SemanticAgent, uma plataforma para desenvolvimento de agentes inteligentes.

Percival Lucena

Orientador: ***Prof. Dr.Dilvan de Abreu Moreira***

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Área de Ciência da Computação e Matemática Computacional.

" VERSÃO REVISADA APÓS A DEFESA"

Data da Defesa:	15/04/2003
Visto do Orientador:	

USP – São Carlos
Maio/2003

And though the face has changed
You're still the same

And it's a long long way
From where you want to be
And it's a long long road (....)

OMD - Pandora's Box

Everyday is a winding road
I get a little bit closer
Every day is a faded sign
I get a little bit closer to feeling fine

Sheryl Crow - Everyday is a winding road.

Agradecimentos

Aos meus pais e minhas irmãs pelo suporte financeiro e emocional nestes longos anos que eu passei em São Carlos. Ao me olhar no espelho hoje, pode ser difícil reconhecer aquele "*PerDeLucena*" que chegou em São Carlos em 1996. Muitas janelas se abriram para mim durante todos estes anos. Contudo, não deixei para trás os meus ideais e agradeço a vocês por terem me incentivado à acreditar.

A todos os professores da USP e em especial ao Professor Dilvan pela colaboração nestes últimos anos. Devo a vocês muito mais que minha formação profissional. Devo a vocês a confiança que tenho hoje que é possível alcançar os objetivos que estabelecemos.

As professoras Maria Carolina Monard e Solange Rezende do ICMC-USP, ao professor Ricardo Gudwin da FEEC-Unicamp e as mestres Elaine Silva e Flávia Linhalis pelas sugestões e correções dadas à este texto.

Aos meus amigos da USP São Carlos pelos momentos de diversão, stress e recompensas que dividimos: Daniel, Martin (McSiu), Edvaldo Föiemborra, Ivan RedMan, Andréia, Adriane, Samanta e todos os colegas da Sharp Software ("yes, we build data"). Não esquecendo o Mestre René, Mestre Murilo, Renata, D.G, Marquinhos, Talita, Farzin Bassiri, Zé Mário, Paul, Porradinha, Flávio, Bruna, Rijomar, Taboca, GusFaria, Kiko, Fábio Chino, Márcio, Wagner, Rodrigo, Juninho, Matheus, Billy e todas as pessoas com quem tive prazer de compartilhar idéias neste tempo. Sempre esqueço alguém. Sorry.

Aos meus colegas de laboratório com quem dividi dias e noites de trabalho, programação e small talk: Orlando, Gustavo, Carlos, Jean, Tanaka Da'ClusterMan.

Ao CNPQ pelo providencial suporte financeiro durante o período de mestrado.

That's all folks. Now back to work.

São Carlos-SP, Brasil, Maio de 2003

Resumo

Agentes inteligentes é um termo guarda-chuva que agrega diversas pesquisas no desenvolvimento de softwares autônomos que utilizam técnicas de Inteligência Artificial a fim de satisfazer metas estabelecidas por seus usuários. A construção de sistemas baseados em agentes inteligentes é uma tarefa complexa que envolve aspectos como comunicação entre agentes, planejamento, divisão de tarefas, coordenação, representação e manipulação de conhecimento e comportamentos, entre outras tarefas. Plataformas para agentes prevêem alguns serviços que permitem a desenvolvedores construir soluções sem a necessidade de se preocupar com todos detalhes da implementação.

Um novo modelo para criação de agentes chamado '*agentes atômicos'* é proposto com o objetivo de oferecer flexibilidade para o gerenciamento de conhecimento e implementação de comportamentos. A arquitetura *Agentes Semânticos* provê um framework para a implementação de tal modelo, oferecendo um conjunto de ferramentas para a criação de agentes inteligentes. Um protótipo de plataforma para agentes, baseado em tal arquitetura, foi desenvolvido em Java e permite a criação de aplicações capazes de processar linguagem natural restrita, manipular conhecimento e executar ações úteis.

Abstract

Intelligent Agents is an umbrella term that aggregates different research on the development of autonomous software that uses Artificial Intelligence techniques in order to satisfy user requests. The construction of systems based on intelligent agents is a complex task that involves aspects such as agent communication, planning, work division, cooperation, representation and manipulation of knowledge,among other activities. Agent Platforms provide some services that allow developers to build solutions without the need of worrying about every implementation detail.

A new model for creating agents, called '*atomic agents*', is proposed with the goal of offering flexible knowledge management and behavior implementation for constructing software agents. The *Semantic Agent Architecture* provides a framework for the implementation of such model, offering a set of tools for the creation of intelligent agents. A prototype *Agent Platform*, based on the architecture, was developed in Java and allows the creation of applications that are able to process restricted natural language, manipulate knowledge and execute useful actions.

Índice

Capítulo VI - Plataforma SemanticAgent

Capítulo VII - Conclusões

Lista de Acrônimos

ACL - Agent Communication Language

API - Application Programming Interface

BDI - Belief, Desire, Intention (Agent Architecture)

CORBA - Common Object Request Broker Architecture

DAML - DARPA Agent Markup Language

DTD - (XML) Document Type Definion

EKN - Extended Knowledge Network

FIPA - Foundation for Physical Intelligent Agents

FTP - File Transfer Protocol

GUI - Graphical User Interface

HTTP - Hypertext Transfer Protocol

IDE - Integrated Development Environment

IDL - Interface Definition Language

IIOP - (CORBA) Internet Inter-ORB Protocol

KL - Knowledge (Representation) Language

KQML - Knowledge Query and Manipulation Language

MAS - Multi Agent System

PLN - Processamento de Linguagem Natural

SMRP - Simple Multicast Routing Protocol

TCP/IP - Transmission Control Protocol / Internet Protocol

TT - Thought Treasure

UCL - Universal Communication Language

UML - Unified Modeling Language

UNL - Universal Network Language

UW - Universal Word

XML - Extensible Markup Language

Lista de Figuras e Tabelas

Heaven made us agents, free to good or ill.

The Cock and the Fox: or, the Tale of the Nun's Priest

Trans. by John Dryden, The Fables (1700)

Capitulo 1- Agentes Inteligentes

1.1- **Introdução**

Durante séculos, a humanidade se preocupou no campo da ficção e muitas vezes no campo da engenharia em criar agentes artificiais que imitassem o comportamento humano. Dispositivos tais como bonecas que choram, secretárias 'eletrônicas', pianos que tocam sozinhos, previam o advento de máquinas inteligentes. Durante quase todo o século XX, andróides, humanóides, robôs, cyborgs e computadores inteligentes foram personagens da ficção científica vistos como criaturas que existiriam em um futuro não tão distante **[Bradshaw 1996]**.

O computador HAL 9000 do famoso filme de ficção científica de 1968, *'2001, a space odyssey',* previa que um dia os computadores seriam inteligentes a ponto de possuírem emoções e vontade própria **[Clarke 1968]**. Muitos anos antes, o matemático Alan Turing, precursor da Ciência da Computação, já propunha o *teste de Turing,* segundo o qual máquinas poderiam ser consideradas inteligentes se um ser humano não pudesse distinguir o comportamento de uma máquina do comportamento de outro ser humano **[Turing 1950]**.

Entusiasmados pela possibilidade de criar máquina inteligentes, pesquisadores precursores do campo da Inteligência Artificial se reuniram na década de 1950 no *Dartmouth Summer Research Project* a fim de determinar as características necessárias para reproduzir a inteligência humana em computadores. Os computadores de então ainda estavam muito longe de oferecer o hardware necessário para implementar todas as camadas de abstração que dariam suporte, anos mais tarde, a inúmeras técnicas de representação e manipulação de conhecimento **[Reingold et al 1999]**.

Os resultados iniciais da pesquisa de máquinas inteligentes, que imitavam o comportamento humano, não ocorreram na medida do esperado pelos precursores iniciais da Inteligência Artificial. Apesar de todos os avanços na área realizados até hoje, o ano 2001 chegou sem que tivéssemos HAL 9000 e há algumas décadas a pesquisa de Inteligência Artificial se voltou para outros problemas mais específicos cujos resultados têm maior aplicabilidade para a sociedade.

Durante a década de 1990 começaram a surgir novas propostas para computação autônoma **[Negroponte 1995], [Nwana et al 1999].** Uma das idéias relacionadas, que passou a entrar em voga, foi o termo *agente,* uma entidade autônoma, tal qual um robô sem corpo, capaz de processar informações e agir em favor de um usuário.

O advento da Internet comercial e a explosão de informação oferecida pela World Wide Web a partir da segunda metade dos anos 1990 coincidiu com o interesse de se criar *agentes inteligentes* capazes de realizar tarefas para usuários aproveitando os recursos oferecidos pela rede. Esforços recentes de estruturação de informações e serviços como a Web Semântica **[Lee et al 2001]**, e os Web Services **[Sun MicroSystems 2002]** prometem permitir que agentes possam processar informações da Internet de maneira eficiente, concretizando, ao menos em parte, o paradigma de agentes de software. Espera-se que, em algum tempo, agentes inteligentes passem a fazer parte do quotidiano das pessoas como tantas outras aplicações computacionais existentes hoje.

Este projeto apresenta um protótipo de uma plataforma para o desenvolvimento de agentes chamada *SemanticAgent,* cuja proposta é facilitar a criação de agentes capazes de compreender solicitações em linguagem natural, manipular conhecimento e executar ações. Tem-se por objetivo avançar no estudo da criação de agentes cujas interfaces sejam mais naturais ao ser humano. Não se espera obter, a partir de um esforço tão limitado, um resultado tão ambicioso quanto o HAL 9000, mas espera-se ter dado mais um passo rumo a este objetivo.

1.2 - Agentes de Software

O termo agente, per se, engloba várias características que costumam ser encontradas em agentes de software. O dicionário Aurélio define um agente como sendo *"1. o que opera, agencia, age. 2. pessoa agente. 3. quem trata de negócios alheios"*. Um agente humano costuma ser um especialista em uma tarefa bem determinada que possui habilidades e conhecimentos que não temos, ou que não nos interessa usar. Um agente de viagem, por exemplo, possui informações sobre companhias aéreas, hotéis, atrações e características de várias cidades, regiões e países. Quando se contata um agente humano, costuma-se delegar tarefas para que este execute as decisões cabíveis. Evita-se assim, preocupações com detalhes da tarefa delegada. Neste contexto, uma secretária de um consultório médico pode por exemplo ser considerada uma agente de um médico **[Nwana et al 1999]**.

O paradigma de agentes de software propõe *delegar* tarefas a softwares capazes de realizar ações de interesse do usuário, sem que este precise intervir na execução da tarefa. Tais softwares inteligentes e autônomos seriam supostamente capazes de executar os objetivos determinados por seus usuários. Tais objetivos poderiam incluir tarefas como obter saldos bancários, marcar compromissos, gerenciar mensagens eletrônicas, pagar contas, fazer compras, buscar informações específicas e realizar outras tarefas que envolvam a negociação com outros agentes e pessoas **[Hendler 1999]**, **[Parks 2001]**.

Embora a autonomia seja uma característica marcante na maioria dos sistemas desenvolvidos segundo esse paradigma orientado a agentes, ainda não há um consenso na comunidade científica sobre as demais características necessárias para caracterizar agentes de software **[Chauhan 1997]** , **[Nwana 1996]**. Atualmente, a palavra *agente* é normalmente utilizada como um termo guarda-chuva para englobar distintas áreas de pesquisa **[Bradshaw 2001]**.

No trabalho proposto, um *agente* é um software que pode agenciar ou realizar algo para o seu usuário de forma automática. Genericamente, um agente pode ser comparado a um elemento similar ao apresentado na *Figura 1.1*.

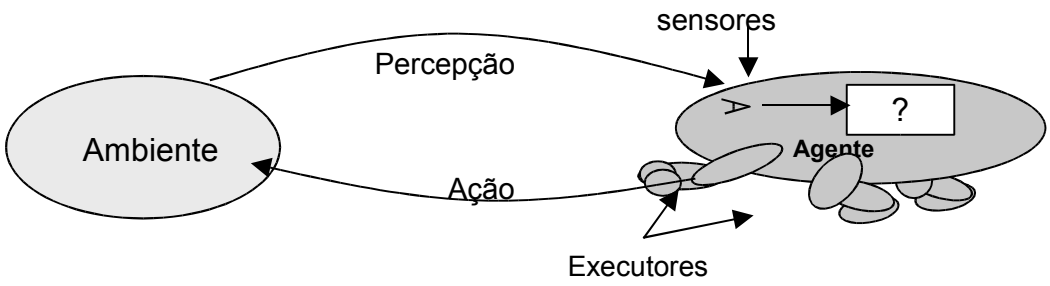


Figura 1.1 - Interação de agentes de software com o ambiente - [Russel et al 1995].

A *Figura 1.1* ilustra o conceito clássico de um agente de software. Segundo a figura, um agente deve ser capaz de perceber seu ambiente através de sensores e agir de forma autônoma sobre este ambiente através de seus executores que realizam ações **[Russel et al 1995]**.

1.3 - Agentes de Software Inteligentes

A chamada *inteligência* dos agentes de software é um tema polêmico na comunidade de pesquisa, uma vez que diferentes pesquisadores procuram características diferentes para definir inteligência no comportamento dos agentes. Segundo o professor Stuart Russel, da Universidade da Califórnia, o que faz um agente ser chamado de inteligente é a sua capacidade de reagir *racionalmente* aos estímulos do ambiente no qual ele está inserido, sendo que seu grau de inteligência pode ser medido em função das ações que ele realiza **[Russel et al 1995]**. Contudo, Russel deixa em aberto a definição da racionalidade em si, o que não esclarece por completo o tipo de comportamentos esperados dos agentes para que eles possam ser considerados inteligentes.

Há também a questão filosófica e psicológica sobre o significado da inteligência. Não se sabe ao certo se o tipo de inteligência, que se pode esperar dos agentes ou de outros software inteligentes, é o mesmo tipo de inteligência encontrada nos seres humanos **[Minsky 2000]**. No intuito de obter uma definição ampla o

bastante para definir as várias pesquisas na área de agentes inteligentes, alguns pesquisadores como **[Wooldridge et al 1995]** e **[Nissen 1995]** optaram por definir a inteligência dos agentes através de uma série de comportamentos esperados dos agentes. Tais comportamentos incluem:

- **Autonomia** - Capacidade do agente de estabelecer comportamento pró-ativo direcionado a metas estabelecidas, tomando decisões sem interferência do usuário. O agente deve possuir certo grau de comportamento independente de forma que possa ser distinto de um software comum.

- **Capacidade de Comunicação** - A comunicação entre agentes pode ser necessária para um agente alcançar seus objetivos. A comunicação entre agentes costuma ser utilizada para acessar informação de outros agentes e de serviços externos. Há também a comunicação entre agentes e usuários.

- **Capacidade de Cooperação** - A capacidade de cooperação entre agentes é uma extensão natural da capacidade de comunicação. Agentes podem colaborar entre si para resolução de problemas formando uma sociedade de agentes. A capacidade de cooperação normalmente está ligada a técnicas de Inteligência Artificial Distribuída, uma das primeiras áreas de pesquisa que fez uso de agentes.

- **Capacidade de Raciocínio** - A capacidade de representar e manipular conhecimento permite obter resultados que satisfaçam as metas do agente. O conhecimento do agente pode ser expresso explicitamente através de modelos simbólicos ou através de dados que podem ser processados por diferentes técnicas de Inteligência Artificial (*Data Mining*, *Redes Neurais*, *Algoritmos Genéticos*,etc).

- **Planejamento** - Para executar suas metas, muitas vezes se faz necessário que os agentes possuam mecanismos de planejamento para executar suas ações. Modelos de planejamento utilizados em agentes incluem *HTN (Hierarchical Task Networks)*, *Ontologias temporais*, entre outros.

- **Adaptabilidade** - A capacidade de analisar o ambiente em que o agente está operando e reagir a eventos que ocorram, é importante quando as condições sobre as quais os agentes atuam são dinâmicas (novo conhecimento, novas inputs, etc). Sob tais circunstâncias, é mister que o agente se adapte à limitações e novas condições do ambiente para que possa atingir suas metas. Técnicas de gerenciamento e planejamento evolutivos podem ser utilizadas para fornecer adaptabilidade a agentes de software.

O interesse específico deste trabalho concentra-se em agentes que podem ser considerados inteligentes devido ao seu conhecimento simbólico de senso comum e de sua capacidade de execução de ações baseadas em um dado contexto específico.

1.4 - Objetivos do Trabalho

Este trabalho visa resolver alguns dos problemas comumente encontrados na criação de sistemas baseados em agentes inteligentes tais como a dificuldade de adicionar novos comportamentos de maneira dinâmica ao sistema e a limitação da interação do usuário com os agentes. Propõe-se o uso de comportamentos implementados por componentes e a interação do usuário com o sistema através de linguagem natural restrita como maneira de trazer maior facilidade no uso e construção de sistemas baseados em agentes inteligentes.

Busca-se, como objetivo deste trabalho, construir um ambiente de desenvolvimento de agentes capazes de executar ações e manipular conhecimento através de linguagem natural restrita. Almeja-se também, criar ferramentas funcionais que tornem possível expandir o conhecimento e adicionar novos comportamentos para criação de agentes inteligentes.

1.5- Contribuições do Trabalho

A linguagem artificial *Universal Communication Language* (UCL) desenvolvida por **[Montesco 2001]** permite representar sentenças em linguagem natural através de redes semânticas. Tais redes também podem ser utilizadas para representação de conhecimento. Assim, se os conceitos definidos em uma sentença em linguagem natural existirem em uma base de conhecimento é possível criar algoritmos que processem uma requisição feita em linguagem natural através de informações existentes na base de conhecimento. Esta é a hipótese para a criação de um ambiente de desenvolvimento de agentes capazes de manipular conhecimento através de requisições feitas em linguagem natural restrita. O desenvolvimento de um protótipo chamado *Semantic Agent Platform* valida esta hipótese e é uma das contribuições trazidas por este trabalho, como descrito em maior detalhe no *Capitulo VI.*

Este projeto propõe ainda um novo modelo para criação de agentes inteligentes chamado *agentes atômicos* que traz uma traz uma contribuição original na forma de integrar comportamentos ao conhecimento de agentes inteligentes. Tais comportamentos, representados através de componentes, podem ser manipulados dinamicamente através de uma rede semântica estendida, como detalhado na seção *4.2.4.2*. A fim de implementar o novo modelo define-se também uma estrutura de dados chamada *Extended Knowledge Network* descrita na seção *4.2.4.3* desta dissertação .

A integração das técnicas utilizadas no ambiente de desenvolvimento de agentes criado neste trabalho traz como contribuição geral a capacidade de manipulação de conhecimento e execução de ações através do uso de linguagem natural restrita. Tal capacidade, útil tanto a desenvolvedores de novos sistemas quanto a

usuários, pode permitir a criação de sistemas baseados em agentes inteligentes mais flexíveis e fáceis de usar.

1.6 - Estrutura do Trabalho

Esta dissertação apresenta uma revisão bibliográfica de algumas características necessárias para a criação de agentes inteligentes, bem como algumas idéias novas que permitem que usuários interajam com agentes através do uso de linguagem natural restrita. Para fins organizacionais, esta monografia foi dividida em sete capítulos. O capítulo inicial apresentou características de agentes inteligentes e o propósito do trabalho. Os demais capítulos analisam, com maior nível de detalhe, algumas características necessárias à construção de agentes inteligentes e o trabalho realizado com este intuito. O *Capítulo II* analisa a comunicação entre agentes de software; o *Capítulo III* discorre sobre representação e manipulação de conhecimento em agentes; o *Capítulo IV* analisa representação e manipulação de comportamentos em agentes de software; o *Capítulo V* aborda aspectos da comunicação usuário-agente; o *Capítulo VI* apresenta maiores detalhes do protótipo do ambiente para criação de agentes desenvolvido neste projeto. Por fim, o *Capítulo VII* apresenta as conclusões, contribuições do trabalho e sugestões para trabalhos futuros.

Capítulo 2 - Comunicação Entre Agentes

2.1 - Considerações Iniciais

Para realizar tarefas do quotidiano necessita-se direta ou indiretamente da ajuda de outras pessoas para comprar algo, se transportar a algum lugar ou realizar quase todo tipo de trabalho. Existe, portanto, uma série de tarefas onde é necessário interagir, cooperar e se comunicar com outras pessoas a fim de se realizar algum trabalho. De maneira similar a seres humanos, há uma série de situações nas quais é necessária a comunicação entre agentes. Tais situações incluem:

- Trocar informações entre si a respeito da parte do mundo que cada um deles está.
- Consultar outros agentes sobre determinados aspectos do mundo.
- Responder a perguntas de outros agentes.
- Aceitar petições e propostas.
- Compartilhar experiências entre si.

Nas seções seguintes descrever-se-á modelos e tecnologias que permitam a comunicação entre agentes. A *seção 2.2* define modelos de comunicação entre agentes como o modelo de quadro-negro descrito na *seção 2.3* e o modelo de troca de mensagens descrito na *seção 2.4*. As linguagens de comunicação entre agentes mais atualmente utilizadas são apresentadas na *seção 2.4.1*. Por fim, a *seção 2.5* apresenta algumas considerações finais sobre o assunto.

2.2 - Modelos de Comunicação Entre Agentes.

Modelos de comunicação entre agentes definem a forma através da qual os agentes se comunicam. As características dos modelos estabelecem o sincronismo da troca de informação e o esquema de inter-relacionamento entre os agentes de um sistema. Nas seções subseqüentes analisar-se-á os dois principais tipos de comunicação entre agentes: esquemas de quadro-negro (blackboard) e troca de mensagens.

2.3 - Comunicação Através de Quadro Negro

Segundo a *arquitetura do quadro negro,* os agentes se comunicam entre si de maneira indireta através de um quadro negro **[Martin et al 1999].** O quadro negro é uma estrutura de dados persistente onde existe uma divisão em regiões ou níveis, visando facilitar a busca de informações. Ele é um meio de interação entre os agentes que funciona como uma espécie de repositório de mensagens. A *Figura 2.1* ilustra a interação de agentes com o quadro negro onde os agentes lêem e escrevem mensagens.

Pode-se dizer que um quadro negro é uma memória de compartilhamento global onde existe uma quantidade de informações e conhecimento usados para leitura e escrita pelos agentes. Em sistemas multi-agentes, os quadros negros são utilizados como um repositório de perguntas e respostas. Os agentes que necessitam de alguma informação escrevem seu pedido no quadro à espera que outros agentes respondam à medida que acessem o mesmo **[Lemos et al 2002]**.

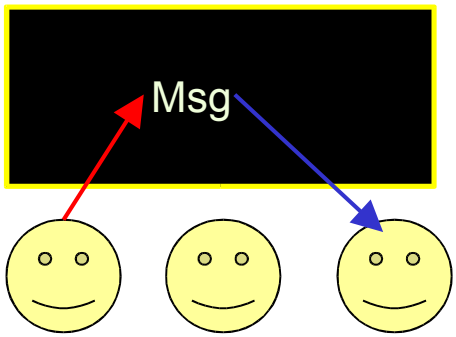


Figura 2.1: Comunicação entre agentes via quadro negro.

A comunicação distribuída utilizando quadro negro foi inicialmente proposta no início da década de 1970 para um sistema de reconhecimento de voz chamado Hearsay **[Carver et al 1994]**. O sistema inicial possuía, além do espaço de busca distribuído, um mecanismo de inferência e gerenciamento de soluções para resoluções de problemas de Inteligência Artificial Distribuída. As tecnologias atuais para comunicação entre agentes segundo o paradigma de quadro negro incluem Linda **[Barry, 1996]**, JavaSpaces **[Sun Microsystems 2003]** e JuspSpace **[Figueiredo 2002]**.

2.4 - Comunicação através de Troca de Mensagens

Outro modelo amplamente adotado para comunicação entre agentes consiste na comunicação direta entre agentes através da troca de mensagens. Segundo este paradigma, é necessário que cada agente saiba os nomes e endereços de todos os agentes que formam o sistema para que as mensagens possam ser trocadas. Para simplificar a comunicação via troca de mensagens, estes sistemas costumam contar com *agentes facilitadores* que armazenam os endereços e serviços oferecidos pelos agentes do sistema **[Helin et al 2002]**, como ilustra a *Figura 2.2*

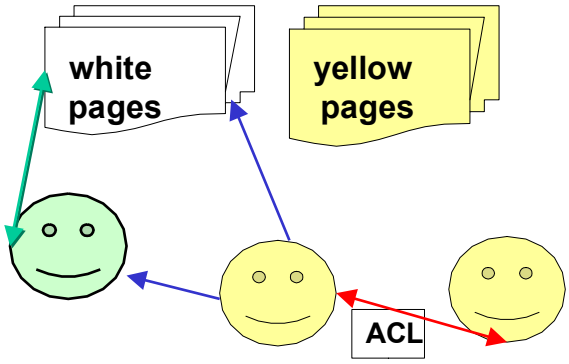


Figura 2.2: Comunicação entre agentes via troca de mensagens.

Quando um agente necessita se comunicar com outro agente do sistema, este pode solicitar que o agente facilitador encontre o agente procurado pelo nome (através das white pages) ou pelo tipo de agente ou serviço oferecido (através das yellow pages). Dependendo da arquitetura do sistema, um agente facilitador pode não ser necessário e todos os agentes acessam os serviços de white pages e yellow pages diretamente.

Para que agentes possam se comunicar via troca de mensagens é preciso:

- uma linguagem para representação de mensagens.
- uma linguagem de representação de conhecimento (conteúdo da mensagem).
- uma maneira de codificar as mensagens (ASCII, binário, XML).
- uma maneira de transportar mensagens (sockets, HTTP, IIOP).

A seguir serão discutidas algumas características gerais das linguagens de comunicação de agentes. As linguagens para representação de conhecimento serão discutidas com maior detalhe no *Capítulo 3*. O modo de codificação e envio de mensagens é dependente da implementação dos agentes e será abordado no *Capítulo 6*.

2.4.1 - Linguagens para Comunicação via Troca de Mensagens

Uma linguagem de comunicação entre agentes (ACL) define os tipos de mensagens utilizadas na comunicação entre agentes. Atualmente existem duas linguagens de comunicação entre agentes amplamente utilizadas: a KQML (*Knowledge Query Manipulation Language*), desenvolvida pelas universidades de Maryland e Stanford; e a FIPA ACL, um padrão estabelecido pela *Foundation for Intelligent Physical Agents*. Ambas as ACLs utilizam *performativas* para definir o protocolo de comunicação entre agentes. O conceito de performativas deriva da "teoria dos atos da fala", proposta pelo filósofo J. L. Austin **[Soemarmo 2002]**. Segundo a teoria dos atos da fala, a comunicação ocorre através de atos tais como: requisições, sugestões, promessas, ameaças, entre outros **[Searle 1969]**. Estes atos representam os tipos de mensagens trocadas pelos agentes. O conteúdo de cada mensagem é expresso em uma linguagem de conteúdo independente da linguagem de comunicação entre agentes **[Labrou et al 1999]**.

Na *sub-seção 2.4.1.1* são apresentados alguns detalhes da KQML enquanto na *sub-seção 2.4.1.2* discutir-se-á características da FIPA-ACL.

2.4.1.1 - KQML

A *Knowledge Query Manipulation Language* faz parte de um projeto financiado pelo DARPA (Defense Advanced Research Projects Agency) e por outras agências ligadas ao departamento de defesa norte-americano. A KQML tem por objetivo desenvolver uma estrutura que permita o compartilhamento de conhecimento entre aplicações **[Finin et al 1997]**.

A KQML é uma linguagem de comunicação de alto nível orientada à troca de mensagens independente da sintaxe do conteúdo e da ontologia associada. KQML é também independente do mecanismo de transporte (TCP/IP, SMRP, IIOP ou outro). As mensagens transmitidas através de KQML podem ter seu conteúdo representado através de Knowledge Languages (KLs) que utilizam tanto o formato texto puro ASCII, quanto o formato binário. As implementações de KQML simplesmente ignoram o conteúdo das mensagens, exceto seu tamanho em bytes. As mensagens KQML contém os parâmetros de transporte com os endereços do agente transmissor e do agente receptor envolvidos na comunicação. Tais parâmetros podem ser combinados com um serviço de nomes, endereçamento e registro para conversão em endereços compatíveis com a camada de transmissão.

Os tipos de mensagem de comunicação da KQML são modelados de acordo com a teoria da fala **[Finin et al 2001]** através da descrição da intenção do agente. Essa descrição se refere aos desejos, crenças, asserções,

comandos, requisições, consultas e outras ações que descrevem o comportamento dos agentes. A linguagem KQML oferece diversos tipos de mensagens, conhecidas como performativas, que representam o intuito do agente ao enviar uma mensagem. As performativas KQML incluem:

- Performativas de consulta (Ask, Ask-if, Ask-one, Ask-all).
- Performativas de troca de informação (Tell, Untell, Insert, Uninsert, Delete-one, Delete-all, Undelete).
- Performativas de requisição de serviço (Subscribe, Achieve, Unachieve, Broker-one, Broker-all).
- Performativas para sincronismo do envio de mensagens (Error, Sorry, Standby, Ready, Next,
- Forward, Broadcast, Rest, Discard, Transport-address).
- Performativas para gerenciamento de agentes (Register, Unregister, Advertive, Unadvertive, Recomended-one, Recomended-all, Recruit-one, Recruit-all).

A *Figura 2.3* ilustra um caso simples de comunicação entre agentes utilizando a linguagem KQML.

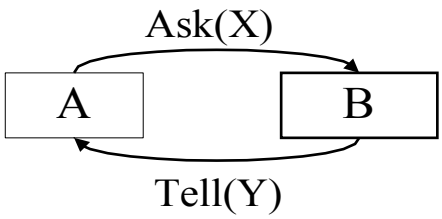


Figura 2.3 - Caso simples de comunicação entre agentes usando KQML.

Conforme ilustra a *Figura 2.3* um agente *A* solicita uma informação a um agente *B* enviando-lhe uma mensagem cuja performativa é *ASK* e o conteúdo é *X*. O agente *B*, recebe a mensagem, a processa e em seguida envia uma mensagem ao agente *A* informando-o do resultado da consulta recebida. Para isto o agente *B* utiliza uma mensagem cuja performativa é *TELL* e cujo conteúdo é *Y*. A *Figura 2.4* ilustra o possível formato de uma mensagem enviada em resposta a uma consulta.

```
(tell
: sender bhkAgent
: receiver sharpAGent
: content
'price(ISBN342959, 24,59)'
: language Prolog
: ontology ecommerce
: in-reply-to messageID#5734A)
```

Figura 2.4- Formato de uma mensagem KQML

A Figura 2.4 ilustra a estrutura de uma mensagem KQML que informa o resultado da consulta do preço de um livro cujo ISBN é 342959. Esta pode ser, por exemplo, a resposta do agente B à consulta realizada pelo

agente A na Figura 2.3. Pode-se observar que a sintaxe da linguagem é baseada em LISP. A mensagem representada é composta pelos campos: remetente (bhkAgent), destinatário (sharpAgent), conteúdo da mensagem (price(ISBN342959,24.95)) linguagem de representação de conhecimento (prolog), nome da ontologia associada (ecommerce), e o campo in-reply-to que informa que esta mensagem esta sendo enviada em resposta a mensagem recebida '57334A'.

2.4.1.2 - FIPA ACL

A FIPA-ACL é a linguagem de comunicação de agentes desenvolvida pela FIPA (Foundation for Intelligent Physical Agents). A FIPA é uma organização sem fins lucrativos fundada em 1996 na Genebra, Suíça, cujo objetivo é estabelecer padrões para o desenvolvimento de agentes inteligentes. De maneira similar a linguagem de comunicação KQML, a FIPA-ACL também se baseia em performativas derivadas da "teoria dos atos da fala". A sintaxe das duas linguagens é semelhante, embora o conjunto de performativas definidas seja diferente. A FIPA-ACL define 22 performativas organizadas em 4 categorias: transferência de informação, negociação, ação e gerenciamento de erros **[FIPA 2001].** A lista completa de performativas FIPA-ACL e seus casos de uso se encontram no *Apêndice A*. A *Figura 2.5* ilustra um exemplo de comunicação utilizando a linguagem FIPA-ACL.

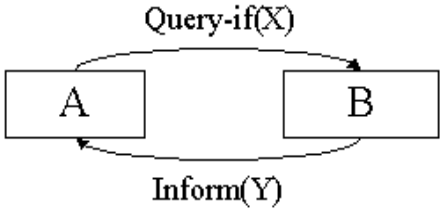


Figura 2.5 - Caso simples de comunicação entre agentes usando FIPA-ACL

Na *Figura 2.5* tem-se um caso simples de comunicação, onde um agente *A* solicita uma informação à um agente *B*, através do envio de uma mensagem cuja performativa FIPA-ACL é *query-if*. O Agente *B* responde ao agente *A* através do envio de uma mensagem cuja perfomativa FIPA-ACL é *inform*.

```
(inform
: sender bhkAgent
: receiver sharpAgent
: content
'price(ISBN342959,24.95)'
: language Prolog
: ontology ecommerce
: in-reply-to 5734A
: conversation-id 5734B )
```

Figura 2.6 - Exemplo de uma mensagem definida em FIPA-ACL

A *Figura 2.6* ilustra a estrutura de uma mensagem FIPA-ACL que informa o resultado da consulta do preço de um livro cujo ISBN é 342959. Esta pode ser por exemplo, a resposta do agente *B* à consulta realizada pelo agente *A* na *Figura 2.5*. A mensagem é composta pelos campos: remetente (bhkAgent), destinatário (sharpAgent), conteúdo (price(ISBN342959,24.95)), linguagem de representação de conhecimento (prolog), nome da ontologia associada (ecommerce), o campo in-reply-to que informa que esta mensagem esta sendo enviada em resposta a mensagem recebida '57334A' e o campo conversation-id que informa que o código da comunicação é '5734B'.

2.5 - Considerações Finais

Uma das características necessárias a criação de agentes inteligentes é que estes possam se comunicar. A comunicação entre agentes é utilizada para tarefas dos agentes que envolvam troca de informação e/ou compartilhamento de experiências. As duas principais metodologias atualmente em uso para comunicação entre agentes são o esquema quadro negro que envolve um espaço de memória distribuída compartilhado para leitura e escrita de mensagens e o método de comunicação através de troca direta de mensagens. Para o último método as duas principais linguagens para comunicação atualmente em uso são a FIPA-ACL e a KQML. Devido a maior quantidade de implementações **[Emorphia 2003]**, **[Greenwood 2000]**, **[Bellifemine et al 2003]**, **[Bergenti et al 2002]**, maior quantidade de serviços desenvolvidos para linguagem e por ser um padrão na plataforma Java , optou-se por utilizar para o protótipo do sistema desenvolvido nesta dissertação a linguagem de comunicação entre agentes FIPA-ACL como discutido no *Capítulo 6*.

Capítulo 3 - Representação e Manipulação de Conhecimento em Agentes de Software

3.1 - Considerações Iniciais

Conhecimento é um recurso poderoso, cuja transformação, utilização e aprimoramento geram riqueza. Durante muito tempo, o conhecimento humano foi transmitido através da palavra falada. A invenção da máquina tipográfica na Europa, por Yoham Gutemberg, por volta de 1440, é considerado um marco na disseminação do conhecimento, pois permitiu a palavra escrita ser acessível a um grande número de pessoas. Outro marco importante para a transmissão de conhecimento foi a criação da Internet comercial que permitiu a cidadãos de todo o planeta ter acesso a um imenso repositório de conhecimento e serviços na rede.

A utilização de conhecimento em aplicações computacionais permite realizar tarefas antes restritas a especialistas em um domínio ou especialistas em computação **[Hayes et al 1983].** O chamado *"conhecimento"* que pode ser representado computacionalmente compreende a combinação de instintos, idéias, regras e procedimentos que guiam as ações e decisões em um dado domínio **[Rezende 2000]**.

As seções seguintes discutem tecnologias para a representação de conhecimento utilizadas em agentes de software. A *seção 3.2* apresenta o conceito de agentes que "raciocinam"; a *seção 3.3* ilustra as linguagens de representação de conhecimento comumente utilizadas em agentes de software; a *seção 3.4* introduz o conceito de ontologias cujas aplicações são discutidas nas seções *3.5 e 3.6.* Por fim, a *seção 3.6* apresenta algumas considerações finais sobre a representação de conhecimento em agentes de software.

3.2 - Agentes que Raciocinam

Para que agente inteligentes "raciocinem" é preciso que estes possuam um modelo do domínio sobre o qual atuam. A abordagem clássica para incorporar conhecimento na construção de agentes propõe o uso de um modelo simbólico do mundo, explicitamente representado, e cujas decisões (ações) são tomadas via um raciocínio lógico, baseado em casamento de padrões e manipulações simbólicas **[Wooldridge et al 1995] [Finin 1998]**. O conhecimento utilizado em agentes inteligentes pode ser descrito por **[Rich et al. 1994]**:

- Uma linguagem de representação de conhecimento.
- Uma base de conhecimento que contenha informações apropriadas a uma determinada tarefa.
- Uma máquina de inferência capaz de processar o conhecimento armazenado.

O conhecimento pode ser incorporado aos agentes através de *Linguagens de Representação de Conhecimento* (KL) que permitem criar bases de conhecimento através de linguagens formais e manipular conhecimento através de máquinas de inferência associadas **[Hayes et al 1983]**. A abordagem simbólica de representação de conhecimento para agentes de software costuma ser, muitas vezes, semelhante a sistemas de produção especialistas. Por isto, agentes desenvolvidos segundo este paradigma, são às vezes descritos como agentes especialistas.

Embora agentes especialistas costumem utilizar a mesma estrutura de sistemas especialistas para armazenar e manipular conhecimento, estes não precisam ter necessariamente conhecimento altamente especializado (como exploração de petróleo, medicina, química, geologia, etc). Comumente agentes especialistas possuem conhecimento de tarefas triviais (contas bancárias, compras de passagens aéreas, etc) que gerenciam informações úteis ao usuário **[Maes 1997]**.

Outras características específicas dos agentes especialistas é que o conhecimento dos agentes costuma ser dinâmico, pois os agentes especialistas podem incorporar novo conhecimento através da interação com o usuário, outros agentes e fontes de informação. Agentes especialistas costumam incorporar novo conhecimento de maneira automática, enquanto sistemas especialistas necessitam de um usuário ou engenheiro de conhecimento para incorporar novo conhecimento **[Maes 1997]**. Também é peculiar aos agentes de software, a sua capacidade de agir sobre o ambiente, a fim de atingir alguma meta estabelecida por um usuário como ilustra a *Figura 3.1*.

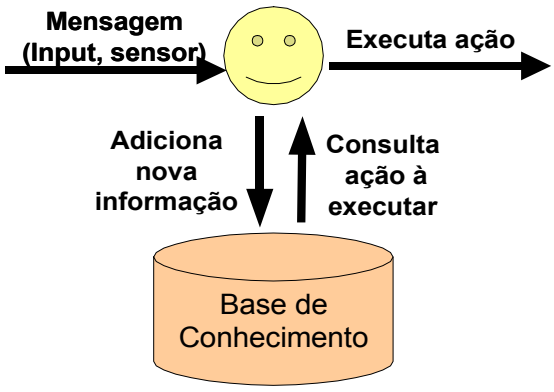


Figura 3.1 - Um modelo de Agente Especialista [Finin 1998]

A *Figura 3.1* ilustra um modelo proposto pelo professor Timothy W. Finin da Universidade de Maryland **[Finin 1998]** para criação de agentes especialistas. Segundo o modelo:

- O agente adiciona à base de conhecimento novas informações que recebe
- Consulta que ação deve executar
- Então, executa ação apropriada

Além de sistemas especialistas, outras técnicas de Inteligência Artificial podem ser utilizadas em agentes de software. O modelo simbólico do agente não precisa ser necessariamente baseado em regras de produção, mas pode fazer uso de cláusulas de Horn ou de redes semânticas como discutido nas seções posteriores deste capítulo. Além disto, o agente de software não necessita ter um modelo simbólico do mundo pré-programado. As regras do agente podem ser obtidas através de data-mining **[Kargupta et al 1997]**. Outra abordagem que pode ser utilizada consiste em utilizar modelos conexionistas, como por exemplo, redes neurais extraídas de bases de dados que definem o comportamento dos agentes **[Choi et al 1999]**.

3.3. - Linguagens de Representação de Conhecimento Baseadas em Lógica

A fim de representar conhecimento de maneira precisa, agentes de software costumam utilizar linguagens de representação de conhecimento baseadas em lógica. Diferentes tipos de lógica podem ser utilizados como lógica proposicional, lógica de primeira ordem, ou ainda lógica fuzzy **[Turban 1998]**.

Os sistemas baseados em lógica proposicional costumam ser compostos por um conjunto de regras de produção do tipo *if-then-else* e um conjunto simples de operadores (implies, equals,not,and,or). Embora simples de usar, os sistemas baseados em lógica proposicional não possuem muita expressividade o que os impede de representar fatos complexos. Exemplos de linguagens proposicionais para agentes especialistas incluem CLIPS **[Riley 2002]** e JESS **[Hill 2002]**.

Os sistemas baseados em lógica de primeira ordem são compostos por fatos, objetos e relações. Normalmente as sentenças destes sistemas são expressas através de cláusulas de Horn. Exemplos de linguagens baseadas em lógica de primeira ordem que costumam ser aplicadas em agentes incluem Prolog **[Rowe 1998]** e Prodigy **[Carbonell et al 1991] [Arriola et al 1995]**.

Os sistemas baseados em lógica Fuzzy são definidos através de graus de veracidade associados à funções de pertinência. Um exemplo de linguagem de representação de conhecimento baseada em lógica fuzzy para agentes é o Fuzzy CLIPS **[Togai 2002]**.

As linguagens de representação de conhecimento têm associadas a si, máquinas de inferências capazes de processar o conhecimento armazenado em bases de conhecimento **[Finin 1998]**. Uma máquina de inferência funciona como um "interpretador de conhecimento" capaz de processar, avaliar e interpretar o conhecimento representado pelas linguagens de representação de conhecimento. O tipo de inferência realizado pela máquina de inferência depende do tipo de lógica da linguagem de representação de conhecimento utilizada.

Em linguagens baseadas em lógica proposicional a inferência costuma ser baseada em um encadeamento das regras através de algoritmos de backward chaining ou forward chaining. Agentes inteligentes costumam fazer uso de forward chaining para inferir conclusões a partir de novas inputs que recebem do ambiente e backward chaining para atingir uma meta a partir dos fatos conhecidos **[Rowe 1998]**.

Já as linguagens baseadas em lógica de primeira ordem costumam utilizar inferência lógica baseada apenas em algoritmos de backward chaining . O raciocínio disponível é então baseado em metas as quais os agentes desejam atingir (goal-directed reasoning) . A máquina de inferência tenta provar estas metas a partir das cláusulas armazenadas na base de conhecimento. Novas sentenças são criadas de acordo com as conseqüências de um conjunto de predicados. O funcionamento de uma máquina de inferência é baseado em um algoritmo de pattern matching capaz de comparar os fatos com padrões determinando quais regras da base de conhecimento são aplicáveis. Uma vez que uma regra é disparada, as ações especificadas pela regra são executadas (o que pode afetar a lista de regras aplicáveis através da adição ou remoção de novos fatos). Novas regras aplicáveis continuam sendo aplicadas até que não exista nenhuma regra que seja aplicável **[Riley 2002]**.

Nas sub-seções seguintes apresentar-se-á algumas linguagens de representação de conhecimento amplamente utilizadas e suas máquinas de inferência associadas.

3.3.1. - CLIPS

O projeto CLIPS (C Language Integrated Production System) teve início em 1984 na agência espacial norte-americana (NASA) como uma linguagem e uma máquina de inferência para construção de sistemas especialistas. CLIPS é atualmente distribuído como software público de código aberto pelos autores que iniciaram o projeto **[Riley 2002]** .

Os fatos e regras expressos em CLIPs têm sintaxe LISP-like e podem ser adicionados, excluídos ou alterados em tempo de execução do sistema. CLIPs trabalha com fatos simples chamados asserções que podem ser compostos para representar fatos complexos. CLIPS possui operadores lógicos tradicionais (maior, menor,

igual, e, ou) para construção de regras e fatos. As regras CLIPs são do tipo if-then-else, mas sua sintaxe é diferente, pois os elementos de uma regra possuem formato de listas.

O código da máquina de inferência CLIPS original **[Riley 2002]** foi desenvolvido em ANSI-C e portado para diversos sistemas operacionais. Uma segunda implementação desta máquina de inferência foi desenvolvida em Java por Ernest Friedman-Hill do Sandia National Laboratories **[Hill 2002]**. A implementação da máquina de inferência CLIPS realizada por Friedman foi chamada JESS (Java Expert System Shell).

A máquina de inferência JESS utiliza uma linguagem derivada de CLIPS que implementa o subconjunto de comandos essenciais da linguagem, sendo capaz de executar diversos programas CLIPS. Para o encadeamento da execução das regras, a máquina de inferência JESS utiliza o algoritmo RETE que dispara as regras através de forward chaining.

3.3.2 - Prolog

Outra linguagem de representação de conhecimento que pode ser utilizada por agentes de software é Prolog. A linguagem Prolog foi desenvolvida no começo da década de 1970 por Alain Colmerauer na Universidade de Marseille e em seguida aperfeiçoada por Robert Kowalski na Universidade de Edinburgh **[Rowe 1998].** Atualmente a linguagem Prolog é padronizada pelo ISO/IEC 13211.

