

# **Desenvolvendo Sistemas Multi-Agentes utilizando Tropos e JADEX**

**Aluna: *Bárbara Siqueira Santos***

**Orientador: *Jaelson Freire Brelaz de Castro***

**Trabalho de Graduação**



Universidade Federal de Pernambuco

RECIFE, AGOSTO/2005

## **Assinaturas**

Este Trabalho de Graduação é resultado dos esforços da aluna Bárbara Siqueira Santos, sob a orientação do professor Jaelson Freire Brelaz e Castro, na participação projeto “Desenvolvendo Sistemas Multi-Agentes utilizando Tropos e JADEX”, conduzido pelo Centro de Informática da Universidade Federal de Pernambuco. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

---

**Bárbara Siqueira Santos**

---

**Jaelson Freire Brelaz de Castro**

# Agradecimentos

Meus sinceros agradecimentos a todos, que direta ou indiretamente, contribuíram para a conclusão deste trabalho e de uma importante fase da minha vida – a graduação. Aqui começa uma nova etapa com muito entusiasmo, perspectivas, esperanças, projetos e sonhos.

Agradeço aos meus pais, Romero Almeida e Marisete Siqueira, minhas irmãs Juliana e Débora Siqueira pelo apoio e compreensão em todos os momentos desta jornada. E também à minha linda e inteligente sobrinha Camila, que sempre me faz rir seja qual for a hora. Amo muito vocês Família!

Ao meu namorado Eduardo Vinicius, que sempre esteve ao meu lado durante, esses, quase cinco anos de graduação, me apoiando, com atenção, compreensão, amizade, respeito, carinho, amor. Obrigada!

Em especial agradeço ao meu orientador Prof. Jaelson Freire, pela confiança e receptividade. As doutorandas Ismênia Galvão e Carla Taciana, pela paciência e disponibilidade.

Agradeço a todos os meus amigos, que me incentivaram e são amigos pra vida toda, obrigada, tia ixx, Carol, Bruno Celso, Daniel Thiago, Dedeco, Diego Ribeiro, Gandhi, Heitor Vital, Hermano Patinadora, Jarbinhas, tia Jennifer, Jujuba, Magão, Marcílio Filho, Mari e Marina, Marquinhos, Rangner, Rodrigo Mendes, Victor Medeiros, Zumbi, e tantos outros que me apoiaram. Obrigada!

# Resumo

Este trabalho de graduação se propõe a investigar a questão do desenvolvimento de sistemas para o paradigma de orientação a agentes. Em particular, estaremos preocupados em estudar o desenvolvimento de sistemas multi-agentes, do projeto à implementação. Para isso, investigaremos em especial a proposta Tropos. Também, estudaremos a implementação destes sistemas fazendo-se uso da plataforma de desenvolvimento orientado a agentes – JADEX, sendo esta uma extensão da plataforma JADE.

A partir desse estudo, utilizaremos um problema do mundo real como estudo de caso, realizando assim, sua modelagem segundo a proposta da metodologia Tropos e seus artefatos, e a plataforma JADEX para propor a implementação do sistema.

**Palavras-chave:** Engenharia de Software Orientada a Agentes, Sistemas Multi-Agentes, Metodologias de Desenvolvimento de Software Orientados a Agentes, Tropos, JADEX.

# **Abstract**

This work considers investigating the question of the systems development focused on the agent-oriented paradigm. Particularly, it will be concerned about studying the development of multi-agents systems, from the design to implementation. Therefore, it will be investigated, in special, the Tropos proposal. Also, it will investigate the implementation of those systems by using the agent-oriented development platform – JADEX, which is an extension of the JADE platform.

Through this study, a real world problem will be used as a case study, thus carrying through, its modeling according to the Tropos proposal and its devices, and the use of the JADEX platform to propose the implementation of the system.

# Sumário

<b>1</b>	<b>Introdução .....</b>	<b>1</b>
1.1	Motivação .....	1
1.2	Objetivos .....	3
1.3	Estrutura do Trabalho .....	3
<b>2</b>	<b>Engenharia de Software Orientada a Agentes .....</b>	<b>4</b>
2.1	Introdução .....	4
2.2	Sistemas Multi-Agentes .....	5
2.2.1	Aplicações .....	6
2.3	Metodologias de Desenvolvimento Orientado a Agentes .....	8
2.4	Implementação de Agentes de Software .....	9
2.4.1	Especificações .....	9
2.4.2	Comunicação entre os agentes .....	11
2.4.3	Plataformas de Desenvolvimento .....	13
<b>3</b>	<b>Desenvolvendo SMA com Tropos .....</b>	<b>23</b>
3.1	Introdução .....	23
3.2	O framework i* .....	23
3.2.1	O Modelo de Dependência Estratégica .....	24
3.2.2	O Modelo de Razão Estratégica .....	27
3.3	O framework NFR .....	27
3.4	A metodologia Tropos .....	29
3.4.1	Requisitos Iniciais .....	31
3.4.2	Requisitos Finais .....	31
3.4.3	Projeto Arquitetural .....	31
3.4.4	Projeto Detalhado .....	33
3.4.5	Implementação .....	34
3.5	Considerações .....	35
<b>4</b>	<b>Desenvolvendo SMA com JADEX .....</b>	<b>36</b>
4.1	Introdução .....	36
4.2	O modelo BDI .....	36
4.3	JADEX: Arquitetura e Modelo de Execução .....	38
4.4	Os Agentes .....	40
4.4.1	Mensagens e Protocolos .....	42
4.4.2	Integração com JADE .....	43
4.4.3	Ontologia .....	48
<b>5</b>	<b>Estudo de Caso .....</b>	<b>52</b>
5.1	Introdução .....	52
5.2	Sistema Gerenciador de Notícias na Web .....	53
5.3	Modelagem com Tropos .....	55
5.3.1	Requisitos Iniciais .....	55
5.3.2	Requisitos Finais .....	60
5.3.3	Projeto Arquitetural .....	62
5.3.4	Projeto Detalhado .....	66
5.4	Implementação em JADEX .....	68
<b>6</b>	<b>Conclusão .....</b>	<b>72</b>
<b>7</b>	<b>Referências Bibliográficas .....</b>	<b>73</b>

# Índice de Figuras

Figura 2.1: Serviços providos por uma plataforma compatível com a [FIPA04].	10
Figura 2.2: Arquitetura de referência da FIPA para uma plataforma de agentes [FIPA04].	11
Figura 2.3: Modelo de referência FIPA - especificação da comunicação entre agentes.	12
Figura 2.4: Arquitetura da plataforma JADE [WQoS03].	16
Figura 2.5: Diagrama de estados de agente definido pela FIPA.	17
Figura 2.6: Elementos de comunicação na plataforma JADE.	18
Figura 2.7: JADEX RMA.	19
Figura 2.8: Janela de inicialização de agentes em JADEX.	20
Figura 2.9: Introspector JADEX.	20
Figura 2.10: Logger JADEX.	21
Figura 2.11: Agente <i>Log Manager</i> .	22
Figura 2.12: Tracer JADEX.	22
Figura 3.1: Elementos de modelagem com <i>i*</i> .	25
Figura 3.2: Tipos de relacionamento de dependência entre atores no <i>i*</i> .	25
Figura 3.3: Associações e representação de agentes, papéis e posições.	26
Figura 3.4: (a) Ligações de decomposição de tarefas (b) Ligações de meios-fins.	27
Figura 3.5: Visão geral do <i>framework</i> NFR.	28
Figura 3.6: Exemplo de catálogo.	28
Figura 3.7: Exemplo de Grafo SIG.	29
Figura 3.8: Estilo organizacional arquitetural União Estratégica ( <i>Joint Venture</i> ).	33
Figura 3.9: Visão geral do mapeamento <i>i*/BDI/JACK</i> [Castro02a].	35
Figura 4.1: Arquitetura BDI Genérica (Adaptado de [Wooldridge95]).	37
Figura 4.2: Arquitetura BDI de JADEX [Pokahr05].	38
Figura 4.3: Ciclo de Vida do Objetivo [Pokahr05].	39
Figura 4.4: Componentes de um agente JADEX.	41
Figura 4.5: Formato das mensagens trocadas entre os agentes.	42
Figura 4.6: Modelo de execução e integração de JADEX com JADE.	43
Figura 4.7: Hierarquia das classes de comportamento definidas em JADE.	46
Figura 4.8: Conversão realizada pelo suporte de JADE para linguagens de conteúdo e ontologias.	49
Figura 4.9: O modelo de referência de conteúdo de JADE.	51
Figura 5.1: Modelo <i>i*</i> de Dependência Estratégica dos Requisitos Iniciais.	56
Figura 5.2: Modelo <i>i*</i> de Razão Estratégica de uma redação de jornal para o ator editor.	57
Figura 5.3: Modelo <i>i*</i> de Razão Estratégica de uma redação de jornal para o ator <i>webmaster</i> .	58
Figura 5.4: Modelo <i>i*</i> de Razão Estratégica dos Requisitos Iniciais para o ator repórter.	59
Figura 5.5: Modelo <i>i*</i> de Razão Estratégica de uma redação de jornal para o ator <i>chefe-redação</i> .	60
Figura 5.6: Modelo <i>i*</i> de Dependência Estratégica dos Requisitos Finais.	61
Figura 5.7: Modelo <i>i*</i> de Razão Estratégica dos Requisitos Finais.	62
Figura 5.8: Grafo de interdependências de metas-soft do <i>Smart Journal</i> .	64
Figura 5.9: Arquitetura do <i>Smart Journal</i> em União Estratégica.	65
Figura 5.10: Decomposição do <i>Smart Journal</i> com padrões sociais.	66
Figura 5.11: Diagrama de classe para o <i>Smart Journal</i> .	66
Figura 5.12: Diagrama de seqüência, interação entre os agentes Editor e <i>Webmaster</i> – Publicar Notícia.	67
Figura 5.13: Diagrama de seqüência, interação entre os agentes Editor e Webmaster – Consultar se a Notícia já foi publicada.	67
Figura 5.14: Mapeamento <i>i*/BDI/JADEX</i> .	68

# Índice de Tabelas

Tabela 3.1: Estilos arquiteturais organizacionais definidos em Tropos. ....	32
Tabela 4.1: Tabela com os tipos de comportamento. ....	45
Tabela 5.1: Catálogo de Correlação. ....	63



# Índice de Exemplos

Exemplo 4.1: Declaração de um ADF. ....	41
Exemplo 4.2: Linha de comando para iniciar o agente. ....	41
Exemplo 4.3: Criando um <i>container</i> de agentes. ....	42
Exemplo 4.4: Modelo para permitir que as mensagens sejam manipuladas por JADEX. ....	44
Exemplo 4.5: Representação de uma expressão de conteúdo ACL. ....	48
Exemplo 4.6: Representação da informação como uma instância de uma classe. ....	49
Exemplo 4.7: Uso de linguagem e ontologia pelo agente. ....	51
Exemplo 5.1: Exemplo de representação de notícias segundo o padrão NITF. ....	69

# Índice de Abreviaturas

ACC	Agent Communication Channel
ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management Service
API	Application Programming Interface
AUML	Agent Unified Modeling Language
BDI	Beliefs, Desires e Intentions
CASE	Computer Aided Software Engineering
CMS	Content Management System
DF	Directory Facilitator
CL	Content Language
ER	Engenharia de Requisitos
ES	Engenharia de Software
ESOA	Engenharia de Software Orientada a Agentes
FIPA	Foundation for Intelligent Physical Agents
GOOD	Goals into Object Oriented Development
GOOSE	Goal Into Object Oriented Standard Extension
GUI	Graphical User Interface
GUID	Globally Unique Identifier
HTML	Hyper Text Markup Language
HTTP	Hypertext Transmission Protocol
IDL	Interface Definition Language
IOP	Internet Inter Orb Protocol
IRE	Identifying Referential Expressions
JADE	Java Agent DEvelopment Framework
JADEX	Java Agent DEvelopment EXtension Framework
JESS	Java Expert System Shell
JVM	Java Virtual Machine
LEAP	Lightweight Extensible Agent Platform
NFR	Non-Functional Requirements
OME	Organizational Modeling Environment
PRS	Procedural Reasoning System
RMA	Remote Monitoring Agent
RMI	Remote Method Invocation
RNF	Requisitos Não Funcionais
SD	Strategic Dependency
SR	Strategic Rationale
SIG	Softgoal Interdependency Graph
SMA	Sistema Multi-Agentes
TCP	Transmission Control Protocol
UML	Unified Modeling Language
XGOOD	eXtended Goal Into Object Oriented Development
XML	Extensible Markup Language
WWW	World Wide Web

## Capítulo

# 1 Introdução

Este capítulo apresenta as principais motivações na realização deste trabalho. As seções seguintes apontam os objetivos deste e por fim, a sua estrutura.

## 1.1 Motivação

Nos dias de hoje, a Engenharia de Software (ES) para novos domínios de aplicações, tais como gerenciamento de conhecimento, computação *peer-to-peer* ou serviços para Internet, lida com a construção de sistemas abertos capazes de tratar informações distribuídas, dinâmicas e heterogêneas. Com isso, a Engenharia de Software Orientada a Agentes (ESOA) está se tornando uma das áreas mais crescentes na academia e na indústria, fundamentalmente por tratar características inerentes a estes tipos de sistemas, tais como: alta interatividade, não-determinismo, distribuição, adaptabilidade, etc.

A maioria destes sistemas de software existe em ambientes organizacionais e operacionais que estão frequentemente mudando, onde novos componentes podem ser adicionados, modificados ou removidos a qualquer momento. Isso geralmente ocorre devido a mudanças nas estruturas organizacionais, modelos de negócio, dinâmicas de mercado, estruturas legais e regulamentares, sentimentos públicos e avanços culturais. Contudo, muitos sistemas falham no suporte às organizações das quais eles são parte integrante. Uma das causas deste problema é a ausência de um entendimento apropriado da organização pelos projetistas do sistema, bem como as mudanças que não conseguem ser acomodadas pelos sistemas de software existentes. Assim, devemos projetar metodologias de ES que sejam adaptáveis a mudanças e foquem no esclarecimento e definição dos relacionamentos entre o sistema que se pretende construir e o mundo.

Neste contexto, a Engenharia de Requisitos (ER) vem sendo conhecida como a fase mais crítica do processo de desenvolvimento de sistemas porque, para construção de um software com qualidade, é preciso que as considerações técnicas estejam em equilíbrio com as sociais e organizacionais desde a modelagem de sistemas [Bresciani04].

Dentre as metodologias da Engenharia de Requisitos, Tropos se apresenta como uma proposta de metodologia para desenvolvimento de sistemas orientados a agentes, inspirada na análise de requisitos e fundamentada em conceitos sociais e intencionais [Castro02] [Giorgini05]. As duas características fundamentais da metodologia Tropos são: (i) o uso de conceitos de nível de conhecimento [Newell93], tais como agente, meta, plano e outros, durante todas as fases do desenvolvimento de software; e (ii) o importante papel atribuído à análise de requisitos quando o ambiente e o sistema a ser desenvolvido são analisados [Castro02]. A palavra “tropos” deriva do grego

“tropé”, que significa fácil de mudar ou adaptar, portanto, Tropos tem como objetivo o desenvolvimento de sistemas estudados de acordo com as reais necessidades de uma organização, buscando um melhor casamento entre o sistema e o ambiente, e permitindo uma melhor estruturação de sistemas adaptáveis. Sua abordagem utiliza os modelos e conceitos oferecidos pelo *framework* i\* [Yu95].

O i\* [Yu95] é um *framework* de modelagem organizacional que possui uma estrutura conceitual rica, capaz de reconhecer motivações, intenções e raciocínios sobre as características de um processo, o que facilita os esforços da ER. Este *framework* é formado por dois modelos: o modelo de Dependência Estratégica (SD) e o modelo de Razão Estratégica (SR). O modelo SD fornece uma descrição intencional do processo em termos de uma rede de relacionamentos de dependência entre atores relevantes. O modelo SR, por sua vez, apresenta uma descrição estratégica do processo, em termos de elementos do processo e das razões que estão por detrás deles, ou seja, fornece uma análise meio-fins de como as metas podem ser cumpridas através das contribuições dos demais atores. Tanto o modelo SD quanto o modelo SR do i\* são usados por Tropos para capturar as intenções dos *stakeholders*, as responsabilidades do novo sistema em relação a estes *stakeholders*, a arquitetura do novo sistema e os detalhes de seu projeto.

Tropos, embora ainda em constante evolução, oferece um *framework* que engloba as principais fases de desenvolvimento de software, com o apoio das seguintes atividades: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural e Projeto Detalhado. Recentemente o escopo de Tropos foi estendido passando a incluir técnicas para geração de uma implementação a partir dos artefatos gerados na fase de Projeto Detalhado. Esta fase complementa propostas para plataformas de programação orientada a agentes. Tropos visa, entre outros objetivos, a definição de arquiteturas de software mais flexíveis, robustas e abertas. Além disso, estão sendo realizados esforços para tornar a metodologia mais compatível com o paradigma de programação orientada a agentes, bem como para dar um melhor suporte a áreas de aplicação baseadas na *Web*, tais como telecomunicações e comércio eletrônico.

Existe um grande número de ferramentas e plataformas de implementação de SMA que dão suporte a atividades ou fases do processo de desenvolvimento de software orientado a agentes. Algumas delas dão suporte a implementação, enquanto que outras também auxiliam a análise, o projeto e as atividades de teste e depuração [Weiss02]. A maioria delas permite integração com a linguagem de programação Java. Dentre as ferramentas e plataformas de desenvolvimento orientado a agentes mais freqüentemente listadas na literatura e que estão de acordo com as especificações da FIPA [FIPA04], podem ser citadas: ZEUS [ZEUS03], JADE [JADE04], JADEX [JADEX04], LEAP [LEAP02], FIPA-OS [FIPA-OS05] e Grasshopper [Grasshopper02].

Este trabalho então, investiga a questão do desenvolvimento de sistemas para o paradigma de orientação a agentes. Em particular, estaremos preocupados em estudar o desenvolvimento de sistemas multi-agentes utilizando a proposta Tropos e a proposta de implementação destes sistemas na plataforma de desenvolvimento orientado a agentes JADEX, que é uma extensão da plataforma JADE.

## 1.2 Objetivos

O principal objetivo deste trabalho é apoiar a implementação de um SMA, utilizando a plataforma JADEX, e especificado segundo a metodologia Tropos. Neste utilizamos as facilidades oferecidas pela engenharia de software ao realizarmos a construção de um sistema usando a metodologia Tropos para capturar aspectos dos agentes que precisam ser levados em consideração quando desejamos implementar um sistema multi-agentes.

Nossa proposta é validada apresentando como estudo de caso o desenvolvimento de um módulo de um sistema de gerenciamento de conteúdo (CMS) que utiliza agentes para simular o processo de publicação de notícias em um site da *web*. Este módulo envolve apenas dois agentes (um editor e o *webmaster*) e a interação entre eles.

Os principais objetivos deste trabalho são:

- Investigar o paradigma de Engenharia de Software Orientada a Agentes e a proposta da metodologia Tropos;
- Investigar o uso de ferramentas e linguagens de modelagem apropriadas ao desenvolvimento de sistemas multi-agentes;
- Investigar as plataformas de desenvolvimento de agentes: JADE e JADEX;
- Identificar as vantagens e deficiências encontradas nas plataformas de desenvolvimento estudadas;
- Desenvolver um estudo de caso modelado com Tropos e propor a sua implementação em JADEX.

## 1.3 Estrutura do Trabalho

Além deste capítulo introdutório sobre motivação e objetivos, o capítulo 2 apresenta os conceitos básicos e as características envolvidas nesse trabalho, como o conceito de agentes, sistemas multi-agentes, algumas metodologias propostas, padrões e plataformas para desenvolvimento de agentes. O capítulo 3 apresenta os principais aspectos envolvidos no desenvolvimento de SMA utilizando a metodologia Tropos, seguindo suas diversas etapas. O capítulo 4 mostra os aspectos envolvidos na implementação de um SMA utilizando a Plataforma JADEX, da definição da implementação, integração com JADE e ferramentas de gerenciamento. O capítulo 5 apresenta um estudo de caso desenvolvido (um módulo de um sistema de gerenciamento de conteúdo – CMS) segundo os estudos realizados no capítulo anterior (metodologia e plataforma escolhidas). O capítulo 6 apresenta alguns comentários finais e considerações para o desenvolvimento de trabalhos futuros. O capítulo 7 relaciona as referências utilizadas para produção deste trabalho. E por fim, nos Apêndices A e B encontram-se os diagramas produzidos como artefatos da metodologia seguida para o desenvolvimento do estudo de caso.

## Capítulo

# 2 Engenharia de Software Orientada a Agentes

Este capítulo apresenta os conceitos relativos à Engenharia de Software Orientada a Agentes, conceito de agentes e de Sistemas Multi-Agentes (SMA), metodologias de desenvolvimento e implementação de agentes de software, além de apresentar plataformas de desenvolvimento.

Inicialmente são apresentadas as motivações para o uso desse novo paradigma da engenharia de software, seguido dos aspectos e propriedades que caracterizam um agente. Depois, apresentamos conceitos relativos à SMA, seguida de uma relação das principais áreas de aplicação dos softwares orientados a agentes, bem como os desafios e obstáculos enfrentados atualmente no seu desenvolvimento. Por fim falaremos um pouco sobre metodologias de desenvolvimento orientadas a agentes e implementação de agentes de software.

## 2.1 Introdução

A Engenharia de Software Orientada a Agentes está evoluindo rapidamente em resposta a necessidade do desenvolvimento de sistemas cada vez mais complexos. É uma nova área de pesquisa que se preocupa com metodologias e técnicas para produção com qualidade de softwares orientados a agentes. Esta abordagem tem como principal abstração computacional, usada para modelagem da aplicação, o conceito de agente.

Segundo a [FIPA04], agente é uma entidade que reside em um ambiente onde interpreta os dados através de sensores que refletem eventos no ambiente e executam ações que produzem efeitos no mesmo. O agente pode ser um software ou hardware puro.

De acordo com [Weiss02], um agente pode ser visto como um sistema computacional encapsulado que está situado em algum ambiente e é capaz agir de forma flexível e autônoma neste ambiente, a fim de atingir os objetivos para os quais foi projetado. Agentes podem ser úteis como entidades isoladas às quais são delegadas tarefas particulares, ou estes podem existir em ambientes contendo outros agentes. Neste último caso, eles têm de cooperar, negociar e coordenar ações [Zambonelli03].

Neste nível de abstração, o conceito de organização é bastante empregado e inclui: agentes, ambiente, organização, papéis, objetivos ou metas, responsabilidades e interações.

Agentes possuem as seguintes propriedades entre outras:

- Autonomia – habilidade do software de agir independentemente sem intervenção direta humana ou de outros agentes [Yu05].

- Proatividade – agentes não agem simplesmente em resposta ao seu ambiente, eles são capazes de exibir proatividade, comportamento dirigido a objetivo e tomar a iniciativa quando necessário [Jennings00].
- Sociabilidade – habilidade de participar de muitos relacionamentos, interagindo com outros agentes ao mesmo tempo ou em tempos diferentes [Yu05].

Assim, podemos justificar o uso dessa abstração do paradigma de agentes, se observamos algumas características [Schwambach04]:

- O domínio da aplicação envolve distribuição de dados, capacidade de resolução de problemas e responsabilidades;
- Há necessidade de se manter autonomia das partes envolvidas, bem como a estrutura organizacional;
- Há interações incluindo negociação, compartilhamento de informações e coordenação;
- O ambiente do domínio da aplicação apresenta características como não-determinismo – a solução do problema não pode ser inteiramente descrita do início ao fim, pois pode haver mudanças no ambiente, imprevisibilidade e dinamicidade.

O aparato conceitual de sistemas orientados a agentes é adequado para construir soluções de software para sistemas complexos. De fato, técnicas orientadas a agentes são adequadas para desenvolver sistemas complexos de software por que [Jennings99]:

- O paradigma orientado a agentes provê uma forma efetiva de decompor o escopo do problema de um sistema complexo;
- As abstrações presentes na orientação a agentes são um meio natural de modelar sistemas complexos;
- O conceito de relacionamentos organizacionais presentes no paradigma é apropriado para estes tipos de sistemas.

Segundo [Jennings99], algumas razões para o crescimento do interesse em desenvolvimento com sistemas multi-agentes são:

- A capacidade de fornecer robustez e eficiência;
- A capacidade de permitir interoperabilidade entre sistemas legados;
- A capacidade de resolver problemas cujo dado, especialidade ou controle é distribuído.

Nas seções seguintes, apresentaremos os principais conceitos relacionados ao paradigma orientado a agentes.

## **2.2 Sistemas Multi-Agentes**

Um SMA pode ser visto como um conjunto de agentes autônomos, que interagem entre si e com o ambiente, estando imerso e atuando neste. Cada agente pode desempenhar um ou mais papéis, possui um conjunto bem definido de responsabilidades e visa atingir os objetivos ou metas. Esses

objetivos podem ser comuns a todos os agentes ou não. Para atingir suas metas, os agentes interagem uns com os outros, sendo essa comunicação suportada por uma linguagem de comunicação entre agentes e por uma ontologia usada para associar significado ao conteúdo das mensagens [Schwambach04].

A pesquisa em sistemas multi-agentes se preocupa com o comportamento de uma coleção de agentes autônomos, possivelmente pré-existent, visando resolver um dado problema. Desse modo, estes tipos de sistemas são apropriados para representar problemas que têm vários métodos de resolução, múltiplas perspectivas e/ou múltiplas entidades de resolução.

Um sistema multi-agente pode ser definido como uma rede fracamente acoplada de solucionadores de problema que trabalham juntos para resolver problemas que estão além de suas capacidades ou conhecimentos individuais. Estes solucionadores de problemas – agentes – são autônomos e podem ser heterogêneos por natureza. Um sistema multi-agente inclui as seguintes características [Jennings00]:

- Cada agente tem informações, recursos ou capacidades insuficientes para solucionar um dado problema. Dessa forma, cada agente tem um ponto de vista limitado;
- A informação é descentralizada;
- A computação e a interação são assíncronas.

Na maioria dos casos, os agentes agem para atingir objetivos em benefício de indivíduos ou de empresas. Assim, quando agentes interagem, há tipicamente algum contexto organizacional em destaque. Este contexto ajuda a definir a natureza dos relacionamentos entre os agentes.

Para suportar sistemas multi-agentes, um ambiente apropriado precisa ser estabelecido, devendo [Odell00]:

- Fornecer uma infra-estrutura de comunicação específica e protocolos de interação;
- Ser tipicamente aberto e não possuir um projeto centralizado ou uma função de controle *top-down*;
- Conter agentes que sejam autônomos e adaptativos.

## 2.2.1 Aplicações

A tecnologia de agentes está ultrapassando rapidamente os limites das universidades e dos laboratórios de pesquisa e está começando a ser usada para resolver problemas do mundo-real em uma escala de aplicações industriais e comerciais.

Aqui listamos algumas das principais áreas onde as abordagens orientadas a agentes estão sendo usadas [Garcia03]:

- Controle de Tráfego Aéreo: Agentes são usados para representar tanto os equipamentos de aviação quanto vários sistemas de controle de tráfego aéreo em operação. A aplicação de sistemas multi-agentes mais comentada é um sistema para controle de tráfego aéreo que está sendo testado para uso no aeroporto de Sydney na Austrália. Este sistema foi implementado



usando o PRS (*Procedural Reasoning System*, sistema que usa uma arquitetura BDI) [PRS]. O PRS foi utilizado também para implementar outras importantes aplicações, como alguns sistemas para controle de espaçonaves e robôs.

- Indústria: Modelar uma linha de produção através de agentes cooperantes é uma área de aplicação bastante grande. Sistemas nessa área incluem projeto de configuração de produtos de fabricação, projeto colaborativo, cronograma e controle de operações de fabricação, controle de fabricação de um robô e determinação de seqüências de produção para uma fábrica.
- Gerência de Negócios: Agentes podem ser empregados em sistemas de *workflow* (i.e., sistemas que visam garantir que as tarefas necessárias serão feitas pelas pessoas certas nas horas certas).
- Interação Homem-Computador: com o intuito de melhorar a qualidade das interfaces de software.
- Ambientes de Aprendizagem: Estes sistemas podem empregar agentes tanto com o mesmo objetivo mencionado acima como para a modelagem dos alunos em tutores inteligentes.
- Entretenimento: Agentes podem ser utilizados para aumentar o realismo de personagens de jogos e sistemas para a indústria de entretenimento em geral.
- Aplicações Distribuídas: Para domínios naturalmente distribuídos, a metáfora de agente é bastante adequada; telecomunicações (controle de rede, transmissão e chaveamento, gerência de serviço e gerência de rede), transportes (gerência de tráfego e transporte) e sistemas para a área de saúde são alguns exemplos.
- Aplicações para a Internet: Esta é uma área muito comum para aplicações multi-agentes. Existem várias classes de sistemas que se enquadram neste item, como sistemas para gerência de informação (em particular agentes conhecidos como *personal digital assistants* ou assistentes digitais pessoais, que auxiliam na gerência da sobrecarga de informação recebida pelas pessoas via Internet), sistemas de comércio eletrônico e sistemas para busca de informações na Internet.
- Simulação Social: O objetivo destas simulações multi-agentes é auxiliar cientistas sociais em seus estudos. Para isto é preciso representar aspectos cognitivos e sociais de comunidades de pessoas, vindo ao encontro dos objetivos dos sistemas multi-agentes. Este tipo de simulação pode auxiliar na compreensão de questões importantes para as ciências sociais.

Diante do uso crescente dessa nova tecnologia, alguns desafios e obstáculos foram encontrados no desenvolvimento de software orientado a agentes. Torna-se importante o uso de notações, ferramentas e metodologias específicas para o desenvolvimento de software orientado a agentes, visando à construção de sistemas mais robustos, confiáveis e reutilizáveis. É importante também o uso de padrões no que diz respeito à comunicação, interação e coordenação entre os agentes.

## 2.3 Metodologias de Desenvolvimento Orientado a Agentes

A construção de um SMA engloba todos os problemas envolvidos no desenvolvimento de sistemas distribuídos e concorrentes tradicionais, e ainda as dificuldades adicionais que surgem dos requisitos de flexibilidade e interações sofisticadas. Além disso, qualquer metodologia para ESOA deve prover abstrações adequadas e ferramentas para modelar tanto as tarefas individuais dos agentes quanto as tarefas sociais, ou organizacionais.

Segundo [Wooldridge02], uma solução orientada a agentes é apropriada: i) quando o ambiente é aberto, ou no mínimo altamente dinâmico, incerto e complexo; ii) quando agentes são uma metáfora natural; iii) quando controle, dados e *expertise* precisam ser distribuídos e atuarem como componentes semi-autônomos ou autônomos; e iv) em sistemas legados.

Odell [Odell00] afirma que sistemas multi-agentes são vistos como uma nova direção importante na engenharia de software por que:

- Provêem uma maneira de pensar sobre o fluxo de controle em um sistema altamente distribuído;
- Oferecem um mecanismo que permitem um comportamento dinâmico ao invés de uma arquitetura estática;
- Codificam melhores práticas de como organizar entidades colaborativas concorrentes.

Nesse sentido, várias metodologias têm sido propostas nos últimos anos para dar suporte à construção de SMA [Weiss02].

GAIA [Wooldridge00] representa a primeira tentativa de obter uma metodologia completa e genérica para especificar análise e projeto de SMA. Nesta metodologia, SMAs são visualizados como sendo um conjunto composto de um grande número de entidades autônomas (como uma sociedade organizada por indivíduos) que apresentam um ou mais papéis específicos. A metodologia GAIA é aplicável a um grande conjunto de SMA, além de tratar aspectos de nível macro (sociedade) e micro (agente) dos sistemas [Zambonelli03]. GAIA é baseada na visão de que um sistema multi-agentes se comporta como uma organização computacional que consiste de vários papéis interagindo. Permitindo que um analista vá sistematicamente do estabelecimento de requisitos até um projeto que seja suficientemente detalhado a ponto de ser implementado. Além disso, GAIA toma emprestada parte da terminologia e notação da análise e projeto orientados a objetos.

