



Pós-Graduação em Ciência da Computação

Projeto e Implementação de Sistemas Multi-Agentes: O Caso Tropos

Por

Ismênia Galvão Lourenço da Silva

Dissertação de Mestrado



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
<http://www.cin.ufpe.br/~posgraduacao>

RECIFE, AGOSTO/2005

Ismênia Galvão Lourenço da Silva

Projeto e Implementação de Sistemas Multi-Agentes: O Caso Tropos

Dissertação apresentada à Coordenação da Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Jaelson Freire Brelaz de Castro

Co-orientadora: Profa. Dra. Patrícia C. A. R. Tedesco

RECIFE, AGOSTO/2005

Para minha mãe, Glória.

Agradecimentos

Ninguém que deseje realizar tudo sozinho, que queira sempre ser o dono da verdade, ou que trabalhe apenas para benefício próprio, pode experimentar o verdadeiro sabor da felicidade e do sucesso.

Aprendi, mais do que posso imaginar, sobre muitas coisas da vida ao realizar o mestrado. Essas coisas não cabem neste documento, mas nele estão registrados parte desse aprendizado e meus profundos agradecimentos:

A Deus, à Nossa Senhora e aos Santos Anjos, pela proteção e bênçãos recebidas.

À Mainha, por sempre ter acreditado, apoiado e conduzido sem medir esforços.

A Painho e meus irmãos (Diogo, Laurinha, Marcos, Rodrigo e Fernando), pela torcida e compreensão.

À minhas tias Carmita e Guguia por toda assistência, carinho e preocupação.

À minha querida avó Carminha e todos da família. Obrigada pelos conselhos e orações.

A Bruno Celso, por seu amor incondicionalmente real, e à sua família pelo apoio recebido.

A Jaelson, pela verdadeira orientação e amizade, pelas oportunidades e confiança depositada.

À Patrícia, Lourdes, Ed e todos os professores que colaboraram para a realização deste mestrado.

Aos colegas do LER, em especial Bárbara e Carla, por todas as contribuições recebidas.

Às colegas de “república”, principalmente Vivi, pela boa convivência e pela grande amizade.

A Luiz Costa, por acreditar que a experiência acadêmica também tem seu valor no mercado.

A todos os meus amigos: os novos ou antigos; os que acompanharam de perto ou de longe.

A CAPES, pelo fomento recebido para que pudesse dedicar-me exclusivamente a este trabalho.

A todos que contribuíram de forma positiva ou negativa. Que Deus lhes abençoe sempre!

Resumo

Sistemas Multi-Agentes (SMA) são sistemas altamente distribuídos, ou seja, que não possuem um único ponto de controle, e que utilizam uma coleção de agentes. Um agente pode ser visto como um sistema computacional encapsulado que está situado em algum ambiente e é capaz de realizar ações autônomas e ter um comportamento flexível neste ambiente, com o intuito de atingir os objetivos para os quais foi projetado. Agentes de um SMA têm objetivos particulares, interagem com um mundo dinâmico e se inter-relacionam, através de uma estrutura organizacional, para resolver tarefas e atingir seus objetivos.

Com o aumento da complexidade desses sistemas, 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. Neste sentido, a metodologia Tropos apresenta uma abordagem centrada em requisitos que objetiva dar suporte a todas as fases do desenvolvimento de software orientado a agentes. Tropos é fundamentada nos conceitos sociais e intencionais do *framework* de modelagem organizacional i* e possui quatro fases de desenvolvimento: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural e Projeto Detalhado.

O objetivo principal desta dissertação é apoiar o desenvolvimento, usando a plataforma JADE, de SMA especificados no contexto da metodologia Tropos. Foram realizadas complementações na fase de projeto detalhado desta metodologia, para capturar aspectos dos agentes que precisam ser levados em consideração quando desejamos implementar um SMA em JADE.

Nossa proposta é validada apresentando como estudo de caso o desenvolvimento de um sistema de gerenciamento de conteúdo (CMS) que utiliza agentes para simular o processo de publicação de notícias em um site da *internet*.

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

Abstract

Multi-Agent Systems (MAS) are large distributed systems that use a collection of agents. An agent can be viewed as a computational encapsulated system situated in an environment, capable of performing autonomous actions and to presenting flexible behaviors, aiming to reach the goals for which it was designed. These agents have particular or general goals, interact with a dynamic world and have some inter-relationships, through an organizational structure to make task resolution and goal reaching possible.

With the demand for robustness, reliability and reusability of increasingly complex systems, the use of specific notations, tools and methodologies for agent oriented software development becomes crucial. In this sense, the Tropos methodology presents a requirements-driven approach that encloses the main development phases of the agent oriented software development. Tropos is founded on the social and intentional concepts offered by the i* organizational modeling framework and has four development phases: Early Requirements, Late Requirements, Architectural Design and Detailed Design.

The main goal of this dissertation is to assist the development of Multi-Agent Systems (MAS) modeled in the context of the Tropos methodology and implemented using the JADE agent development framework. There was made some complementation in the Tropos detailed design phase to capture agent characteristics that have to be considered when we aim to implement a MAS in JADE.

Our approach was validated using as case study the development of a Content Management System (CMS) that uses agents to simulate the publication process of news at internet.

Keywords: Agent Oriented Software Engineering, Multi-Agent Systems, Software Development Methodologies, Tropos, JADE

Conteúdo

1	Introdução	1
1.1	Motivação.....	1
1.2	Objetivos e Abordagem	2
1.3	Estrutura da Dissertação	3
2	Engenharia de Software Orientada a Agentes	5
2.1	Introdução	5
2.2	Sistemas Multi-Agentes	5
2.2.1	<i>Tipos de Sistemas Baseados em Agentes</i>	<i>6</i>
2.2.2	<i>Sistemas Multi-Agentes na Engenharia de Software.....</i>	<i>7</i>
2.2.3	<i>Desafios e Obstáculos do Desenvolvimento Orientado a Agentes.....</i>	<i>8</i>
2.3	Metodologias de Desenvolvimento Orientado a Agentes	11
2.4	Implementação de Agentes de Software	13
2.4.1	<i>Visão Geral das Especificações da FIPA</i>	<i>13</i>
2.4.2	<i>Visão Geral das Plataformas de Desenvolvimento</i>	<i>16</i>
2.5	Considerações Finais.....	17
3	Desenvolvendo SMA com Tropos	18
3.1	Introdução	18
3.2	O Framework i*.....	19
3.2.1	<i>O modelo de Dependência Estratégica</i>	<i>19</i>
3.2.2	<i>O modelo de Razão Estratégica</i>	<i>22</i>
3.3	A Metodologia Tropos	22
3.3.1	<i>Requisitos Iniciais.....</i>	<i>24</i>
3.3.2	<i>Requisitos Finais.....</i>	<i>24</i>
3.3.3	<i>Projeto Arquitetural.....</i>	<i>25</i>
3.3.4	<i>Projeto Detalhado.....</i>	<i>29</i>
3.3.5	<i>Implementação.....</i>	<i>30</i>
3.4	Considerações Finais.....	31
4	Desenvolvendo SMA com JADE	33
4.1	Introdução	33
4.2	A plataforma JADE.....	34
4.2.1	<i>Principais Características de JADE</i>	<i>35</i>
4.2.2	<i>Agentes em JADE.....</i>	<i>36</i>
4.2.3	<i>Comportamentos de Agentes.....</i>	<i>40</i>
4.2.4	<i>Mensagens em JADE</i>	<i>43</i>
4.2.5	<i>Protocolos de interação</i>	<i>46</i>
4.2.6	<i>Ontologias em JADE</i>	<i>49</i>
4.2.7	<i>Ferramentas de Gerenciamento da Plataforma JADE</i>	<i>54</i>
4.3	Considerações Finais.....	57
5	Do Projeto Detalhado em Tropos à Implementação com JADE	58
5.1	Introdução	58
5.2	O Projeto Detalhado para JADE	59
5.2.1	<i>Projeto da dimensão estrutural.....</i>	<i>61</i>
5.2.2	<i>Projeto da dimensão comunicativa</i>	<i>62</i>
5.2.3	<i>Projeto da dimensão dinâmica</i>	<i>64</i>
5.3	A implementação em JADE a partir de Tropos	65
5.4	Considerações Finais.....	66
6	Estudo de Caso.....	67

6.1	Introdução	67
6.2	O Smart Journal	68
6.3	Modelagem com Tropos	68
6.3.1	<i>Requisitos Iniciais</i>	68
6.3.2	<i>Requisitos Finais</i>	73
6.3.3	<i>Projeto Arquitetural</i>	75
6.3.4	<i>Projeto Detalhado</i>	79
6.4	Implementação em JADE	84
6.4.1	<i>Estrutura geral do sistema</i>	88
6.5	Considerações Finais.....	90
7	Conclusões e Trabalhos Futuros.....	92
7.1	Principais Contribuições	92
7.2	Trabalhos Relacionados	93
7.3	Trabalhos Futuros	94
7.4	Considerações Finais.....	95
8	Referências Bibliográficas.....	97
	Apêndice A	102
	Apêndice B	103

Lista de Figuras

Figura 2.1: Serviços providos por uma plataforma compatível com a FIPA.....	14
Figura 2.2: Arquitetura de referência da FIPA para uma plataforma de agentes.....	15
Figura 2.3: Ciclo de vida das especificações FIPA	15
Figura 3.1: Elementos de modelagem com i*.....	20
Figura 3.2: Tipos de relacionamento de dependência entre atores no i*	20
Figura 3.3: Associações e representação de agentes, papéis e posições.....	21
Figura 3.4: (a) Ligações de decomposição de tarefas (b) Ligações de meios-fins.....	22
Figura 3.5: Estilo organizacional arquitetural União Estratégica (Joint Venture)	26
Figura 3.6: Visão geral do mapeamento i*/BDI/JACK [Castro02a].....	30
Figura 4.1: Plataforma de Agentes JADE distribuída por vários <i>containers</i>	35
Figura 4.2: Arquitetura interna de um agente genérico em JADE.....	37
Figura 4.3: Ciclo de vida de um agente definido pela FIPA	39
Figura 4.4: Hierarquia das classes de comportamentos em JADE derivadas de <i>Behaviour</i>	41
Figura 4.5: Componentes do modelo de comunicação da FIPA.....	44
Figura 4.6: Exemplo de mensagem de acordo com o padrão ACL	45
Figura 4.7: Estrutura homogênea dos protocolos de interação em JADE.....	47
Figura 4.8: Protocolo de interação FIPA-Contract-Net	49
Figura 4.9: Conversão realizada pelo suporte de JADE para linguagens de conteúdo e ontologias	51
Figura 4.10: O modelo de referência de conteúdo de JADE	51
Figura 4.11: Tela da ferramenta Remote Monitoring Agent	54
Figura 4.12: Tela da ferramenta <i>Sniffer Agent</i>	55
Figura 4.13: Tela da ferramenta <i>Introspector Agent</i>	56
Figura 4.14: Tela da ferramenta <i>Dummy Agent</i>	57
Figura 6.1: Modelo i* de Dependência Estratégica dos Requisitos Iniciais.....	69
Figura 6.2: Modelo i* de Razão Estratégica de uma redação de jornal para o ator <i>editor</i>	70
Figura 6.3: Modelo i* de Razão Estratégica de uma redação de jornal para o ator <i>webmaster</i>	71
Figura 6.4: Modelo i* de Razão Estratégica dos Requisitos Iniciais para o ator repórter.....	72
Figura 6.5: Modelo i* de Razão Estratégica de uma redação de jornal para o ator chefe-redação	72
Figura 6.6: Modelo i* de Dependência Estratégica dos Requisitos Finais	74
Figura 6.7: Modelo i* de Razão Estratégica dos Requisitos Finais	75
Figura 6.8: Grafo de interdependências de metas-soft do <i>Smart Journal</i>	76
Figura 6.9: Arquitetura do <i>Smart Journal</i> baseada no estilo União Estratégica.....	78
Figura 6.10: Diagrama de relacionamentos para os agentes da aplicação.....	80
Figura 6.11: Visão parcial do diagrama de ontologias para classes de conceitos.....	81
Figura 6.12: Visão parcial do diagrama de ontologias para classes de ações de agentes	82
Figura 6.13: Visão parcial do diagrama de ontologias para classes de predicados	82
Figura 6.14: Diagrama de interação para o comportamento PublishNews	83
Figura 6.15: Diagrama de interação para o comportamento CheckAlreadyPublish	83
Figura 6.16: Visão arquitetural em camadas do <i>Smart Journal</i>	88
Figura 6.17: Tela principal do sistema <i>Smart Journal</i>	89
Figura 8.1: Modelo Entidade-Relacionamento do banco de dados definido no <i>Smart Journal</i>	102
Figura 8.2: Diagrama de ontologias simplificado para classe NewsOntology.....	107