Prolog utiliza lógica de predicados através de cláusulas de Horn. Estas cláusulas que são um subconjunto da lógica de primeira ordem, expressam fatos e regras sobre um determinado domínio. A máquina de inferência Prolog possui o mecanismo de backtracking, responsável por analisar as regras e fatos expressos em uma base de conhecimento através de deduções e derivações do conhecimento representado. Prolog responde a consultas a sua base de conhecimento utilizando o mecanismo de backtracking. Este mecanismo utiliza procura por largura disparando as regras da base de conhecimento através de backward chaining.

Implementações de Prolog de código aberto publicamente disponíveis para plataforma Java incluem Javalog (capaz de ativar ações que executam componentes Java) **[Zunino et al 2000]**, Java Internet Prolog (JIP) **[Chirico 2002]** e jProlog **[Demoen 2002]**.

3.3.3 - Soar

Soar é um sistema de produção para solução de problemas gerais baseado em um conjunto de hipóteses e princípios para processamento de modelos cognitivos. Soar oferece estruturas especificas para codificação de conhecimento e execução de ações baseadas em metas. Soar suporta raciocínio simbólico, aprendizado, planejamento e outras capacidades utilizadas para implementação de comportamento inteligente **[Rosenbloom et al 1993]**.

A implementação original da máquina de inferência Soar foi desenvolvida em C/TCL. *Soar2Java* é um projeto de um grupo de desenvolvedores de software livre que vem implementando em Java uma versão do Soar compatível com a licença GPL (Gnu Public License) **[Abdallah 2002]**.

3.3.4 – ACT-R

ACT-R (Adaptive Control of Thought) é um sistema que representa modelos cognitivos através de conhecimento procedural e conhecimento declarativo. O conhecimento declarativo é representado através de "chunks" que representam fatos memorizados ou percebidos como "Brasilia é a capital do Brasil" ou "Há um carro à frente". O conhecimento procedural codifica processos e habilidades necessárias para atingir uma certa meta. As unidades do conhecimento procedural são denominadas "produções" e representam condições para ações que são ativadas quando certas condições são satisfeitas. As condições dependem da meta a ser alcançada, do estado do conhecimento declarativo e de parâmetros de entrada do ambiente externo. As ações disparadas pelo sistema podem alterar o conhecimento do sistema, alterar metas ou iniciar ações externas no ambiente **[Anderson et al 1998]**.

A arquitetura do ACT-R é dividida em 4 subsistemas principais: o sistema perceptivo-motor, o sistema de metas, o sistema declarativo e o sistema procedural. O sistema perceptivo motor é responsável por associar as entradas sensoriais e ações externas no ambiente ao modelo cognitivo do agente. O sistema de metas é responsável por gerenciar as intenções de forma a obter um comportamento que satisfaça um dado conjunto de metas. As resoluções das metas são armazenadas no sistema declarativo que armazena e processa os chunks de conhecimento declarativo de acordo com um algoritmo baseado na *global workspace theory,* uma teoria psicológica para consciência humana **[Bogner et al 2001]**. O sistema procedural processa as produções (regras) a fim de alcançar uma dada meta.

O projeto ACT-R é desenvolvido pelo departamento de Psicologia em conjunto com o departamento de computação da Universidade Carnegie Mellon. O software desenvolvido em LISP pode ser obtido e distribuído livremente.

3.4 - Ontologias

Ontologias são ferramentas utilizadas para conceitualização de conhecimento, isto é a definição do vocabulário que representa o conhecimento. A conceitualização do conhecimento consiste em definir a *relação* entre objetos, conceitos e outras entidades que compoem um domínio específico **[Chandrasekaran et al 2002]**. Através de uma conceitualização podemos criar uma visão abstrata e simplificada do mundo que queremos representar para algum propósito **[Guarino 1998].**

O termo ontologia é utilizado na Filosofia como a ciência que estuda o ser **[Bosh 1997]**. Ontologias vêm sendo criadas por filósofos desde Aristóteles na Grécia antiga como meio de categorizar elementos distintos, possibilitando estabelecer relações, similaridades e diferenças entre estes **[Gruninger et al 2002]** No último século, o surgimento de abordagens formais para criação de ontologias despertou o interesse para sua utilização como técnica de representação de conhecimento em sistemas computacionais na área de inteligência artificial **[Corazzon 2002]**. Ontologias costumam organizar conceitos de maneira hierárquica mantendo em sua estrutura as relações e dependências entre os conceitos definidos no domínio, como ilustra a *Figura 3.2.*

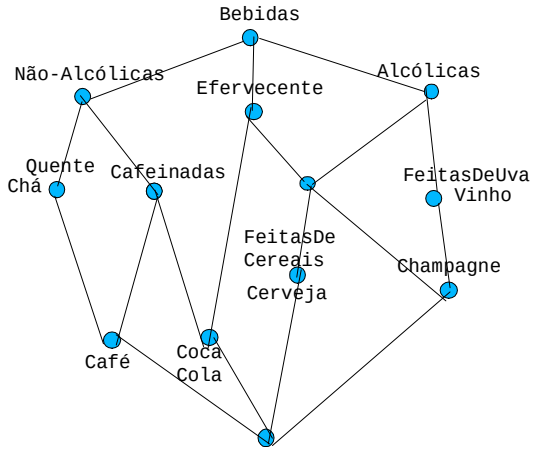


Figura 3.2 - Ontologia como um grafo de conceitos.

No caso mais simples, uma ontologia descreve uma hierarquia (ou taxonomia) de conceitos. O conjunto de objetos descritos por uma ontologia é conhecido como universo do discurso. Em uma ontologia definições associam os nomes das entidades do universo do discurso (classes, relações, funções) com uma descrição sobre o significado e com axiomas sobre as entidades descritas **[Gruber 1993]**.

Ontologias podem ser organizadas para domínios específicos (comércio eletrônico, hospitais, companhia aéreas, etc) ou podem ser genéricas. Ontologias genéricas possuem conhecimento de senso comum, grandes coleções de informações enciclopédicas sobre o mundo **[Chandrasekaran et al 2002]**. Embora as

linguagens utilizadas para criação de ontologias sobre domínios específicos possam ser utilizadas para criação de ontologias de senso comum, existem linguagens mais utilizadas para cada caso.

3.5 - Ontologias para Compartilhamento de Conhecimento entre Agentes

Ontologias estabelecem uma terminologia comum entre membros de uma comunidade. Ontologias definem também relações entre os elementos de um domínio, de forma que agentes de software distintos possam compartilhar um mesmo significado de um dado conhecimento compartilhado entre agentes **[Castel 2002]**. Segundo este modelo, ontologias costumam ser utilizadas como dicionário dos termos utilizados pelas linguagens de representação de conhecimento. Pode-se considerar que as ontologias armazenam meta-informações que representam a semântica do conhecimento, enquanto as linguagens de conhecimento definem a lógica do conhecimento. Assim, se dois ou mais agentes podem concordar no uso de uma mesma ontologia, eles podem então se comunicar, com garantia semântica dos termos envolvidos na comunicação **[Nwana et al 1999]**.

Para que agentes de software possam compartilhar uma ontologia é necessário o uso de uma *linguagem de definição de ontologia* padronizada **[Gruber 1993]**. Analisar-se-á, nas sub-seções seguir, as características de algumas linguagens de definição de ontologias.

3.5.1 - DAML

A DAML (DARPA Agent Markup Language) é uma linguagem para definição de ontologias, desenvolvida pela Agência de Pesquisa do Departamento de Defesa Norte-Americano em conjunto com diversas universidades. A DAML visa permitir a criação de ontologias para o projeto da web semântica **[Lee et al 2001]**. A linguagem tem por objetivo permitir que agentes de software, mecanismos de procura e outras aplicações troquem informações e conhecimento via Internet **[Hendler 2001].**

A linguagem DAML foi projetada como uma linguagem para representação de conhecimento em documentos na web através de ontologias. Sua sintaxe é especificada por um DTD XML publicamente disponível. A DAML permite também a inferência sobre o conhecimento representado, através do suporte a um subconjunto de lógica de primeira ordem. Um diferencial desta linguagem em relação a outras linguagens de definição de ontologias é o mecanismo de múltipla herança entre ontologias que permite a adaptação das ontologias a aplicações específicas **[Harmelen et al 2001]**.

A linguagem DAML vem se tornando a linguagem padrão para criação de ontologias sendo apoiada por autoridades de padronização importantes tais como o W3C (World Wide Web Consortium) e o OWG (Ontology Working Group). Atualmente, algumas ferramentas e repositórios de ontologias vêm sendo desenvolvidos para a linguagem. Entre as ferramentas para o desenvolvimento de ontologias DAML pode-se destacar os editores de ontologias Protegé **[Noy et al 2001]** e DAML Emacs Editor **[Burstein 2002]** . Entre os repositórios de ontologias DAML pode-se destacar o DAML Repository **[Burke 2002]** e o repositório de ontologias DAML da Universidade de Stanford **[Fikes 2002]**.

Outro esforço que visa criar um repositório de ontologias DAML é o projeto Opensyc **[Cyc Corporation 2002]**. Trata-se de um projeto de código aberto que está convertendo a ontologia CYC para o formato DAML A ontologia opensyc inclui 6.000 conceitos e 60.000 relações entre os conceitos definidos. O projeto opensyc vem desenvolvendo também uma máquina de inferência para a ontologia capaz de analisar relações entre as asserções e conceitos representados na ontologia.

3.5.2 - Ontolingua

Ontolingua é uma ferramenta para criação de ontologias utilizando uma extensão da linguagem de representação de conhecimento KIF com suporte a axiomas para ontologias. A Ontolingua foi desenvolvida em LISP visando o compartilhamento e reaproveitamento de ontologias através de mecanismos de derivação de ontologias (derivação, polimorfismo e refinamento) **[Farquhar et al 1996].**

A Ontolingua possui atualmente um repositório centralizado de ontologias na Internet **[KSL Network Services 2002]** . O servidor de ontologias Ontolingua na Internet pode ser acessado remotamente através da API OKBC 2.0 por agentes de software ou outras aplicações baseadas em conhecimento. A OKBC 2.0 (Open Knowledge Base Connectivity) é uma API padrão para acesso a servidores de ontologias semelhante em propósito às APIs padrões para acesso a base de dados (ODBC, JDBC, etc). A OKBC é a API padrão para acesso a servidores de bases de conhecimento adotada pela FIPA e tem implementações (cliente e servidor) em diversas linguagens de programação tais como Java, Lisp e C **[Chaudhri et al 1998]**.

A ferramenta Ontolingua inclui um editor de ontologias acessível pela web e um módulo para organização taxonômica e resolução de conflitos em ontologias, chamado Chimera. O editor de ontologias permite a manipulação de ontologias através de mecanismos de objetos disponíveis em sistemas de frames. Diversos usuários podem compartilhar e expandir ontologias existentes em um servidor facilitando assim o desenvolvimento de ontologias por um grupo de especialistas em um domínio **[Farquhar et al 1996]** .

3.6 - Ontologias para Representação de Conhecimento

A *seção 3.5* analisou o uso de ontologias como ferramentas para definição da semântica de linguagens de representação de conhecimento, permitindo a comunicação entre agentes. A estrutura de ontologias utilizadas para comunicação entre agentes é bastante simples, consistindo de uma hierarquia ou grafo de conceitos, utilizada para representar a relação entre conceitos de um domínio. Entretanto, ontologias são estruturas genéricas o bastante que podem ser utilizadas não somente para estruturar um domínio, como também podem ser estendidas para representação lógica de conhecimento. Pode-se estender as ontologias adicionando relações ao grafo ou taxonomia de conceitos de uma ontologia obtendo-se como resultado uma rede semântica **[Gruber 2002]**.

Redes Semânticas são estruturas que representam relações entre conceitos. A rede semântica mais antiga conhecida foi criada no século III AC pelo filósofo grego Porfírio como forma de explicitar as relações da ontologia Aristotélica. O uso de redes semânticas na computação teve início na década de 60 **[Quillian 1968]** como ferramenta para modelagem da memória humana, sendo mais tarde utilizadas para modelagem da linguagem humana e de conhecimento em geral.

Redes semânticas podem ser representadas graficamente como um conjunto de nós (círculos) e links (arcos) que conectam os nós. Os nós representam conceitos que podem ser objetos físicos (como carro, bola, cadeira, etc) ou objetos abstratos como (bom, mal, etc). Os links representam as relações entre os conceitos. Embora seja possível definir qualquer tipo de relação entre conceitos, certos tipos de relação são normalmente utilizados. Dois tipos importantes são as relações IS-A (é-um) e A-KIND-OF (um-tipo-de). As duas relações são similares: A-KIND-OF é utilizada para mostrar que elementos de uma classe são um subconjunto de outra classe, enquanto a relação IS-A é utilizada para mostrar que um elemento é membro de uma classe **[Sowa 2003]**. A *Figura 3.3* ilustra uma rede semântica.

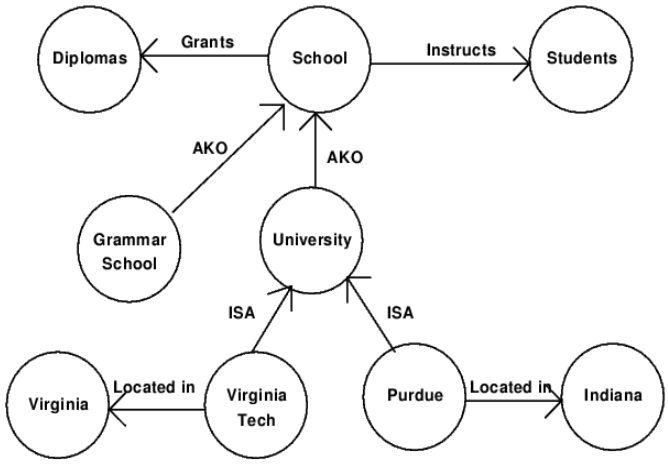


Figura 3.3 - Rede Semântica [Turban 1998]

Costuma-se denominar as relações existentes entre os conceitos de uma rede semântica como sendo *axiomas* ou *asserções*. A fim de manipular informação, agentes podem processar as asserções entre conceitos a fim de inferir conclusões. A rede semântica da *Figura 3.3,* por exemplo representa asserções tais como 'Uma escola instrui estudantes' e 'Viginia-Tech é uma universidade'. As asserções existentes nas rede semânticas permitem inferir algumas conclusões. De acordo com o exemplo da *Figura 3.3* pode-se concluir, por exemplo, que a Virginia-Tech instrui estudantes, pois Viginia-Tech é uma Universidade que é um tipo de escola e escolas instruem estudantes.

O tipo de lógica utilizada para processar as redes semânticas é um sub-conjunto da lógica de primeira ordem chamada lógica monolítica. De acordo com a lógica monolítica, novas informações aumentam a quantidade de teoremas a serem provados e nenhuma relação jamais é descartada ou alterada. Temos portanto, um esquema de herança simples, o qual necessita de operadores que descaracterizem certas relações, permitindo assim a criação de heranças múltiplas. A forma de herança ontológica simples existente em redes semânticas pode gerar conflitos como o ilustrado na *Figura 3.4*:

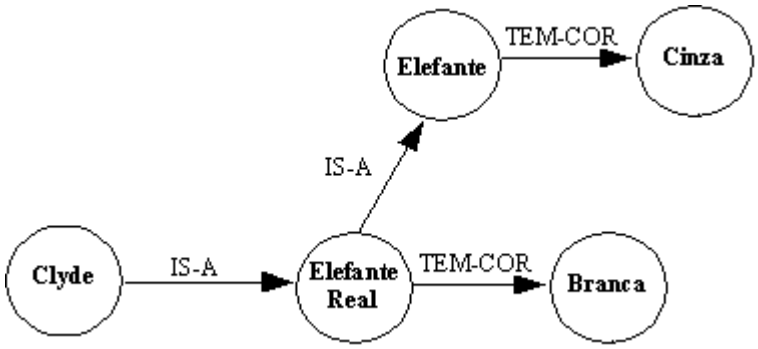


Figura 3.4 - Resolução de conflitos em redes semânticas

Na *Figura 3.4* Clyde é um elefante real branco. Se a rede semântica fornecesse apenas um mecanismo de herança simples e não permitisse o uso de sobrecarga (overriding), não seria possível especificar que os elefantes reais são brancos e não cinza **[Sowa 2000]**.

Como se pode notar, a criação e uso de redes semânticas gera às vezes certos conflitos agravados pelo fato de que a estrutura destas redes ainda não foi tão formalizada quanto as linguagens de representação de conhecimento baseadas em lógica simbólica. Apesar disto, a adição de axiomas e de operadores de resolução de conflitos permite que redes semânticas sejam úteis para o representação de conhecimento em vários domínios. Extensões de redes semânticas, como redes semânticas assertivas, redes semânticas implicacionais e redes semânticas executáveis garantem controle lógico mais rígido e maior flexibilidade para modelagem de domínios de conhecimento.

As sub-seções seguintes analisaram duas linguagens que representam conhecimento através de redes semânticas (ou ontologias estendidas): CycL e Thought Treasure.

3.6.1 - Cyc

A ontologia Cyc é o maior esforço para criação de uma ontologia unificada do senso comum já realizado. O senso comum armazenado se refere a uma série de fatos e relações que qualquer ser humano conhece sobre o mundo, mas que computadores ignoram completamente. Exemplos de conhecimento de senso comum incluem fatos tais como: 'Copos cheios de água devem ser carregados com a boca virada para cima' , 'Pessoas mortas não fazem compras', 'Toda Pessoa é mais velha do que seus pais', 'Gato é um tipo de animal', etc **[Mueller 2000].**

O projeto Cyc foi iniciado em 1984 pelo professor Doug Lenat (Universidade Carnegie-Mellon e Universidade Stanford) e contém hoje mais de 1.500.000 conceitos e asserções. Hoje a ontologia Cyc é um produto comercializado pela Cycorp, mas parte da ontologia é distribuída livremente. O conhecimento em Cyc é representado através de conceitos que representam os elementos da ontologia e asserções que representam as relações ente os elementos da ontologia **[Stoffer 2000]**. Um exemplo de conceito em CycL (Cyc Language) é definido na *Figura 3.5*

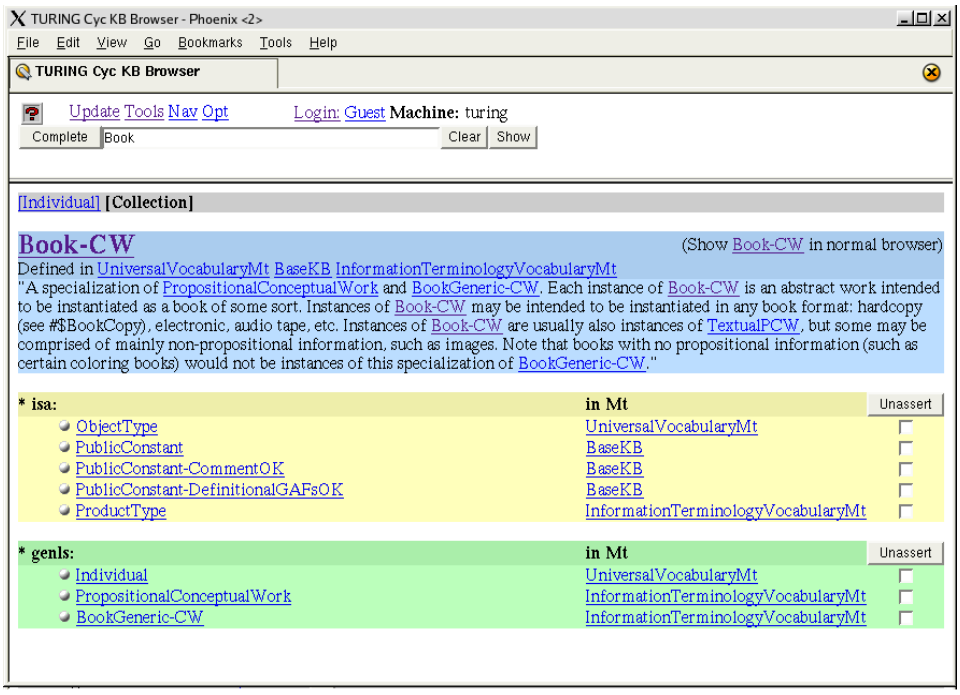


Figura 3.5 - Exemplo de um conceito Cyc

Como se pode observar a partir da *Figura 3.5,* um conceito em CycL é definido por um nome (conceito), uma descrição do conceito e seu contexto (asserções diretamente relacionadas). As asserções em CycL representam relações de uma rede semântica entre os conceitos.

- Uma *fórmula* - Expressão em linguagem formal que faz alguma definição sobre o mundo (define um conceito, por exemplo).
- Um *valor de verdade* - O grau de verdade da formula (0-100 ou desconhecido)
- Um *contexto* - Links para asserções relacionadas.
- Uma *direção* - Tipo de inferência associada à asserção: forward ou backward. Inferência com asserções do tipo forward são avaliadas a medida que novas asserções são adicionadas a base de conhecimento, enquanto asserções do tipo backward são avaliadas quando uma query é executada. A direção padrão de regras é backward. Contudo as regras do tipo *Ground Atomic* da forma predicado(arg1, arg2 ... argn) tem direção forward por default.
- Um *suporte* - justificativa do porquê da asserção existir na base de conhecimento.

Cada asserção em CYC é tratada de forma independente de outras asserções na base de conhecimento através de contextos. Isto permite manter. em uma mesma base, asserções tais como 'Drácula é um vampiro' e 'Vampiros não existem'. Esta abordagem desconexa representa a ontologia CYC como um 'mar' de asserções. Toda vez que uma nova asserção é adicionada a base de conhecimentos, gera-se uma rede de relacionamentos com todos os termos expressos na nova asserção. Algumas vezes, novas asserções geram novo conhecimento útil, mas algumas vezes geram contradições. Cada conceito da ontologia pode ter um ou mais contextos associados como forma de resolver os conflitos entre as asserções. O conceito teoria econômica pode ter, por exemplo, diversos contextos relacionados, cada qual com um conjunto de asserções diferente **[Cycorp 2003]**.

3.6.2 - Thought Treasure

O software Thought Treasure (TT) é um software para processamento de linguagem natural baseado em ontologias desenvolvido por Erik T Mueller **[Mueller 1998]**. De maneira similar ao Cyc, o Thought Treasure mantém uma grande ontologia com informações sobre senso comum. As ontologias do Thought Threasure são compostas por conceitos e asserções que estabelecem relações entre os conceitos. A base de conhecimento é composta por 27.093 conceitos e 51.305 asserções. O Thought Treasure organiza os conceitos e relações ontológicas de maneira hierárquica como ilustra a *Figura 3.6.*

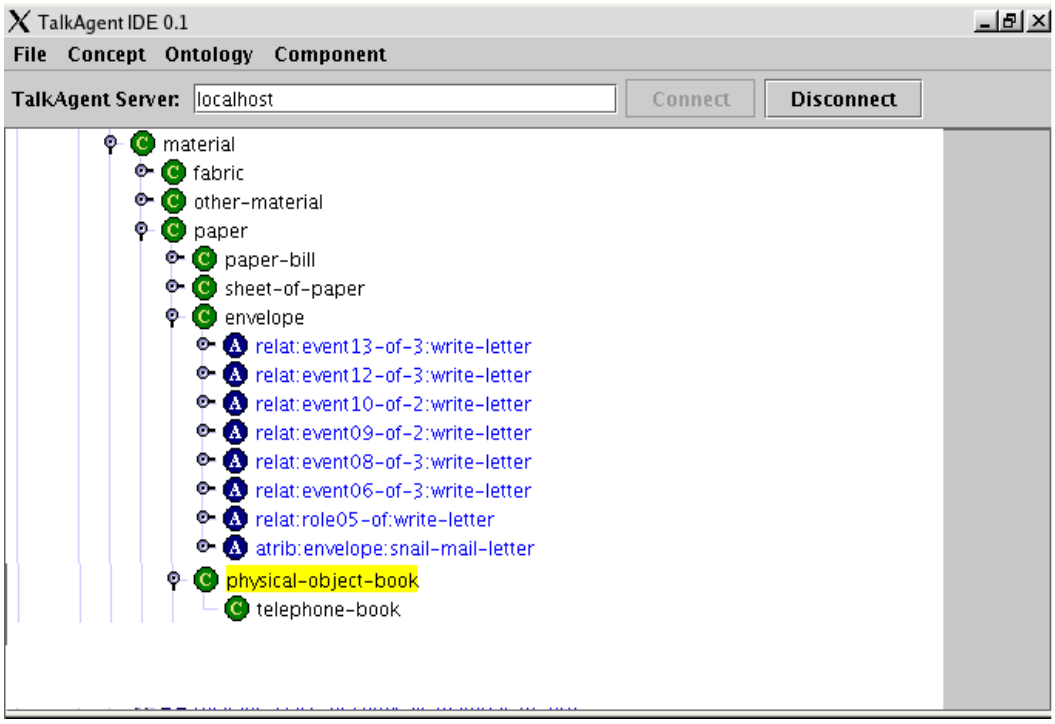


Figura 3.6 - Estrutura de uma ontologia TT composta por conceitos e asserções.

A *Figura 3.6* ilustra um trecho da estrutura da ontologia Thought Treasure composta por conceitos e asserções sobre os conceitos. Os tipos de relações suportadas pelo Thought Treasure podem ser arbitrárias e de qualquer aridade, mas existem algumas convenções para definição das asserções:

- A asserção *is* é um mecanismo de relação de atributos ou características de elementos. A asserção *is(tomato,spheric)* especifica, por exemplo, que um tomate é esférico.
- A asserção *atrib* e *part-of* são mecanismos de herança múltipla utilizados para definir que um elemento é parte de outro elemento maior. Um exemplo de asserção do tipo *part-of* é *part-of(our-universe,milk-way)* que especifica que nosso universo é parte da Via Láctea. Um exemplo de asserção *atrib* é *atrib(tomato,color,red)* que a cor vermelha é um atributo do tomate.
- As relações do tipo *role* e *event* são utilizadas em scripts que descrevem situações. A asserção *role(14,restaurant-bill,table)* especifica que no script 14 existe uma relação entre a conta de restaurante e a mesa. A asserção *event(19,confetti,wedding-ceremony)* indica que confete é o evento 19 no script que representa uma cerimônia de casamento.
- Os demais tipos de asserções são considerados como sendo do tipo *relat* e podem estabelecer qualquer tipo de relação entre os conceitos. A asserção *entry-condition(baseball-ticket,baseball-game)*, por exemplo, especifica que um ingresso de baseball é uma condição necessária para se assistir a um jogo de baseball.

3.7 - Considerações Finais

Para que agentes inteligentes possam "raciocinar" é necessário criar mecanismos para a representação, manipulação e inferência de conhecimento. A abordagem tradicional simbólica para representação de conhecimento utiliza linguagens de representação de conhecimento baseadas em lógica de primeira ordem ou ainda lógica proposicional. Tais linguagens permitem um controle exato sobre a inferência do conhecimento especificado. O uso de ontologias ocorre normalmente na fase de conceitualização da informação e é utilizada para garantir a semântica do vocabulário utilizado pelas linguagens de conhecimento. Contudo, ontologias são estruturas flexíveis o bastante e podem ser estendidas através de redes semânticas para representação de conhecimento. Dois exemplos do uso de redes semânticas para representação de conhecimento incluem os softwares Cyc e Thought Treasure que incluem quantidades massivas de informações de senso comum. Este trabalho propõe a extensão de ontologias e redes semânticas para representação de conhecimento, como será discutido nos capítulos a seguir. Devido à licença livre, à quantidade de informação e à disponibilidade de ferramentas de processamento de linguagem natural associadas, optou-se por utilizar a ontologia Thought Treasure como base de conhecimento padrão do protótipo desenvolvido como será discutido no *Capítulo 6*.

Capítulo 4 - Representação e Execução de Comportamentos em Agentes de Software

4.1- Considerações Iniciais

Os sistemas baseados em conhecimento tradicionais têm sido voltados para sistemas "fechados" que não têm uma interação direta com os domínios (problem domains) sobre os quais eles codificam conhecimento e resolvem problemas **[Maes 1994]**. A sua conexão com o ambiente real representado ocorre de maneira indireta através de um usuário humano. O usuário reconhece um problema em um domínio e descreve-o no sistema através de um modelo simbólico que o sistema entende. O sistema então retorna a descrição simbólica de uma resposta, que, por sua vez, deve ser executada pelo usuário no domínio. Em contraste, agentes de software são entidades autônomas capazes de realizar ações (comportamentos) que podem afetar e alterar o domínio no qual trabalham **[Maes 1994]**. Tais comportamentos podem ser acionados através de estímulos provindos do ambiente, raciocínio baseado em conhecimento ou alguma técnica de planejamento utilizada pelo agente **[Helin et al 2002]**.

Para facilitar o desenvolvimento rápido e eficiente de sistemas baseados em agentes é necessário representar e implementar uma série de comportamentos nestes agentes. Entende-se por comportamentos, a capacidade de executar tarefas. Serão analisadas neste capítulo algumas arquiteturas para representação de comportamentos em agentes.

4.2- Arquiteturas para Representação de Comportamentos

Arquiteturas para sistemas multi-agentes podem ser consideradas como sendo compostas por agentes individuais que interagem entre si **[Kolp et al 2001]**. A literatura especializada em agentes apresenta diversos modelos para representação de comportamentos em agentes e a forma de interação entre estes. A seguir, serão analisadas características de alguns dos principais modelos para representação de comportamentos existentes e será apresentado um novo modelo híbrido capaz de representar comportamentos e conhecimento de maneira integrada.

4.2.1- Agentes Reativos

O *emergentismo* é uma corrente filosófica naturalista que propõe que os fenômenos da mente (tal como a inteligência) estão situados no mundo material e podem ser descritos através de fenômenos físicos e químicos. O *emergentismo* proposto pelos filósofos C. Lloyd Morgans, Samuel Alexander e C. D. Broad afirma que o todo é maior que a soma da partes. Isto é, a partir da combinação de comportamentos simples, novos comportamentos emergem, contudo eles não podem ser executados ou explicados pelos comportamentos individuais. Pode-se fazer uma analogia entre comportamentos e elementos químicos, uma vez que ao se combinar certos elementos químicos novos elementos emergem, diferentes dos originais **[Susse 2003].**

O *emergentismo* filosófico passou a servir como modelo para as ciências cognitivas e para a inteligência artificial. O professor do MIT Marvin Minsky em seu livro a *'Sociedade da Mente'* propõe que a inteligência humana possa ser representada através de coleções enormes de agentes simples semi-autônomos **[Minsky 1988]**. Seguindo a corrente *emergentista*, o professor do MIT Rodney Brooks, argumenta que para construir um sistema inteligente é necessário ter suas representações baseadas no mundo físico real. Brooks se fundamenta no princípio de que a maioria dos animais possui algum grau de inteligência apesar deles não possuírem modelos simbólicos complexos da realidade. Segundo ele, o comportamento racional é fruto da interação que o agente têm com o ambiente. Assume-se que o comportamento emerge da interação entre os tipos simples de comportamentos. A proposta para construção de sistemas baseados no mundo real supõe um modelo de agentes capazes de receber estímulos e executar ações, possuindo um modelo mínimo necessário para que os agentes possam alcançar suas metas **[Brooks 1990]**. Tais tipos de agentes costumam ser conhecidos como agentes reativos pois normalmente apenas reagem aos estímulos recebidos do ambiente, sem realizar raciocínio simbólico complexo sobre o domínio.

Agentes reativos se comunicam através de sinais, mensagens ou estímulos. Seu comportamento é modelado como respostas das mensagens ou estímulos que recebem do meio. Os agentes reativos não costumam ter um modelo simbólico complexo do domínio, que costuma ser modelado através de regras ou máquinas de estado finito que gerenciam os estímulos recebidos **[Brena 2003]**. Os agentes reativos mais simples não mantém histórico dos estímulos recebidos e simplesmente respondem aos estímulos através da execução de ações como ilustra a *Figura 4.1*. Agentes reativos mais complexos podem manter histórico dos estímulos recebidos para representar seus estados internos.

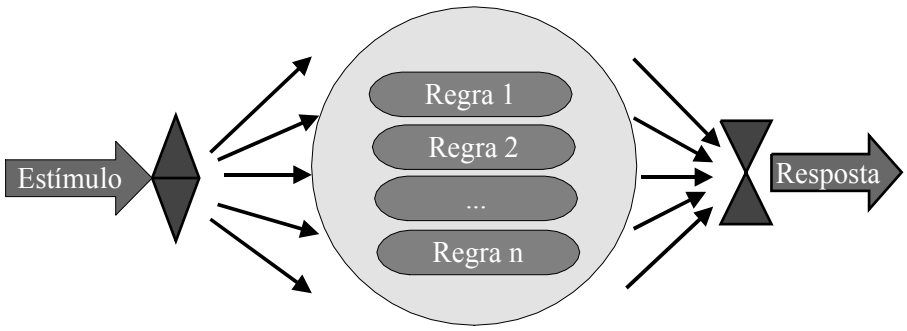


Figura 4.1 - Agente Reativo Simples

Uma abordagem biológico-social para o *emergentismo,* que pode ser aplicada a agentes reativos, é o conceito de *enxames* ou *swarms*. Segundo tal paradigma, a comunicação entre diversos agentes reativos simples permite a criação de enxames inteligentes. Tal modelo baseia-se na *teoria da sociedade de insetos,* segundo a qual a interação entre agentes permite encontrar soluções para problemas complexos, como por exemplo o de distribuição de trabalho entre formigas **[Maisonet 2003]**.

Métodos e algoritmos computacionais como *particle swarm optimization* (PSO) foram desenvolvidos a fim de implementar sistemas baseados em enxames de agentes reativos. A PSO é uma técnica evolucionária semelhante ao algoritmo genético desenvolvida pelos professores Russ Eberhart do departamento de Engenharia Elétrica da Universidade de Indianapolis e pelo psicólogo James Kennedy. Os agentes do sistema PSO iniciam com soluções aleatórias que são otimizadas pelos agentes do sistema. A escolha de novas soluções é feita baseada na melhor solução encontrada até o momento. O processo continua até que se obtenha uma solução satisfatória para o problema **[Eberhart et al 1995]**.

4.2.2 - Agentes Deliberativos

Diferentemente dos agentes reativos, os agentes deliberativos (ou agentes cognitivos) possuem um modelo simbólico elaborado o qual usam para raciocinar (*reasoning*) a respeito de um domínio. Mecanismos de planejamento costumam ser associados a estes agentes como forma dos agentes alcançarem seus objetivos **[Wooldridge et al 1995]**. Um exemplo de arquitetura de agentes deliberativos é a arquitetura *BDI* apresentada na subsessão a seguir.

4.2.2.1 - Agentes BDI

Em 1975, o filósofo norte-americano, professor de Harvard, Hilary Putnam publicou o artigo "Mentes e Máquinas", onde propõe uma analogia entre mentes, cérebros e computadores digitais. Putnam deu origem a corrente filosófica chamada *funcionalismo*, segundo a qual a mente seria uma função do cérebro do mesmo modo que é o software para o hardware **[Putnam 1975]**. Daniel Dennett, filósofo e professor da Tufts University, propôs a *teoria dos sistemas intencionais*, como uma versão específica do funcionalismo. Segundo Dennet, os estados mentais, sobretudo as crenças, desejos e intenções (que compõe a chamada *folk psychology*) nada mais são do que sistemas hipotéticos de conceitos articuladores, que são utilizados para tornar inteligível os comportamentos humanos. Assim, pode-se também atribuir intenções a sistemas artificiais, desde que sejam capazes de produzir comportamentos dotados de padrões mínimos de racionalidade **[Dennett 1971]**.

A notação intencional (composta por atitudes tais como crenças, desejos, intenções) serve como uma ferramenta de abstração, fornecendo uma forma de descrever, explicar e prever o comportamento de sistemas de agentes. A arquitetura BDI (Belief, Desire, Intention) proposta pelos pesquisadores Michael Georgeff e Anand Rao do Australian Artificial Intelligence Institute tem sua raiz na tradição filosófica de que o raciocínio e o processo de decisão ocorrem momento a momento, de forma a atingir metas estabelecidas. O processo de execução de ações envolve decidir que metas se deseja alcançar (tomada de decisão) e como se pode alcançar tais metas (planejamento). O processo de decisão depende da manipulação das estruturas que representam as crenças, desejos e intenções do agente. As intenções dos agentes persistem e influenciam as ações tomadas e as opções de ações possíveis. Um modelo para um sistema de agentes BDI é apresentado na *Figura 4.2*:

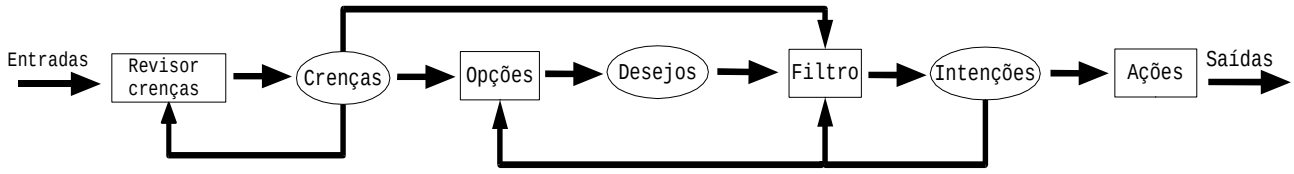


Figura: 4.2 - Exemplo de uma arquitetura para agentes BDI [Brena 2003]

Segundo o modelo, o conhecimento do sistema é expresso através das *crenças* (que podem ter um certo grau de incerteza associado). Novas crenças são acrescentados a medida que o sistema recebe novos estímulos (novas entradas). Os objetivos do sistema (metas) são representados através dos *desejos*. Os planos do agente para alcançar as metas propostas são expressos através das *intenções*. O revisor de crenças analisa a nova informação que chega ao sistema e atualiza a base de conhecimento (crenças) do sistema. Baseado no conjunto de crenças atuais, são geradas as opções possíveis de metas do sistema (desejos). Um filtro representa o sistema deliberativo do agente e determina as intenções do agente baseado nas crenças,

desejos e atuais intenções do agente. As ações são selecionadas baseadas no conjunto de intenções do agente e finalmente são executadas **[Brena 2003]**.

Diversas técnicas de representação de conhecimento e planejamento podem ser utilizadas para a criação de agentes deliberativos BDI. Uma abordagem para implementar a lógica multi-modal utilizada para definição de agentes BDI consiste em utilizar programação lógica (Prolog) para representação de conhecimento e planos hierárquicos para planejamento. Tal abordagem é utilizada pelas plataformas de agentes AgentSpeak **[Machado, et al 2001]** e X-BDI **[Mora et al 1998]**.

4.2.3 - Agentes Federados

A arquitetura de agentes federados propõe que agentes inteligentes sejam agregados em uma sociedade e estejam sobre o controle de um governo central. Este governo é composto por agentes intermediários responsáveis por coordenar os agentes inteligentes a fim de satisfazer as metas do sistema. Agentes intermediários são responsáveis por governar os agentes especialistas, executando funções como registrar, encontrar, rotear, monitorar e gerenciar agentes da sociedade de agentes **[Genesereth et al 1994]**.

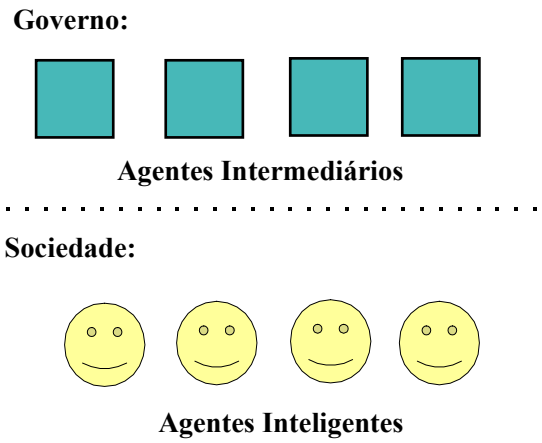


Figura 4.3 - Agentes Federados

A *Figura 4.3* ilustra uma sociedade de agentes federados. Cada agente inteligente da sociedade anuncia suas características para os agentes intermediários que se encarregarão de contactá-lo quando necessário. Os agentes intermediários provêm suporte ao fluxo de informação ajudando a localizar e conectar os provedores de informação aos solicitadores de informação. Os serviços prestados pelos agentes intermediários permitem que novos agentes se incorporem dinamicamente ao sistema **[Decker et al 1997]**. Exemplos de agentes intermediários incluem **[Shen et al 1997]**:

- *Facilitadores*: Agentes coordenadores que transferem informações entre agentes do sistema. A comunicação costuma acontecer entre os agentes inteligentes e seus facilitadores e entre facilitadores, mas não diretamente entre os agentes da plataforma. Os agentes facilitadores auxiliam a comunicação entre agentes roteando mensagens baseado em seus conteúdos, submetendo requisições apropriadas aos agentes do sistema. Ao gerenciar as mensagens recebidas os agentes facilitadores podem traduzir mensagens, subdividi-las em problemas menores e coordenar diversas atividades a fim de satisfazer as requisições.

- *Despachantes (Brokers)*: Agentes que recebem requisições e executam ações para outros agentes em conjunto com seus próprios recursos a fim de satisfazer requisições de terceiros. Agentes corretores são similares a agentes facilitadores com algumas funções adicionais, tais como monitoração e notificação. A diferença funcional entre um facilitador e um corretor é que o facilitador é responsável apenas por um determinado grupo de agentes, enquanto um corretor pode ser contactado por qualquer agente solicitando um determinado serviço.

- *Mediadores*: Agentes mediadores auxiliam a coordenação entre agentes do sistema. Além das características oferecidas pelos facilitadores e corretores agentes mediadores podem ter mecanismos de aprendizado que permitam facilitar a cooperação entre agentes.

- *Matchmakers* (páginas amarelas): Agentes que ajudam os agentes que solicitam serviços a encontrar agentes que os provêm baseado nas capacidades anunciadas. Os agentes matchmakers permitem que os agentes inteligentes do sistema encontrem o nome de um agente que oferece um serviço específico. Uma vez identificado o provedor do serviços os agentes podem se comunicar sem interferência do matchmaker.

Existem diversas plataformas que suportam agentes disponíveis comercialmente ou distribuídas de maneira livre. Exemplos de plataformas que utilizam arquiteturas federativas incluem FIPA-OS **[Emorphia 2001]**, Retsina MAS **[Sycara 1999]** e DECAF **[Graham et al 2000]**.

4.2.4 - Agentes Baseados em Componentes

A maioria dos modelos para representação de comportamentos em agentes prevê comportamentos diretamente associados aos agentes do sistema. Tais modelos se mostram inflexíveis para adicionar, combinar, alterar ou excluir novos comportamentos. Quando se deseja adicionar novos comportamentos, por exemplo, deve-se criar novos agentes ou estabelecer estratégias de comunicação e cooperação entre os

agentes do sistema a fim de gerar o comportamento desejado. Para alterar ou excluir comportamentos costuma ser necessário realizar mudanças custosas na estrutura do sistema de agentes.

A medida que sistemas multi-agentes vão se tornando mais complexos torna-se necessário criar estruturas que permitam o desenvolvimento de agentes de uma forma planejada aproveitando comportamentos existentes e facilitando a integração com outros agentes existentes no sistema. Uma forma para atingir este objetivo é a especificação de uma arquitetura para agentes composta por componentes reutilizáveis que dêem suporte aos comportamentos dos agentes.

Segundo as arquiteturas baseadas em componentes, os comportamentos são descritos através de ações implementadas pelos componentes. Um agente pode consistir de vários comportamentos simultâneos através da instanciação de diversos componentes. Diversos tipos de componentes podem ser agrupados de acordo com a sua utilidade a fim de facilitar a criação de novas aplicações. Tais componentes de software podem ser adicionados e combinados, criando uma estrutura facilmente gerenciável como ilustra a *Figura 4.4*.

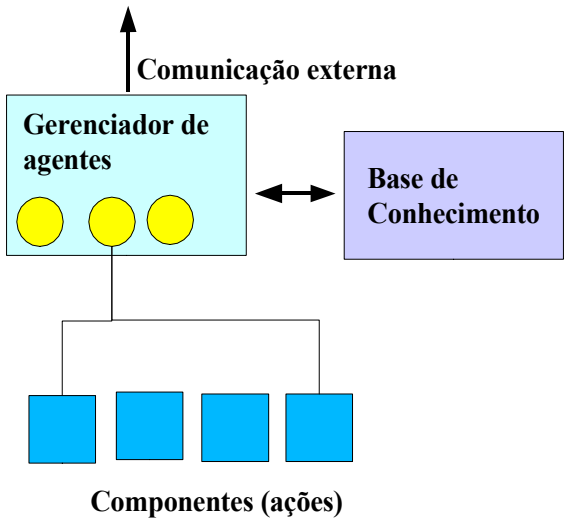


Figura 4.4 - Uma arquitetura baseada em componentes [Skarmeas 1999]

Como ilustra a *Figura 4.4*, um serviço de indexação de componentes permite que vários agentes compartilhem comportamentos e ações. Os componentes presentes na arquitetura de agentes podem ter sido desenvolvidos a priori ou ter sido projetados especificamente para uma determinada aplicação. Para fim de reuso e reaproveitamento, os comportamentos dos agentes podem ser implementados sob a forma de componentes de software genéricos que podem ser alterados ou estendidos **[Skarmeas 1999]**.

Componentes de software podem ser utilizados para implementar qualquer tipo de ação de acordo com as características e necessidades dos agentes. Um agente monitor de curso de educação à distância poderia ter, por exemplo, componentes que permitissem o gerenciamento de aulas, alunos, notas e trabalhos. Já um

agente secretário poderia, por sua vez, ter componentes para gerenciar compromissos, contatos, tarefas, projetos e outras atividades relacionadas ao seu domínio.
O uso de componentes para representação de comportamentos já se mostrou viável, como na arquitetura baseada em componentes proposta pelo professor Keith L. Clark do Imperial College de Londres **[Skarmeas 1999],** na arquitetura Gypsy desenvolvida pelo grupo de sistemas distribuídos da Universidade Técnica de Viena **[Lugmayr et al 2003]**, entre outras.

