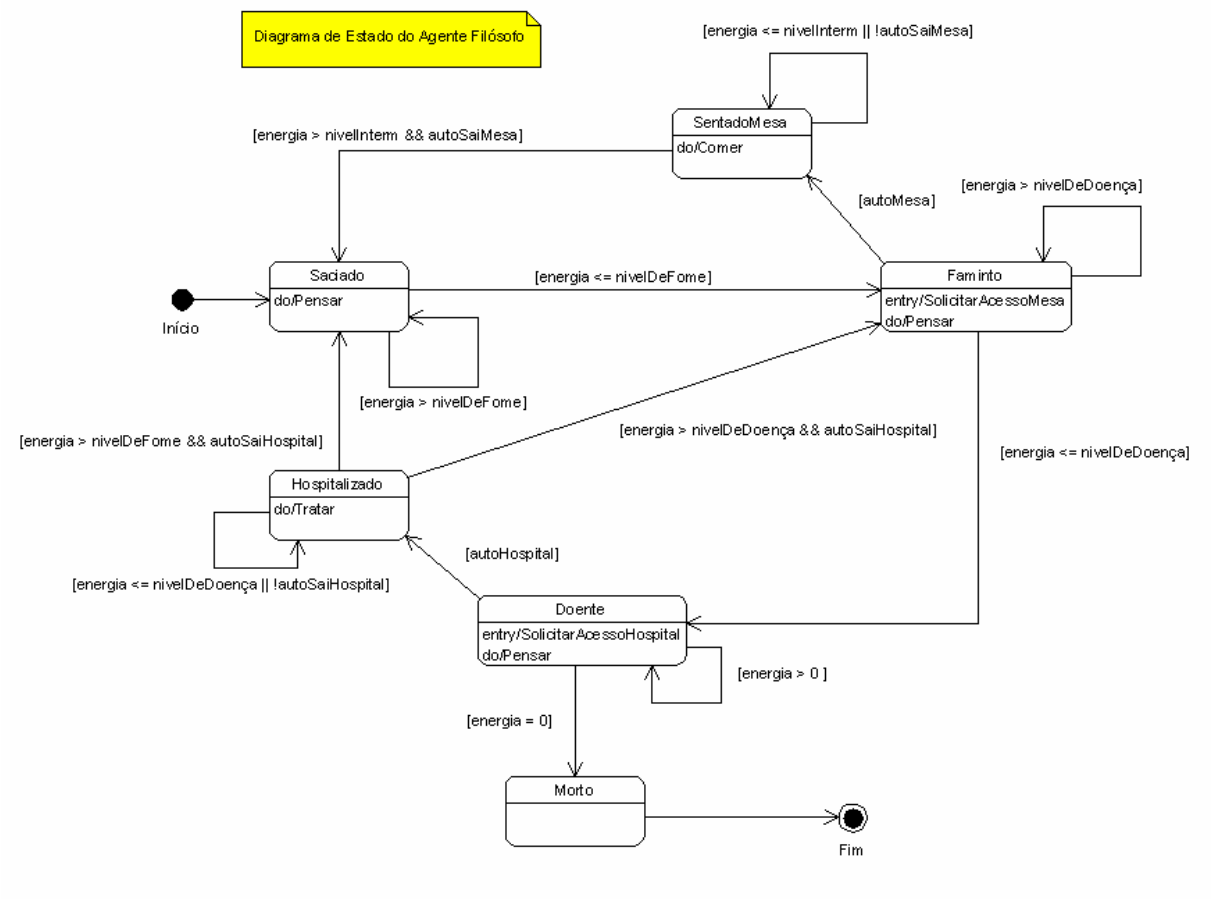

Figura 31 - Estudo de Caso - Diagrama de Estados - Filósofo.

Alguns detalhes do código serão apresentados a seguir.

Distribuição dos Arquivos-Fonte

A Tabela 43 apresenta a distribuição dos arquivos entre os pacotes.

Pacote	Descrição
phil	Classes principais. Filósofos, Gerente e Matriz.
onto	Classes relacionadas à ontologia.
util	Classes úteis.

Tabela 43 - Estudo de Caso - Distribuição dos arquivos-fonte.

Registro do *Directory Facilitator* (DF)

A Tabela 44 apresenta o código referente ao registro do agente Gerente no DF. Por questões de simplicidade, optou-se por registrar um único serviço do tipo "*manager*".

```
// register in DF
ServiceDescription sd  = new ServiceDescription();
sd.setType("manager");
sd.setName(getLocalName());
```

```
DFAgentDescription dfd = new DFAgentDescription();
dfd.setName(getAID());

try {
        DFAgentDescription list[] = DFService.search(this,dfd);
        if(list.length>0) {
                DFService.deregister(this);
        }

        dfd.addServices(sd);
        DFService.register(this,dfd);
} catch (FIPAException fe) {
        fe.printStackTrace();
}
```

Tabela 44 - Estudo de Caso - Registro no DF - Gerente.

A Tabela 45 apresenta o código relativo à busca no DF pelo serviço "*manager*" realizada pelo agente do tipo Filósofo.

```
// search manager agent in DF
DFAgentDescription dfdf = new DFAgentDescription();
ServiceDescription sdf = new ServiceDescription();
sdf.setType("manager");
dfdf.addServices(sdf);

try {
        DFAgentDescription[] result = DFService.search(this,dfdf);
        manager = result[0].getName() ;
} catch (FIPAException fe) {
        fe.printStackTrace();
}
```

Tabela 45 - Estudo de Caso - Busca pelo serviço *manager*.