Lista de Tabelas

Tabela 3.1: Estilos arquiteturais organizacionais definidos em Tropos	25
Tabela 3.2: Catálogo de correlação entre estilos arquiteturais e atributos de qualidade	27
Tabela 5.1: Dimensões de modelagem de padrões de projeto em Tropos	60
Tabela 5.2: Estereótipos sugeridos para o projeto detalhado para JADE	61
Tabela 5.3: <i>Template</i> para definição de mensagens ACL	63
Tabela 5.4: Template para definição do modelo de capacidades	65
Tabela 6.1: Estrutura da classe <i>smagnw.agent.WebmasterAgent</i>	84
Tabela 6.2: Estrutura da classe <i>smagnw.behaviours.HandlePublishBehaviour</i>	86
Tabela 6.3: Representação de uma notícia de acordo com o padrão NITF	90
Tabela 8.1: Código em Java da classe <i>smagnw.ontology.NewsOntology</i>	103

Lista de Abreviaturas

ACC	Agent Communication Chanel
ACL	Agent Communication Language
AID	Agent Identifier
AMS	Agent Management Service
API	Application Programming Interface
AUML	Agent Unified Modeling Language
CASE	Computer Aided Software Engineering
CMS	Content Management System
DF	Directory Facilitator
FIPA	Foundation for Intelligent Physical Agents
GOOD	Goals into Object Oriented Development
GOOSE	Goal Into Object Oriented Standard Extension
GUI	Graphical User Interface
JADE	Java Agent DEvelopment Framework
JVM	Java Virtual Machine
LGPL	Lesser GNU Public License
MTP	Message Transport Protocol
OME	Organizational Modeling Environment
RMA	Remote Monitoring Agent
RMI	Remote Method Invocation
SIG	Softgoal Interdependency Graph
SMA	Sistema Multi-Agentes
UML	Unified Modeling Language
XGOOD	eXtended Goal Into Object Oriented Development
XML	Extensible Markup Language
WWW	World Wide Web

Capítulo

1 Introdução

Este capítulo introdutório apresenta as principais motivações para realização desta dissertação. As seções seguintes descrevem os objetivos deste trabalho e abordagem utilizada para desenvolvê-lo, bem como suas principais contribuições. Por fim é apresentada a estrutura na qual está organizada esta dissertação.

1.1 Motivação

O crescente grau de sofisticação dos sistemas de computação traz para engenharia de software o desafio de lidar com o aumento da dificuldade em analisar, projetar e implementar esses sistemas. Isso tem motivado os desenvolvedores de software a compreenderem melhor as características dos softwares complexos e a proporem novos paradigmas de desenvolvimento, tais como a Orientação a Aspectos [Kiczales97] e a Orientação a Agentes [Jennings00].

Atualmente, a complexidade de muitos sistemas de software está relacionada à sua necessidade de operar em ambientes distribuídos, abertos, dinâmicos, não determinísticos, heterogêneos e com alta interatividade [Weiss00]. Além disso, é considerável o aumento na necessidade desses sistemas operarem em ambientes com mudanças organizacionais freqüentes. Infelizmente, as metodologias de software existentes (orientadas a objetos, estruturadas, etc.) foram tradicionalmente inspiradas em conceitos de programação, ao invés de conceitos organizacionais, resultando em um distanciamento semântico entre o sistema e seu ambiente operacional. Dessa forma, a orientação a agentes pode servir como um paradigma poderoso e útil para a engenharia de software porque suas características provêem maneiras de lidar com essas características, bem como oferece um alto nível de abstração com relação às peculiaridades e comportamento de sistemas de software complexos.

A tecnologia de agentes tem sido aplicada em vários domínios, incluindo comércio eletrônico, interface homem-máquina, telecomunicações e engenharia concorrente [Luck03]. Em geral, agentes de software são vistos como objetos complexos com uma atitude. Assim como objetos, agentes provêem um conjunto específico de serviços para seus usuários. De fato, agentes e objetos exibem pontos de similaridade, mas o desenvolvimento de Sistemas Multi-Agentes (SMA) impõe desafios à engenharia de software uma vez que agentes de software são inerentemente abstrações mais complexas do que objetos.

Um agente de software é dirigido por objetivos, conhecimento e um número de propriedades comportamentais tais como autonomia, adaptação, interação, colaboração, aprendizado e mobilidade [Weiss02, Russel02]. Enquanto estas características introduzem complexidade adicional às diferentes fases do processo de desenvolvimento de software, existem muitas técnicas para lidar com agentes

individuais ou sistemas construídos usando apenas poucos agentes [Garcia03]. Infelizmente, as abordagens existentes para engenharia de software ainda estão inaptas para dar suporte a complexidade de SMAs, uma vez que a engenharia de software para o paradigma de orientação a agentes ainda está em sua infância.

As características de SMAs estão começando a ser aplicadas ao desenvolvimento de grandes sistemas de software industriais. Tais sistemas envolvem centenas, talvez milhares de agentes, e existe uma necessidade emergente por técnicas de engenharia de software que permitam o gerenciamento efetivo desta complexidade inerente e por métodos para guiar o processo de desenvolvimento de SMAs. Sem técnicas e métodos adequados, tais sistemas não serão suficientemente confiáveis, extensíveis, compreensíveis, e seus componentes não serão reusáveis.

Assim, o sucesso comercial das aplicações de SMAs requer soluções escaláveis baseadas em abordagens da engenharia de software com o intuito de permitir reuso e entrega efetivos. Entretanto, muitas soluções orientadas a agente estão longe do ideal; na prática, os sistemas são freqüentemente construídos de maneira *ad hoc* e suscetíveis a erro, não escaláveis, não dinâmicos, e não aplicáveis genericamente para o desenvolvimento industrial de SMAs.

Nos últimos anos, por conta deste crescente interesse no potencial da tecnologia de agentes no contexto da engenharia de software, foram propostos diversos ambientes de desenvolvimento para criação de agentes, por exemplo, Zeus [Collis99] e AgentBuilder [Reticular99], e *frameworks* para desenvolvimento de aplicações de agentes de acordo com os padrões da FIPA [FIPA04], tais como FIPA-OS [FIPA-OS05] e JADE [Bellifemine05]. Estes ambientes e *frameworks* de desenvolvimento de software demandam que as metodologias de análise e projeto de sistemas, linguagens e procedimentos possam dar suporte adequado a eles. Em consequência disto, também têm sido propostas algumas metodologias para desenvolvimento de software orientado a agentes promissoras, tais como Tropos [Castro01a], Gaia [Wooldridge00, Zambonelli03], e AUML [Odell01].

1.2 Objetivos e Abordagem

Este trabalho se propõe a investigar a questão do desenvolvimento de sistemas para o paradigma de orientação a agentes. O objetivo principal desta dissertação é apoiar a implementação, usando a plataforma JADE [Bellifemine05], de sistemas multi-agentes especificados no contexto da proposta de Tropos [Bresciani04].

Esta pesquisa foi motivada principalmente pelos seguintes fatores:

- Ausência de um estudo de caso no contexto do projeto Tropos implementado utilizando a plataforma JADE;
- Necessidade de evolução da Engenharia de Software Orientada a Agentes, no que diz respeito às notações, ferramentas e metodologias de desenvolvimento;
- Necessidade de aprimorar a fase de projeto detalhado para o desenvolvimento de sistemas multi-agentes, principalmente para a metodologia Tropos;
- Capacidade de extensão da UML para dar suporte ao projeto detalhado de agentes, através do provimento de um conjunto rico e extensível de elementos de construção que podem ser adaptados para o paradigma de orientação a agentes.

JADE (*Java Agent DEvelopment framework*) é um *framework* que propõe uma boa infra-estrutura de suporte e desenvolvimento de agentes, tornando sua implementação mais simples e robusta. Nesta dissertação foram propostas complementações na fase de projeto detalhado da metodologia Tropos, para capturar aspectos dos agentes que precisam ser levados em consideração quando desejamos implementar um sistema multi-agentes em JADE.

Melhorias no apoio ferramental, propostas relacionadas a padrões de projeto para agentes e características de implementação com a linguagem de programação Java não farão parte deste trabalho de pesquisa uma vez que esse foco é dado em diversos outros trabalhos disponíveis na literatura. Além disso, a questão da mobilidade dos agentes não será tratada neste trabalho, contudo, JADE também dá suporte a aplicações com agentes móveis.

Nossa proposta é validada apresentando como estudo de caso o desenvolvimento de um sistema de gerenciamento de conteúdo (CMS) que utiliza agentes para simular o processo de publicação de notícias na internet.

1.3 Estrutura da Dissertação

Além deste capítulo introdutório, que apresentou motivações, objetivos e contribuições de nosso trabalho, essa dissertação consiste de mais seis capítulos, são eles:

Capítulo 2 - Engenharia de Software Orientada a Agentes

Neste capítulo são mostradas as principais motivações para usar a abordagem de agentes, sendo apresentados algumas definições e conceitos relacionados com a Engenharia de Software Orientada a Agentes, bem como os obstáculos que esta nova área vem sofrendo.

Capítulo 3 - Desenvolvendo SMA com Tropos

Este capítulo aborda o desenvolvimento de Sistemas Multi-Agentes utilizando a proposta Tropos. Os conceitos básicos do *framework* para modelagem organizacional i* são apresentados e em seguida é realizada uma explanação de todas as fases da metodologia Tropos.

Capítulo 4 - Desenvolvendo SMA com JADE

Este capítulo fornece uma visão geral da plataforma de desenvolvimento de agentes JADE. São apresentados: sua estrutura básica, principais conceitos e ferramentas de gerenciamento.

Capítulo 5 - Do Projeto Detalhado em Tropos à Implementação com JADE

Neste capítulo é apresentada a principal contribuição desta dissertação, a proposta de adaptação da fase de projeto detalhado da metodologia Tropos, que objetiva auxiliar o desenvolvimento de sistemas multi-agentes implementados utilizando a plataforma JADE.

Capítulo 6 - Estudo de Caso

Este capítulo apresenta um estudo de caso desenvolvido para validar nossa proposta. Neste contexto, será avaliada a proposta apresentada no capítulo 5 para o projeto detalhado em Tropos direcionado

para implementação com JADE. O estudo de caso desenvolvido consiste de um sistema de software orientado a agentes para publicação de notícias na internet.

Capítulo 7 - Conclusões e Trabalhos Futuros

Este capítulo mostra as conclusões obtidas no desenvolvimento deste trabalho, assim como as principais contribuições que ele fornece para Engenharia de Software Orientada a Agentes. São apresentados também alguns trabalhos relacionados, bem como possíveis trabalhos futuros que podem ser realizados a partir deste. Por fim, apresentamos as considerações finais desta dissertação.

Capítulo

2 Engenharia de Software Orientada a Agentes

Este capítulo aborda a Engenharia de Software Orientada a Agentes. São apresentadas as principais motivações para usar a abordagem de agentes, bem como sua importância para a Engenharia de Software. Algumas definições e conceitos relacionados com a Engenharia de Software Orientada a Agentes, bem como os obstáculos que esta nova área vem sofrendo, também são explicados neste capítulo.

2.1 Introdução

De acordo com [Weiss02], um agente pode ser visto como um sistema computacional encapsulado que está situado em algum ambiente e é capaz de realizar ações autônomas e ter um comportamento flexível neste ambiente com o intuito de atingir os objetivos para os quais foi projetado. Sistemas multi-agentes (SMA) são sistemas altamente distribuídos, ou seja, que não possuem um único ponto de controle, e que utilizam uma coleção de agentes. Esses agentes têm objetivos particulares, interagem com um mundo dinâmico e se inter-relacionam, através de uma estrutura organizacional, para resolver tarefas e atingir seus objetivos [Jennings00]. Agentes são úteis porque provêem uma maneira de pensar sobre o fluxo de controle em um sistema altamente distribuído, oferecem um mecanismo que permitem um comportamento emergente ao invés de uma arquitetura estática, e codificam práticas melhores de como organizar entidades colaborativas concorrentes [Garcia03, Lucena04].

Um sistema orientado a agentes é aquele no qual a principal abstração usada para a modelagem da aplicação é a de agente. Com o aumento da complexidade desses sistemas, torna-se importante o uso de notações, ferramentas e metodologias específicas para o desenvolvimento de software orientado a agentes, visando a construção de sistemas mais robustos, confiáveis e reutilizáveis [Bergenti00, Wooldridge02]. A chamada a Engenharia de Software Orientada a Agentes (ESOA) é uma nova área de pesquisa que se preocupa com metodologias e técnicas para produção com qualidade de softwares orientados a agentes.