4.2.4.1 -Modelos de componentes

Com os avanços e difusão da programação orientada a objetos, componentes *off-the-shelf* podem ser utilizados e reutilizados para a construção de grandes sistemas computacionais. O reuso de componentes pode ser feito tanto a nível de projeto (*software patterns*) **[Buschmann 1996]** quanto em níveis mais concretos, através da implementação de componentes **[Nierstrasz et al 1992].**

O professor Robert Orfail da Universidade de San José (U.S.) **[Orfail 1998]** define um componente de software como sendo um objeto suficientemente genérico, que não esteja diretamente associado a uma aplicação específica, sistema operacional, infra-estrutura de rede ou ferramenta de desenvolvimento de aplicações. O desenvolvimento de software através de componentes é visto como uma extensão natural da programação orientada a objetos, visando maior reusabilidade e flexibilidade para criação de aplicações .

Uma das vantagens da utilização de componentes é a possibilidade de obtê-los ou comprá-los através de desenvolvedores independentes e integrá-los a aplicações desenvolvidas através de componentes. Existem vários modelos para o desenvolvimento de componentes de software sendo atualmente utilizados. As principais tecnologias incluem: o modelo COM/DCOM da Microsoft **[Chappel 1996]**, o modelo Corba da OMG **[Orfail 1998]** e o modelo OpenDoc da IBM **[Deri 2001]**. A linguagem Java também define dois modelos específicos para desenvolvimento de componentes: JavaBeans **[Horstmann et al 1998]** e Enterprise JavaBeans (EJBs) **[Allamaraju et al 2001]**.

A arquitetura padrão utilizada para o desenvolvimento de componentes em Java é a JavaBeans. Esta arquitetura foi desenvolvida por um consórcio composto por diversas empresas de software que incluiu a Apple, Borland, IBM, JustSystem, Microsoft, Netscape, RogueWave, Sun, Symantec, entre outras **[Sun MicroSystems 2001]**. A arquitetura de componentes JavaBeans é baseada em classes Java normais, que possuem uma convenção específica para definir o nome de propriedades, métodos e eventos. O mecanismo de introspecção baseado em reflexão oferecido em Java permite que outros componentes ou aplicações acessem os recursos oferecidos por um JavaBean. Não é necessário definir nenhum arquivo de definição da interface (IDL) para o componente. Caso se deseje ignorar a nomenclatura padrão estabelecida para os

JavaBeans, é possível personalizar manualmente a interface BeanInfo definindo explicitamente como o componente JavaBean publicará seus métodos **[Horstmann et al 1998]**.

JavaBeans é uma arquitetura para definição de componentes bastante flexível. Os componentes desenvolvidos segundo esta arquitetura podem fazer uso de qualquer outra classe definida em Java e podem utilizar como container padrão de execução qualquer máquina virtual Java. Assim, componentes Javabeans podem ser facilmente executados dentro de qualquer aplicativo Java, como um ambiente de execução de agentes Java, por exemplo. JavaBeans podem também ser executados em containers de aplicativos não Java, como por exemplo containers ActiveX, presentes em alguns sistemas operacionais **[Sun MicroSystems 1998]**.

4.2.4.2 - Agentes Atômicos

Os agentes das arquiteturas baseadas em componentes possuem uma descrição explícita dos serviços (ações) oferecidos, especificados à priori pelo desenvolvedor do sistema. O uso de redes semânticas ao invés de lógica de primeira ordem ou lógica proposicional como sistema de representação de conhecimento dos agentes permite uma nova abordagem para a organização de arquiteturas baseadas em agentes, mais flexível e mais apropriada para resolução de alguns tipos de problemas. Por isto, propõe-se aqui um novo modelo para criação de agentes chamado *agentes atômicos* que explora o uso de redes semânticas para fazer a relação entre o modelo de conhecimento dos agentes e suas ações sobre o mundo.

O modelo de agentes atômicos, aqui proposto, tem por inspiração a filosofia reducionista de Aristóteles e Democritus, segundo a qual os comportamentos podem ser reduzidos à combinação de componentes menores. Isto é, o todo é a soma das partes **[Green 1998]**. De acordo com a arquitetura baseada em componentes, as ações básicas que os agentes podem executar são representadas através de componentes e estão associadas aos seus símbolos representativos (átomos).

Da mesma forma que Aristóteles propunha que o desenho de uma casa pode representar a casa, propõe-se que o conceito 'casa' represente o objeto do mundo real casa e que os comportamentos associados a ela (pintá-la, comprá-la, vendê-la, etc) sejam representados por componentes associados, como mostra a *Figura 4.5*.

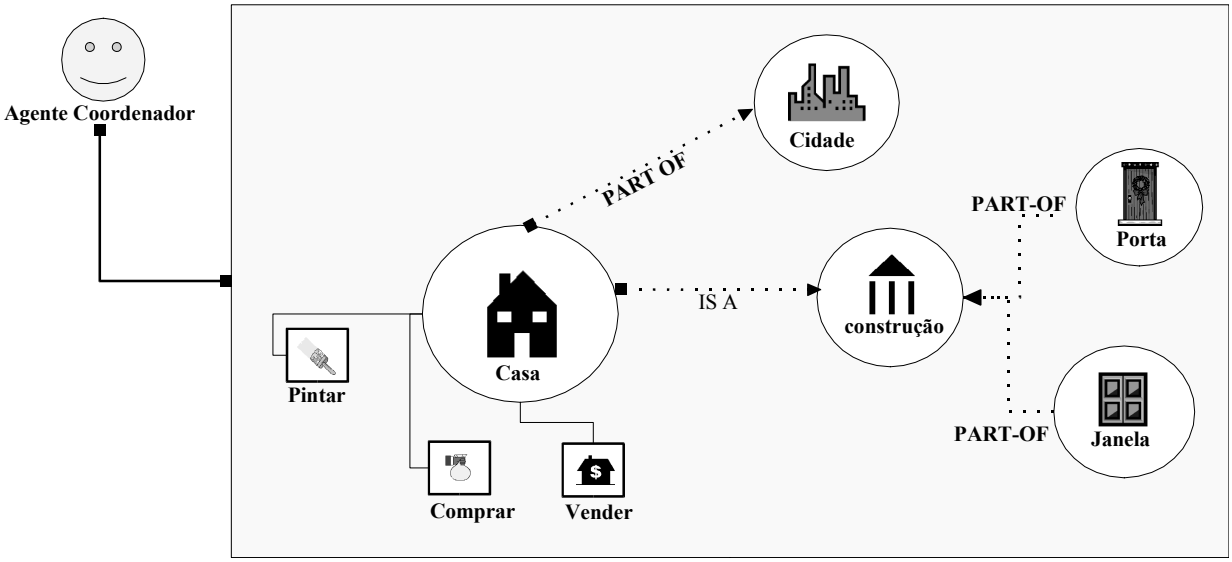


Figura 4.5 - Agentes Atômicos

Um agente atômico é composto por uma série de componentes cujas funcionalidades contribuem para o comportamento geral do agente. Os átomos, que também representam o conhecimento do sistema, são ligados entre si através de uma rede semântica. Esta rede semântica mantém informação compartilhada pelos agentes atômicos, funcionando como uma memória global do sistema. Esta base de conhecimento pode ser manipulada pelos componentes do sistema através do envio de mensagens para o agente coordenador.

O agente coordenador do sistema possui um mecanismo de gerenciamento dos átomos (conceitos) baseado em funções que analisam padrões das mensagens recebidas pelo sistema. O componente gerenciador de mensagens decide quais métodos dos componentes devem ser ativados, realizando um planejamento básico das ações necessárias para satisfazer as metas. Adicionalmente, os componentes podem fazer planejamento interno para executar as ações necessárias.

Os agentes atômicos guardam algumas semelhanças com os *agentes* definidos por Minsky em 'Sociedade da Mente' **[Minsky 1995]**. Tais agentes possuem comportamento limitado e possuem conexões entre si através de redes semânticas, as quais podem ser comparadas com as redes de interações de uma sociedade de agentes. Contudo, os agentes atômicos, possuem várias características que os diferenciam dos agentes propostos por Minsky e os aproximam de *agências. Os agentes atômicos* não são puramente reativos e podem possuir inteligência própria independente do agente coordenador. Uma vez que um comportamento de um agente atômico é ativado, ele pode passar a exercer atitudes pró-ativas como obter, armazenar e analisar informação da Internet ou realizar transações (como pagar contas, comprar ações, produtos, etc). Os comportamentos implementados através dos componentes também podem ter certa inteligência definida através de métodos de planejamento que descentralizam o processo de decisão do agente coordenador.

4.2.4.3 - Extended Knowledge Network

A fim de implementar o modelo de agentes atômicos propõe-se a criação de uma estrutura de dados capaz de satisfazer as características do modelo ilustrado na *Figura 4.6*. Propõe-se como estrutura básica para base de conhecimento o uso de redes semânticas assercionais e executáveis como as definidas pelo pesquisador John Sowa da IBM **[Sowa 2003]**. As características especificas deste tipo de representação permitem estabelecer relações entre os conceitos (nós) da rede e procurar padrões específicos baseados na estrutura da rede. Tal representação guarda semelhanças estruturais com os grafos conceituais definidos por Sowa e os grafos existenciais propostos pelo matemático e filósofo Charles Sanders Peirce **[Sowa 2001]**.

A estrutura aqui proposta, chamada Extended Knowledge Network (EKN), consiste de um grafo unidirecional composto por conceitos, relações conceituais e relações comportamentais. Os conceitos representam os elementos semânticos básicos da base de conhecimento. As relações conceituais estabelecem relações entre dois conceitos e são compostas por um tipo e um conjunto de parâmetros ou atributos associados. Relações comportamentais são estabelecidas entre conceitos e módulos de comportamento (que podem ser representados por um nome de um componente, um nome de um método, ou um parâmetro de um determinado método).

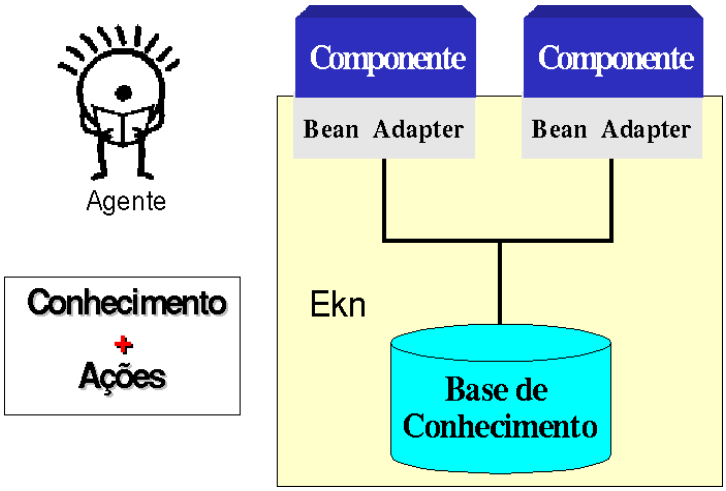


Figura 4.6 - Visão Geral da Extended Knowledge Network (Ekn)

Os comportamentos do sistema são implementados através de componentes Javabeans. Tais componentes apresentam características especificas que facilitam sua integração à EKN. A introspecção automática permite, por exemplo, extrair as características de um componente executável sem a necessidade de se descrever formalmente o componente. A introspecção existente nos componentes Javabeans permite obter os nomes dos métodos e parâmetros através da própria estrutura do componente de maneira automática, como ilustra o *Bean Adapter* da *Figura 4.7*.

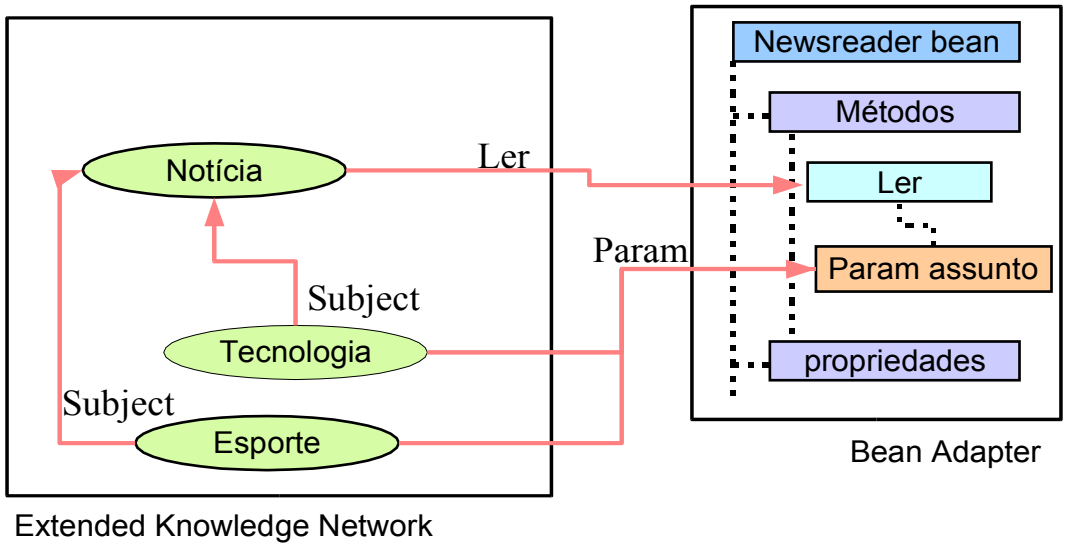


Figura 4.7: Extended Knowledge Network

A *Figura 4.7* ilustra a associação dos elementos presentes no Bean Adapter com conceitos representados na Extended Knowledge Network. A figura descreve algumas relações conceituais tais como "tecnologia e esporte são assunto para notícias". A *Figura 4.7* também ilustra relações comportamentais entre os conceitos e o JavaBean *NewsReader*. O conceito notícia está associado ao método *"ler"* do JavaBean NewsReader através da relação comportamental *"ler"*. O método ler requer um parâmetro que, por sua vez, possui relações comportamentais do tipo *"param"* com os conceitos tecnologia e esporte. É importante ressaltar que não existe tipagem explícita dos tipos de relação, de forma que, se os conceitos tecnologia e esporte estivessem ligados à notícia por outra relação que não *subject,* isto não impediria que tais conceitos fossem relacionados ao parâmetro *subject* do método *ler*.

A Extended Knowledge Network representa as relações e conceitos através de objetos que podem ser armazenados em uma base de dados orientada à objetos que será discutida com maior detalhe no *Capítulo 6*.

4.3- Considerações Finais

Agentes de software são entidades ativas as quais apresentam comportamentos, que os permitem executar ações. A literatura especializada em agentes apresenta diversos modelos de representação de comportamentos baseados em diversas filosofias distintas. Duas correntes antagônicas, os agentes deliberativos e os agentes reativos discutem sobre a necessidade de um modelo simbólico complexo do domínio versus uma abordagem minimalista de modelagem de domínio associada a capacidade reativa dos agentes.

O modelo de representação de comportamentos, proposto para este trabalho, é um modelo híbrido que utiliza características dos agentes reativos (estímulos, coordenação externa) e dos agentes deliberativos (representação explicita do conhecimento através de redes semânticas). O modelo aqui proposto, chamado 'agentes atômicos', inspirado no reducionismo aristotélico e na *"Sociedade da Mente"* de Minsky **[Minsky 1995]**, utiliza redes semânticas estendidas para representação de conhecimento e de comportamentos.

A Extended Knowledge Network é uma estrutura de dados proposta para dar suporte ao modelo de agentes atômicos. Tal estrutura baseada em grafos unidirecionais compostos por objetos e ligações será discutida com maior detalhe no *Capítulo 6* .

Capítulo V - Comunicação usuário-agente

5.1 - Considerações Iniciais

A interface gráfica (GUI) como meio de interação ser humano - computador teve seus primeiros protótipos desenvolvidos em 1971 no laboratório PARC (Palo Alto Research Center) da empresa norte-americana Xerox. O advento dos computadores pessoais (PCs) no início da década de 80 popularizou o uso da interface gráfica através dos computadores Macintosh e seus posteriores clones. O novo paradigma introduzido, baseado na manipulação direta de ícones e outros elementos gráficos, facilitou a utilização dos computadores por usuários leigos. Tal mudança teve profundo impacto na computação pessoal e comercial. Graças à interface gráfica, usuários domésticos passaram a utilizar computadores para agilizar tarefas do dia-a-dia como escrever textos e documentos, controlar finanças pessoais, compromissos e outras tarefas quotidianas. A interface gráfica teve também grande impacto nas empresas, onde funcionários não especialistas em computação começaram a utilizar softwares na automação dos mais diversos tipos de processos presentes nas atividades econômicas.

Os computadores hoje estão tão ubíquos quanto automóveis e eletrodomésticos, mas ainda não se tornaram tão fáceis de usar. Embora as GUIs tenham trazido inúmeras vantagens em relação a geração anterior de interfaces baseadas em linha de comando, as interfaces gráficas não são perfeitas. Cada vez mais as pessoas passam seu tempo em frente a computadores trocando correspondência, trabalhando, fazendo compras, se entretendo e a grande quantidade de softwares e interações que são necessárias podem se tornar um obstáculo à ampla adoção destas tecnologias. O overload cognitivo já se mostra por exemplo em softwares como suites de escritórios que têm hoje muito mais ferramentas do que as pessoas conseguem usar. A medida que novas tecnologias como tvs interativas, palmtops e celulares começam a fazer parte do quotidiano, a necessidade por interfaces mais eficientes se faz mais necessária para o real uso do poder destas tecnologias **[Maes 1997]**.

5.2 - Paradigma de Manipulação Direta

Atualmente os computadores usam GUIs que implementam o paradigma de manipulação direta através da qual usuários manipulam objetos da interface com ações ou comandos. Um exemplo típico do uso deste paradigma consiste em um usuário mover um arquivo de um diretório para outro através de um click de mouse que arrasta o arquivo para o diretório destino. Nada costuma acontecer até que o usuário insira novos comandos através do teclado, mouse, ou outro dispositivo de entrada. O computador funciona como uma

entidade passiva esperando para executar instruções com alto grau de detalhe. Os softwares atuais não oferecem muita ajuda para tarefas complexas que não tenham controle direto do usuário, como busca ou gerenciamento de informações, por exemplo.

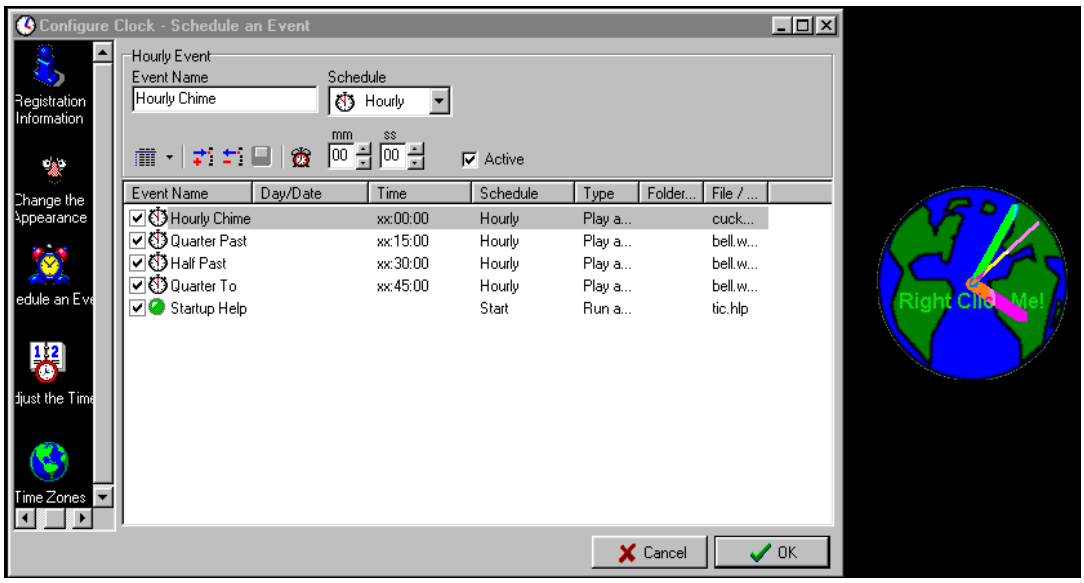


Figura 5.1 - Interação usuário computador através de uma GUI

Embora as GUIs facilitem o uso de aplicativos por usuários, estas apresentam algumas limitações. Um exemplo da limitação das GUIs é o problema da generalização. A maioria das GUIs é desenvolvida para uma *aplicação* específica tendo seu uso limitado a uma tarefa específica. O usuário normalmente escolhe uma das aplicações existentes em seu computador, realiza alguma tarefa permitida pela aplicação e então inicializa uma nova aplicação para realizar outro tipo de trabalho. Cada aplicação tem sua própria interface e as aplicações podem ter funções redundantes **[Lieberman 1998]**.

A *Figura 5.1* ilustra um agente que permite apenas gerenciar compromissos. O usuário não pode utilizar a GUI para manipular telefones se a lógica do software e a GUI assim não o permitirem a priori. A menos que o software faça uso de plug-ins é difícil extender a utilidade deste sem a necessidade de recriá-lo .

O paradigma de manipulação direta envolve também problemas para gerenciar variáveis, ou distinguir mais de um elemento de um conjunto ou classe de elementos. O problema da manipulação direta é que o usuário deve manipular diretamente todos os objetos que deseja utilizar. Ao invés de fornecer comandos executivos de alto nível, o usuário é reduzido ao trabalhador de linha de produção que têm que fornecer o mesmo comando, de maneira monótona, para alcançar um objetivo determinado. Este trabalho repetitivo deveria ser feito pelo software ao invés do usuário. Um exemplo de trabalho repetitivo é a conversão de um conjunto de imagens de um formato para outro. A menos que o software de manipulação de imagens

ofereça uma macro ou se escreva um script, cabe ao usuário converter as imagens uma por uma, repetindo o mesmo comando infindáveis vezes até alcançar seu objetivo.

Para tornar os computadores mais fáceis de usar, estes devem se tornar mais sensíveis as necessidades dos usuários. O software ideal deveria entender quais são os objetivos do usuário e agir para satisfazê-los. Se o usuário pudesse delegar tarefas ao invés de manipular diretamente a informação, o software seria mais fácil de usar escondendo detalhes técnicos das tarefas sendo realizadas, guiando usuários através de problemas complexos. Contudo o modelo atual não é capaz de oferecer isto, pois as aplicações não são sensíveis a contexto, têm uso específico e não apresentam a pró-atividade necessária para ajudar o usuário na execução de suas tarefas **[Maes 1995]**.

5.3 - Interfaces com Suporte Ao Uso de Linguagem Natural e com Senso Comum

O uso de linguagem natural é um dos diversos tipos de estilos para interação usuário computador. Existe um grande apelo em poder se interagir com um software usando o mesmo tipo de linguagem utilizada na interação entre seres humanos no dia-a-dia. A possibilidade do usuário se comunicar diretamente com o software pode aumentar, de maneira expressiva, a facilidade de uso dos computadores. Uma interface baseada em linguagem natural irrestrita permitiria aos usuários aprender e utilizar a interface de maneira muito simples, pois a estrutura e vocabulário já seriam conhecidos do usuário. Devido ao fato da mesma linguagem ser utilizada para diversas aplicações, haveria menor dificuldade em utilizar aplicações diferentes. Usuários poderiam ter liberdade ao utilizar software solicitando o que eles realmente necessitam, como por exemplo solicitar ao processador de texto: '*Formate meu artigo de acordo com o padrão ISOxxx*' ou ao gerenciador de finanças pessoais: '*Pague meu aluguel, tão logo eu tenha meu salário creditado em minha conta*'.

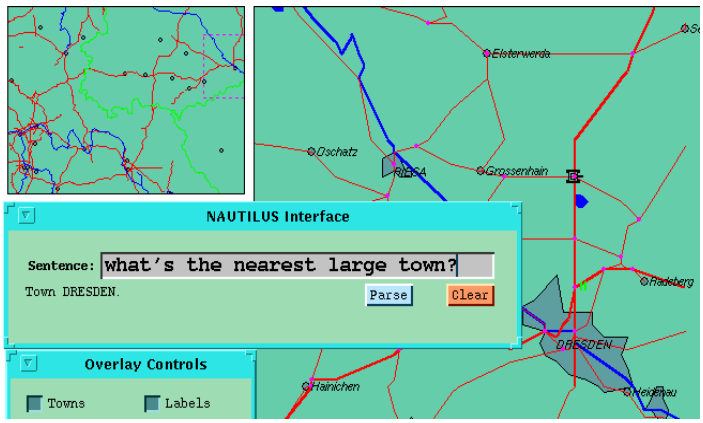


Figura 5.2 - Interface cartográfica com suporte ao uso de linguagem natural [Wauchope 1999]

Outra grande utilidade para o processamento de linguagem natural é a recuperação de informações textuais não estruturadas (documentos, relatórios, páginas web,etc). Se o usuário puder comunicar seus objetivos através de linguagem natural, este poderá, teoricamente, encontrar e manipular informação de maneira mais precisa. Contudo, qualquer pessoa que já tenha digitado uma consulta em linguagem natural para o assistente do Microsoft Office ou em uma máquina de busca na web já deve ter percebido a dificuldade que os softwares atuais possuem em compreender linguagem natural. Um dos problemas do uso de linguagem natural como interface ser humano-computador é que a linguagem natural é bastante ambígua. Outro grande problema é que a informação do domínio para o qual se faz o uso de linguagem natural, muitas vezes não está semanticamente estruturada, o que dificulta processar uma consulta, mesmo quando esta é compreendida corretamente **[Laird et al 1995]**.

Para solucionar estes problemas e possibilitar o uso de linguagem natural para comunicação usuário-computador é preciso:

- Criar ou utilizar ferramentas de processamento de linguagem natural que permitam ao usuário se comunicar com softwares de maneira não ambígua.
- Permitir que a aplicação tenha conhecimento especialista do domínio sobre o qual o usuário deseja atuar.
- Permitir que a aplicação tenha conhecimento de senso comum para resolver ambigüidades.

Para resolver as ambigüidades existentes é necessário reconhecer não somente a relação entre as palavras, mas a semântica dos conceitos relacionados às palavras. Um dos problemas do uso de linguagem natural para comunicação usuário-agente é que nem sempre tem-se disponível uma modelagem semântica do domínio que permita a compreensão do contexto da comunicação **[Long 1994]**.

Alguns dos softwares mais primitivos já criados podiam resolver problemas bastante complexos em domínios específicos (resolução de problemas numéricos, simulações de problemas físicos, etc). Contudo, os sofisticados softwares atuais não têm conhecimento de fatos que crianças pequenas sabem. O problema é que os softwares, mesmo os sistemas especialistas, não possuem conhecimento de senso comum. Qualquer criança aprende milhares de palavras, mas os softwares não costumam conhecer nenhuma delas, impedindo que compreendam sentenças em linguagem natural. Uma criança pode conhecer 10.000 palavras, cada uma ligada de vários modos a outras estruturas de conhecimento. Cada ligação por sua vez, está conectada a outras ligações, formando uma rede semântica complexa **[Minsky 2000]** .

Situações ligadas a um contexto baseado em senso comum parecem obvias a um ser humano, mas não a um software comum. Caso se pergunte a um adulto: "Há água na geladeira?", a maioria dos adultos considerá a possibilidade de haver uma garrafa de água dentro da geladeira. A existência de moléculas de água em uma alface na geladeira seria considerada uma circunstância fora de contexto. Circunstâncias normais de uso da

linguagem como esta, são baseadas no conhecimento de senso comum e se assume que as pessoas que se comunicam possuam noção do contexto utilizado **[Inman 2002]**.

Como os softwares atuais não têm conhecimento de contexto estes são capazes de gerenciar apenas solicitações cujo tipo de entrada seja conhecido, falhando nos demais casos. Para que informação possa ser processada como conhecimento é preciso que esta incorpore relações entre idéias. E para que o conhecimento seja útil as ligações descrevendo os conceitos devem ser facilmente acessadas, atualizadas e manipuladas.

Infelizmente, não é tão simples assim adicionar todo o conhecimento especialista e de senso comum para a criação de interfaces inteligentes baseadas em linguagem natural. Muitas vezes a solução possível é restringir o domínio da aplicação fornecendo o conhecimento necessário ao funcionamento desta. O software *Interlace* desenvolvido pelo Laboratório de Pesquisa da Marinha Norte-Americana e ilustrado na *Figura 5.2* alia o uso de GUIs ao de interfaces de linguagem natural para facilitar a visualização de mapas cartográficos. O projeto optou por restringir a entrada de linguagem natural a uma sentença por vez e o vocabulário foi especialmente construído por 12.000 conceitos necessários ao domínio da aplicação **[Wauchope 1999]**.

Como qualquer outro tipo de interface, a interface baseada em linguagem natural possui suas limitações. Seu uso nem sempre é o mais direto possível, requerendo as vezes que o usuário digite muitas informações para realizar uma dada tarefa requerida. Outro problema visto é que os usuários de interfaces baseadas em linguagem natural tendem a antropomorfizar o software, o que gera decepções por parte dos usuários ao descobrir que o software nem sempre se comporta como um ser humano.

5.4- Abordagens para Processamento de Linguagem Natural

Uma interface baseada em linguagem natural não necessita possibilitar que o usuário converse diretamente sobre qualquer assunto com ela. Ao invés disto, a característica principal das interfaces baseadas em linguagem natural é que um usuário não necessita aprender de maneira explícita o léxico e a sintaxe do sistema, de forma que este possa expressar o que deseja fazer na forma que está acostumado. Analisar-se-á nesta seção algumas abordagens factíveis para o processamento de linguagem natural que podem ser integradas a interfaces e agentes inteligentes.

O processamento de linguagem natural é uma tarefa complexa que envolve determinar diversos tipos de relações entre as palavras das sentenças para extrair o significado destas. O entendimento da linguagem natural pode se dar em vários níveis **[Martin 2000]**:

- Nível Morfológico: Investiga como as palavras são construídas de elementos mais básicos.
- Nível Sintático: Visa determinar a relação (papel) de um conjunto de palavras em uma sentença.
- Nível Semântico: Visa determinar o significado e inter-relacionamento semântico das palavras.
- Nível Discursivo: Visa determinar o significado de um conjunto de sentenças.
- Nível Pragmático: Visa determinar o objetivo do uso da língua.

Quanto maior o nível de entendimento, mais conhecimento sobre a linguagem se faz necessário. Para gerenciar a complexidade do entendimento da linguagem em diversos níveis, o processamento de linguagem natural utiliza técnicas computacionais aliadas a técnicas estatísticas e lingüísticas.

Atualmente existem dois principais paradigmas para o processamento de linguagem natural: o paradigma racionalista e o paradigma empírico. Os termos racionalista e empírico derivam do campo da Filosofia e são duas correntes epistemológicas que possuem opiniões antagônicas sobre a origem do conhecimento. De acordo com os filósofos adeptos da corrente empírica (Hobbes, Locke, Hume e outros) o conhecimento é adquirido através dos sentidos. Já os filósofos racionalistas (Descartes, Leibniz, Spinoza e outros) acreditam que o conhecimento é adquirido através da razão inata ao homem **[Esfeld 2002]**.

A abordagem empírica supõe que a linguagem deriva da capacidade humana de organizar e gerenciar a informação que recebe. Segundo o paradigma empírico a associatividade das palavras pode determinar a semântica e contexto. Pode-se, supostamente, criar um modelo simples da linguagem e induzir os parâmetros específicos através de técnicas estatísticas, reconhecimento de padrões e aprendizado de máquina **[Weber 2002]**.

A abordagem racionalista derivada do trabalho do professor do MIT Noam Chomsky **[Chomsky 1976]** assume que a linguagem é uma capacidade inata do homem e pode ser representada por um conjunto complexo de regras. Segundo o paradigma racionalista seria possível criar uma *gramática universal* capaz de representar a essência de todas as linguagens humanas. A abordagem racionalista Chomskina mostrou-se muito útil para a criação de gramáticas para análise sintática.

Para cobrir todos os níveis de compreensão da linguagem natural, algumas ferramentas de processamento de linguagem natural fazem uso de combinações de técnicas racionalistas e empíricas. Analisar-se-á na sessão seguinte algumas características presentes em algumas ferramentas de linguagem natural.

5.5- Ferramentas para Processamento de Linguagem Natural

Esta seção analisa algumas características de ferramentas de processamento de linguagem natural que podem ser utilizadas para criação de interfaces inteligentes. Espera-se de tais ferramentas a capacidade de processar sentenças simples em linguagem natural e a noção do contexto semântico da linguagem utilizada no domínio.

5.5.1- Nautilus

A ferramenta de processamento de linguagem natural Nautilus (Navy AUTomated Intelligent Language Understanding System) vem sendo desenvolvida pelo Laboratório de Pesquisa em Inteligência Artificial da Marinha dos EUA (NCARAI) desde 1986. A ferramenta escrita em Common LISP é composta pelo parser léxico-sintático Proteus desenvolvido pela Universidade de Nova York **[Sekine et al 1996]**, pelo interpretador semântico Tinsel desenvolvido pelo NCARAI **[Wauchope 1999]**, pela ferramenta de resolução de referências Focus desenvolvida por **[Levow 1998]** e pela ferramenta Funtran **[Wauchope et al 1997]** que transforma a sentença em linguagem natural em comandos que podem ser executados pelo sistema. A *Figura 5.3* ilustra a arquitetura do sistema Nautilus:

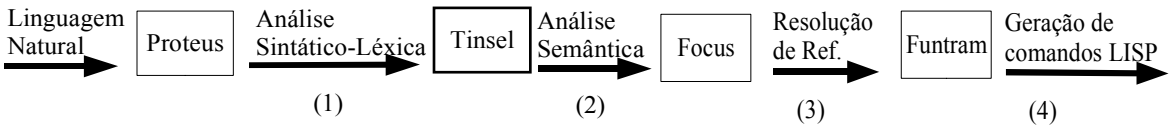


Figura 5.3 - Arquitetura do PLN Nautilus

A *Figura 5.4* ilustra as transformações ocorridas através dos passos de compreensão de linguagem natural através do sistema Nautilus. A consulta: '*Todos os F14s estão se movendo para a estação onde o piloto 1 está esperando?'* é convertida para uma estrutura LISP pela ferramenta Proteus. Em seguida as ambigüidades e resoluções de objetos específicos são realizados pelas ferramentas Tinsel e Focus. Por fim, a sentença é convertida em um formato entendido pelo sistema através da ferramenta Funtram. O processo é ilustrado pela *Figura 5.4*:

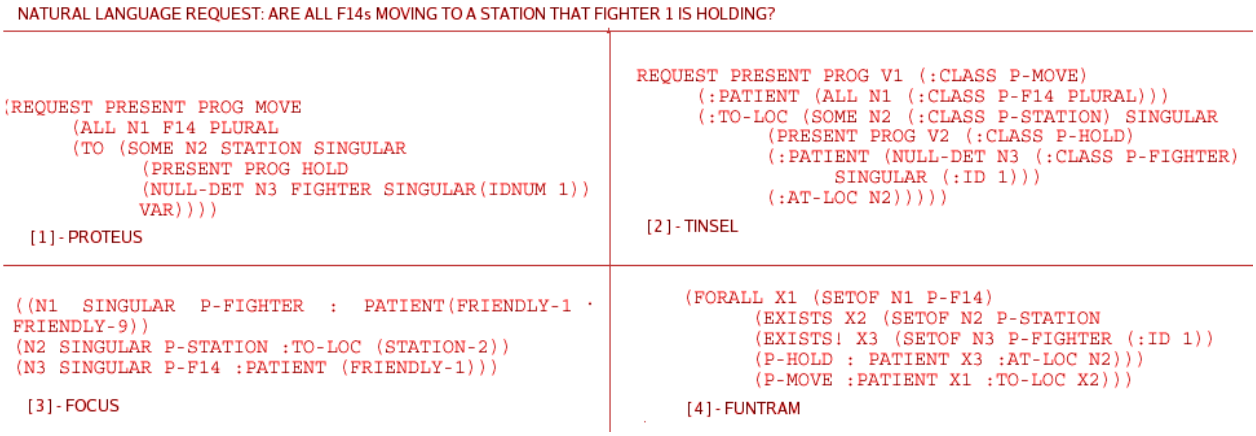


Figura 5.4 - Compreensão de linguagem natural através do sistema Nautilus

A ferramenta Nautilus é uma ferramenta de arquitetura aberta porém proprietária. O NCARAI não disponibiliza nem mesmo uma cópia executável do projeto, embora alguns componentes da arquitetura (como o parser Proteus) possam ser obtidos livremente.

5.5.2 - Thought Treasure

Thought Treasure (TT) é uma ferramenta de processamento de linguagem natural cujo conhecimento (léxico, sintático e semântico) é armazenado em ontologias. As ontologias TT são compostas por conceitos e relações que têm associados a si informações léxicas e sintáticas utilizadas pelos algoritmos de processamento de linguagem natural da ferramenta **[Ogbuji 2000]**. A parte da arquitetura do Thought Treasure responsável por interpretar linguagem natural é dividida em módulos chamados componentes e agências como ilustrado na *Figura 5.5*:

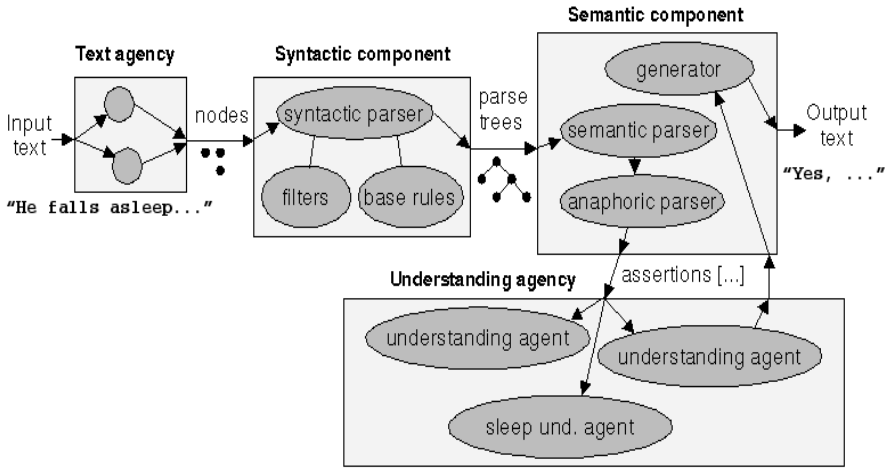


Figura 5.5 - Módulos da Ferramenta Thought Treasure [Mueller 1998]

Como demonstra a *Figura 5.5*, o processo de compreensão de linguagem natural é quebrado em quatro fases: *Agência Textual* (Text Agency), *Componente Sintático*, *Componente Semântico* e *Agência de Entendimento*.

A agencia textual lê o texto de entrada e produz uma série de símbolos que representam palavras, frases e outras entidades do texto. Em seguida o componente sintático cria a árvore sintática a partir dos símbolos gerados pela agência textual. O parser sintático Thought Treasure é um parser simples que faz análise do tipo bottom-up. As *base rules* (regras de transformação de base) selecionam símbolos léxicos que servirão como entrada para um conjunto de regras de transformação estabelecidas. Os filtros ilustrados na *Figura 5.5* impõem condições às transformações realizadas de forma que a estrutura gerada seja válida.

O componente semântico é composto por três componentes principais: o *parser semântico* responsável por transformar uma árvore sintática em uma rede semântica, o *parser anafórico* responsável por resolver a dependência de elementos como pronomes dentro de uma sentença e entre sentenças e o *gerador* capaz de transformar redes semânticas novamente em sentenças. Por fim, a agência de entendimento é responsável por resolver as ambuigüidades existentes nas sentenças utilizando o conhecimento de senso comum armazenado nas ontologias em conjunto com algoritmos de entendimento causais, temporais espaciais e lógicos **[Mueller 1998]**.

A ferramenta Thought Treasure é distribuída livremente junto com seu código fonte escrito em C para Unix. API especiais permitem acessar as funcionalidades da ferramenta através de outras linguagens de programação tais como Java, Python, Perl e TCL.

5.5.3 - Universal Networking Language

O projeto Universal Networking Language (UNL) é financiado pela Universidade das Nações Unidas, mais particularmente pelo Instituto de Estudos Avançados, com sede em Tóquio. Seu principal objetivo é promover e facilitar a comunicação internacional por meio do uso de sistemas computacionais de processamento automático de linguagem natural (PALN) **[Sossolote et al 1997]**.

A Universal Networking Language é uma interlíngua, isto é uma linguagem intermediária utilizada para tradução entre idiomas. A UNL foi concebida para representar de forma única o conteúdo semântico de uma sentença escrita em qualquer língua natural. Mais especificamente, a UNL é uma metalinguagem que serve para descrever aspectos especiais do significado de sentenças, tais como as relações semânticas que podem ser representadas por relações formais (morfológicas ou sintáticas) entre palavras de uma sentença. **[Oliveira et al 2001]**.

O Projeto UNL tem por objetivo disponibilizar codificadores e decodificadores de diversos idiomas (árabe, alemão, chinês, espanhol, francês, hindi, indonésio, inglês, italiano, japonês, letão, mongol, português, russo e tailandês) para UNL. Segundo esta abordagem, ao invés de se realizar a tradução direta de um idioma para outro, realiza-se a codificação do conteúdo de um texto de um dado idioma para a interlíngua UNL. O texto já codificado em UNL pode então ser decodificado para a língua destino. Já foram assinados contratos entre universidades, institutos de pesquisa e empresas de diversos países, o que distingue o projeto UNL como uma tentativa de conjugar esforços de especialistas em processamento automático das línguas naturais (PLN) do mundo todo. **[Uchida et al 2001].** A *Figura 5.6* ilustra o uso dos codificadores e decodificadores UNL.

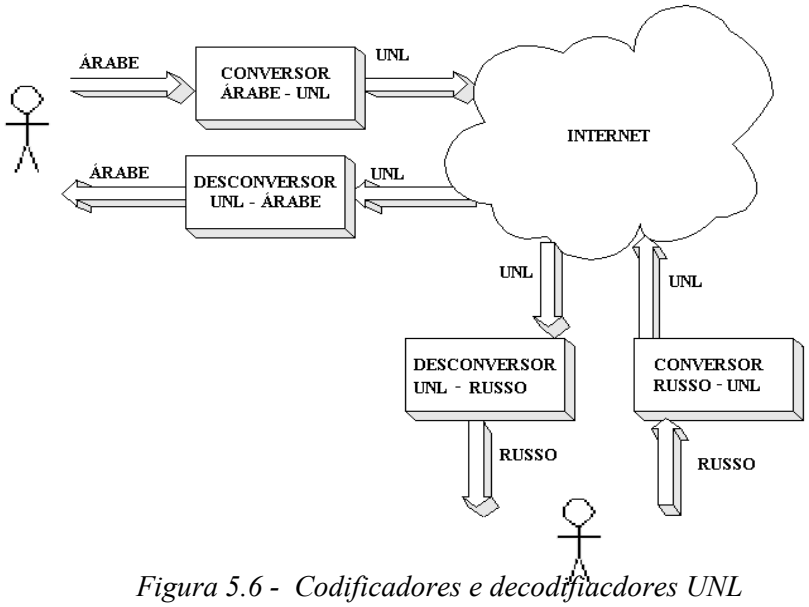


Figura 5.6 - Codificadores e decodificadores UNL

A linguagem UNL é representada através de redes semânticas compostas por conceitos padrões ou Universal Words (UWs) e um conjunto de relações conceituais e atributos que podem ser expressos estruturalmente, em termos de relações sentenciais, como ilustra a a *Figura 5.7.* Essas relações são rotuladas adequadamente, por meio de "rótulos de relações" (Relation Labels - RLs) e "rótulos de atributos" (Attribute Labels - ALs) que expressam informações adicionais e, em geral, restritivas, sobre as UWs. Há ainda uma ontologia que estrutura o léxico **[Uchida et al 2001]**.

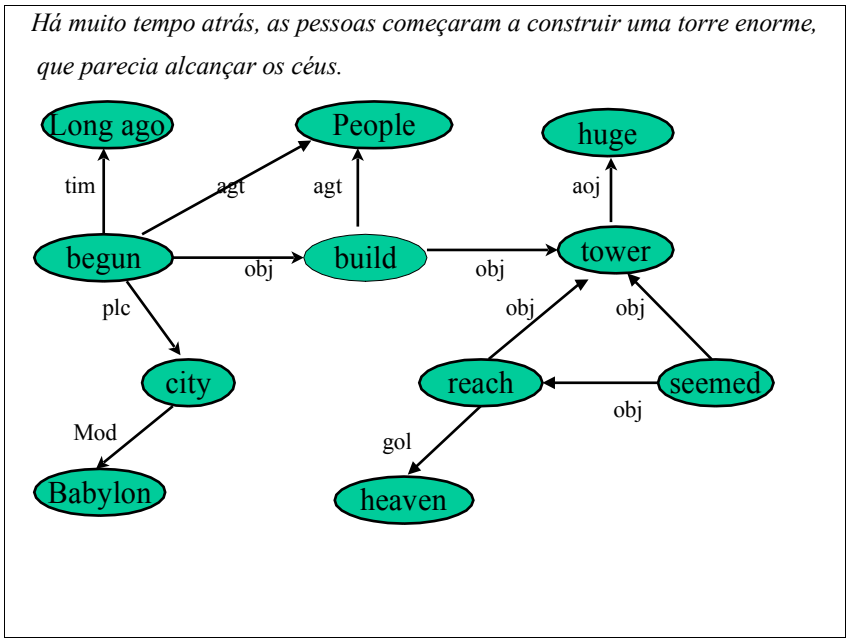


Figura 5.7: Uma sentença em linguagem natural representada em UNL através de uma Rede Semântica [Uchida et al 2001]

A *Figura 5.7* ilustra uma sentença representada em UNL através de uma rede semântica de conceitos e relações. Nesta figura as elipses representam as UWs e os arcos representam rótulos de relações (RLs). Os rótulos de relações (RLs) são utilizados para estabelecer um significado pertinente para a mensagem a partir das universal words (UWs).