A Tabela 46 apresenta a implementação da máquina de estados do agente tipo Filósofo implementado através da classe *FSMBehaviour* do JADE.

```
// create complex fsm behavior
FSMBehaviour fsm = new FSMBehaviour(this) {
        public int onEnd() {
                myAgent.doDelete();
                return super.onEnd();
        }
};

// register state FULL (first state)
fsm.registerFirstState(new Full(), "FULL");
```

```
// register state HUNGRY
fsm.registerState(new Hungry(), "HUNGRY");

// register state EATING
fsm.registerState(new Eating(), "EATING");

// register state ILL
fsm.registerState(new Ill(), "ILL");

// register state ON_HOSPITAL
fsm.registerState(new OnHospital(), "ON_HOSPITAL");

// register state DEAD (last state)
fsm.registerLastState(new Dead(), "DEAD");

// register transitions
fsm.registerTransition("FULL", "FULL", 0);
fsm.registerTransition("FULL", "HUNGRY", 1);

fsm.registerTransition("HUNGRY", "HUNGRY", 0);
fsm.registerTransition("HUNGRY", "EATING", 1);
fsm.registerTransition("HUNGRY", "ILL", 2);

fsm.registerTransition("EATING", "EATING", 0);
fsm.registerTransition("EATING", "FULL", 1);
fsm.registerTransition("EATING", "HUNGRY", 2);

fsm.registerTransition("ILL", "ILL", 0);
fsm.registerTransition("ILL", "ON_HOSPITAL", 1);
fsm.registerTransition("ILL", "DEAD", 2);

fsm.registerTransition("ON_HOSPITAL", "ON_HOSPITAL", 0);
fsm.registerTransition("ON_HOSPITAL", "HUNGRY", 1);
fsm.registerTransition("ON_HOSPITAL", "FULL", 2);

// add fsm behaviour
addBehaviour(fsm);
```

Tabela 46 - Estudo de Caso - Implementação da máquina de estados - Filósofo.

Finalmente, a Tabela 47 apresenta parte do algoritmo inteligente encarregado de gerenciar as concessões para acesso à comida ou tratamento ou, ainda, para saída da mesa ou do hospital.

```
public boolean requisitionAccepted(Request request,String cid) {
//-------------------------------------------------------------
int test=0;
boolean found=false;
```

```
int i=0;
test = (int) (Math.random() * 100);

// process need FOOD
//------------------
if(request.getNeed().getNeedName().equals("FOOD")) {

        // look for phil at the hospital and remove it
        found=false;
        i=0;

        while(i<hospital.getBeds().size() && !found) {
                Bed bed=(Bed)hospital.getBeds().get(i);
                found=request.getPhil().getId().equals
                        (bed.getPhil().getId());

                if(found) {
                        bed=(Bed)hospital.getBeds().remove(i);
                        hospital.setVacancy(hospital.getVacancy() + 1);
                }
                i++;
        }

        // log seats occupied
        int x=0;

        while(x<table.getSeats().size()) {
                Seat seat=(Seat)table.getSeats().get(x);
                System.out.println(getLocalName() + "-> TABLE (" +
                        cid + "): " +
                        "seat #" + x + ": " + seat.getPhil().getName() + "
                        (" + seat.getRequestId() + ")");
                x++;
        }

        // no vacancy
        if(table.getVacancy()<=0) {
                return false;
        }

        // nobody is hungry or via probability
        if(matrix.getTotHungry()<=0 || test<CHANCE_FOOD) {

                found=false;
                i=0;

                while(i<table.getSeats().size() && !found) {
                        Seat seat=(Seat)table.getSeats().get(i);
                        found=request.getPhil().getId().equals
                                (seat.getPhil().getId());
```

```
                    i++;
                }

                // phil isn't at the table
                if(!found) {
                    Seat seat=new Seat();
                    seat.setPhil(request.getPhil());
                    seat.setRequestId(request.getRequestId());
                    table.getSeats().add(seat);
                    table.setVacancy(table.getVacancy() - 1);
                    return true;
                }
            }
            return false;
        }
```

Tabela 47 - Estudo de Caso - Algoritmo inteligente.

7.4.5. Resultados Obtidos

A arquitetura apresentada mostrou-se bastante eficiente e eficaz para a solução do problema proposto. As Figuras 32 e 33 mostram o sistema em execução.