2.2 Sistemas Multi-Agentes

Não há uma definição padrão para agente. Russel e Norvig [Russel02] definem agente como sendo uma entidade que percebe seu ambiente através de sensores e atua no ambiente através de atuadores. Nesta dissertação direcionamos a conceituação do que seriam agentes de software para a seguinte definição [Wooldridge02]:

“Um agente é um sistema computacional encapsulado que está situado em algum ambiente e é capaz de ação flexível autônoma neste ambiente, a fim de alcançar os seus objetivos de projeto”.

Agentes possuem as seguintes propriedades, entre outras:

- Autonomia: habilidade do software de agir independentemente sem intervenção direta humana ou de outros agentes [Yu01].
- 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 [Yu01].

A seguir apresentamos os tipos de sistemas baseados em agentes encontrados hoje na ESOA.

2.2.1 Tipos de Sistemas Baseados em Agentes

Um sistema baseado em agentes pode conter um ou mais agentes. Existem casos em que um único agente é apropriado. Um bom exemplo é a classe de sistemas conhecida como assistentes especialistas, na qual um agente age como um especialista que auxilia um usuário de computador na execução de alguma tarefa. Entretanto, quando adotamos uma visão orientada a agentes do mundo, torna-se claro que um único agente é insuficiente [Jennings00]. Do ponto de vista da Engenharia de Software, a modelagem de sistemas projetados e implementados com vários agentes interagindo entre si é mais geral e mais interessante do que a modelagem de sistemas com um único agente.

Um Sistema Multi-Agentes (SMA) 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-agentes 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.

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. Desta forma, os sistemas multi-agentes são apropriados para representar problemas que têm vários métodos de resolução, múltiplas perspectivas e / ou múltiplas entidades de resolução. Tais sistemas têm as vantagens tradicionais da resolução de problemas concorrentes e distribuídos, mas incluem as vantagens dos padrões de interações sofisticados. Por exemplo, agentes podem se organizar para realizar tarefas não previstas em tempo de projeto.

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.

Na próxima seção destacaremos a importância dos sistemas multi-agentes para ESOA.

2.2.2 Sistemas Multi-Agentes na Engenharia de Software

Nos dias de hoje, sistemas multi-agentes são vistos como um novo direcionamento importante na engenharia de software. Cada vez mais novos avanços têm sido alcançados para o paradigma de ESOA. Existem várias razões que justificam o aumento desse interesse na pesquisa de sistemas multi-agentes:

- Distribuição de dado e controle: para muitos sistemas de software o controle global é distribuído em vários nós de computação frequentemente distribuídos geograficamente. A fim de fazer com que esses sistemas trabalhem efetivamente, esses nós devem ser capazes de interagir autonomamente uns com os outros;
- Sistemas abertos: muitos sistemas são abertos no sentido de que não é possível saber, em tempo de projeto, quais componentes farão parte exatamente do sistema, nem como esses componentes interagem entre si. Para operarem efetivamente, estes sistemas devem ser capazes de tomar decisões autônomas flexíveis;
- Dividir funcionalidade entre muitos agentes permite que a aplicação seja modular, flexível, modificável e extensível;
- Conhecimento especializado não é frequentemente disponível em um único agente (onisciência). O conhecimento que é distribuído em várias fontes (agentes) pode ser integrado para uma maior completude quando for necessária flexibilidade.

Devido a todos esses fatores, observa-se que sistemas orientados a agentes estão sendo cada vez mais usados na indústria como, por exemplo, em aplicações voltadas para telecomunicações e comércio eletrônico. A tecnologia de agentes está ultrapassando rapidamente os limites das universidades e os laboratórios de pesquisa e está começando a ser usada para resolver problemas do mundo-real em uma escala de aplicações industriais e comerciais. Aplicações em uso existem hoje e novos sistemas estão sendo desenvolvidos cada vez mais.

Algumas áreas onde as abordagens baseadas em agente estão sendo usadas para construção de sistemas são [Lucena04,Luck03]:

- Aplicações industriais: agentes estão sendo aplicados em uma diversidade de sistemas industriais, entre eles:
 - Fabricação: 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;
 - Telecomunicações: controle de rede, transmissão e chaveamento, gerência de serviço e gerência de rede;

- Controle de Tráfego Aéreo: agentes são usados para representar tanto os equipamentos de aviação quanto vários sistemas de controle de tráfego aéreo em operação;
- Sistemas de transporte: o domínio de gerência de tráfego e transporte é adequado para uma abordagem baseada em agentes devido a sua natureza geograficamente distribuída.
- Aplicações Comerciais: enquanto aplicações industriais tendem a serem altamente complexas, indicando sistemas que operam em áreas de nicho comparativamente pequeno, as aplicações comerciais, especialmente aquelas preocupadas com gerência da informação, tendem a serem utilizadas por uma fatia maior do mercado. Dentre aplicações comerciais podemos citar:
 - Gerência de informação: Mecanismos de busca são realizados por agentes, que agem autonomamente para buscar na web em benefício de algum usuário. De fato esta é provavelmente uma das áreas mais ativas para aplicações de agente. Como exemplos temos: um assistente pessoal que aprende sobre os interesses do usuário e com base neles compila um jornal pessoal, um agente assistente para automatizar várias tarefas de usuário em um desktop de computador;
 - Comércio eletrônico: Algumas tomadas de decisão comerciais podem ser colocadas nas mãos de agentes. Um aumento da quantidade de comércio está sendo empreendido por agentes. Entre estas aplicações comerciais encontra-se um agente que descobre os Compact Discs mais baratos, um assistente pessoal de compras capaz de buscar lojas on-line para disponibilizar o produto e informação sobre o preço, um mercado virtual para comércio eletrônico e vários catálogos interativos baseados em agentes;
 - Gerência de processo de negócio: Organizações têm procurado desenvolver sistemas de Tecnologia da Informação para dar assistência a vários aspectos da gerência de processos de negócio. Outras aplicações nesta área incluem um sistema de gerência de cadeias fornecedoras, um sistema para gerência de fluxos de trabalho heterogêneos e um sistema baseado em agentes móveis para gerência de fluxo de trabalho interorganizacional.
- Aplicações de entretenimento: agentes têm um papel óbvio nos jogos de computador, cinema interativo, e aplicações de realidade virtual: tais sistemas tendem a ser cheios de caracteres animados autônomos, que podem naturalmente ser implementados como agentes.

Diante do uso crescente dessa nova tecnologia, alguns obstáculos e desafios foram encontrados no desenvolvimento de software orientado a agentes, os quais serão apresentados a seguir.

2.2.3 Desafios e Obstáculos do Desenvolvimento Orientado a Agentes

Uma vez que destacamos os benefícios potenciais da engenharia de software orientada a agentes, apresentamos nesta seção desafios inerentes à construção de software usando esta nova tecnologia. Estes podem ser tanto específicos da tecnologia de agentes, como armadilhas que surgem no processo de desenvolvimento de sistemas multi-agentes (ou aquelas comuns a qualquer nova tecnologia).

O fato de agentes terem de agir para satisfazer os seus objetivos enquanto mantêm uma interação contínua com seu ambiente torna difícil projetar software capaz de manter um equilíbrio entre comportamentos pró-ativos e reativos dos agentes. Para atingir este equilíbrio, é necessário que a tomada de decisão seja sensível ao contexto, resultando num grau significativo de imprevisibilidade

sobre quais objetivos o agente perseguirá, em quais circunstâncias e quais os métodos que serão usados para atingir os objetivos escolhidos.

Como agentes são entidades autônomas, os padrões e os efeitos de suas interações são imprevisíveis. Em resumo, tanto a natureza (uma requisição simples ou uma negociação prolongada) como a consequência de uma interação pode não estar determinada desde o início do projeto do sistema multi-agentes. Outro aspecto da imprevisibilidade no projeto de sistemas orientados a agentes refere-se à noção de comportamento emergente. Foi reconhecido que a interação de muitos agentes individuais resulta no fenômeno comportamental que não pode ser entendido de forma geral exclusivamente em termos do comportamento dos agentes individuais.

Além desses problemas específicos da tecnologia de agentes, identificamos vários desafios que ocorrem repetidamente no desenvolvimento de sistemas multi-agentes, entre outros que são comuns a qualquer nova tecnologia [Wooldridge02]. Esses desafios são classificados geralmente em cinco categorias [Tveit01]: Política, Conceitual, Análise e Projeto, Nível de Agente e Nível de Sociedade. A seguir os desafios dessas categorias são descritos em maiores detalhes:

- **Política:**
 - *Apresentar com entusiasmo excessivo as soluções de agente ou falhar em entender onde agentes podem ser aplicados adequadamente:* a tecnologia de agentes representa uma inovação e uma importante maneira de conceituar e implementar software, mas é importante entender suas limitações;
 - *Não tomar agentes como sendo uma religião ou dogma:* embora haja muitas aplicações usando agentes, eles não são uma solução universal. Há muitas aplicações para as quais os paradigmas convencionais de desenvolvimento de software (tal como programação orientada a objetos) são mais apropriados.
- **Conceitual:**
 - *Esquecer que está desenvolvendo software:* a ausência de técnicas testadas e confiáveis para dar assistência no desenvolvimento de software multi-agentes encoraja os projetistas a esquecerem que estão na verdade desenvolvendo software. Nesse caso, a falha do projeto não é por causa dos problemas específicos de agente, mas porque a boa prática da engenharia de software foi ignorada;
 - *Esquecer que está desenvolvendo software com múltiplas linhas de controle (multithreaded):* por natureza, sistemas multi-agentes tendem a ter múltiplas linhas de controle (tanto num agente quanto numa sociedade de agentes). Assim, na construção de software multi-agentes, é vital não ignorar as lições aprendidas da comunidade de sistemas concorrentes e sistemas distribuídos, pois os problemas inerentes aos sistemas com múltiplas linhas de controle não desaparecem apenas porque você adotou uma abordagem baseada em agentes.
- **Análise e Projeto:**
 - *Não saber porque necessita de uma solução orientada a agentes:* pode não haver entendimento de como agentes podem ser usados para melhorar seus produtos nem como eles podem capacitá-los a gerar novas linhas de produto, etc.;
 - *Querer construir soluções genéricas para problemas exclusivos:* tipicamente se manifesta no planejamento de uma arquitetura que supostamente possibilita que uma classe completa de sistemas seja construída, quando o que realmente é solicitado é uma solução feita sob encomenda direcionada para um único problema;

- *Acreditar que usar agentes é a única solução (silver bullet)*: a tecnologia de agentes é um paradigma de software novo, emergente e ainda não testado na sua essência. Portanto, ela não pode ser considerada uma solução mágica. Há bons argumentos que indicam que a tecnologia de agentes levará a melhorias no desenvolvimento de software distribuído complexo, mas estes argumentos ainda não foram quantitativamente comprovados.
 - *Criar sua própria arquitetura de agentes*: quando se inicia um projeto multi-agentes, tende-se a imaginar que nenhuma arquitetura de agentes existente atende aos requisitos do seu problema. Portanto, há uma tentação para projetar uma arquitetura a partir do zero. Isto é freqüentemente um erro, pois deve-se analisar as várias arquiteturas descritas na literatura antes de construir uma própria.
- **Nível de Agente:**
 - *Os agentes usam excessiva Inteligência Artificial*: ao se focar demais em aspectos de “inteligência”, o framework de construção de um agente fica muito sobrecarregado com as técnicas (e.g., processamento de linguagem natural, planejadores, provadores de teorema, sistemas de manutenção de razão, etc.). Para ser útil sugere-se construir agentes com um mínimo de técnicas da Inteligência Artificial. De fato, alguns pesquisadores chegam a afirmar que, dependendo da aplicação, somente 1% de inteligência artificial é necessário [Jennings03].
 - **Nível de Sociedade:**
 - *O projeto não explora concorrência*: se há a necessidade de uma única linha de controle num sistema, então a utilidade de uma solução baseada em agentes deve ser seriamente questionada;
 - *Ver agentes em qualquer lugar*: quando se adota este ponto de vista, a tendência é utilizar agentes para tudo. Porém, a sobrecarga gerada para gerenciar agentes e a comunicação entre agentes pesará mais do que os benefícios de uma solução baseada em agentes;
 - *Ter pouquíssimos agentes*: alguns projetistas criam um sistema que completamente falha em explorar o poder oferecido pelo paradigma de agente e desenvolvem uma solução com um número muito pequeno de agentes fazendo todo o trabalho. Tais soluções tendem a não atender o teste de coerência da engenharia de software, que requer que um módulo de software tenha uma única função coerente.