As *palavras universais* (UWs) definem o vocabulário da UNL. A função de uma UW é denotar um significado de um conceito específico. Sua representação genérica é um rótulo simples (que indica o significado genérico de uma palavra normalmente em inglês) ou um rótulo limitado por um intervalo específico, que denota significados distintos quando há ambigüidade em relação à palavra original. Por exemplo, "book" permite a representação das seguintes UWS: *book*, *book(icl>publication)*, *book(=account)* e *book(obj>room)*. A primeira UW, *book*, é a representação mais genérica do significado. As demais limitam este significado a conceitos particulares. No caso, a publicações, livro de contabilidade ou reserva de um quarto (em um hotel), respectivamente **[Oliveira et al 2001]**.

As relações entre os conceitos são expressas por meio de *rótulos de relações* (RLs). Tais rótulos servem para expressar relações binárias entre significados, i.e., entre duas UWs distintas. Sua representação geral é dada por um par ordenado do tipo *relation_label(UW1, UW2)*, onde UW1 e UW2 são duas UWs diferentes relacionadas pela relação semântica indicada por *relation_label*. A linguagem UNL define 42 rótulos de relações (RLs) encontrados no *Apêndice B*.

Os *Rótulos de atributos* (ALs) servem para limitar o significado de uma UW genérica, i.e., para particularizar seu significado. Informações adicionais tais como tempo verbal, aspecto, intenção ou estrutura sentencial são exemplos de atributos específicos de uma UW. A representação genérica de um AL é dada pela UW, seguida por tantos atributos quantos forem os sugeridos na sentença de origem. Cada um deles é identificado pelo símbolo inicial "@". A linguagem UNL define 68 rótulos de atributos (ALs), organizados em 7 classes, que podem ser aplicados às UWs como descrito no *Apêndice B* **[UNDL Foundation 2001]**.

A base de conhecimento da UNL, publicada em 29 de janeiro de 2003, possui 5.200 palavras universais, relacionadas hierarquicamente utilizando as 44 relações descritas na especificação da UNL. A base de conhecimento é dividida em 7 categorias que permitem uma melhor contextualização das palavras em uma hierarquia de conceitos **[Hoeschl et al 2002]**.

Lingüistas especialistas estão trabalhando na associação de conceitos de linguagens nativas com as UWs da base de conhecimento UNL através de relações. O esforço envolve ainda especificar atributos gramaticais (tais como morfologia, inflexões verbais semântica, etc) de forma que seja possível traduzir expressões de UNL para um idioma específico e vice versa.

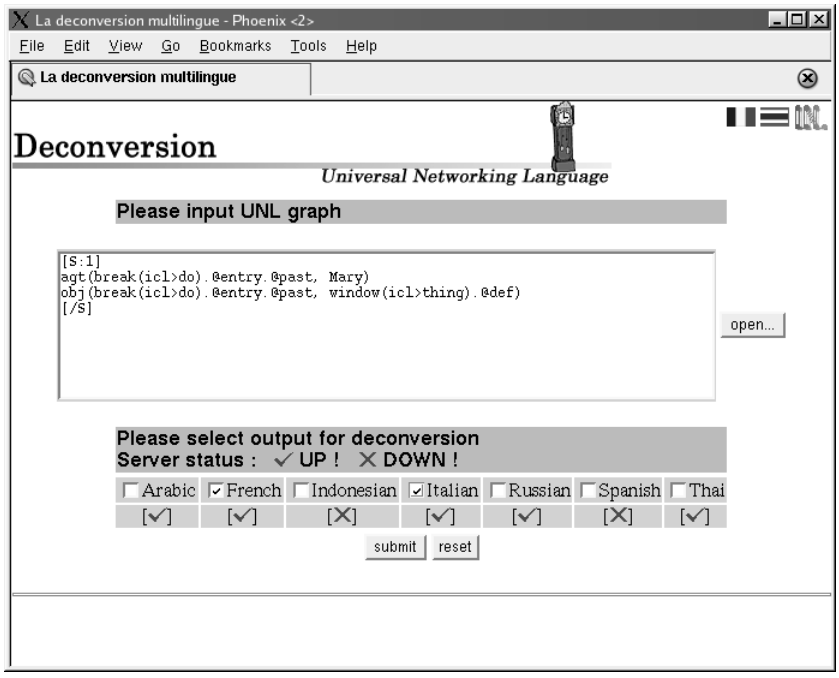


Figura 5.8 - Decodificador UNL - [Tsai 2003]

A *Figura 5.8* ilustra um descoversor UNL que permite transformar grafos UNL em sentenças em linguagem natural **[Tsai 2003]**. Apesar do futuro promissor da linguagem, apenas alguns codificadores e decodificadores UNL já chegaram à fase de teste (Inglês, Italiano, Russo e Espanhol). De acordo com informações da Universidade das Nações Unidas o projeto só será concluído entre 2005 e 2010 e os

codificadores e decodificadores serão disponibilizados à medida que a tecnologia se torne madura **[UNDL Foundation 2002]**.

5.5.4 - Universal Communication Language

A Universal Communication Language (UCL) é uma linguagem desenvolvida por **[Montesco & Moreira 2000]** para representação de conhecimento através de redes semânticas. A linguagem, baseada em relações entre predicados (conceitos), possui estrutura adequada para representar sentenças em linguagem natural. Pode-se considerar, a UCL como sendo uma implementação das especificações da UNL, com a característica especifica de seu formato padrão XML.

Para especificar a representação semântica da mensagem, a UCL utiliza uma ontologia que define formalmente o domínio de conhecimento. A linguagem utiliza a ontologia padrão Thought Treasure **[Mueller 2000],** devido ao fato de que a linguagem UNL não dispõe de uma ontologia explícita e aberta. Os conceitos em UCL são representados através de UWs e têm valores arbitrários dependentes das ontologias utilizadas.

O codificador Inglês-UCL e o decodificador UCL-Inglês utilizam o formato estruturado proveniente da interpretação léxica, sintática e semântica realizada pela ferramenta de processamento de linguagem natural Thought Treasure. A *Figura 5.9* representa a rede semântica gerada pelo codificador Inglês-UCL para a sentença : *'Agent, list students'.*

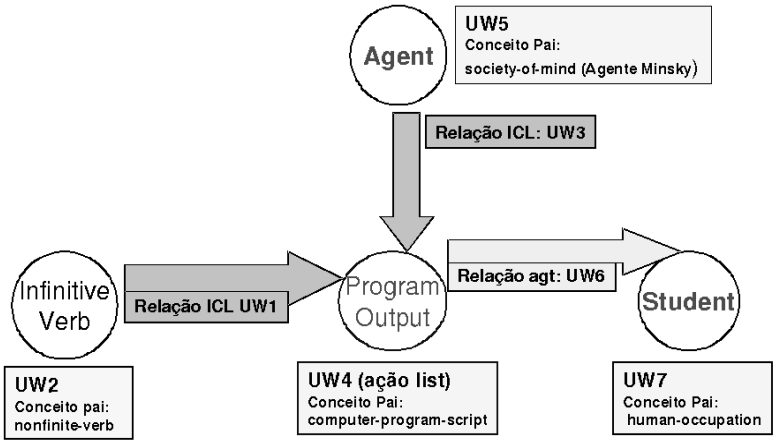


Figura 5.9 - Rede semântica de uma mensagem UCL

Uma restrição existente na transformação Inglês-UCL é a necessidade que os conceitos representados nas frases em Inglês estejam presentes no conjunto de ontologias do software Thought Treasure. Contudo, o

conjunto de ontologias pode ser manualmente expandido conforme a necessidade da aplicação a ser desenvolvida.

A UCL é uma linguagem expressiva cuja compatibilidade com a linguagem UNL permite a integração com futuros sistemas desenvolvidos nesta linguagem. Quando os codificadores e decodificadores UNL para diversos idiomas estiverem disponíveis (inclusive para Português), será possível gerar UCL através de um parser simples que faça a tradução entre a sintaxe das duas linguagens. Codificadores e decodificadores Português-UNL vêm sendo desenvolvidos por grupos de pesquisa no Brasil e no exterior **[Specia 2002]**, **[NILC 2001]**, **[UNDL Foundation 2001]**, trazendo a promessa que, em um futuro breve, será possível criar interfaces inteligentes capazes de aceitar comandos em Português.

5.6 - Considerações Finais

Usuários estão acostumados a interagir com aplicações computacionais por meio de interfaces gráficas que implementam o paradigma de manipulação direta. O atual estágio de desenvolvimento do processamento de linguagem natural (PLN) vem permitindo a criação de interfaces baseadas em linguagem natural capazes de oferecer a comunicação direta do usuário com o software. Tal paradigma se mostra de especial interesse para criação de interfaces para agentes inteligentes, pois permite ao usuário comunicar seus objetivos aos agentes de forma direta e natural.
Exemplos de ferramentas de processamento de linguagem natural que podem ser utilizadas para criação de interfaces inteligentes incluem Nautilus, Thought Treasure e UNL. A UCL, uma implementação livre da linguagem UNL para Inglês, desenvolvida por **[Montesco & Moreira 2000]** no laboratório Intermídia do ICMC-USP contribui com um dos primeiros conjuntos de ferramentas publicamente disponível para representação semântica de sentenças derivadas de linguagem natural. Devido à compatibilidade da interlingua UCL com a linguagem UNL cujos codificadores/decodificadores para diversos idiomas estão em fase de desenvolvimento, acredita-se poder, em um futuro breve, integrar ferramentas desenvolvidas para ambas as linguagens.

Este projeto propõe a utilização da linguagem UCL como forma de comunicação do usuário com os agentes do sistema. Tem-se por objetivo permitir que usuários sejam capazes de manipular conhecimento e executar ações a partir de sentenças expressas em linguagem natural convertidas para UCL e interpretadas pelos agentes do sistema. O capítulo seguinte apresenta detalhes desta proposta.

Capítulo VI - Plataforma SemanticAgent

6.1 - Contexto do Trabalho

O Laboratório Intermídia do Instituto de Ciências Matemáticas e Computação da Universidade de São Paulo vem pesquisando há alguns anos tecnologias que permitam a criação de agentes de software inteligentes. Alguns pesquisas desenvolvidos na área de agentes são:

Servidor Universal - Status: Concluído.
O Servidor Universal provê um ambiente aberto e seguro para a execução de agentes. Suas principais funções são receber os agentes, autenticá-los e prover acesso seguro aos recursos do sistema **[Linhalis e Moreira 2002]**.

Codificador-Decodificador UCL - Status: Concluído
A UCL (Universal Communication Language) é uma implementação das especificações da UNL (Universal Network Language). Os codificadores e decodificadores UCL construídos a partir da ferramenta de processamento de linguagem natural Thought Treasure **[Mueller 2000]**, permitem a representação de linguagem natural de forma estruturada e não-ambígua através de redes semânticas **[Montesco e Moreira 2002]**.

JuspSpace - Status: Concluído
Framework para criação de aplicações distribuídas. O JuspSpace oferece suporte à criação de agentes capazes de se comunicar segundo o modelo de quadro negro. O protótipo desenvolvido implementa algumas das especificações do JavaSpaces **[Sun Microsystems 2003]** definidas pela Sun Microsystems **[Figueiredo 2002]**.

No contexto do trabalho proposto, a plataforma SemanticAgent se caracteriza como uma plataforma para criação de agentes inteligentes, a qual pode fazer uso de alguns dos projetos anteriormente realizados, não apenas tirando proveito das contribuições já obtidas, mas também criando oportunidades para pesquisas futuras. O objetivo principal da plataforma SemanticAgent é prover os elementos necessários à criação de agentes inteligentes capazes de manipular conhecimento e executar ações a partir de requisições feitas em linguagem natural restrita. A plataforma SemanticAgent se propõe a prover, a princípio, apenas a funcionalidade básica para alcançar este objetivo. O projeto inclui uma implementação básica de vários componentes que podem ser aprimorados e aperfeiçoados através de novas pesquisas em áreas tais como

definição formais de padrões de matching de mensagens, técnicas de planejamento, especificação formal de inferência em redes semânticas, integração de ontologias, etc.

6.2- Objetivos do Trabalho

O uso de dispositivos digitais como computadores, handhelds e telefones celulares vem crescendo de forma bastante rápida no decorrer dos últimos anos em todo o mundo. Contudo, a interface do ser humano com este dispositivos continua sendo baseada em elementos gráficos que geram uma interação limitada. **[Mueller 2000]**. Apesar de todos os avanços realizados no campo da Ciência da Computação, a maioria do software oferecido aos usuários não possui inteligência sobre o contexto no qual atua, o que limita o seu escopo de uso. O projeto, que aqui se apresenta, pretende investigar formas de interação ser humano–computador mais simples através do uso de linguagem natural e agentes de software que possuam conhecimento sobre o contexto no qual atuam.

Este trabalho propõe o desenvolvimento de uma plataforma para a criação de agentes inteligentes que possam se comunicar através de linguagem natural com os seus usuários e possuam inteligência sobre o contexto no qual atuam. Almeja-se criar uma plataforma que ofereça serviços básicos para execução de ações e manipulação de conhecimento. A plataforma deve ser genérica o bastante a fim de permitir o desenvolvimento de aplicações para diversos domínios.

Para alcançar este objetivo, propõe-se o uso da linguagem artificial livre de ambigüidade UCL (Universal Communication Language) e a representação de conhecimento do domínio e de comportamentos do domínio através da estrutura de dados Extended Knowledge Network apresentada na seção 4.2.4.3 do *Capítulo IV*.

6.3 - A Plataforma SemanticAgent

O professor Yoav Shoham, da Universidade de Stanford, propõe um novo paradigma para o desenvolvimento de aplicações chamado programação orientada a agentes. Segundo tal paradigma, um sistema pode ser construído através de *agentes* capazes de se comunicar através de troca de mensagens **[Shoham 1993]**. A arquitetura da plataforma SemanticAgent foi concebida segundo o paradigma de programação orientada à agentes proposta por Shoham. Os principais componentes do sistema são implementados através de agentes desenvolvidos utilizando a linguagem Java. Os agentes do sistema se comunicam via troca de mensagens como ilustra a *Figura 6.1*.

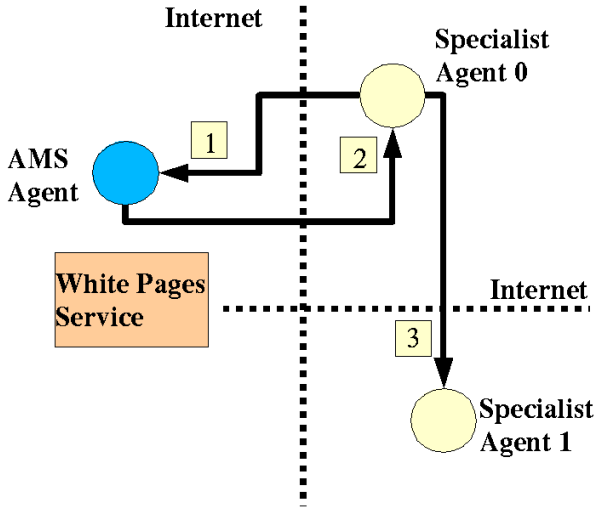


Figura 6.1 – Comunicação entre agentes na plataforma SemanticAgent

De acordo com o modelo de comunicação por troca de mensagens, agentes trocam mensagens com um conteúdo semântico com o propósito de executar alguma tarefa. Para que esta comunicação entre agentes possa ocorrer é necessário definir:

- Um modelo para representar mensagens (FIPA-ACL, KQML. etc)
- Uma linguagem de representação de conhecimento para representar a semântica da mensagem (FIPA-CL, Lisp, Prolog, UCL, etc)
- Uma maneira de codificar as mensagens (ASCII, binário, XML)
- Uma maneira de transportar mensagens (sockets, HTTP, IIOP, etc)

A comunicação entre agentes da plataforma SemanticAgent adota o padrão de mensagens proposto pela FIPA (Foundation For Physical Intelligent Agents) **[FIPA 2001]**. O padrão de comunicação FIPA é amplamente aceito e foi submetido ao comitê padronizador da linguagem Java (Java Community Process) que está homologando a API "*javax.agent*". As especificações propostas pela FIPA possuem diversas implementações entre elas: JAS (Java Agent Services API) **[Greenwood 2000]**, FIPA-OS **[Emorphia 2003]** e JACK **[AOS Group 2003]**.

No protótipo da plataforma SemanticAgent, cada mensagem trocada pelos agentes é composta por quatro elementos encapsulados dentro de um envelope FIPA, como ilustra a *Figura 6.2*

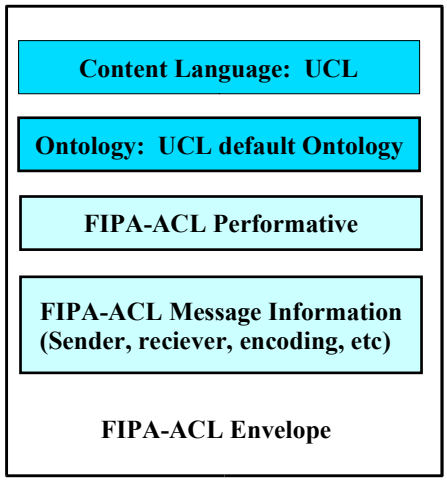


Figura 6.2 - Estrutura das mensagens utilizadas para comunicação entre agentes na plataforma SemanticAgent

Para facilitar a intercomunicação com outros agentes, optou-se por utilizar o padrão de comunicação *FIPA 00061*. A linguagem de conteúdo utilizada nas mensagens é UCL. A codificação de mensagens utiliza o formato binário através da serialização de objetos *ACLMessage*. Embora esta codificação não seja a mais eficiente, ela é simples, portável e não requer a construção de parsers adicionais. As mensagens são enviadas na forma de objetos serializados através de sockets TCP. Não se faz uso de nenhum esquema específico da FIPA para codificação de mensagens (string, binary) nem se faz uso dos serviços de comunicação definidos pela FIPA (HTTP, IIOP).

O payload das mensagens trocadas entre os agentes da arquitetura são compostas por sentenças UCL (que representam a solicitação feita pelo usuário) e por uma performativa FIPA-ACL, que especifica o contexto da comunicação. As performativas FIPA-ACL utilizadas na comunicação entre agentes para consulta, resposta, ordem e validação ajudam a diminuir a ambigüidade de uma sentença expressa em UCL, pois explicitam o seu propósito, como ilustra a *Tabela 6.1*.

Performativa FIPA-ACL	Uso
QUERYREF	Consulta a base de conhecimento
QUERYIF	Inferência sobre a base de conhecimento
INFORM	Novo conhecimento
REQUEST	Execução de ação

Tabela 6.1 - Performativas FIPA-ACL utilizadas na comunicação entre agentes

O envelope FIPA-ACL define os campos de informação que possuem parâmetros utilizados no envio de mensagens. Os endereços utilizados nas mensagens são endereços relativos. Tais endereços são resolvidos por um agente facilitador que possui o endereço físico dos agentes registrados no sistema. Esta estratégia permite que os agentes do sistema sejam distribuídos em várias máquinas conectadas à Internet.

Todos os agentes da plataforma herdam a capacidade de comunicação através de um agente básico que implementa os mecanismos de envio de mensagens fornecido pelo pacote FIPA-OS **[Emorphia 2003]**. O agende básico da plataforma foi desenvolvido em Java na forma de um servidor multi-threaded. Assim, todos os agentes da plataforma são capazes de gerenciar múltiplas requisições simultaneamente, mantendo conexões estáticas ou stateless de acordo com a necessidade. Diagramas UML que descrevem detalhes do *Agente Básico SemanticAgent* e os demais agentes da arquitetura podem ser encontrados no *Apêndice C*.

6.3.1 - Agente Facilitador AMS

Serviços de diretório do tipo páginas amarelas e lista telefônica são normalmente encontrados em arquiteturas de sistemas federados como discutido nas seções *2.4* e *4.2.3*. Tais serviços armazenam as informações sobre a localização dos agentes do sistema. Possibilita-se assim, que novos agentes sejam incorporados de forma simples ao sistema.

Para facilitar a comunicação entre agentes, implementou-se um agente facilitador AMS (*Agent Managment Service*) que armazena o endereço dos agentes que fazem parte do sistema. O agente facilitador AMS permite que os agentes do sistema utilizem nomes ao invés de endereços nas mensagens enviadas. O agente AMS resolve os endereços baseado no diretório de endereços dinâmicos mantido pelo agente. Para manter atualizado o diretório de agentes, todos os agentes do sistema se registram no serviço AMS através de um método padrão implementado no construtor do Agente Básico do qual todos os agentes do sistema derivam. Quando um agente para de operar, ele deve retirar seu registro do diretório AMS, de forma a manter o serviço atualizado.

O agente AMS permite que os agentes do sistema possam ser executados de maneira distribuída em uma rede de computadores. Assim, quando um agente necessita se comunicar com outro agente, ele solicita o endereço deste agente e então passa a se comunicar diretamente com ele. O serviço AMS permite que os agentes rodem distribuídos em diferentes máquinas.

6.4- Arquitetura da Plataforma SemanticAgent

A plataforma SemanticAgent é composta por diferentes tipos de agentes, tais como o agente de interface "*User agent*", o agente de comunicação "*UCL converter agent*" e o agente inteligente "*UCL interpreter agent*" (coordenador dos agentes atômicos manipuladores de conhecimento e componentes). Para fins organizacionais a arquitetura da plataforma SemanticAgent foi dividida em três níveis, de acordo com a

funcionalidade provida por cada um na plataforma: o *nível do usuário,* que é composto por aplicações que permitem que usuários e desenvolvedores interajam com a plataforma, o *nível de comunicação usuário-agente*, que gerencia as requisições realizadas pelos usuários e o *nível de agentes atômicos*, que é responsável pela lógica de funcionamento da plataforma. A *Figura 6.3* ilustra os níveis da plataforma SemanticAgent.

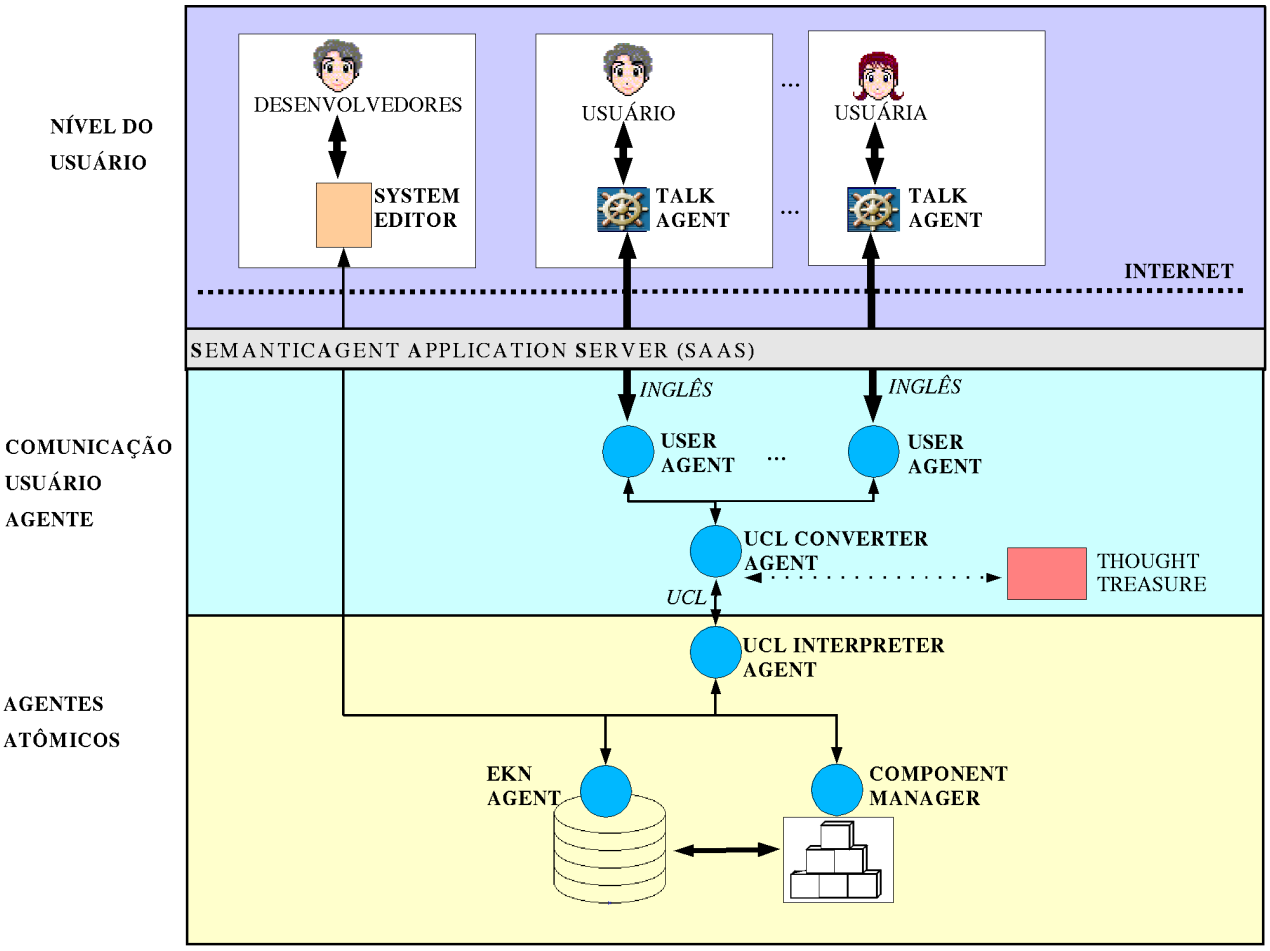


Figura 6.3 – Arquitetura da Plataforma SemanticAgent

O *nível do usuário* da plataforma SemanticAgent é composto por aplicações que permitem que os usuários e desenvolvedores interajam com os agentes da plataforma. Tais aplicativos costumam ser instalados nas máquinas dos usuários. Dois aplicativos foram desenvolvidos para permitir este tipo de interação:

- **TalkAgent** – O TalkAgent é uma aplicação web que pode ser acessada através de qualquer browser HTML que tenham suporte ao protocolo HTTP. Tais browsers estão atualmente disponíveis para diversos dispositivos tais como computadores, tablets, handhelds e telefones celulares 3G (GPRS,GSM). O TalkAgent será discutido com detalhes na *seção 6.4.1.1* deste capítulo.

- **System Editor** – Desenvolvedores da plataforma SemanticAgent podem manipular conhecimento e comportamentos utilizando esta ferramenta. Esta aplicação, desenvolvida em Java, é instalada na máquina do desenvolvedor e permite o acesso aos agentes através da Internet. O System Editor será discutido com detalhes na *seção 6.4.4*.

O nível de *Comunicação Usuário-Agente* é responsável por gerenciar as requisições realizadas pelos usuários, convertendo-as para a linguagem UCL, a qual é interpretada pelos agentes atômicos do sistema. A comunicação usuário agente é também responsável por transmitir o resultado gerado pelos agentes do sistema aos usuários finais. Os agentes responsáveis pela comunicação usuário agente são:

- **User Agent** - Agente responsável pela comunicação do usuário com os demais agentes da plataforma. Recebe requisições feitas em linguagem natural em conjunto com informações sobre o contexto da comunicação. Envia ao usuário o resultado das requisições solicitadas. O *User Agent* é discutido com detalhes na *seção 6.4.2.1*.

- **UCL Converter Agent** - Converte as requisições de linguagem natural para UCL e vice-versa com ajuda de um conversor-desconversor UCL implementado em conjunto com o software Thought Treasure. O UCL Converter Agent será discutido com detalhes na *seção 6.4.2.2*.

O módulo de *Agentes Atômicos* é responsável pela lógica da plataforma, incluindo tarefas como planejamento, manipulação de conhecimento e execução de comportamentos. Os agentes atômicos têm por objetivo satisfazer as requisições dos usuários. Três agentes principais gerenciam os agentes atômicos, são eles:

- **UCL Interpreter Agent** – Agente coordenador do sistema, interpreta as requisições UCL recebidas e realiza planejamentos simples a fim de satisfazer a requisição recebida. Este agente interage com os agentes EKN e CM realizando consultas à base de conhecimento e solicitando a execução de métodos de componentes. Uma vez obtidos os resultados, o agente interpretador os retransmite para o agente do usuário. O UCL Interpreter Agent é discutido com detalhes na seção *6.4.3.1*.

- **Extended Knowledge Network (EKN)** – Agente que manipula a base de conhecimento estendida do sistema. Esta base possui o conhecimento do sistema expresso através de conceitos e asserções, além de informações sobre os componentes do sistema. A EKN é discutida na seção *6.4.3.2*.

- **Component Manager (CM)** - Agente que gerencia os componentes de software existentes no sistema. Ele é responsável por exportar informações e pela execução dos componentes existentes no repositório.

Os componentes são responsáveis por executar ações específicas ao domínio de aplicação. O Component Manager é discutido com detalhes na seção *6.4.3.3* .

Detalhar-se-á nas seções seguintes os módulos que compõem a plataforma SemanticAgent, explicitando-se as características principais de cada módulo e as inter-relações entre os módulos do sistema.

6.4.1- Aplicações do Usuário

Identificou-se dois tipos distintos de usuários para a plataforma: o desenvolvedor de aplicações que utiliza a plataforma SemanticAgent para criar novas aplicações e o usuário de aplicativos (ou simplesmente usuário) que utiliza aplicativos já existentes na plataforma. O diagrama UML presente na *Figura 6.4* ilustra alguns tipos de casos de uso da plataforma SemanticAgent.

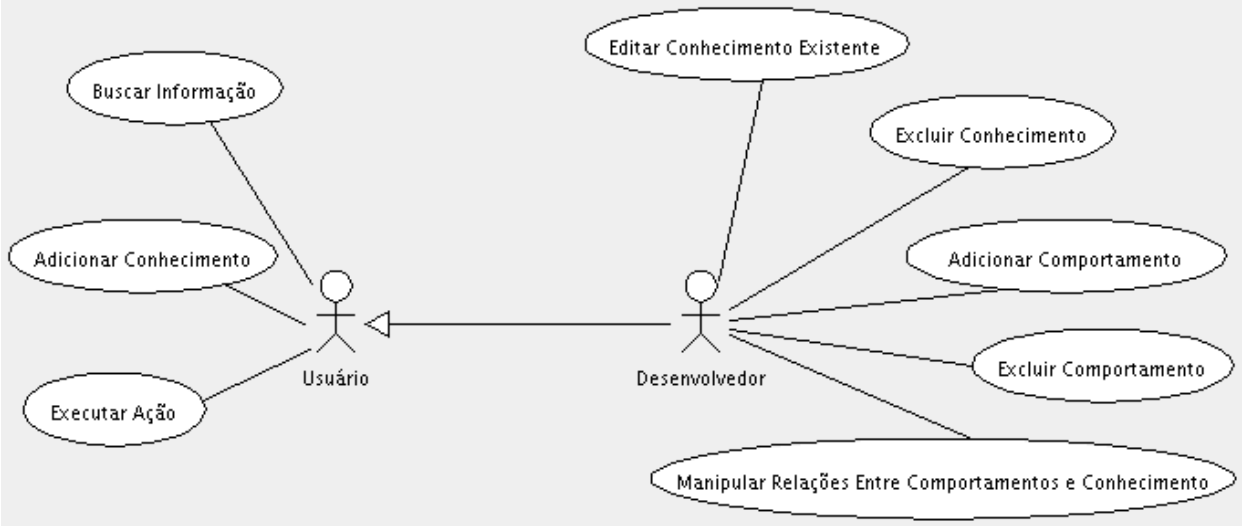


Figura 6.4 – Diagrama de casos de uso para o plataforma SemanticAgent

Como se pode observar a partir do diagrama de casos de uso, usuários possuem necessidades mais simples que os desenvolvedores do sistema. A maior parte da interação do usuário se restringe a consultas e solicitações de serviços.

O desenvolvedor de aplicações, por sua vez, necessita ter maior controle sobre o sistema. Devido a sua necessidade de manipular comportamentos e a base de conhecimento, optou-se por criar uma aplicação cliente-servidor específica, chamada System Editor (seção *6.4.4)*.

6.4.1.1 - TalkAgent

O TalkAgent é uma aplicação desenvolvida de forma que o usuário possa interagir com a plataforma SemanticAgent através do uso de linguagem natural restrita. Apesar das limitações do processamento da linguagem natural, o TalkAgent permite realizar tarefas como consultar a base de conhecimento e delegar algumas tarefas simples aos agentes.

Devido ao fato da linguagem natural ser intrinsecamente imprecisa, muitas vezes uma requisição do usuário apresenta mais de uma acepção. Para resolver tais problemas de ambiguidade, o usuário costuma ter que especificar precisamente a sua requisição, como ilustra a *Figura 6.5*.

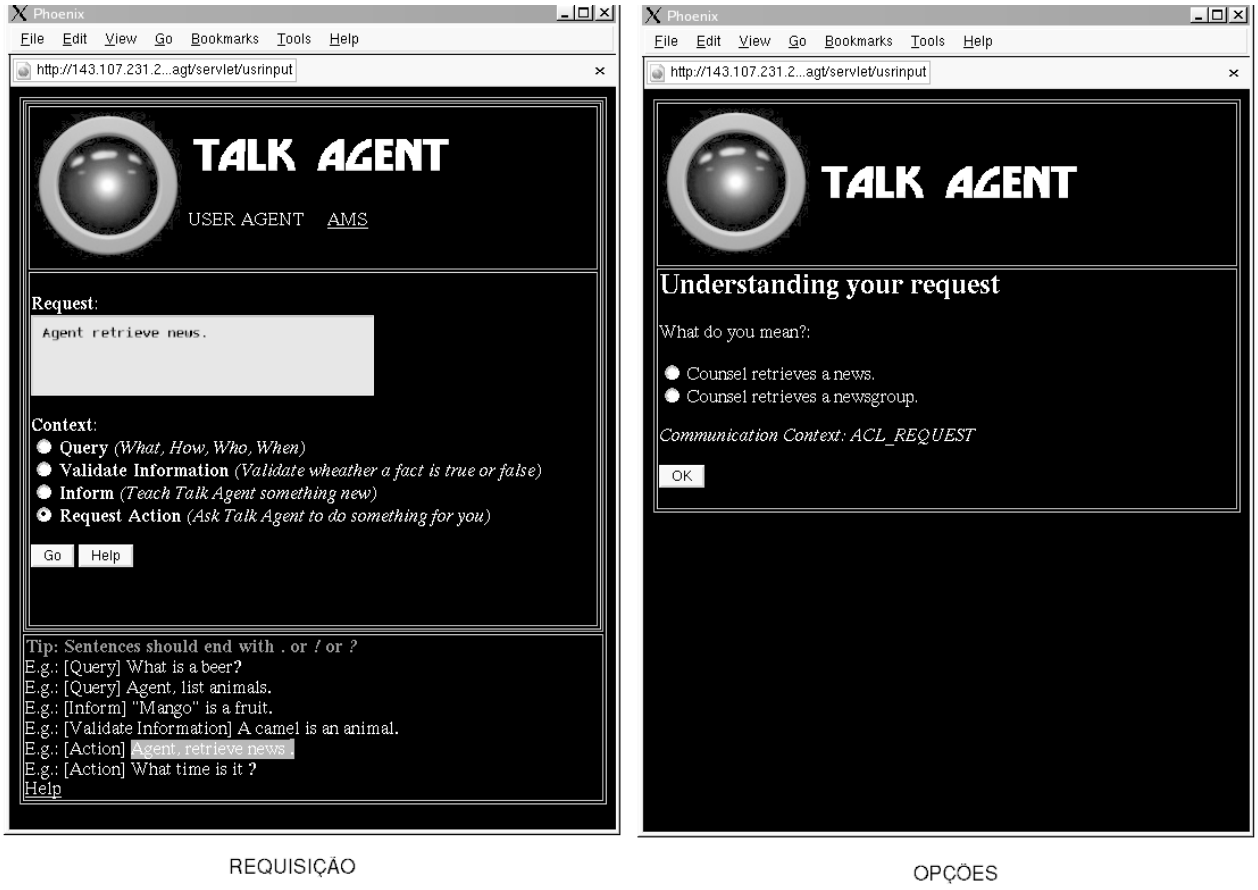


Figura 6.5 - Web Interface: interação do usuário com o sistema através de linguagem natural.

No exemplo ilustrado pela *Figura 6.5* a requisição feita pelo usuário "*Agent retrieve news*" tem duas acepções. A primeira acepção possível é "Retrieve news", i.e obter notícias. A outra acepção é "*Retrieve a newsgroup*" i.e, acessar um serviço de newsgroups, comumente chamado apenas de news. Uma vez que o usuário consiga expressar sua requisição de maneira não ambígua, os agentes da plataforma processam esta requisição como será discutido nas sub-seções a seguir.

6.4.2 - Comunicação Usuário - Agente

O módulo de *comunicação usuário-agente* é responsável por gerenciar a comunicação dos usuários com a plataforma. Este módulo é composto pelo *"User Agent"* responsável pelo gerenciamento da comunicação do sistema e pelo *"UCL Agent"* responsável pela conversão Inglês-UCL. Tais agentes são discutidos nas sub-seções a seguir.

6.4.2.1 - User Agent

O User Agent é responsável por gerenciar a comunicação dos usuários com os demais agentes da plataforma. O User Agent recebe as requisições do usuário e as envia para o UCL Agent que as codificará em UCL. Este agente é implementado através de um servlet executado no servidor de código aberto Jetty escrito na linguagem Java **[Mort Bay Consulting 2003]**. O servlet desenvolvido aceita requisições de clientes HTTP como browsers, por exemplo. Devido ao fato do protocolo HTTP não manter estado entre as interações do usuário com o servlet, um componente roteador é necessário ao gerenciamento de sessões como ilustra a *Figura 6.6.*

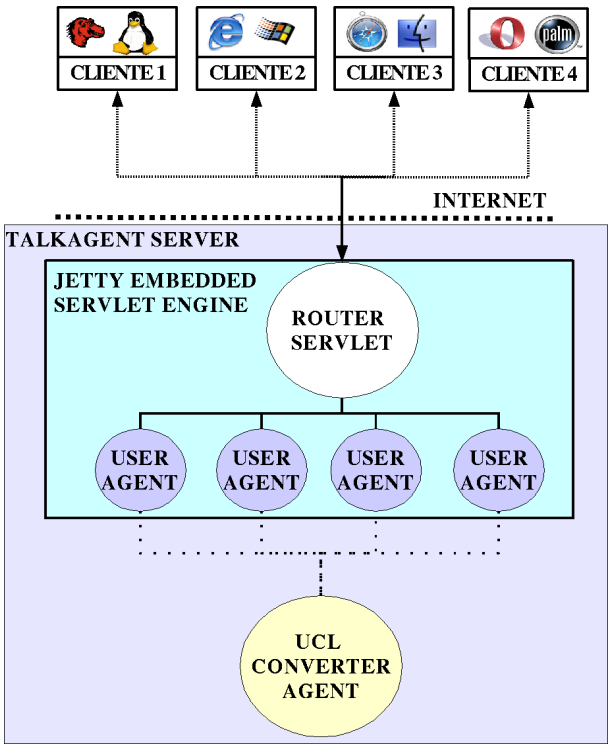


Figura 6.6 – User Agent

Toda vez que um cliente inicia uma requisição é criado um identificador de comunicação que é utilizado em todas as comunicações posteriores do cliente com o sistema, até que este tenha sua requisição satisfeita. O *Router Servlet* utiliza os identificadores de comunicação para rotear as mensagens dos usuários do sistema para os agentes "User Agent" que são instanciados para cada usuário que se comunica com a plataforma.

6.4.2.2 - UCL Converter Agent

O UCL Converter Agent é responsável por converter as requisições feitas pelo usuário em linguagem natural (Inglês) para a linguagem artificial livre de ambigüidades UCL. Este agente é baseado no codificador/decodificador UCL desenvolvido no trabalho de mestrado de **[Montesco 2001]**. O processo de codificação de uma sentença de inglês para UCL é realizado com o auxílio da ferramenta de processamento de linguagem natural Thought Treasure como ilustra a Figura 6.7:

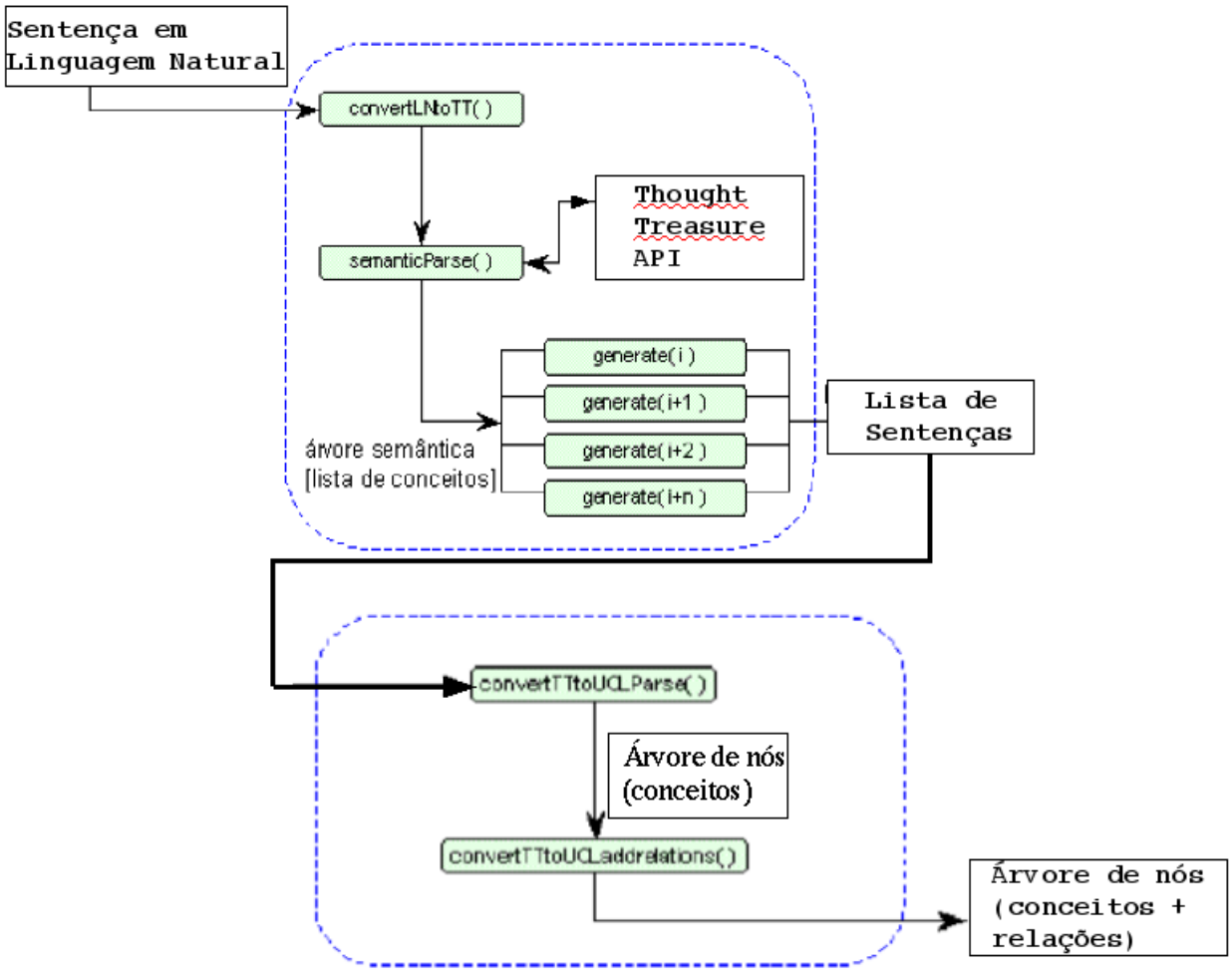


Figura 6.7 – Processo de Conversão Inglês-UCL [Montesco e Moreira 2002]

O método *ConvertNLToTT* é chamado para realizar a interpretação de uma sentença em linguagem natural, e retorna uma lista de sentenças que expressam a mensagem inicial. O resultado do método é uma lista de alternativas, com as diversas representações semânticas para a sentença submetida. Para poder avaliar sintaticamente e semanticamente a sentença, que está em linguagem natural, são criadas internamente duas estruturas. A primeira refere-se a uma árvore de nós representando a avaliação sintática e a segunda é outra árvore de nós representando a avaliação semântica. Cabe ao usuário escolher qual das interpretações é mais apropriada.

Uma vez escolhida a sentença que se deseja converter, o método *convertTTtoUCLRelations*, gera uma estrutura que representa uma rede semântica UCL composta por conceitos e relações. Na primeira etapa são gerados os conceitos na forma de UWs. Além disso, é inserido um identificador único para cada conceito. Como é mostrado na *Figura 6.7*, a entrada inicial é a lista de conceitos, a partir dela se verifica a existência de cada conceito na ontologia e se obtém o conceito pai de cada um (desta forma se descobre a que hierarquia o conceito pertence). Na segunda etapa, as relações existentes entre os conceitos são determinadas e elas são explicitadas através de uma nova relação. Este processo dá forma e estrutura à rede semântica que representa a sentença em linguagem natural.