*Agent UML* (AUML) [Odell01] é uma metodologia de análise e projeto que estende a UML para representar agentes. Ela sintetiza uma preocupação crescente das metodologias de software baseado em agentes com o aumento da aceitação da UML para o desenvolvimento de software orientado a agentes. Agentes são vistos como objetos ativos, exibindo autonomia dinâmica (capacidade de ação pró-ativa) e autonomia determinística (autonomia para recusar uma solicitação externa). O objetivo da AUML é fornecer uma semântica semi-formal e intuitiva, através de uma notação gráfica amigável para o desenvolvimento de sistemas orientados a agentes. A AUML fornece extensões da UML, adicionando uma representação em três camadas para os protocolos de interação de agentes, que descrevem um padrão de comunicação com uma seqüência permitida de mensagens entre agentes e as restrições sobre o conteúdo destas mensagens.

Uma outra abordagem orientada a agentes, chamada Tropos [Castro02], é uma metodologia que está sendo desenvolvida, visando dar suporte a todas as fases do desenvolvimento de software orientado a agentes, e propõe uma abordagem centrada em requisitos, adotando na fase inicial os conceitos oferecidos pelo i\* [Yu95], tais como ator e dependências sociais entre atores, incluindo dependências de metas, *softgoals*, tarefas e recursos. Ela tem adotado a perspectiva orientada a agentes desde a fase de requisitos, visando reduzir tanto quanto possível o mau casamento de impedância entre o sistema e seu ambiente. Tal metodologia será detalhada no próximo capítulo.

## 2.4 Implementação de Agentes de Software

### 2.4.1 Especificações

Devido a natureza dos SMAs, os mesmos não podem ser desenvolvidos recorrendo às tecnologias de software tradicionais dada as suas limitações relativas à distribuição e interoperabilidade. Observando que muitas das características básicas dos SMAs são independentes de aplicação e buscando facilitar o desenvolvimento destes sistemas, começaram a surgir *frameworks* para o desenvolvimento destes tipos de sistemas. As tecnologias baseadas em agentes parecem ser uma resposta promissora capaz de facilitar o desenvolvimento destes sistemas.

Tais *frameworks* oferecem todas as funcionalidades básicas de um SMA, o que permite ao desenvolvedor se preocupar apenas com a parte que lhe interessa: o design dos agentes. Essa é a chamada abordagem horizontal ou *middleware*, ou seja, os *frameworks* oferecem uma biblioteca de nível relativamente alto, porém genérica e independente de aplicação, diferente da abordagem vertical onde soluções *ad-hoc* específicas são implementadas.

As primeiras tentativas de criar *frameworks* para SMA aconteceram ainda nos anos 80. Mas foi nos anos 90, com a explosão da Internet, que o interesse em SMA cresceu. Os pesquisadores imaginavam que a Internet seria o ambiente ideal para o surgimento de vários sistemas multi-agentes. No entanto, a falta de padrões de reconhecimento internacional era uma barreira para o desenvolvimento dessas tecnologias por impedir a interoperabilidade entre os sistemas.

Assim, em 1996, surgiu a FIPA (*Foundation for Intelligent Physical Agents*), uma sociedade da IEEE. Esta fundação surgiu como uma associação, sem fins lucrativos, de empresas e organizações com o intuito de trabalhar na criação de padrões para a interoperabilidade de agentes de software heterogêneos [FIPA04]. O primeiro conjunto de especificações da FIPA só seria lançado em 1997 e só no fim de 2002 essas especificações seriam finalizadas e definidas como padrão.

As especificações da FIPA foram um grande incentivo para o surgimento de novos esforços para a criação de *frameworks* genéricos, que garantissem a interoperabilidade de SMA. Exemplos de *frameworks* dessa geração são o FIPA-OS [FIPA-OS05] e o JADE [JADE04].

A principal missão da FIPA é: a promoção e aprimoramento contínuo de tecnologias e especificações de interoperabilidade que facilitem e promovam a interligação de sistemas de agentes inteligentes nos setores industrial e comercial, visando à interoperabilidade entre sistemas autônomos. O trabalho de padronização da FIPA destina-se a facilitar a interoperabilidade entre sistemas de agentes, uma vez

que, além da linguagem de comunicação, a FIPA especifica também quais os principais agentes envolvidos na gestão de um sistema, a ontologia necessária para a interação entre sistemas, e ainda o nível de transporte dos protocolos.

A FIPA definiu um conjunto de especificações para guiar a implementação de plataformas multi-agentes (ambiente de execução onde operam os agentes). Tais especificações definem apenas um modelo de referência e, portanto, se limitam a descrever o comportamento externo dos componentes do sistema, deixando em aberto detalhes relativos à estrutura interna e implementação. Essas especificações definem um conjunto de serviços que precisam ser providos pelas plataformas. A Figura 2.1 oferece uma visão funcional das especificações da FIPA, mostrando serviços obrigatórios e opcionais em uma plataforma.

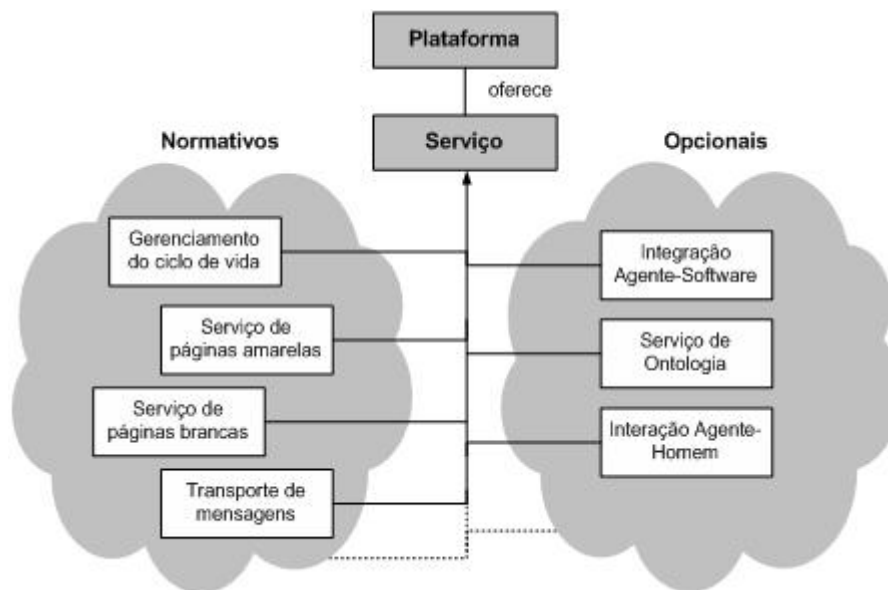


Figura 2.1: Serviços providos por uma plataforma compatível com a [FIPA04].

O serviço de gerenciamento do ciclo de vida do agente é responsável pela criação, remoção e migração de agentes entre plataformas. A FIPA define os possíveis estados no ciclo de vida um agente e as transições que ocasionam as mudanças de estados.

Quando um agente é criado, a plataforma deve associar ao agente um identificador. Este identificador deve ser imutável e único para permitir que um agente ou serviço possa contatar outro agente de maneira não ambígua.

O serviço de páginas brancas permite a um agente encontrar agentes capazes de prover um determinado serviço de que necessita. Já o serviço de páginas amarelas, lista os serviços oferecidos por um dado agente. Qualquer agente pode se cadastrar (e remover seu cadastro) nos serviços de páginas brancas e amarelas da plataforma para anunciar seus serviços. Uma vez cadastrado, ele pode também alterar a descrição fornecida inicialmente.

O serviço de transporte de mensagens é o principal serviço da plataforma, pois permite a interação entre os agentes. Esse serviço é o responsável por entregar as mensagens (assíncronas) trocadas entre os agentes, estejam eles na mesma plataforma ou em plataformas distintas. O padrão da FIPA

prevê a possibilidade de implementação de mecanismos de segurança (e.g., encriptação) no transporte de mensagens.

Dos serviços opcionais, destacaremos apenas o serviço de ontologia que oferece a possibilidade de os agentes compartilharem um repositório com as ontologias utilizadas pelos agentes do sistema, evitando ambigüidades. Abaixo apresentamos a arquitetura de referência da FIPA [FIPA04].

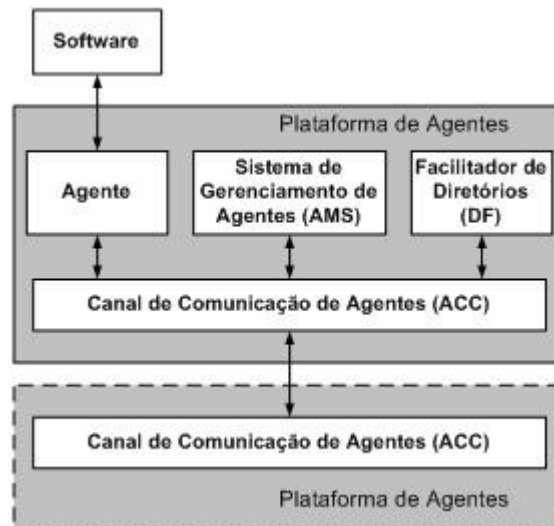


Figura 2.2: Arquitetura de referência da FIPA para uma plataforma de agentes [FIPA04].

O Sistema de Gerenciamento de Agentes (AMS, *Agent Management System*) é o agente responsável por supervisionar o acesso e o uso da plataforma. Apenas um AMS deve existir por plataforma. Ele é o responsável pela autenticação de agentes residentes e pelo controle de registro. O AMS oferece os serviços de páginas brancas e de ciclo de vida, gerenciando um repositório de identificadores de agentes (AID, *Agent Identifiers*) e dos próprios agentes. Cada agente deve se registrar com o AMS para poder obter um AID válido. O AMS ativa e/ou desativa os protocolos de transporte de mensagem (MTP – *Message Transport Protocol*) nos *containers*.

O Diretório Facilitador (DF, *Directory Facilitator*) é o agente que provê o serviço de nomes (páginas amarelas) na plataforma. A plataforma pode suportar outros DFs, além do default, e esses DFs podem se organizar entre si formando federações.

O Sistema de Transporte de Mensagem (*Message Transport System*), também chamado de Canal de Comunicação de Agentes (ACC, *Agent Communication Channel*), é o componente responsável por prover o serviço de transporte de mensagens, comunicação básica entre agentes dentro e fora da plataforma. Ele deve suportar o protocolo IIOP (*Internet Inter Orb Protocol* do padrão CORBA) para permitir a interoperabilidade entre diferentes plataformas de agentes. A especificação permite que haja mais de uma instância de tal serviço, desde que todo agente tenha acesso a pelo menos uma dessas instâncias.

## 2.4.2 Comunicação entre os agentes

Agentes comunicam-se na busca de atingir os seus objetivos em um ambiente compartilhado. Em uma sociedade, as ações de agentes são ações coordenadas, seja para cooperar ou competir (negociar).

Em qualquer modelo de coordenação, a comunicação tem, em geral, um papel central. A comunicação é um meio de o agente compartilhar seu estado com outros agentes, e também um meio para tentar influenciar o estado dos outros agentes [Bordini01].

Uma vez que se tenha um ambiente com vários agentes e vários serviços, deve-se padronizar a forma de comunicação entre os diversos agentes e a forma de acessar os serviços comuns. A comunicação permite que os agentes em um ambiente multi-agente troquem informações que servirão de base para coordenar suas ações e realizar cooperação [Mangan01].

Uma arquitetura de comunicação deve prover suporte para [Shahmehri98]: (a) comunicação agente para agente; (b) comunicação através da rede; e (c) mecanismos de segurança. Esta arquitetura de comunicação pode ser independente de linguagem e plataforma ou pode ser baseada em uma linguagem ou implementação particular.

Existem várias formas de permitir que agentes conversem com outros agentes. Uma forma é deixar que eles conversem diretamente, desde que eles conversem a mesma linguagem. Outra forma é através do uso de interpretadores ou facilitadores, garantindo que eles saibam como conversar com o interpretador, e então o interpretador pode conversar com o outro agente [Bigus98].

Para que os agentes possam se comunicar, eles precisam compartilhar um vocabulário de palavras e seus significados. Este vocabulário compartilhado é denominado ontologia (*ontology*).

ACL (*Agent Communication Language*) é o padrão de comunicação estabelecido pela FIPA. A especificação da FIPA não estabelece nenhuma restrição sobre a tecnologia utilizada de fato na implementação.

O padrão ACL estabelece a linguagem de mensagem padrão. As especificações FIPA sobre comunicação de agentes possuem três partes como ilustrado na Figura 2.3: (1) FIPA *Interaction Protocols* (IPs): tratam de protocolos de troca de mensagem pré-estabelecidas para mensagens ACL; (2) FIPA *Communicative Act* (CA): especificações que tratam das diferentes expressões (*utterances*) para mensagens ACL; (3) FIPA *Content Language* (CL): especificações que tratam das diferentes representações dos conteúdos das mensagens ACL [FIPA02].



Figura 2.3: Modelo de referência FIPA - especificação da comunicação entre agentes.

Outro padrão de troca de mensagens entre agentes é KQML (*Knowledge Query and Manipulation Language*) que provê um *framework* para programas e agentes trocarem informações e conhecimento. KQML foca no formato das mensagens, que são denominadas performativas (*performatives*) e nos protocolos para tratamento de mensagens entre os agentes. KQML usa ontologias de forma explícita para garantir que dois agentes se comunicando com a mesma linguagem possam interpretar os comandos corretamente nesta linguagem. Assim, para eliminar qualquer ambigüidade, cada mensagem explicitamente estabelece qual a ontologia que está sendo usada.

Dois agentes que queiram se comunicar usando KQML exigem os serviços de um facilitador KQML (*facilitator* ou *matchmaker*), que funciona como um ponto de encontro centralizado, ou de forma mais técnica, como um servidor de nomes. A comunicação com o facilitador é feita através de mensagens KQML padrão. Os agentes podem se registrar com o facilitador como um provedor de serviços ou de informações usando performativas *advertise*. A existência do facilitador evita que um agente precise enviar uma mensagem a todos os outros agentes quando precisar realizar uma colaboração. Por outro lado, o facilitador representa um elemento centralizador do sistema e, portanto trazendo problemas de escalabilidade e tolerância à falhas.

As sintaxes de ACL e KQML são muito parecidas. Embora ACL seja o padrão estabelecido pela FIPA, KQML tem se mostrado o padrão de fato para boa parte das implementações. Esta tendência pode, no entanto se alterar em função da possibilidade de adotar o padrão FIPA como um todo. Além disso, a existência de uma especificação formal para ACL evitaria ambigüidades no uso da linguagem.

### 2.4.3 Plataformas de Desenvolvimento

Muitos ambientes de desenvolvimento de sistemas com agentes têm sido desenvolvidos para dar assistência à construção de SMA. Dentre esses ambientes, podemos citar os três principais que estão de acordo com as especificações da FIPA: JACK [JACK05], FIPA-OS [FIPA-OS05] e JADE [JADE04].

JACK *Intelligent Agents* é um ambiente de desenvolvimento orientado a agentes construído inteiramente em Java e oferece uma extensão Java específica para implementação de comportamentos dos agentes. E permite a modelagem do comportamento dos agentes de acordo com a arquitetura BDI (*Belief Desire Intention*).

Para suportar a programação de agentes BDI, JACK oferece cinco construtores principais de linguagem: agentes, capacidades, relações da base de dados, eventos, e planos. Capacidades agregam eventos, planos, base de dados ou outras capacidades, cada uma delas assume uma função específica unido ao agente. As relações da base de dados armazenam crenças e dados de uma agente. Eventos identificam as circunstâncias e mensagens que um agente pode responder. Planos são instruções que um agente segue para alcançar seus objetivos e que manipulam seus eventos designados.

FIPA-OS é a primeira implementação *open-source* dos padrões da FIPA. É totalmente implementada em Java e oferece um conjunto de ferramentas baseadas em componente para construção de agentes de acordo com os padrões da FIPA.

JADE *Framework* (Java Agent Development Framework) é um *middleware* que facilita o desenvolvimento de SMA conforme os padrões da FIPA para agentes inteligentes. JADE não apenas facilita o desenvolvimento como também é utilizado para o gerenciamento de agentes. Nele está incluso dois produtos um compilador (FIPA - *Compliant Agent Platform*) e um pacote de desenvolvimento.

JADE possui as seguintes características principais [WQoS03]:

- Escalável (pode ser utilizado desde pequena escala a grande escala);
- Suporta mobilidade dos agentes;
- Oferece uma interface gráfica de usuário (GUI) para permitir o controle de vários agentes e plataformas ao mesmo tempo;
- Permite implementação de aplicações multi-domínio (a interface simplifica o registro de agentes em um ou mais domínios);
- Possui um mecanismo interno de transporte de mensagens;
- Lida, de forma transparente ao usuário, com todos os aspectos da comunicação.

JADE é um software aberto e está disponível sob licença LGPL (*Lesser General Public License*) em [JADE04]. O *framework* suporta a definição, armazenamento e carregamento de ontologias.

JADE é totalmente implementado em Java, sendo composto de vários pacotes Java (*Java packages*). A seguir seguem alguns deles [WQoS03]:

- O pacote **jade.core**, que implementa o kernel do sistema. Ele inclui a classe de agente (*Agent Class*), uma classe de comportamento (*Behaviour Class*) que está contida no sub-pacote **jade.core.behaviours**. Behaviour não é propriamente comportamento e sim uma implementação de uma tarefa ou intenção do agente, os programadores definem os agentes e escrevem os *behaviours* esperados.
- O sub-pacote **jade.lang.acl** é provido para processar as ACLs de acordo com as especificações da FIPA.
- O pacote **jade.content** é utilizado para permitir a utilização das ontologias e conteúdos de linguagem (CL – *Content languages*) definidos pelo usuário.
- O pacote **jade.domain** contém todas as classes Java que definem os agentes de gerenciamento de entidades estabelecidos pela FIPA, em particular o AMS e DF, além de outros conceitos como mobilidade e *sniffers*.
- O pacote **jade.gui** contém um conjunto de classes úteis para criar as GUIs para exibir e editar *Agent-Identifiers*, Agentes de descrição (*Agent Descriptions*), ACLMessages, dentre outros.
- O pacote **jade.mtp** contém uma interface Java que todo protocolo de transporte de mensagens deve implementar para ser integrado a JADE.
- O pacote **jade.proto** contém classes para modelar protocolos de interação (fipa-request, fipa-query, fipa-subscribe, além de muitos outros definidos pela FIPA)



- O pacote FIPA contém o módulo IDL para mensagens baseadas no IIOp-transporte.
- O sub-pacote **jade.tools** contém algumas ferramentas que facilitam a administração da plataforma.

Atualmente, JADE disponibiliza as seguintes ferramentas:

- *Agente de Monitoramento Remoto (RMA - Remote Monitoring Agent)*: console (agente) responsável pela administração e controle da plataforma, sendo que mais de uma GUI pode ser ativado. O JADE mantém coerência entre os RMAs através de envio de *multicasting* entre eles.
- *Dummy Agent*: ferramenta (e agente) de monitoramento e *Debugg*. Esta ferramenta possui funções típicas tais como enviar, receber e armazenar mensagens ACL.
- *Sniffer Agent*: ferramenta utilizada para debugar, realizar o rastreamento das mensagens e salvar em arquivo a comunicação entre agentes. Essa ferramenta é capaz de interceptar as mensagens ACL em trânsito e exibir uma anotação gráfica muito semelhante ao UML que é de grande utilidade para depuração da sociedade de agentes.
- *Introspector Agent*: ferramenta que permite monitorar o ciclo de vida dos agentes, suas mensagens ACL trocadas e os *behaviours* em execução.
- *Socket Proxy Agent*: agente simples que age como uma porta bidirecional entre a plataforma e uma conexão TCP/IP. As mensagens ACL trafegam sobre o serviço de transporte JADE, e são convertidas para ASC II. Este agente é útil para manipular *firewalls* ou prover interações entre a plataforma e *applets* java .
- *DF GUI*: um agente, interface de usuário que é usado devido à falta do diretório facilitador no JADE. Este permite um simples e intuitivo modo para controlar a base de conhecimentos, ou seja, provê o serviço de páginas amarelas.

JADE é uma plataforma distribuída, os *containers* podem estar espalhados entre vários *hosts* sendo que apenas uma máquina virtual Java (JVM - *JAVA Virtual Machine*) é executada em cada *host*, contanto que estes possam ser conectados por RMI (*Remote Method Invocation*), porém apenas um *host* terá um *container* principal que se difere dos demais devido ao fato de não conter AMS, o DF e o módulo IIOp. O *JADE Container Agent* (container não principal) é um servidor RMI que localmente administra a plataforma [WQoS03].

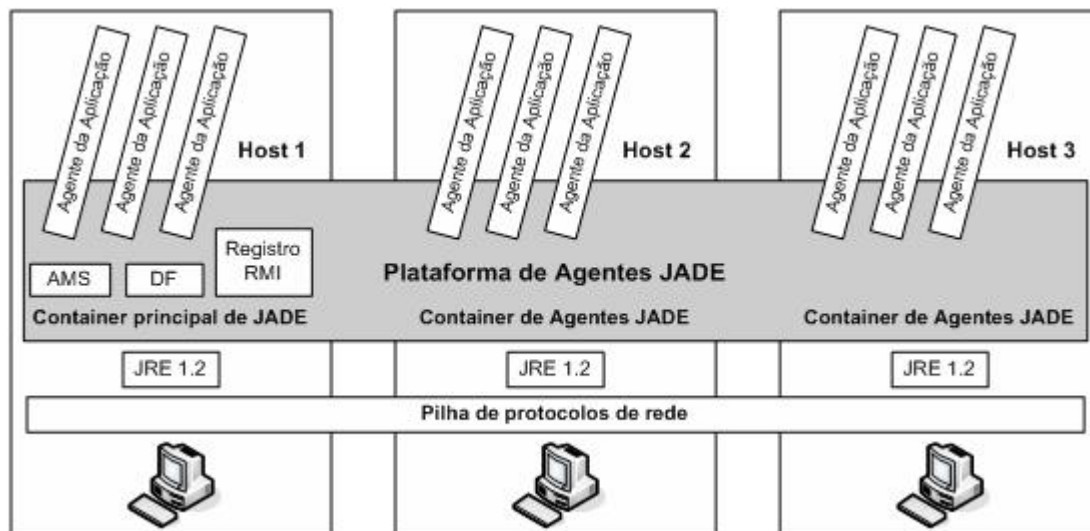


Figura 2.4: Arquitetura da plataforma JADE [WQoS03].

Ao iniciar a plataforma, os agentes recebem seus identificadores únicos globais (GUID – *Globally Unique Identifier*), sendo que o AMS registra automaticamente os agentes criados. O nome do agente é composto por um apelido, seguido do @ e do nome do *host.domínio*: porta/JADE; por exemplo agente@mycomputer:1099/JADE. Cada container é um ambiente de execução *multithreaded*.

Um agente é encarado pela plataforma como um objeto ativo que carrega consigo uma *thread* de controle. Ele é completamente autônomo (controla sua execução, decide quando ler as mensagens e quais ler), pode iniciar diversas conversações simultâneas e executar varias tarefas concorrentemente.

O programador usa os *behaviours* para modelar as ações que o agente é capaz de executar (usa-se uma *thread* por agente e não por *behaviour*, o que mantém razoável o número de *threads* necessárias para rodar a plataforma). Os *behaviours* trabalham em modo de agendamento cooperativo. Cada *behaviour* deve ceder o controle para permitir que os outros possam ser executados.

O *framework* já inclui uma biblioteca de protocolos de interação e de comportamentos genéricos, que podem ser customizados. JADE suporta mobilidade intra-plataforma e clonagem: os agentes podem migrar entre os *containers*, podem ser clonados entre os *containers*, podem ser iniciados pelo próprio agente (*doMove()*, *doClone()*) ou solicitados pela plataforma via AMS.

Na visão do programador, um agente JADE é simplesmente uma classe Java. Um agente é mapeado numa classe Java definida pelo usuário e as tarefas do agente são mapeadas nas subclasses definidas pelo usuário da classe *behaviour* em *jade.core.behaviours*.

Não há necessidade de implementar plataforma uma vez que o JADE inclui o AMS, o DF e o ACC que são iniciados automaticamente com a plataforma.

Os agentes da plataforma JADE podem estar em diversos estados de acordo com o ciclo de vida especificado pela FIPA. Os estados são [WQoS03]:

- **AP\_ INITIATED:** o *agent object* está construído, contudo ainda não se registrou com AMS, não tem nem um nome nem um endereço e não pode se comunicar com outros agentes;

- **AP\_ACTIVE:** o *agent object* está registrado com AMS, portanto já possui seu AID e pode ter acesso às várias características JADE;
- **AP\_SUSPENDED:** o *agent object* está atualmente parado, sua *thread* interna está suspensa e nenhum *behaviour* está sendo executado;
- **AP\_WAITING:** o *agent object* está bloqueado, esperando por algo. Sua *thread* está dormindo temporariamente.
- **AP\_DELETED:** o agente definitivamente está morto. Sua *thread* interna foi terminada e o agente não está mais registrado com AMS;
- **AP\_TRANSIT:** um agente móvel entra neste estado enquanto está migrando. O sistema continua armazenando as mensagens que serão enviadas a nova localidade;
- **AP\_COPY:** esse estado é usado internamente pelo JADE para estados que estão começando a ser clonados.
- **AP\_GONE:** esse estado é usado internamente pelo JADE quando o agente móvel migrou para uma nova localidade e possui um estado estável.

A relação entre esses estados pode ser mais bem entendida analisando o diagrama estado abaixo:

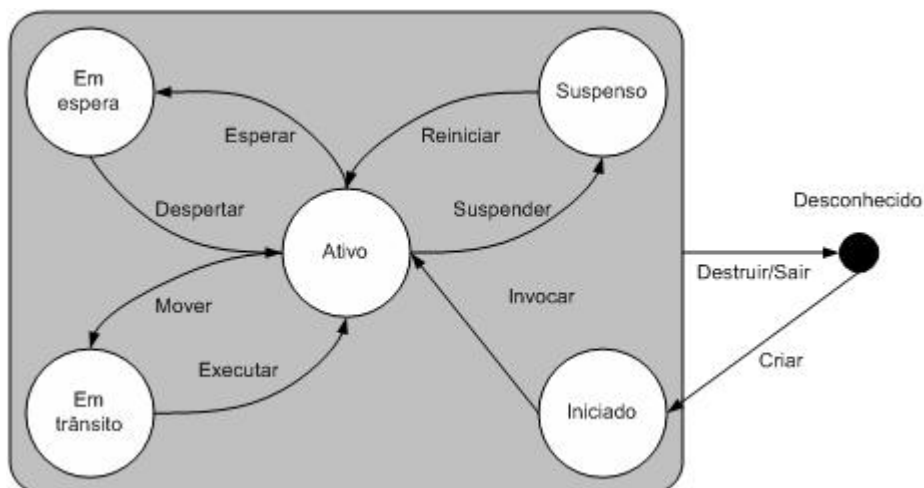


Figura 2.5: Diagrama de estados de agente definido pela FIPA.

JADE pode ser facilmente integrado com outras ferramentas, por exemplo, JESS (*Java Expert System Shell*). Que pode ser muito útil, pois pode funcionar como o motor que executa o raciocínio necessário para o agente.

Na comunicação entre agentes na plataforma JADE, os agentes se comunicam enviando mensagens individuais uns para os outros. Mensagem é entendida pelos agentes como atos falados e não chamadas. Os agentes enviam e recebem objetos Java, que representam as mensagens ACL, dentro do escopo dos protocolos de interação. JADE codifica transparentemente todas as mensagens. As linguagens utilizadas em cada nível são:

- Nível de envelope: baseada em String, baseada em XML;
- Nível de ACL: baseada em String, baseada em XML-based ou *bit-efficient*;

- Nível de CL: utiliza FIPA SL-0 mais a API para registrar o CL (Linguagem de Conteúdo), a serialização direta de objetos Java com codificação base 64;
- Nível de Ontologia: utiliza FIPA *agent management*, API para registrar os CLs definidos pelo usuário;

O padrão FIPA define que somente mensagens ACL são transportadas na plataforma, porém não define nenhum padrão para o conteúdo das mensagens. Cada agente possui uma fila privada de mensagens ACL a qual ele processa da forma que o mesmo achar mais interessante

JADE utiliza o mecanismo de transporte adaptável, ou seja, o protocolo de transporte de mensagem é escolhido de acordo com a situação. Tentando sempre obter menor custo possível de transmissão de mensagem

A arquitetura do software se baseia na coexistência de várias JVMs que podem estar espalhadas por vários *hosts*, sendo que cada VM é um container de agentes que provê um ambiente de *runtime* completo para execução de agentes e permite que vários agentes sejam executados concorrentemente no mesmo *host*.

As mensagens intra-container são transportadas por meio de eventos Java (*Java Events*), já a comunicação inter-containers é feita utilizando Java RMI, não é possível realizar a comunicação inter-containers caso haja *firewalls* entre os *hosts* que abrigam os agentes. A comunicação inter-plataforma é realizada utilizando o ACC em conjunto com um tradutor e o protocolo IIOP.

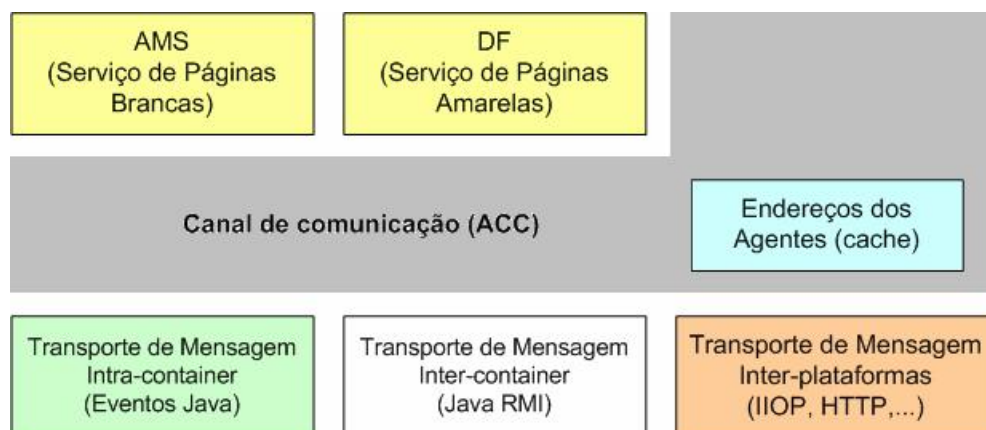


Figura 2.6: Elementos de comunicação na plataforma JADE.

O *front-end* da plataforma mantém internamente: os registros de RMI usados por outros *containers* ao se registrarem na plataforma; uma tabela de todos os *containers* registrados e de suas referências de objeto RMI (*Agent Container Table*); *Agent Global Descriptor Table* que relaciona cada nome de agente com seus dados AMS e suas referências de objeto RMI.

Cada *container* arquiva as referências de objetos dos outros *containers* em sua *cache* sempre que mensagens são enviadas, isso é feito para evitar que a *Agent Global Descriptor Table* seja consultada inúmeras vezes, melhorando assim o desempenho da plataforma (diminui o *overhead*). O *overhead* depende da localização do receptor e do status da *cache*. Quando agentes da mesma plataforma, porém residindo em diferentes *containers*, tentam se comunicar é necessário fazer atualização da *cache*, caso tal recurso não esteja disponível se faz necessário à atualização da *Agent Global*

*Descriptor Table*, esta comunicação com a *cache* e também com *Agent Global Descriptor Table* é realizada utilizando *Java events* (nas comunicações intra-plataforma não há necessidade de realizar atualizações). Só então a mensagem é enviada para o *message dispatcher* que utiliza o *Java RMI* para realizar comunicação com o outro container.

JADE possui uma extensão, JADEX (*Java Agent DEvelopment eXtension*) [JADEX04], que permite desenvolver agentes orientados a objetivos seguindo o modelo BDI.

O RMA de JADEX [Figura 2.7] é essencialmente um clone do RMA de JADE. O RMA estendido fornece ícones de atalho para iniciar as novas ferramentas dos agentes *Jadex Introspector* e *Logger*.