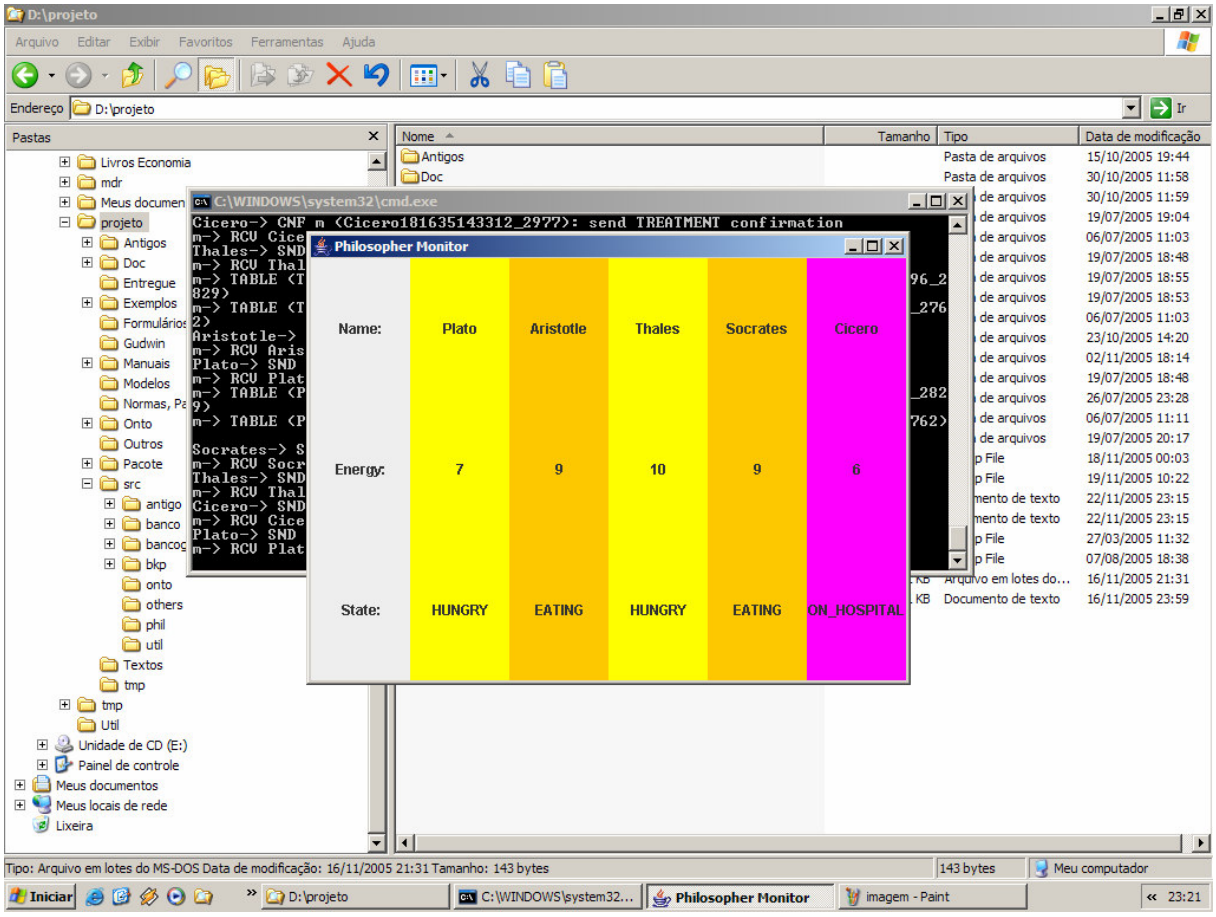

Figura 32 - Estudo de Caso - Sistema em execução.

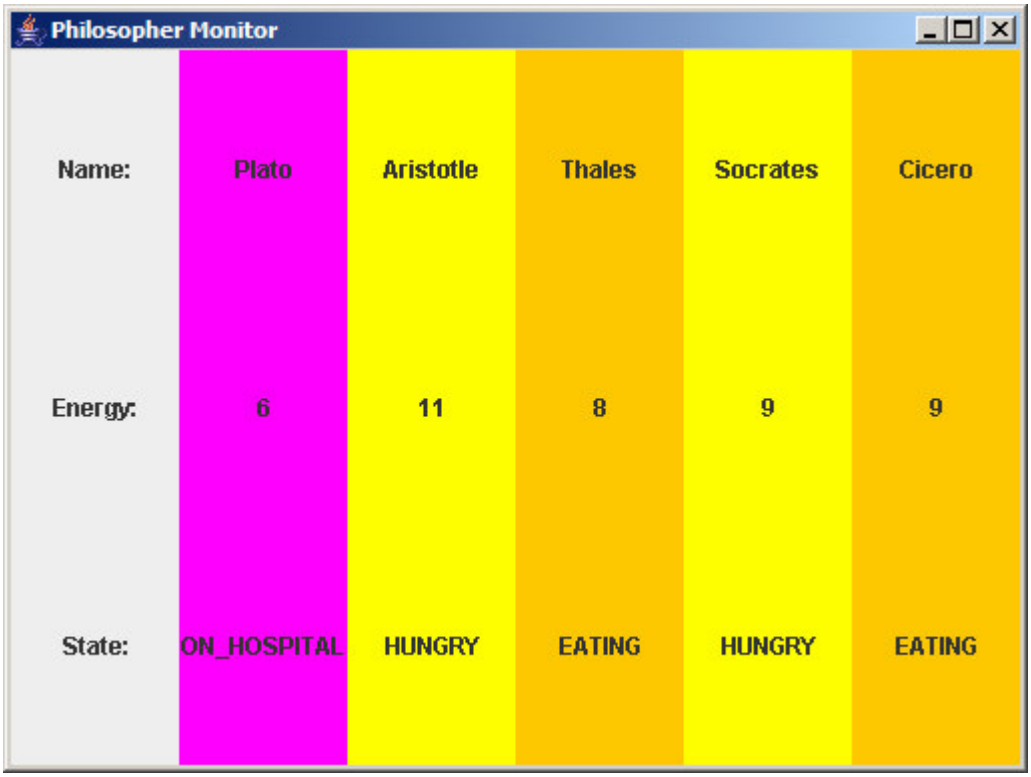

Figura 33 - Estudo de Caso - Sistema em execução (zoom).