Uma vez que uma requisição do usuário é convertida para UCL ela é encaminhada para o UCL Interpreter que coordena os agentes atômicos a fim de satisfazer à requisição do usuário, como discutido com detalhes na seção seguinte.

6.4.3 - Agentes Atômicos

Apresentou-se na seção 4.2.4.2 do *Capítulo IV* o modelo de "agentes atômicos". O gerenciamento de conhecimento e comportamentos proposto para plataforma SemanticAgent é uma implementação inicial das idéias propostas por este novo paradigma de desenvolvimento de sistemas multi-agentes. O módulo da plataforma SemanticAgent responsável pelos agentes atômicos é composto por três agentes facilitadores: UCL Interpreter Agent, o EKN Manager Agent e o Component Manager Agent. Tais agentes são descritos nas subseções a seguir.

6.4.3.1 - UCL Interpreter Agent

A UCL é uma interlíngua que representa sentenças em Inglês de maneira não ambígua. O objetivo do UCL Interpreter Agent, é processar requisições expressas através de mensagens UCL a fim de satisfazer as requisições dos usuários. Os requisitos funcionais do UCL Interpreter Agent incluem:

- O gerenciamento de conhecimento do agente através de consultas e inserção de novas informações na base de conhecimento do agente.
- O acionamento da execução de ações genéricas programadas sob a forma de componentes.
- A análise de condições, restrições e completude de informações necessárias à execução de ações.

O processo de análise de uma mensagem UCL se inicia com a decomposição da mensagem UCL em uma lista de conceitos e relações. Uma vez separados os conceitos e relações da mensagem UCL, ela é analisada a procura de algum padrão reconhecido pelo sistema. Se um padrão for reconhecido, um script apropriado para a requisição *(Tabela 6.2)* é executado e os resultados são enviados ao User Agent, como ilustra a *Figura 6.8*. Caso a mensagem UCL possua alguma estrutura não reconhecida, o User Agent é informado que a requisição não pode ser satisfeita. Novos scripts podem ser adicionados pelos desenvolvedores à plataforma para satisfazer requisições específicas. Para tal, pode-se estender as classes *UCLDoc* e *Script* cujos diagramas UML se encontram no *Apêndice C*.

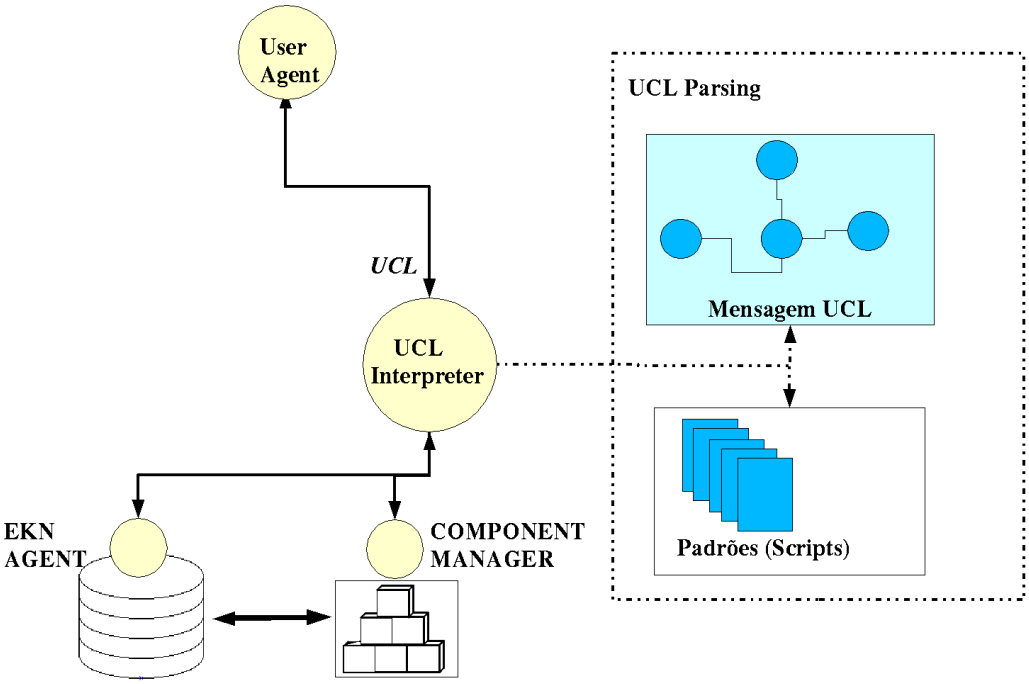


Figura 6.8 – UCL Parsing

As performativas FIPA presentes nas mensagens UCL (*Tabela 6.1*) auxiliam a determinar que tipos de padrões devem ser reconhecidos nas mensagem UCL. Existem dois tipos básicos de scripts para manipulação de mensagens: os scripts lógicos e os scripts de ações. Os scripts lógicos realizam operações sobre a base de conhecimento, enquanto os scripts de ação analisam informações sobre comportamentos, executando-os quando apropriado. Um conjunto inicial de scripts foi criado para satisfazer alguns tipos de requisições freqüentes. Tais scripts foram divididos em categorias de uso como ilustra a Tabela 6.2

Tipo de Script	**Categorias de Script**	**Script UCL**	**Exemplo de uso**
Scripts Lógicos	**QUERY KB** - Consultas à Base de Conhecimento	*Query Definition* - Retorna a definição de um conceito.	* What is ObjectX? * What is an apple?
		Query Person Information - Retorna informação sobre pessoas.	* Who is "PersonX"? * Who is "Percival Lucena"?
		Query Elements - Retorna elementos que obedecem certos critérios	* Agent, list CategoryX * Agent, list planets. * Agent, show *rich people.*
	VALIDATE - Valida asserções	*Simple Validation* - Valida se um fato é verdadeiro ou falso baseado nos conceitos existentes na base de conhecimento.	* Banana is a fruit. (Yes) * Lake is a planet. (No)
		Complex validation - Valida se um fato é verdadeiro ou falso baseado nas regras da base de conhecimento.	* Is an apple red? * Does birds flies? * Are oranges good for the health? * How much one lives?
	INFORM - Insere nova informação na Base de Conhecimento	*Insert Concept* - Insere novo conceito na base de conhecimento.	* Insert fruit avocado. * Insert student “Paulo Cunha”
		Insert Assertion - Insere nova relação na base de conhecimento.	* Grades are greater than zero.
Scripts para ações	**REQUEST ACTION** - Executa uma Ação	*Execute* - Executa um componente associado à base de conhecimento.	* Agents, get last mail from Professor Dilvan.

Tabela 6.2 – Categorias de Scripts UCL

Os scripts desenvolvidos fazem acesso à EKN para manipular conhecimento ou para descobrir se existe um componente no sistema que pode executar uma ação que satisfaça a requisição do usuário. O EKN Agent implementa uma série de serviços que permitem ao UCL Interpreter Agent satisfazer alguns tipos de consultas como os descritos na *Tabela 6.2*. Caso o UCL Interpreter Agent necessite executar uma ação (ao invés de uma consulta) para satisfazer uma requisição, um componente Java deve ser invocado através do Component Manager, que executará métodos específicos gerenciando os detalhes da instanciação e execução do componente.

Nas seções seguintes analisar-se-á os agentes facilitadores EKN Manager Agent e Component Manager Agent descrevendo sua forma de interação com o UCL Interpreter Agent a fim de satisfazer as requisições do usuário.

6.4.3.2 - EKN Manager Agent

A base de conhecimento da plataforma SemanticAgent é uma implementação experimental da estrutura de dados Extended Knowledge Network (EKN) descrita na *seção 4.2.4.3* do *Capítulo IV*. O EKN Manager Agent é responsável por gerenciar a base de conhecimento da plataforma. O conhecimento do sistema é representado por conceitos, relações e informações sobre componentes. Os requisitos funcionais deste agente incluem:

- Importar e exportar ontologias.
- Manipular conceitos
- Manipular asserções
- Manipular informações sobre componentes.

No primeiro protótipo do sistema implementado é possível importar bases de conhecimentos expressas no formato TTKB (Thought Treasure Knowledge Base). Contudo, a arquitetura é abstrata e poderá ser estendida para importar bases de conhecimento em outros formatos como DAML ou CycL, por exemplo. A capacidade de importar ontologias Thought Treasure justifica-se devido a grande quantidade de informação já existente neste formato e a compatibilidade dos conceitos expressos nesta ontologia com as UWs UCL geradas pelo codificador/decodificador UCL .

As ações dos agentes atômicos são implementadas através de componentes. Informações sobre os componentes são integradas a base de conhecimento, permitindo que ações possam ser executadas a partir de requisições solicitadas pelos usuários do sistema. Para representar um componente de software, utiliza-se uma estrutura de dados chamada *BeanAdapter*. Esta estrutura representa os elementos utilizáveis de um componente tal como o nome do componente, métodos, argumentos dos métodos, propriedades e eventos. O diagrama UML da classe BeanAdapter pode ser encontrado no *Apêndice C*.

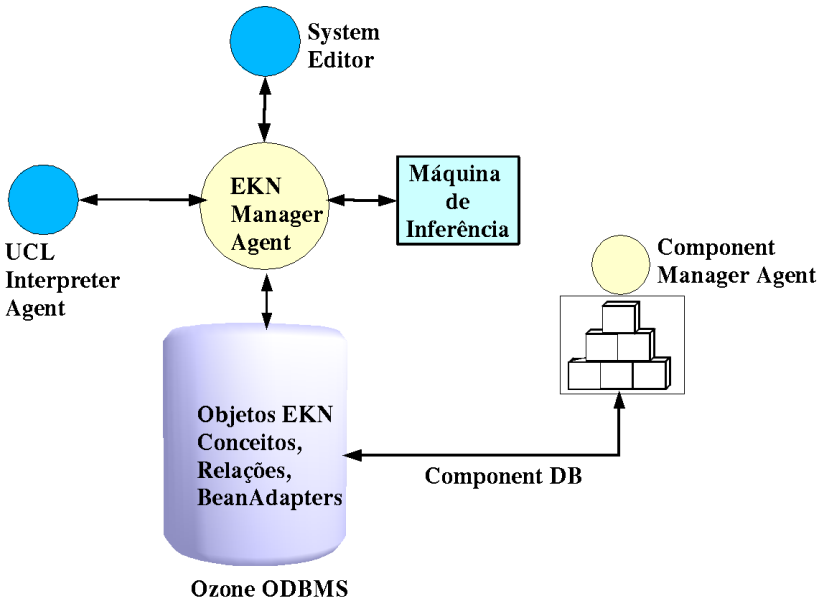


Figura 6.9 – EKN Manager Agent

A *Figura 6.9* ilustra a arquitetura do EKN Manager Agent. A máquina de inferência associada à base de conhecimento tem capacidade de realizar operações limitadas baseada em lógica hierárquica. Segundo esta abordagem, as operações básicas definidas analisam as relações parte-todo e relações binárias arbitrárias entre conceitos. O mecanismo implementado é capaz de validar relações entre asserções baseado em um mecanismo simples de forward chaining.

Os conceitos, relações e BeanAdapters são representados como objetos e armazenados na base de dados orientada a Objetos *Ozone*, cujo código fonte é aberto e a implementação foi feita em Java **[Braeutigam 2003]**. A opção pelo uso de uma base de dados orientada a objetos simplificou o desenvolvimento do EKN Manager Agent, pois a base de dados pôde ser integrada de maneira transparente com a aplicação, sem a necessidade de codificação e decodificação dos objetos em tabelas relacionais. Obteve-se também a vantagem da portabilidade da solução que pode ser executada diretamente (sem nenhuma configuração extra) em qualquer sistema operacional que suporte Java.

6.4.3.3 - Component Manager Agent

Segundo o modelo de agentes atômicos, os comportamentos dos agentes são implementados através de componentes que representam as ações que os agentes podem executar. O *Component Manager Agent é* responsável por gerenciar os componentes de software armazenados no sistema. Os requisitos funcionais deste agente incluem:

- Exportar automaticamente o nome e as interfaces (métodos, propriedades e eventos) dos componentes armazenados no repositório de componentes.
- Instanciar e executar um ou mais métodos de um dado componente.
- Gerenciar eventos de componentes, acionando métodos específicos quando necessário.

Na implementação do protótipo da plataforma SemanticAgent, o único modelo de componentes suportado é o modelo de componentes padrão do Java, o modelo JavaBeans. A simplicidade deste modelo permite que qualquer software desenvolvido em Java possa ser facilmente convertido em um componente JavaBean. Caso se deseje utilizar componentes desenvolvidos em outras linguagens o desenvolvedor pode criar um wrapper para o componente desejado utilizando a API JNI (Java Native Interface) oferecida pela linguagem Java.

Embora a implementação atual do Component Manager Agent suporte apenas componentes do tipo JavaBeans, a arquitetura de gerenciamento de componentes é abstrata e poderá ser estendida para suportar outros formatos de componentes como Enterprise Java Beans (EJBs) ou objetos CORBA. A *Figura 6.10* ilustra a arquitetura do Component Manager Agent.

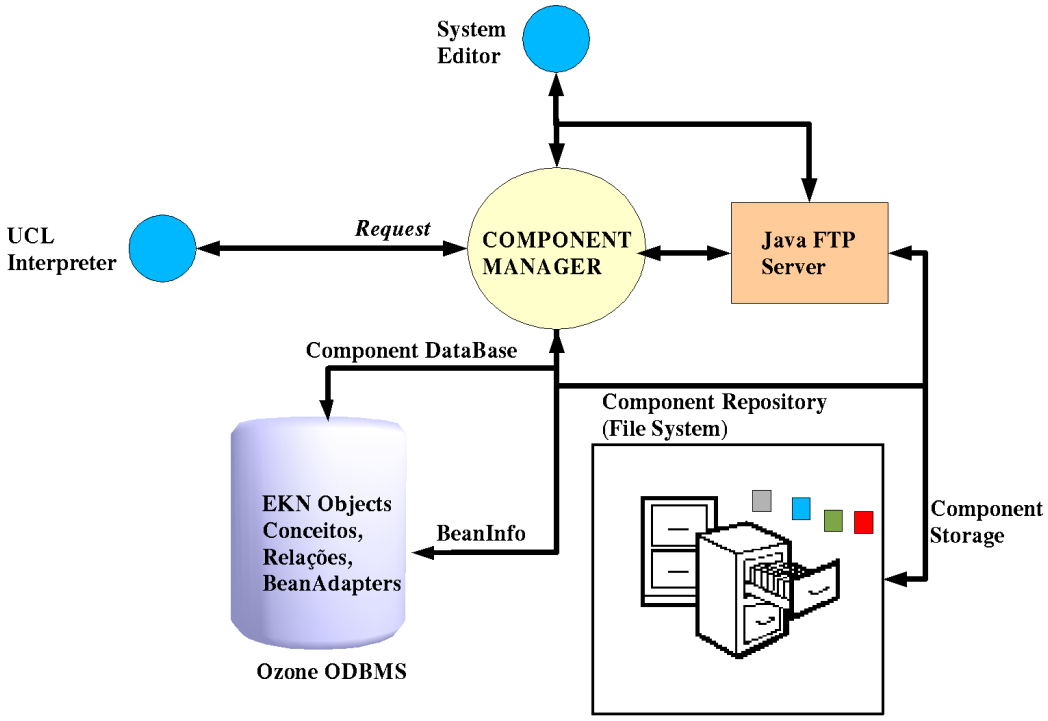


Figura 6.10 – Component Manager Agent

A implementação do Component Manager Agent é baseada no pacote JavaBean Box, uma implementação da API JavaBeans desenvolvida pela Sun Microsystems **[Sun MicroSystems 2001]**. Tal pacote permite a introspecção automática, instanciação e execução de métodos de componentes JavaBeans.

Os componentes são armazenados em sistema de arquivos local, mas as informações sobre a sua localização e estrutura são armazenados no ODBMS Ozone através do objeto *CRDB* (Component Repository DataBase), cujo diagrama UML é representado no *Apêndice C*. Um servidor de FTP de código aberto desenvolvido em Java por **[Bhattacharyya 2002]** é integrado ao repositório de componentes, permitindo que aplicações remotas transfiram novos componentes para o sistema através da Internet.

O Component Manager Agent executa métodos de componentes cujos parâmetros são fornecidos pelo UCL Interpreter Agent. A *Figura 6.11* ilustra o caso de um usuário solicitando notícias ao sistema. Uma vez que a requisição é compreendida, o UCL Interpreter Agent solicita ao Component Manager Agent que execute o método *getNews(newstype)* do componente NewBean que acessa serviços de notícias disponíveis na Internet e retorna os resultados ao UCL Interpreter Agent. Este agente, por sua vez, informa o User Agent dos resultados obtidos.

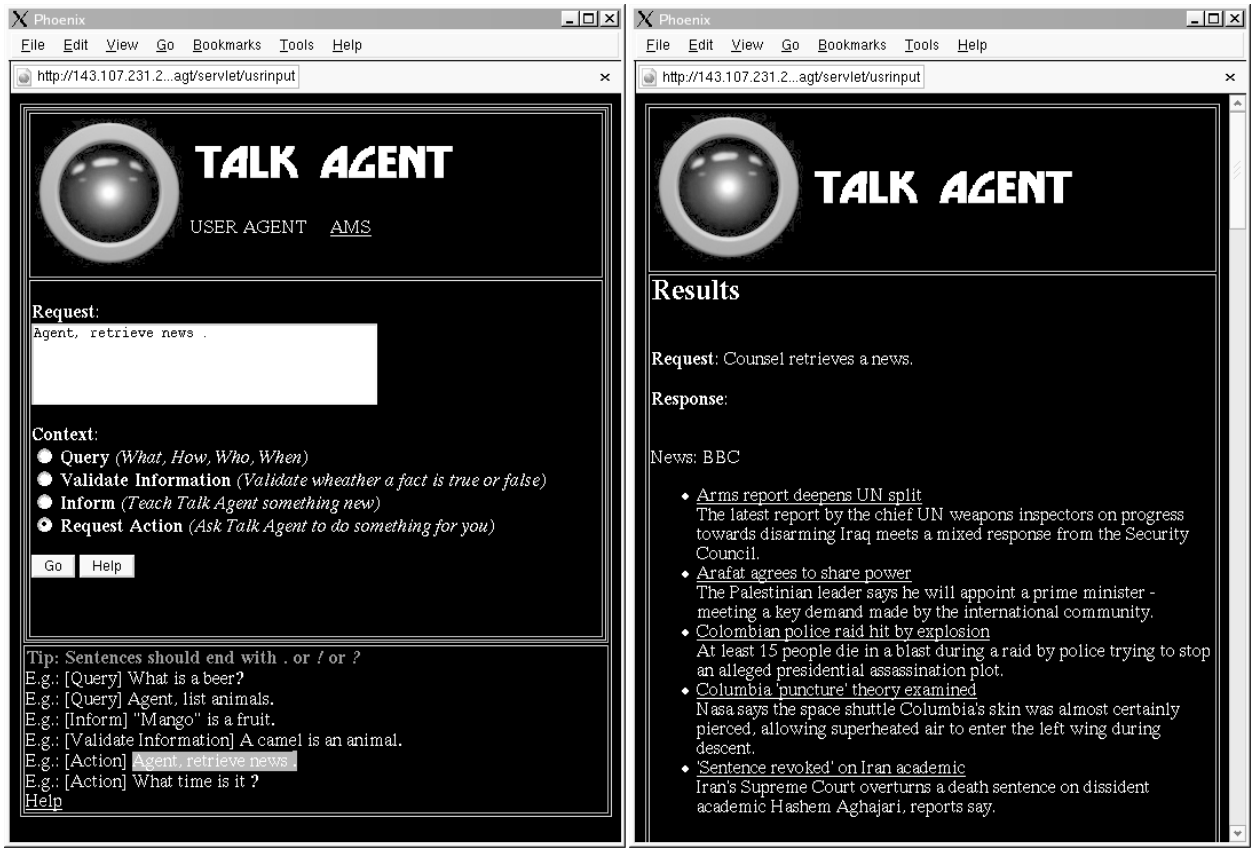


Figura 6.11 – Requisição do usuário satisfeita através da execução de um JavaBean

6.4.4 - System Editor

O *System Editor* é um IDE (Integrated Development Environment) que auxilia o desenvolvimento de aplicações baseadas na plataforma SemanticAgent. Este IDE permite manipular diretamente a base de conhecimento (EKN) e o repositório de componentes do SemanticAgent Server descrito nas seções anteriores deste capítulo.

O System Editor é uma aplicação cliente-servidor escrita em Java, o que permite que desenvolvedores de aplicações baseadas na plataforma SemanticAgent utilizem a Internet como forma de conexão com o SemanticAgent Application Server. Através do System Editor pode-se editar, inserir e remover conceitos, relações e informações sobre JavaBeans. Pode-se ainda adicionar novos componentes ao Component Repository. A *Figura 6.12* ilustra a interface para manipulação da base de conhecimento do SemanticAgent Server.

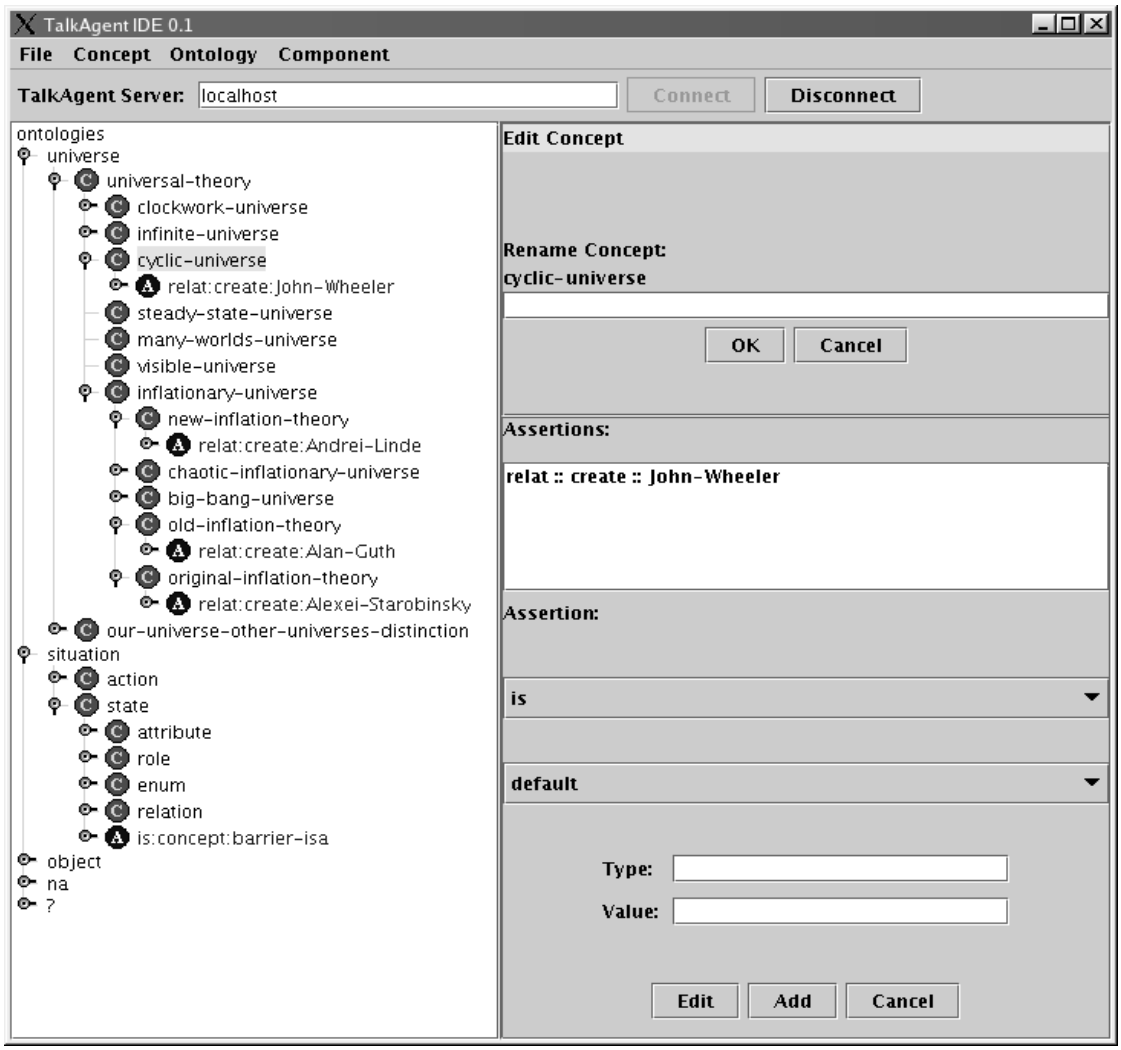


Figura 6.12 – System Editor – Edição da base de conhecimento

Conforme ilustra a *Figura 6.12*, o painel esquerdo do System Editor representa a rede semântica (base de conhecimento do sistema) através de uma árvore composta por conceitos (círculos C) e relações entre conceitos (círculos A). O painel do lado direito permite editar, adicionar e excluir relações e conceitos. O System Editor permite também gerenciar os componentes da plataforma e a suas conexões com a rede semântica. A *Figura 6.13* ilustra o uso do System Editor para manipulação de componentes:

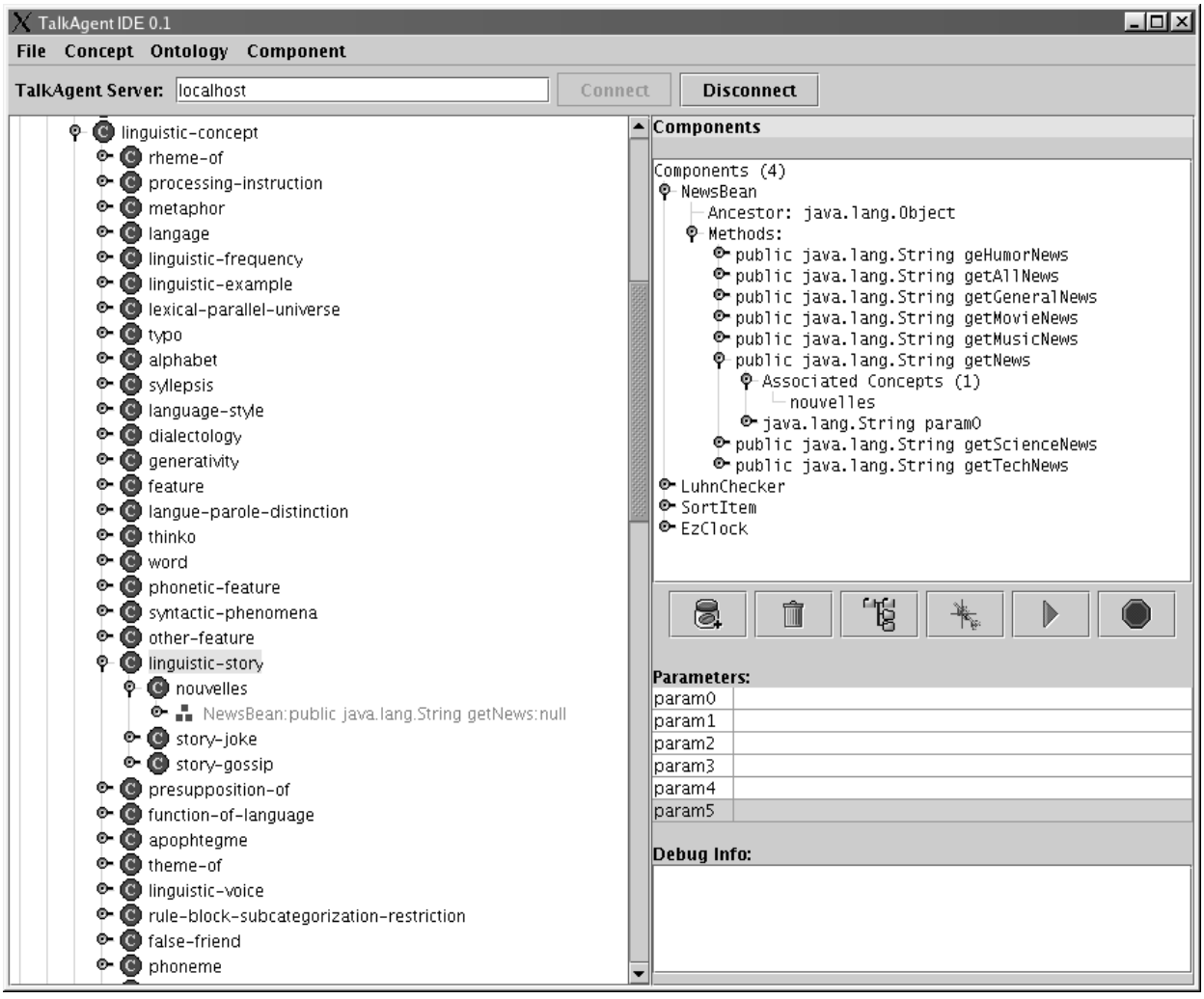


Figura 6.13 – System Editor – Manipulação de Componentes

Conforme ilustra a *Figura 6.13,* o painel direito do System Editor exibe informações detalhadas sobre o componentes existentes no sistema. O System Editor permite a manipulação de métodos, parâmetros e associações com a base de conhecimento. As associações entre métodos ou parâmetros são representados através de pirâmides ligadas à rede semântica na árvore localizada ao lado esquerdo do System Editor. Pode se observar a partir da *Figura 6.13* que o método *getNews(newsType)* do componente *NewsBean* está associado ao conceito *nouvelles* (notícias jornalísticas).

O System Editor permite ainda carregar novos componentes ou excluí-los do sistema e executar métodos específicos a fim de depurar o comportamento do sistema. A transferência de novos componentes do System Editor para o SemanticAgent Application Server se faz através do protocolo FTP, implementado através do cliente FTP de código aberto "GNU FTP Client" **[Blackshaw 2002]**.

6.5- **Considerações Finais**

A plataforma SemanticAgent foi desenvolvida como um protótipo para o modelo de agentes atômicos. Este protótipo permite manipular conhecimento e comportamentos através de redes semânticas estendidas. Os comportamentos do sistema representam ações, que os agentes atômicos podem executar, e são implementados no sistema sob a forma de componentes JavaBeans.

Graças à definição da linguagem UCL e a implementação de um codificador-decodificador, por **[Montesco & Moreira 2000]**, tornou-se possível representar requisições feitas em linguagem natural limitada de forma não ambígua. A plataforma SemanticAgent utiliza a UCL como linguagem de conteúdo na comunicação entre agentes do sistema. O protótipo do UCL Interpreter Agent, desenvolvido para a plataforma, permite processar alguns padrões de mensagens expressas em UCL realizando manipulação de conhecimento e comportamentos a fim de satisfazer requisições de outros agentes ou usuários.

Apresentar-se-á no capítulo seguinte uma análise dos resultados e contribuições trazidos pela implementação da plataforma, apresentando as conclusões sobre o trabalho desenvolvido.

Capítulo VII - Conclusões

7.1 – Visão Geral

A plataforma SemanticAgent, proposta neste trabalho, fornece uma arquitetura abstrata e ferramentas para a criação de agentes inteligentes. Desenvolvido em Java, o protótipo implementado para a plataforma permite criar aplicações capazes de compreender linguagem natural restrita, manipular conhecimento e executar ações. A plataforma criada está madura o suficiente para o desenvolvimento de aplicações simples e pode ser utilizada em outras áreas de pesquisa do grupo. Este capítulo apresenta alguns resultados e conclusões do desenvolvimento do trabalho proposto bem como sugestões para melhorias e futuras pesquisas.

7.2 – Contribuições

A plataforma SemanticAgent contribuiu com uma implementação de um protótipo para o modelo de agentes atômicos, oferecendo ferramentas para o desenvolvimento de agentes inteligentes. Discutir-se-á a seguir algumas contribuições específicas do trabalho realizado.

7.2.1 - Um novo modelo para criação de agentes

Uma das contribuições trazidas por este trabalho é um novo modelo para a representação de comportamentos de agentes inteligentes chamado *agentes atômicos*. Este modelo híbrido alia características de agentes reativos e deliberativos a fim de criar uma rede de agentes reconfiguráveis semelhante ao modelo de agentes e agências proposto pelo professor Marvin Minsky **[Minsky 1988]**. O modelo de agentes atômicos é composto por uma série de componentes cujas funcionalidades contribuem para o comportamento geral de um agente. Os átomos, que também representam o conhecimento do sistema, são ligados entre si através de uma rede semântica. A nova arquitetura proposta se mostra bastante flexível para o gerenciamento de conhecimento e comportamento do sistema, simplificando o processo de coordenação entre agentes existente na maioria dos sistemas multi-agentes.

7.2.2 - Contribuições na área de representação de comportamentos de agentes

A arquitetura de agentes atômicos, propõe a expansão de redes semânticas para a manipulação de conhecimento e integração da descrição de comportamentos (ações) de agentes. Através do modelo proposto e da estrutura de dados *Extended Knowledge Network* pode-se representar o conhecimento dos agentes e a descrição das ações as quais os agentes podem executar. O modelo proposto permite processar as requisições de outros usuários e agentes representadas através de redes semânticas através de técnicas simples de processamento de padrões.

7.2.3 - Contribuições para a criação de sistemas baseados em agentes

Agentes de software costumam ser criados para domínios específicos. Uma dificuldade encontrada para o desenvolvimento de aplicações complexas consiste em compor, alterar e estender um conjunto de agentes de maneira simples e consistente. A arquitetura de agentes atômicos, proposta neste trabalho permite criar novos agentes através de componentes OTS (off-the-shelf) disponíveis para diversos domínios. Tal abordagem, permite a fácil integração de comportamentos possibilitando a criação de novas aplicações a partir de outras existentes através do mínimo esforço necessário.

7.2.4 - Contribuições no campo de HCI

O modelo de manipulação direta apresenta limitações de usabilidade em domínios não estruturados como as descritas por **[Maes 1995]**. O uso de linguagem natural mesmo que restrita aliado a conhecimento de senso comum permite criar as chamadas interfaces inteligentes, capazes de permitir a comunicação usuário computador de maneira mais humana e natural. Tais interfaces podem ser sensíveis ao contexto da aplicação facilitando a manipulação em domínios dinâmicos ou não estruturados como os casos da interface cartográfica descrita por **[Wauchope 1999],** da interface de consulta a bases de dados *Precise* descrita por **[Popescu et al 2003]** ou do ambicioso Knowledge Navigator proposto pela Apple **[Apple 1992]**.

A interface usuário-agente implementada na plataforma SemanticAgent possui conhecimento de senso comum através da base de conhecimento do sistema, permitindo interações simples utilizando linguagem natural restrita, uma sentença por vez. Embora as restrições da representação e manipulação de linguagem natural limitem os tipos de interações do usuário com a plataforma, as características implementadas podem oferecer grande utilidade em domínios como recuperação de informação (Informtion Retrieval).

7.3 – Trabalhos Futuros

Embora a plataforma SemanticAgent implemente vários mecanismos que facilitam a criação de agentes inteligentes, identificou-se na análise do protótipo desenvolvido vários pontos que poderiam melhorar os resultados da plataforma. As limitações existentes podem dar origem a trabalhos futuros como os discutidos nas sub-seções a seguir.

7.3.1 - Limitações do processamento de linguagem natural

Restrições do codificador-decodificador UCL limitam os tipos de sentenças que podem ser representadas pelos agentes da plataforma SemanticAgent. As limitações quanto aos tipos de sentenças que podem ser expressas em UCL poderiam ser minimizadas através do uso de codificadores/ decodificadores UNL, supostamente mais expressivos que o codificador Inglês-UCL desenvolvido. O processo de integração dos codificadores/ decodificadores UNL poderia ser feito através de um tradutor UNL-UCL cuja construção seria facilitada devido a compatibilidade entre as duas linguagens.

7.3.2 - Limitações do processamento de UCL

Restrições no processamento de mensagens UCL via scripts limitam os tipos de sentenças que podem ser processadas pelos agentes da plataforma SemanticAgent. As limitações de processamento poderiam ser amenizadas através de uma especificação formal em expressões regulares das classes de mensagens UCL freqüentemente utilizadas, o que melhoraria o processo de pattern matching de mensagens UCL.

7.3.3 - Limitações da base de conhecimento global

A plataforma SemanticAgent implementa uma base de conhecimento global compartilhada por todos os agentes atômicos da plataforma. Embora esta solução facilite o processo de manipulação de informação pelos agentes, esta solução trás, por outro lado, algumas limitações e complexidades ao desenvolvimento de aplicações baseadas na plataforma. Um problema típico é o armazenamento de perfis. Cabe ao desenvolvedor da aplicação criar componentes que gerenciem os perfis para aplicações multi usuários. Tem-se por exemplo que *Joe User* gosta de noticias de esporte e *Mary User* gosta de noticias de cinema. Cabe ao desenvolvedor armazenar estas informações pessoais através da API de acesso à base de conhecimento do sistema. Tal processo poderia ser simplificado caso a plataforma oferecesse uma interface padrão para o gerenciamento de perfis. Uma solução possível para este problema poderia ser a extensão do Servidor

Universal, desenvolvido por **[Linhalis e Moreira 2002],** para o gerenciamento das informações dos agentes atômicos do sistema.

7.3.4 - Limitações da Máquina de Inferência

O uso limitado de lógica monolítica implementado na máquina de inferência associada ao agente EKN restringe o tipos de inferências que o agente pode realizar sobre o conhecimento do sistema. O mecanismo de forward chaining implementado não prevê, condições de loop entre as relações implementadas na EKN e falta um mecanismo de backward chaining para a máquina de inferência. O uso de lógica de primeira ordem nas relações da rede semântica como nos grafos existenciais propostos pelo matemático Charles Sanders Peirce **[Sowa 2001],** poderia ampliar a capacidade de processamento de conhecimento da plataforma.

7.3.5 - Problemas na incorporação de novos domínios de conhecimento

Para que novos conceitos sejam adicionados ao sistema de forma que possam ser utilizados em mensagens geradas a partir de linguagem natural é necessário que os novos conceitos incorporados estejam presentes na ontologia global da plataforma utilizada pelo conversor UCL. A incorporação de ontologias de novos domínios de conhecimento se faz a partir da extensão da ontologia global para a plataforma. Tal limitação dificulta a importação de ontologias já estruturadas de outros domínios sem causar inconsistências na ontologia global. Uma solução para amenizar as restrições do uso de uma ontologia global seria estabelecer processos de integração automática de novas ontologias a base de conhecimento. Contudo, como analisado por **[Pinto et al 2001]** este é um problema complexo que também possui limitações, pois necessita da intervenção do usuário a fim de evitar inconsistências nas base de conhecimento do domínio.

7.3.6 - Melhorias na usabilidade do System Editor

O IDE desenvolvido permite manipular a base de conhecimento do sistema incluindo conceitos, relações entre conceitos e comportamentos. Contudo, a interface de visualização da rede semântica do sistema através de uma árvore poderia ser aprimorada através de visualização de um grafo, de forma que as duas âncoras de uma relação pudessem ser visualizadas de maneira mais clara. Outro ponto que poderia ser aprimorado seria a transformação do System Editor em uma aplicação web ou em applet, de forma que não fosse necessário aos desenvolvedores instalar uma aplicação específica para o sistema.

Como se pode observar a partir da breve análise sobre a plataforma SemanticAgent, o trabalho proposto implementa diversas funcionalidades úteis à criação de agentes inteligentes. Contudo, o protótipo implementado não visa ser um trabalho definitivo que contemple todos os problemas existentes na complexa tarefa de criar agentes inteligentes.

7.4 – Considerações Finais

A plataforma SemanticAgent se caracteriza como um projeto de código aberto desenvolvido em Java para a criação de agentes inteligentes. O projeto que faz parte de uma linha de pesquisa deve ser aprimorado por futuros projetos de mestrado e doutorado desenvolvidos pelo grupos de pesquisadores do laboratório Intermidia do ICMC-USP. A plataforma SemanticAgent é disponibilizada de acordo com a licença de software livre LGPL da Free Software Foundation **[GNU 1999]**. O código fonte e os binários encontram-se disponíveis para download nos endereços http://java.icmc.usp.br/research/talkagent e http://talkagentfw.sf.net. A disponibilização do código para a comunidade reforça a iniciativa de tornar pública a tecnologia desenvolvida pela universidade. Espera-se que tal esforço se mostre útil e que a utilização do projeto por terceiros aponte problemas e idéias para melhoria do trabalho desenvolvido.

• *Bibliografia*

1. [**Abdallah 2002**] Sherief Abdallah, Soar2Java, 2002, online: http://www.geocities.com/sharios/soar.htm

2. [**Allamaraju et al 2001**] Subrahmanyam Allamaraju, Karl Averdal, Richard Browett et al, Professional Java Server Progrgramming J2EE, Wrox,2000, pp 216-339

3. [**Anderson et al 1998**] John R. Anderson, Christian J. Lebiere , The Atomic Components of Thought, Lawrence Erlbaum Assoc, 1998, pp 5-53

4. [**AOS Group 2003**] AOS Group, JACK Intelligent Agents Group., 2003, online: http://www.agent-software.com

5. [**Apple 1992**] Apple Inc, Knowledge Navigator, TV Commercial, 1992, online: http://www.billzarchy.com/clips/clips_apple_nav.htm

6. [**Arriola et al 1995**] George Arriola, Matthew Ford, MeeSook Hyun, Ken Larson, Bill Lemon, A Survey of Cognitive and Agent Architectures, 1995, online: http://ai.eecs.umich.edu/cogarch0/common/prop/fopc.html

7. [**Barry, 1996**] Alwyn Barry, Linda : Overview, 1996, online: http://www.cems.uwe.ac.uk/teaching/notes/PARALLEL/FUTURE/FUTUR220.HTM

8. [**Bellifemine et al 2001**] Fabio Bellifemine, Agostino Poggi,and Giovanni Rimassa, Developing multi agent systems with a FIPA-compliant agent framework, Software - Practice And Experience, 31, 2001, pp 103-128

9. [**Bergenti et al 2002**] Federico Bergenti and Agostino Poggi , LEAP: Lightweight Extensible Agent Platform, 2002, online: http://leap.crm-paris.com

10. [**Bhattacharyya 2002**] Rana Bhattacharyya, FTP Server Component, 2002, online: http://jakarta.apache.org/avalon

11. [**Blackshaw 2002**] Bruce P. Blackshaw, GNU FTP Client, 2002, online: http://www.gnu.org/software/java/java-software.html

12. [**Bogner et al 2001**] Myles Bogner, Jonathan Maletic & Stan Franklin, ConAg: a reusable framework for developing "conscious" software agents, The International Journal of Artificial Intelligence Tools, 2001, pp n.a.

13.[**Bosh 1997**] Philippe van den Bosh, La philosophie et le bonheur, Ed Flammarion, 1997, pp 82-83

14.[**Bradshaw 1996**] Jeffrey M Bradshaw, KAoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility , Knowledge Acquisition Workshop (KAW'96), Banff, Alberta, Canada, 1996, pp n.a.

15.[**Bradshaw 2001**] Jeffrey M Bradshaw, Terraforming Cyberspace, Computer IEEE Magazine, Vol. 34, No. 7, 2001, pp 48-57

16.[**Braeutigam 2003**] Falko Braeutigam, Ozone, The Open Source Java ODBMS, 2003, online: http://www.ozone-db.org

17.[**Brena 2003**] Ramón Brena, Sistemas Multiagentes, 2003, online: http://puccini.mty.itesm.mx/~rbrena/SMA

18.[**Brooks 1990**] Rodney A. Brooks, Elephants Don't Play Chess , Robotics and Autonomous Systems 6, 1990, pp 3-15

19.[**Burke 2002**] Murray Burke , The DAML Ontology Repository, 2002, online: http://www.daml.org/ontologies

20.[**Burstein 2002**] Mark Burstein, The Emacs DAML Editor, 2002, online: http://openmap.bbn.com/~burstein/daml-emacs

21.[**Buschmann 1996**] Frank Buschmann , Building Software with Patterns, European Conference on Pattern Languages of Programming (EuroPLoP 96), Kloster Irsee, Germany, 1996, pp n.a.

22.[**Carbonell et al 1991**] J. G. Carbonell, C. A Knoblock, S. Minton, , An Integrated Architecture for Prodigy, 1991, online: http://ai.eecs.umich.edu/cogarch0/prodigy

23.[**Carver et al 1994**] Norman Carver, Victor Lesser, The Evolution of Blackboard Control Architectures , Expert Systems with Applications, Special Issue on The Blackboard Paradigm and Its Applications, vol. 7, no. 1, 1994, pp 1-30

24.[**Castel 2002**] Felipe Castel , Ontological Computing , Communications of ACM vol 45, 2002, pp 29-30

25.[**Chandrasekaran et al 2002**] B. Chandrasekaran, J. R. Josephson and V. Richard Benjamins, The Ontology of Tasks and Methods , KAW'98 - Eleventh Workshop on Knowledge Acquisition, Modeling and ManagementVoyager Inn, Banff, Alberta, Canada, 2002, pp n.a.