A seguir expomos os obstáculos chave que devem ser superados para que a engenharia de software orientada a agentes se torne popular [Wooldridge01, Castro02b]:

- *Organização da relação entre agentes e outros paradigmas*: não está claro como o desenvolvimento de sistemas multi-agentes irá coexistir com outros paradigmas de software, tal como o desenvolvimento orientado a objetos;
- *Metodologias orientadas a agentes*: embora várias metodologias preliminares de análise e projeto orientados a agentes estejam sendo propostas, há comparativamente pouco consenso entre elas e nenhum acordo sobre os tipos de conceitos que a metodologia deveria suportar [Luck03, Silva04].
- *Engenharia para sistemas abertos*: precisa-se de um melhor entendimento de como fazer engenharia em sistemas abertos. Em tais sistemas, acredita-se que é essencial ser capaz de reagir a eventos imprevistos, explorar oportunidades onde estes eventos surgem, e conseguir dinamicamente acordos com componentes do sistema cuja presença não poderia ser prevista em tempo de projeto;

- *Engenharia de escalabilidade*: precisa-se de um melhor entendimento de como fazer engenharia de forma segura e previsível em sistemas que consistem de um grande número de agentes interagindo dinamicamente uns com os outros a fim de atingir suas metas;
- *Rastreamento de requisitos*: a inclusão dessa atividade no processo de desenvolvimento orientado a agentes proporcionará muitos benefícios, pois não apenas permitirá um melhor atendimento aos requisitos do cliente, como também apoiará a avaliação do impacto de mudanças solicitadas (mesmo antes das suas implementações no sistema). Como resultado, ter-se-á uma melhoria na qualidade do software em desenvolvimento em decorrência da melhoria do processo de desenvolvimento o que levará a redução de tempo e custo da manutenção do software orientado a agentes.

Nesta seção nossa intenção não foi sugerir que o desenvolvimento baseado em agentes é mais complexo e inclinado a erro do que as abordagens tradicionais de engenharia de software. Ao invés disso, nós reconhecemos que há certas armadilhas que são comuns às soluções de agente, assim como há certas armadilhas comuns às soluções baseadas em objetos.

2.3 Metodologias de Desenvolvimento Orientado a Agentes

Projetar e construir software industrial de alta qualidade é difícil. Diante disso, vários paradigmas de implementação foram propostos na literatura, tais como a programação estruturada, programação declarativa, programação orientada a objetos e baseada em componentes [Pressman00]. Esta evolução dos paradigmas visa tornar a engenharia de software mais adequada para gerenciar a crescente demanda por desenvolvimento de softwares com capacidades computacionais cada vez maiores.

As empresas de desenvolvimento de sistemas de grande porte vêm apresentando grandes problemas e desejam produzir software de qualidade. Na visão do usuário, software de qualidade é aquele que além de satisfazer as suas necessidades, é feito no custo e prazo acordados. Já na visão do projetista de software, um bom software é aquele que apresenta as seguintes características [Silva03]:

- *Flexibilidade*: satisfazer novos requisitos de negócio de forma fácil e rápida;
- *Adaptabilidade*: configurar uma aplicação para vários usuários, usando várias alternativas para oferecer os serviços da aplicação com o mínimo de impacto no seu núcleo;
- *Manutenibilidade*: alterar partes de uma aplicação, de modo que as outras partes sofram um impacto mínimo;
- *Reusabilidade*: construir aplicações únicas e dinâmicas rapidamente;
- *Adequação a código legado*: reusar a funcionalidade de sistemas legados em novas aplicações;
- *Interoperabilidade*: possibilitar que aplicações que executam em plataformas diferentes troquem informações;
- *Escalabilidade*: distribuir e configurar a execução da aplicação de modo a gerenciar grandes volumes de transação;
- *Robustez*: implementar soluções de software com menos falhas.

A construção de SMA engloba todos os problemas dos sistemas distribuídos e concorrentes tradicionais, assim como 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.

Segundo [Wooldridge02], uma solução baseada em 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:

- provêm uma maneira de pensar sobre o fluxo de controle em um sistema altamente distribuído;
- oferecem um mecanismo que permitem um comportamento emergente 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]. Dentre elas, destaca-se o Tropos [Castro02a], uma metodologia que está sendo desenvolvida, visando dar suporte a todas as fases do desenvolvimento de software orientado a agentes, e propõe uma abordagem de desenvolvimento orientado a agentes 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.

GAIA [Wooldridge00] representa a primeira tentativa de obter uma metodologia completa e genérica para especificar análise e projeto de SMA. Nesta metodologia, SMA são visualizados como sendo compostos 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. Ela permite que um analista vá sistematicamente do estabelecimento de requisitos até um projeto que seja suficientemente detalhado a ponto de ser implementado diretamente. 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 [Bresciani04], é baseada nos conceitos usados durante a análise de requisitos iniciais. Ela tem adotado uma 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

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.

2.4.1 Visão Geral das Especificações da FIPA

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 1995, 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 JADE [JADE05].

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.

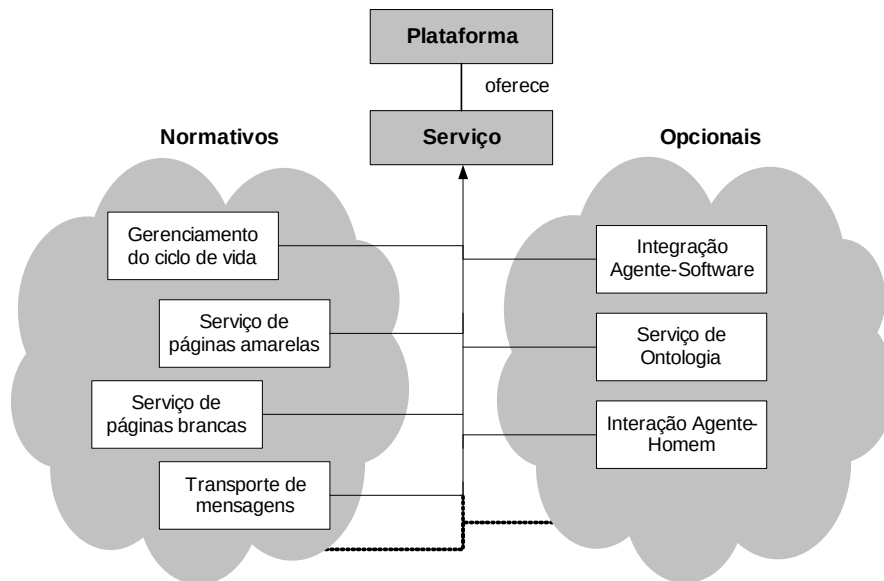


Figura 2.1: Serviços providos por uma plataforma compatível com a FIPA

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 a ele. 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 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 (por exemplo, dois agentes entenderem a mesma ontologia de maneira diferente).

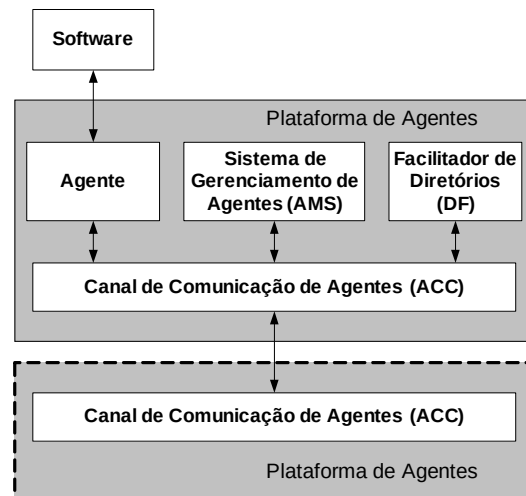


Figura 2.2: Arquitetura de referência da FIPA para uma plataforma de agentes

Além dos serviços que são especificados pela FIPA, ela ainda sugere uma arquitetura de referência para plataformas de agentes. Na Figura 2.2 podemos visualizar os seguintes elementos:

- O *Agent Management System* (AMS) é o agente responsável por supervisionar o acesso e o uso da plataforma multiagentes. Apenas um AMS deve existir por plataforma. O AMS oferece os serviços de páginas brancas e de ciclo de vida, gerenciando um repositório de identificadores de agentes (AID) e dos próprios agentes. Cada agente deve se registrar com o AMS para poder obter um AID válido.
- O *Directory Facilitator* (DF) é o agente que provê o serviço padrão de páginas amarelas na plataforma. A plataforma pode suportar outros DFs, além do padrão, e esses DFs podem se organizar entre si formando federações.
- O *Message Transport System*, também chamado de Agent Communication Channel (ACC), é o componente responsável por prover o serviço de transporte de mensagens. A especificação permite que haja mais de uma instância de tal serviço, desde que todo agente tenha acesso a pelo menos uma dessas instâncias.

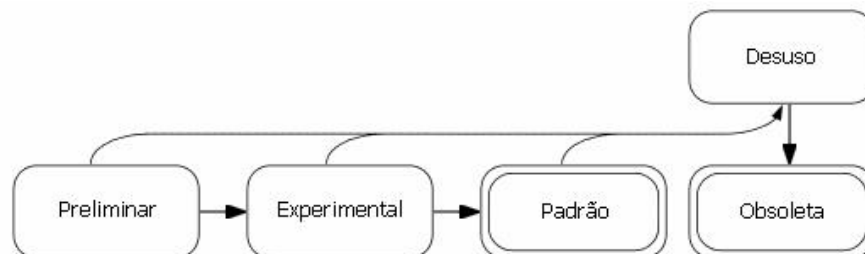


Figura 2.3: Ciclo de vida das especificações FIPA

A FIPA está preocupada com a produção e aprimoramento contínuo de especificações para SMAs. Essas especificações possuem por um ciclo de vida que passa pelas seguintes fases: preliminar,

experimental, padrão, em desuso e obsoleta (Figura 2.3). A maturidade da especificação se dá quando ela atinge o status de padrão.

A primeira especificação trabalhada pela FIPA foi a da linguagem FIPA-ACL (*Agent Communication Language*) [FIPA00], visando padronizar o protocolo de comunicação usado pelos agentes. Essa linguagem é baseada na teoria dos atos de fala e apresenta a grande vantagem de possuir uma definição formal de sua biblioteca de performativas. Ela não representa o conteúdo da mensagem, que deve vir encapsulado como um de seus atributos.

Para representação do conteúdo da mensagem, a FIPA criou a linguagem FIPA-SL (*Semantic Language*). Esta linguagem baseia-se em lógica de primeira ordem e visa permitir aos agentes expressar propriedades e relações entre objetos de um domínio específico. O domínio para o qual se refere o conteúdo é definido através de uma ontologia, que vem referenciada na linguagem externa (e.g., FIPA-ACL). A FIPA ainda especifica um conjunto de protocolos de interação definidos para padronizar as conversações entre os agentes. Todas as especificações da FIPA podem ser encontradas em [FIPA04].

2.4.2 Visão Geral das Plataformas de Desenvolvimento

Devido à inerente dificuldade de construção de SMAs, surgiram *frameworks* genéricos para simplificar a sua construção. Em geral, tais *frameworks* podem ser descritos como uma camada *middleware* sobre a qual operam os agentes. Esses *frameworks* são, em geral, construídos em conformidade com as especificações da FIPA [FIPA04], que é uma entidade responsável por padronizar os SMAs visando interoperabilidade entre eles.

Dentre esses ambientes de desenvolvimento de sistemas com agentes têm sido desenvolvidos para dar assistência à construção de SMA, podemos citar os 3 principais que estão de acordo com as especificações da FIPA e também são completamente implementados em Java [Deitel03]: JACK [Coburn01], FIPA-OS [FIPA-OS05] e JADE [Bellifemine04].

FIPA-OS é a primeira implementação de código livre 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.

JACK Intelligent Agents [JACK05] é um ambiente de desenvolvimento orientado a agentes construído inteiramente em Java que 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*) [Rao95]. 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 os eventos atribuídos a eles.

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. O capítulo 4 desta dissertação é totalmente dedicado à descrição do framework JADE.

Apesar desses *frameworks* serem desenvolvidos independentes de aplicação, os SMAs que utilizem recursos multimídias não encontram nessas ferramentas as condições ideais para o seu funcionamento. Isso se deve principalmente ao uso de Java na implementação de quase todos os *frameworks* atuais. Como se sabe, aplicações multimídias são aplicações que utilizam muitos recursos (memória e processamento) e Java não oferece o desempenho adequado para essa classe de sistemas.

2.5 Considerações Finais

A orientação a agentes surge como uma nova abordagem para construir sistemas complexos e está começando a ser usada em aplicações industriais e comerciais, deixando de existir apenas nas universidades e laboratórios de pesquisa. Mas, infelizmente, os benefícios prometidos pelo paradigma de agentes ainda não podem ser completamente obtidos porque não existe até o momento um consenso sobre a padronização de notações e processos de desenvolvimento para construção de software orientado a agentes. Atualmente, muitos deles são construídos de forma *ad-hoc* e outros utilizam metodologias ainda imaturas.