A Tabela 48 apresenta um resumo dos testes realizados.

Teste #	Duração	# Mortos	Observações
1	5 min	0	
2	22 min	0	
3	31 min	0	
4	42 min	1	Diversos aplicativos abertos.
5	45 min	0	
6	72 min	1	
7	10 min	0	Máquina conectada à rede local.
8	25 min	1	Máquina conectada à rede local.
9	53 min	2	Máquina conectada à rede local.
10	62 min	1	Máquina conectada à rede local.

Tabela 48 - Estudo de Caso - Resumo dos testes realizados.

7.5. Resumo

Este capítulo apresentou:

- Uma descrição do problema proposto como estudo de caso;
- O processo de desenvolvimento utilizado para implementar o estudo de caso, exibindo os principais modelos, diagramas e códigos.

- Um resumo dos testes realizados para comprovação da eficiência e eficácia da solução proposta.

8. Conclusões e Trabalhos Futuros

8.1. Conclusões

A abordagem multiagentes é capaz resolver, de modo simplificado, uma gama de problemas cuja solução através da abordagem tradicional exibe características extremamente mais complexas ou mesmo inviáveis. O escopo da tecnologia engloba, principalmente, problemas tipicamente distribuídos, que envolvem negociações, que ultrapassam fronteiras organizacionais e/ou que envolvem entidades inerentemente autônomas.

A construção de ontologias é extremamente importante na viabilização da interação entre os agentes de *software*. A utilização de uma ferramenta de suporte é imprescindível ao processo de criação desses artefatos. Destaca-se, neste contexto, a ferramenta *Protégé*, desenvolvida como um projeto *open-source* pela *Stanford University*.

Como todo processo de Engenharia de *Software*, a construção de SMA deve ser suportada por um processo de desenvolvimento e uma linguagem de modelagem capaz de expressar os diversos níveis de abstração necessários à criação do sistema. A metodologia Gaia, a despeito das diversas metodologias disponíveis no mercado, é simples, direta e abrange todas as etapas do processo de desenvolvimento. Nenhuma ferramenta de suporte à metodologia foi encontrada durante a pesquisa. A ausência deste tipo de ferramenta é um dos poucos pontos negativos imputáveis à Gaia.

O JADE é uma excelente ferramenta para desenvolvimento de sistemas baseados em agentes. Fornece toda a infra-estrutura necessária, tais como ferramentas de suporte e depuração e farta documentação. Trata-se, ainda por cima, de um projeto *open-source* o que garante a independência em relação a fornecedores de *software*. Todas as alterações e evoluções são controladas pelo TILAB e a comunidade JADE influencia diretamente nas decisões. O JADE é aderente às especificações FIPA, o que garante ampla interoperabilidade com sistemas desenvolvidos através de outras ferramentas. É possível notar a facilidade com que se trabalha como *framework* JADE, possibilitando aumento da produtividade e redução da probabilidade de erros de implementação. Outro diferencial na ferramenta está no fato de ter sido totalmente implementado em Java, o que garante a independência em relação ao sistema operacional em uso.

8.2. Sugestões para Trabalhos Futuros

Análise e *Design*

Um ponto importante seria a construção de uma ferramenta para suporte à metodologia Gaia. Dentre os aspectos fundamentais da ferramenta a ser desenvolvida é possível citar: a criação de uma linguagem de modelagem, a possibilidade de realização de críticas cruzadas entre os diversos diagramas e modelos e a possibilidade de exportação dos modelos para diversos formatos de imagem e texto.

Outra sugestão consistiria na investigação mais profunda de outras metodologias de modelagem SMA, como o MESSAGE. A favor desta encontra-se a utilização da linguagem UML, extremamente difundida na comunidade de desenvolvedores.

Ontologia

A integração entre a ferramenta de construção de ontologias e o *framework* de desenvolvimento de SMA é um ponto importante no processo de construção deste tipo de sistema. Uma investigação mais detalhada sobre *plug-ins* para a ferramenta *Protégé* que permitam a geração automática de código a partir do modelo da ontologia é de suma relevância neste contexto. Uma pesquisa mais aprofundada a respeito de outras ferramentas de construção de ontologias também é um caminho interessante.

Implementação

Em relação à implementação é possível sugerir um trabalho de pesquisa objetivando traçar um paralelo entre os diversos *frameworks* de desenvolvimento de SMA disponíveis no mercado, expondo vantagens e desvantagens de cada uma.

No tocante à implementação em JADE, é importante pesquisar a utilização de protocolos padronizados de interação entre agentes. A FIPA possui listados e documentados uma série de protocolos que regem as interações mais comuns possíveis entre agentes. O próprio JADE implementa, nativamente, diversos destes protocolos. Talvez, e muito possivelmente, a utilização de tais protocolos permitiria uma agilidade maior aos processos de interação, além de tornar o código um pouco mais claro.

No que se refere à inteligência do sistema, é possível utilizar técnicas avançadas de IA de modo a proporcionar resultados melhores em comparação aos obtidos no contexto deste trabalho. Um ponto crucial localiza-se no fato de que o sistema, tal qual implementado, não armazena nem as decisões e muito menos os impactos destas decisões no resultado passado. O

registro histórico permitiria ao agente tomador de decisão uma avaliação mais segura a respeito do melhor fluxo de ação a seguir.

Referências

[BELLIFEMINE, 2003] BELLIFEMINE, Fabio et al. **Jade**: A White Paper. 2003. Disponível em: <http://exp.telecomitalialab.com>. Acesso em: 26 maio 2005.