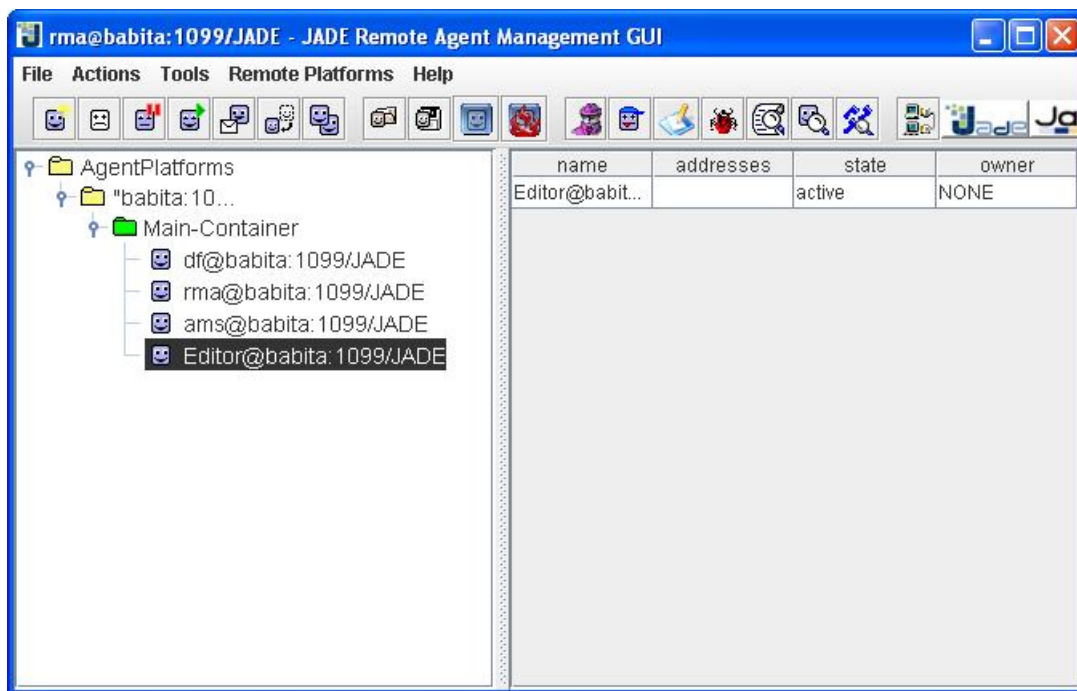


Figura 2.7: JADEX RMA.

A principal diferença entre o RMA de JADE e o de JADEX é uma opção para se iniciar os agentes. Com o RMA de JADEX, um arquivo de definição de agente (*agent definition file*, ADF) pode ser selecionado para execução. Este ADF é fornecido pela entrada do nome do arquivo no campo “Model (ADF)”, ou pela sua seleção do mesmo através do botão “Browser...”. Os outros parâmetros de inicialização são os mesmos de JADE. Quando o modelo do agente é carregado do ADF, o nome do da classe do agente aparece no container.

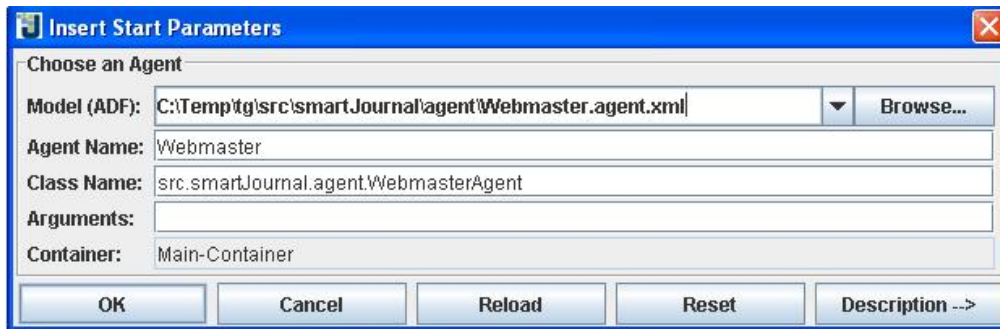


Figura 2.8: Janela de inicialização de agentes em JADEX.

O agente Introspector de JADEX é similar ao de JADE. Você pode usar a árvore de agentes do lado esquerdo para selecionar os agentes que você deseja observar. Nas janelas de observação aparecem quatro opções de aba: o “*Debugger*” cria opções de execução do agente; e as abas a “*Beliefbase*”, “*Goalbase*” e “*Planbase*”, que correspondem a visão BDI dos agentes, exibem os conteúdos das bases de crença, objetivos e planos respectivamente.

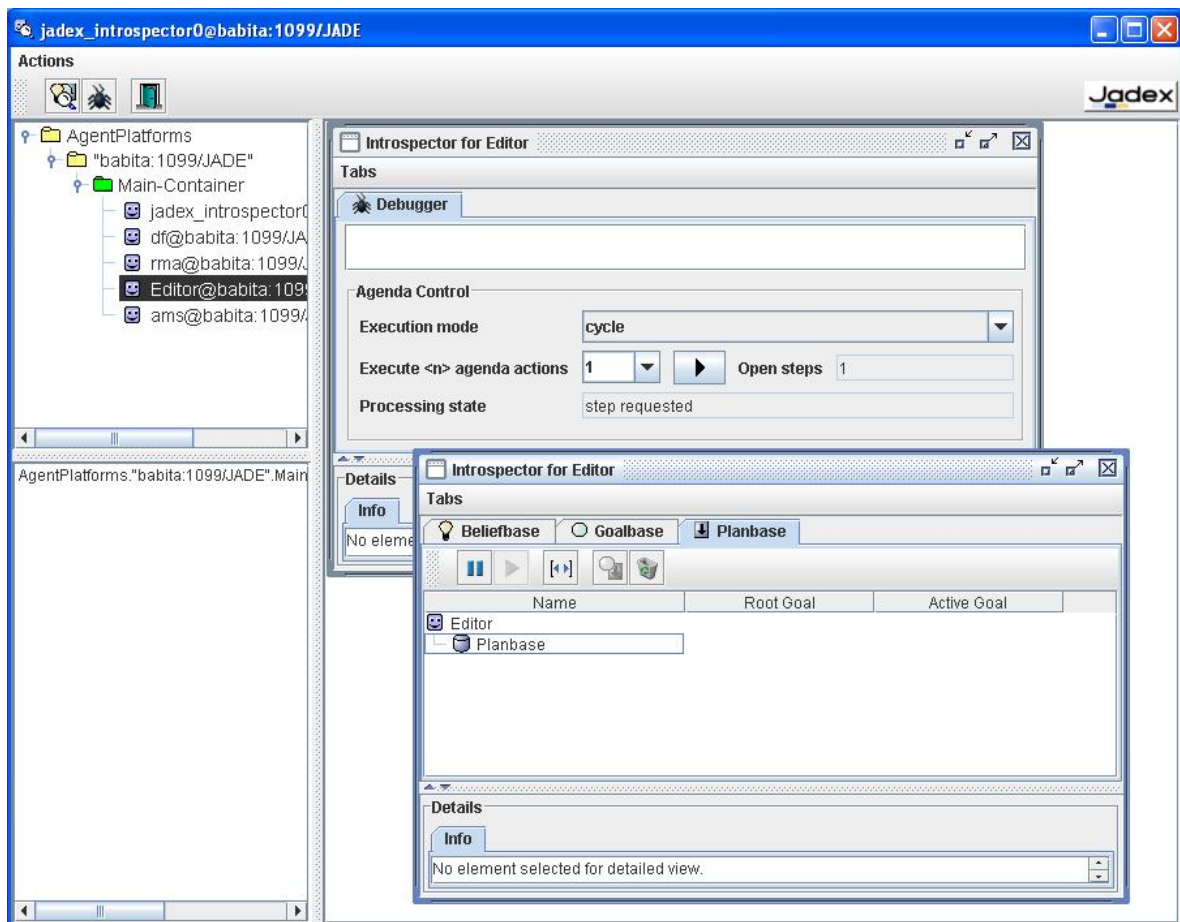


Figura 2.9: Introspector JADEX.

Os propósitos do *debugger* são: (i) primeiro, ele suporta a visualização e modificação dos conceitos internos do BDI permitindo a inspeção e reconfiguração de um agente em tempo de execução; (ii) segundo, simplifica o *debugg* através da facilidade de execução do agente passo a passo. Nesta etapa

é possível observar e controlar cada evento que processam e a etapa da execução do plano que detalha o controle sobre o *dispatcher* (expedidor) e o *scheduler*. Portanto ele pode ser facilmente compreender que planos estão selecionados por evento ou objetivo.

Um grande problema do agente *debugger* consiste na quantidade e seqüência de saídas que o agente produz no console. Com a ajuda do *logger* as saídas do agente podem ser direcionadas a um ponto único de responsabilidade em tempo de execução. O agente *logger* preserva informações de tempo e fonte (o agente e método). Usando esses artefatos o *logger* oferece facilidades para filtragem e classificação de mensagens por vários critérios, permitindo que uma visão personalizada seja criada.

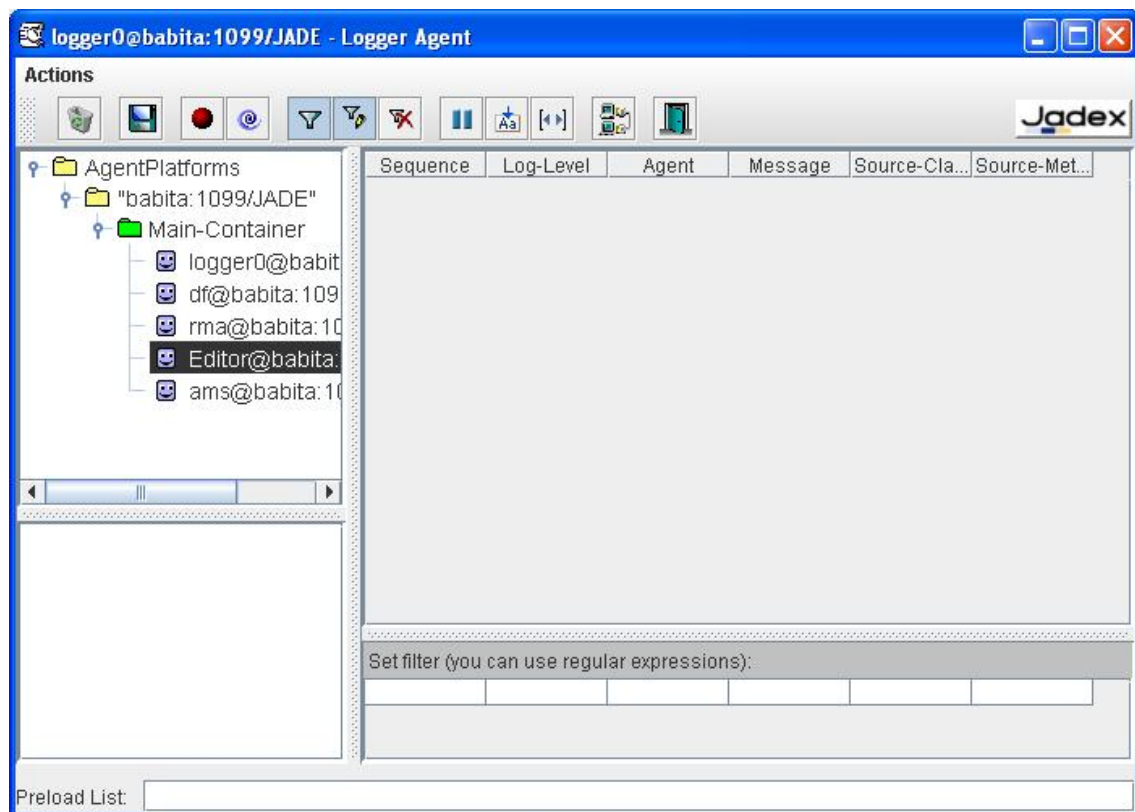
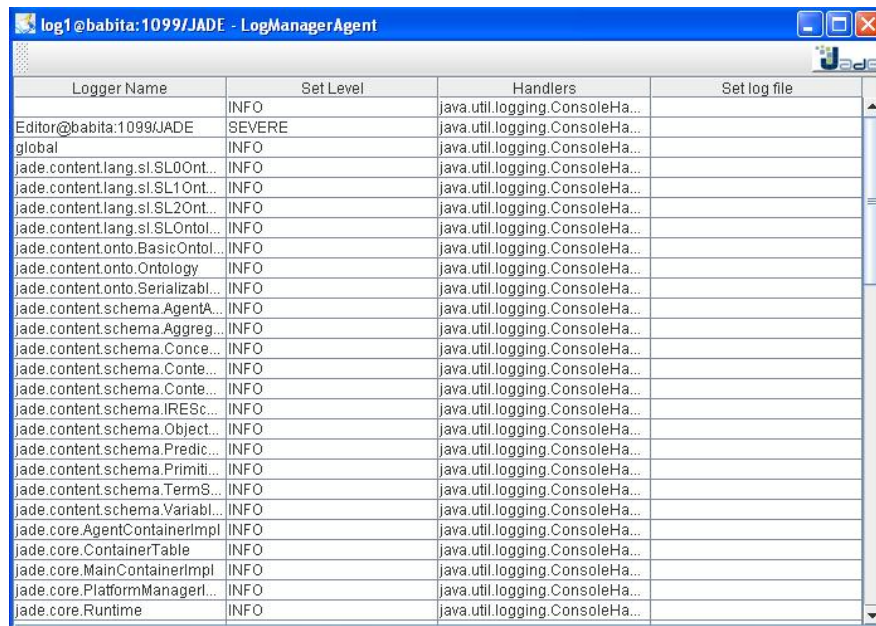


Figura 2.10: Logger JADEX.

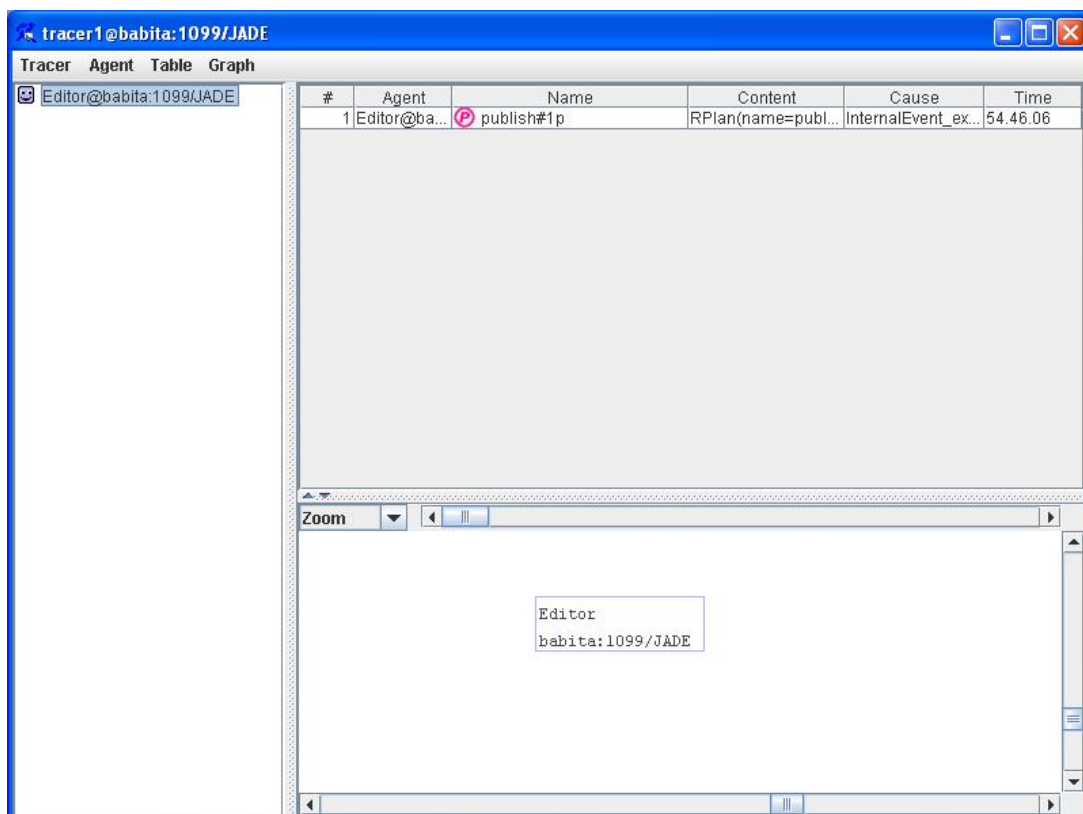




Logger Name	Set Level	Handlers	Set log file
Editor@babita:1099/JADE	SEVERE	java.util.logging.ConsoleHa...	
global	INFO	java.util.logging.ConsoleHa...	
jade.content.lang.sl.SLOnt...	INFO	java.util.logging.ConsoleHa...	
jade.content.lang.sl.SL1Ont...	INFO	java.util.logging.ConsoleHa...	
jade.content.lang.sl.SL2Ont...	INFO	java.util.logging.ConsoleHa...	
jade.content.lang.sl.SLOntol...	INFO	java.util.logging.ConsoleHa...	
jade.content.onto.BasicOntol...	INFO	java.util.logging.ConsoleHa...	
jade.content.onto.Ontology	INFO	java.util.logging.ConsoleHa...	
jade.content.onto.Serializabl...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.AgentA...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Aggreg...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Conce...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Conte...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Conte...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.IRESc...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Object...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Predic...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Primiti...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.TermS...	INFO	java.util.logging.ConsoleHa...	
jade.content.schema.Variabl...	INFO	java.util.logging.ConsoleHa...	
jade.core.AgentContainerImpl	INFO	java.util.logging.ConsoleHa...	
jade.core.ContainerTable	INFO	java.util.logging.ConsoleHa...	
jade.core.MainContainerImpl	INFO	java.util.logging.ConsoleHa...	
jade.core.PlatformManagerI...	INFO	java.util.logging.ConsoleHa...	
jade.core.Runtime	INFO	java.util.logging.ConsoleHa...	

Figura 2.11: Agente Log Manager.

O agente Tracer, fornece a análise e visualização das informações registradas pelo Logger. Ele é responsável por fornecer a análise das informações trocadas entre os agentes, armazená-las, e apresentá-las de forma gráfica.



#	Agent	Name	Content	Cause	Time
1	Editor@ba...	publish#1p	RPlan(name=publ...	InternalEvent_ex...	54.46.06

Zoom: [Slider]

Editor  
babita:1099/JADE

Figura 2.12: Tracer JADEX.



## Capítulo

# 3 Desenvolvendo SMA com Tropos

Este capítulo aborda o desenvolvimento de Sistemas Multi-Agentes utilizando a proposta Tropos. Inicialmente introduziremos o *framework* para modelagem organizacional *i\**. Em seguida, apresentaremos todas as etapas referentes à metodologia Tropos. Por fim, apresentaremos as considerações que nos levaram a escolher esta metodologia para o desenvolvimento do nosso estudo de caso.

### 3.1 Introdução

A construção de sistemas abertos, capazes de tratar informações distribuídas, dinâmicas e heterogêneas, onde a maioria desses sistemas de software existe em ambientes organizacionais e operacionais que estão frequentemente mudando, gera a necessidade de um entendimento apropriado da organização pelos projetistas do sistema, bem como às mudanças organizacionais que não conseguem ser acomodadas pelos sistemas de software existentes.

Desta forma a Engenharia de Requisitos vem sendo conhecida como a fase mais crítica do processo de desenvolvimento de sistemas porque, para construção de um *software* com qualidade, é preciso que as considerações técnicas estejam em equilíbrio com as sociais e organizacionais desde a modelagem de sistemas [Bresciani04]. O *framework i\** foi proposto em [Yu95] e possui uma estrutura conceitual rica, capaz de reconhecer motivações, intenções e raciocínios sobre as características de um processo, o que facilita os esforços da Engenharia de Requisitos.

A metodologia Tropos por sua vez inspirada na análise de requisitos iniciais e fundamentada em conceitos sociais e intencionais [Castro02], que adota o *framework i\** em suas fases iniciais de modelagem e utiliza seus conceitos durante todas as fases de desenvolvimento, tem como objetivo o desenvolvimento de sistemas estudados de acordo com as reais necessidades de uma organização, buscando um melhor casamento entre o sistema e o ambiente, e permitindo uma melhor estruturação de sistemas adaptáveis.

Nas próximas seções detalharemos este *framework* de modelagem organizacional, *i\**, e a metodologia Tropos.

### 3.2 O *framework i\**

O *framework i\** [Yu05] (que simboliza “intencionalmente distribuído”) é um *framework* de modelagem conceitual desenvolvido para modelagem e análise, sob uma visão estratégica e intencional, de processos que envolvem vários participantes, ou seja, uma modelagem organizacional. Para realizar

esta análise, os engenheiros de requisitos modelam os *stakeholders* como atores e suas intenções como metas. Cada meta é analisada do ponto de vista do seu ator, resultando em um conjunto de dependências entre pares de atores. Desta forma, o ambiente do sistema e o sistema em si são visto como organizações de atores, que dependem da ajuda de outros atores para que suas metas sejam cumpridas.

O  $i^*$  é formado por dois modelos: o modelo de Dependência Estratégica (SD, *Strategic Dependency model*) e o modelo de Razão Estratégica (SR, *Strategic Rationale model*). O modelo SD fornece uma descrição intencional de um processo em termos de uma rede de relacionamentos de dependência entre atores relevantes, ou seja, as relações de dependências externas entre os atores da organização. O modelo SR, por sua vez, apresenta uma descrição estratégica do processo, em termos de elementos do processo e das razões que estão por detrás deles, ou seja, fornece uma análise meio-fins de como as metas podem ser cumpridas através das contribuições dos demais atores.

A estrutura conceitual do *framework*  $i^*$  é utilizada para: (i) obter uma compreensão mais apurada sobre os relacionamentos da organização, de acordo com os diversos atores do sistema; (ii) compreender as razões envolvidas nos processos de decisões; e (iii) ilustrar as várias características de modelagem que podem ser apropriadas à Engenharia de Requisitos. Além de auxiliar especialmente as fases iniciais de especificação de requisitos, o  $i^*$  pode ser aplicado em outras áreas, tais como reengenharia de processos de negócio, análise de impactos organizacionais e modelagem de processos de software [Yu95].

### 3.2.1 O Modelo de Dependência Estratégica

No modelo SD, capturamos as motivações e os desejos dos atores que fazem parte da organização e apresentamos a sua rede de relacionamentos. Dado um modelo SD, podemos perguntar que novos relacionamentos entre os atores são possíveis, identificar a viabilidade ou não das dependências, ou ainda relacionar os desejos de um agente com as habilidades do agente do qual depende, para poder explorar as oportunidades disponíveis a esses atores.

De acordo com o *framework*  $i^*$  [Yu05], um ator é considerado uma entidade ativa que realiza ações para atingir metas, exercitando seu *know-how*. O termo ator é utilizado para referenciar genericamente qualquer unidade para a qual dependências intencionais possam ser atribuídas. Atores podem ser considerados: intencionais, por possuírem motivações, intenções e razões por trás de suas ações; e estratégicos, quando não focam apenas o seu objetivo imediato, mas quando se preocupam com as implicações de seu relacionamento estrutural com outros atores.

A Figura 3.1 apresenta os elementos usados para a modelagem com  $i^*$ .

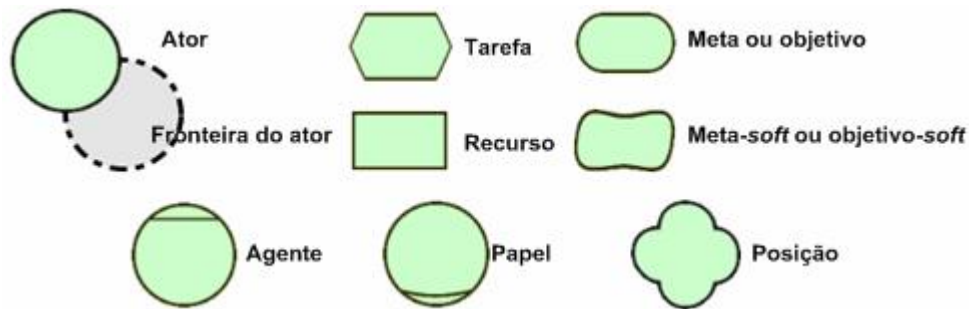


Figura 3.1: Elementos de modelagem com i\*.

Cada dependência em i\* é um relacionamento intencional, i.e., um “acordo” (chamado *dependum*), entre dois atores: o *dependor* e o *dependee*. O *dependee* é o ator que depende de um outro ator (*dependor*) para que o acordo (*dependum*) possa ser realizado. Os relacionamentos de dependência usados em i\* descrevem a natureza do acordo e podem ser de quatro tipos: tarefas, recursos, meta ou meta-soft. Quando o ator *dependee*, numa relação de dependência, disponibilizar um recurso necessário, executar uma tarefa de sua responsabilidade, atender a um objetivo do ator dependente, ou realizar alguma tarefa para alcançar uma meta-soft, teremos a relação de dependência satisfeita pelo *dependee* responsável por ela. A Figura 3.2 apresenta os tipos de ligações de dependência do *framework* i\*.

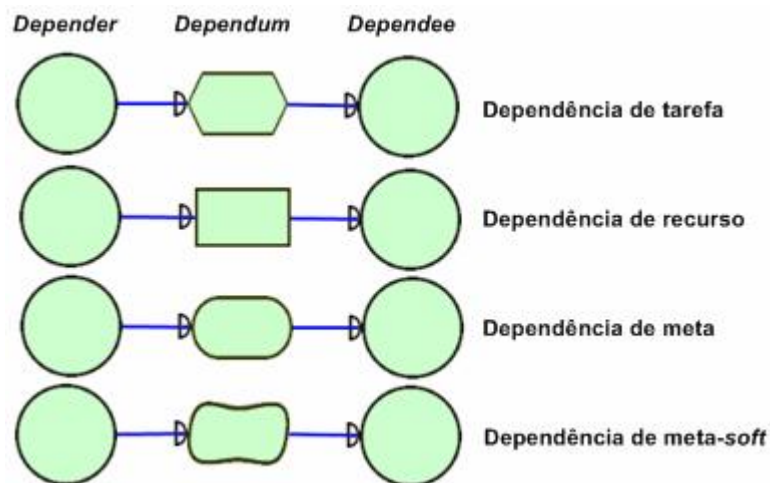


Figura 3.2: Tipos de relacionamento de dependência entre atores no i\*.

Uma tarefa especifica uma forma particular de se fazer algo. Tarefas podem ser vistas como soluções que provêem operações, processos, representações de dados, estruturação, restrições e agentes para atender às necessidades estabelecidas nas metas e metas-soft (ou *softgoals*). Na dependência de tarefa, o *dependee* é requisitado a executar uma dada atividade, sendo informado sobre o que deve ser feito. Contudo, a descrição de uma tarefa em i\* não tem por intenção ser uma completa especificação dos passos necessários à execução dessa tarefa, nem há preocupação em se informar o “porque” da solicitação de sua realização.

Na dependência de recurso, o agente depende da disponibilidade de uma entidade física ou de uma informação. Por recurso entendemos o produto final de alguma ação, em um processo, que estará ou

não disponível para o ator dependente. Neste tipo de dependência assume-se que não haja aspectos pendentes a serem tratados ou decisões a serem tomadas.

Por meta ou objetivo se entende uma condição ou estado do mundo que um ator gostaria de alcançar. Na dependência de meta, um ator depende de outro para que uma determinada condição seja satisfeita, não importando a maneira com a qual haverá satisfação. Geralmente uma meta é expressa como um desejo. De forma similar à dependência de meta, a dependência de meta-*soft* ou objetivo-*soft* também representa um desejo a ser satisfeito, exceto pelo fato de que não há um critério claramente definido de verificação se a condição desejada foi atingida. A sua satisfação depende do julgamento subjetivo e interpretação dos *stakeholders*.

Os atores complexos podem ser diferenciados em três tipos de noções especializadas de atores [Figura 3.3]: papéis, agentes e posições. Quando esta distinção é feita, a análise torna-se mais detalhada e exige uma atenção maior. Os mapeamentos de agente/papel/posição são úteis para modelagem do agrupamento de um grande número de dependências que agentes no mundo real geralmente possuem [Yu95], estes agentes podem executar diversos papéis e participar em vários processos.

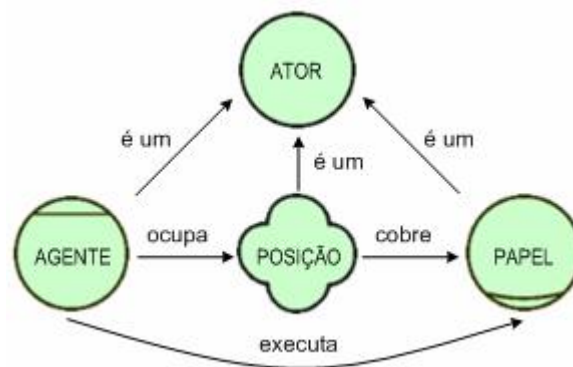


Figura 3.3: Associações e representação de agentes, papéis e posições.

A Figura 3.3 ilustra a representação gráfica e as possíveis associações entre agentes, papéis e posições. Dizemos que um agente executa um determinado papel e ocupa uma determinada posição, que cobre um papel. A seguir são definidos os tipos de atores:

- Agente: é um ator que possui manifestações físicas concretas. Tanto pode referir-se a humanos quanto a agentes artificiais de software ou hardware.
- Papel: representa a caracterização abstrata do comportamento de um ator social dentro de determinados contextos sociais ou domínio de informação. Apresenta as funções que podem ser exercidas por um agente dentro da organização.
- Posição: representa uma entidade intermediária entre um agente e um papel. É o conjunto de papéis tipicamente ocupados por um agente, ou seja, representa uma posição dentro da organização onde o agente pode desempenhar várias funções.

### 3.2.2 O Modelo de Razão Estratégica

Enquanto o modelo SD trata apenas dos relacionamentos externos entre os atores, o modelo SR é utilizado para descrever os interesses, preocupações e motivações dos atores participantes de um processo. Ele possibilita a avaliação das possíveis alternativas de definição do processo, investigando mais detalhadamente as razões existentes por trás das dependências entre os atores.

O modelo SR também é composto por nós e elos que, juntos fornecem uma estrutura para expressar as razões envolvidas no processo. Ele também utiliza quatro tipos de nós, que se baseiam nos tipos de dependências do modelo SD: recurso, tarefa, meta e meta-*soft*. Neste modelo, são apresentados outros dois tipos de relacionamentos que interconectam os nós: as ligações de meios-fins e as ligações de decomposição de tarefas.

Uma ligação de meios-fins indica um relacionamento entre um nó fim – que pode constituir uma meta a ser atingida, uma tarefa a ser realizada, um recurso a ser produzido ou uma meta-*soft* a ser satisfeita – e um meio para atingi-lo (Figura 3.4 (a)). Este tipo de ligação sugere alternativas existentes para se alcançar um determinado fim. Os meios geralmente são expressos em forma de tarefas, e os fins podem ser uma meta, um recurso, uma tarefa ou uma meta-*soft*. Contudo, existe a possibilidade de o meio e o fim serem ambas as meta ou meta-*soft*. Já as ligações de decomposição de tarefas ligam um nó de tarefa a seus nós componentes, que podem ser outras tarefas, metas, recursos ou metas-*soft* (Figura 3.4 (b)). Uma tarefa decomposta só pode ser considerada completa quando todos os seus nós componentes são cumpridos.

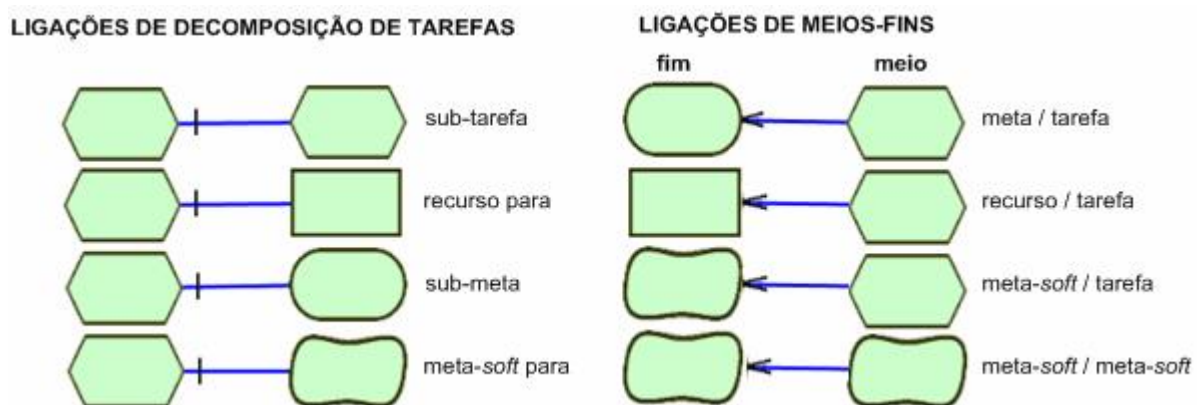


Figura 3.4: (a) Ligações de decomposição de tarefas (b) Ligações de meios-fins.