26.[**Chappel 1996**] David Chappel , ActiveX and Ole, 1996 , Microsoft Press, pp 265-282

27.[**Chaudhri et al 1998**] V. K. Chaudhri, A. Farquhar, R. Fikes, P. D. Karp, & J. P. Rice. , OKBC: A programmatic foundation for knowledge base interoperability , Proceedings 15th National Conference on Artificial Intelligence (AAAI-98). Madison, Wisconsin. AAAI Press/MIT Press , 1998, pp 600-607

28.[**Chauhan 1997**] Deepika Chauhan, JAFMAS: A Java-based Agent Framework for Multiagent Systems Development and Implementation , University of Cincinnati OH - ECECS Department Thesis, 1997, pp 1-40

29.[**Chirico 2002**] Ugo Chirico, JIP, Java Internet Prolog, 2002, online: http://www.ugosweb.com/jiprolog/index.shtml

30.[**Choi et al 1999**] Yong S. Choi, Suk I. Yoo, Discovering Text Databases on the Internet: A Neural Net Agent Approach, International IEEE Systems, Man and Cybernetics Conference (SMC'99), 1999, pp n.a.

31.[**Chomsky 1976**] Noam Chomsky, Empiricism and Rationalism, On Language: Chomsky's Classic Works Language and Responsibility and Reflections on Language in One Volume, New Press, 1976, pp 81-100

32.[**Clarke 1968**] Arthur C. Clarke, 2001, A Space Odyssey, 1968, online: http://www.palantir.net/2001/meanings/clarke2.html

33.[**Corazzon 2002**] Raul Corazzon, Descriptive and Formal Ontology, 2002, online: http://www.formalontology.it

34.[**Cyc Corporation 2002**] Cyc Corporation, Opensyc, 2002, online: http://opencyc.sourceforge.net/daml/cyc.daml

35.[**Cycorp 2003**] Cycorp, Cyc Ontology Guide: Introduction, 2003, online: http://www-users.itlabs.umn.edu/classes/Fall-2002/csci5511/cyclexicon.pdf

36.[**Decker et al 1997**] Keith Decker, Anandeep Pannu, Katia Sycara, Mike Williamson, Designing Behaviors for Information Agents , Proceedings of the First International Conference on Autonomous Agents (Agents'97), 1997, pp 404-413

37.[**Demoen 2002**] Segment order preserving and generational garbage collection for Prolog, Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002, Lecture Notes in Computer Science, vol. 2257, Springer, 2002, pp. 299-317.

38.[**Dennett 1971**] Daniel Dennett, Intentional Stance , Journal of Philosophy 68, 1971, pp 87-106

39.[**Deri 2001**] Luca Deri , OpenDoc , 2001,
online: http://iamwww.unibe.ch/~scg/Research/ComponentModels/opendoc.html

40.[**Eberhart et al 1995**] Russ Eberhart e James Kennedy, PSO - Particle swarm optimization, 1995,
online: http://web.ics.purdue.edu/~hux/tutorials.shtml

41.[**Emorphia 2001**] Emorphia Open Source Development Group , FIPA-OS Agent Toolkit, 2001, online: http://fipa-os.sourceforge.net

42.[**Emorphia 2003**] Emorphia Inc, FiPA-OS Summary, 2003,
online: http://fipa-os.sourceforge.net/summary.htm

43.[**Esfeld 2002**] Michael Esfeld, Thorie de la connaissance, 2002, online: http://www.unil.ch/philo/documents.pdf/pdf_epistemo/cours2002-03/Cours-epist1.pdf

44.[**Farquhar et al 1996**] A. Farquhar, R. Fikes e J. Rice, The Ontolingua server: A tool for collaborative ontology construction , Technical report, Stanford KSL 96-26, 1996, pp 1-19

45.[**Figueiredo 2002**] Orlando de Andrade Figueiredo. , Implementação de Espaços de Tuplas do tipo Java Spaces. , Dissertação de Mestrado - ICMC - USP, 2002, pp 1-50

46.[**Fikes 2002**] Richard Fikes, Chimera DAML Ontology, 2002,
online: http://www.ksl.stanford.edu/projects/DAML

47.[**Finin 1998**] Tim Finin, Agents that Reason Logically, 1998,
online: http://www.csee.umbc.edu/471/lectures/6

48.[**Finin et al 1997**] Tim Finin, Yannis Labrou, and James Mayfiel, KQML as an agent communication language, Software Agents, MIT Press, 1997, pp 291 - 316

49.[**Finin et al 2001**] Tim Finin, Yannis Labrou, and James Mayfield, KIF101 A brief introduction to the knowledge interchange format , UMBC, University of Maryland, Baltimore County, 2001, pp 1-15, http://agents.umbc.edu/article

50.[**FIPA 2001**] Foundation for intelligent physical agents, FIPA ACL Message Structure Specification - RFC, 2001, online: http://www.fipa.org/rfcs

51.[**Genesereth et al 1994**] M. Genesereth and S. Ketchpel., Software Agents., Communications of the ACM, 37(7), 1994, pp 48--53,

52.[**GNU 1999**] Free Software Foundation, Inc., GNU Lesser General Public License, 1999, online: http://www.gnu.org/copyleft/lesser.html

53.[**Graham et al 2000**] John Graham, Victoria Windley, Daniel McHugh, Foster McGeary, David Cleaver, and Keith Decker , A Programming and Execution Environment for Distributed Multi Agent Systems Workshop , Fourth International Conference on Autonomous Agents, Barcelona, Spain ,2000, pp n.a.

54.[**Green 1998**] Christopher D. Green, The thoroughly modern Aristotle: Was he really a functionalist?, 1998, online: http://www.yorku.ca/christo/papers/Aristotle-functionalist.htm

55.[**Greenwood 2000**] Dominic Greenwood, Java Agent, 2000, online: http://www.java-agent.org/JAS_Intro.htm

56.[**Gruber 1993**] T. R. Gruber, A translation approach to portable ontologies, Knowledge Acquisition, v5 n2, Ed Academic Press, 1993, pp 199-220

57.[**Gruber 2002**] T. R. Gruber, Sharable Ontology Library, 2002, online: http://www-ksl.stanford.edu/knowledge-sharing/ontologies

58.[**Gruninger et al 2002**] Michael Gruninger, and Jintae Lee, Ontology Applications and Design, 2002, Communications of ACM, Feburary 2002 vol 45, pp 39-41

59.[**Guarino 1998**] Nicola Guarino, Formal Ontology in Information Systems., 1998 , Proceedings of International Conference On Formal Ontology in Information Systems, FOIS, IOS Press, 1998, pp 3-15

60.[**Harmelen et al 2001**] Frank Van Harmelen, Dieter Fensel, Ian Horrocks, Deborah McGuiness, Peter Patel-Schineider, Oil: An Ontology Infrastructure for the Semantic Web , IEEE Inteligent Systems 04, 2001, pp 38-45

61.[**Hayes et al 1983**] Frederick Hayes, Donald Waterman, Douglas Lenat, Building Expert Systems, Adilson Wesley, 1983, pp 3-59

62.[**Helin et al 2002**] Heikki Helin, Mikko Laukkanen, Performance Analysis of Software Agent Communication in Slow Wireless Networks, IEEE International Conference on Computer Communications and Networks, 2002, pp 1-8

63.[**Hendler 1999**] James Hendler, Is There an Intelligent Agent in Your Future?, Nature Magazine, March 11, 1999, pp n.a.

64.[**Hendler 2001**] James Hendler, Agents and the Sematic Web, 2001 , IEEE Inteligent Systems 04, 2001, pp 30-37

65.[**Hill 2002**] Ernest Friedman-Hill, Jess, the Rule Engine for the Java Platform, 2002, online: http://herzberg.ca.sandia.gov/jess

66.[**Hoeschl et al 2002**] Hugo Cesar Hoeschl, Tânia Cristina D´Agostini Bueno, André Bortolon, Eduardo da Silva Mattos e Ricardo Miranda Barcia, Aplicações em Engenharia do Conhecimento: UNL, 2002, online: http://www.ijuris.org/Capacitacao/Disciplinas_2002/UNL

67.[**Horstmann et al 1998**] Cay Horstmann, Gary Cornell, Core Java, Advanced Features, SunSoft Press,1998, pp 333-422

68.[**Inman 2000**] Dave Inman, "Ambiguity in Web searches" Proc. Internet and Multimedia Systems and Applications, Nevada, USA, 2000, pp 52-57

69.[**Kargupta et al 1997**] Hillol Kargupta, Ilker Hamzaoglu, Brian Stafford, Scalable, Distributed Data Mining Using An Agent Based Architecture , Proceedings of Knowledge Discovery And Data Mining, AAAI Press, 1997, pp 211-215

70.[**Kolp et al 2001**] Manuel Kolp, Paolo Giorgini, John Mylopoulos , An Organizational Perspective on Multi-agent Architectures , Proceedings of the Eighth International Workshop on Agent Theories, architectures, and languages Seattle, USA, August 2001, pp 1-8

71.[**KSL Network Services 2002**] Stanford KSL Network Services, The Ontolingua Server, 2002, online: http://www-ksl-svc.stanford.edu:5915

72.[**Labrou et al 1999**] Y.Labrou, T.Finin, Y.Peng, Agent Communication Language: The Current Landscape, IEEE Intelligent Systems, 04, 1999, pp 45-52

73.[**Laird et al 1995**] J.E. Laird, W.L.Johnson, et al., Simulated Intelligent Forces for Air: The SOAR/IFOR Project, Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation, 1995, pp 1-7

74.[**Lee et al 2001**] Tim Bernes Lee, James Hendler e Ora Lassila, The semantic Web , Scientific American - Volume 284, Number 5, May 2001, 2001, pp 34-43

75.[**Lemos et al 2002**] Alessandra Lemos, Rosario Girardi , Steferson Ferreira, ABARFI, Uma arquitetura reutilizável para a recuperação e filtragem de informação, 2002,
online: http://maae.deinf.ufma.br/Artigos/Grupo/ABARFI.PDF

76.[**Levow 1998**] Gina-Anne Levow, Characterizing and RecognizingSpoken Corrections in Human-Computer Dialog, 1998,
online: http://people.cs.uchicago.edu/~levow/talks/thesis/

77.[**Lieberman 1998**] Henry Lieberman, Integrating User Interface Agents with Conventional Applications , International Conference on Intelligent User Interfaces, January 1998, pp 15-24

78.[**Linhalis e Moreira 2002**] Flavia Linhalis e Dilvan Moreira, A Secure Interface for a Universal Data Server , Proc.of the Inter. Conference on Security and Management, Las Vegas USA, July 2002, CSREA Press , 2002, pp 286-291

79.[**Long 1994**] Byron Long, Natural Language as an Interface Style, 1994,
online: http://www.dgp.toronto.edu/people/byron/papers/nli.html

80.[**Lugmayr et al 2003**] Wolfgang Lugmayr, Sebastian Fischmeister, Stefan Haberl, Markus Mayer, Niko Popitsch, Georg Neuwirther e Antonio Batovanja, The Gypsy Project on Mobile Agents, 2003, online: http://www.infosys.tuwien.ac.at/Gypsy

81.[**Machado, et al 2001**] R Machado, R Bordini, Running AgentSpeak(L) agents on SIM_AGENT , Working Notes of the Fourth UK Workshop on Multi-Agent Systems (UKMAS), 2001, pp 27-32

82.[**Maes 1994**] Pattie Maes, Modeling Adaptive Autonomous Agents , Artificial Life: An Overview, Cambridge, MIT press, 1994, pp 135-162

83.[**Maes 1995**] Pattie Maes, Intelligent Software , Scientific American - 150 Anniversary Issue: "Key Technologies for the 21st Century", September 1995, pp 66-68

84.[**Maes 1997**] Pattie Maes, Software Agents Tutorial , CHI97 - ACM Conference on Human Factors in Computing Systems, 1997, pp n.a.

85.[**Maisonet 2003**] Josephine Maisonet, What is Swarm Intelligence? Why would society need it?, 2003, online: http://domanski.cs.csi.cuny.edu/csdeptweb/_disc3/000001b3.htm

86.[**Martin 2000**] James H. Martin, Speech and Language Processing, Prentice-Hall, 2000, pp 1-18

87.[**Martin et al 1999**] David L. Martin, Adam J. Cheyer, and Douglas B. Moran, The Open Agent Architecture: A framework for building distributed software systems, Applied Artificial Intelligence, 13, 1999, pp 91-128

88.[**Minsky 1988**] Marvin Minsky, The Society of Mind,Touchstone Books, 1988, pp 18-30

89.[**Minsky 1995**] Marvin Minsky, First Person: Marvin Minsky - The Society of Mind, 1995, CD-ROM

90.[**Minsky 2000**] Marvin Minsky, Common Sense Based Interfaces , Communications of the ACM, Vol. 43, No. 8, August 2000, pp 66-73

91.[**Montesco & Moreira 2000**] Carlos Montesco & Dilvan Moreira, Universal Communication Language (UCL) , Tese De Mestrado, ICMC-USP, 2000, pp 1-100

92.[**Montesco 2001**] Carlos Estombelo Montesco, Un Lenguaje de comunicación para Agentes en la Internet basado en Ontologias , V Congreso Internacional - Sudamericano de Ingeniería de Sistemas e Informática. Area de Inteligencia Artificial. Arequipa - Perú., 2001, pp n.a.

93.[**Montesco e Moreira 2002**] Carlos Montesco e Dilvan Moreira, "UCL - Universal Communication Language , Proc. of the First International Workshop on UNL, other Interlinguas and their Applications, Las Palmas Spain, June 2002, ELRA - European Language Resources Association, 2002, pp 33-37

94.[**Mora et al 1998**] Michael da Costa Móra, J.G. Lopes, J.G. Coelho, R Viccari, BDI models and systems: reducing the gap , Agents Theory, Architecture and Languages Workshop, Canarias. - Springer-Verlag, 1998, pp n.a

95.[**Mort Bay Consulting 2003**] Mort Bay Consulting , Java HTTP Server & Servlet Container, 2003, online: jetty.mortbay.org/jetty

96.[**Mueller 1998**] Erik T. Mueller, Natural language processing with ThoughtTreasure, 1998, online: http://www.signiform.com/tt/book

97.[**Mueller 2000**] Erik T. Mueller, A Calendar with Common Sense , Proceedings of the 2000 International Conference on Intelligent User Interfaces , ACM, New York, 2000, pp 198-201

98.[**Negroponte 1995**] Nicholas Negroponte, Beeing Digital, Editora Vintage, 1995, pp 89-183

99.[**Nierstrasz et al 1992**] Oscar Nierstrasz, Simon Gibbs, Dennis Tsichritzis, Component-oriented software development , Communications of the ACM, Vol.35 09, 1992, pp 160-165

100.[**NILC 2001**] NILC, Nucleo Interistitucional de Linguística Computacional, UNL: Universal Networking Language, 2001, online: http://143.107.183.175/site2001/projetos/unl.htm

101.[**Nissen 1995**] Mark Nissen, Intelligent Agents:A Technology and Business Application Analysis, 1995, online: http://www.mines.u-nancy.fr/~gueniffe/CoursEMN/I31/heilmann/heilmann.html

102.[**Noy et al 2001**] N. F. Noy, M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, & M. A. Musen, Creating Semantic Web Contents with Protege-2000 , IEEE Intelligent Systems 04, 2001, pp 60-71

103.[**Nwana 1996**] Hyacinth S. Nwana, Software Agents: An Overview, 1996 , The Knowledge Engineering Review, Volume 11, No 3, October/November 1996, pp 205-244

104.[**Nwana et al 1999**] Hyacinth S. Nwana & Divine T. Ndumu, A Perspective on Software Agents Research, The Knowledge Engineering Review, Vol 14, No 2, Cambridge Press, 1999, pp 1-18

105.[**Ogbuji 2000**] Chimezie Thomas Ogbuji, The future of natural-language processing , Unix Insider 12/2000, pp n.a

106.[**Oliveira et al 2001**] Osvaldo Oliveira Jr, Ronaldo T. Martins, Lucia Helena Rino, Maria das Gracas V. Nunes , O uso de interlíngua para comunicação via Internet: O Projeto UNL/Brasil, Série de Relatórios do NILC. NILC-TR-01-3, Julho 2001, pp 1-14

107.[**Orfail 1998**] Robert Orfail and Dan Harkey, Client/Server Programming with Java and Corba, Ed Wiley, 1998, pp 665-830

108.[**Parks 2001**] David Parks, Agent-Oriented Programming: A Practical Evaluation , 2001, online: http://www.cs.berkeley.edu/~davidp/cs263

109.[**Pinto et al 2001**] Helena Sofia Pinto e Joao P. Martins, A Methodology for Ontology Integration , K-CAP 01 - International Conference On Knowledge Capture, Victoria, British Columbia, Canada, 2001, pp 1-8

110.[**Popescu et al 2003**] Ana-Marie Popescu, Oren Etzioni, e Henry Kautz, PRECISE: A Reliable Natural Language Interface To Databases , International Conference on Intelligent User Interfaces, 2003, pp n.a.

111.[**Putnam 1975**] Hilary Putnam , Minds and Machines in Philosophical Papers Volume 2 , Mind, Language and Reality , Cambridge Press, 1975, pp 362-385

112.[**Quillian 1968**] M. Quillian , Semantic Memory, Semantic Information Processing, MIT Press, 1968, pp 227-270

113.[**Reingold et al 1999**] Eyal Reingold and Johnathan Nightingale, Early Works in AI, 1999,
online: http://psych.utoronto.ca/~reingold/courses/ai/early.html

114.[**Rezende 2000**] Solange Oliveira Rezende, Introdução à Aquisição de Conhecimento, 2000, online: http://labic.icmc.sc.usp.br/didatico/Ac

115.[**Rich et al. 1994**] Elaine Rich, Keving Knight, Inteligência Artificial, Makron Books, 1994, pp 35-49

116.[**Riley 2002**] Gary Riley, CLIPS, A Tool for Building Expert Systems, 2002, online: http://www.ghg.net/clips/CLIPS.html

117.[**Rosenbloom et al 1993**] Paul S. Rosenbloom, J.E. Laird, A Newell et al, The Soar Papers: Readings on Integrated Intelligence, MIT Press, 1993, pp 1-10

118.[**Rowe 1998**] Neil C Rowe, Artificial Intelligence Through Prolog, Ed Pretince Hall, 1998, pp 4-21

119.[**Russel et al 1995**] Stuart Russell and Peter Norvig, Artificial Intelligence, a Modern Approach, Ed. Prentice Hall, 1995, pp 31-50

120.[**Searle 1969**] J.R. Searle, Speech Acts: An Essay in Philosophy of Language, Cambridge University Press, 1969, pp 1-35

121.[**Sekine et al 1996**] Satoshi Sekine, Ralph Grishman, Apple Pie Parser - Proteus Project, 1996, online: http://nlp.cs.nyu.edu/app

122.[**Shen et al 1997**] Weiming Shen and Douglas Norrie, Facilitator, Mediator or Autonomous Agents , Proceedings of the Second International Workshop on CSCW in Design, Bangkok, Thailand, November, 1997, pp 119-124

123.[**Shoham 1993**] Yoav Shoham. , Agent-Oriented Programming , Artificial Intelligence (60), 1993, pp 51-92

124.[**Skarmeas 1999**] Nikolaos Skarmeas e Keith L. Clark, Component based agent construction , Autonomous Agents and Multi-Agent systems journal, 1999, pp n.a.

125.[**Soemarmo 2002**] Marmo Soemarmo , Introduction to Semantics - Intensive Course, 2002, online: http://www.ohiou.edu/dlcds

126.[**Sossolote et al 1997**] C.R.C. Sossolote, C. Zavaglia, L.H.M. Rino, M.G.V. Nunes, As Manifestações Morfossintáticas da Linguagem UNL no Português do Brasil, Notas do ICMSC, Nro.36, 1997, pp 1-10

127.[**Sowa 2000**] John F. Sowa, Ontology, metadata, and semiotics , Conceptual Structures: Logical, Linguistic, and Computational Issues, Lecture Notes in AI #1867, Springer-Verlag, Berlin, 2000 , pp 55-81

128.[**Sowa 2001**] John F. Sowa., Conceptual Graphs , Knowledge Representation: Logical, Philosophical, and Computational Foundations, Brooks Cole Publishing Co., Pacific Grove, CA, 2001, pp 476-488

129.[**Sowa 2003**] John F. Sowa, Semantic Networks, 2003, online: http://www.jfsowa.com/pubs/semnet.htm

130.[**Specia 2002**] Lucia Specia,). Um gerador conceitual para o português visando a tradução automática. Congresso de Pós Graduação,UFSCar - São Carlos, Agosto de 2002, pp n.a.

131.[**Stoffer 2000**] Shawn Stoffer, CYC: Building HAL, 2000, online: http://www.cs.unm.edu/~storm/docs/Cyc.htm

132.[**Sun MicroSystems 1998**] Sun MicroSystems, The JavaBeans Bridge for ActiveX - February 1998, online: http://java.sun.com/products/javabeans/software/bridge

133.[**Sun MicroSystems 2001**] Sun MicroSystems, Javabeans FAQ, 2001, online: http://java.sun.com/products/javabeans

134.[**Sun MicroSystems 2002**] Sun MicroSystems, Java Web Services Tutoria, 2002,online: http://java.sun.com/webservices

135.[**Sun Microsystems 2003**] Sun Microsystems, JavaSpaces, 2003, online: http://java.sun.com/products/javaspaces

136.[**Susse 2003**] Jennifer Susse, A New Look at British Emergentism , 2003, online: http://www-unix.oit.umass.edu/~jsusse/Emergentist%20Paper.doc

137.[**Sycara 1999**] K. Sycara, J. Lu, M Klusch, Matchmaking among Heterogeneous Agents on the Internet , Proceedings AAAI Spring Symposium on Intelligent Agents in Cyberspace, Stanford, USA, 1999, pp n.a

138.[**Togai 2002**] Togai Infralogic Inc, FuzzyCLIPS, 2002, online: http://www.ortech-engr.com/fuzzy/fzy_images/TogaiName.gif

139.[**Tsai 2003**] Wang-Ju Tsai, Synchronous multi-phrase multilingual deconverter, 2003, online: http://www-clips.imag.fr/geta/User/cgi-bin/jitkuep/stage/prg/DeconEn.html

140.[**Turban 1998**] Efraim Turban & Jay E. Aronson, Decision Support Systems and Intelligent Systems, Prentice Hall, 4th edition, 1998, pp 51-97

141.[**Turing 1950**] Alan Turing, The imitation game , Computing Machinery and Intelligence Mind, Vol. 59, No. 236, 1950, pp 433-460

142.[**Uchida et al 2001**] Hiroshi Uchida, Meiying Zhu, The Universal Networking Language beyond Machine Translation, 2001, online: http://www.unl.ias.unu.edu/publications/UNL-beyond%20MT.html

143.[**UNDL Foundation 2001**] UNU/IAS/UNL Center, The Universal Networking Language (UNL) Specifications, 2001, online: http://www.unl.ias.unu.edu/MissionUNLP.html

144.[**UNDL Foundation 2002**] UNU/IAS/UNL Center, Universal Networking Language, 2002, online: http://www.unld.org

145.[**Wauchope 1999**] Kenneth Wauchope, InterLACE (Interface to LACE), 1999, online: http://www.aic.nrl.navy.mil/~wauchope/interlace.html

146.[**Wauchope et al 1997**] Kenneth Wauchope, Stephanie Everett, Dennis Perzanowisky, Elaine Marsh, Natural Language in Four Spatial Interfaces, Proceedings of the Fifth Conference on Applied Natural Language Processing, Washington, DC, 1997, pp 8-11

147.[**Weber 2002**] Nico Weber, Studies in Words Using Computers: explicating word context and meaning, 2002, online: http://www.spr.fh-koeln.de/Personen/Weber/weber_papers

148.[**Wooldridge et al 1995**] Wooldridge, M. and Jennings, N., Intelligent Agents: Theoryand Practice, Knowledge Engineering Review - Volume 10 No 2, June 1995, pp n.a.

149.[**Zunino et al 2000**] Alejandro Zunino, Luis Berdún, Analía Amandi, JavaLog: un Lenguaje para la Programación de Agentes, Revista Iberoamericana de Inteligencia Artificial Vol. 13, 2000, pp 94-99

Apêndice A - Performativas FIPA

Performativa	Tipo	Uso
Confirm	Transferência de Informação	O agente remetente informa ao destinatário que dada proposição é verdadeira.
Disconfirm	Transferência de Informação	O agente remetente informa ao destinatário que dada proposição é falsa.
Inform	Transferência de Informação	O agente remetente informa ao destinatário certa proposição.
Inform-if	Transferência de Informação	O agente remetente informa ao agente destinatário se ele acredita que dada preposição é verdadeira ou falsa.
Inform-ref	Transferência de Informação	O agente remetente informa certa referência ao agente destinatário tal como um nome ou uma descrição.
Query-if	Transferência de Informação	O agente remetente questiona o destinatário se certa informação é ou não verdadeira.
Query-ref	Transferência de Informação	O agente remetente questiona o destinatário sobre o valor de alguma variável (Ex: o preço de um produto).
Subscribe	Transferência de Informação	O agente remetente solicita que seja informado pelo destinatário quando o valor de uma variável mudar.
Propagate	Transferência de Informação	O agente remetente envia uma mensagem para o agente destinatário cujo conteúdo deve ser retransmitido a um conjunto destinatários.
Proxy	Transferência de Informação	O agente remetente envia uma mensagem para o agente destinatário cujo conteúdo deve ser retransmitido a um conjunto destinatários. O agente de destinatário não precisa confiar nem endorsar o conteúdo da mensagem recebida do remetente.
Propose	Negociação	O agente remetente propõe algo ou responde a uma proposta onde restrições e condições podem ser aplicadas.
Call for Proposal	Negociação	O agente remetente solicita propostas do destinatário para executar um dada ação. O destinatário envia propostas ao agente remetente para uma certa ação dadas as pré-condições acordadas.
Accept-proposal	Negociação	O agente remetente informa que aceitou uma proposta para execução de uma dada ação que foi previamente informada ao agente destinatário.

Performativa	**Tipo**	**Uso**
Reject-proposal	Negociação	O agente remetente informa que recusou uma proposta para execução de uma dada ação que foi previamente informada ao agente destinatário.
Agree	Ação	O agente remetente concorda em executar uma ação.
Cancel	Ação	O agente remetente que não pode executar uma ação, antes acordada (agree).
Refuse	Ação	O agente remetente se recusa a executar uma ação.
Request	Ação	O agente remetente solicita que o destinatário execute alguma ação.
Request-when	Ação	O agente remetente solicita que o destinatário execute alguma ação quando uma dada condição for verdadeira.
Request-whenever	Ação	O agente remetente solicita que o destinatário execute alguma ação quando uma dada condição for verdadeira e toda vez que a condição voltar a ser verdadeira.
Failure	Gerenciamento de Erros	O agente remetente informa ao destinatário erro ao executar uma dada ação.
Not-understood	Gerenciamento de Erros	O agente remetente informa ao destinatário que não entendeu uma mensagem recebida.

Apêndice B - Relações e Atributos Definidos pela UNL / UCL

Relações Definidas pela UNL / UCL

ICL	Inclusão	Representação de hiperonímia (super e subclasses) ou meronímia (relação parte-todo).
AGT	Agente	Define um agente que causa uma ação volitiva, por exemplo, um objeto.
AND	União	Define uma relação de conjunção envolvendo dois objetos ou conceitos
AOJ	Objeto Atributivo	Define um objeto de um atributo
BAS	Grau	Define um objeto usado como a base para expressar um grau de comparação.
BEN	Beneficiário	Define um beneficiário que está indiretamente relacionado com outro objeto ou evento.
CAG	Concomitância co-agência	Define um objeto (não evidente) que inicia implicitamente um evento de forma simultânea.
CAO	Concomitância co-objeto com atributos	Define um objeto (não evidente) dentro de um estado, efetuando-se de forma simultânea.
CNT	Conteúdo	Define um conceito equivalente.
COB	Concomitância co-objeto afetado	Define um objeto que está diretamente afetado por um evento implícito de forma simultânea ou por uma condição.
CON	Condição	Define uma condição que causa (voluntária ou involuntariamente) a ocorrência de um evento.
COO	Co-ocorrência	Define uma progressão simultânea de eventos.
DUR	Duração	Define um período de tempo de existência de um estado ou evento.
FMT	Alcance: origem-destino	Define uma abrangência de objetos ou eventos.
FRM	Origem	Define a origem de um objeto.

GOL	Estado Final	Define o local (físico ou lógico) de um agente/objeto
INS	Instrumento	Define o instrumento utilizado para levar a cabo um evento.
MAN	Maneira, modo	Define a maneira de levar a cabo um evento ou caracterizar um estado.
MET	Método	Define um método para levar a cabo um evento.
MOD	Modificador	Define um objeto evidenciando-o de forma restrita.
NAM	Nome	Define o nome de um objeto.
OBJ	Objeto afetado	Define um objeto que está diretamente afetado por um evento ou estado.
OPL	Lugar objetivo	Define um lugar onde um evento acontece.
OR	Disjunção	Define uma relação de disjunção entre dois conceitos.
PER	Proporção, taxa ou distribuição	Define uma base ou unidade de proporção, taxa ou distribuição.
PLC	Lugar	Define o lugar onde acontece um evento ou está presente um objeto com um estado determinado.
PLF	Lugar inicial	Define o lugar onde um evento começa ou define o lugar onde um estado começa a ser verdadeiro.
PLT	Lugar final	Define o lugar onde um evento termina ou define o lugar onde um estado começa a ser falso.
POF	Parte de	Define um conceito que forma parte de outro.
POS	Possuidor	Define o possuidor de um objeto.
PTN	Companheiro	Define um objeto de forma não evidente, mas imprescindível para que se comece uma ação.
PUR	Propósito ou objetivo	Define o propósito ou objetivo de um agente relacionado a um evento ou o propósito de um objeto que existe.
QUA	Quantidade	Define a unidade associada à quantidade de um objeto ou grau de mudança.

RSN	Razão	Define a razão pela qual um evento ou um determinado estado aconteceria.
SCN	Cenário	Define um mundo virtual onde acontece um evento, ou um estado se torna verdadeiro, ou onde um objeto existe.
SEQ	Seqüência	Define um evento ou estado prévio ao evento ou estado que está sendo evidenciado.
SRC	Estado inicial	Define o local (físico ou lógico) ou estado de um agente/objeto relativo a um evento.
TIM	Tempo	Define o tempo em que acontece um evento ou o tempo em que um estado é válido
TMF	Tempo inicial	Define o tempo em que um evento começa ou o tempo inicial em que um estado começa a ser verdadeiro.
TMT	Tempo final	Define o tempo em que um evento termina ou o tempo em que um estado se torna falso.
TO	Destino	Define o destino de um objeto.
VIA	Lugar ou estado intermediário	Define o lugar ou estado intermediário de um evento.

Atributos Definidos pela UNL / UCL

Tense (Tempo Verbais)	Past Present Future	**Focus** (Tipo de ênfase)	Emphasis Entry (beginning of a concept) Qfocus (focus) Theme Title
Aspect (Tempos específicos)	Begin-soon Begin-just (In) Progress End-soon End-just Complete(d) State Repeat	**Convention** (Símbolos convencionais)	Angle_bracket Double_parenthesis Double_quotation Parenthesis Pl(ural) Single_quotation Square_bracket
Reference (Tipo de Referência)	Generic Def(ined) Indef(ined) Not Order		

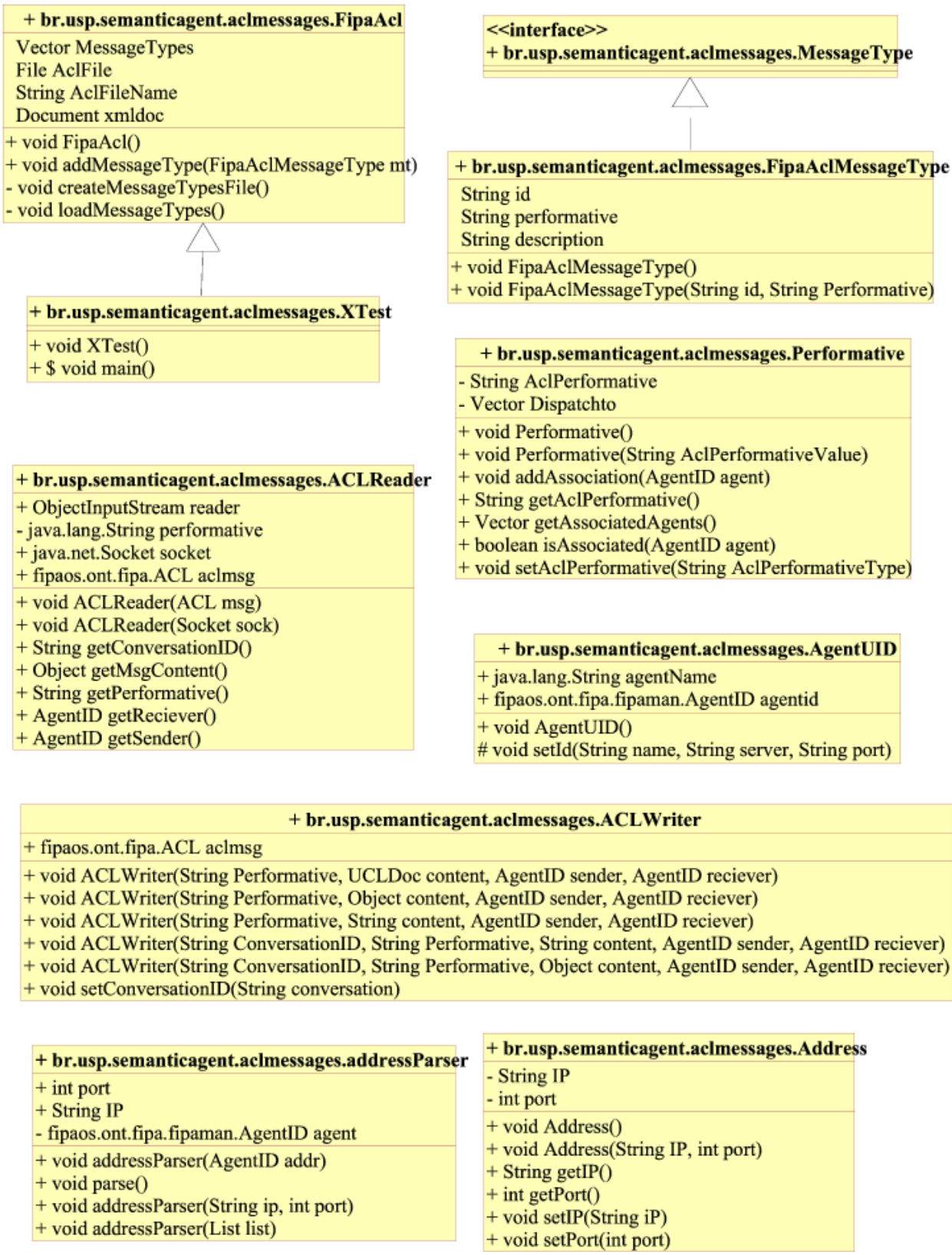
Attitude (Tipo de atitude)	Affirmative Confirmation Exclamation Imperative Interrogative Invitation Politeness Respect Vocative	
Viewpoint (quão valido é o conceito)	Ability Ability-past Apodosis-real Apodosis-unreal Apodosis-cond Conclusion Custom Expectation Grant Grant-not Insistence Intention Inevitability May Obligation Obligation-not Possibility Probability Should Unexpected-presumption Unexpected-consequence Will	

Apêndice C - Diagramas UML

LIB COMMON

Biblioteca de objetos comuns à diversos agentes da plataforma. Incluí os seguintes pacotes.

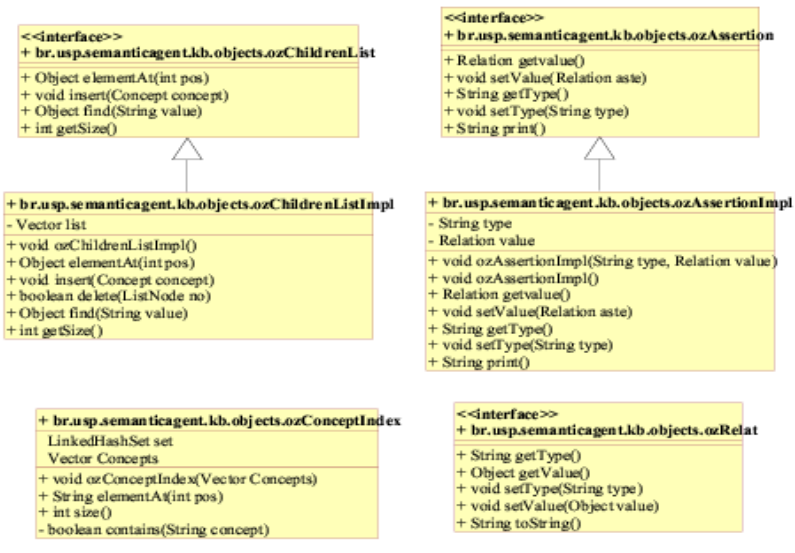
br.usp.semanticagent.aclmessages
br.usp.semanticagent.basicagent
br.usp.semanticagent.components
br.usp.semanticagent.kb
br.usp.semanticagent.kb-objects
br.usp.semanticagent.sentence
br.usp.semanticagent.ucl
br.usp.semanticagent.util


+ br.usp.semanticagent.aclmessages.FipaAcl
Vector MessageTypes
File AclFile
String AclFileName
Document xmldoc
+ void FipaAcl()
+ void addMessageType(FipaAclMessageType mt)
- void createMessageTypesFile()
- void loadMessageTypes()
<<interface>>
+ br.usp.semanticagent.aclmessages.MessageType
+ br.usp.semanticagent.aclmessages.FipaAclMessageType
String id
String performative
String description
+ void FipaAclMessageType()
+ void FipaAclMessageType(String id, String Performative)
+ br.usp.semanticagent.aclmessages.XTest
+ void XTest()
+ $ void main()
+ br.usp.semanticagent.aclmessages.Performative
- String AclPerformative
- Vector Dispatchto
+ void Performative()
+ void Performative(String AclPerformativeValue)
+ void addAssociation(AgentID agent)
+ String getAclPerformative()
+ Vector getAssociatedAgents()
+ boolean isAssociated(AgentID agent)
+ void setAclPerformative(String AclPerformativeType)
+ br.usp.semanticagent.aclmessages.ACLReader
+ ObjectInputStream reader
- java.lang.String performative
+ java.net.Socket socket
+ fipaos.ont.fipa.ACL aclmsg
+ void ACLReader(ACL msg)
+ void ACLReader(Socket sock)
+ String getConversationID()
+ Object getMsgContent()
+ String getPerformative()
+ AgentID getReciever()
+ AgentID getSender()
+ br.usp.semanticagent.aclmessages.AgentUID
+ java.lang.String agentName
+ fipaos.ont.fipa.fipaman.AgentID agentid
+ void AgentUID()
# void setId(String name, String server, String port)
+ br.usp.semanticagent.aclmessages.ACLWriter
+ fipaos.ont.fipa.ACL aclmsg
+ void ACLWriter(String Performative, UCLDoc content, AgentID sender, AgentID reciever)
+ void ACLWriter(String Performative, Object content, AgentID sender, AgentID reciever)
+ void ACLWriter(String Performative, String content, AgentID sender, AgentID reciever)
+ void ACLWriter(String ConversationID, String Performative, String content, AgentID sender, AgentID reciever)
+ void ACLWriter(String ConversationID, String Performative, Object content, AgentID sender, AgentID reciever)
+ void setConversationID(String conversation)
+ br.usp.semanticagent.aclmessages.addressParser
+ int port
+ String IP
- fipaos.ont.fipa.fipaman.AgentID agent
+ void addressParser(AgentID addr)
+ void parse()
+ void addressParser(String ip, int port)
+ void addressParser(List list)
+ br.usp.semanticagent.aclmessages.Address
- String IP
- int port
+ void Address()
+ void Address(String IP, int port)
+ String getIP()
+ int getPort()
+ void setIP(String iP)
+ void setPort(int port)

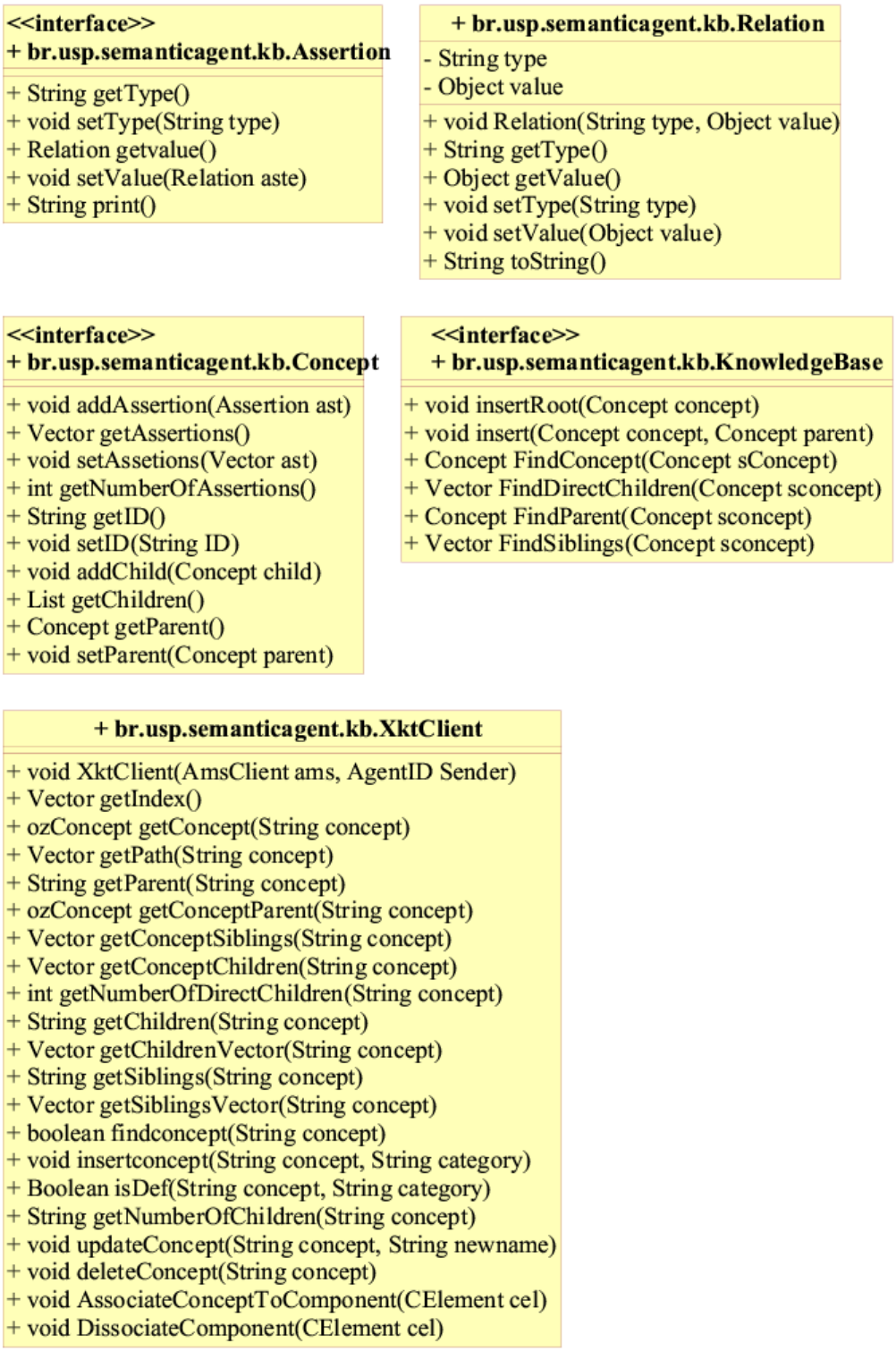

+ br.usp.semanticagent.basicagent.AgentDescriptor
- String name
- String address
- boolean running
+ void AgentDescriptor()
+ String getAddress()
+ String getName()
+ boolean isRunning()
+ void setAddress(String address)
+ void setName(String name)
+ void setRunning(boolean running)
+ br.usp.semanticagent.basicagent.SharpServerAgent
# fipaos.ont.fipa.fipaman.AgentID agentid
# java.net.ServerSocket server
# java.util.Vector sentMessages
# java.util.Vector recievedMessages
# AmsClient amsclient
- int running_clients
- int port
- boolean running
+ void SharpServerAgent()
+ void SharpServerAgent(boolean connect)
+ void Connect()
+ void disRegister()
+ void register()
+ void run()
+ void setAMS(String IP, int port)
+ void setId(String name, String server, String port)
# void startConnection(int port, int maxConnections)
+ String createConversationID()
+ int getPort()
+ boolean isRunning()
+ void setPort(int port)
+ void setRunning(boolean running)
+ br.usp.semanticagent.basicagent.SharpAgent
# AgentID agentid
# boolean running
# Socket connection
# Socket outputSocket
# AmsClient amsclient
# String conversationID
+ void SharpAgent()
+ void SharpAgent(AgentID agentid)
+ void SharpAgent(AgentID agentid, AmsClient amsclient)
# void init()
# ACL readMsg()
+ void sendMsg(ACL msg)
# void setId(AgentID agentid)
# void setId(String name, String server, String port)
+ AgentID getAgentID(String AgentName)
+ br.usp.semanticagent.basicagent.AmsClient
+ fipaos.ont.fipa.fipaman.AgentID amsServer
+ addressParser amsAddr
# fipaos.ont.fipa.fipaman.AgentID agentid
- String amsaddr
- Socket connection
- String configfile
- boolean cfg
- boolean param
- boolean connected
+ void AmsClient()
+ void AmsClient(String configfile)
+ void AmsClient(String host, int Port)
+ void Connect()
+ void disConnect()
+ void setAMS(String ip, int port)
+ void disRegister(AgentID agentid)
+ AgentID getAgentID(String AgentName)
+ void register(AgentID agentid)
+ Vector listAgents()
+ AgentID getSelfID()
+ String getAmsAddr()
# ACL readMsg()
+ void sendMsg(ACL msg)
+ boolean isConnected()
+ br.usp.semanticagent.basicagent.AgtConnector
- AmsClient amsclient
- String recieverName
- AgentID reciever
- AgentID sender
- ACLWriter aclwriter
- addressParser addr
- Socket con
- ObjectOutputStream output
- ACL aclmsg
- ObjectInputStream input
- ACLReader aclreader
+ void AgtConnector(AmsClient ams, AgentID Sender, String Reciever)
+ void AgtConnector(AmsClient ams, AgentID Sender, AgentID Reciever)
+ void Connect()
+ void disConnect()
+ void SendString(String perf, String msg)
+ void SendObject(String perf, Object obj)
+ void sendMessage(ACL msg)
+ String ReadString()
+ Object ReadObject()
+ ACL ReadMessage()
+ AgentID getMsgSender()
+ String getMsgPerformative()
+ String getMsgConversationID()
+ Address getRecieverAddr()
+ AmsClient getAmsclient()