[BELLIFEMINE, 2005] BELLIFEMINE, Fabio et al. **JADE Programmer´s Guide**. 2005. Disponível em: <http://jade.tilab.com>. Acesso em: 27 maio 2005.

[CAIRE, 2004] CAIRE, Giovanni; CABANILLAS, David. **JADE Tutorial** Application-Defined Content Languages and Ontologies. 2005. Disponível em: <http://jade.tilab.com>. Acesso em: 30 set. 2005.

[DIJKSTRA, 1971] DIJKSTRA, E. W. **Hierarchical Ordering of Sequential Processes**. 1971. Disponível em: <http://www.cs.utexas.edu/users/EWD/index03xx.html>. Acesso em: 30 jul. 2005.

[ELAMY, 2005] ELAMY, A. Halim. **Perspectives in Agents-Based Technology**. 2005. Disponível em: <http://www.agentlink.com>. Acesso em: 10 set. 2005.

[FERREIRA, 2004] FERREIRA, Aurélio Buarque de Holanda. **Miniaurélio**: o dicionário da língua portuguesa. 6. ed. rev. atual. Curitiba: Positivo, 2004.

[FIPA, 2005] FIPA Web Site. [2005] Disponível em: <http://www.fipa.org>. Acesso em: 20 maio 2005.

[FRANKLIN, 1996] FRANKLIN, Stan; GRAESSER, Art. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. **Springer-Verlag**, Berlim, 1996.

[GOMES, 2000] GOMES, Antônio S. **Contribuições ao Estudo de Redes de Agentes**. 2000. 115 f. Dissertação (Mestrado em Engenharia Elétrica) – Universidade Estadual de Campinas, Campinas, 2000.

[ICA, 2005] PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO. Núcleo de Pesquisa em Inteligência Computacional Aplicada. **Cursos em Inteligência Computacional**: ICA-Book. [2005]. Disponível em: <http://www.puc-rio.br>. Acesso em: 20 maio 2005.

[JADE, 2005] JADE Web Site. [2005]. Disponível em: <http://jade.tilab.com>. Acesso em: 20 maio 2005.

[JENNINGS, 1996] JENNINGS, Nicholas R. Coordination Techniques for DAÍ. In: O'HARE, Greg; JENNINGS, Nicholas (Eds.). **Foundations of distributed artificial intelligence**. [S.1.]: John Wiley and Sons, 1996.

[JUCHEM, 2001] JUCHEM, Murilo; BASTOS, Ricardo Melo. **Engenharia de Sistemas Multiagentes**: Uma Investigação sobre o Estado da Arte. Technical Report Series, n.14. Porto Alegre: Campus Global, 2001.

[JUCHEM, 2001a] JUCHEM, Murilo; BASTOS, Ricardo Melo. **Arquitetura de Agentes**. Technical Report Series, n.13. Porto Alegre: Campus Global, 2001.

[KIMLAYCHUK, 2003] KIMLAYCHUK, V. 5 **Hungry philosophers' problem**: Modelling with JADE. 2003. Disponível em: <http://jade.tilab.com/papers/EXP/Vadim.pdf>. Acesso em: 1 ago. 2005.

[LARMAN, 2002] LARMAN, Craig. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2.ed. Upper Saddle River, NJ: Prentice-Hall, 2002.

[MASSONET, 2002] MASSONET, Philippe; DEVILLE, Yves; NÈVE, Cédric. **From AOSE Methodology to Agent Implementation**. Proceedings of AAMAS'02, Bologna, 2002.

[NOY, 2005] NOY, Natalya F.; McGUINNESS, Deborah L. **Ontology Development 101**: A Guide to Creating Your First Ontology. 2005. Disponível em: <http://protege.stanford.edu>. Acesso em: 01 nov. 2005.

[ODELL, 2002] ODELL, James. Objects and Agents Compared. **Journal of Object Technology**, Zurich, vol. 1, no. 1, p. 41-53, maio 2002.

[OLIVEIRA, 2001] OLIVEIRA, Toacy Cavalcante de. **Uma Abordagem Sistemática para a Instanciação de Frameworks Orientados a Objetos**. 2001. 166 f. Tese (Doutorado em Informática) – Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2001.

[PATACO, 2004] PATACO, Vera Lucia Paracampos; VENTURA, Magda Maria; RESENDE, Érica dos Santos. **Metodologia para Trabalhos Acadêmicos e Normas de Apresentação Gráfica**. Rio de Janeiro: Ed. Rio, 2004.

[PINTO, 2005] PINTO, Marcus Vinícius; BRAGA, José Luis. **Ambiente Multiagentes para Recuperação de Informação na Rede Municipal de Informática de Belo Horizonte**. [2005]. Disponível em: <http://www.ufmg.br>. Acesso em: 25 maio 2005.

[PLACCA, 2005] PLACCA, José Avelino. **Agentes Inteligentes e Sistemas Multiagentes**. Rio de Janeiro, [2005]. *Slides*.

[RUSSEL, 1995] RUSSEL, Stuart; NORVIG, Peter. **Artificial Intelligence**: A Modern Approach. Englewood Cliffs, NJ: Prentice Hall, 1995.