Para construção do modelo SR, é realizado um detalhamento, a partir do modelo SD, de alguns dos atores envolvidos no processo. Neste modelo podemos identificar no comportamento interno dos atores detalhados, ligações do tipo decomposição de tarefas e ligações do tipo meio-fins.

### 3.3 O framework NFR

O *framework* NFR foi proposto por Chung [Chung00], cuja abordagem é orientada a processo. Este fornece uma representação sistemática e global dos requisitos não-funcionais (RNF) tratando-os de forma explícita como metas a serem atingidas.

A Figura 3.5 mostra a visão geral do *framework*:

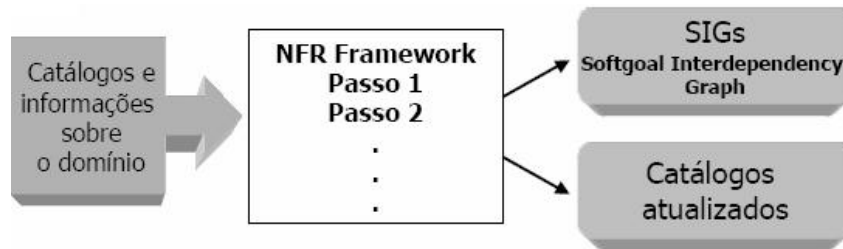


Figura 3.5: Visão geral do *framework* NFR.

Os principais conceitos envolvidos são:

- *Softgoal* – que representa uma meta que não tem definição clara nem um critério para decidir se está sendo satisfeito ou não;
- Catálogos – é um artefato produzido como saída pelo uso do *framework*, um catálogo armazena conhecimento acumulado em experiências prévias, é uma fonte de informação sobre RNFs [Figura 3.6];
- SIG (*Softgoal Interdependency Graph*) – grafo que representa o inter-relacionamento entre os requisitos não-funcionais [Figura 3.7]. O grafo é inspirado na estrutura de árvore E/OU para solução de problemas. Os *softgoals* são conectados por links de interdependência (positivas ou negativas) onde um *softgoal* é refinado em outros *softgoals*. À medida que os *softgoals* são refinados, deve-se decidir aceitar ou não as possíveis operacionalizações obtidas no grafo, ou seja, decidir se os *softgoals* estão suficientemente detalhados a ponto de se tomar decisões sobre o projeto.



Figura 3.6: Exemplo de catálogo.

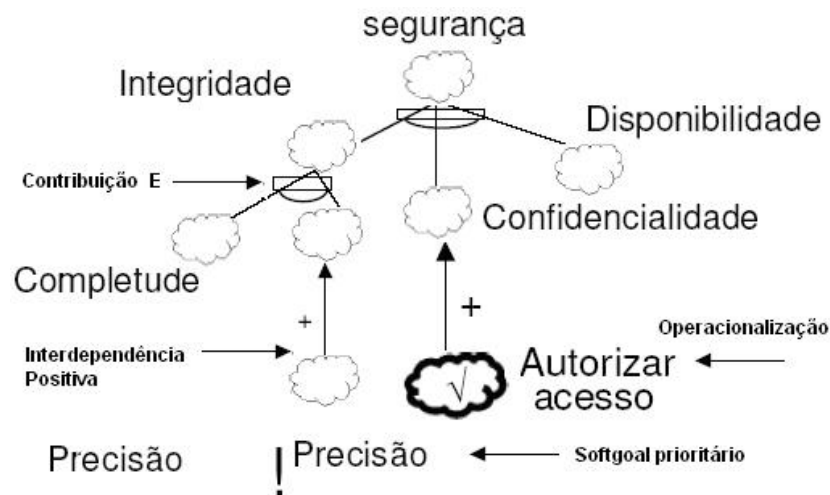


Figura 3.7: Exemplo de Grafo SIG.

A utilização do *framework* NFR [Chung00] consiste em uma série de passos, através dos quais os requisitos não funcionais são identificados, refinados, correlacionados com outros requisitos e operacionalizados. Esses passos ajudam a decompor os requisitos não funcionais, além de tratar ambigüidades e outros conflitos de especificação. De um modo geral, pode-se considerar que o *framework* possui três fases:

1. A criação prévia de catálogos relacionados com requisitos não funcionais, que servem para expressar o conhecimento sobre os Requisitos Não-Funcionais (RNF);
  - Catálogos de tipos de RNFs: são usados para fornecer uma terminologia e classificação de conceitos dos RNFs;
  - Catálogos de métodos, ou técnicas de operacionalização: possuem informações que ajudam a refinar os gráficos através da decomposição de *softgoals* e operacionalizações (estratégias de implementação);
  - Catálogos de correlação, ou interdependências entre RNFs: possuem conhecimento que ajudam a detectar interdependências implícitas entre os RNFs.
2. A definição dos gráficos relacionados com o problema em questão, a saída é um grafo SIG;
3. A seleção de alternativas e avaliação do impacto das decisões no problema sendo tratado, avaliando as contribuições e operacionalizações.

### 3.4 A metodologia Tropos

Tropos é uma proposta de desenvolvimento de sistemas orientada a agentes, inspirada na análise de requisitos e fundamentada em conceitos sociais e intencionais [Castro02] [Giorgini05]. Sua abordagem utiliza os modelos e conceitos oferecidos pelo *framework* i\* [Yu95]. Tanto o modelo SD quanto o modelo SR do i\* são usados por Tropos para capturar as intenções dos *stakeholders*, as responsabilidades do novo sistema em relação a estes *stakeholders*, a arquitetura do novo sistema e os detalhes de seu projeto.

As duas características fundamentais da metodologia Tropos são: (i) o uso de conceitos de nível de conhecimento [Newell93], tais como agente, meta, plano e outros, durante todas as fases do desenvolvimento de software; e (ii) o importante papel atribuído à análise de requisitos quando o ambiente e o sistema a ser desenvolvido são analisados [Castro02]. Além disso, Tropos propõe um *framework* de modelagem que visualiza o software sob cinco perspectivas complementares, [Bresciani04] [Giorgini05]:

- Social: quais são os atores relevantes, o que eles querem? Quais são suas obrigações? Quais são suas capacidades?
- Intencional: quais são as metas relevantes e como elas se relacionam? Como elas estão sendo reconhecidas e quem solicita as dependências?
- Comunicativa: como os atores dialogam e como eles podem interagir uns com os outros?
- Orientada a processos: quais são os processos de negócio ou computação relevantes? Quem é responsável pelo que?
- Orientada a objetos: quais são os objetos e classes relevantes, bem como seus relacionamentos?

Tropos oferece um *framework* que engloba as principais fases de desenvolvimento de software, com o apoio das seguintes atividades: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural e Projeto Detalhado. Recentemente o escopo do Tropos foi estendido passando a incluir técnicas para geração de uma implementação a partir dos artefatos gerados na fase de Projeto Detalhado. Esta fase complementa propostas para plataformas de programação orientada a agentes. Tropos visa, entre outros objetivos, a definição de arquiteturas de software mais flexíveis, robustas e abertas. Além disso, estão sendo realizados esforços para tornar a metodologia mais compatível com o paradigma de programação orientada a agentes, bem como para dar um melhor suporte a áreas de aplicação baseadas na *Web*, tais como telecomunicações e comércio eletrônico. Contudo, Tropos ainda não contempla explicitamente algumas atividades de desenvolvimento de software tradicional [Sommerville01], tais como testes, distribuição, gerência de configuração, entre outras.

O apoio ferramental necessário para tornar efetivo o desenvolvimento de software multi-agentes na metodologia Tropos é fundamental. Por isso, a ferramenta OME (*Organizational Modeling Environment*) [Yu05] que foi construída para dar suporte à construção dos modelos dos *frameworks* *i\** [Yu95] e NFR [Chung00], vem sendo continuamente evoluída, de acordo com os avanços alcançados pela proposta Tropos. Os demais modelos da fase de projeto detalhado podem ser construídos utilizando-se qualquer ferramenta de modelagem de diagramas da UML [Booch99]. A partir de descrições organizacionais expressas é possível a geração de modelos orientados a objetos. Ainda assim, outras ferramentas CASE podem ser utilizadas para apoiar a construção dos modelos do *framework i\**. Dentre elas destacamos as seguintes: GOOD, XGOOD e GOOSE.

A ferramenta GOOD (*Goals into Object Oriented Development*) [Cysneiros02] consiste de uma extensão da ferramenta Rational Rose [Rational99] e permite a integração com a OME com a finalidade de importar a especificação de modelos organizacionais produzidos pela OME e gerar o modelo de negócio através de diagramas de classes expressos em UML. A XGOOD (*eXtended Goal*



*Into Object Oriented Development*) [Alencar03, Pedroza04] é uma versão aprimorada da GOOD, que contém algumas outras funcionalidades e suporta mais diretrizes de mapeamento. Por fim, temos a ferramenta GOOSE (*Goal Into Object Oriented Standard Extension*), que também é uma extensão da ferramenta Rational Rose, a qual realiza a interpretação das descrições dos requisitos organizacionais modelados em i\* contidos nos modelos SD e SR e gera os diagramas de casos de uso da UML correspondentes, de acordo com diretrizes de mapeamento propostas em [Pedroza04].

A seguir, é apresentada cada uma das fases de desenvolvimento da metodologia Tropos.

### 3.4.1 Requisitos Iniciais

A fase de Requisitos Iniciais está preocupada com o entendimento de um problema estudando uma configuração organizacional existente [Silva03]. Durante esta fase, os engenheiros de requisitos modelam os *stakeholders* como atores e suas intenções como metas. Cada meta é analisada do ponto de vista de seu ator resultando em um conjunto de dependências entre pares de atores. As saídas desta fase são dois modelos:

1. O modelo de dependência estratégica que captura os atores relevantes, suas metas respectivas e suas interdependências; e
2. O modelo de razão estratégica que determina através de uma análise meio-fim como as metas podem ser cumpridas através das contribuições de outros atores.

### 3.4.2 Requisitos Finais

A fase de Requisitos Finais introduz o sistema a ser desenvolvido como um outro ator no modelo de dependência estratégica [Silva03]. O ator que representa o sistema é relacionado aos atores sociais em termos de dependências. Considera-se este ator que representa o sistema e se faz uma análise meio-fim para produzir um novo modelo de razão estratégica. Suas metas são analisadas e irão eventualmente levar a revisar e adicionar novas dependências com um subconjunto de atores sociais (os usuários). Se necessário decompõe-se o sistema que representa o ator em vários sub-atores e se revisa os modelos de razão e dependência estratégica.

### 3.4.3 Projeto Arquitetural

A fase de Projeto Arquitetural define a arquitetura global do sistema em termos de subsistemas, interconectados através de fluxos de controle de dados [Silva03]. A arquitetura de um sistema constitui um modelo da estrutura relativamente pequeno e intelectualmente gerenciável, que descreve como os componentes do sistema trabalham juntos. Subsistemas são representados como atores e interconexões de dado/controle são representados como dependências de ator (que representa o sistema). Esta fase consiste de três passos: 1. Selecionar o estilo arquitetural; 2. Refinar os modelos de razão e dependência estratégica e; 3. Aplicar padrões sociais.

Passo 1. Escolher o estilo arquitetural usando como critério as qualidades desejadas que foram identificadas na fase anterior, o catálogo de correlação entre os estilos arquiteturais

organizacionais e requisitos não-funcionais, e o *framework* NFR [Chung00] para conduzir esta análise de qualidade. Incluir novos atores, de acordo com a escolha de um estilo arquitetural específico de agente;

Passo 2. Incluir novos atores e dependências, assim como decompor os atores e as dependências existentes em sub-atores e sub-dependências. Revisar os modelos de razão e dependência estratégica. As capacidades de ator são identificadas da análise das dependências indo e vindo do ator, bem como das metas e planos que o ator irá executar a fim de cumprir requisitos funcionais e não-funcionais.

Passo 3. Definir como as metas associadas a cada ator são cumpridas por agentes com respeito a padrões sociais. Eles são usados para solucionar uma meta específica que foi definida ao nível arquitetural através da identificação de estilos organizacionais e atributos de qualidade relevantes (metas-*soft* ou *softgoals*). Uma análise detalhada de cada padrão social permite definir um conjunto de capacidades associadas com os agentes envolvidos no padrão. Uma capacidade estabelece que um ator é capaz de agir a fim de atingir uma meta dada. Em particular, para cada capacidade o ator tem um conjunto de planos que podem aplicar em diferentes situações. Um plano descreve a seqüência de ações a executar e as condições sob o qual o plano é aplicável. Capacidades são coletadas num catálogo e associadas ao padrão. Isto permite definir os papéis e capacidade do ator que são adequados para um domínio particular.

Os padrões sociais de Tropos estão classificados em duas categorias [Kolp02]: padrões entre pares e padrões de mediação. Entre os padrões pares podemos citar: *booking*, *call-for-proposal*, *subscription*, e *bidding*. Estes padrões sociais descrevem interações diretas entre agentes negociadores. Por sua vez, os padrões de mediação são compostos por agentes intermediários que ajudam outros agentes a atingirem um acordo ou realizar troca de serviços. São eles: *monitor*, *broker*, *matchmaker*, *mediator*, *embassy*, e *wrapper*. Maiores detalhes sobre os padrões sociais podem ser encontrados em [Kolp05].

Tabela 3.1: Estilos arquiteturais organizacionais definidos em Tropos.

Estilo Arquitetural	Principais características
<b>Estrutura Plana</b> (Flat Structure)	Não possui estrutura fixa e assume que nenhum ator tenha controle sobre outro. Suporta autonomia, distribuição e evolução da arquitetura de um ator.
<b>Estrutura em 5</b> (Structure in 5)	Consiste dos componentes típicos (estratégicos e logísticos) geralmente encontrados em várias organizações.
<b>Pirâmide</b> (Pyramid)	Estrutura de autoridade hierárquica exercida com limites organizacionais, onde atores nos níveis mais baixos dependem daqueles nos níveis mais altos.
<b>União Estratégica</b> (Joint Venture)	Envolve acordos entre dois ou mais parceiros principais, relacionados com um gerente comum e com parceiros secundários.
<b>Oferta</b> (Bidding)	Abrange mecanismos de competitividade e os atores se comportam como se estivessem tomando parte em um leilão.
<b>Tomada de Controle</b> (Takeover)	Envolve a delegação total de autoridade e gerenciamento de dois ou mais parceiros para um único ator, semelhante ao estilo <i>União Estratégica</i> .
<b>Comprimento de Braço</b> (Arm's Length)	Implica em acordos entre atores independentes e competitivos, porém parceiros. Não há delegação de autoridade.
<b>Contratação Hierárquica</b> (Hierarchical Contracting)	Identifica mecanismos de coordenação que combinam características do acordo <i>Comprimento de Braço</i> com aspectos de autoridade <i>Pirâmide</i> .

<b>Integração Vertical</b> (Vertical Integration)	Consiste de atores comprometidos em atingir metas ou realizar tarefas relacionadas em estágios diferentes de um processo de produção.
<b>Apropriação</b> (Co-optation)	Envolve a incorporação de agentes externos na estrutura ou no comportamento tomador de decisão ou conselheiro de uma organização iniciante.

Os estilos arquiteturais organizacionais são estruturas genéricas que podem ser instanciadas para projetar a arquitetura de uma aplicação específica. O estilo arquitetural União Estratégica, apresentado na Figura 3.8, é uma meta-estrutura que define um sistema organizacional que envolve acordos entre dois ou mais parceiros principais, com o intuito de obter benefícios derivados da operação em larga-escala e reuso da experiência de colaboração com demais parceiros. Cada parceiro principal, em uma dimensão local, pode gerenciar e controlar a si próprio (são autônomos) e interagir diretamente com outros parceiros principais para trocar, prover e receber serviços, dados e conhecimento. É feita a delegação de autoridade a um ator específico que será responsável pelo gerenciamento comum, coordenando tarefas e operações e gerenciando o compartilhamento de informações e recursos. Em uma dimensão global, a operação e coordenação estratégica de tal sistema e dos atores parceiros no sistema apenas são garantidas pelo gerente comum. Parceiros secundários fornecem serviços ou tarefas de suporte para o núcleo da organização.

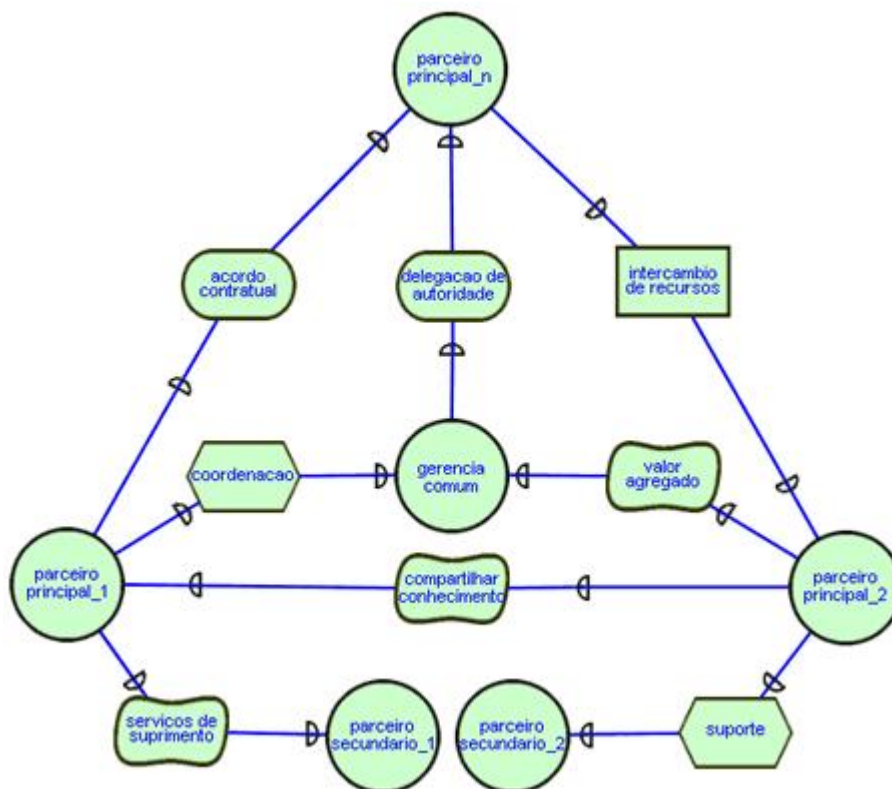


Figura 3.8: Estilo organizacional arquitetural União Estratégica (Joint Venture).

### 3.4.4 Projeto Detalhado

A fase Projeto Detalhado visa especificar o nível micro de agente, detalhando capacidades e planos usando o diagrama de atividades, como também protocolos de comunicação e coordenação

(modelados, por exemplo, em versões estendidas da UML, tais como AUML [Odell01]) [Silva03]. Nesta etapa ocorre:

1. O desenvolvimento de diagramas de atividades para representar as capacidades de cada agente;
2. O desenvolvimento de diagramas de seqüência e colaboração para capturar as interações entre os agentes;
3. A elaboração dos diagramas de atividade baseados em estado para representar cada plano identificado nos diagramas de capacidade.

### 3.4.5 Implementação

A fase de implementação segue passo a passo, em uma maneira natural, a especificação de projeto detalhado que é transformada em um esqueleto para a implementação. Isto é feito através de um mapeamento entre os conceitos Tropos e os elementos da plataforma de implementação de agente escolhido, tal como JACK [Busetta01], ou JADE [Bellifemine03]. O código é adicionado ao esqueleto usando a linguagem de programação suportada pela plataforma de programação.

No caso do esqueleto gerado pra JACK, o restante do código do sistema deve ser implementado em Java.

Da mesma forma que diversas ferramentas CASE, tal como Rational Rose [Rational99], incluem geradores de código para padrões de projeto orientados a objetos, a ferramenta SKwyRL [SKwyRL03] propõe um gerador de código para automatizar o uso dos padrões de projeto orientados a agentes propostos por Kolp [Kolp05]. SKwyRL foi desenvolvida com a linguagem Java [Deitel03] e produz o código genérico para os padrões, gerando arquivos (*.agent*, *.event*, *.plan*, *.bel*) para a plataforma de desenvolvimento orientado a agentes JACK. Para a plataforma de implementação de agentes JADE, ainda não existe um gerador de código que auxilie o mapeamento correspondente do Projeto Detalhado de Tropos para implementação em JADE ou JADEX.

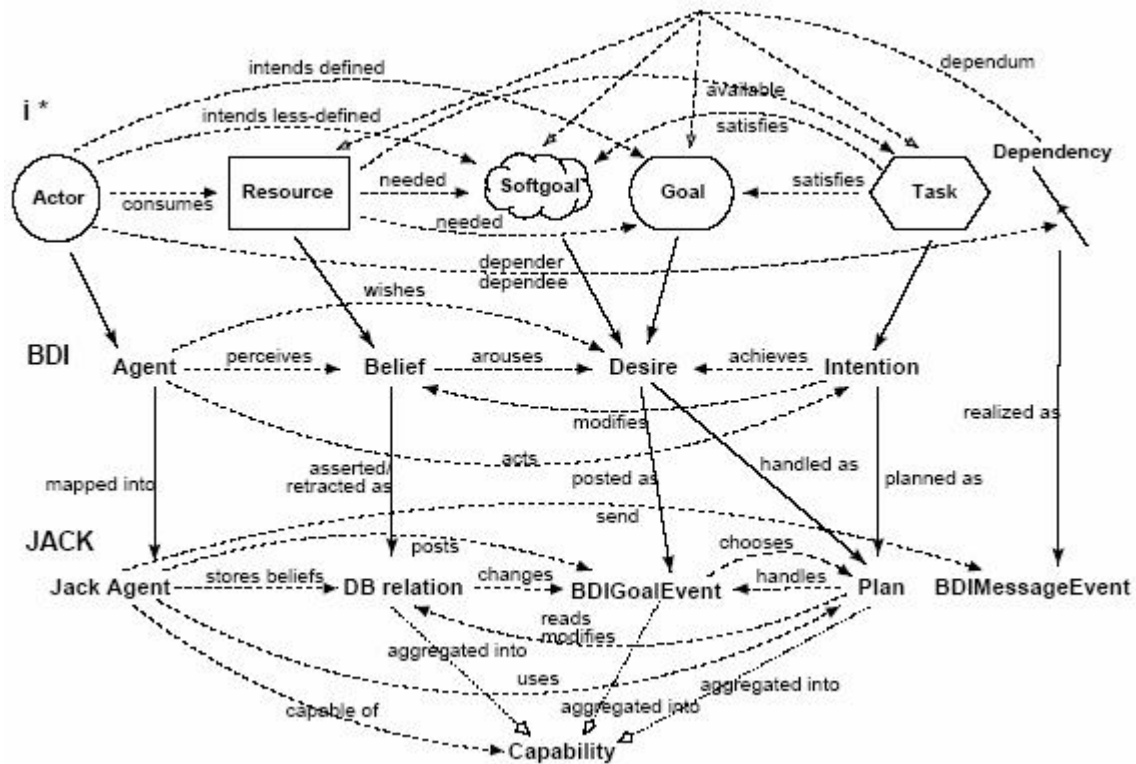


Figura 3.9: Visão geral do mapeamento i\*/BDI/JACK [Castro02a].

A Figura 3.9 apresenta a visão geral do mapeamento dos conceitos de i\* para os construtores de JACK, bem como os relacionamentos dos conceitos com outros dentro do mesmo modelo. Atores, recursos, metas-soft, metas e tarefas, são mapeados, respectivamente, para agentes, crenças, desejos e intenções do modelo BDI. Por sua vez, um agente BDI é mapeado como um agente de JACK, uma crença será declarada (ou retirada) como uma relação da base de dados, um desejo será publicado (emitido internamente) como um *BDIGoalEvent* (que representa um objetivo que um agente deseja conseguir) e manipulado como um plano, e uma intenção é implementada como um plano. Finalmente, uma dependência será realizada diretamente através de um *BDIMessageEvent* (recebido por agentes de outros agentes).

### 3.5 Considerações

Em resumo, de uma perspectiva de engenharia de software, a proposta Tropos, embora ainda em constante evolução, ainda apresenta as vantagens de levar a arquiteturas de software mais flexíveis, robustas e abertas e oferecer um *framework* coerente que engloba todas as fases de desenvolvimento de software, dos requisitos iniciais até a implementação.

Além disso, Tropos é consistente com a próxima geração de paradigma de programação, isto é, a programação orientada a agentes, que está ganhando uma base sólida em áreas de aplicação chave, tais como telecomunicações, comércio eletrônico e sistemas baseados em *web*.

## Capítulo

# 4 Desenvolvendo SMA com JADEX

Este capítulo aborda o desenvolvimento de Sistemas Multi-Agentes utilizando a plataforma JADEX. Inicialmente introduziremos a plataforma e em seguida, apresentaremos a arquitetura BDI de agentes, na qual JADEX se baseia e por fim, apresentaremos as características da plataforma: arquitetura e implementação, integração com JADE, como representamos os agentes, modelo de execução.

## 4.1 Introdução

Entre as plataformas de desenvolvimento descritas para implementação de SMA, escolhemos a JADE, e sua extensão JADEX, como apropriadas para executar a implementação de SMA com a metodologia Tropos. JADEX é um pacote construído para permitir o desenvolvimento de agentes de acordo com a FIPA e a arquitetura BDI de agentes.

Escolhemos JADE pelas suas características já listadas anteriormente. Dentre elas:

- As especificações de JADE estão de acordo com os padrões da FIPA.
- É uma plataforma distribuída de agentes.
- Apresenta transporte eficiente de mensagens ACL entre os agentes.

Escolhemos utilizar JADEX porque [JADEX04]:

- JADEX permite a construção de agentes de software seguindo os conceitos do modelo BDI (*Belief Desire Intention*);
- Usa tecnologias como XML e Java;
- JADEX é projetado para facilitar a implementação de agentes em Java, e, portanto permite o reuso de várias ferramentas e bibliotecas.

Um agente JADEX é também um agente JADE e, portanto todas as ferramentas disponíveis em JADE podem ser usadas também para desenvolver agentes em JADEX. A maior parte da plataforma JADE lida com a visão externa de um agente, que não difere entre um agente JADE ou JADEX. Apenas o agente *Introspector* de JADE tem seu uso limitado, porque ele exibe apenas os quarto comportamentos padrões e não os planos do agente.

## 4.2 O modelo BDI

No estudo de caso deste trabalho usaremos SMA baseado na arquitetura BDI de agentes. Esta arquitetura é baseada em um modelo de cognição fundamentado em três principais atitudes mentais que são as crenças, os desejos, e as intenções (abreviadas por BDI, *Beliefs Desires Intentions*). As

idéias básicas desta abordagem são: (i) descrever o processamento interno do estado de um agente utilizando um conjunto de categorias mentais (crença, desejo e intenções) e (ii) definir uma arquitetura de controle através da qual o agente seleciona racionalmente o curso de suas ações.

De forma esquemática e genérica, a arquitetura BDI pode ser apresentada como na Figura 4.1, como proposto por [Wooldridge95], e que está organizada na seguinte forma [Jesus03]:

- Crenças: são uma fundamental parte do estado mental do agente, representam o possível conhecimento do agente. Um agente pode ter crenças sobre o mundo, sobre crenças de outros agentes, sobre interações com outros agentes e sobre suas próprias crenças.
- Desejos: os desejos de um agente são um conjunto de metas a serem realizadas em um dado período de tempo. Uma meta é tipicamente uma descrição de um estado desejado do ambiente. Os desejos motivam o agente a agir de forma a realizar as metas, tais ações são realizadas através das intenções causadas pelos desejos.
- Intenções: as intenções representam seqüências de ações específicas que um agente se compromete a fazer para atingir um determinado objetivo (meta).
- Função de Revisão de Crença (FRC): percebe alterações no ambiente e, consultando as crenças anteriores do agente, atualiza estas crenças para que elas reflitam o novo estado do ambiente.
- Função Gera Opções: consultando quais as intenções com que o agente já está comprometido, verifica quais as novas possibilidades de coisas a serem feitas, e então uma deliberação deve ocorrer para a escolha de algumas destas novas opções com as quais o agente se comprometerá, atualizando então os desejos do agente.
- Função Filtro: atualiza o conjunto de intenções do agente com base nas crenças e desejos atualizados e nas intenções já existentes.
- Função Ação: determina, dentro de um conjunto de intenções, qual ação específica será realizada no ambiente pelo agente a cada momento.

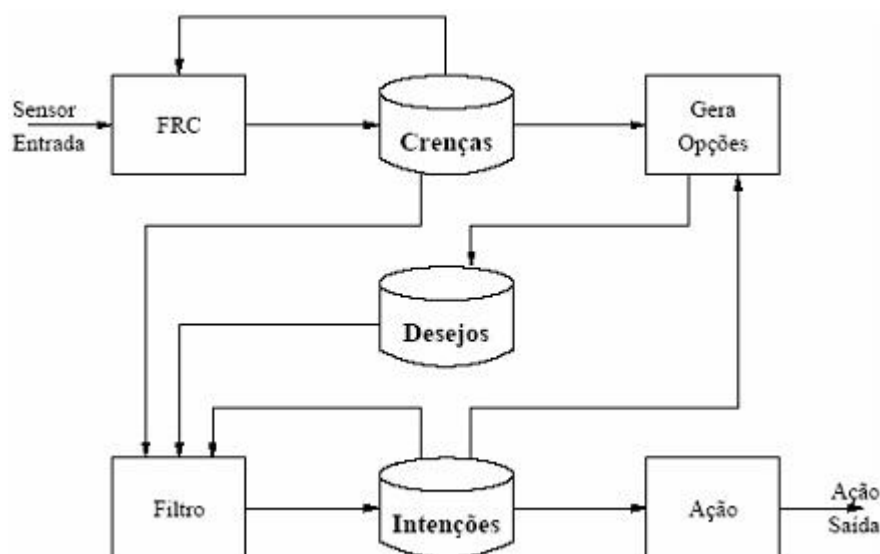


Figura 4.1: Arquitetura BDI Genérica (Adaptado de [Wooldridge95])

JADEX incorpora este modelo dentro dos agentes de JADE, introduzindo crenças, objetivos (metas) e planos como classes de objetos, que podem ser criados e manipulados dentro do agente. Em JADEX, agentes tem crenças, que podem ser de qualquer tipo de objeto Java e podem ser armazenadas numa base de crenças. Objetivos representam as motivações concretas (estados a serem alcançados) que influenciam um comportamento de um agente. Para alcançar seus objetivos o agente executa planos, que são um *guideline* codificado em Java e a ser seguido.

Na próxima seção explicamos esta arquitetura de JADEX.

### 4.3 JADEX: Arquitetura e Modelo de Execução

Na Figura 4.2, apresentamos uma visão geral da arquitetura BDI de JADEX. Visto de fora, um agente é uma caixa preta, o qual recebe e envia mensagens. As mensagens recebidas, bem como os eventos internos e os novos objetivos servem como entrada para a reação interna do agente e para o mecanismo de deliberação. Baseado nos resultados do processo de deliberação esses eventos são despachados para executar os planos. A execução dos planos pode acessar e modificar a base de crenças, enviar mensagens para outros agentes, criar novos objetivos ou sub-objetivos, e causar eventos internos [Braubach04].

O mecanismo de reação e deliberação é geralmente o mesmo para todos os agentes. O comportamento de agente específico é, portanto determinado apenas por suas crenças, objetivos, e planos. A seguir, os conceitos centrais desta arquitetura de JADEX serão descritos [Pokahr05].