Atualmente, especificar, projetar e implementar SMAs é extremamente difícil. Esta atividade torna-se ainda mais crítica pela imaturidade da tecnologia de agente e pela ausência de notações, ferramentas, metodologias e processos de desenvolvimento específicos para o desenvolvimento de software orientado a agentes. Sem isto, torna-se impossível explorar os benefícios oferecidos por este novo nível de abstração.

Contudo, o crescente interesse em agentes de software e SMA tem conduzido ao desenvolvimento de novas tecnologias baseadas nos conceitos do paradigma de orientação a agentes. Metodologias e linguagens de modelagem, tais como GAIA [Wooldridge00, Zambonelli03], AUMML [Odell01] e Tropos [Castro02a, Bresciani04] estão sendo projetadas com o intuito de prover apoio metodológico e ferramental para o desenvolvimento de SMA. Essas metodologias propõem diferentes técnicas e métodos para utilizar os conceitos de agente nos diversos estágios do ciclo de desenvolvimento de software, ajudando a entender estruturas sociais complexas no domínio do problema da aplicação e traduzi-las em soluções tecnológicas.

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 a metodologia Tropos, juntamente com a explanação de cada uma de suas fases.

3.1 Introdução

Nos dias de hoje, a Engenharia de Software 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. A maioria desses 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. Contudo, muitos desses sistemas falham no suporte às organizações das quais eles são parte integrante. Isto acontece devido à ausência 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.

Neste contexto, 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.

Por sua vez, Tropos é uma metodologia de desenvolvimento de sistemas orientada a agentes inspirada na análise de requisitos iniciais e fundamentada em conceitos sociais e intencionais [Castro02a], que adota o *framework* de modelagem organizacional i* em suas fases iniciais de modelagem e utiliza seus conceitos durante todas as fases de desenvolvimento. 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.

A programação orientada a agentes, particularmente para Sistemas Multi-Agentes, propõe abstrações interessantes para lidar com sistemas pertencentes aos domínios de aplicação citados acima. Dentre as plataformas de implementação existentes, optamos por utilizar JADE [Bellifemine05] juntamente com a metodologia Tropos para o desenvolvimento de nosso trabalho.

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. Para realizar esta análise, os engenheiros de requisitos modelam os *stakeholders*¹ 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 *framework* i* 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 de um 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 meios-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

Para descrever as relações de dependências externas entre os atores da organização, utilizamos o modelo SD. Neste modelo, 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

¹ *Stakeholders* são pessoas ou organizações que serão afetadas pelo sistema e que possuem influência, direta ou indireta, sobre os requisitos do sistema [Kotonya98]: usuários finais, gerentes, engenheiros responsáveis pelo desenvolvimento e manutenção do sistema, clientes da organização que usarão o sistema para fornecer algum serviço, etc.

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.

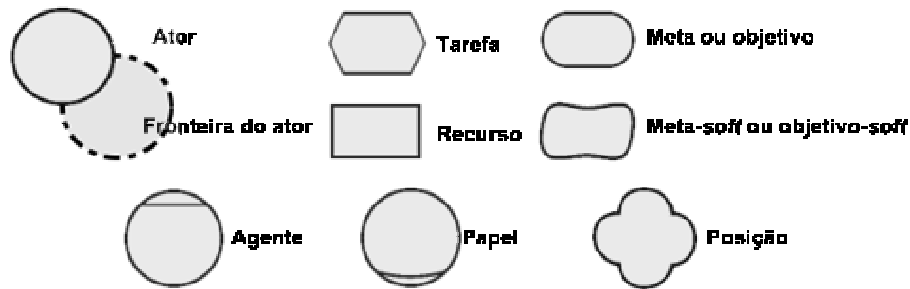


Figura 3.1: Elementos de modelagem com i*

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*.

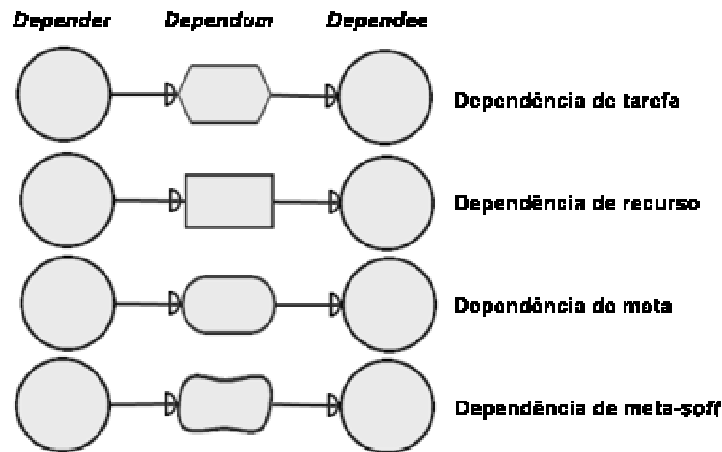


Figura 3.2: Tipos de relacionamento de dependência entre atores no i*

Uma **tarefa** especifica uma forma particular de se fazer algo. Tarefas podem ser vistas como soluções que provêm operações, processos, representações de dados, estruturação, restrições e agentes para atender às necessidades estabelecidas nas metas e metas-soft. 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 motivo 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: 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.

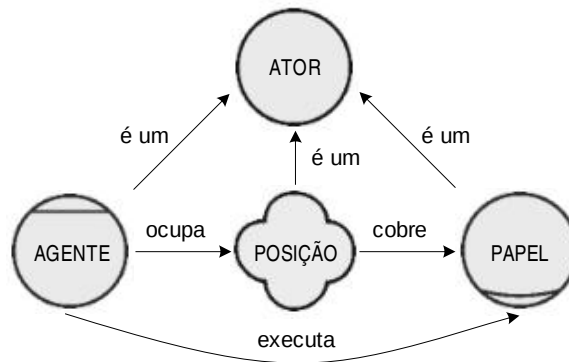


Figura 3.3: Associações e representação de agentes, papéis e posições

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 ambos meta ou ambos *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.

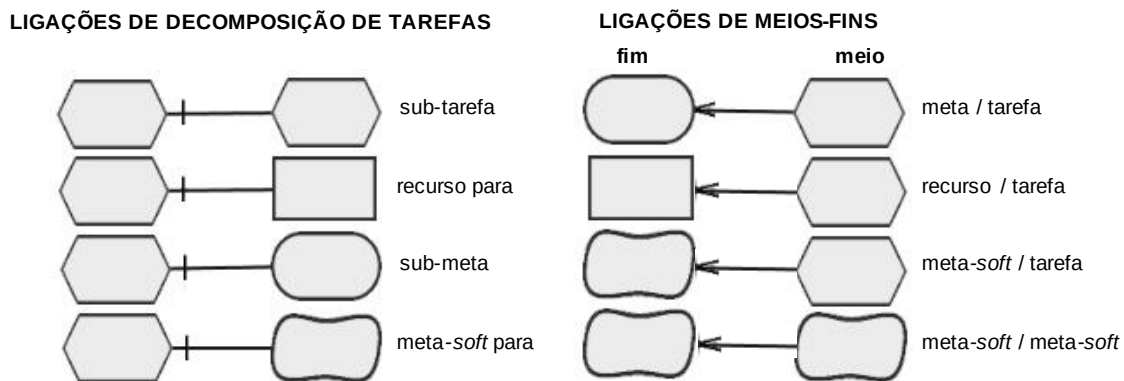


Figura 3.4: (a) Ligações de decomposição de tarefas (b) Ligações de meios-fins

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 meios-fins.

3.3 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 [Castro02a] [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 [Castro02a]. 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] foi construída para dar suporte à construção dos modelos dos *frameworks* *i** [Yu95] e NFR [Chung00] e vem sendo continuamente evoluída, de acordo com os avanços alcançados pela proposta Tropos. A partir de descrições organizacionais expressas através na notação do *framework i** é possível a geração de modelos orientados a objetos [Castro01b, Santander02].

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, Pedroza05] 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].

Além disso, Tropos possui mecanismos que permitem que os requisitos possam ser expressos em uma linguagem de modelagem formal denominada Formal Tropos [Fuxman01], a qual permite que a análise da especificação dos requisitos possa ser realizada formalmente. Esta linguagem oferece os conceitos primitivos de atores, metas, e dependências entre atores, e os complementa com uma linguagem de especificação temporal rica inspirada em KAOS [Dardenne93]. Formal Tropos também suporta uma metodologia para a análise e animação automatizada de especificações, fundamentada nas técnicas de checagem de modelo [Clarke99], através do protótipo de uma ferramenta baseada na ferramenta de checagem simbólica de modelo chamada NuSMV [Cimatti00].

As subseções a seguir apresentam as fases de desenvolvimento da metodologia Tropos.

3.3.1 Requisitos Iniciais

A análise de requisitos representa a etapa inicial do desenvolvimento de software em muitas metodologias. Na metodologia Tropos, a modelagem e análise dos requisitos é feita utilizando-se o *framework i** e está dividida em duas fases fundamentais: os Requisitos Iniciais e os Requisitos Finais.

A fase de Requisitos Iniciais está interessada na compreensão de um contexto organizacional no qual o sistema a ser desenvolvido será implantado, ou que servirá de modelo para este sistema. Os engenheiros de requisitos modelam os *stakeholders* como atores sociais que dependem de outros atores para atingir metas, realizar tarefas e fornecer recursos. A análise realizada nesta fase é orientada a metas. Cada meta é analisada do ponto de vista de seu ator, resultando em um conjunto de dependências entre pares de atores.

Através destas dependências, nós podemos realizar questionamentos do tipo “por que”, com relação à maneira com a qual os processos estão estruturados. Questionamentos deste tipo ligam as funcionalidades do sistema às necessidades, preferências e objetivos dos *stakeholders*. Os resultados desta fase são dois modelos:

- Um modelo SD, que captura os atores relevantes do ambiente organizacional, suas respectivas metas e suas interdependências;
- Um modelo SR, que determina através de uma análise meios-fins como as metas podem ser cumpridas através das contribuições de outros atores.

3.3.2 Requisitos Finais

A fase de Requisitos Finais está preocupada com o refinamento dos modelos SD e SR produzidos na primeira fase do Tropos [Castro01a]. Durante a análise dos requisitos finais, os modelos conceituais desenvolvidos na fase anterior são revisados e estendidos para incluir novos atores, que representam tanto o sistema a ser desenvolvido quanto os seus subsistemas.

Para dar origem ao modelo SD dos requisitos finais, esses atores devem ser relacionados, em termos de novos pares de dependência, aos atores do ambiente operacional do sistema que foram

previamente identificados. Todas as dependências em relação ao sistema refletem as responsabilidades atribuídas a ele, de forma a contribuir com as necessidades dos *stakeholders*. Essas dependências descrevem os requisitos funcionais e não funcionais do sistema a ser desenvolvido [Giorgini05].

Para construção do modelo SR desta fase, o ator que representa o sistema a ser desenvolvido deve ser expandido para apresentar as razões que estão por trás de suas dependências. Suas tarefas e metas precisam ser revisadas, analisadas e detalhadas através de ligações de decomposição e de meios-fins. Além disso, pode haver a atribuição de responsabilidades aos seus subsistemas. O objetivo desta etapa é dar suporte ao processo de sugestão, exploração e avaliação de soluções alternativas para o sistema.

3.3.3 Projeto Arquitetural

O Projeto Arquitetural define a arquitetura global do sistema em termos de subsistemas, interconectados através de fluxos de controle de dados. A arquitetura de um sistema pode ser considerada um modelo, relativamente pequeno e intelectualmente gerenciável da estrutura do sistema que descreve como seus componentes trabalham juntos. Em [Kolp02] foram definidos estilos arquiteturais organizacionais para auxiliar o projeto da arquitetura de aplicações cooperativas, dinâmicas e distribuídas, tais como SMA. A Tabela 3.1 apresenta os estilos arquiteturais organizacionais utilizados em Tropos.