[SILVA, 2003] SILVA, Leonardo Ayres de Morais e Silva. **Estudo e Desenvolvimento de Sistemas Multiagentes usando JADE**: *Java Agent Development framework*. 2003. 96f. Monografia de Conclusão de Curso (Graduação em Informática) – Universidade de Fortaleza, Fortaleza, 2003.

[WOOLDRIDGE, 1995] WOOLDRIDGE, Michael; JENNINGS, Nicholas R. Intelligent Agents: Theory and Practice. **The Knowledge Engineering Review**, v.10, n.2, p.115-152, 1995.

[WOOLDRIDGE, 2000] WOOLDRIDGE, Michael; JENNINGS, Nicholas R.; KINNY, David. The Gaia Methodology for Agent-Oriented Analysis and Design. **Journal of Autonomous Agents and Multi-Agent Systems**, v.3, n.3, p. 285-312, 2000.

Apêndice A. Características Técnicas do *Framework JADE*

A.1. Instalação

A.1.1. Requisitos Mínimos

Como qualquer ferramenta implementada em Java, o JADE precisa da *Java Runtime Enviroment* (JRE) para executar. No tocante ao desenvolvimento é necessário também que o desenvolvedor possua instalado em seu equipamento o *Java Development Kit* (JDK). O JADE 3.3, versão corrente da ferramenta no momento da elaboração deste trabalho, demanda as versões 1.4 do JRE e JDK.

A.1.2. *Software* JADE

A ferramenta pode ser obtida através de seu *site* oficial no endereço *http://jade.tilab.com*. Para instalar basta descompactar os arquivos da ferramenta em qualquer pasta. A Tabela 49 exibe os arquivos compactados disponibilizados e suas descrições.

Arquivo	Descrição
JADE-bin	Os arquivos binários da ferramenta (JAR) para uso.
JADE-src	Código-fonte.
JADE-doc	Documentação completa, incluindo API e exemplos.
JADE-examples	Códigos-fonte dos exemplos de programas em JADE.
JADE-all	Todos os arquivos acima agrupados.

Tabela 49 - Arquivos do Pacote JADE.

Os arquivos binários (JAR), existentes na pasta *<JADE_HOME>/lib* após a descompressão, podem ser incluídos no *classpath* do sistema operacional a fim de facilitar a execução da ferramenta.

A.2. Execução do Ambiente

Para executar o JADE, utilizar a seguinte linha de comando:

java jade.Boot [opções] [agentes]

Na Figura 33, exibimos uma sintaxe completa descrita em EBNF da linha de comando para execução do JADE.

```
java jade.Boot Option* AgentSpecifier*

Option          - "-container"
                | "-host" HostName
                | "-port" PortNumber
                | "-name" PlatformName
                | "-gui"
                | "-mtp" ClassName "(" Argument* ")"
                  (";" ClassName "(" Argument* ")")*
                | "-nomtp"
                | "-aclcodec" ClassName (";" ClassName )*
                | "-nomobility"
                | "-version"
                | "-help"
                | "-conf" FileName?
                | "-" Keyword Value

ClassName       - PackageName? Word

PackageName     - (Word ".")+

Argument        - Word | Number | String

HostName        - Word ( "." Word )*

PortNumber      = Number

AgentSpecifier = AgentName ":" ClassName ("(" Argument* ")")?

AgentName       = Word

PlatformName    - Word

Keyword         - Word

Value           - Word
```

Figura 34 - EBNF - Linha de Comando jade (SILVA, 2003).

A.3. Ferramentas de Gerenciamento e Depuração

O pacote *java.tools* engloba uma série de classes que representam ferramentas que o JADE disponibiliza para desenvolvedores de agentes (SILVA, 2003).

A.3.1. *Remote Management Agent*

O *Remote Management Agent* (RMA) é um console gráfico para controle e gerenciamento da plataforma JADE. Pode ser carregado através de uma das seguintes linhas de comando:

java jade.Boot –gui

java jade.Boot meuComputador:jade.tools.rma.rma

A Figura 35 mostra a interface gráfica do RMA.

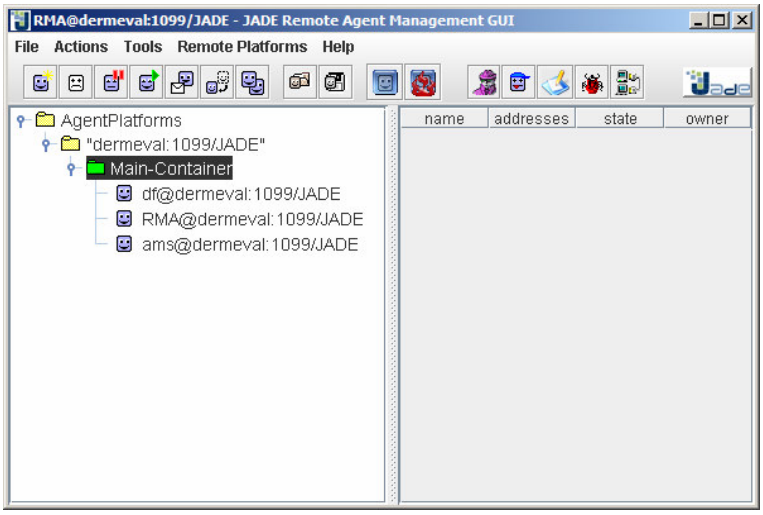

Figura 35 - Interface Gráfica do RMA.