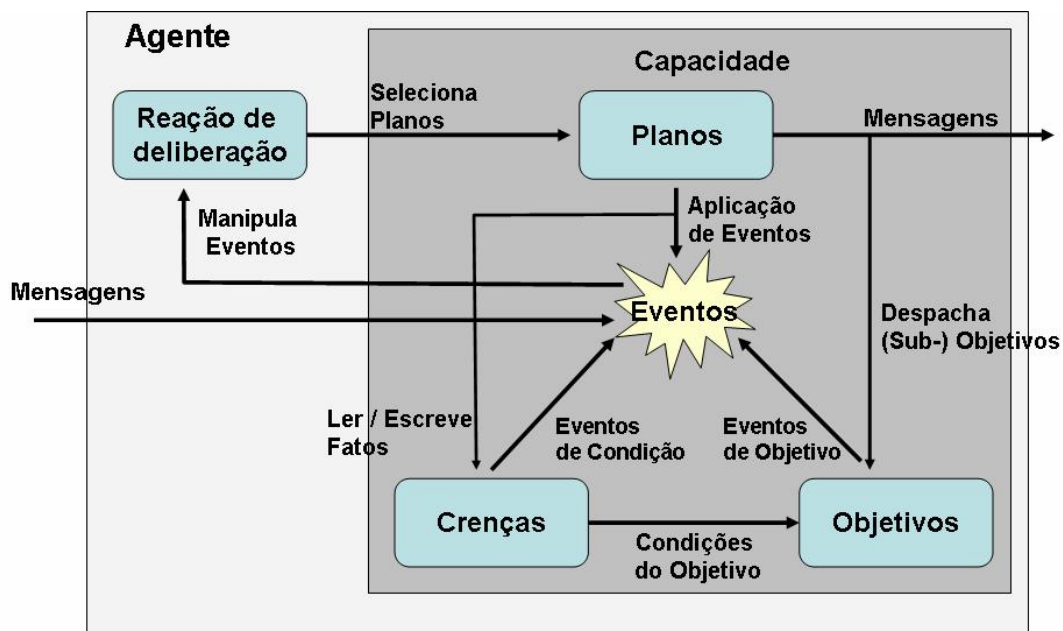


Figura 4.2: Arquitetura BDI de JADEX [Pokahr05].

[Pokahr05] A base crença armazena fatos acreditados e é um ponto de acesso aos dados contidos no agente. Portanto, ela fornece uma maior abstração e representa uma visão unificada do conhecimento do agente. Em JADEX, a representação da crença é muito simples, e atualmente não suporta qualquer mecanismo de inferência. A base de crença contém *strings* que representam um identificador para uma



crença específica (similar aos nomes de tabela em um modelo de dados relacional). Estes identificadores são mapeados em valores de crenças, fatos nomeados, que podem ser objetos Java. Atualmente duas classes de crenças são suportadas: crenças simples (*simple single-fact beliefs*), e um conjunto de crenças (*belief sets*). Crenças e conjunto de crenças são fortemente tipadas, e a base de crenças checa em tempo de execução, que somente os objetos corretamente tipados são armazenados. A essa representação de crença, JADEX adiciona diversas características, tais como uma linguagem de consulta OQL-like (adotada das bases de dados objeto-relacional), condições que disparam planos ou objetivos quando alguma crença muda, e crenças que estão armazenadas como expressões e avaliadas dinamicamente sob demanda.

[Pokahr05] Diferentemente dos sistemas BDIs tradicionais, que tratam objetivos meramente como um tipo especial de evento, em JADEX os objetivos são um conceito central. JADEX segue a idéia geral de que objetivos são concretos, desejos momentâneos de um agente. Para qualquer objetivo que o agente tenha ele irá se engajar mais ou menos diretamente nas ações apropriadas, até que ele considere o objetivo como sendo alcançado, não alcançado, ou não mais desejado.

Diferentemente da maioria dos sistemas, JADEX não assume que todos os objetivos adotados precisam ser consistentes uns com os outros. Para distinguir entre objetivos adotados e objetivos perseguidos, um ciclo de vida do objetivo é introduzido, que consiste nos estados do objetivo opção, ativo, e suspenso [Figura 4.3].

Quando um objetivo é adotado, ele se torna uma opção que é adicionada à estrutura de desejo do agente. Os mecanismos de deliberação de um objetivo específico são responsáveis por gerenciar a transição de estados de todos os objetivos adotados (por exemplo, decidir que objetivos são ativos e quais são apenas opções). Além disso, alguns objetivos podem apenas serem válidos em contextos específicos determinados pelas crenças do agente. Quando o contexto de um objetivo é inválido, este será suspenso até o contexto ser válido novamente.

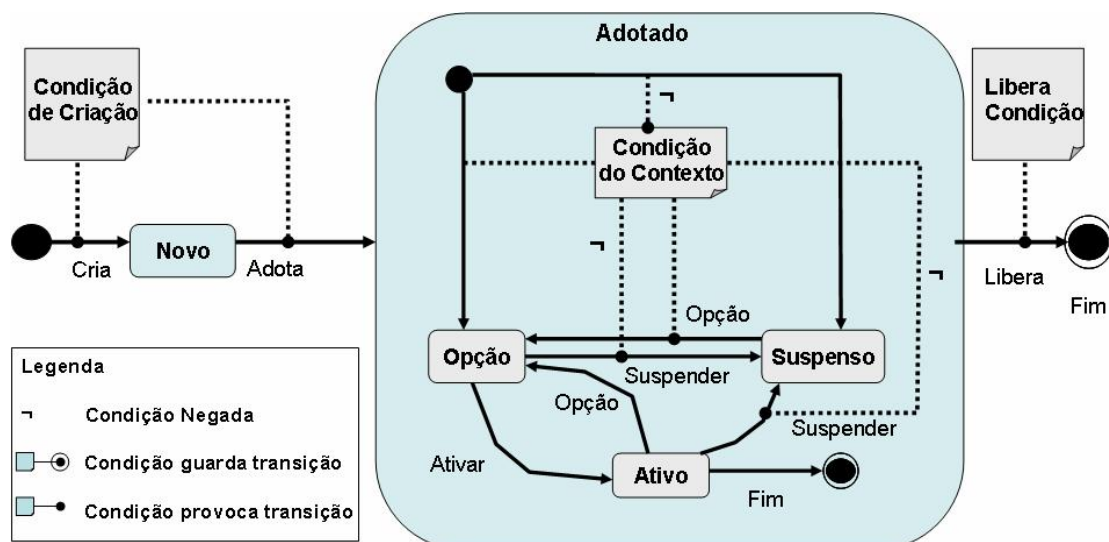


Figura 4.3: Ciclo de Vida do Objetivo [Pokahr05].

Quatro tipos de objetivos são suportados por JADEX: objetivos de *Executar*, *Conseguir* (atingir), *Consultar* (query), e *Manter*. O objetivo de *executar* indica que algo deveria ser, mas não pode

necessariamente levar a nenhum resultado específico. O objetivo de *conseguir* descreve um estado alvo abstrato a ser alcançado, sem especificar como consegui-lo. Portanto, um agente pode tentar diferentes alternativas para alcançar o objetivo. Considere um agente jogador que precisa de certos recursos num jogo de estratégia: ele poderia escolher negociar com outros jogadores ou tentar achar os recursos requeridos ele mesmo. O objetivo de *consultar* representa uma necessidade de informação. Se a informação não está disponível de imediato, planos são selecionados e executados para recolher informação necessária. O objetivo de *manter* especifica um estado que poderia ser mantido uma vez que este fosse alcançado. Essa é a maioria dos objetivos abstratos em JADEX: ele não só abstrai das ações concretas requeridas para alcançar o objetivo, como também desacopla a criação e adoção do objetivo do ponto em que ele é executado [Pokahr05].

Em JADEX, objetivos são representados com objetos com diversos atributos. O estado alvo dos objetivos de *conseguir* podem ser explicitamente especificados por uma expressão, que é válida para checar se o objetivo é alcançado. Os atributos de um objetivo, tais como nome, facilitam a seleção do plano, por exemplo, especificando que o plano pode manipular todos os objetivos de uma dado nome. A estrutura dos objetivos atuais adotados é armazenada na base de objetivos do agente. O agente tem um número de objetivos de alto nível, que servem como pontos de entrada na base de objetivos. Objetivos por sua vez podem ter sub-objetivos, formando uma hierarquia ou árvore de objetivos [Pokahr05].

A principal funcionalidade dos agentes é capturar planos. O desenvolvedor do agente tem que definir o cabeçalho (*head*) e o corpo (*body*) de um plano. O cabeçalho contém as condições sobre as quais o plano pode ser executado e é especificado no arquivo de definição do agente (ver seção 4.4). O corpo do plano é um guia que descreve as ações a serem realizadas para alcançar um objetivo ou reagir a algum evento. A versão atual de JADEX suporta que os corpos dos planos sejam escritos em Java, fornecendo toda a flexibilidade da linguagem (programação orientada a objetos, acesso a outros pacotes, etc) [Pokahr05].

Em tempo de execução, os planos são instanciados para manipular os eventos e atingir os objetivos. Disparadores de ativação na cabeça dos planos são usados para especificar se um plano deverá ser instanciado quando certo evento ocorrer. Além disso, o tão chamado plano inicial executa assim que o agente nasce. Executar os planos cria filtros adicionais para esperar por eventos específicos, que disparam as etapas subsequentes dos planos [Pokahr05].

## 4.4 Os Agentes

Para criar e iniciar um agente o sistema precisa saber as propriedades do agente a ser instanciado. O estado de um agente é determinado pelas crenças, objetivos, e planos executados, bem como a biblioteca de planos conhecidos. A definição completa de um agente é capturada num ADF (*Agent Definition File*), um arquivo XML. O ADF é um tipo de descrição de classe para agentes: do ADF agentes são instanciados como objetos de suas classes.

No ADF, o desenvolvedor define as crenças e objetivos iniciais usando uma sintaxe parecida com Java para fatos e parâmetros iniciais do objetivo. Os planos são declarados especificando como instanciá-los a partir das suas classes Java. O disparador pode ser omitido no caso de um plano ser executado,

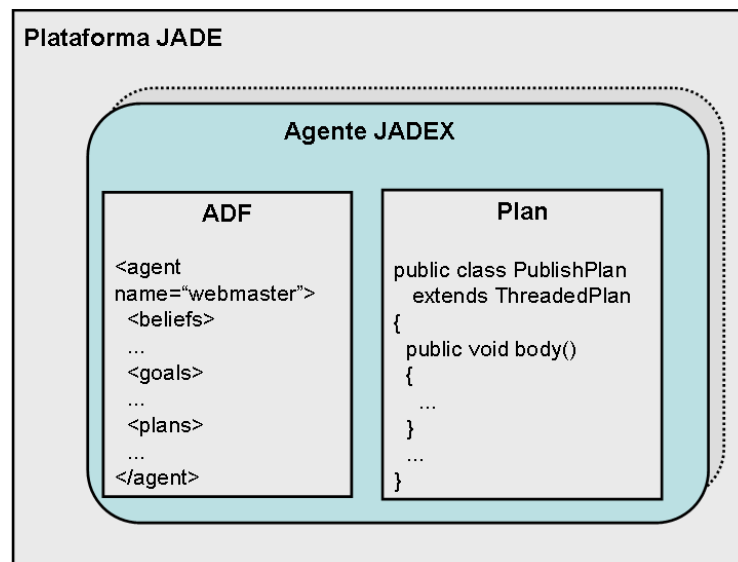
quando o agente inicia (plano inicial). Em adição aos componentes BDI, alguma outra informação é armazenada no ADF, por exemplo, serviço de descrição para registrar o agente no DF.

A declaração do cabeçalho de um ADF pode ser exemplificada como abaixo:

**Exemplo 4.1: Declaração de um ADF.**

```
<agent xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://jadex.sourceforge.net/jadex.xsd"
name="Editor" package="src.agentes"
class="jadex.runtime.impl.JadeWrapperAgent"
properties="jadex.config.runtime">
. . .
</agent>
```

A Figura 4.4 descreve como estes arquivos definem a funcionalidade de uma agente JADEX [Pokahr05].



**Figura 4.4: Componentes de um agente JADEX.**

Para iniciar os agentes JADEX, além de usar o RMA, JADE provê duas outras formas de iniciar novos agentes (como explicado na Seção 2.4.3). Estes dois mecanismos também são usados para agentes JADEX. A classe de um agente JADEX é usualmente um *JadeWrapperAgent*. Para iniciar o agente pela linha de comando deve-se ter que informar a localização do ADF como primeiro argumento do agente. Linha de comando:

**Exemplo 4.2: Linha de comando para iniciar o agente.**

```
java jade.Boot "myagent:jadex.runtime.impl.JadeWrapperAgent(mypackage.MyAgent)"
```

Para iniciar o agente de uma aplicação externa deve-se usar o mecanismo de carregamento modelo do agente, para ler os parâmetros de iniciais. Seguindo as instruções do guia do programador JADE [JADE04] em como criar um novo *container* de agentes e então executar o seguinte código:

**Exemplo 4.3: Criando um *container* de agentes.**

```

AgentContainer container;

. . .

// Carrega o modelo do agente, inicializa os parâmetros de início e
// dá o start no agente.

IMBDIAgent modelo =

    jadex.tools.model.SXML.loadAgentModel("mypackage.MyAgent", null);

String classe = model.getName();

// Qualquer string pode ser usada como nome do agente.

String nome = model.getAgentClass().getName();

Object[] args = new Object[]{modelo};

// Os argumentos (args) podem ser incluídos depois do modelo.

AgentController ac = container.createNewAgent(nome, classe, args);

ac.start();

```

**4.4.1 Mensagens e Protocolos**

As mensagens trocadas entre os agentes estão de acordo com o padrão da FIPA. E seguem o formato abaixo Figura 4.5:

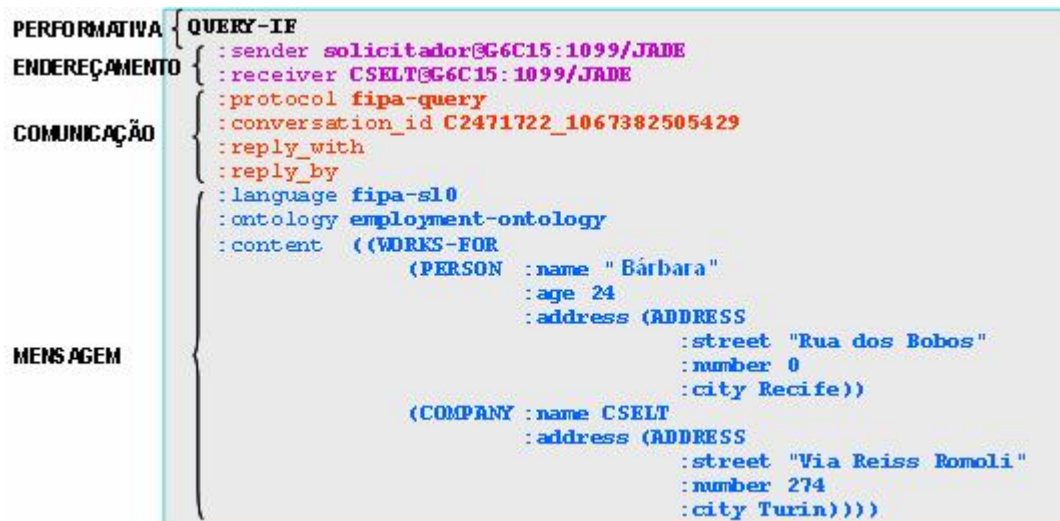


Figura 4.5: Formato das mensagens trocadas entre os agentes.

JADE fornece diversas implementações prontas para uso dos padrões dos protocolos de interação da FIPA. Com exceção do *FIPARequestInitiatorPlan*, não existe nada similar disponível ainda para JADEX.

#### 4.4.2 Integração com JADE

Para integrar JADEX aos agentes em JADE, uma classe de agente *wrapper* é fornecida, a qual cria e inicializa uma instancia de JADEX com as crenças, objetivos e planos de um ADF. Os componentes (crenças, objetivos e planos) são implementados em três comportamentos JADE, que são automaticamente criados e adicionados ao agente *wrapper*. Além disso, há um comportamento de sincronização com o propósito de adicionar eventos a lista de eventos (por exemplo, quando mensagens esperadas não chegam).

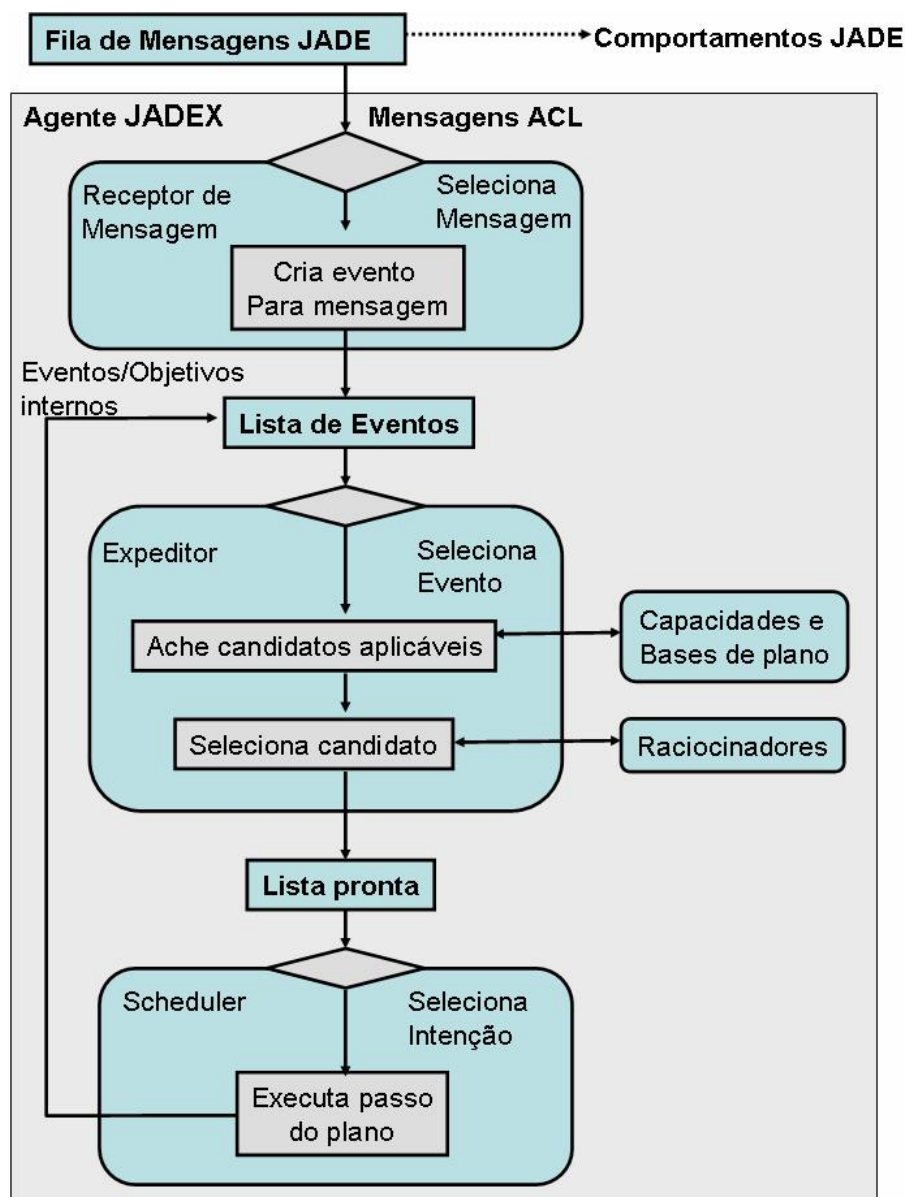


Figura 4.6: Modelo de execução e integração de JADEX com JADE.

[Pokahr05] Toda a funcionalidade disponível em JADE pode ainda ser usada em planos JADEX. Para usar comportamentos de JADE conjuntamente com os planos em JADEX, o comportamento do receptor da mensagem deve suportar a filtragem de mensagens ACL. É necessário classificar as mensagens que são manipuladas pelos planos e tem que conseqüentemente ser despachado ao sistema interno de JADEX e manter as outras mensagens disponíveis para os comportamentos de JADE.

Todos os agentes JADEX são também agentes JADE, conseqüentemente todas as características de JADE podem ser usadas em JADEX, mas isso pode ser inapropriado em alguns casos [Pokahr05]. Algumas características de JADE podem causar efeitos colaterais não desejados quando usadas em conjunto com JADEX, enquanto que o uso de outras características é desencorajado em favor de características introduzidas por JADEX.

### Usando JADE Behaviours

Quando se tem algum código em JADE que se quer aproveitar para usar num agente JADEX, mas você não deseja converter seus comportamentos (*behaviours*) em planos (*plans*), você pode utilizá-los. Para adicionar comportamentos a uma agente JADEX, basta apenas utilizar o método *addBehaviour()* do *JadeWrapperAgent* (o método que é herdado do *jade.core.Agent*). Você pode chamar o método de dentro de um plano usando *getScope().getJadeAgent()* para acessar o agente *wrapper*, ou você estende a classe *JadeWrapperAgent* e cobre o método *setup()* como é usualmente feito em JADE (não esquecendo de chamar ao *super.setup()*, senão as propriedades BDI do agente não serão inicializadas) [Pokahr05].

Por *default* todas as entradas de mensagens são manipuladas por JADEX. Para permitir que comportamentos em JADE manipulem entradas de mensagens, estes têm que ser ignorados por JADEX. Pode-se especificar um *template* de mensagem como uma propriedade de um agente no ADF para identificar aquelas mensagens que não deveriam ser manipuladas por JADEX [Pokahr05]:

**Exemplo 4.4: Modelo para permitir que as mensagens sejam manipuladas por JADEX.**

```
<agent . . .>
. . .
<properties>
<property name="jadefilter">
MessageTemplate.MatchPerformative(ACLMessage.QUERY_REF)
</property>
. . .
</properties>
. . .
</agent>
```

As ações ou comportamentos ou planos que um agente desempenha dentro de um SMA são fundamentais para que o objetivo final da aplicação seja alcançado. Em JADE, ao desenvolver os agentes, devemos implementar as tarefas específicas para o agente escrevendo um ou mais comportamentos através das subclasses de *Behaviour*, instanciando-as e adicionando-as ao agente.

JADE contém comportamentos específicos para a maioria das tarefas comuns na programação de agentes, tais como envio e recebimento de mensagens e até a construção de tarefas mais complexas.

A estrutura dos comportamentos de JADE é dada através de um escalonador não preemptivo, interno à superclasse *Agent*, que gerencia automaticamente o escalonamento desses comportamentos. Em outras palavras, um comportamento é executado por vez por um determinado tempo (*quantum*). Quando acaba esse tempo, o comportamento pode voltar para o fim da fila, caso não tenha acabado de executar sua tarefa. O fato de ser não preemptivo significa dizer que não existe prioridade entre os comportamentos, ou seja, cada comportamento adicionado deve ir para o fim da fila e todos têm a mesma fatia de tempo.

*Behaviour* é uma classe abstrata de JADE disponível no pacote *jade.core.behaviours*. Uma classe abstrata é uma classe que possui alguns métodos implementados e outros não [Deitel03]. Ela tem como finalidade prover a estrutura de comportamentos para os agentes JADE implementarem.

O principal método da classe *Behaviour* é o *action()*. É nele que serão implementadas as tarefas ou ações que este comportamento irá tomar. Outro método importante é o *done()*, ele é usado para informar ao escalonador do agente se o comportamento foi terminado ou não. Ele retorna *true* caso o comportamento tenha terminado e assim este é removido da fila de comportamentos, e retorna *false* caso o comportamento ainda não tenha terminado, obrigando o método *action()* a ser executado novamente.

Outros atributos importantes da classe *Behaviour* são:

- Protected Agent *myAgent* – retorna o agente ao qual este comportamento pertence
- void *block()* – Bloqueia esse comportamento. Pode ser passado como parâmetro um o tempo de bloqueio em milissegundos: void *block* (long tempo).
- void *restart()* – Reinicia um comportamento bloqueado.
- void *reset()* – Restaura o estado inicial do comportamento.
- boolean *isRunnable()* – Retorna se o comportamento está bloqueado ou não.

JADE possui várias classes de comportamentos prontas para uso pelo desenvolvedor adequando-as de acordo com a necessidade específica do agente.

**Tabela 4.1: Tabela com os tipos de comportamento.**

Tipo do comportamento	Característica principal
<i>Behaviour</i>	Modela uma tarefa genérica
<i>SimpleBehaviour</i>	Modela uma tarefa simples (que não tem sub-tarefas)
<i>OnShotBehaviour</i>	Modela uma tarefa atômica (seu método <i>done()</i> retorna <i>true</i> )
<i>CyclicBehaviour</i>	Modela uma tarefa cíclica (seu método <i>done()</i> retorna <i>false</i> )
<i>CompositeBehaviour</i>	Modela uma tarefa complexa (realizada através da composição de outras tarefas)
<i>SequentialBehaviour</i>	Modela uma tarefa na qual suas sub-tarefas são sequenciais

<i>ParallelBehaviour</i>	Modela uma tarefa na qual suas sub-tarefas são concorrentes
<i>FSMBehaviour</i>	Modela uma tarefa na qual suas sub-tarefas correspondem as atividades realizadas em estados de uma Máquina de Estados Finitos

Hierarquicamente, os comportamentos estão estruturados da seguinte forma:

- Classe jade.core.behaviours.Behaviour
  - Classe jade.core.behaviours.CompositeBehaviour
    - Classe jade.core.behaviours.FSMBehaviour
    - Classe jade.core.behaviours.ParallelBehaviour
    - Classe jade.core.behaviours.SequentialBehaviour
  - Classe jade.core.behaviours.ReceiverBehaviour
  - Classe jade.core.behaviours.SimpleBehaviour
    - Classe jade.core.behaviours.CyclicBehaviour
    - Classe jade.core.behaviours.OneShotBehaviour
      - Classe jade.core.behaviours.SenderBehaviour
    - Classe jade.core.behaviours.TickerBehaviour
    - Classe jade.core.behaviours.WakerBehaviour

Na Figura 4.7 temos um diagrama de classes em UML que mostra a hierarquia de classes que derivam da classe abstrata *Behaviour*.

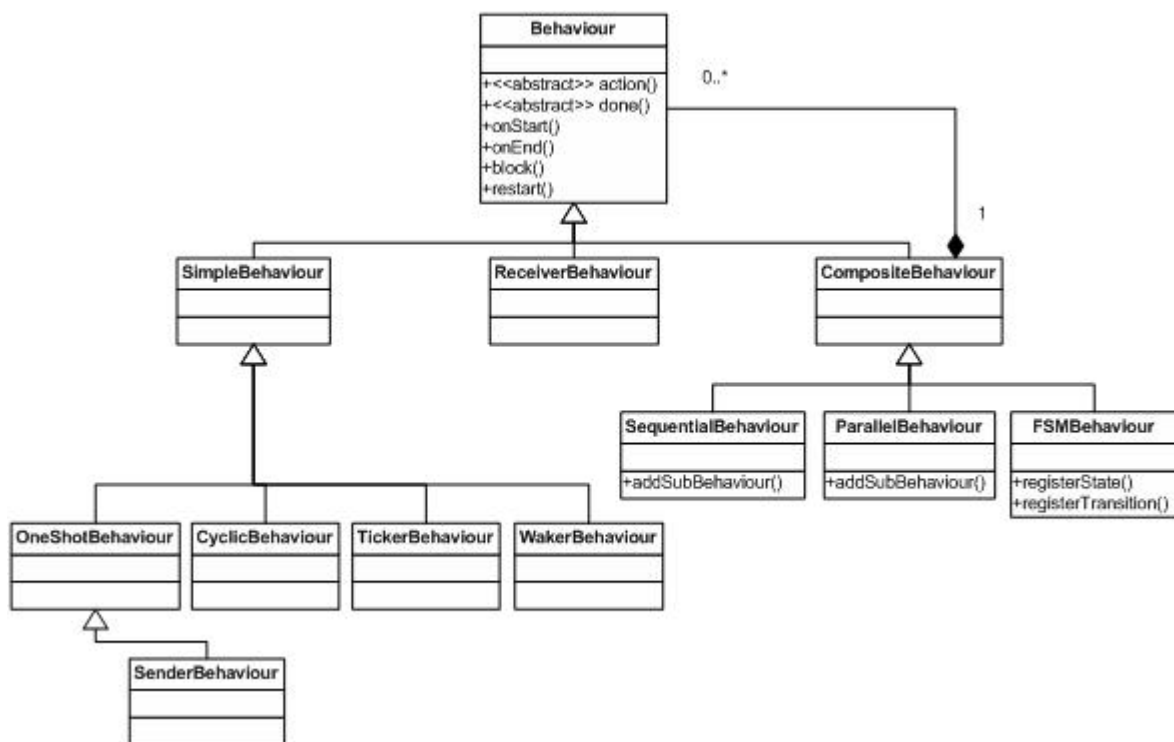


Figura 4.7: Hierarquia das classes de comportamento definidas em JADE.



A classe *ReceiverBehaviour* implementa um comportamento para recebimento de mensagens ACL. Ela encapsula o método *receive()* como uma operação atômica, tendo o comportamento encerrado quando uma mensagem ACL é recebida. Possui o método *getMessage()* que permite receber a mensagem.

Já a classe *CompositeBehaviour* é uma classe abstrata para comportamentos, como o próprio nome diz, compostos por diferentes partes. Ela controla internamente comportamentos filhos dividindo seu tempo de execução de acordo com alguma política de escalonamento. Possui três classes herdadas disponibilizadas pelo JADE. São elas:

- ***jade.core.behaviours.FSMBehaviour*** – Comportamento composto baseado no escalonamento por máquina de estados finitos (*Finite State Machine*). Mais especificamente cada comportamento-filho representa um estado na máquina de estados finitos. *FSMBehaviour* fornece métodos para registrar estados (sub-comportamentos) e transições que definem como se dará o escalonamento.
- ***jade.core.behaviours.ParallelBehaviour*** - Comportamento composto com escalonamento concorrente dos comportamentos filhos. *ParallelBehaviour* executa seus comportamentos filhos concorrentemente e finaliza quando uma condição particular em seus sub-comportamentos é atingida. Essas condições são definidas no construtor da classe, passando como parâmetro as constantes *WHEN\_ALL*, quando forem todos, *WHEN\_ANY*, quando for qualquer um, ou um valor inteiro que especifica o número de comportamentos filhos terminados que são necessários para finalizar o *ParallelBehaviour*. O método para adicionar comportamentos-filho é o *addSubBehaviour()*.
- ***jade.core.behaviours.SequentialBehaviour*** - Comportamento composto com escalonamento seqüencial de comportamentos-filho. Ou seja, os comportamentos-filho são executados numa ordem seqüencial e só é encerrado quando o ultimo comportamento-filho é finalizado. O método para adicionar comportamentos-filho é o *addSubBehaviour()*.

Por último, temos a classe abstrata *SimpleBehaviour*. Ela é um comportamento atômico. Isto é, modela comportamentos que são feitos para serem únicos e monolíticos e que não podem ser interrompidos. Possui quatro classes herdadas disponibilizadas pelo JADE.

São elas:

- ***jade.core.behaviours.CyclicBehaviour*** – Comportamento atômico que deve ser executado sempre. Essa classe abstrata pode ser herdada para criação de comportamentos que se manterão executando continuamente. No caso, o método *done()* herdado da classe *Behaviour*, sempre retorna *false*, o que faz com que o comportamento se repita como se estivesse em um *loop* infinito.
- ***jade.core.behaviours.TickerBehaviour*** – Comportamento que executa periodicamente tarefas específicas. Ou seja, é uma classe abstrata que implementa um comportamento que executa periodicamente um pedaço de código definido pelo usuário em uma determinada frequência de repetições. O desenvolvedor redefine o método *onTick()* e inclui o pedaço de código que deve ser executado periodicamente. O período de “*ticks*” é definido no construtor da classe em milissegundos.
- ***jade.core.behaviours.WakerBehaviour*** – Comportamento que é executado depois de determinado tempo expirado. Em outras palavras, é uma classe abstrata onde uma tarefa “*OneShot*” é executada apenas depois que determinado tempo é expirado. No caso, a tarefa é inserida no método *handleElapsedTimeout()* o qual é chamado sempre que o intervalo de tempo é transcorrido.

- ***jade.core.behaviours.OneShotBehaviour*** – Comportamento atômico que é executado uma única vez. Essa classe abstrata pode ser herdada para a criação de comportamentos para tarefas que precisam ser executadas em apenas uma única vez. Tem como classe filha a classe *SenderBehaviour*.
  - ***jade.core.behaviours.SenderBehaviour*** – é um comportamento do tipo *OneShotBehaviour* para envio de mensagens ACL. Essa classe encapsula o método *send()* como uma operação atômica. No caso, esse comportamento envia determinada mensagem ACL e se finaliza. A mensagem ACL é passada no construtor da classe.

Devido ao modelo não-preemptivo de escolha de comportamentos dos agentes, os programadores de agentes devem evitar uso de *loops* infinitos e até executar atividades muito longas dentro do método *action()* dos *Behaviours*. Isso porque enquanto o método *action()* de algum comportamento estiver sendo executado, nenhum outro comportamento deste mesmo agente poderá ser executado até que o fim do método *action()* ocorra.