Tabela 3.1: Estilos arquiteturais organizacionais definidos em Tropos

Estilo Arquitetural	Principais características
Estrutura Plana (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.
Estrutura em 5 (Structure in 5)	Consiste dos componentes típicos (estratégicos e logísticos) geralmente encontrados em várias organizações.
Pirâmide (Pyramid)	Estrutura de autoridade hierárquica exercida com limites organizacionais, onde atores nos níveis mais baixos dependem daqueles nos níveis mais altos.
União Estratégica (Joint Venture)	Envolve acordos entre dois ou mais parceiros principais, relacionados com um gerente comum e com parceiros secundários.
Oferta (Bidding)	Abrange mecanismos de competitividade e os atores se comportam como se estivessem tomando parte em um leilão.
Tomada de Controle (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> .
Comprimento de Braço (Arm's Length)	Implica em acordos entre atores independentes e competitivos, porém parceiros. Não há delegação de autoridade.
Contratação Hierárquica (Hierarchical Contracting)	Identifica mecanismos de coordenação que combinam características do acordo <i>Comprimento de Braço</i> com aspectos de autoridade <i>Pirâmide</i> .
Integração Vertical (Vertical Integration)	Consiste de atores comprometidos em atingir metas ou realizar tarefas relacionadas em estágios diferentes de um processo de produção.
Apropriação (Co-optation)	Envolve a incorporação de agentes externos na estrutura ou no comportamento tomador de decisão ou conselheiro de uma organização iniciante.

Uma vez que os conceitos fundamentais deste tipo de sistema são intencionais e sociais, ao invés de serem orientados à implementação, tais estilos arquiteturais são baseados em conceitos e alternativas de projeto vindas da pesquisa na Teoria Organizacional [Scott98] e Alianças Estratégicas [Gomes96] [Yoshino95], que analisam a estrutura e o projeto de organizações do mundo real. A Teoria Organizacional descreve a estrutura e o projeto de uma organização, enquanto que as Alianças Estratégicas modelam as colaborações estratégicas de *stakeholders* organizacionais independentes, mas que possuem acordos para dedicar-se a um conjunto de regras de negócios [Morabito99].

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 3.5, 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 como 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.

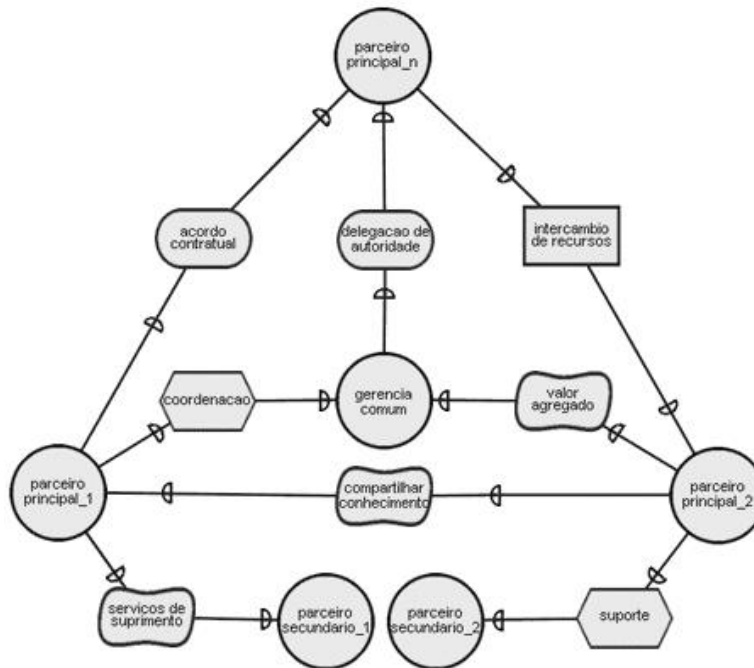


Figura 3.5: Estilo organizacional arquitetural União Estratégica (Joint Venture)

Atualmente, Tropos utiliza a notação do *framework* i^* [Yu05] para representar os modelos arquiteturais [Kolp02]. Nesta notação, subsistemas são representados como atores e interconexões de dado/controle são representados como dependências de atores. Em [Silva03] é apresentado um

conjunto de extensões baseadas na UML-RT como alternativa para representar os estilos arquiteturais definidos pelo Tropos. Contudo, neste trabalho optamos por utilizar a notação do *framework* i*.

Três atividades principais compõem a fase de projeto arquitetural da metodologia Tropos: a primeira consiste na seleção do estilo arquitetural, a segunda, no refinamento dos modelos SD e SR para refletirem a arquitetura escolhida; e a última é a aplicação dos padrões sociais [Bastos04]. As subseções seguintes apresentam o passo a passo destas atividades.

Atividade 1: Escolha do estilo arquitetural

Para realizar a escolha do estilo arquitetural organizacional que auxiliará no projeto da arquitetura do sistema, Tropos utiliza o *framework* NFR [Chung00] e o catálogo de correlação entre os estilos arquiteturais organizacionais e requisitos não-funcionais, ou atributos de qualidade [Kolp01a] [Kolp01b].

A notação do *framework* NFR trata requisitos não-funcionais de forma explícita como metas a serem atingidas e representa as metas-*soft* e suas sub-metas como nuvens de contorno simples, operacionalizações dessas metas como nuvens de borda espessa, e razões do projeto como nuvens tracejadas. Estas podem representar informação de conteúdo (tais como prioridades) a serem consideradas e apropriadamente refletidas no processo de tomada de decisão. São consideradas as notações *help*, *make*, *hurt* e *break*, para modelar, respectivamente, contribuições *parcial/positiva* (+), *suficiente/positiva* (++) , *parcial/negativa* (-) e *suficiente/negativa* (--) entre essas metas. 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. Mais detalhes sobre a seleção e o processo de decomposição de requisitos não-funcionais podem ser encontrados em [Kolp01a] [Kolp01b].

Tabela 3.2: Catálogo de correlação entre estilos arquiteturais e atributos de qualidade

CATÁLOGO DE CORRELAÇÃO	Previsibilidade	Segurança	Adaptabilidade	Coordenação	Cooperação	Disponibilidade	Integridade	Modularidade	Agregabilidade
Estrutura Plana	Break	Break	Hurt			Help	Help	Make	Hurt
Estrutura em 5	Help	Help		Help	Hurt	Help	Make	Make	Make
Pirâmide	Make	Make	Help	Make	Hurt	Help	Break	Hurt	
União Estratégica	Help	Help	Make	Help	Hurt	Make		Help	Make
Oferta	Break	Break	Make	Hurt	Make	Hurt	Break	Make	
Tomada de Controle	Make	Make	Hurt	Make	Break	Help		Help	Help
Comprimento de Braço	Hurt	Break	Help	Hurt	Make	Break	Make	Help	
Contratação Hierárquica			Help	Help	Help	Help		Help	Help
Integração Vertical	Help	Help	Hurt	Help	Hurt	Help	Break	Break	Break
Apropriação	Hurt	Hurt	Make	Make	Help	Break	Hurt	Break	

Devido à natureza organizacional e intencional dos SMAs, os estilos têm sido avaliados e comparados usando atributos de qualidade de software identificados por arquiteturas que envolvem componentes coordenados autônomos, tais como *previsibilidade*, *segurança*, *adaptabilidade*, *coordenação*, *cooperação*, *disponibilidade*, *integridade*, *modularidade* e *agregabilidade*. A Tabela 3.2 apresenta o catálogo de correlação entre os estilos arquiteturais e estes atributos [Kolp01a] [Kolp01b].

Os principais atributos de qualidade (requisitos não-funcionais) que foram identificados como metas-soft no modelo SR da fase de Requisitos Finais são empregados para conduzir o processo de seleção do estilo arquitetural apropriado para o sistema. A partir deles, deve ser construído um grafo SIG (*Softgoal Interdependency Graph*) [Chung00] do *framework* NFR para representar os inter-relacionamentos entre os atributos de qualidade e seus refinamentos. Ao construir o grafo SIG, os atributos de qualidade identificados devem ser refinados para sub-metas, mais precisas e mais específicas do que eles. O próximo passo consiste em identificar interdependências positivas e negativas entre as metas-soft. Em seguida deve-se identificar e tratar prioridades. A partir daí, os estilos arquiteturais, representados como operacionalizações, devem ser avaliados com relação ao cumprimento das metas-soft refinadas. Após esta análise, deve-se optar por um dos estilos arquiteturais. A Figura 6.8 apresenta o grafo SIG utilizado no estudo de caso desta dissertação.

Atividade 2: Modelagem da arquitetura

Uma vez que o estilo arquitetural mais apropriado para representar a estrutura principal do sistema foi escolhido, os modelos SD e SR produzidos na fase de Requisitos Finais devem servir como base para criação do modelo SR da arquitetura do sistema. Normalmente é necessário, após a escolha do estilo arquitetural apropriado realizada na primeira atividade, o acréscimo de novos atores e novas dependências, bem como a decomposição dos atores e dependências existentes nos modelos SD e SR derivados dos requisitos finais.

Tanto durante esta atividade quanto na seguinte, deve-se continuar levando em consideração os requisitos levantados para o sistema. Isso implica que os atores devem ter capacidade de, através de suas tarefas, recursos, metas e metas-soft, cumprir os requisitos funcionais e não-funcionais do sistema desejados pelos stakeholders e definidos nas fases anteriores. Uma vez que a arquitetura do sistema foi modelada, podemos saber qual ator possui capacidade para resolver determinada parte do sistema.

Atividade 3: Aplicação dos padrões sociais

Os padrões sociais em Tropos são padrões de projeto que estão preocupados com aspectos sociais e intencionais recorrentes em sistemas cooperativos ou multi-agentes. Suas estruturas foram inspiradas nos padrões federados introduzidos em [Hayden99]. Esses padrões são usados para solucionar uma meta específica que foi definida em nível arquitetural, através da identificação de estilos organizacionais e atributos de qualidade relevantes (*meta-soft*).

Os padrões sociais de Tropos estão classificados em duas categorias [Kolp02]: *padrões entre pares* e *padrões de mediação*. Como exemplos de padrões entre 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].

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 determinada meta. Em particular, para cada capacidade o ator tem um conjunto de comportamentos que podem ser aplicados em diferentes situações. Um plano descreve a sequência de ações a executar e as condições sob as quais 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.

Nesta última atividade, precisamos definir como as metas associadas a cada ator são cumpridas por agentes quando aplicamos os padrões sociais definidos para a metodologia Tropos [Do03]. Deve ser construído mais um modelo SR do i^* , a partir do modelo da arquitetura sem os padrões de projeto, desta vez com a aplicação dos padrões sociais identificados que contribuirão com a estrutura da arquitetura do sistema. Os componentes arquiteturais identificados neste modelo deverão ser detalhados na fase de Projeto Detalhado.

3.3.4 Projeto Detalhado

A fase de projeto detalhado lida com a especificação do micro nível dos agentes, estando preocupada em apresentar detalhes adicionais para cada componente arquitetural do sistema. Isto inclui fundamentalmente os aspectos dos agentes que envolvem a comunicação e seus comportamentos. Nesta fase precisamos definir como os objetivos atribuídos a cada ator serão realizados pelos agentes.

Os padrões sociais que foram identificados na arquitetura do sistema também são detalhados nesta fase, de acordo com o catálogo de padrões sociais definidos em [Kolp05]. Além disso, as abordagens práticas para o projeto detalhado são usualmente propostas dentro de plataformas de desenvolvimento específicas e dependem de outras características da linguagem de programação de agentes adotada [Bresciani04], o que relaciona estritamente o projeto detalhado com as escolhas de implementação.

Em Tropos, o projeto detalhado recebe como entrada as especificações resultantes da fase de projeto arquitetural e as razões para um dado elemento que, mesmo projetadas detalhadamente, podem ser rastreadas para a análise dos requisitos iniciais [Bresciani04]. Tropos propôs e definiu em [Mylopoulos01], um conjunto de estereótipos, etiquetas (*tagged values*) e restrições para acomodar seus conceitos dentro da UML, tais como $\ll i^* actor \gg$ e $\ll i^* dependency \gg$, que podem ser usados na modelagem de diagramas de classes e demais diagramas.

Na modelagem do projeto detalhado, utilizamos diagramas de atividade da UML [Booch99] para representar capacidades e planos, e adotamos um subconjunto dos diagramas AUML (*Agent Unified Modeling Language*) propostos em [Bauer01] para especificar as interações entre os agentes. Para representar os protocolos de interação e as trocas de mensagens entre os agentes, Tropos utiliza os diagramas de sequência da AUML [Odell00]. O que difere este tipo de diagrama de um diagrama de sequência da UML [Booch99] é o fato de que as mensagens são enviadas assincronamente, bem

como os agentes podem escolher o caminho da interação, de acordo com suas crenças, intenções, objetivos ou condições.

Com relação ao suporte para comunicação dos agentes, Tropos sugere a adoção de linguagens de comunicação de agentes já existentes, tais como FIPA-ACL [Labrou99] ou KQML [Finin97], mecanismos de transporte de mensagens, entre outros conceitos e ferramentas.