A.3.2. *DummyAgent*

O *DummyAgent* é uma ferramenta gráfica de monitoramento e depuração de agentes desenvolvidos em JADE. Pode ser carregado através do RMA ou da seguinte linha de comando:

java jade.Boot da:jade.tools.DummyAgent.DummyAgent

A Figura 36 apresenta a interface gráfica do *DummyAgent*.

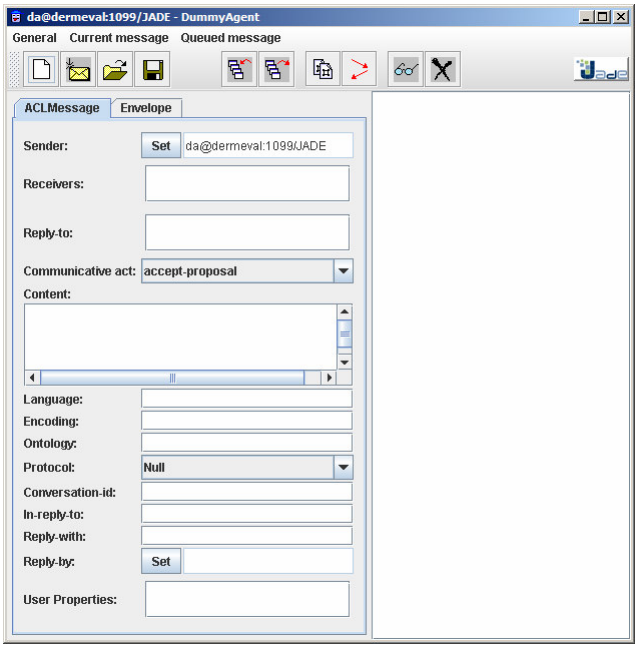

Figura 36 - Interface Gráfica do *DummyAgent*.

A.3.3. *SnifferAgent*

O *SnifferAgent* é uma ferramenta que monitora de forma gráfica a troca de mensagens entre determinados agentes. O monitoramento é exibido em notação semelhante aos diagramas de seqüências da UML. Quando um determinado agente é escolhido para o *"sniff"* toda comunicação realizada com esse agente é exibida na interface gráfica da ferramenta. O *SnifferAgent* pode ser iniciado através do RMA ou da seguinte linha de comando:

java jade.Boot sniffer:jade.tools.sniffer.Sniffer

A Figura 37 apresenta a interface gráfica do *SnifferAgent*.

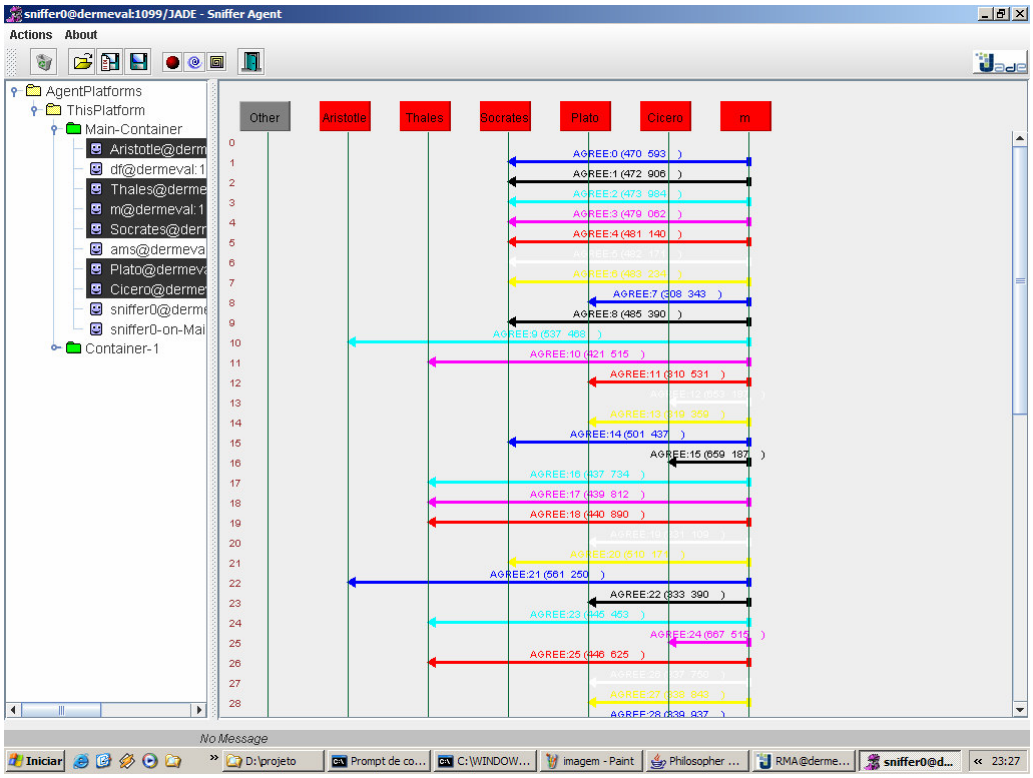

Figura 37 -Interface Gráfica do *SnifferAgent* (SILVA, 2003).

A.3.4. *Introspector Agent*

O *Introspector Agent* é uma ferramenta que permite monitorar o ciclo de vida de um agente, suas trocas de mensagens e seus comportamentos em execução. Permite, adicionalmente, o controle da execução do agente. Pode-se executar passo-a-passo ou lentamente. A ferramenta pode ser iniciada através do RMA ou pela seguinte linha de comando:

java jade.Boot intro:jade.tools.introspector.Introspector

A Figura 38 apresenta a interface gráfica do *IntrospectorAgent*.