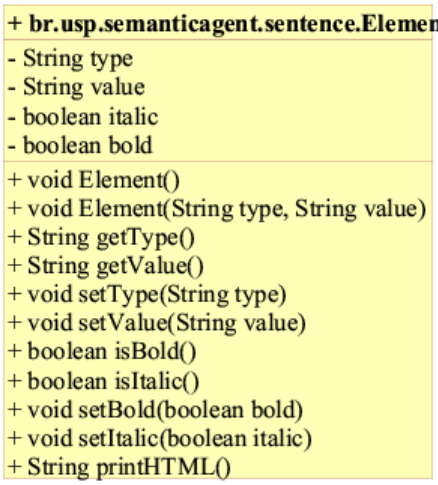

<<interface>>
+ br.usp.semanticagent.component.Component
+ String toString()
+ String getFileName()
+ void setFileName_update(String filename)
+ void execute(String MethoodName, Vector Parameters)
+ String getAncestor()
+ void setAncestor_update(String Ancestor)
+ String getBeanName()
+ void setBeanName_update(String beanName)
+ Vector getMethods()
+ CompLink getComponentInfo()
+ void setComponentInfo_update(CompLink complink)
+ void setComponentInfo_update(String filename)
+ CMethod getMethod(String methodName)
+ void AssociateConceptToMethod_update(String concept, String method)
+ void AssociateConceptToParam_update(String concept, String method, String param)
+ void DissociateConceptToMethod_update(String concept, String method)
+ void DissociateConceptToParam_update(String concept, String method, String param)
+ br.usp.semanticagent.component.CParameter
- String type
- Object value
- String name
- Vector AssociatedConcepts
+ void CParameter()
+ void CParameter(String type, String value)
+ String getType()
+ Object getValue()
+ void setType(String type)
+ void setValue(Object value)
+ String getName()
+ void setName(String name)
+ String toString()
+ void associateConcept(String concept)
+ void dissociateConcept(String concept)
+ Vector getAssociatedConcepts()
+ int getNumberOfAssociatedConcepts()
+ String getID()
+ br.usp.semanticagent.component.ComponentImpl_Proxy
$ long serialVersionUID
+ void ComponentImpl_Proxy()
+ void ComponentImpl_Proxy(ObjectID oid, OzoneInterface link)
+ $ Component createObject(OzoneInterface link)
+ String toString()
+ Vector getMethods()
+ CMethod getMethod(String arg0)
+ void execute(String arg0, Vector arg1)
+ void setComponentInfo_update(CompLink arg0)
+ void setComponentInfo_update(String arg0)
+ String getAncestor()
+ String getBeanName()
+ void setAncestor_update(String arg0)
+ void setBeanName_update(String arg0)
+ void AssociateConceptToMethod_update(String arg0, String arg1)
+ void DissociateConceptToMethod_update(String arg0, String arg1)
+ void AssociateConceptToParam_update(String arg0, String arg1, String arg2)
+ void DissociateConceptToParam_update(String arg0, String arg1, String arg2)
+ CompLink getComponentInfo()
+ String getFileName()
+ void setFileName_update(String arg0)
+ ObjectID getObjectID()
+ br.usp.semanticagent.component.ComponentImpl
- String component
- String filename
- String Ancestor
- CompLink clk
- String beanName
- String fileName
+ void ComponentImpl()
+ void setComponentInfo_update(String filename)
- boolean validname(String name)
+ String toString()
+ void execute(String MethoodName, Vector Parameters)
+ String getAncestor()
+ String getBeanName()
+ CMethod getMethod(String methodName)
+ void addMethod(CMethod method)
+ Vector getMethods()
+ void setAncestor_update(String Ancestor)
+ void setBeanName_update(String beanName)
+ void AssociateConceptToMethod_update(String concept, String method)
+ void DissociateConceptToMethod_update(String concept, String method)
+ void AssociateConceptToParam_update(String concept, String method, String param)
+ void DissociateConceptToParam_update(String concept, String method, String param)
+ CompLink getComponentInfo()
+ void setComponentInfo_update(CompLink clk)
+ String getFileName()
+ void setFileName_update(String fileName)
+ br.usp.semanticagent.component.CElement
- String Concept
- String Component
- String Method
- String Param
- int numberofparam
- int paramnumber
- String Type
+ void CElement()
+ void CElement(String concept, String bean, String method, String param)
+ String getComponent()
+ String getConcept()
+ String getMethod()
+ String getMethodName()
+ String getParam()
+ String getParamName()
+ void setComponent(String component)
+ void setConcept(String concept)
+ void setMethod(String method)
+ void setParam(String param)
+ boolean isMethod()
+ boolean isParam()
+ String getType()
+ void setType(String type)
+ int getNumberofparam()
+ int getParamnumber()
+ void setNumberofparam(int numberofparam)
+ void setParamnumber(int paramnumber)
+ br.usp.semanticagent.component.CMethod
- String name
- Vector parameters
- String returntype
- String permission
- Vector AssociatedConcepts
+ void CMethod()
+ void addParameter(CParameter property)
+ CParameter getParameter(int count)
+ CParameter getParameter(String name)
+ int getParametersNumber()
+ String getName()
+ String getPermission()
+ String getReturntype()
+ String toString()
+ void associateConcept(String concept)
+ void dissociateConcept(String concept)
+ void setReturntype(String returntype)
+ void setPermission(String permission)
+ void setName(String name)
+ Vector getAssociatedConcepts()
+ int getNumberOfAssociatedConcepts()
+ String getID()
+ br.usp.semanticagent.component.CRclient
+ void CRclient(AmsClient ams, AgentID Sender)
+ void upload(String filename)
+ Vector getComponentList()
+ void DeleteComponent(String componentName)
+ void Associate(CElement cel)
+ void Dissociate(CElement cel)
+ String runMethod(String component, String Method, Vector params)
+ String runMethod(String component, String Method, Vector params, String assocparam)
+ br.usp.semanticagent.component.CompLink
- Vector methods
- Vector atributes
- String name
- String Ancestor
+ void CompLink()
+ CMethod getMethod(int count)
+ int getNumeberofMethods()
+ CMethod getMethod(String methodname)
+ void addMethod(CMethod method)
+ String getAncestor()
+ String getName()
+ void setAncestor(String ancestor)
+ void setName(String name)
+ Vector getMethods()
+ void setMethods(Vector methods)

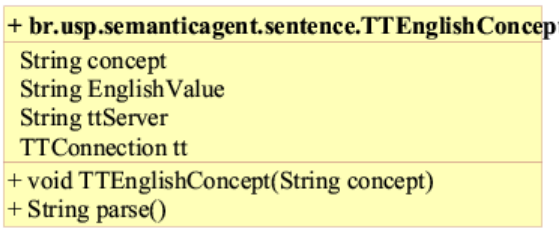

<<interface>>
+ br.usp.semanticagent.kb.objects.ozConcept
+ void addChild_update(String childID)
+ int getNumberOfDirectChildren()
+ void setNumberOfDirectChildren_update(int node)
+ int getNumberOfChildren()
+ void setNumberOfChildren_update(int noc)
+ Vector getChildrenVector()
+ void setChildren_update(Vector children)
+ String getID()
+ void setID_update(String ID)
+ Vector getAssertions()
+ void addAssertion_update(Assertion ast)
+ void setAssetions_update(Vector ast)
+ String getParentID()
+ void setParent_update(String parentID)
+ int getNumberOfAssertions()
+ Vector print()
+ void addComponentLink_update(CElement cel)
+ Vector getComponetLinks()
+ void setCompontLinks_update(Vector cels)
+ void removeCompLink(CElement cel)
+ br.usp.semanticagent.kb.objects.ozRelationImpl
- String type
- Object value
+ void ozRelationImpl(String type, Object value)
+ String getType()
+ Object getValue()
+ void setType(String type)
+ void setValue(Object value)
+ String toString()
+ br.usp.semanticagent.kb.objects.ozConceptImpl_Proxy
$ long serialVersionUID
+ void ozConceptImpl_Proxy()
+ void ozConceptImpl_Proxy(ObjectID oid, OzoneInterface link)
+ $ ozConcept createObject(OzoneInterface link)
+ $ ozConcept createObject(String arg0, OzoneInterface link)
+ Vector print()
+ void setParent_update(String arg0)
+ String getParentID()
+ Vector getAssertions()
+ void addAssertion_update(Assertion arg0)
+ void setAssetions_update(Vector arg0)
+ int getNumberOfAssertions()
+ Vector getChildrenVector()
+ void addChild_update(String arg0)
+ String getID()
+ void setID_update(String arg0)
+ int getNumberOfChildren()
+ int getNumberOfDirectChildren()
+ void setNumberOfChildren_update(int arg0)
+ void setNumberOfDirectChildren_update(int arg0)
+ void setChildren_update(Vector arg0)
+ void addComponentLink_update(CElement arg0)
+ Vector getComponetLinks()
+ void setCompontLinks_update(Vector arg0)
+ void removeCompLink(CElement arg0)
+ ObjectID getObjectID()
+ br.usp.semanticagent.kb.objects.ozConceptImpl
- String name
- Vector assertions
- Vector compLinks
- HashSet children
- String parentID
- int noa
- int node
- int noc
+ void ozConceptImpl()
+ void ozConceptImpl(String name)
+ void setParent_update(String parentID)
+ String getParentID()
+ Vector getAssertions()
+ void addAssertion_update(Assertion ast)
+ void setAssetions_update(Vector ast)
+ int getNumberOfAssertions()
+ Vector getChildrenVector()
+ void addChild_update(String childID)
+ String getID()
+ void setID_update(String ID)
+ Vector print()
+ int getNumberOfChildren()
+ int getNumberOfDirectChildren()
+ void setNumberOfChildren_update(int noc)
+ void setNumberOfDirectChildren_update(int node)
+ void setChildren_update(Vector children)
+ void addComponentLink_update(CElement cel)
+ Vector getComponetLinks()
+ void setCompontLinks_update(Vector cels)
+ void removeCompLink(CElement cel)

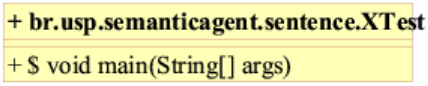

<<interface>>
+ br.usp.semanticagent.kb.objects.ozChildrenList
+ Object elementAt(int pos)
+ void insert(Concept concept)
+ Object find(String value)
+ int getSize()
<<interface>>
+ br.usp.semanticagent.kb.objects.ozAssertion
+ Relation getvalue()
+ void setValue(Relation aste)
+ String getType()
+ void setType(String type)
+ String print()
+ br.usp.semanticagent.kb.objects.ozChildrenListImpl
- Vector list
+ void ozChildrenListImpl()
+ Object elementAt(int pos)
+ void insert(Concept concept)
+ boolean delete(ListNode no)
+ Object find(String value)
+ int getSize()
+ br.usp.semanticagent.kb.objects.ozAssertionImpl
- String type
- Relation value
+ void ozAssertionImpl(String type, Relation value)
+ void ozAssertionImpl()
+ Relation getvalue()
+ void setValue(Relation aste)
+ String getType()
+ void setType(String type)
+ String print()

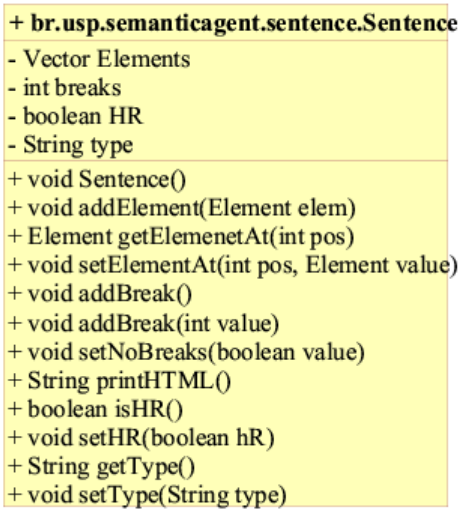
+ br.usp.semanticagent.kb.objects.ozConceptIndex
LinkedHashSet set
Vector Concepts
+ void ozConceptIndex(Vector Concepts)
+ String elementAt(int pos)
+ int size()
- boolean contains(String concept)
<<interface>>
+ br.usp.semanticagent.kb.objects.ozRelat
+ String getType()
+ Object getValue()
+ void setType(String type)
+ void setValue(Object value)
+ String toString()

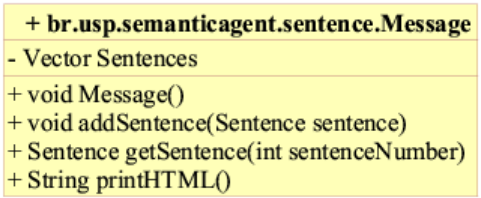

<<interface>>
+ br.usp.semanticagent.kb.Assertion
+ String getType()
+ void setType(String type)
+ Relation getvalue()
+ void setValue(Relation aste)
+ String print()
+ br.usp.semanticagent.kb.Relation
- String type
- Object value
+ void Relation(String type, Object value)
+ String getType()
+ Object getValue()
+ void setType(String type)
+ void setValue(Object value)
+ String toString()
<<interface>>
+ br.usp.semanticagent.kb.Concept
+ void addAssertion(Assertion ast)
+ Vector getAssertions()
+ void setAssetions(Vector ast)
+ int getNumberOfAssertions()
+ String getID()
+ void setID(String ID)
+ void addChild(Concept child)
+ List getChildren()
+ Concept getParent()
+ void setParent(Concept parent)
<<interface>>
+ br.usp.semanticagent.kb.KnowledgeBase
+ void insertRoot(Concept concept)
+ void insert(Concept concept, Concept parent)
+ Concept FindConcept(Concept sConcept)
+ Vector FindDirectChildren(Concept sconcept)
+ Concept FindParent(Concept sconcept)
+ Vector FindSiblings(Concept sconcept)
+ br.usp.semanticagent.kb.XktClient
+ void XktClient(AmsClient ams, AgentID Sender)
+ Vector getIndex()
+ ozConcept getConcept(String concept)
+ Vector getPath(String concept)
+ String getParent(String concept)
+ ozConcept getConceptParent(String concept)
+ Vector getConceptSiblings(String concept)
+ Vector getConceptChildren(String concept)
+ int getNumberOfDirectChildren(String concept)
+ String getChildren(String concept)
+ Vector getChildrenVector(String concept)
+ String getSiblings(String concept)
+ Vector getSiblingsVector(String concept)
+ boolean findconcept(String concept)
+ void insertconcept(String concept, String category)
+ Boolean isDef(String concept, String category)
+ String getNumberOfChildren(String concept)
+ void updateConcept(String concept, String newname)
+ void deleteConcept(String concept)
+ void AssociateConceptToComponent(CElement cel)
+ void DissociateComponent(CElement cel)

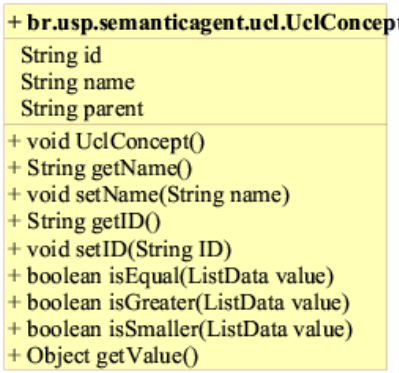

+ br.usp.semanticagent.sentence.Element
- String type
- String value
- boolean italic
- boolean bold
+ void Element()
+ void Element(String type, String value)
+ String getType()
+ String getValue()
+ void setType(String type)
+ void setValue(String value)
+ boolean isBold()
+ boolean isItalic()
+ void setBold(boolean bold)
+ void setItalic(boolean italic)
+ String printHTML()

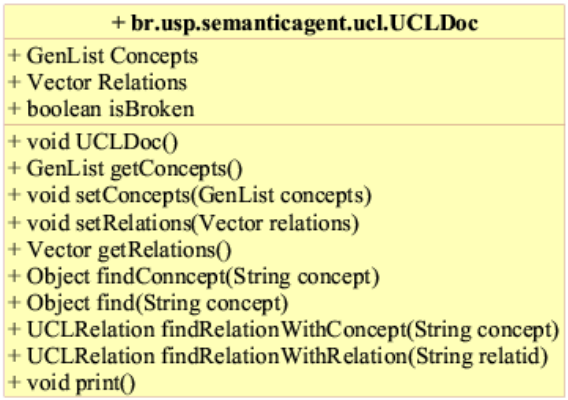

+ br.usp.semanticagent.sentence.TTEnglishConcept
String concept
String EnglishValue
String ttServer
TTConnection tt
+ void TTEnglishConcept(String concept)
+ String parse()

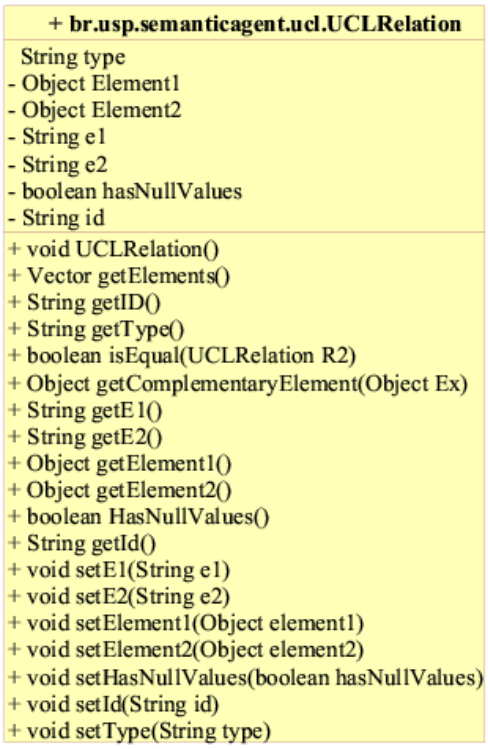

+ br.usp.semanticagent.sentence.XTest
+ $ void main(String[] args)

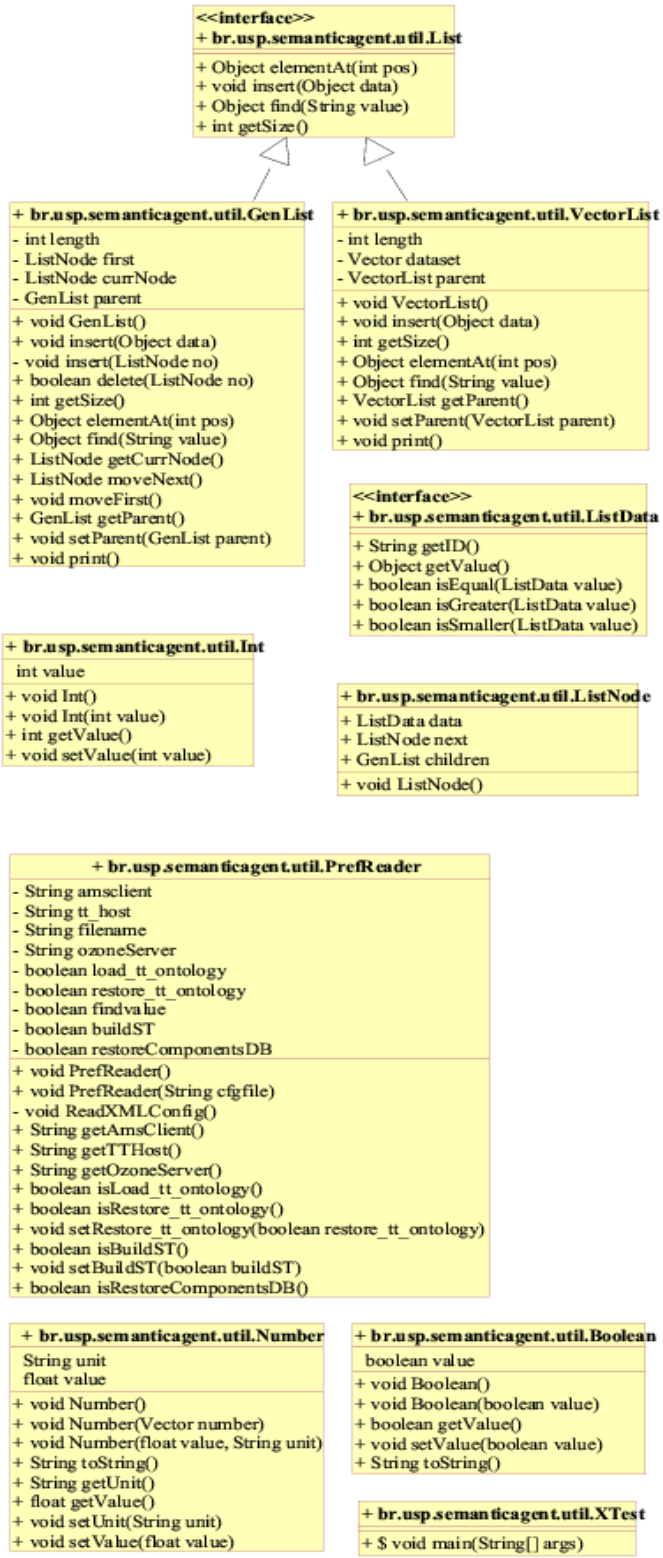

+ br.usp.semanticagent.sentence.Sentence
- Vector Elements
- int breaks
- boolean HR
- String type
+ void Sentence()
+ void addElement(Element elem)
+ Element getElemenetAt(int pos)
+ void setElementAt(int pos, Element value)
+ void addBreak()
+ void addBreak(int value)
+ void setNoBreaks(boolean value)
+ String printHTML()
+ boolean isHR()
+ void setHR(boolean hR)
+ String getType()
+ void setType(String type)


+ br.usp.semanticagent.sentence.Message
- Vector Sentences
+ void Message()
+ void addSentence(Sentence sentence)
+ Sentence getSentence(int sentenceNumber)
+ String printHTML()


+ br.usp.semanticagent.ucl.UclConcept
String id
String name
String parent
+ void UclConcept()
+ String getName()
+ void setName(String name)
+ String getID()
+ void setID(String ID)
+ boolean isEqual(ListData value)
+ boolean isGreater(ListData value)
+ boolean isSmaller(ListData value)
+ Object getValue()


+ br.usp.semanticagent.ucl.UCLDoc
+ GenList Concepts
+ Vector Relations
+ boolean isBroken
+ void UCLDoc()
+ GenList getConcepts()
+ void setConcepts(GenList concepts)
+ void setRelations(Vector relations)
+ Vector getRelations()
+ Object findConncept(String concept)
+ Object find(String concept)
+ UCLRelation findRelationWithConcept(String concept)
+ UCLRelation findRelationWithRelation(String relatid)
+ void print()

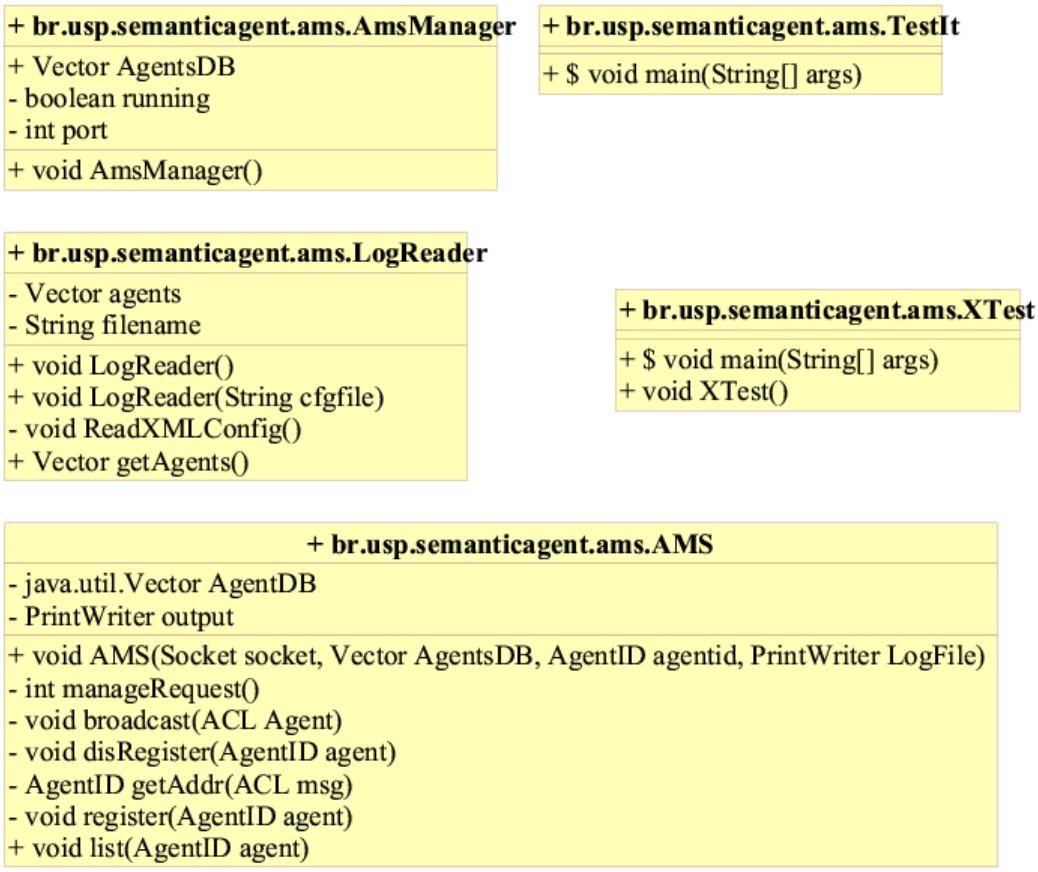

+ br.usp.semanticagent.ucl.UCLRelation
String type
- Object Element1
- Object Element2
- String e1
- String e2
- boolean hasNullValues
- String id
+ void UCLRelation()
+ Vector getElements()
+ String getID()
+ String getType()
+ boolean isEqual(UCLRelation R2)
+ Object getComplementaryElement(Object Ex)
+ String getE1()
+ String getE2()
+ Object getElement1()
+ Object getElement2()
+ boolean HasNullValues()
+ String getId()
+ void setE1(String e1)
+ void setE2(String e2)
+ void setElement1(Object element1)
+ void setElement2(Object element2)
+ void setHasNullValues(boolean hasNullValues)
+ void setId(String id)
+ void setType(String type)


<<interface>>
+ br.usp.semanticagent.util.List
+ Object elementAt(int pos)
+ void insert(Object data)
+ Object find(String value)
+ int getSize()
+ br.usp.semanticagent.util.GenList
- int length
- ListNode first
- ListNode currNode
- GenList parent
+ void GenList()
+ void insert(Object data)
- void insert(ListNode no)
+ boolean delete(ListNode no)
+ int getSize()
+ Object elementAt(int pos)
+ Object find(String value)
+ ListNode getCurrNode()
+ ListNode moveNext()
+ void moveFirst()
+ GenList getParent()
+ void setParent(GenList parent)
+ void print()
+ br.usp.semanticagent.util.VectorList
- int length
- Vector dataset
- VectorList parent
+ void VectorList()
+ void insert(Object data)
+ int getSize()
+ Object elementAt(int pos)
+ Object find(String value)
+ VectorList getParent()
+ void setParent(VectorList parent)
+ void print()
<<interface>>
+ br.usp.semanticagent.util.ListData
+ String getID()
+ Object getValue()
+ boolean isEqual(ListData value)
+ boolean isGreater(ListData value)
+ boolean isSmaller(ListData value)
+ br.usp.semanticagent.util.Int
int value
+ void Int()
+ void Int(int value)
+ int getValue()
+ void setValue(int value)
+ br.usp.semanticagent.util.ListNode
+ ListData data
+ ListNode next
+ GenList children
+ void ListNode()
+ br.usp.semanticagent.util.PrefReader
- String amsclient
- String tt_host
- String filename
- String ozoneServer
- boolean load_tt_ontology
- boolean restore_tt_ontology
- boolean findvalue
- boolean buildST
- boolean restoreComponentsDB
+ void PrefReader()
+ void PrefReader(String cfgfile)
- void ReadXMLConfig()
+ String getAmsClient()
+ String getTTHost()
+ String getOzoneServer()
+ boolean isLoad_tt_ontology()
+ boolean isRestore_tt_ontology()
+ void setRestore_tt_ontology(boolean restore_tt_ontology)
+ boolean isBuildST()
+ void setBuildST(boolean buildST)
+ boolean isRestoreComponentsDB()
+ br.usp.semanticagent.util.Number
String unit
float value
+ void Number()
+ void Number(Vector number)
+ void Number(float value, String unit)
+ String toString()
+ String getUnit()
+ float getValue()
+ void setUnit(String unit)
+ void setValue(float value)
+ br.usp.semanticagent.util.Boolean
boolean value
+ void Boolean()
+ void Boolean(boolean value)
+ boolean getValue()
+ void setValue(boolean value)
+ String toString()
+ br.usp.semanticagent.util.XTest
+ $ void main(String[] args)

AMS AGENT

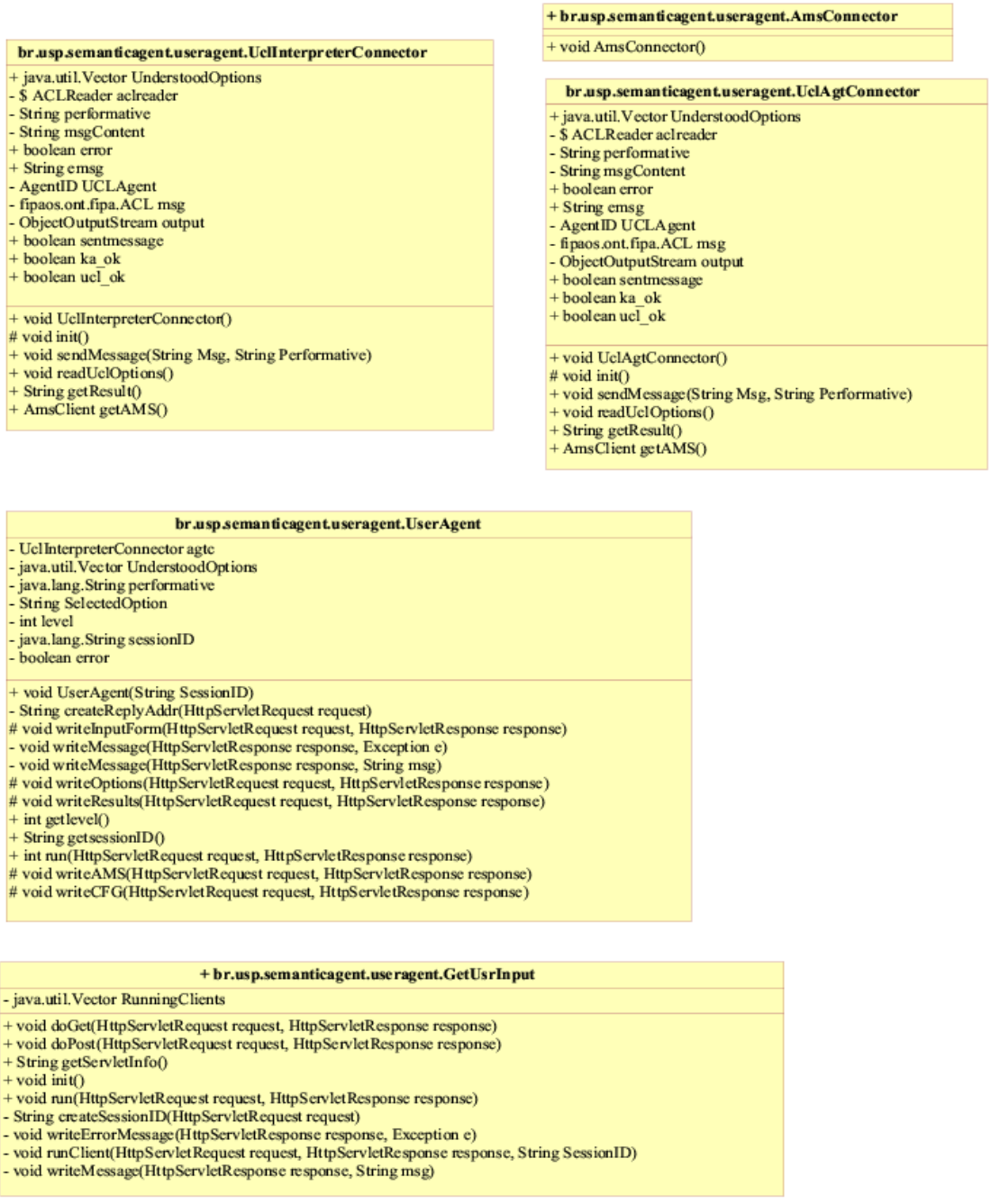
Agente responsável por gerenciar os endereços dos agentes registrados na plataforma. Provê serviço de diretório de endereços (white pages). Incluí os pacotes:

br.usp.semanticagent.ams


+ br.usp.semanticagent.ams.AmsManager
+ Vector AgentsDB
- boolean running
- int port
+ void AmsManager()
+ br.usp.semanticagent.ams.TestIt
+ $ void main(String[] args)
+ br.usp.semanticagent.ams.LogReader
- Vector agents
- String filename
+ void LogReader()
+ void LogReader(String cfgfile)
- void ReadXMLConfig()
+ Vector getAgents()
+ br.usp.semanticagent.ams.XTest
+ $ void main(String[] args)
+ void XTest()
+ br.usp.semanticagent.ams.AMS
- java.util.Vector AgentDB
- PrintWriter output
+ void AMS(Socket socket, Vector AgentsDB, AgentID agentid, PrintWriter LogFile)
- int manageRequest()
- void broadcast(ACL Agent)
- void disRegister(AgentID agent)
- AgentID getAddr(ACL msg)
- void register(AgentID agent)
+ void list(AgentID agent)

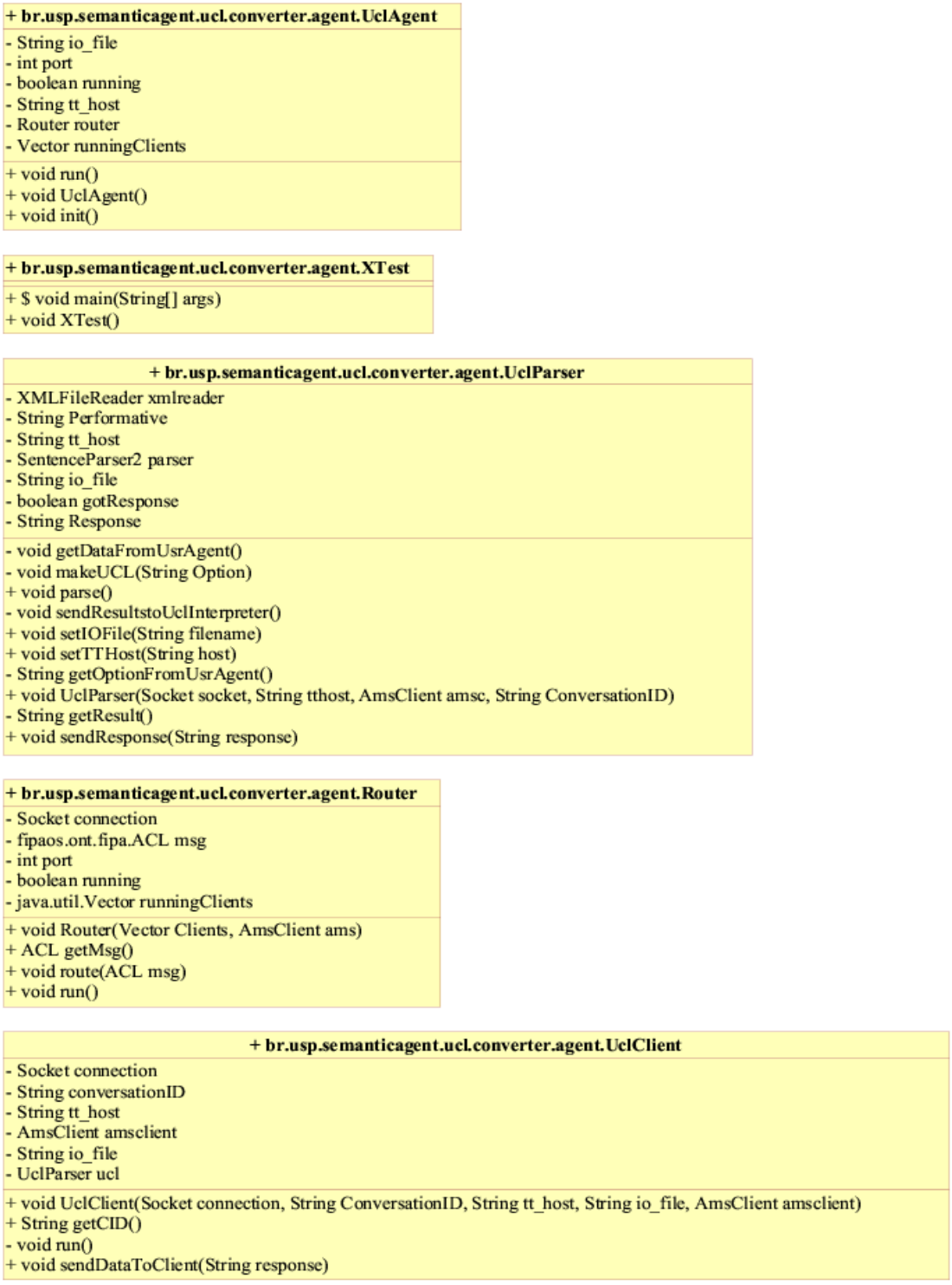
Agente responsável por gerenciar as conexões do usuário com o sistema. Implementado através de um servlet. Incluí os pacotes:

br.usp.semanticagent.useragent

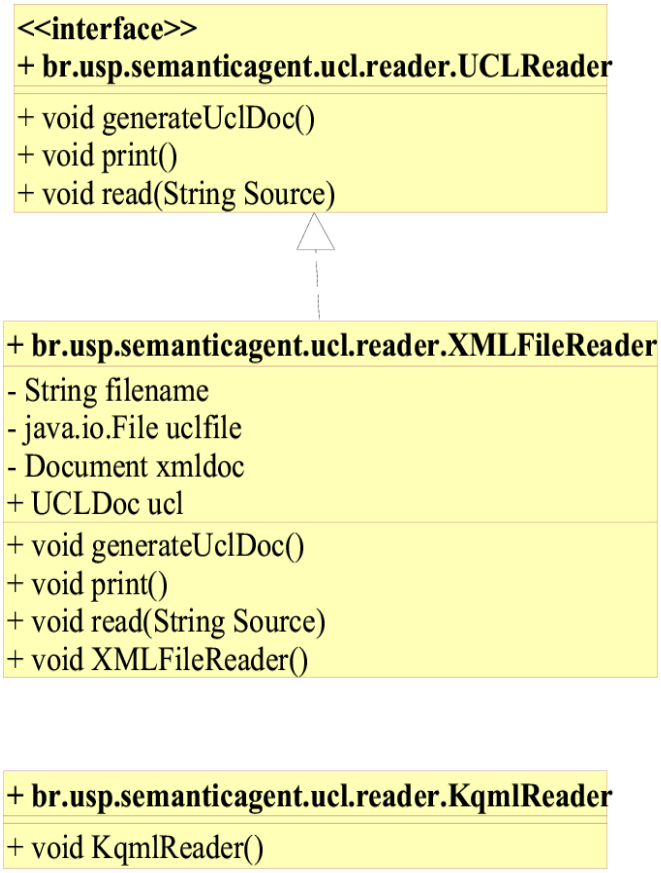

br.usp.semanticagent.useragent.UclInterpreterConnector
+ java.util.Vector UnderstoodOptions
- $ ACLReader aclreader
- String performative
- String msgContent
+ boolean error
+ String emsg
- AgentID UCLAgent
- fipaos.ont.fipa.ACL msg
- ObjectOutputStream output
+ boolean sentmessage
+ boolean ka_ok
+ boolean ucl_ok
+ void UclInterpreterConnector()
# void init()
+ void sendMessage(String Msg, String Performative)
+ void readUclOptions()
+ String getResult()
+ AmsClient getAMS()
+ br.usp.semanticagent.useragent.AmsConnector
+ void AmsConnector()
br.usp.semanticagent.useragent.UclAgtConnector
+ java.util.Vector UnderstoodOptions
- $ ACLReader aclreader
- String performative
- String msgContent
+ boolean error
+ String emsg
- AgentID UCLAgent
- fipaos.ont.fipa.ACL msg
- ObjectOutputStream output
+ boolean sentmessage
+ boolean ka_ok
+ boolean ucl_ok
+ void UclAgtConnector()
# void init()
+ void sendMessage(String Msg, String Performative)
+ void readUclOptions()
+ String getResult()
+ AmsClient getAMS()
br.usp.semanticagent.useragent.UserAgent
- UclInterpreterConnector agtc
- java.util.Vector UnderstoodOptions
- java.lang.String performative
- String SelectedOption
- int level
- java.lang.String sessionID
- boolean error
+ void UserAgent(String SessionID)
- String createReplyAddr(HttpServletRequest request)
void writeInputForm(HttpServletRequest request, HttpServletResponse response)
- void writeMessage(HttpServletResponse response, Exception e)
- void writeMessage(HttpServletResponse response, String msg)
void writeOptions(HttpServletRequest request, HttpServletResponse response)
void writeResults(HttpServletRequest request, HttpServletResponse response)
+ int getlevel()
+ String getsessionID()
+ int run(HttpServletRequest request, HttpServletResponse response)
# void writeAMS(HttpServletRequest request, HttpServletResponse response)
# void writeCFG(HttpServletRequest request, HttpServletResponse response)
+ br.usp.semanticagent.useragent.GetUsrInput
- java.util.Vector RunningClients
+ void doGet(HttpServletRequest request, HttpServletResponse response)
+ void doPost(HttpServletRequest request, HttpServletResponse response)
+ String getServletInfo()
+ void init()
+ void run(HttpServletRequest request, HttpServletResponse response)
- String createSessionID(HttpServletRequest request)
- void writeErrorMessage(HttpServletResponse response, Exception e)
- void runClient(HttpServletRequest request, HttpServletResponse response, String SessionID)
- void writeMessage(HttpServletResponse response, String msg)

Agente responsável por transformar as requisições de linguagem natural para UCL e vice-versa. Incluí os pacotes:

br.usp.semanticagent.ucl.agent
br.usp.semanticagent.ucl.parser
br.usp.semanticagent.ucl.reader

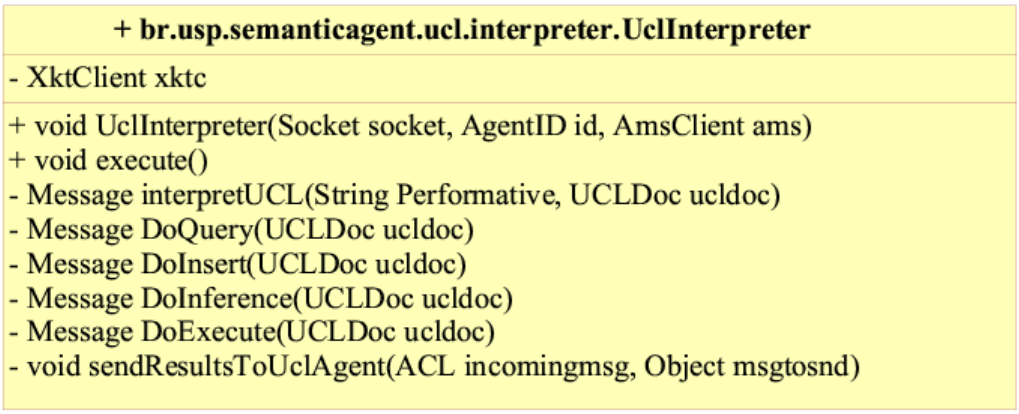

+ br.usp.semanticagent.ucl.converter.agent.UclAgent
- String io_file
- int port
- boolean running
- String tt_host
- Router router
- Vector runningClients
+ void run()
+ void UclAgent()
+ void init()
+ br.usp.semanticagent.ucl.converter.agent.XTest
+ $ void main(String[] args)
+ void XTest()
+ br.usp.semanticagent.ucl.converter.agent.UclParser
- XMLFileReader xmlreader
- String Performative
- String tt_host
- SentenceParser2 parser
- String io_file
- boolean gotResponse
- String Response
- void getDataFromUsrAgent()
- void makeUCL(String Option)
+ void parse()
- void sendResultstoUclInterpreter()
+ void setIOFile(String filename)
+ void setTTHost(String host)
- String getOptionFromUsrAgent()
+ void UclParser(Socket socket, String tthost, AmsClient amsc, String ConversationID)
- String getResult()
+ void sendResponse(String response)
+ br.usp.semanticagent.ucl.converter.agent.Router
- Socket connection
- fipaos.ont.fipa.ACL msg
- int port
- boolean running
- java.util.Vector runningClients
+ void Router(Vector Clients, AmsClient ams)
+ ACL getMsg()
+ void route(ACL msg)
+ void run()
+ br.usp.semanticagent.ucl.converter.agent.UclClient
- Socket connection
- String conversationID
- String tt_host
- AmsClient amsclient
- String io_file
- UclParser ucl
+ void UclClient(Socket connection, String ConversationID, String tt_host, String io_file, AmsClient amsclient)
+ String getCID()
- void run()
+ void sendDataToClient(String response)