3.3.5 Implementação

Na fase de Implementação, a especificação do Projeto Detalhado deve ser seguida para geração de um esqueleto para implementação do SMA. Isto é feito através de um mapeamento entre os conceitos de Tropos e os elementos de uma plataforma de implementação de agente, tal como JACK [Coburn01, JACK05] ou JADE [Bellifemine05]. Este mapeamento serve para guiar o processo de implementação de SMA em determinada plataforma. O código do sistema deve complementar a estrutura básica de implementação, oferecida pelo esqueleto do sistema obtido através do mapeamento, usando a linguagem de programação suportada pela plataforma de implementação de agentes escolhida. No caso do esqueleto gerado para JACK, o restante do código do sistema deve ser implementado em Java [Deitel03].

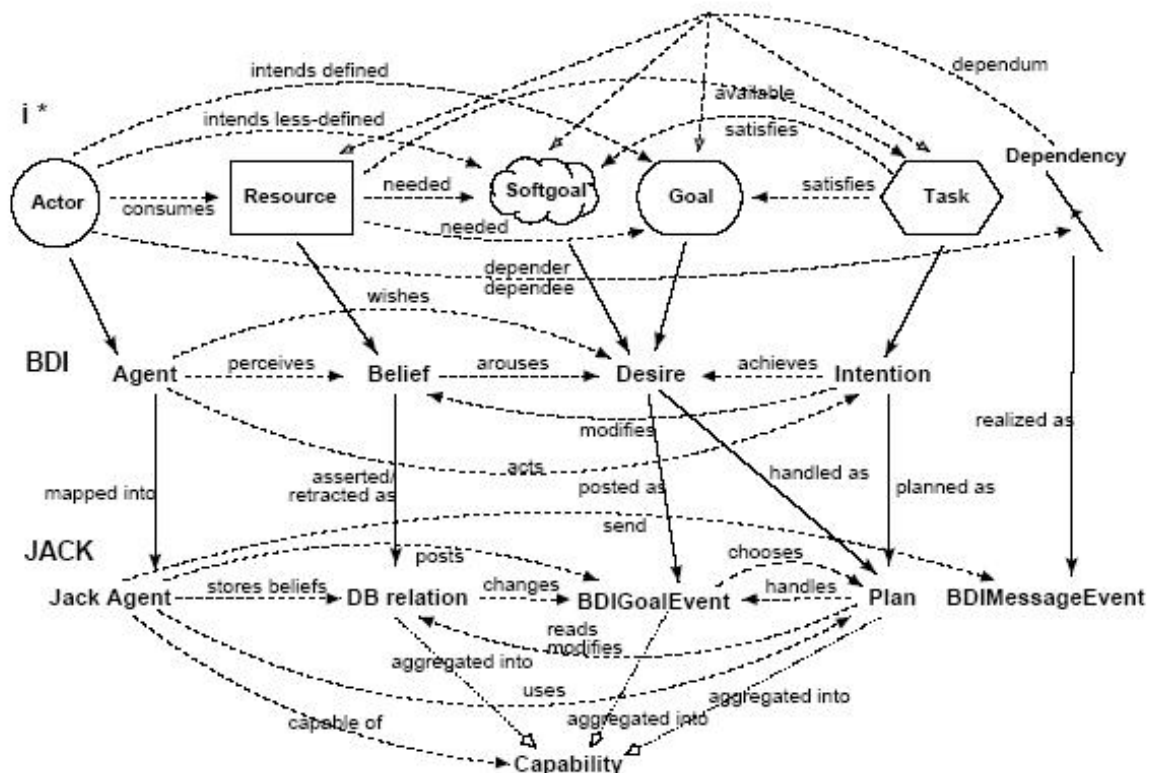


Figura 3.6: Visão geral do mapeamento i*/BDI/JACK [Castro02a]

Da mesma forma que diversas ferramentas CASE, tais como Rational Rose [Rational99] e Together [Borland05], incluem geradores de código para padrões de projeto orientados a objetos, a ferramenta SKwyRL [Do04] 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 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.

A Figura 3.6 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).

3.4 Considerações Finais

A definição de metodologias de desenvolvimento de softwares orientados a agentes vem se tornando cada vez mais uma área promissora da engenharia de software. Algumas metodologias de desenvolvimento orientadas a agentes já foram propostas e estendem ou adaptam metodologias orientadas a objeto para propósitos da engenharia de software orientada a agentes, tais como GAIA [Zambonelli03] e AUML [Bauer01], ou adaptam a engenharia do conhecimento. Entretanto, tais propostas estão principalmente preocupadas com o projeto de software orientado a agentes não dando muita importância, por exemplo, à análise de requisitos, e conseqüentemente deixando de abordar todas as fases do ciclo de desenvolvimento do sistema.

Este capítulo apresentou a metodologia Tropos [Bresciani04], que oferece um framework de desenvolvimento orientado a agentes considerando a modelagem de características deste paradigma desde a fase de requisitos. Tropos provê técnicas para construir modelos que incluem os conceitos de atores (agentes, posições e papéis), suas metas e suas interdependências intencionais. Comparado à análise orientada a objetos [Booch94], o Tropos provê um nível mais alto de modelagem, introduzindo a abstração de agente para a modelagem de requisitos a fim de dar suporte à elicitación, exploração e à análise dos sistemas considerando o seu ambiente organizacional, das possíveis alternativas em como os agentes se relacionam uns com os outros e os prós e os contras destas alternativas.

O framework Tropos está em desenvolvimento e atualmente cobre as seguintes fases do ciclo de vida do software: Requisitos Iniciais, Requisitos Finais, Projeto Arquitetural, Projeto Detalhado e Implementação. Comparado a outras metodologias, Tropos atualmente tem se apresentado como uma proposta muito adequada por abordar o ciclo de desenvolvimento de software mais completamente do que as demais metodologias, apesar da ausência de um processo bem-definido para o desenvolvimento de software baseando-se na metodologia Tropos.

Ainda assim, 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 importantes áreas de aplicação, tais como telecomunicações, comércio eletrônico e sistemas baseados na *Web*.

No entanto, a metodologia Tropos está em desenvolvimento e apresenta algumas deficiências, como por exemplo:

- Ausência de mecanismos de estruturação adequados que permitam construir modelos simplificados e fáceis de entender quando aplicados a domínios complexos;
- Ausência de um conjunto de ferramentas integradas que suporte todas as fases da metodologia Tropos;
- Ausência de uma linguagem de restrições que permita representar pré e pós-condições e ordenação na execução de tarefas nos modelos Tropos;
- Ausência de uma linguagem formal completa que permita formalismo e checagem de modelo, simulação de modelo e mecanismos temporais, além de provadores de teoremas.

Dentre as várias fases que Tropos necessita evoluir, encontra-se o projeto detalhado de software orientados a agentes. Nessa dissertação nós propomos refinar a fase de projeto detalhado da proposta Tropos. Contudo, antes de detalharmos nossa abordagem, no próximo capítulo iremos fazer uma introdução aos conceitos básicos relacionados à plataforma de desenvolvimento de agentes JADE.

Capítulo

4 Desenvolvendo SMA com JADE

Este capítulo aborda o desenvolvimento de Sistemas Multi-Agentes utilizando a plataforma de desenvolvimento JADE. Primeiramente apresentamos as principais características da plataforma. Em seguida, são apresentados os conceitos e estruturas relacionados à implementação de agentes, mensagens, protocolos, comportamentos e ontologias em JADE. O capítulo é concluído com a apresentação da estrutura e funcionamento da plataforma de agentes JADE, através de suas ferramentas de gerenciamento.

4.1 Introdução

Os sistemas de software tradicionais tornam difícil a modelagem de comportamentos racionais, e por isso softwares com “agentes” escritos de forma tradicional apresentam frequentemente muitas limitações, principalmente quando objetivam operar em ambientes que exijam execução em tempo real. A programação orientada a agentes é um paradigma de modelagem de software que surgiu a partir da inteligência artificial distribuída e está direcionado para a necessidade de sistemas de software exibirem comportamentos racionais dentro dos seus respectivos domínios de problema.

O desenvolvimento de software orientado a agentes satisfaz adequadamente às necessidades de muitas áreas de aplicação, incluindo ferramentas inteligentes, simuladores, e sistemas de negócio distribuídos. Embora ainda seja um paradigma recente, já tem demonstrado que pode ser bem aplicado para resolução de vários tipos de problemas, tais como gerenciamento de tráfego aéreo e simulações de combate [Luck03]. Devido à sua capacidade de oferecer uma solução elegante e modular para muitos problemas que envolvem processamento em tempo real e com muitas mudanças organizacionais, a programação orientada a agentes apresenta-se adequada para esses sistemas.

O modelo de orientação a agentes segue os mesmos princípios fundamentais da programação orientada a objetos, levando em consideração que o aumento da escalabilidade e confiabilidade de um sistema pode ser garantido através do encapsulamento dos comportamentos desejados para os agentes em unidades modulares, que contenham as definições e estruturas requeridas para que eles possam operar independentemente no ambiente. Agentes estendem o conceito de encapsulamento para incluir a representação de comportamentos em um nível de abstração alto como em abordagens orientadas a objetos.

4.2 A plataforma JADE

JADE (*Java Agent Development Framework*) [JADE05] [Bellifemine05] é um *framework* para desenvolvimento de SMA, completamente implementado em Java [Deitel03], cujo objetivo principal é facilitar o desenvolvimento de sistemas orientados a agentes.

JADE provê tecnologias de agentes padronizadas e oferece para o desenvolvedor algumas características com o intuito de simplificar o processo de desenvolvimento, dentre elas podemos citar:

- Fornece um ambiente de agentes compatível com as especificações da FIPA, que inclui o sistema gerenciador de agentes (AMS - Agent Management System), o facilitador de diretórios (DF - Directory Facilitator) e o canal de comunicação de agentes (ACC - Agent Communication Channel). Esses componentes são automaticamente iniciados quando o ambiente de desenvolvimento é executado.
- Possui uma interface visual que dá suporte ao gerenciamento de vários agentes e conjuntos de agentes, inclusive remotamente;
- Oferece ferramentas que auxiliam o desenvolvimento e a depuração de aplicações multi-agentes implementadas em JADE;
- Dá suporte a execuções múltiplas, paralelas ou concorrentes de atividades de agentes, através dos modelos de comportamentos disponíveis na sua API;
- Dispõe de uma biblioteca de protocolos da FIPA para interação entre agentes pronta para ser usada;
- Automatiza no AMS os registros e cancelamento de registros dos agentes na plataforma, permitindo que o registro dos agentes seja transparente para o desenvolvedor;
- Realiza o transporte de mensagens no formato FIPA-ACL dentro da mesma plataforma de agentes;
- Possui um serviço de nomes em conformidade aos padrões FIPA, permitindo que os agentes recebam um identificador único na sua inicialização válido para todo ambiente;
- Oferece mecanismos que tornam possível a sua integração com aplicações externas que executem agentes de JADE.

Em JADE, um comportamento representa uma tarefa que um agente pode realizar. Comportamentos são *threads* de execução lógica que podem ser compostos de várias maneiras para atingir padrões de execução, podendo ser inicializados, suspensos ou gerados em qualquer momento [Bellifemine04]. Uma das características mais importantes que os agentes de JADE provêm é a capacidade de comunicação. O paradigma de comunicação adotado é o de passagem assíncrona de mensagens. Cada agente tem um tipo de caixa de mensagens (a fila de mensagens do agente) onde o ambiente de execução de JADE afixam mensagens postadas por outros agentes. As mensagens trocadas por agentes de JADE têm um formato especificado pela linguagem ACL definida pelo padrão internacional da FIPA para interoperabilidade de agentes.

O serviço de páginas amarelas em JADE obedece à especificação da FIPA e é representado por um agente chamado DF (Directory Facilitator). Por sua vez, o suporte para linguagens de conteúdo e ontologias fornecido por JADE é projetado para executar automaticamente a conversão de uma representação interna da informação do agente dentro da representação da expressão de conteúdo ACL correspondente [Bellifemine04].

4.2.1 Principais Características de JADE

JADE é um software livre desenvolvido e é distribuído pelo TILAB [TILAB05], de acordo com a licença LGPL (*Lesser GNU Public License*) [GNU99]. Desde maio de 2003, um comitê foi criado com o objetivo de supervisionar o gerenciamento do Projeto JADE. Atualmente esse comitê possui cinco membros: TILAB, Motorola, Whitestein Technologies AG., Profactor GmbH, e a France Telecom R&D. A última versão de JADE é JADE 3.3, lançada em 2 de março de 2005. Contudo, em nosso trabalho utilizamos a versão 3.0b1, disponível no início de nossas pesquisas.