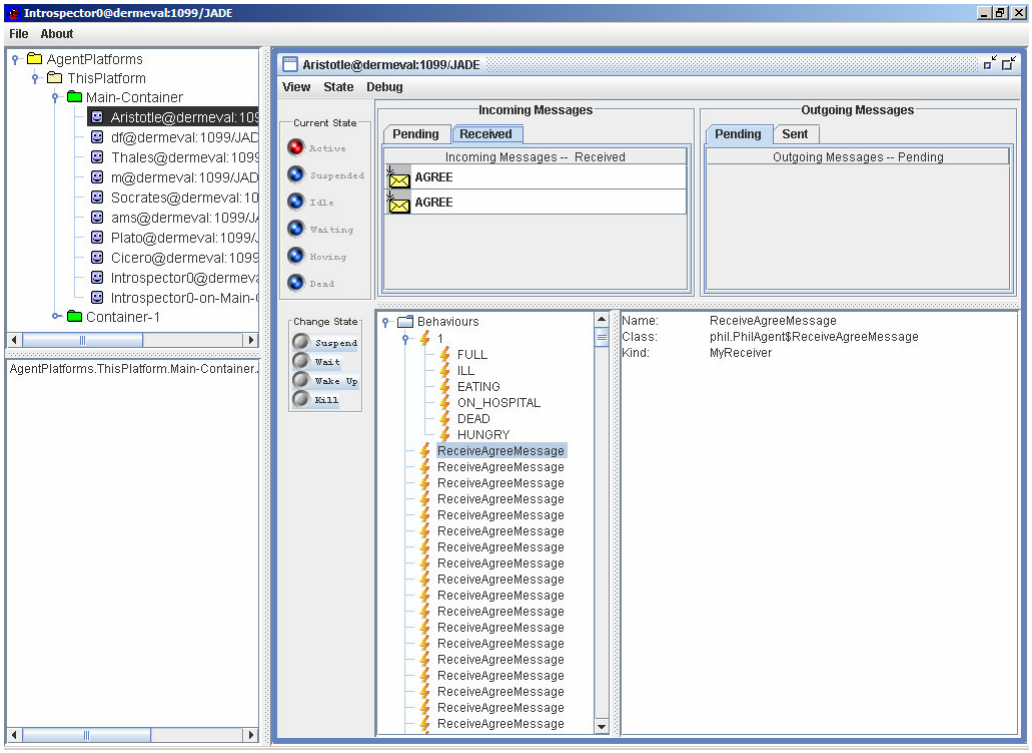

Figura 38 - Interface Gráfica do *IntrospectorAgent* (SILVA, 2003).

A.3.5. *DF GUI*

O *DF GUI* é uma ferramenta para interação com o *Directory Facilitator* (DF) do JADE. Através da ferramenta é possível criar uma complexa rede de domínios e sub-domínios de *yellow-pages*. Permite, também, registrar, cancelar registros, modificar e procurar agentes e serviços e, até mesmo, integrar DFs. Pode ser carregada através do RMA (SILVA, 2003).

A Figura 39 apresenta a interface gráfica do DF GUI.

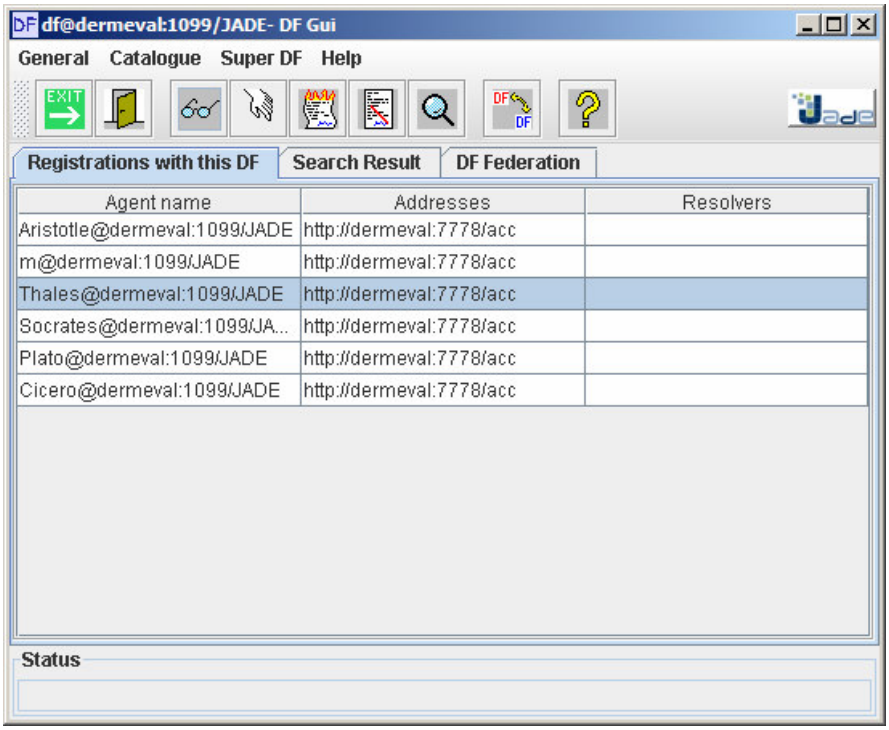

Figura 39 - Interface Gráfica do DF GUI (SILVA, 2003).