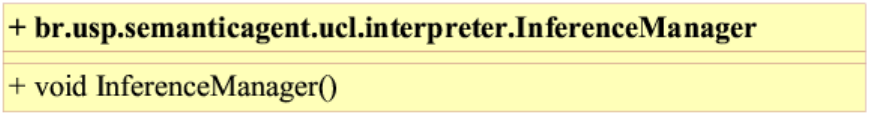

<<interface>>
br.usp.semanticagent.ucl.converter.parser.UclLanguage
$ String outputEncoding
$ String uw
$ String icl
$ String head
$ String id
$ String relation
$ String label
$ String id1
$ String id2
+ $ Vector v_relations
void closeSession()
Vector convertLNtoTT()
void convertTTtoUCL()
void convertTTtoUCLwrite()
String deconvertTTtoLN(String s_language)
String deconvertUCLtoTT()
String getFilenameoutput()
Element getRoot()
String getSentenceLN()
String getSentenceTT()
void init()
void setFilenameoutput(String s_name)
void setSentenceLN(String s_sentence)
void setSentenceTT(int i_op)
+ void takeAttOfConcept(int i_op)
Vector understood(String s_string)
Vector understood()
void convertedUCLwriter(String io)
+ br.usp.semanticagent.ucl.converter.parser.ConvertTTtoXML
$ String outputEncoding
$ String uw
$ String icl
$ String head
$ String id
$ String relation
$ String label
$ String id1
$ String id2
$ int Numid
int level
+ $ Vector v_relations
$ Document document
$ Document docrelations
$ java.lang.String ioFile
+ void addrelations(Element root, Element node)
+ String convertUcltoTT(Element node)
+ String inputSentence(TTConnection tt)
+ $ boolean isItIn(Object o_syntactic, String s_citationForm_features)
+ $ void main()
+ void output(Object o_parsesentence, Element node, Element noderelations, TTConnection tt)
+ void parse(String parsesentence, Element node, Element noderelations, TTConnection tt)
+ void ConvertTTtoXML()
+ void ConvertTTtoXML(String io)
+ br.usp.semanticagent.ucl.converter.parser.UclLanguageImpl
+ String cleanUp0(String s_param1)
+ void closeSession()
+ Vector convertLNtoTT()
+ void convertTTtoUCLaddrelations(Element root, Element node)
+ void convertTTtoUCLbuildDOM(Object o_parsesentence, Element node, Element noderelations, TTConnection tt)
+ void convertTTtoUCLparse(String parsesentence, Element node, Element noderelations, TTConnection tt)
+ void convertTTtoUCLparseOLD()
+ void convertTTtoUCL()
+ void convertTTtoUCLwrite()
+ String deconvertTTtoLN(String s_language)
+ void deconvertUCLtoTTbuildstring(Element root, Element o_node)
+ Element deconvertUCLtoTTfindElement(NodeList o_NodeListRel, String s_uwx)
+ String deconvertUCLtoTT()
+ String getFilenameoutput()
+ Element getRoot()
+ Element getRootRelation()
+ String getSentenceLN()
+ String getSentenceTT()
+ void ifQuestion(String s_param1)
+ void init()
+ $ boolean isItIn(Object o_syntactic, String s_citationForm_features)
+ void setFilenameoutput(String s_name)
+ void setSentenceLN(String s_sentence)
+ void setSentenceTT(int i_op)
+ void takeAttOfConcept(int i_op)
+ Vector understood(String s_string)
+ Vector understood()
+ void UclLanguageImpl(String host)
+ void UclLanguageImpl(String host, String io)
+ void convertedUCLwriter(String io)
+ br.usp.semanticagent.ucl.converter.parser.SentenceParser2
+ String performative
+ Vector Options
- String ttHost
- UclLanguage ucl_message
+ void SentenceParser2(String tthost)
+ void generateUCL(String option, String io_file)
+ void Understand(String sentence, String io_file)
+ br.usp.semanticagent.ucl.converter.parser.ConvertUCLtoTT
$ Document document
$ NodeList o_NodeListRel
$ NodeList o_NodeListUw
$ String s_stringtt
+ void convert(Element root, Element o_node)
+ Element findElement(NodeList o_NodeListRel, String s_uwx)
+ $ void main()
+ String run()
+ void ConvertUCLtoTT()
+ br.usp.semanticagent.ucl.converter.parser.Relations
+ $ void main(String[] argv)
void output(String name, String prefix, String line)
+ void parse(File f)
+ void Relations()

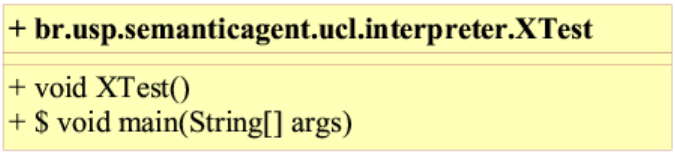

<<interface>>
+ br.usp.semanticagent.ucl.reader.UCLReader
+ void generateUclDoc()
+ void print()
+ void read(String Source)
+ br.usp.semanticagent.ucl.reader.XMLFileReader
- String filename
- java.io.File uclfile
- Document xmldoc
+ UCLDoc ucl
+ void generateUclDoc()
+ void print()
+ void read(String Source)
+ void XMLFileReader()
+ br.usp.semanticagent.ucl.reader.KqmlReader
+ void KqmlReader()

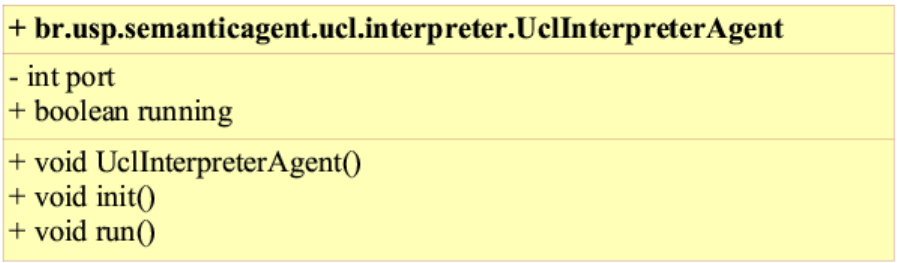
Agente responsável por gerenciar as requisções UCL recebidas pela plataforma. Incluí os pacotes:

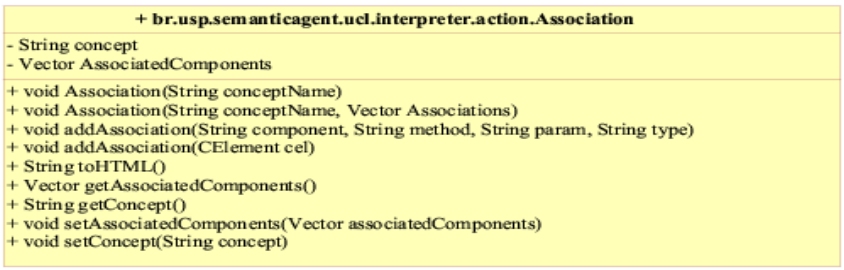
br.usp.semanticagent.ucl.interpreter
br.usp.semanticagent.ucl.interpreter.action
br.usp.semanticagent.ucl.scripts

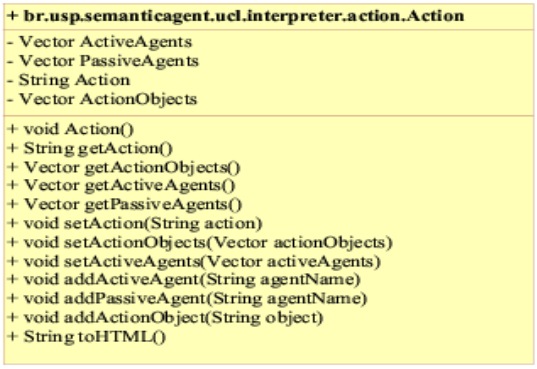

+ br.usp.semanticagent.ucl.interpreter.UclInterpreter
- XktClient xktc
+ void UclInterpreter(Socket socket, AgentID id, AmsClient ams)
+ void execute()
- Message interpretUCL(String Performative, UCLDoc ucldoc)
- Message DoQuery(UCLDoc ucldoc)
- Message DoInsert(UCLDoc ucldoc)
- Message DoInference(UCLDoc ucldoc)
- Message DoExecute(UCLDoc ucldoc)
- void sendResultsToUclAgent(ACL incomingmsg, Object msgtosnd)

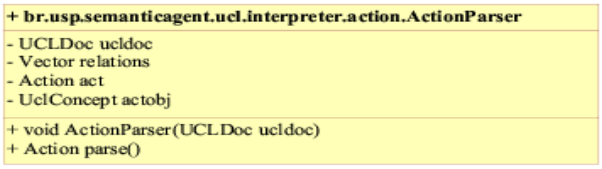

+ br.usp.semanticagent.ucl.interpreter.InferenceManager
+ void InferenceManager()

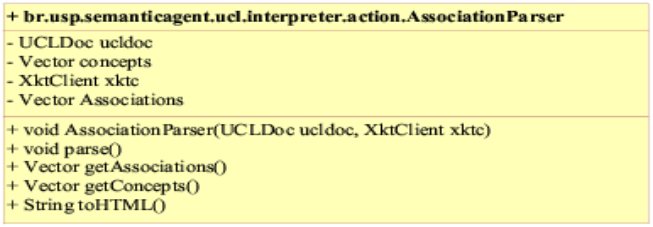

+ br.usp.semanticagent.ucl.interpreter.XTest
+ void XTest()
+ $ void main(String[] args)

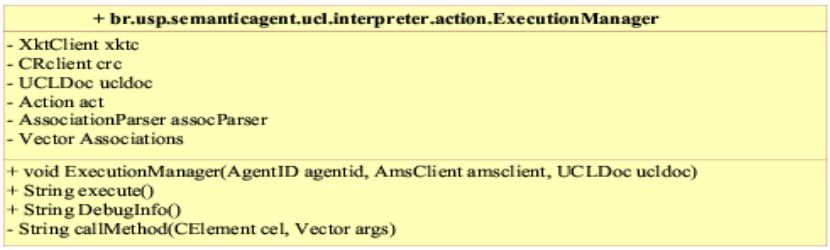

+ br.usp.semanticagent.ucl.interpreter.UclInterpreterAgent
- int port
+ boolean running
+ void UclInterpreterAgent()
+ void init()
+ void run()

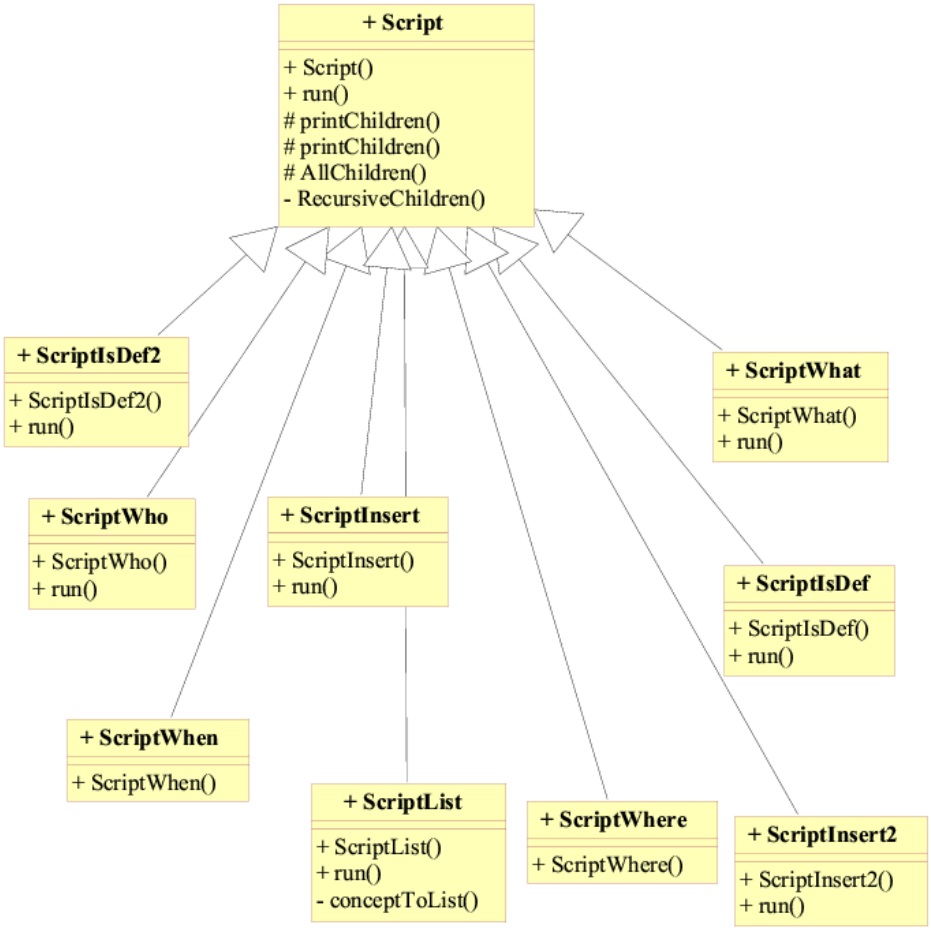

+ br.usp.semanticagent.ucl.interpreter.action.Association
- String concept
- Vector AssociatedComponents
+ void Association(String conceptName)
+ void Association(String conceptName, Vector Associations)
+ void addAssociation(String component, String method, String param, String type)
+ void addAssociation(CElement cel)
+ String toHTML()
+ Vector getAssociatedComponents()
+ String getConcept()
+ void setAssociatedComponents(Vector associatedComponents)
+ void setConcept(String concept)


+ br.usp.semanticagent.ucl.interpreter.action.Action
- Vector ActiveAgents
- Vector PassiveAgents
- String Action
- Vector ActionObjects
+ void Action()
+ String getAction()
+ Vector getActionObjects()
+ Vector getActiveAgents()
+ Vector getPassiveAgents()
+ void setAction(String action)
+ void setActionObjects(Vector actionObjects)
+ void setActiveAgents(Vector activeAgents)
+ void addActiveAgent(String agentName)
+ void addPassiveAgent(String agentName)
+ void addActionObject(String object)
+ String toHTML()


+ br.usp.semanticagent.ucl.interpreter.action.ActionParser
- UCLDoc ucldoc
- Vector relations
- Action act
- UclConcept actobj
+ void ActionParser(UCLDoc ucldoc)
+ Action parse()


+ br.usp.semanticagent.ucl.interpreter.action.AssociationParser
- UCLDoc ucldoc
- Vector concepts
- XktClient xktc
- Vector Associations
+ void AssociationParser(UCLDoc ucldoc, XktClient xktc)
+ void parse()
+ Vector getAssociations()
+ Vector getConcepts()
+ String toHTML()

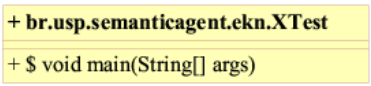

+ br.usp.semanticagent.ucl.interpreter.action.ExecutionManager
- XktClient xktc
- CRclient crc
- UCLDoc ucldoc
- Action act
- AssociationParser assocParser
- Vector Associations
+ void ExecutionManager(AgentID agentid, AmsClient amsclient, UCLDoc ucldoc)
+ String execute()
+ String DebugInfo()
- String callMethod(CElement cel, Vector args)

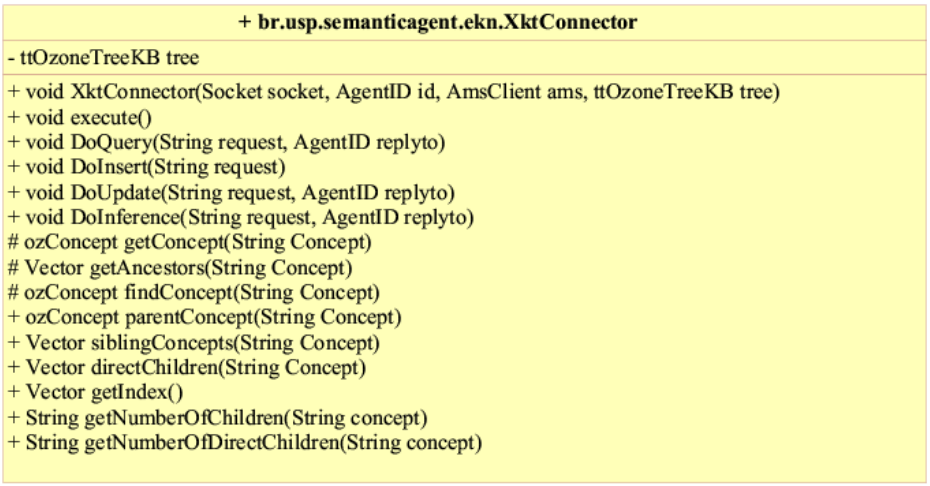

+ Script
+ Script()
+ run()
# printChildren()
# printChildren()
# AllChildren()
- RecursiveChildren()
+ ScriptIsDef2
+ ScriptIsDef2()
+ run()
+ ScriptWho
+ ScriptWho()
+ run()
+ ScriptInsert
+ ScriptInsert()
+ run()
+ ScriptWhen
+ ScriptWhen()
+ ScriptList
+ ScriptList()
+ run()
- conceptToList()
+ ScriptWhere
+ ScriptWhere()
+ ScriptInsert2
+ ScriptInsert2()
+ run()
+ ScriptIsDef
+ ScriptIsDef()
+ run()
+ ScriptWhat
+ ScriptWhat()
+ run()

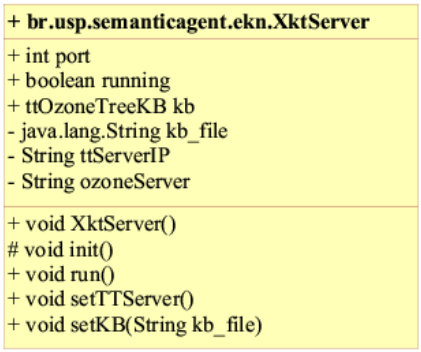
EKN AGENT

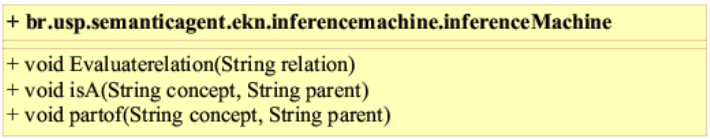
Agente responsável por representar e manipular o conhecimento do sistema através de redes semânticas extendidas (Extended Knowledge Networks). Incluí os pacotes:

br.usp.semanticagent.ekn
br.usp.semanticagent.ekn.inferencemachine
br.usp.semanticagent.ekn.ontology
br.usp.semanticagent.ekn.ozone


+ br.usp.semanticagent.ekn.XTest
+ $ void main(String[] args)


+ br.usp.semanticagent.ekn.XktConnector
- ttOzoneTreeKB tree
+ void XktConnector(Socket socket, AgentID id, AmsClient ams, ttOzoneTreeKB tree)
+ void execute()
+ void DoQuery(String request, AgentID replyto)
+ void DoInsert(String request)
+ void DoUpdate(String request, AgentID replyto)
+ void DoInference(String request, AgentID replyto)
# ozConcept getConcept(String Concept)
# Vector getAncestors(String Concept)
# ozConcept findConcept(String Concept)
+ ozConcept parentConcept(String Concept)
+ Vector siblingConcepts(String Concept)
+ Vector directChildren(String Concept)
+ Vector getIndex()
+ String getNumberOfChildren(String concept)
+ String getNumberOfDirectChildren(String concept)


+ br.usp.semanticagent.ekn.XktServer
+ int port
+ boolean running
+ ttOzoneTreeKB kb
- java.lang.String kb_file
- String ttServerIP
- String ozoneServer
+ void XktServer()
# void init()
+ void run()
+ void setTTServer()
+ void setKB(String kb_file)


+ br.usp.semanticagent.ekn.inferencemachine.inferenceMachine
+ void Evaluaterelation(String relation)
+ void isA(String concept, String parent)
+ void partof(String concept, String parent)

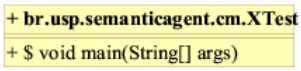

+ br.usp.semanticagent.ekn.ontology.tt.ttAssertion
- Vector TTassertions
- Vector Assertions
- String concept
- int size
+ void ttAssertion(String Concept)
+ Vector getAssertions()
+ Object getAssertion(int element)
+ int size()
- Vector getTTAssertion(String Concept)
- Vector parseAssertions()
+ int ttrawsize()
+ void printTTAssertions()
+ br.usp.semanticagent.ekn.ontology.tt.XTest
+ $ void main(String[] args)
<<interface>>
+ br.usp.semanticagent.ekn.ontology.daml.damlParser
+ br.usp.semanticagent.ekn.ontology.tt.ttMemTreeKB
- TTConnection tt
- mtKB ttmtKB
- String ttServer_ip
+ void ttMemTreeKB(String ttServer_ip)
+ int numberOfConcepts()
+ int numberOfAssertions()
+ void Load()
- void LoadChildren(String conceptName, Concept parent)
+ mtConcept findConcept(String Concept)
+ Vector findSiblings(String Concept)
+ mtConcept findParent(String sconcept)
+ Vector findDirectChildren(String sconcept)
+ void insertConcept(String concept, String parent)
+ Boolean isA(String concept, String ancestor)
+ br.usp.semanticagent.ekn.ontology.tt.ttAssertionParser
- Vector ttassertion
- String concept
- ozAssertion assertion
+ void ttAssertionParser(Vector TTassertion, String Concept)
- void parseAssertion()
- ozAssertion isAssertion()
- ozAssertion atribAssertion()
- ozAssertion relatAssertion()
+ String getConcept()
+ void setConcept(String concept)
+ ozAssertion getAssertion()
+ void setAssertion(ozAssertion assertion)
+ br.usp.semanticagent.ekn.ontology.tt.ttOzoneTreeKB
+ TTConnection tt
- String ttServer_ip
- String ozoneServer_ip
- ozKB ttKB
+ void ttOzoneTreeKB(String ttServer_ip, String ozoneServer_ip)
+ int numberOfConcepts()
+ int numberOfAssertions()
+ void Load()
+ void Restore()
- void LoadChildren(String conceptName, ozConcept parent)
+ ozConcept findConcept(String Concept)
+ ozConcept getConcept(String Concept)
+ Vector findSiblings(String Concept)
+ Vector getAncestors(String Concept)
+ ozConcept findParent(String sconcept)
+ Vector findDirectChildren(String sconcept)
+ Vector getIndex()
+ void insertConcept(String concept, String parent)
+ void delete(String concept)
+ Boolean isA(String concept, String ancestor)
+ int getNumberOfChildren(String concept)
+ int computeNumberOfChildren(String concept)
+ int getNumberOfDirectChildren(String concept)
+ void renameConcept(String Concept, String NewName)
+ void Associate(String concept, String bean, String method, String param, int nparam)
+ void Dissociate(String concept, String bean, String method, String param, int nparam)
+ void buildTT()

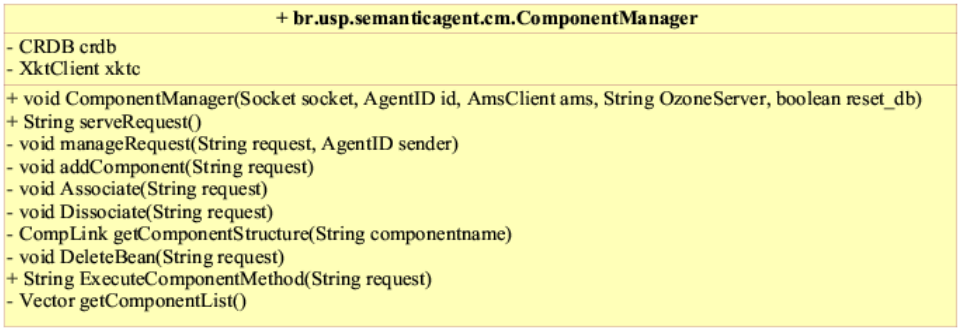

<<interface>>
+ br.usp.semanticagent.ekn.kb.ozone.ozCategories
+ Vector getCategories()
+ void addCategory(ozConcept concept)
+ int getNumberOfConcepts()
+ void setNumberOfConcepts(int nc)
+ int getNumberOfAssertions()
+ void setNumberOfAssertions(int na)
+ int makeOID()
+ void adjustCount()
+ void adjustAssertions(int noa)
+ br.usp.semanticagent.ekn.kb.ozone.ozCategoriesImpl_Proxy
$ long serialVersionUID
+ void ozCategoriesImpl_Proxy()
+ void ozCategoriesImpl_Proxy(ObjectID oid, OzoneInterface link)
+ $ ozCategories createObject(OzoneInterface link)
+ Vector getCategories()
+ void addCategory(ozConcept arg0)
+ int getNumberOfAssertions()
+ int getNumberOfConcepts()
+ void setNumberOfAssertions(int arg0)
+ void setNumberOfConcepts(int arg0)
+ int makeOID()
+ void adjustCount()
+ void adjustAssertions(int arg0)
+ ObjectID getObjectID()
+ br.usp.semanticagent.ekn.kb.ozone.ozCategoriesImpl
+ Vector index
- int numberOfConcepts
- int numberOfAssertions
- int OID
+ void ozCategoriesImpl()
+ Vector getCategories()
+ void addCategory(ozConcept concept)
+ int getNumberOfAssertions()
+ int getNumberOfConcepts()
+ void setNumberOfAssertions(int numberOfAssertions)
+ void setNumberOfConcepts(int numberOfConcepts)
+ int makeOID()
+ void adjustCount()
+ void adjustAssertions(int noa)
+ br.usp.semanticagent.ekn.kb.ozone.ozXTest
+ $ void main(String[] args)
+ br.usp.semanticagent.ekn.kb.ozone.ozKB
ozOODB oz
+ void ozKB(String server)
+ ozConcept insertRoot(ozConcept sConcept)
+ ozConcept insert(ozConcept concept, ozConcept parent)
+ void delete(ozConcept concept)
+ void print()
+ int getNumberOfConcepts()
+ int getNumberOfAssertions()
+ ozConcept FindConcept(ozConcept sConcept)
+ ozConcept getConcept(ozConcept sConcept)
+ ozConcept getOzConcept(ozConcept sConcept)
+ void renameConcept(ozConcept concept, ozConcept newname)
+ ozConcept FindParent(ozConcept ctf)
+ Vector FindDirectChildren(ozConcept ctf)
+ Vector getAncestors(ozConcept ctf)
+ Vector FindSiblings(ozConcept ctf)
+ Boolean isA(ozConcept con, ozConcept Ancestor)
+ int getNumberofChildren(ozConcept ctf)
+ void Associate(CElement cel)
+ void Dissociate(CElement cel)
+ int computeNumberofChildren(ozConcept ctf)
- int getNumberChildren(ozConcept ctf)
+ int getNumberOfDirectChildren(ozConcept ctf)
+ Vector getIndex()
+ void BTT()
- void btt(ozConcept cpt, int sLevel)
+ br.usp.semanticagent.ekn.kb.ozone.ozOODB
- ExternalDatabase db
- ozCategories ConceptRootIndex
- int noc
- int noa
- PrintWriter output
- String msg
+ void ozOODB(String server)
+ void close()
+ ozConcept addConcept(String id, String name)
+ ozConcept addConcept(ozConcept concept)
+ ozConcept addConcept(ozConcept concept, ozConcept parent)
+ void printConcept(String id)
+ void deleteConcept(String id)
+ ozConcept getConcept(String conceptName)
+ void addCategory(ozConcept category)
+ Vector getCategories()
+ void printCategories()
- void Statistics()
- void statElems(ozConcept cpt, int sLevel)
+ void print()
- void printCat(ozConcept cpt, int sLevel)
+ int getNumberOfConcepts()
+ int getNumberOfAssertions()
+ void renameConcept(ozConcept concept, ozConcept newname)
+ void delete(ozConcept concept, ozConcept parent)
+ ozConcept getConceptWith(ozConcept cpt)
- ozConcept searchInnerChild(ozConcept cpt, String children)
- void log(String value)

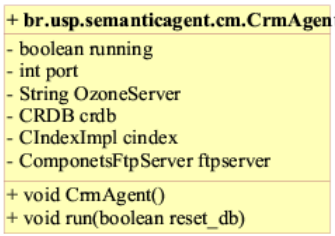
COMPONENT MANAGER AGENT

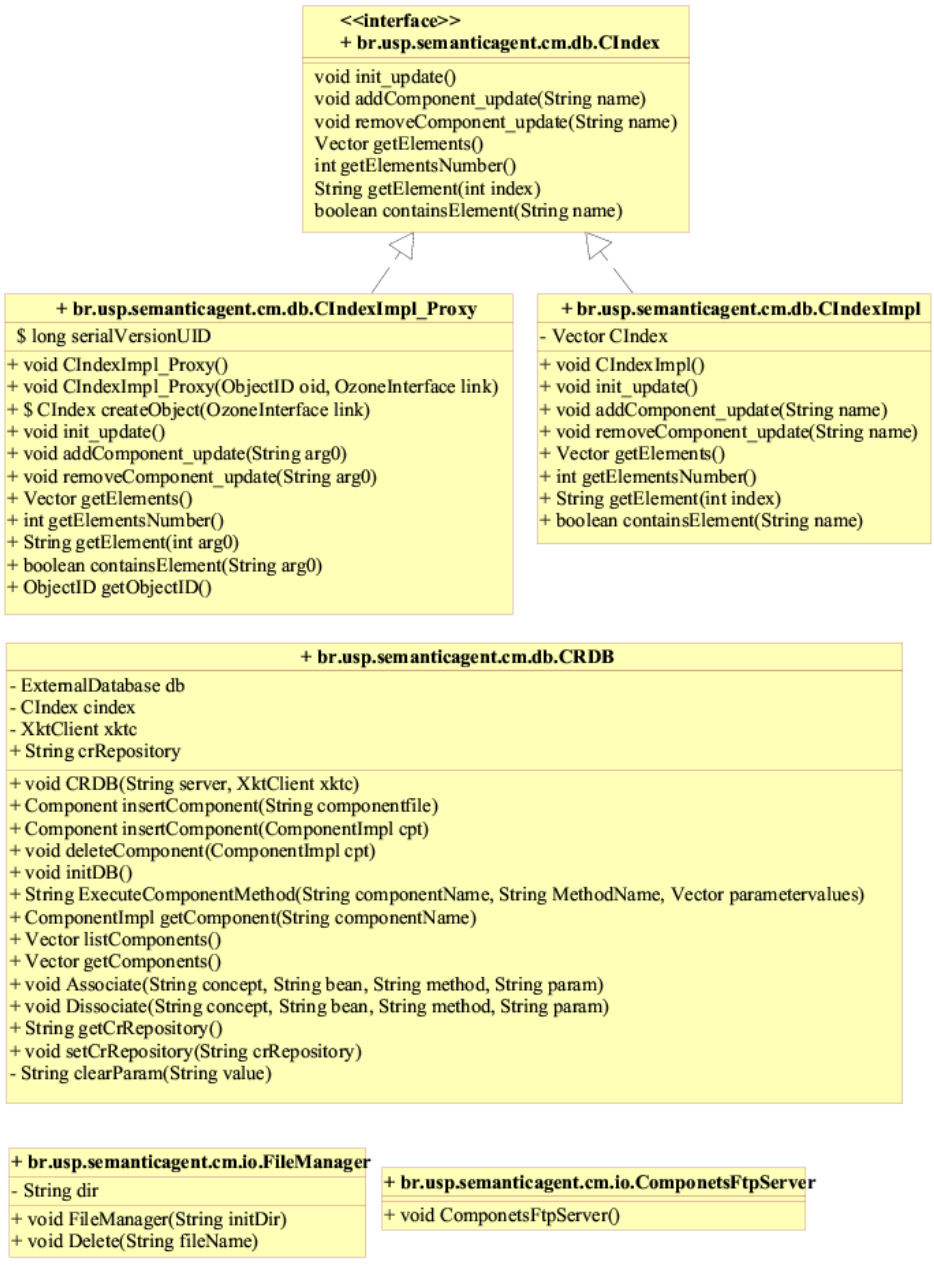
Agente responsável por gerenciar os comportamentos dos agentes atômicos do sistema, implementados através de componentes Javabeans. Incluí os pacotes:

br.usp.semanticagent.cm
br.usp.semanticagent.cm.io


+ br.usp.semanticagent.cm.XTest
+ $ void main(String[] args)


+ br.usp.semanticagent.cm.ComponentManager
- CRDB crdb
- XktClient xktc
+ void ComponentManager(Socket socket, AgentID id, AmsClient ams, String OzoneServer, boolean reset_db)
+ String serveRequest()
- void manageRequest(String request, AgentID sender)
- void addComponent(String request)
- void Associate(String request)
- void Dissociate(String request)
- CompLink getComponentStructure(String componentname)
- void DeleteBean(String request)
+ String ExecuteComponentMethod(String request)
- Vector getComponentList()

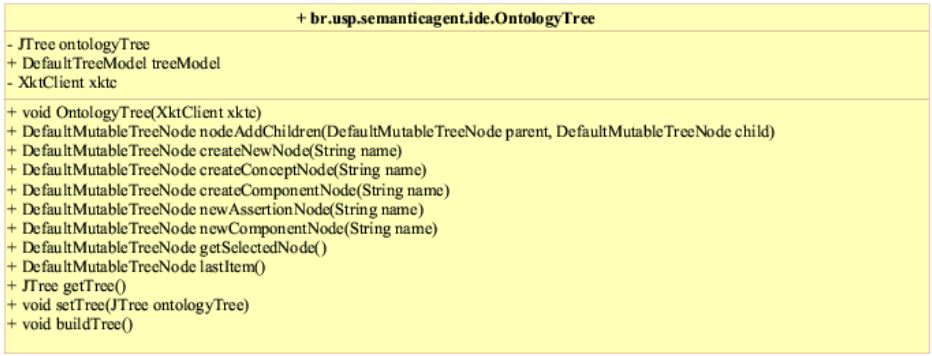

+ br.usp.semanticagent.cm.CrmAgent
- boolean running
- int port
- String OzoneServer
- CRDB crdb
- CIndexImpl cindex
- ComponetsFtpServer ftpserver
+ void CrmAgent()
+ void run(boolean reset_db)

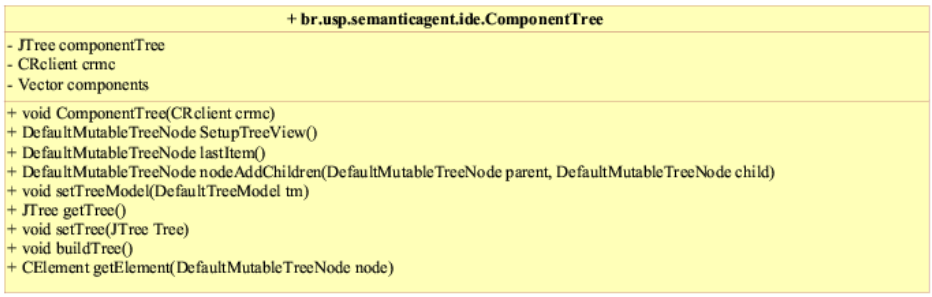

<<interface>>
+ br.usp.semanticagent.cm.db.CIndex
void init_update()
void addComponent_update(String name)
void removeComponent_update(String name)
Vector getElements()
int getElementsNumber()
String getElement(int index)
boolean containsElement(String name)
+ br.usp.semanticagent.cm.db.CIndexImpl_Proxy
$ long serialVersionUID
+ void CIndexImpl_Proxy()
+ void CIndexImpl_Proxy(ObjectID oid, OzoneInterface link)
+ $ CIndex createObject(OzoneInterface link)
+ void init_update()
+ void addComponent_update(String arg0)
+ void removeComponent_update(String arg0)
+ Vector getElements()
+ int getElementsNumber()
+ String getElement(int arg0)
+ boolean containsElement(String arg0)
+ ObjectID getObjectID()
+ br.usp.semanticagent.cm.db.CIndexImpl
- Vector CIndex
+ void CIndexImpl()
+ void init_update()
+ void addComponent_update(String name)
+ void removeComponent_update(String name)
+ Vector getElements()
+ int getElementsNumber()
+ String getElement(int index)
+ boolean containsElement(String name)
+ br.usp.semanticagent.cm.db.CRDB
- ExternalDatabase db
- CIndex cindex
- XktClient xktc
+ String crRepository
+ void CRDB(String server, XktClient xktc)
+ Component insertComponent(String componentfile)
+ Component insertComponent(ComponentImpl cpt)
+ void deleteComponent(ComponentImpl cpt)
+ void initDB()
+ String ExecuteComponentMethod(String componentName, String MethodName, Vector parametervalues)
+ ComponentImpl getComponent(String componentName)
+ Vector listComponents()
+ Vector getComponents()
+ void Associate(String concept, String bean, String method, String param)
+ void Dissociate(String concept, String bean, String method, String param)
+ String getCrRepository()
+ void setCrRepository(String crRepository)
- String clearParam(String value)
+ br.usp.semanticagent.cm.io.FileManager
- String dir
+ void FileManager(String initDir)
+ void Delete(String fileName)
+ br.usp.semanticagent.cm.io.ComponetsFtpServer
+ void ComponetsFtpServer()

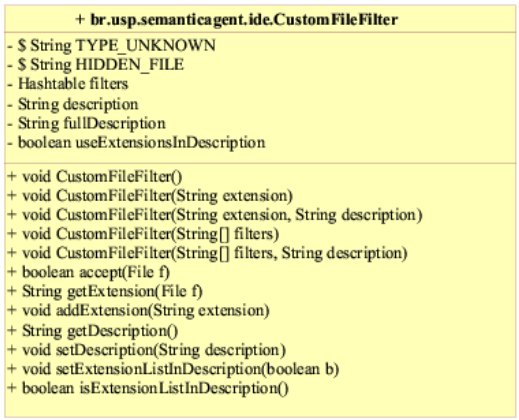
A Semantic Agent IDE permite desenvolver aplicações baseadas no Semantic Agent Application Server. A IDE incluí os pacotes:

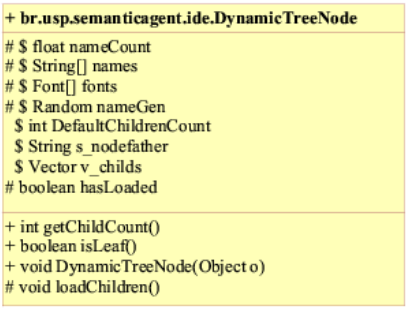
br.usp.semanticagent.ide

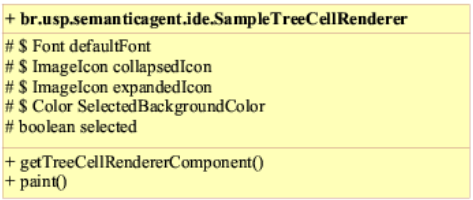

InsertConceptAction
+ void actionPerformed(ActionEvent e)
ListCompAction
+ void actionPerformed(ActionEvent e)
DeleteComponent
+ void actionPerformed(ActionEvent e)
ImportComp
+ void actionPerformed(ActionEvent e)
AssociateComp
+ void actionPerformed(ActionEvent e)
SideBarCancelAction
+ void actionPerformed(ActionEvent e)
+ br.usp.semanticagent.ide.IDEform
# JFrame frame
# OntologyTree tree
# ComponentTree ctree
# DefaultMutableTreeNode lastItem
# int newIndex
# int insertCount
# JScrollPane MLeftBar
# JScrollPane sp
# JPanel panel
# JPanel retPanel
# JPanel borderPane
# JScrollPane Footer
# JPanel Sidebar
# JLabel labelp
# JScrollPane sb
# JScrollPane sb2
# JScrollPane sb3
# JPanel LeftPanel
# JPanel pn
# JPanel pn2
# JPanel pn3
# JButton connectButton
# JButton disconnectButton
# JList astBox
# JTextField host
# JTextArea debug
# JMenuBar menuBar
# JTable jTable1
# JTextField conceptName
# String Parent
# String Concept
+ $ XktClient xktc
+ AmsClient ams
+ CRclient crnc
- JMenuBar constructMenuBar()
- JPanel constructOptionsPanel()
+ void SideBar(String Label, String Action, String Parent, String Concept)
+ void SideBar2(String Concept)
# void BoxAddAssertions(String ConceptName)
+ void ComponentsBar()
+ void clearSideBars()
+ void renderOntologyTree()
+ $ void main(String[] args)
+ void IDEform()
EditConceptAction
+ void actionPerformed(ActionEvent e)
DissociateComponent
+ void actionPerformed(ActionEvent e)
InsertAction
+ void actionPerformed(ActionEvent e)
FindAction
+ void actionPerformed(ActionEvent e)
RemoveConceptAssociation
+ void actionPerformed(ActionEvent e)
RemoveAction
+ void actionPerformed(ActionEvent e)
ConnectAction
+ void actionPerformed(ActionEvent e)
FindConceptAction
+ void actionPerformed(ActionEvent e)
EditAction
+ void actionPerformed(ActionEvent e)
RunComponent
+ void actionPerformed(ActionEvent e)
SubItems
+ void actionPerformed(ActionEvent e)


+ br.usp.semanticagent.ide.OntologyTree
- JTree ontologyTree
+ DefaultTreeModel treeModel
- XktClient xktc
+ void OntologyTree(XktClient xktc)
+ DefaultMutableTreeNode nodeAddChildren(DefaultMutableTreeNode parent, DefaultMutableTreeNode child)
+ DefaultMutableTreeNode createNewNode(String name)
+ DefaultMutableTreeNode createConceptNode(String name)
+ DefaultMutableTreeNode createComponentNode(String name)
+ DefaultMutableTreeNode newAssertionNode(String name)
+ DefaultMutableTreeNode newComponentNode(String name)
+ DefaultMutableTreeNode getSelectedNode()
+ DefaultMutableTreeNode lastItem()
+ JTree getTree()
+ void setTree(JTree ontologyTree)
+ void buildTree()

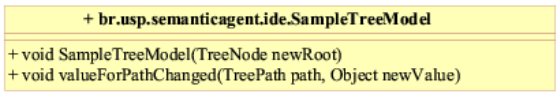

+ br.usp.semanticagent.ide.ComponentTree
- JTree componentTree
- CRclient crmc
- Vector components
+ void ComponentTree(CRclient crmc)
+ DefaultMutableTreeNode SetupTreeView()
+ DefaultMutableTreeNode lastItem()
+ DefaultMutableTreeNode nodeAddChildren(DefaultMutableTreeNode parent, DefaultMutableTreeNode child)
+ void setTreeModel(DefaultTreeModel tm)
+ JTree getTree()
+ void setTree(JTree Tree)
+ void buildTree()
+ CElement getElement(DefaultMutableTreeNode node)


+ br.usp.semanticagent.ide.CustomFileFilter
- $ String TYPE_UNKNOWN
- $ String HIDDEN_FILE
- Hashtable filters
- String description
- String fullDescription
- boolean useExtensionsInDescription
+ void CustomFileFilter()
+ void CustomFileFilter(String extension)
+ void CustomFileFilter(String extension, String description)
+ void CustomFileFilter(String[] filters)
+ void CustomFileFilter(String[] filters, String description)
+ boolean accept(File f)
+ String getExtension(File f)
+ void addExtension(String extension)
+ String getDescription()
+ void setDescription(String description)
+ void setExtensionListInDescription(boolean b)
+ boolean isExtensionListInDescription()


+ br.usp.semanticagent.ide.DynamicTreeNode
# $ float nameCount
# $ String[] names
# $ Font[] fonts
# $ Random nameGen
$ int DefaultChildrenCount
$ String s_nodefather
$ Vector v_childs
# boolean hasLoaded
+ int getChildCount()
+ boolean isLeaf()
+ void DynamicTreeNode(Object o)
# void loadChildren()


+ br.usp.semanticagent.ide.SampleTreeCellRenderer
# $ Font defaultFont
# $ ImageIcon collapsedIcon
# $ ImageIcon expandedIcon
# $ Color SelectedBackgroundColor
# boolean selected
+ getTreeCellRendererComponent()
+ paint()


+ br.usp.semanticagent.ide.NodeData
# Font font
# Color color
# String string
- String toolTipText
- String Type
- Icon icon
+ Color getColor()
+ Font getFont()
+ void NodeData(Font newFont, Color newColor, String newString)
+ void NodeData(Font newFont, Color newColor, String newString, String toolTipText)
+ void NodeData(Font newFont, Color newColor, String newString, String toolTipText, String Type)
+ void NodeData(Color newColor, String newString, String toolTipText)
+ void NodeData(String data, String Type)
+ void setColor(Color newColor)
+ void setFont(Font newFont)
+ void setString(String newString)
+ String string()
+ String toString()
+ String getToolTipText()
+ void setToolTipText(String toolTipText)
+ String getValue()
+ String getType()
+ void setType(String type)
+ Icon getIcon()
+ void setIcon(Icon icon)
+ void ConceptNodeLookAndFeel(boolean triger)
+ void AssertionNodeLookAndFeel(boolean triger)
+ void ComponentNodeLookAndFeel(boolean triger)


+ br.usp.semanticagent.ide.SampleTreeModel
+ void SampleTreeModel(TreeNode newRoot)
+ void valueForPathChanged(TreePath path, Object newValue)