Além disso, como não há armazenamento em pilha, o método *action()* sempre será executado do início. Não sendo possível, por exemplo, interromper um comportamento no meio de um *action()*, passar o controle para outros comportamentos e voltar para o comportamento original de local onde tinha sido interrompido [Bellifemine03].

### 4.4.3 Ontologia

Quando um agente se comunica com um outro agente, determinada quantidade de informação é transferida de um para o outro através de uma mensagem ACL. Dentro da mensagem ACL, que é representada como uma expressão de conteúdo, este consistente de uma linguagem de conteúdo apropriada e codificada em um dado formato, tal como string. Tanto um quanto o outro agente possuem sua própria forma de representar esse conteúdo.

Por exemplo, a informação de que existe uma pessoa cujo nome é Bárbara e que esta tem 23 anos numa expressão de conteúdo ACL poderia ser representado como abaixo:

#### Exemplo 4.5: Representação de uma expressão de conteúdo ACL.

```
(Pessoa :nome Bárbara :idade 23)
```

Armazenando esta informação dentro de um agente como uma variável do tipo string não é apropriada para se manipular, por exemplo, pegar a idade da pessoa iria requerer tempo para realizar o *parser* na string.

Considerando agentes de software escritos em Java, a informação pode ser convenientemente representada dentro de um agente também como objetos Java. Por exemplo, para representar a informação acima sobre a pessoa como uma instância de uma classe, temos:

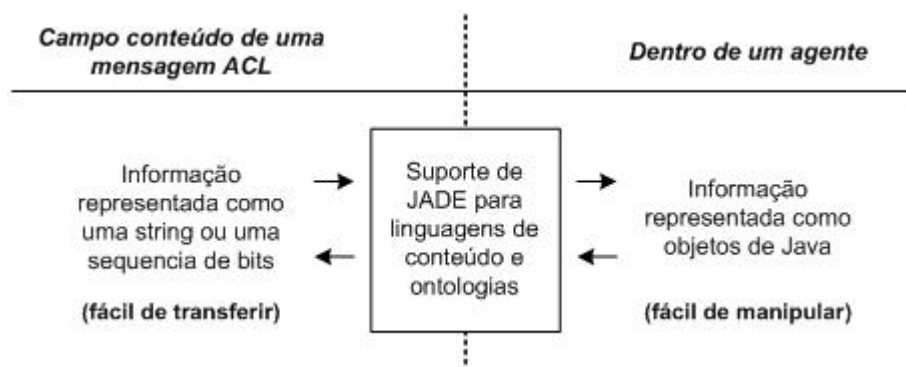
**Exemplo 4.6: Representação da informação como uma instância de uma classe.**

```

class Pessoa {
    String nome;
    int idade;
    public String getNome () {return nome; }
    public void setNome (String n) {nome = n; }
    public int getIdade () {return idade; }
    public void setIdade (int i) {idade = i; }
    ...
}

```

Toda vez, portanto que um agente enviar uma informação para outro agente; (i) haverá a necessidade de converter sua representação interna da informação numa expressão de conteúdo ACL e o agente que receber esta informação no formato ACL fará a conversão contrária; (ii) além disso, o agente que receber essas informações deverá também executar checagens semânticas para verificar se esta informação tem significado, por exemplo, i.e. que cumpre com as regras (por exemplo, que a idade da pessoa é realmente um valor inteiro) da ontologia tal que ambos os agentes atribuem um significado apropriado à informação.



**Figura 4.8: Conversão realizada pelo suporte de JADE para linguagens de conteúdo e ontologias.**

O suporte para linguagens de conteúdo e ontologias providos por JADE é projetado para realizar automaticamente as conversões citadas acima e a checagem das operações, conforme apresentado na Figura 4.8. Desta forma, JADE permite que os desenvolvedores manipulem informações dentro de seus agentes como objetos de Java, sem a necessidade de qualquer esforço extra.

Desta forma, para que JADE execute a checagem da semântica é necessário classificar todos os elementos no domínio de acordo com suas características semânticas genéricas. Esta classificação é derivada da linguagem ACL definida pela FIPA que requer que o conteúdo de cada *ACLMessage* tenha uma semântica apropriada de acordo com a performativa da *ACLMessage*. Na Figura 4.8 apresentamos o modelo de referência de conteúdo de JADE, e abaixo explicamos alguns de seus elementos.

Predicados são expressões que dizem algo sobre o status do mundo e pode ser *true* ou *false*. Por exemplo, (Trabalha-para (Pessoa :nome Eduardo) (Empresa :nome TILAB)) indica que “a pessoa Eduardo trabalha para a empresa TILAB”. Predicados podem ser usados por exemplo como conteúdo

de uma mensagem do tipo INFORM ou QUERY-IF, enquanto que não faria sentido ser usado por uma mensagem do tipo REQUEST.

Termos são expressões que identificam entidades (abstratas ou concretas) que existem no mundo e que os agentes falam e raciocinam sobre. Os termos são classificados em:

- Conceitos (*Concepts*): são expressões que indicam entidades com uma estrutura complexa que pode ser definida em termos de *slots*, por exemplo, (Pessoa :nome Eduardo). Conceitos tipicamente não fazem sentido como conteúdo de mensagens ACL. Geralmente são referenciados dentro de predicados e outros conceitos tais como: (Livro :titulo “Senhor dos Anéis” :autor (Pessoa :nome “J.R.R. Tolkien”)).
- Ações de agentes (*Agent actions*): são conceitos que indicam ações que podem ser executadas por alguns agentes, por exemplo, (Vender (Livro :titulo “Senhor dos Anéis”) (Pessoa :nome Eduardo)). Isto faz sentido como conteúdo de *ACLMessage* tais como REQUEST.
- Primitivas (*Primitives*): são expressões que indicam uma entidade atômica tal como strings e inteiros.
- Agregados (*Aggregates*): são expressões que indicam entidades que são grupos de outras entidades, por exemplo, (seqüência (Pessoa :nome Eduardo) (Pessoa :nome Bárbara))
- Identificando Expressões Referencial (*Identifying Referential Expressions*) (IRE): são expressões que identificam entidade (ou entidades) para as quais um dado predicado é verdadeiro, por exemplo, (todo ?x (Trabalha-para ?x (Empresa :nome TILAB)) identificando que “todo elemento x para o qual o predicado é verdadeiro, todas as pessoas trabalham para empresa TILAB”. Essas expressões são tipicamente usadas em mensagens do tipo *query*, por exemplo QUERY\_REF, e requer variáveis.
- Variável (*Variables*): são expressões (tipicamente usadas em *queries*) que indicam um elemento genérico não conhecido a priori.

Uma linguagem de conteúdo deve ser capaz de distinguir entre todos os tipos de elementos acima. Uma ontologia para um dado domínio é, portanto um conjunto de esquemas definindo a estrutura de predicados, ações dos agentes e conceitos que são pertinentes ao domínio.

O modelo da Figura 4.9 inclui também mais dois tipos de elementos que são introduzidos considerando que apenas predicados ações de agentes, IREs e listas de elementos desses tipos de elementos (*ContentElementList*) são um conteúdo significativo de pelo menos uma mensagem ACL. Todos estes herdam de *ContentElement*.

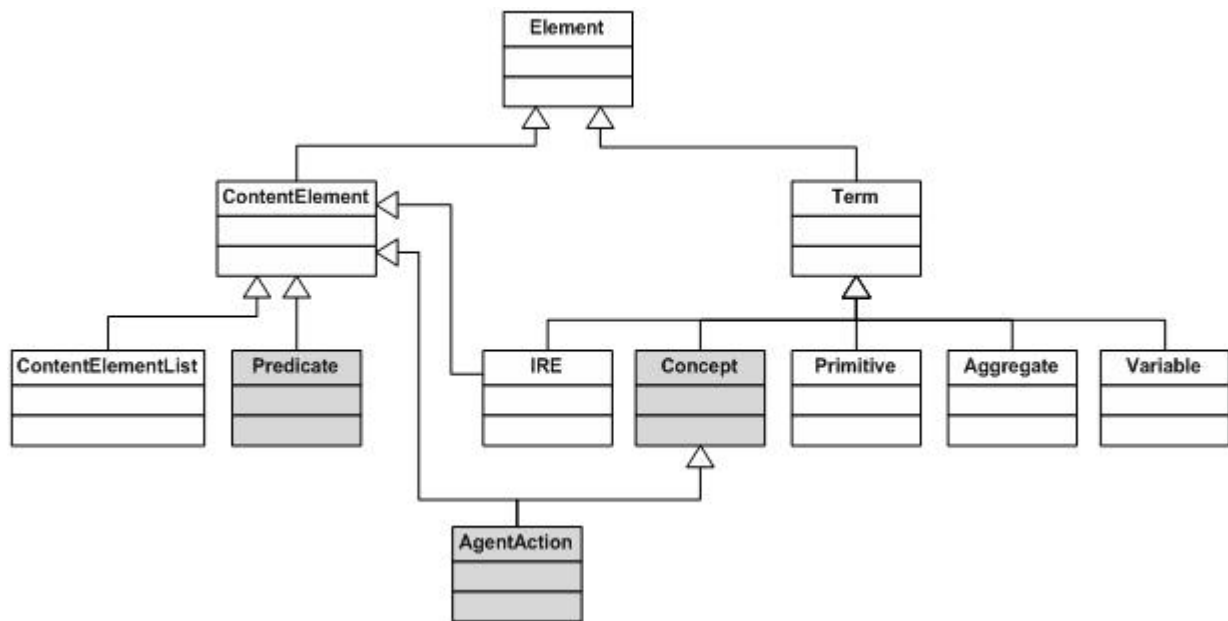


Figura 4.9: O modelo de referência de conteúdo de JADE.

Para extrair esses conteúdos das mensagens ACL tanto em JADE quanto em JADEX pode-se usar os métodos *getContent()* e *setContent()* da classe *IMessageEvent*. Quando algum problema acontece uma *jadex.runtime.ContentException* é lançada, esta exceção herda de *RuntimeException*.

Para o codificador e decodificador de mensagens poderem trabalhar corretamente, a definição da linguagem e ontologia correspondentes tem de ser registradas no gerenciador de conteúdo do agente (ACM). JADEX fará isso, quando se declara as linguagens e ontologias no ADF. Nas tags `<languages>` e `<ontologies>` especifica-se os objetos da linguagem e ontologia, da mesma forma como se registraria num agente JADE convencional. Por exemplo, para usar a linguagem de conteúdo *fipa-sl0* e a ontologia *fipa-agent-management*, temos:

Exemplo 4.7: Uso de linguagem e ontologia pelo agente.

```

<agent . . .>
  <imports>
    <import>jade.content.lang.sl.SLCodec</import>
    <import>jade.domain.FIPAAgentManagement.*</import>
  </imports>
  . . .
  <languages>
    <language>new SLCodec(0)</language>
  </languages>
  <ontologies>
    <ontology>FIPAMangementOntology.getInstance()</ontology>
  </ontologies>
</agent>

```

## Capítulo

# 5 Estudo de Caso

Este capítulo apresenta um estudo de caso desenvolvido, com o propósito de aplicar a metodologia Tropos e utilizar a plataforma JADEX, para propor a implementação de um SMA. A aplicação escolhida é um sistema de gerenciamento de conteúdo que utiliza agentes para simular o processo de publicação de notícias em um site de um jornal na *Web*.

### 5.1 Introdução

A *World Wide Web* (WWW) pode ser vista como um grande repositório de informações, uma vez que atualmente é possível encontrar praticamente qualquer tipo de informação em diversos *sites*, apresentados em várias línguas e formatos. Dentre esses *sites*, os dos veículos de comunicação e integrantes da imprensa possuem fundamental importância para disseminação de informações, sendo responsáveis pela rápida difusão de notícias pelo mundo.

Nos dias de hoje, não existe nenhum jornal de grande circulação no país que não esteja disponível na *Web*, através de uma versão *on-line* do seu conteúdo publicado em mídia impressa, complementada ou não por outras matérias ou serviços. Mesmo assim, apesar das redações responsáveis pela publicação de informações nesses *sites* geralmente constituírem uma organização ágil, o processo de gerenciamento de publicação de notícias na *Web* ainda é muito precário, uma vez que, por mais estruturada e conectada com outros canais de comunicação a redação esteja, o volume de informações que precisa ser gerenciado é cada vez maior. Além disso, a tarefa de atualizar o site com notícias que chegam de fontes de informação como as agências de notícias nacionais ou internacionais, muitas vezes é feita de forma manual, mesmo que esta notícia seja publicada integralmente como foi recebida de alguma agência de notícias.

Quando esta tarefa é realizada automaticamente, as notícias apenas são categorizadas ou dispostas em ordem decrescente de data de publicação, não havendo uma seleção inteligente auxiliada por computador das notícias a serem publicadas. Para sistematizar e automatizar o processo de criação e atualização de informações *on-line* podemos utilizar sistemas gerenciadores de conteúdo (*Content Management System* - CMS), visando garantir a precisão da informação, reduzir a duplicidade de conteúdo e organizar o crescimento da capacidade de armazenamento de informações, entre outros objetivos.

O problema escolhido para ser utilizado como estudo de caso desta dissertação consiste em propor um sistema que simule a redação de jornal *on-line* utilizando um SMA. O sistema proposto possui o gerenciamento da publicação de notícias na *Web*, através da automatização inteligente do processo de publicação num site de jornal do conteúdo fornecido por agências de notícias.

## 5.2 Sistema Gerenciador de Notícias na Web

Hoje há uma grande necessidade de se compartilhar todos os tipos e formatos de documentos de maneira rápida e fácil utilizando a *web*. Assim, surgem os sistemas de gestão de conteúdo. Gestão de conteúdo é o gerenciamento de informações, focando a captação, ajuste, distribuição e gerenciamento dos conteúdos para apoio ao processo de negócios de toda a empresa. Esses conteúdos podem ser estruturados ou não, procedentes de sistemas de Imagem, COLD, Gerenciamento de Documentos, sistemas legados, bancos de dados, arquivos nos diretórios e de qualquer outro arquivo digital como som e vídeo [CENADEM04]. A característica básica de uma solução de gestão de conteúdo é oferecer acesso a todos os conteúdos da empresa através de uma interface única baseada em *browser*. As funcionalidades essenciais, dentre muitas outras, que caracterizam o conceito e que se desenvolvem à medida que novos produtos de mercado chegam à maturidade são [Parreiras04]:

- Gestão de usuários e dos seus direitos (autenticação, autorização, auditoria);
- Criação, edição e armazenamento de conteúdo em formatos diversos (html, doc, pdf, etc);
- Uso intensivo de metadados (ou propriedades que descrevem o conteúdo);
- Controle da qualidade de informação (com fluxo ou trâmite de documentos ou *workflow*);
- Classificação, indexação e busca de conteúdo (recuperação da informação com mecanismos de busca);
- Gestão da interface com os usuários (atenção à usabilidade, arquitetura da informação);
- Sindicalização (disponibilização de informações em formatos XML visando seu agrupamento ou agregação de diferentes fontes);
- Gestão de configuração (gestão de versões);
- Gravação das ações executadas sobre o conteúdo para efeitos de auditoria e possibilidade de desfazê-las em caso de necessidade.

As principais áreas de aplicação são [Parreiras04]:

- Sítios editoriais: É talvez o tipo de sítio mais comum hoje na *web*, que assume natureza de mídia de comunicação. Um sítio deste tipo permite a um indivíduo ou a um grupo posicionar-se como fonte de informação sobre assuntos específicos. Apresentam-se sob diferentes formas de acordo com o modelo econômico, o objetivo visado, e a tendência do momento.
- Comunidades de prática em linha: Este tipo de sítio é o mais utilizado por comunidades dedicadas em desenvolver software livre. Uma comunidade em linha reúne pessoas que compartilham centros de interesse de ordem geral ou profissional, não se resumindo obviamente a códigos de programas. Estes sítios oferecem a possibilidade de contribuir com informações na forma de artigos e notícias, dentre outros, e alertar a comunidade para informações disponíveis em outros lugares da *web*.
- Portais corporativos: São aplicações que funcionam em intranets ou extranets, mas também podem ser acessadas pela internet. Dentre vários benefícios, essas aplicações permitem capitalizar a informação, o conhecimento e a competência das organizações: idéias

estruturadas ou não, documentação, procedimentos administrativos, técnicos, de marketing, dentre outros.

No nosso estudo de caso, como já mencionamos, vamos abordar o projeto e implementação de um sítio editorial, um jornal *on-line*.

Resumidamente, o processo de publicação começa quando uma pauta é gerada. Na pauta estão diretrizes, guias, que direcionam o repórter a produzir uma notícia, ou um repórter a registrar imagens. A pauta é decidida numa reunião, onde editores e repórteres discutem e avaliam a necessidade de se abordar, investigar um determinado assunto. Cada notícia gerada pertence a uma categoria: esportes, tempo, economia, etc. O repórter/fotógrafo recebe uma pauta e quando este está com seu artefato pronto (notícia ou fotografia), envia-o para que o editor aprove e encaminhe para publicação.

Uma das ferramentas de gerenciamento de conteúdo encontrada na *Web*, é o Publique! [Publique03]. É uma ferramenta de gerenciamento de conteúdo, desenvolvida para sistematizar e automatizar o processo de criação e atualização de informações *on-line*. O Publique! pode ser operado através de qualquer *browser Web*, sem necessidade de um programa específico. Ele utiliza um sistema de formulários *Web* que permite o gerenciamento das informações até mesmo por usuários que não tenham qualquer conhecimento de HTML, um conjunto de *templates* para a apresentação dessas informações e um banco de dados que torna possível gerar e armazenar dinamicamente as páginas do *site*. O sistema utiliza o conceito de usuários e personagens e implementa, entre eles, um fluxo de trabalho para a criação e apresentação das informações geradas. Além disto, como funciona via *browser*, o sistema permite que os diversos usuários envolvidos na elaboração do site desenvolvam suas funções com a mesma eficiência quer estejam na mesma sala ou em países distintos.

No nosso estudo de caso propomos então a implementação de um jornal para web, utilizando os mesmo conceitos apresentados pelo Publique!, e utilizando a metodologia Tropos e os conceitos referentes a SMAs.

Aqui, identificamos como principais atores na redação de um jornal com publicação de conteúdo na internet foram:

- Usuário: ator que representa o leitor do conteúdo publicado no site do jornal;
- Chefe de redação: ator responsável por delegar tarefas aos repórteres e fotógrafos através da distribuição de pautas, bem como interagir com os editores para negociar os assuntos e publicação das matérias. É responsável por coordenar as reuniões de pauta (reuniões diárias com o editor geral e todos os editores, realizadas para escolher as manchetes, matérias futuras e pautas do dia). O secretário de redação e o chefe de reportagem trabalham auxiliando o chefe de redação.
  - Secretário de redação: ator responsável pela comunicação entre a chefia de redação e os editores. Deve sempre saber a que categoria uma notícia pertence, quem é o editor responsável por ela e em que seção deverá ser publicada. Controla as pautas e notícias para que não ocorra falta, falha ou repetição de conteúdo na edição do jornal.
  - Chefe de reportagem: ator responsável por orientar os repórteres e fotógrafos com relação às pautas. Mantém-se sempre informado sobre o que os repórteres e



fotógrafos estão fazendo, controlando o recebimento das notícias e fotografias e alocando repórteres para cumprir pautas de outros que estejam sobrecarregados ou com algum problema que os impossibilite de fazer aquilo que foi delegado. Deve repassar as notícias recebidas para os editores.

- **Repórter:** ator responsável por providenciar para o chefe de reportagem as matérias de acordo com a pauta recebida contendo os assuntos mais relevantes daquela edição. Pode produzir a matéria localmente, procurar ou buscar notícias nas agências de notícias com as quais o jornal tem convênio. Deve cumprir a pauta recebida, bem como buscar furos de reportagem.
- **Fotógrafo:** ator responsável por obter as fotografias necessárias relacionadas às matérias e aos assuntos que constarem na pauta que receber do chefe de reportagem. Trabalha sempre acompanhando os repórteres na cobertura dos eventos.
- **Editor:** ator responsável pela editoria de algum domínio de informação, como por exemplo, esportes, moda, política, etc. Coordena a montagem e publicação das páginas relacionadas a esse domínio. Tem como responsabilidades editar as matérias, titulá-las, diagramá-las e revisar o seu conteúdo. Tem autonomia para trocar matérias ou a estrutura de páginas do jornal para inserir novas matérias. Interage com o editor geral, com os demais editores, chefe de redação, e chefe de reportagem.
- **Editor geral:** é o ator responsável por fiscalizar e autorizar a publicação das edições finais das editorias, interagindo com os editores e com o chefe de redação. Revisa as partes críticas do jornal, bem como orienta os editores para publicar ou não matérias sobre determinado assunto.
- **Webmaster:** ator responsável por publicar no site do jornal as páginas das editorias e a página principal do jornal, de acordo com as configurações adotadas para apresentação do conteúdo na WEB;

Para fins de simplificação, os atores chefe de redação, secretário de redação e chefe de reportagem serão representados por um único ator, denominado chefe de redação, uma vez que as responsabilidades que cabem a cada um deles estão intrinsecamente ligadas.

Aqui, consideraremos que o conteúdo publicado na *web* possuirá um ciclo de vida com quatro etapas principais: criação, organização e submissão, distribuição, e arquivamento.

## **5.3 Modelagem com Tropos**

### **5.3.1 Requisitos Iniciais**

Através da análise do ambiente organizacional descrito acima, conseguimos elementos suficientes para produzir a modelagem dos requisitos iniciais do sistema, representados nos modelos SD e SR descritos nas subseções seguintes.

## Modelo de Dependência Estratégica

Neste modelo está representada a análise do ambiente organizacional estudado, sendo apresentados os principais atores identificados para uma redação de jornal com publicação na *Web* e as relações de dependências externas entre eles. Os atores desse ambiente são: *usuário*, *webmaster*, *editor*, *editor-geral*, *fotógrafo*, *repórter* e *chefe-redação*. A Figura 5.1 apresenta o modelo *i\** de dependência estratégica da fase de Requisitos Iniciais deste trabalho.

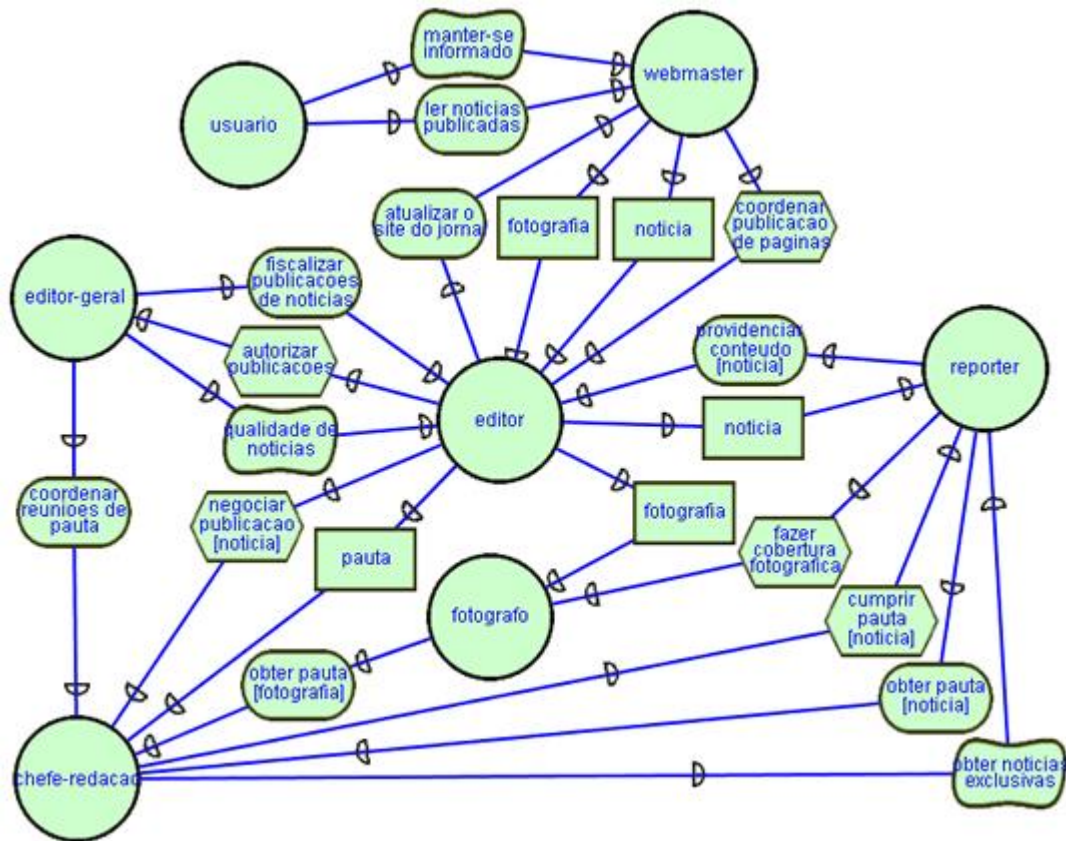


Figura 5.1: Modelo *i\** de Dependência Estratégica dos Requisitos Iniciais.

O *usuário* depende do *webmaster* para completar seu objetivo de *ler notícias publicadas* e para atingir a meta-soft *manter-se informado*. O *editor*, por sua vez, depende do *webmaster* para que o objetivo *atualizar o site do jornal* seja atingido. Para tanto, o *webmaster* necessita que o *editor* realize a tarefa de *coordenar publicação de páginas* e que os recursos *notícia* e *fotografia* lhe sejam fornecidos. Além disso, o *editor* também depende do *repórter* e do *fotógrafo* para que os recursos *notícia* e *fotografia* sejam providenciados. Além disso, o *repórter* depende do *editor* para que o objetivo *providenciar conteúdo de notícias* seja satisfeito e do *fotógrafo* para que a tarefa de *fazer cobertura fotográfica* seja realizada.

O ator *editor-geral* está relacionado com o *editor* através da dependência de meta *fiscalizar publicações de notícias* e da meta-soft *qualidade de notícias*, que para serem atingidos dependem do *editor*. Inversamente, o *editor* depende do *editor-geral* a fim de que a tarefa *autorizar publicações* seja realizada. Do mesmo modo, o *editor-geral* depende do *chefe-redação* para que o objetivo *coordenar reuniões de pauta* seja completado. É esperado que o *chefe-redação* forneça o recurso *pauta* e realize a tarefa *negociar publicação de notícias*, sendo estas dependências em relação ao ator *editor*. O

*fotógrafo* e o *repórter* também dependem do *chefe-redação* para atingirem os objetivos *obter pauta de fotografias* e *obter pauta de notícias*, respectivamente. Os últimos relacionamentos de dependência identificados apresentam a necessidade do *repórter* de realizar a tarefa *cumprir pauta de notícias* e atingir a meta-soft *obter notícias exclusivas* para o *chefe-redação*.

### Modelo de Razão Estratégica

Uma vez que os atores e suas principais dependências externas foram identificados no modelo de dependência estratégica, o modelo de dependências estratégicas permite um refinamento das descrições dos processos descrevendo os relacionamentos intencionais que são internos aos atores. A Figura 5.2 apresenta o modelo SR da redação de um jornal, expandido para o ator editor, de acordo com a fase de Requisitos Iniciais de Tropos. A seguir são descritas as expansões realizadas que apresentam os relacionamentos intencionais dos atores *repórter*, *editor*, *chefe-redação* e *webmaster*.

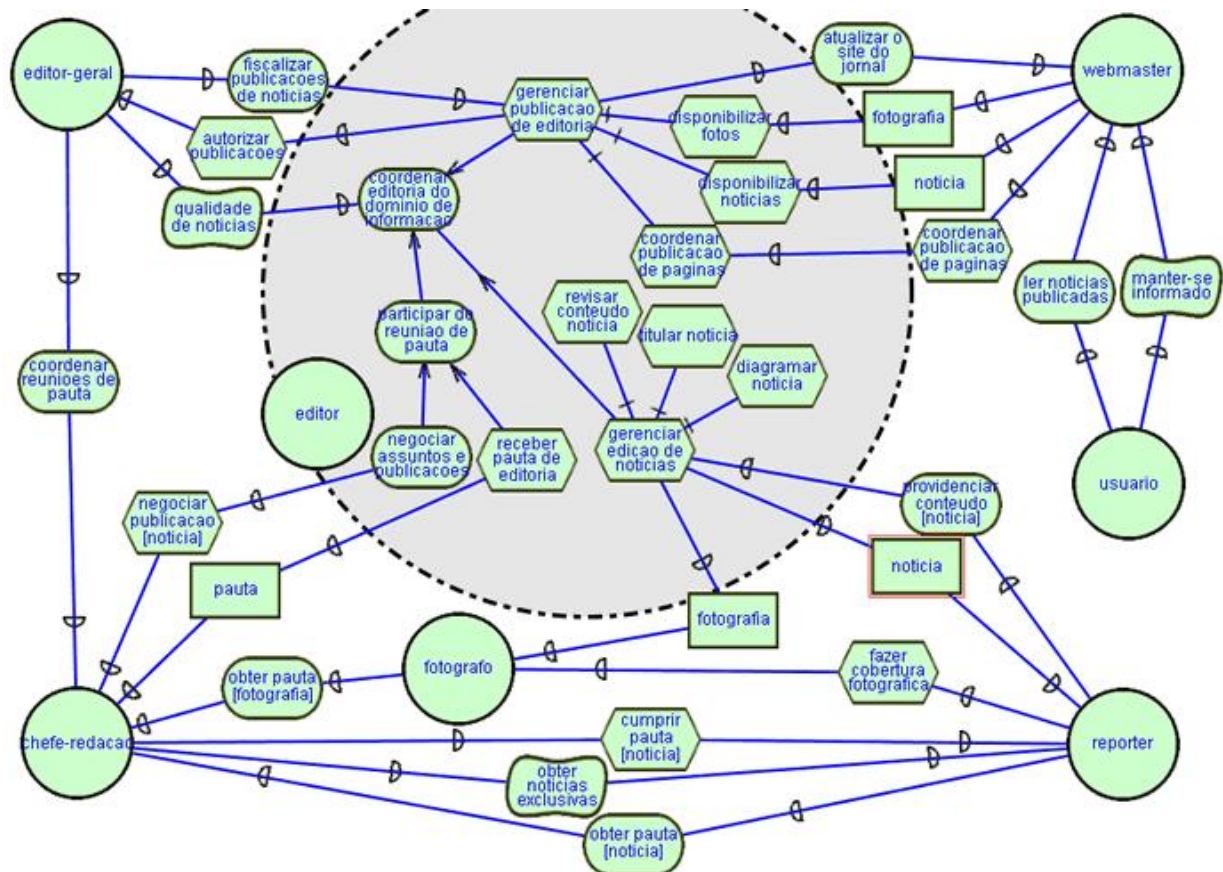


Figura 5.2: Modelo i\* de Razão Estratégica de uma redação de jornal para o ator editor.

O ator editor tem como principal objetivo *coordenar a editoria do domínio de informação*. Para atingir esse fim, há três meios: realizar uma das tarefas de *gerenciar publicação de editoria* ou *gerenciar edição de notícias*, ou atingir o objetivo *participar de reunião de pauta*. A tarefa *gerenciar publicação de editoria* pode ser decomposta nas sub-tarefas *coordenar publicação de páginas*, *disponibilizar fotos* e *disponibilizar notícias*, que devem sempre ser realizadas visando atender às dependências entre o *editor* e o *webmaster*. Como razões para atender às dependências em relação ao ator *chefe-redação*, o *editor* pode atingir a meta *participar de reunião de pauta* há duas maneiras: realizar a tarefa *receber*



*pauta de editoria* ou atingir a meta *negociar assuntos e publicações*. A tarefa *gerenciar edição de notícias*, por sua vez, pode ser decomposta nas seguintes tarefas: *diagramar notícia*, *revisar conteúdo de notícia*, e *titular notícia*.

As intenções do ator *webmaster*, apresentadas na Figura 5.3, podem ser determinadas através do processamento de uma tarefa principal denominada *publicar site do jornal*, que pode ser decomposta em três sub-tarefas: *publicar conteúdo*, *manter páginas de editorias*, e *manter página principal*; além de exigir a satisfação da meta *utilizar padrões de layout* e da meta-soft *atualizar com rapidez*, que também fazem parte da decomposição de *publicar site do jornal*.

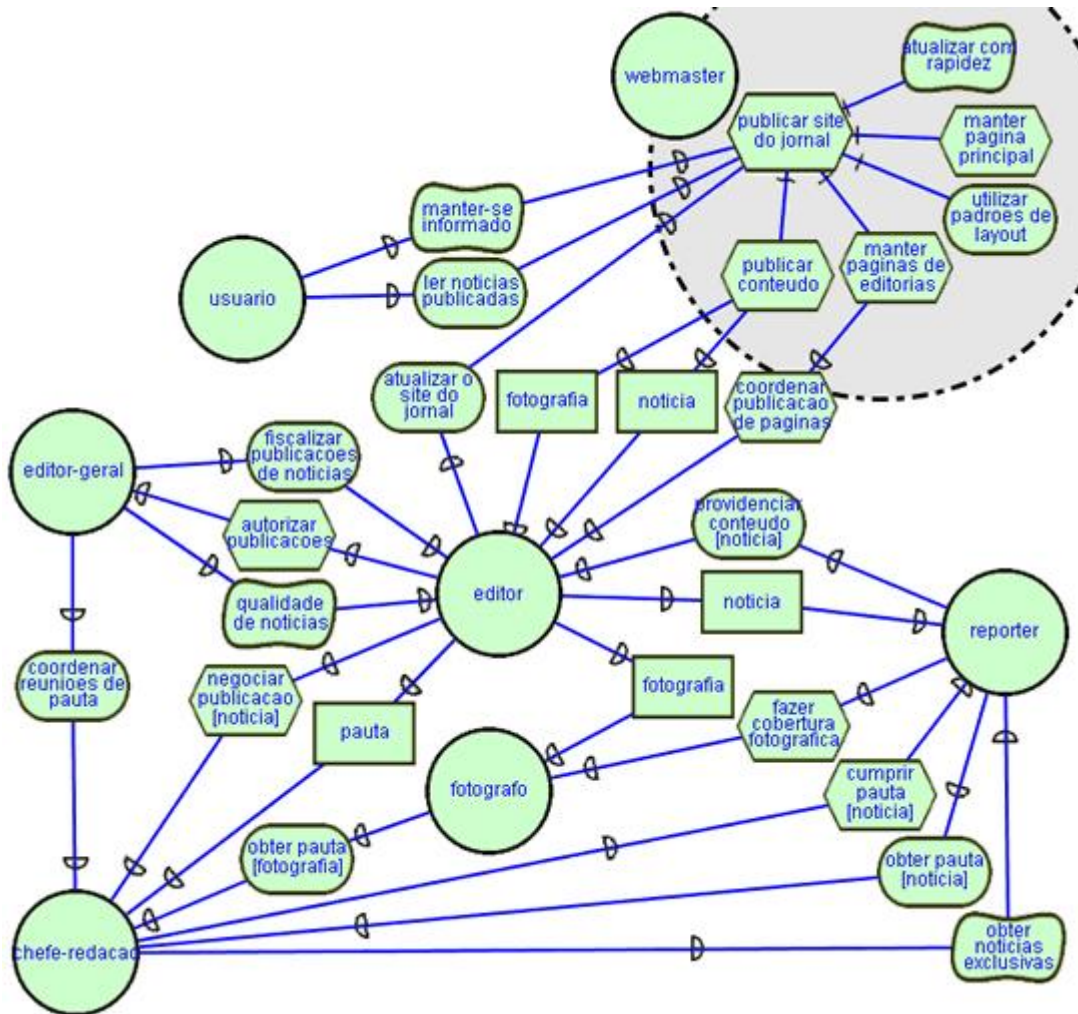


Figura 5.3: Modelo i\* de Razão Estratégica de uma redação de jornal para o ator *webmaster*.

O ator *repórter* tem como tarefa fundamental *cumprir pauta de notícias*. Esta tarefa é decomposta e considerada como realizada quando o ator atinge a meta *providenciar notícia* e realiza as tarefas de *receber pauta de notícia*, *anexar cobertura fotográfica* e *enviar notícia para editoria*. O repórter pode atingir a meta *providenciar notícia* através da meta-soft *furo de reportagem*, da tarefa *produzir matéria*, ou da meta *buscar notícia*. A busca de notícias, por sua vez, pode ser cumprida pelas tarefas *pesquisar na internet*, *pesquisar em agências de notícias* ou *pesquisar no arquivo do jornal*.

Com relação ao ator *chefe-redação*, seu principal objetivo é o *gerenciamento da edição* do jornal com determinada periodicidade. Os meios de atingir essa meta são: a meta-soft *manter-se sempre informado* ou a tarefa *coordenar reuniões de pauta*. Essa tarefa, no entanto, pode ser decomposta em duas outras tarefas: *negociar publicação de notícias* e *gerenciar alocação de pautas*. O gerenciamento da alocação de pautas dá-se através das tarefas *distribuir pauta de editoria*, *distribuir pauta de fotografias*, *distribuir pauta de notícias*, *controlar recebimento de notícias*, e da meta-soft *contornar imprevistos*. Os modelos SR para os atores *repórter* e *chefe-redação* podem ser encontrados no abaixo:

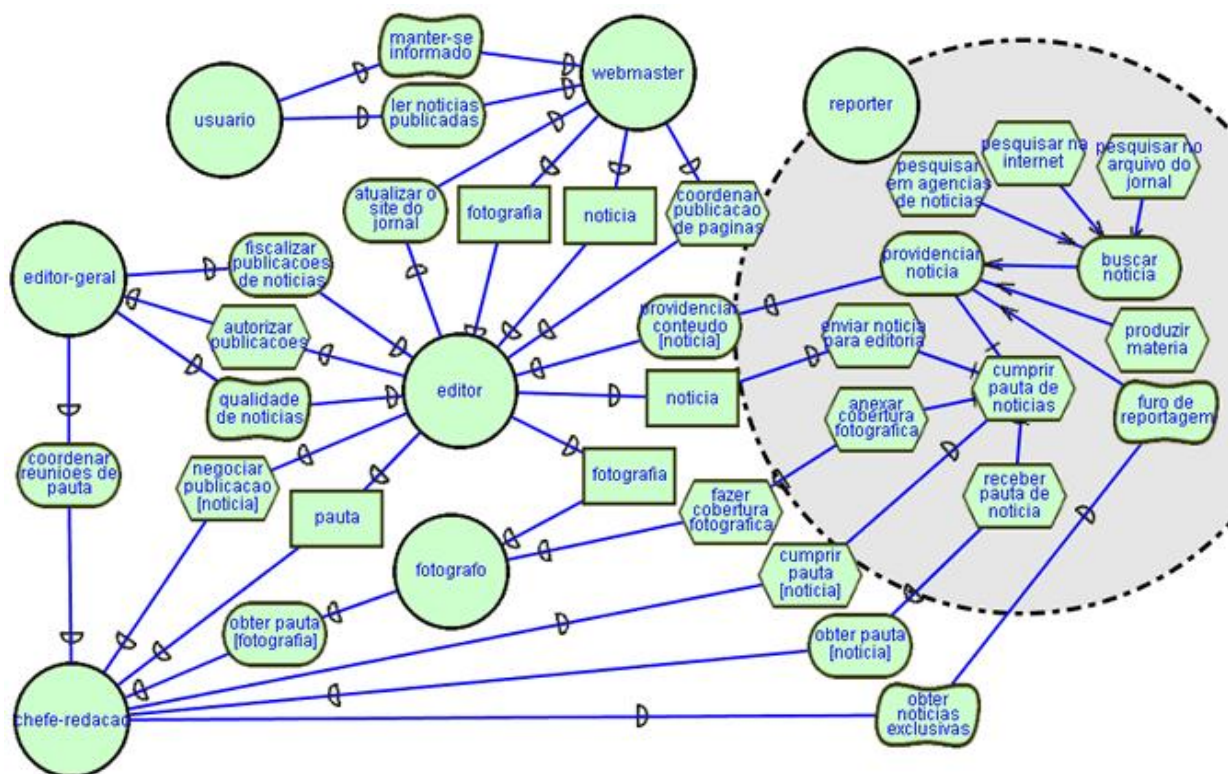


Figura 5.4: Modelo i\* de Razão Estratégica dos Requisitos Iniciais para o ator repórter.

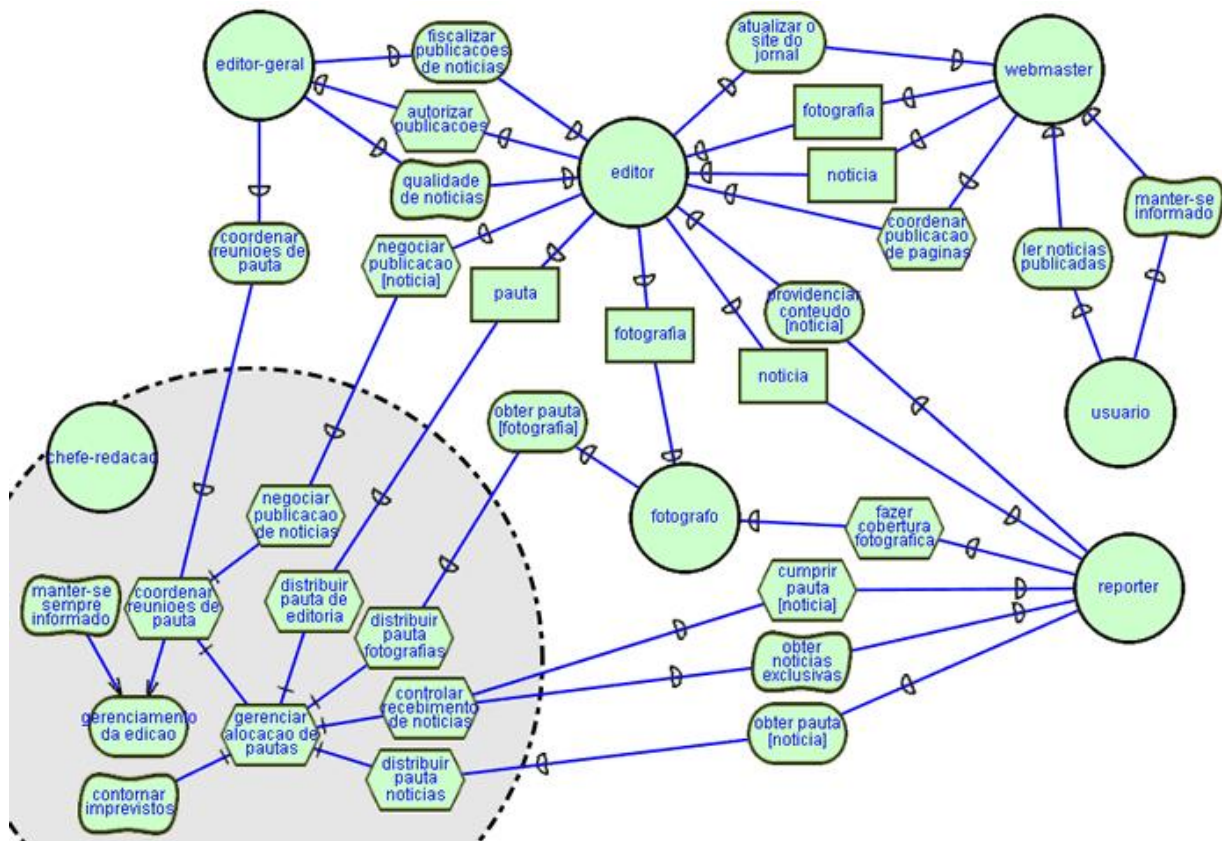


Figura 5.5: Modelo  $i^*$  de Razão Estratégica de uma redação de jornal para o ator *chefe-redação*.

### 5.3.2 Requisitos Finais

Como visto no Capítulo 3, a fase de Requisitos Finais de Tropos está preocupada com o refinamento dos modelos SD e SR produzidos na fase de Requisitos Iniciais. Este refinamento deve ser feito a partir da revisão e extensão dos modelos conceituais desenvolvidos na fase anterior, para incluir novos atores que representam tanto o sistema a ser desenvolvido quanto os seus subsistemas.

#### Modelo de Dependência Estratégica

O *Smart Journal* está representado no modelo SD descrito na Figura 5.6 através do ator *sistema gerenciador de notícias*. Neste modelo, os principais atores que interagem com o sistema são: o *usuário*, que representa o leitor do jornal na internet; a *agência de notícias*, a qual provê os conteúdos das notícias e fotografias a serem publicadas; o *editor* e o *editor-chefe*, cujas principais preocupações estão relacionadas ao gerenciamento da publicação das notícias.

Com relação ao *sistema gerenciador de notícias*, o ator *usuário* possui quatro dependências de meta-soft. Elas dizem respeito à exigência de *rapidez* no acesso às informações, *disponibilidade* e *adaptabilidade* do sistema, para que a também meta-soft *manter-se informado* seja atingida. A partir da interface do *Smart Journal* na Web, deve ser possível o *usuário* cumprir a meta de *navegar no site*. Além disso, o *usuário* depende do sistema para realizar a tarefa *buscar palavra-chave*, que deve



possibilitar a busca por notícias publicadas através de palavras-chave fornecidas como entrada para o sistema.

O sistema possui relacionamentos de dependência com os atores *editor* e *agência de notícias* para serem fornecidos os recursos *notícia* e *fotografia*. O ator *editor* também pode fornecer recursos do tipo *pauta* para o *sistema gerenciador de notícias*, bem como é responsável pela tarefa de *coordenar publicação de páginas* quando ela for realizada através da interface do sistema. Desta forma, o sistema também depende do *editor* para que seja realizado o objetivo *atualizar o site do jornal*. Por sua vez, a *agência de notícias* além de fornecer estes recursos, possui restrições de *segurança* e *integridade* do conteúdo associadas ao objetivo de *prover conteúdo* para o *sistema gerenciador de notícias*. O editor-chefe depende do sistema para atingir as metas de *gerenciar o sistema* e *fiscalizar publicações* de notícias, que depende do editor-chefe para realizar a tarefa de *autorizar publicações*. Por fim, as ligações de dependência de meta-soft *segurança* e *qualidade de notícia* são desejadas pelo *editor-chefe* em relação ao *sistema gerenciador de notícias*.

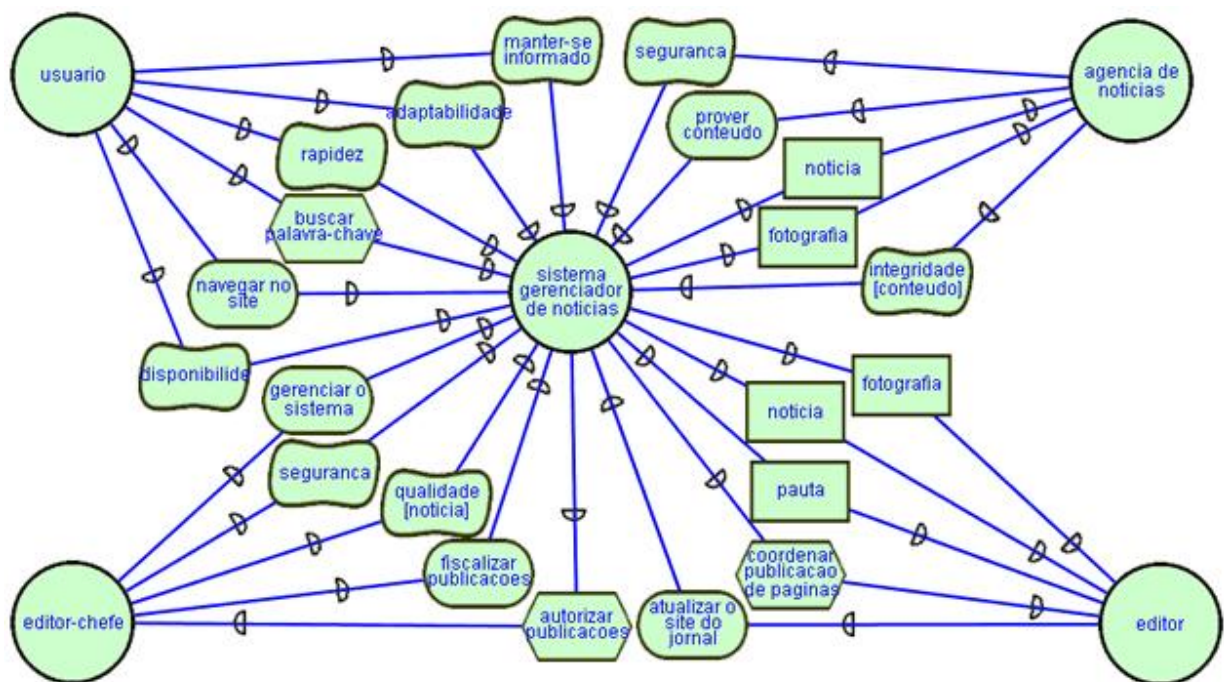


Figura 5.6: Modelo i\* de Dependência Estratégica dos Requisitos Finais.

### Modelo de Razão Estratégica

Uma vez que os atores principais e seus objetivos foram identificados, o modelo SR apresenta como esses objetivos podem ser atingidos através das contribuições dos atores. Como a intenção do sistema *Smart Journal* é automatizar parte do processo de publicação, alguns atores identificados para redação do jornal nos Requisitos Iniciais não fazem parte do modelo de relacionamentos externos com o ator que representa o sistema. Contudo, parte das tarefas executadas por esses atores será considerada para o sistema neste modelo de razões estratégicas.

Na Figura 5.7, podemos visualizar os principais objetivos identificados para o sistema, que são: *atualizar site*, *capturar profile*, *gerenciar site*, *controlar recebimento de conteúdo* e *coordenar editorias*. Esses objetivos, por sua vez, são apresentados numa estrutura de decomposição e finalidades, através de ligações do tipo meio-fins e ligações de decomposição de tarefas.

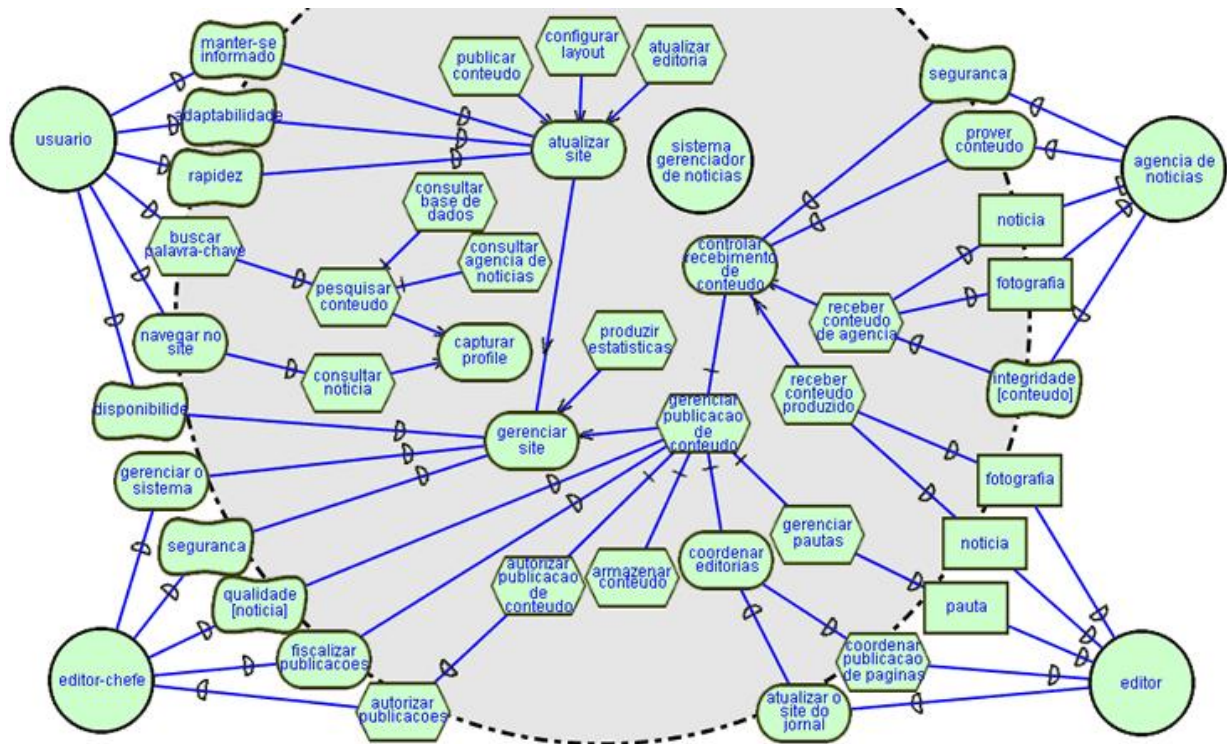


Figura 5.7: Modelo i\* de Razão Estratégica dos Requisitos Finais.

O objetivo de *atualizar site* é alcançado através da execução das tarefas *publicar conteúdo*, *configurar layout*, e *atualizar editoria*. Este objetivo por sua vez, é um meio de satisfazer o objetivo *gerenciar site*, que também é satisfeito pelas tarefas *produzir estatísticas* e *gerenciar publicação de conteúdo*. Esta última tarefa pode ser executada por meio das tarefas *autorizar publicação de conteúdo*, *armazenar conteúdo*, *gerenciar pautas*, e pelo objetivo *coordenar editorias*.

Para atingir o objetivo *capturar profile*, podem ser executadas duas tarefas por parte do ator usuário: *consultar notícia*, e *pesquisar conteúdo*. Esta última tarefa por sua vez pode ser decomposta nas tarefas *consultar base de dados* e *consultar agência de notícias*.

Já o objetivo *controlar recebimento de conteúdo*, pode ser atingido pelas tarefas *receber conteúdo da agência*, *receber conteúdo produzido* através do ator *editor*, e pela tarefa *gerenciar publicação de conteúdo*.

### 5.3.3 Projeto Arquitetural

A primeira atividade realizada na fase de projeto arquitetural é a seleção dos estilos arquiteturais usando como critério as qualidades desejadas identificadas na fase anterior. Como mostramos no Capítulo 3, Tropos utiliza o *framework* NFR para conduzir esta análise. A análise envolve refinar estas



qualidades, representadas como *softgoals*, em sub-metas que são mais específicas e mais precisas e então avaliar as alternativas de estilos arquiteturais comparando uma as outras, como mostrado na Figura 5.8.

Os estilos arquiteturais são apresentados como *softgoals* operacionalizados. As razões de projeto são representadas por metas-*soft* requeridas desenhadas como nuvens tracejadas. Estas podem representar informação de conteúdo (tais como prioridades) a serem consideradas e apropriadamente refletidas no processo de tomada de decisão. Marcas de exclamação (! (prioritário) e !! muito (prioritário)) são usadas para marcar prioridade de metas-*soft*. Uma marca de *check* “√” indica uma meta-*soft* cumprida, enquanto que uma cruz “X” rotula uma meta-*soft* que não pode ser cumprida.

A Tabela 5.1 resume o catálogo de correlação entre os estilos organizacionais e os atributos de qualidade [Kolp01] que guiam nossa escolha. As seguintes notações usadas pelo *framework* NFR [Chung00], *help*, *make*, *hurt*, *break*, respectivamente modelam contribuições parcial/positiva, suficiente/positiva, parcial/negativa e suficiente/negativa.

Tabela 5.1: Catálogo de Correlação.

Catálogo de Correlação	Previsibilidade	Segurança	Adaptabilidade	Coordenabilidade	Cooperatividade	Disponibilidade	Integridade	Modularidade	Agregabilidade
Estrutura Plana	<i>Break</i>	<i>Break</i>	<i>Hurt</i>			<i>Help</i>	<i>Help</i>	<i>Make</i>	<i>Hurt</i>
Estrutura em cinco	<i>Help</i>	<i>Help</i>		<i>Help</i>	<i>Hurt</i>	<i>Help</i>	<i>Make</i>	<i>Make</i>	<i>Make</i>
Pirâmide	<i>Make</i>	<i>Make</i>	<i>Help</i>	<i>Make</i>	<i>Hurt</i>	<i>Help</i>	<i>Break</i>	<i>Hurt</i>	
União Estratégica	<i>Help</i>	<i>Help</i>	<i>Make</i>	<i>Help</i>	<i>Hurt</i>	<i>Make</i>		<i>Help</i>	<i>Make</i>
Leilão	<i>Break</i>	<i>Break</i>	<i>Make</i>	<i>Hurt</i>	<i>Make</i>	<i>Hurt</i>	<i>Break</i>	<i>Make</i>	
Tomada de Controle	<i>Make</i>	<i>Make</i>	<i>Hurt</i>	<i>Make</i>	<i>Break</i>	<i>Help</i>		<i>Help</i>	<i>Help</i>
Comprimento de braço	<i>Hurt</i>	<i>Break</i>	<i>Help</i>	<i>Hurt</i>	<i>Make</i>	<i>Break</i>	<i>Make</i>	<i>Help</i>	
Contratação Hierárquica			<i>Help</i>	<i>Help</i>	<i>Help</i>	<i>Help</i>		<i>Help</i>	<i>Help</i>
Integração Vertical	<i>Help</i>	<i>Help</i>	<i>Hurt</i>	<i>Help</i>	<i>Hurt</i>	<i>Help</i>	<i>Break</i>	<i>Break</i>	<i>Break</i>
Apropriação	<i>Hurt</i>	<i>Hurt</i>	<i>Make</i>	<i>Make</i>	<i>Help</i>	<i>Break</i>	<i>Hurt</i>	<i>Break</i>	

Lembramos que os atributos de qualidade de software Segurança, Disponibilidade e Adaptabilidade foram apresentados no modelo de requisitos finais (Seção 5.3.2). Nesta fase eles serão guiarão o processo de seleção do estilo arquitetural apropriado (Figura 5.8). A análise envolve refinar estas qualidades, representadas como metas-*soft*, para sub-metas que são mais precisas e mais específicas e então avaliar os estilos arquiteturais alternativos contra elas.

### Escolha do estilo arquitetural

Disponibilidade. A rede de comunicação pode não ser confiável podendo causar perda de dados. Há um interesse crítico quanto à integridade dos dados e que o sistema faça aquilo que ele realmente se compromete em fazer em tempo pedido pelo usuário.

Segurança. O sistema por ser acessível através de um *browser*, expõe seus usuários ao risco com relação à segurança das informações manipuladas. São importantes, portanto os mecanismos que garantam a consistência e confidencialidade, bem como mecanismos de validação (autorização) e checagem (controle) de possíveis conflitos.

Adaptabilidade. Envolve a maneira como o sistema pode ser projetado a fim de permitir que as páginas *web* sejam atualizadas dinamicamente e de forma prática.

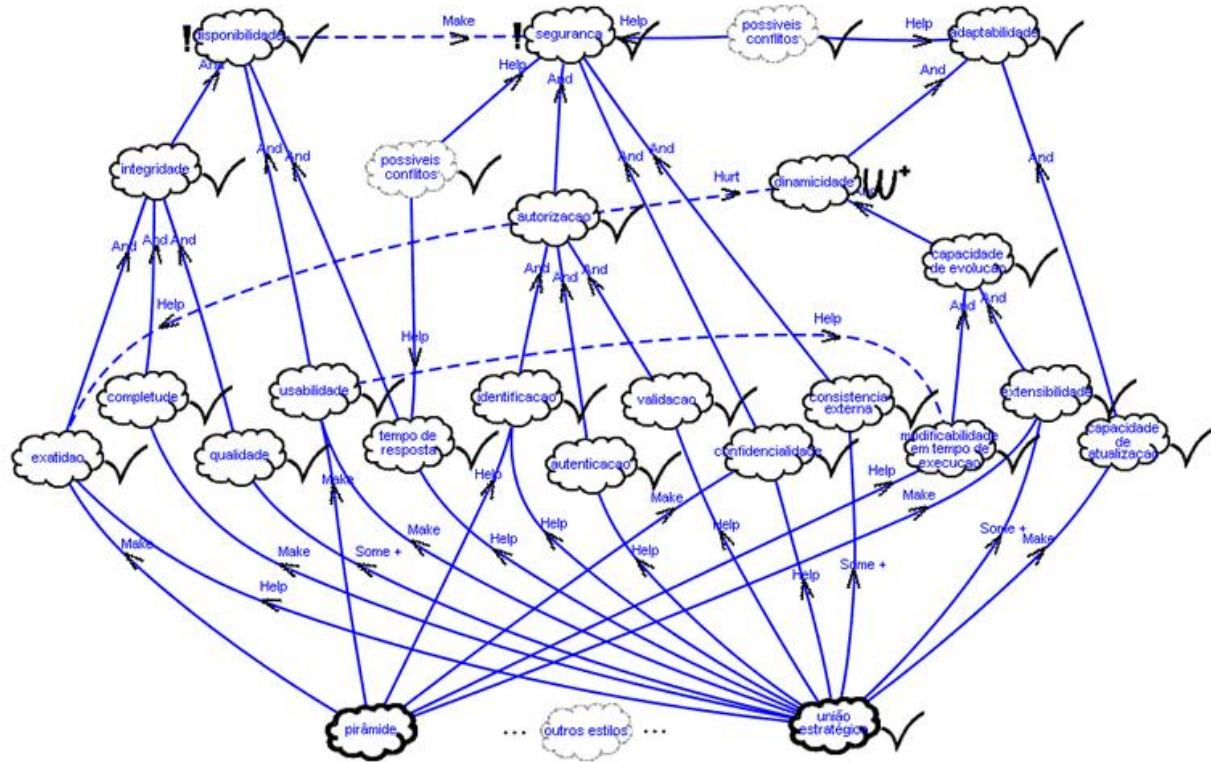


Figura 5.8: Grafo de interdependências de metas-soft do *Smart Journal*.

## Modelagem da arquitetura

Após analisarmos o grafo SIG apresentado na Figura 5.8, optamos por modelar a arquitetura do sistema a partir do estilo arquitetural organizacional *União Estratégica*. Uma vez que o sistema a ser desenvolvido trata-se de um simulador, os parceiros principais e secundários escolhidos para fazer parte da arquitetura equivalem a atores previamente identificados, que no mundo real realizariam as tarefas e teriam objetivos semelhantes a atingir. A Figura 5.7 sugere uma possível atribuição de responsabilidades para o *Smart Journal* seguindo o estilo União Estratégica [Figura 3.8]. Ele é decomposto em três atores que representam os parceiros principais, um ator que representa uma gerência comum e outros dois que representam parceiros secundários. Os parceiros principais identificados foram: *repórter*, *fotógrafo* e *editor*. A gerência comum será representada pelo *chefe de redação* e os parceiros secundários são o *webmaster* e *agência de notícias*.

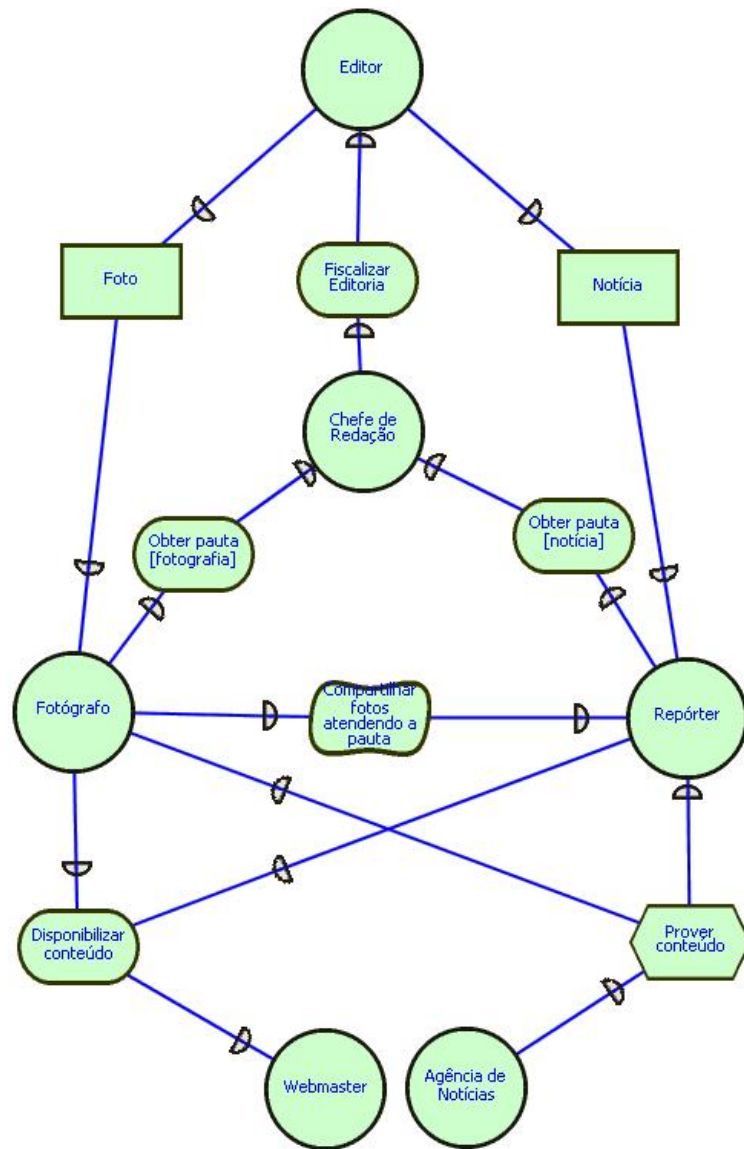


Figura 5.9: Arquitetura do *Smart Journal* em União Estratégica.

### Aplicação dos Padrões Sociais

Considerando a descrição do sistema segundo as Seções 5.3.1 e 5.3.2, listamos as seguintes considerações em relação aos padrões sociais a serem adotados:

- O uso padrão *Wrapper* permite os Repórteres e Fotógrafos interagirem com a Agência de Notícias e com o *Webmaster* e manipular a informação trocada entre eles.
- O uso do padrão *Mediator* permite o Chefe de Redação coordenar a cooperação entre os Editores a fim de fiscalizar a editoria, distribuindo pautas.
- Um agente *Matchmaker* deve ser adicionado ao sistema a fim de guardar a localização do provedor de serviço dos agentes (Editor e Editor-Chefe) e responder as requisições dos clientes (Editor-Chefe e *Webmaster*) permitindo os clientes interagirem com os fornecedores de serviços. O DF de JADE faz o papel de *Matchmaker*.

- Cada Editor trabalhar como um agente *Broker*, selecionando um ou mais reporters/fotógrafos para achar conteúdos para cobrir uma pauta específica (Política, Crise Política no Brasil).

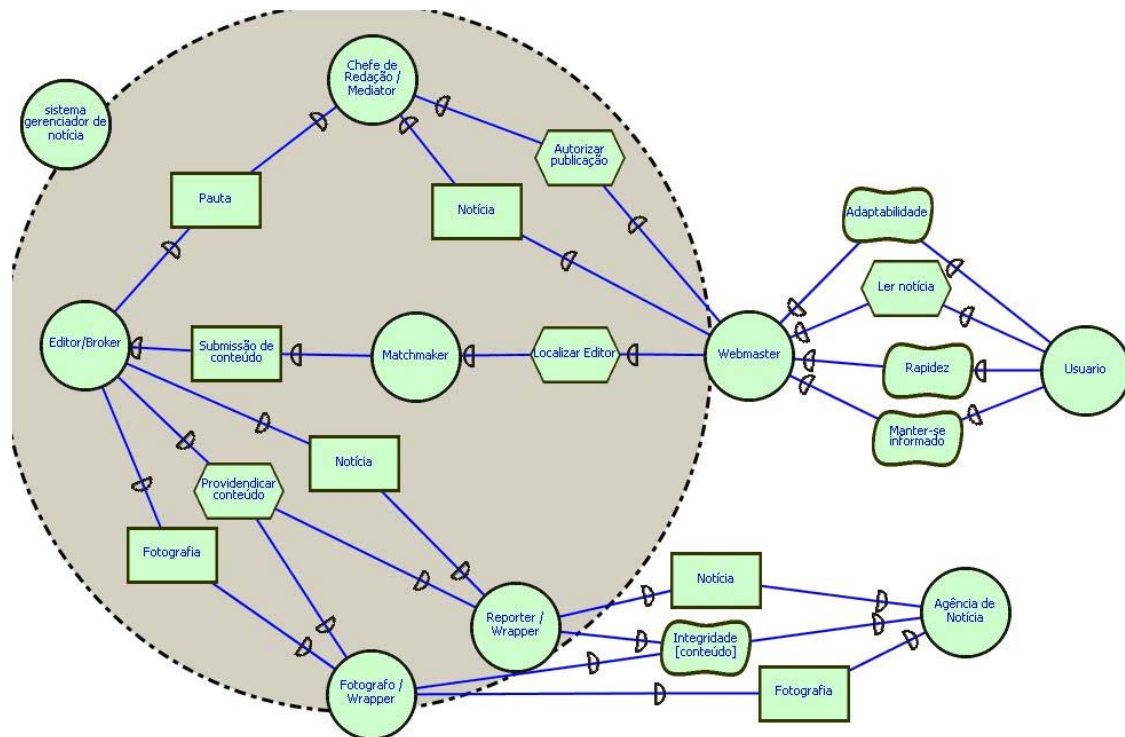


Figura 5.10: Decomposição do *Smart Journal* com padrões sociais.

### 5.3.4 Projeto Detalhado

Nesta fase do Tropos, vamos então especificar o agente, detalhando suas capacidades e planos usando o diagrama de atividades, como também protocolos de comunicação e coordenação.

Segue abaixo os diagramas de atividade que representam as capacidades dos agentes *Editor* e *Webmaster*, que serão o foco da nossa proposta de implementação:

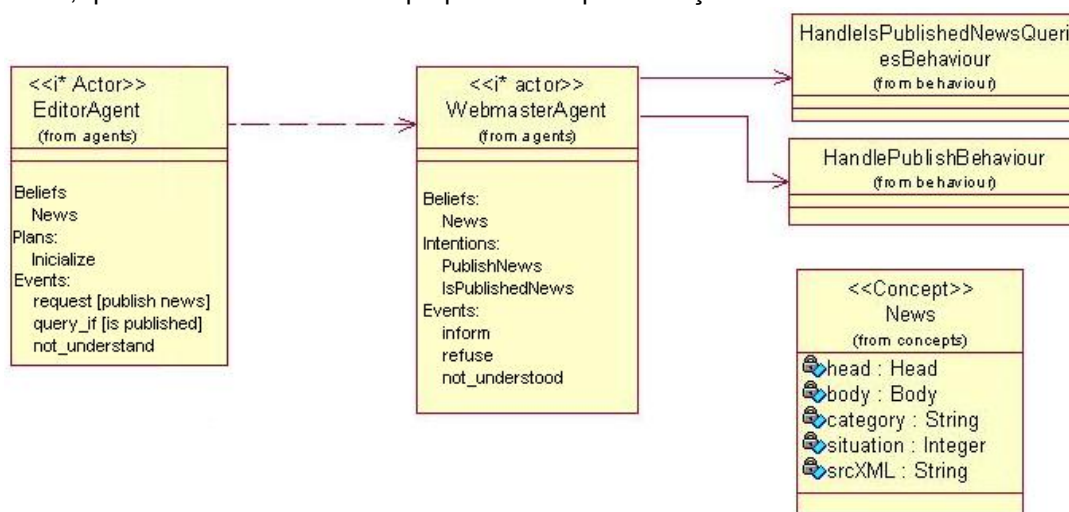


Figura 5.11: Diagrama de classe para o *Smart Journal*.

Os diagramas de seqüência abaixo, capturam algumas das interações entre os agentes Editor e *Webmaster*:

O Editor pode solicitar ao *Webmaster* que publique uma notícia (REQUEST), este por sua vez pode recusar a publicação se a notícia já foi publicada (REFUSE), enviar uma mensagem identificando que a requisição não foi entendida (NOT-UNDERSTOOD), ou ainda aceitar a requisição (ACCEPT). Sendo aceita a requisição, o *Webmaster* tenta publicar a notícia, e então envia ao Editor uma mensagem de informação (INFORM) de sucesso se esta foi executada sem problemas e de falha se ocorreu algum problema.

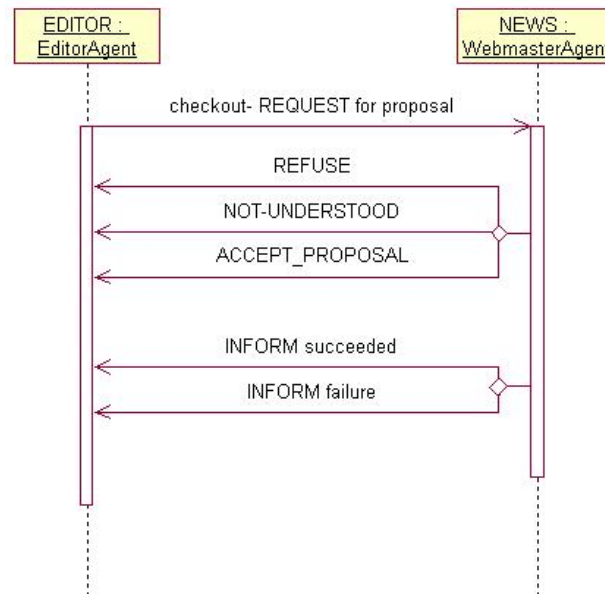


Figura 5.12: Diagrama de seqüência, interação entre os agentes Editor e *Webmaster* – Publicar Notícia.

O Editor pode ainda solicitar que o *Webmaster* verifique se uma determinada notícia já foi publicada (QUERY\_IF). O *Webmaster* então responde ao Editor com uma mensagem de INFORM se a notícia já foi publicada.

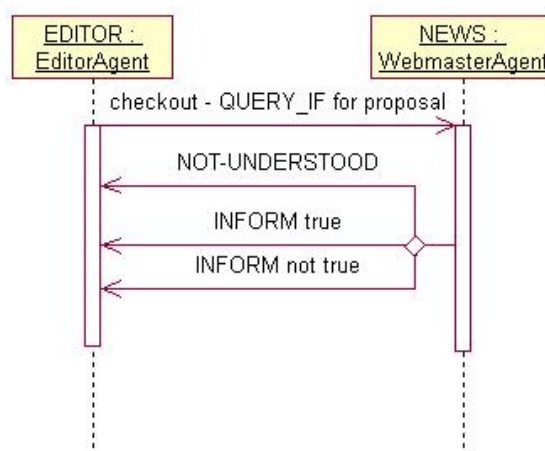


Figura 5.13: Diagrama de seqüência, interação entre os agentes Editor e *Webmaster* – Consultar se a Notícia já foi publicada.



## 5.4 Implementação em JADEX

Para implementação do sistema, propomos a reutilização algumas classes (ontologia, conceitos, predicados, comportamentos, etc) da implementação deste mesmo sistema já realizado em JADE. Escolhemos utilizar a extensão JADEX, por que (i) nos permite desenvolver agentes orientados a objetivos seguindo o modelo BDI e utilizando todos os recursos da plataforma JADE (ferramentas de gerenciamento), além de oferecer mais duas ferramentas de gerenciamento (*Debugger* e *Logger*) e um novo agente *Introspector* que possibilita a visualização dos conceitos BDI dos agentes, (ii) nos permite reutilizar código escrito para a JADE (comportamentos e ontologia), (III) e facilita a implementação de agentes em Java, e, portanto permite o reuso de várias ferramentas e bibliotecas. Para cada agente, criamos seu ADF, e nele identificamos seus respectivos comportamentos, linguagens e ontologia utilizadas [Figura 5.14].

Desta forma, os agentes a serem implementados podem reutilizar seus comportamentos já implementados em JADE:

- Editor
  - Comportamentos: Solicitar publicação de notícia, checar se uma notícia já foi publicada, checar se a notícia pertence a uma determinada categoria, checar se a notícia já foi expirada.
- Webmaster
  - Comportamentos: Publicar notícia, informar se uma notícia já foi publicada, informar a qual categoria uma notícia pertence, informar se uma notícia é expirada.

Sugerimos aqui, de forma bem simples, um mapeamento entre a modelagem realizada com o *i\**, a arquitetura BDI de agentes e os conceitos inerentes a plataforma JADEX. Para cada dependência de recurso identificada através dos artefatos produzidos pela aplicação da metodologia Tropos, identificamos conceitos que estão refletidos na ontologia. Cada dependência de tarefa e de objetivo refletiu em ações dos agentes, que podem envolver os conceitos definidos na ontologia. As dependências de objetivos sugerem a troca de mensagens entre os agentes [Figura 5.14].

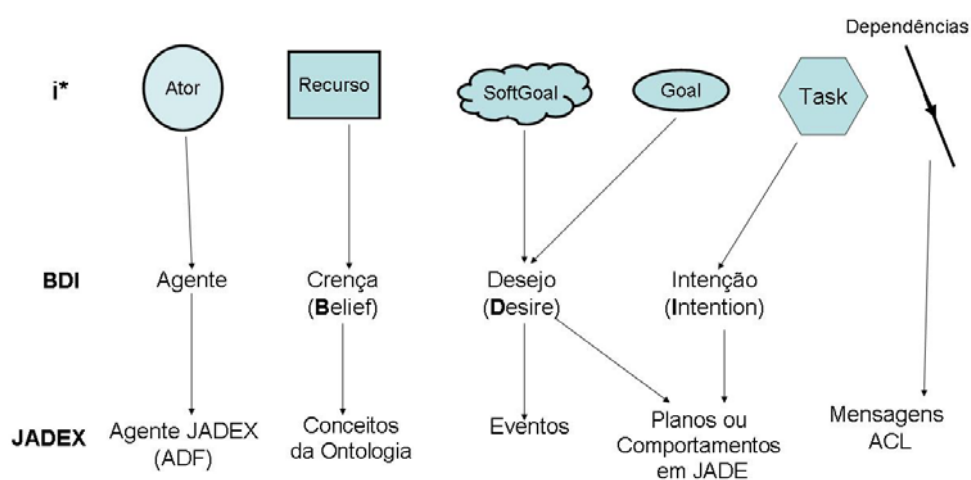


Figura 5.14: Mapeamento *i\**/BDI/JADEX.

Os tipos de Mensagens ACL, que serão utilizadas são:

- REQUEST: para requisição de publicação de notícias,
- REFUSE: para recusar uma solicitação (REQUEST),
- ACCEPT: para aceitar uma solicitação (REQUEST),
- INFORM: para informar sucesso ou falha na execução de alguma tarefa (comportamento ou intenção do agente), true ou false em resposta a mensagens do tipo QUERY,
- QUERY-IF: para solicitar que se verifique alguma proposição,
- NOT-UNDERSTOOD: para informar que uma mensagem não foi entendida pelo agente.

Os conceitos da ontologia, como já dissemos, foram identificados através dos recursos representados nos modelos SD e SR. Assim, identificamos conceitos tais como: notícias (*News*), pautas (*Guideline*) e fotografias (*Media*). Sendo o conceito de notícia associado à representação da mesma no sistema como um arquivo XML seguindo a estrutura proposta pelo NITF [NITF04]. O NITF (*News Industry Text Format*) é um projeto lançado pela IPTC (*International Press Telecommunications Council*) nos anos 90 a fim de criar um padrão e fornecer uma plataforma comum aos serviços de notícias e jornais a possibilidade de trocar conteúdos de forma mais fácil. Dentre as instituições que usam seu padrão estão: *The Associated Press*, *Dow Jones*, *The New York Times*, *Deutsche Presse-Agentur*, dentre outros.

Então, resumidamente, a notícia no *Smart Journal* poderá ser representada por um arquivo XML, seguindo as especificações do NITF e constando de: um cabeçalho (*NewsHead*) e seus elementos (título, data de expiração, data de publicação, hora da publicação, lista de palavras-chave, etc.), e um corpo (*Body*) e seus elementos (*abstract*, títulos secundários, parágrafos, etc.).

**Exemplo 5.1: Exemplo de representação de notícias segundo o padrão NITF.**

```
<?xml version="1.0"?>

<?xml-stylesheet type="text/xsl" href="../xsl/nitf-to-html.xsl"?>

<!DOCTYPE nitf SYSTEM "nitf-3-2.dtd">

<nitf>

<head>

<title>Norfolk Weather and Tide Updates</title>

<tobject tobject.type="news">

<tobject.subject tobject.subject.type="Weather"/>

</tobject>

<docdata>

<date.expire norm="20120226T093000-0500"/>

<key-list> <keyword key="fishing"/> ... </key-list>
```

```

</docdata>

</head>

<body>

  <body.head>

    <hedline> <h1>Weather in Norfolk</h1> </hedline>

    <note><p>This sample article ... </p></note>

    <abstract>

      <p>The weather in Norfolk today...</p>

    </abstract>

  </body.head>

  <body.content>

    <media media-type="image" style="align:right">

      <media-reference mime-type="image/jpeg" height="185"
      width="278" source="high-tide.jpg" alternate-text="...">

        </media-reference> ...

    </media>

    <p>The weather was superb today in Norfolk, Virginia...</p>

    <block><p>There are many... </p></block>

  </body.content>

  <body.end><tagline>A NITF contribution.</tagline></body.end>

</body>

</nitf>

```

O sistema então, pode ser dividido em três camadas principais: interface, comunicação, e sistema JADEX.

A camada responsável pela interface com o usuário deverá ser implementada utilizando a tecnologia JSP, HTML e XML (para exibição das notícias). A partir dessa camada, temos a exibição do conteúdo das notícias e a interface para administração do sistema. A partir desta última interface, inicializamos então os agentes da aplicação *Smart Journal* e a plataforma JADE/JADEX, através do agente RMA, para que a troca de mensagens entre os agentes possa ser visualizada.

A segunda camada deverá ser construída utilizando-se da tecnologia *servlets* para servir de canal de interação entre a GUI e a aplicação em JADEX. Por exemplo, para enviar uma notícia para publicação através do site do jornal, podemos selecionar o arquivo XML correspondente à notícia, e através do JSP o XML é submetido para o sistema através de um *servlet*.



A última camada do sistema, será o sistema implementado de acordo com a arquitetura JADEX. Para melhor estruturação do mesmo, propomos a divisão dos arquivos em pacotes por conceitos equivalentes. Por exemplo, no pacote ontologia (*src.ontologia*) está o arquivo com a definição da mesma, e dentro deste pacote estão organizados os conceitos utilizados por ela, por exemplo, pacotes de *src.ontology*: *src.ontology.concepts*, *src.ontology.predicates*, etc.

Como já dissemos, o sistema deverá possuir dois módulos: a interface para usuário leitores, e a interface para os usuários representados pelos agentes para a administração do sistema. No primeiro módulo serão apresentadas as notícias mais atuais cadastradas no sistema, um menu para que as notícias sejam visualizadas por categoria, um campo para busca de notícias na base do sistema, e um campo de *login* para a área restrita (administração do sistema). No último módulo são apresentadas as funcionalidades agrupadas por atores (agentes). Cada link desses abrirá uma lista das tarefas que aquele ator selecionado pode executar. É a partir da solicitação da execução de uma tarefa que a plataforma JADE/JADEX será iniciada.

## Capítulo

# 6 Conclusão

Neste trabalho apresentamos a aplicação da metodologia Tropos e da plataforma de desenvolvimento JADEX, no desenvolvimento de sistemas multi-agentes.

É essencial para a ESOA a cobertura completa do processo de desenvolvimento de software. As metodologias orientadas a agente são inerentemente intencionais, baseadas em conceitos como agente, meta, plano, etc. Isto mostra claramente a necessidade de se focar no ambiente (organizacional) onde o sistema em desenvolvimento eventualmente vai operar.

Portanto, entender tal ambiente pode reduzir consideravelmente a incompatibilidade entre o sistema e seu ambiente operacional. Neste contexto, fica clara a relevância do Projeto Tropos no desenvolvimento de software orientado a agentes. A proposta Tropos, embora em constante evolução, ainda apresenta as vantagens de levar a arquiteturas de software mais flexíveis, robustas e abertas e oferecer um *framework* coerente que engloba todas as fases de desenvolvimento de software, dos requisitos iniciais até a implementação.

Além disso, Tropos é consistente com a próxima geração de paradigma de programação, a programação orientada a agentes, que está ganhando uma base sólida em importantes áreas de aplicação, tais como telecomunicações, comércio eletrônico e sistemas baseados na *web*.

A plataforma que utilizamos para desenvolvimento do estudo de caso, JADEX, se mostrou eficiente na representação dos conceitos extraídos da análise realizada com Tropos, mas ainda há deficiência em relação à implementação. Ainda há poucos exemplos, base histórica, de sistemas que se utilizam dessa plataforma e usem as tecnologias que nós propomos (JSP, *Servlets*), os exemplos que existem são bem simples e não mostram na prática, principalmente como se pode iniciar agentes de um mecanismo externo como, por exemplo, através de um *Servlets*. Como ainda é uma plataforma recente (2003), JADEX possui guias (*toolguide*, *userguide*, tutorial) que estão incompletas e uma API ainda com alguns problemas (*bugs*) a serem resolvidos, que foi o caso do método sugerido no tutorial para iniciar agentes através de um mecanismo externo fazendo a carga do ADF, *SXML.loadAgentModel*.

Desta forma, para trabalhos futuros temos como proposta, o desenvolvimento do sistema aqui modelado e de acordo com a nossa proposta de implementação. Observamos que há a necessidade de realização de outros estudos de caso para que se possa ter uma base histórica, bem validada da aplicação dessa plataforma em ambientes de aplicações para a *Web*.

## 7 Referências Bibliográficas

- [Alencar03] Alencar, F. M. R., Pedroza, F. P., Castro, J. F. B., and Amorim, R. C. O.: **New Mechanisms for the Integration of Organizational Requirements and Object Oriented Modeling**. In: VI Workshop on Requirements Engineering, Piracicaba, p. 109-123, (2003).
- [Bellifemine03] Bellifemine, F., Caire, G., Trucco, T. and Rimassa, G. (2005) **Jade Programmer's Guide - JADE 3.3**. Disponível em: <http://sharon.cselt.it/projects/jade/>, Last access in March.
- [Bigus98] Bigus, J. P., Bigus, J., **Constructing Intelligent Agents with Java: a programmer's guide to smarter applications**. John Wiley and Sons, 1998.
- [Braubach04] Braubach, L., Pokahr, A. and Lamersdorf, W. (2004) "**Jadex: A Short Overview**", In: Main Conference Net.ObjectDays 2004, AgentExpo.
- [Bresciani04] Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: **Tropos: An Agent -Oriented Software Development Methodology**. In Autonomous Agents and Multi -Agent Systems v. 8 (3): 203-236, May 2004.
- [Booch99] Booch, G., Rumbaugh, J., Jacobson, I.: **The Unified Modeling Language User Guide**, The Addison Wesley Object Technology Series, Addison Wesley, (1999).
- [Bordini01] Bordini, R. H. ; Vieira, R. ; Moreira, A. F., **Fundamentos de Sistemas Multiagentes**. Anais do XXI Congresso da SBC (SBC 2001), XX Jornada de Atualização em Informática (JAI 2001), 30 Julho - 3 de Agosto, 2001, v. 2, p. 3-41. Disponível em: <http://www.inf.unioeste.br/~cbrizzi/FSMA-bordini.pdf>
- [Castro01] Castro, J., Kolp, M. and Mylopoulos, J.: **A requirements-driven development methodology**. In Proc. of the 13th Int. Conf. on Advanced Information Systems Engineering, CAiSE'01, pages 108–123, Interlaken, Switzerland, June 2001.
- [Castro02] Castro, J. Kolp, M., Mylopoulos, J.: **Towards Requirements-Driven Information Systems Engineering: The Tropos Project**. Information Systems Journal, 27: 365-89. 2002.
- [CENADEM04] CENADEM, **Centro Nacional de Desenvolvimento do Gerenciamento da Informação**, 2004. Disponível em: <http://www.cenadem.com.br/>.
- [Chung00] Chung, L. K., Nixon, B. A., Yu, E., Mylopoulos, J.: **Non-Functional Requirements in Software Engineering**. Kluwer Publishing, (2000).
- [Cysneiros02] Cysneiros, G.: **Ferramenta para o Suporte do Mapeamento da Modelagem Organizacional em i\* para UML**. Dissertação de Mestrado, Centro de Informática, Universidade Federal de Pernambuco, 2002.
- [Deitel03] Deitel, H. M. & Deitel, P. J.: **Java: Como Programar**. 4ª ed., Bookman, (2003).
- [Ferber99] Ferber, J., **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**. Addison-Wesley. (1999).
- [FIPA02] FIPA, **FIPA ACL Message Structure Specification**. Disponível em: <http://www.fipa.org/specs/aclspecs.tar.gz>.

- [FIPA04] **FIPA (The Foundation for intelligent agents)**, Disponível em: <http://www.fipa.org>, 2004.
- [FIPA-OS05] **FIPA-OS: A component-based toolkit enabling rapid development of FIPA compliant agents**. Disponível em: <http://fipa-os.sourceforge.net/>, 2005.
- [Garcia03] Garcia, A., Lucena, C., Zambonelli, F., Omicini, A., Castro, J. (editors). Book: **Software Engineering for Large-Scale Multi-Agent Systems: Research Issues and Practical Applications**. Lecture Notes in Computer Science - LNCS Vol. 2603, State-of-the-Art Survey. 284 pages. 2003.
- [Giorgini05] Giorgini, P., Kolp, M., Mylopoulos, J., Castro, J.: **Tropos: A Requirements-Driven Methodology for Agent-Oriented Software**. In B. Henderson-Sellers and P. Giorgini (Eds) Agent-Oriented Methodologies, Idea Group, (2005).
- [Grasshopper02] **Grasshopper**, Disponível em: <http://www.grasshopper.de>, 2002.
- [JACK05] **JACK Intelligent Agents**. <http://www.agent-software.com/>, 2005.
- [JADE04] **JADE (Java Agent DEvelopment Framework)**, Disponível em: <http://jade.cselt.it/>, 2004.
- [JADEX04] **JADEX (Java Agent DEvelopment eXtension) A Framework to BDI Agent System**, Disponível em: <http://vsiis-www.informatik.uni-hamburg.de/projects/jadex/>, 2004.
- [Jennings99] Jennings, N. R.: **On agent-based software engineering**. Department of Electronics and Computer Science, University of Southampton, Southampton SO17 1BJ, UK, Setembro, 1999. Disponível em: <http://www.iiaa.csic.es/~puyol/SEIAD2001/publicacions/aij2000.pdf>.
- [Jennings00] Jennings, N.: **On Agent-Based Software Engineering**. In Bradshaw, J. (ed): Handbook of Agent Technology, AAAI/MIT Press, (2000).
- [Jesus03] Jesus, A.: **Sistemas Tutores Inteligentes Uma Visão Geral**, RESI – Revista Eletrônica de Sistemas de Informação, ISSN 1677-3071, Dezembro, 2003. Disponível em: <http://www.presidentekennedy.br/resi/>.
- [Kolp01] Kolp, M., Castro, J., Mylopoulos, J.: **A social organization perspective on software architectures**. In Proc. of the 1st Int. Workshop From Software Requirements to Architectures (5–12). STRAW'01, Toronto, Canada, (2001).
- [Kolp02] Kolp, M., Giorgini, P., Mylopoulos, J.: **Information Systems Development through Social Structures**. 14<sup>th</sup> International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia, Italy, (2002).
- [Kolp05] Kolp, M., Do, T. T., Faulkner, S., Hoang, H. T. T.: **Introspecting Agent Oriented Design Patterns**. In: S. K. Chang (Eds), Advances in Software Engineering and Knowledge Engineering, vol. III, World Publishing, (2005).
- [Kotonya98] Kotonya, G. and Sommerville, I.: **Requirements Engineering – Processes and Techniques**. John Willy & Sons, 1998.
- [LEAP02] **LEAP (Lightweight Extensible Agent Platform)**, Disponível em: <http://leap.crm-paris.com/>, 2002.
- [Mangan01] Mangan, P. K. V., **Implementação de Sistemas Multiagentes**. Disponível em: <http://www.cos.ufrj.br/~kayser/cos761/MonografiaAgentes.pdf>
- [Newell93] Newell, A.: **Reflections on the Knowledge Level**. Artificial Intelligence, 59:31--38, 1993.
- [NITF04] NITF, **News Industry Text Format**, IPTC – International Press Telecommunications Council, 2004. Disponível em: <http://www.nitf.org/>

- [Odell00] Odell, J., Parunak, H., Bauer, B.: **Extending UML for Agents**. In: Wagner, G. Lesperance, Y., and Yu, E. (Eds.), *Proceedings of the Agent-Oriented Information Systems, Workshop at the 17<sup>th</sup> National Conference on Artificial Intelligence*, (2000).
- [Odell01] Odell, J., Parunak, H.V. D., Bauer, B. **Representing Agent Interaction Protocols in UML. Agent-Oriented Software Engineering**, Paolo Ciancarini and Michael Wooldridge eds. (121-140), Springer-Verlag, Berlin, (Held at the 22nd International Conference on Software Engineering (ISCE)) (2001).
- [Parreiras04] Parreiras, F, **Do GED à gestão do conhecimento foi um pulo**, Maio, 2004. Disponível em: <http://webinsider.uol.com.br/vernoticia.php/id/2128>.
- [Pedroza04] **Ferramentas para Suporte do Mapeamento da Modelagem i\* para a UML: eXtended GOOD – XGOOD e GOOSE**.
- [Pokahr05] Pokahr, A., Braubach, L., **Jadex: User Guide**, 2005. Disponível em: <http://prdownloads.sourceforge.net/jadex/userguide-0.932.pdf?download>
- [PRS] PRS, **Procedural Reasoning System**, Disponível em: <http://www.ai.sri.com/~prs>
- [Publique03] Publique!, 2003. Disponível em: <http://www.fabricadigital.com.br/publique/system/hlp/HelpPub222.htm>
- [Rational99] Rational Software Corp. et al.: **Unified Modelling Languages Semantic, version 1.3**. June 1999. Disponível em: [www.rational.com/uml/index.jtmpl](http://www.rational.com/uml/index.jtmpl)
- [Schwambach04] Schwambach, M., Pezzin, J., Falbo, R.: **OplA: Uma Metodologia para o Desenvolvimento de Sistemas Baseados em Agentes e Objetos**. JIISIC'04, 4ª Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del Conocimiento, Novembro, 2004. Disponível em: <http://www.inf.ufes.br/~falbo/download/pub/2004-JIISIC-3.pdf>.
- [Shahmehri98] Shahmehri, N., Lambrix, P., **Notes on Intelligent Software Agents**, Agosto, 1998. Disponível em: <http://www.ida.liu.se/labs/iislab/courses/Agents/paper/preface.html>
- [Silva03] Silva, C. T. L. L.: **Detalhando o projeto arquitetural no desenvolvimento de software orientado a agentes: O caso Tropos**. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco, Fevereiro, 2003.
- [SKWYRL03] **SKWYRL: Social arChitectures for Agent Software Systems EngineeRing**, 2003.
- [Sommerville01] Sommerville, I.: **Software Engineering**, Ed.6. Addison Wesley (2001).
- [Weiss00] Weiss, G.: **Multi-Agent Systems: A Modern Approach to Distributed Artificial Intelligence**, Ed. 2. Second Printing Massachusetts Institute of Technology, (2000).
- [Weiss02] Weiss, G.: **Agent Orientation in Software Engineering. Knowledge Engineering Review**, Vol. 16, n. 4, (2002): 349-373.
- [Wooldridge95] Wooldridge, M., Jennings, N.: **Intelligent Agents: Theory and Practice**. Disponível em: <http://www.doc.mmu.ac.uk/STAFF/mike/ker95/ker95-html.html>.
- [Wooldridge00] Wooldridge, M., Jennings, N. R., Kinny, D., **The Gaia Methodology for Agent-Oriented Analysis and Design, Journal of Autonomous Agents and Multi-Agent Systems**, 3 (3):285-312 (2000).
- [Wooldridge02] Wooldridge, M.: **Introduction to Multi-Agent Systems**. Jonh Wiley and Sons, New York (2002).
- [WQoS03] Projeto WQoS, **Núcleo Brasileiro de JADE (Java Agent DEvelopment Framework)**. Disponível em: <http://qos.tecnolink.com.br/>, 2003.
- [Yu95] Yu, E.: **Modelling Strategic Relationships for Business Process Reengineering**, Ph.D. thesis. Dept. of Computer Science, University of Toronto (1995).

- [Yu01] Yu, E.: **Agent-Oriented Modeling: Software Versus the World**. In Agent-Oriented Software Engineering AOSE-2001 Workshop Proceedings, Montreal, Canada - May 29th 2001. LNCS 2222.
- [Yu05] Yu, E. and Liu, L.: **OME (Organization Modeling Environment)**, Disponível em: <http://www.cs.toronto.edu/km/ome> (Home Page do Projeto).
- [Zambonelli03] Zambonelli, F., Jennings, N. R. and Wooldridge, M. **Developing Multiagent Systems: the Gaia Methodology**. ACM Trans on Software Engineering and Methodology, 12(3): 317-370, 2003.
- [ZEUS03] **ZEUS**, Disponível em: <http://more.btexact.com/projects/agents/zeus/index.htm>, 2003.