JADE foi desenvolvido em Java devido a características próprias da linguagem, particularmente pela programação orientada a objetos em ambientes distribuídos heterogêneos [Bellifemine04]. Foram desenvolvidos tanto pacotes Java com funcionalidades prontas pra uso, quanto interfaces abstratas para serem adaptadas conforme as funcionalidades requeridas para determinada aplicação de agentes. Ou seja, JADE oferece uma API para desenvolvimento de agentes em Java e uma plataforma distribuída de agentes em conformidade com a FIPA. Toda comunicação entre agentes em JADE é feita através de trocas de mensagens.

Uma outra característica de JADE é que sua plataforma de agentes pode ser distribuída por vários *hosts*, cada um deles executando apenas uma JVM (*Java Virtual Machine*). Os agentes são implementados como *threads* Java e inseridos dentro de repositórios de agentes chamados de *containers*, que provêm todo o suporte para a execução do agente e representam o ambiente de execução das aplicações de agentes. O conjunto de todos os *containers* é chamado de plataforma [Bellifemine04], a qual provê uma camada homogênea que encapsula para os agentes, bem como para os desenvolvedores de aplicações, a complexidade e a diversidade de características tais quais: hardware, sistemas operacionais, tipos de redes ou JVMs.

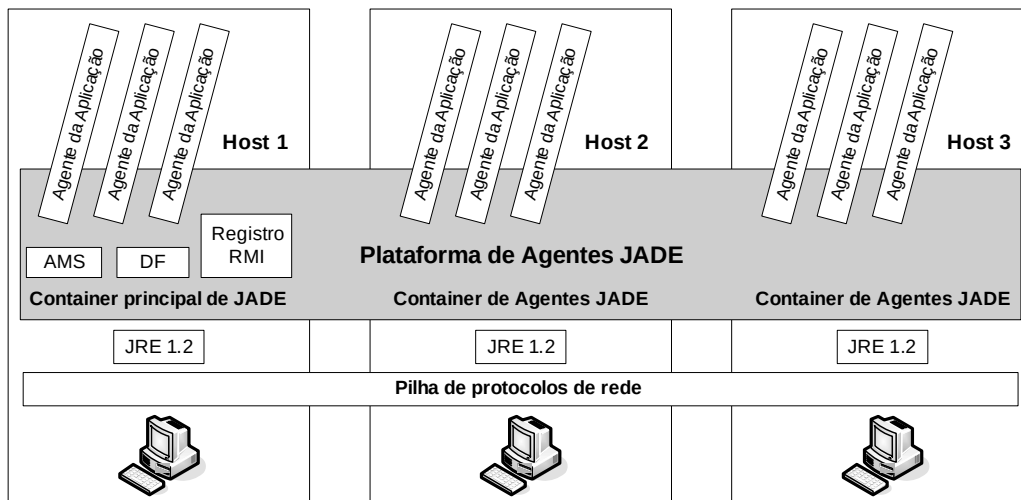


Figura 4.1: Plataforma de Agentes JADE distribuída por vários *containers*

Todo *container* equivale a um processo e podemos ter diferentes *containers* na mesma plataforma, sendo uma JVM por *container*. Além disso, diferentes agentes podem existir no mesmo *container* porque cada agente tem sua própria *thread* de execução e a JVM escalona um ambiente *multi-*

threaded pré-emptivo. A comunicação entre JVMs é feita através da invocação remota de métodos (RMI - *Remote Method Invocation*) de Java.

A Figura 4.1 apresenta a estrutura da plataforma de agentes JADE distribuída por vários *hosts*. O *container* principal, localizado no *host 1*, é o *container* onde se encontra o AMS, o DF e registro RMI (*Remote Method Invocation Registry*). Esse registro RMI é um servidor de nomes que Java utiliza para registrar e recuperar referências a objetos através do seu nome. Ou seja, é o meio que JADE usa em Java para manter as referências aos outros *containers* de agentes que se conectam à plataforma. Os outros *containers* de agentes são conectados ao *container* principal, fazendo com que o desenvolvedor fique abstraído da separação física dos *hosts*, caso exista, ou das diferenças existentes entre os *hosts*.

A plataforma JADE integra todos os *hosts* atuando como um elo de ligação e provê um completo ambiente de execução para qualquer conjunto de agentes. JADE possui algumas ferramentas úteis para simplificar a administração da plataforma de agentes e o desenvolvimento de aplicações. A seção 4.2.7 abordará essas ferramentas com mais detalhes.

4.2.2 Agentes em JADE

Agentes em JADE são processos que representam entidades autônomas independentes, que possuem uma identidade e a capacidade de se comunicar com outros agentes para atingir os seus objetivos de execução [JADE05], podendo realizar múltiplas tarefas e conversações de forma simultânea.

Um agente em JADE é uma instância da classe *Agent*, contida no pacote *jade.core*, ou de alguma subclasse de *Agent*. Esta classe facilita o desenvolvimento de agentes porque provê todas as características necessárias para realizar as interações básicas com a plataforma, tais como métodos para registro, configuração ou gerenciamento remoto do agente. Além disso, a estrutura da classe *Agent* também fornece o conjunto básico de métodos que podem ser chamados para implementar o comportamento personalizado do agente, como por exemplo o envio de mensagens e a utilização de protocolos de interação.

Cada serviço ou funcionalidade de um agente deve ser implementado como um ou mais comportamentos. Os comportamentos de JADE modelam arquiteturas internas reativas, contudo, a abstração de comportamento do modelo do agente de JADE permite a integração de softwares externos para enriquecer a arquitetura do agente. Isso é possível porque nós podemos integrar JADE com alguma estrutura para projetar, por exemplo, a base de conhecimentos do agente usando Jeops [Figueira00]. Além disso, JADE oferece também bibliotecas para implementação de diversas características dos agentes, como por exemplo mobilidade e clonagem. Estas características de JADE permitem o desenvolvimento de sistemas mais sofisticados uma vez que é possível, além de tornar os agentes capazes de migrar entre plataformas, agregar em uma mesma aplicação: agentes de aprendizado, agentes baseados em utilidade, agentes móveis, e assim por diante. A definição de ontologias para mobilidade de agentes em JADE está disponível no pacote *jade.domain.mobility*, ela inclui uma lista de símbolos utilizados para mobilidade e de classes que implementam estas ontologias.

Todo agente possui uma identificação única, denominada de AID (*Agent Identifier*). Esse identificador é constituído por: um nome globalmente único que deve obedecer à estrutura `<localname>@<hostname>:<port>/JADE`, como por exemplo: `ams@G6C15:1099/JADE`; um conjunto

de endereços de agentes (uma vez que cada agente herda os endereços de transporte de sua plataforma de origem); e um conjunto de solucionadores, ou seja, páginas brancas nas quais o agente é registrado.

A seguir são listados os principais métodos da classe *Agent*:

- ***protected void setup()***: é um método protegido, indicado para inserção da codificação de inicialização do agente. Este método pode ser sobrescrito para fornecer os comportamentos necessários ao agente, além de tarefas comuns de inicialização como por exemplo, o registro no DF;
- ***public void addBehaviour(Behaviour comportamento)***: adiciona um novo comportamento ao agente, que será executado concorrentemente com outros comportamentos do agente. Geralmente este método é chamado dentro do método *setup ()* para disparar algum comportamento inicial, mas pode ser usado também para gerar comportamentos dinamicamente. Inversamente, o método *removeBehaviour ()* realiza o processo de remoção de determinado comportamento do agente;
- ***public final void send(ACLMessage mensagem)***: envia uma mensagem ACL para outro agente. Esse agente destino é especificado no campo *receiver* da mensagem na qual um ou mais agentes podem ser definidos como receptores;
- ***public final ACLMessage receive()***: recebe uma mensagem ACL da fila de mensagens do agente. Este método não é bloqueável e retorna a primeira mensagem da fila caso ela possua mensagens ou o valor null caso a fila esteja vazia.

A criação de uma nova instância de agente em JADE ocorre da seguinte maneira: o método construtor do agente é executado e o agente recebe um AID, em seguida o agente é registrado no serviço de páginas brancas (AMS) e colocado no estado AP_ACTIVE, só então o seu método *setup()* é executado. O método *setup()*, por sua vez, necessariamente deve ser implementado para inicialização de um agente, devendo adicionar pelo menos um comportamento para o agente através do método *addBehaviour()*. Ao final da execução do método *setup()*, JADE executa automaticamente os comportamentos, a partir do que foi primeiramente declarado, usando um escalonamento circular não-preemptivo.

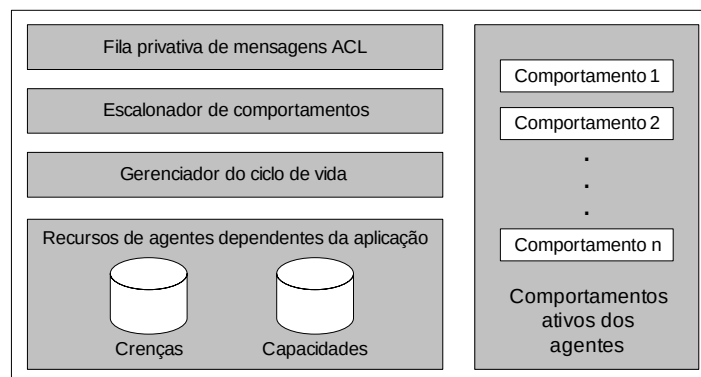


Figura 4.2: Arquitetura interna de um agente genérico em JADE

As estruturas internas dos agentes não são especificadas pela FIPA. Contudo, a interoperabilidade entre os agentes pode ser garantida através das estruturas especificadas para os comportamentos externos dos agentes, tais como protocolos e linguagens de conteúdo, definidos em conformidade com a FIPA [FIPA04].

A Figura 4.2 apresenta a descrição de uma arquitetura interna para um agente genérico em JADE. Podemos ver nesta arquitetura que todo agente tem uma fila privativa de mensagens ACL, podendo decidir quando e quais mensagens recebidas ele quer ler. O subsistema de comunicação de JADE foi projetado de forma a alcançar um menor custo possível na passagem de mensagens. Se uma mensagem é enviada e o subsistema não consegue encontrar o destinatário, ele a envia para ser gerenciada pelo AMS. Associado ao agente também temos um escalonador de comportamentos e um gerenciador do seu ciclo de vida. O fato de cada agente em JADE possuir o seu próprio gerenciador permite que ele seja mais autônomo, por poder controlar completamente sua *thread* de execução.

Um agente genérico em JADE pode ter estruturas dependentes de cada aplicação para armazenar recursos como crenças e capacidades. Ao mesmo tempo, cada agente possui um conjunto de comportamentos ativos associados que representam as tarefas, funcionalidades ou serviços que eles têm capacidade de realizar. Esses comportamentos podem ser executados concorrentemente porque o modelo computacional de um agente em JADE é multitarefa.

O Ciclo de Vida de um Agente

De acordo com a especificação da FIPA para ciclo de vida de uma plataforma de agentes [FIPA04], um agente pode estar em um dos seis estados apresentados na Figura 4.3: *Desconhecido*, *Ativo*, *Suspenso*, *Iniciado*, *Em trânsito* ou *Em espera*. Ainda existem mais dois estados que são utilizados internamente por JADE: *Copiado* e *Migrado*. Sua representação em JADE é feita através de constantes estáticas da classe *Agent*. As constantes que representam os possíveis estados de agentes na plataforma JADE são os seguintes:

- **AP_INITIATED**: o objeto da classe *Agent* foi instanciado, mas ainda não solicitou registro no AMS, não tem nome ou endereço e não pode comunicar com outros agentes;
- **AP_ACTIVE**: o objeto da classe *Agent* está registrado no AMS, tem nome formal e endereço e tem acesso a todas as funcionalidades de JADE;
- **AP_SUSPENDED**: o objeto da classe *Agent* está suspenso no momento. Sua *thread* interna está suspensa e nenhum comportamento está sendo executado;
- **AP_WAITING**: o objeto da classe *Agent* está bloqueado. Sua *thread* interna está em estado de espera e será reativada quando alguma condição ocorrer, por exemplo, quando o agente receber determinada mensagem;
- **AP_DELETED**: o agente está definitivamente encerrado, foi removido da plataforma. Sua *thread* interna terminou sua execução e o agente não está mais registrado no MAS;
- **AP_TRANSIT**: um agente móvel entra nesse estado enquanto estiver migrando para uma outra plataforma. O sistema continua a armazenar as mensagens para o agente em um fila, mas estas mensagens serão redirecionadas para essa nova localização;
- **AP_COPY**: a plataforma utiliza internamente este estado para indicar os agentes que foram clonados;
- **AP_GONE**: esse estado também é usado internamente por JADE, para indicar que um agente móvel migrou para uma outra localização e se encontra em uma situação estável.

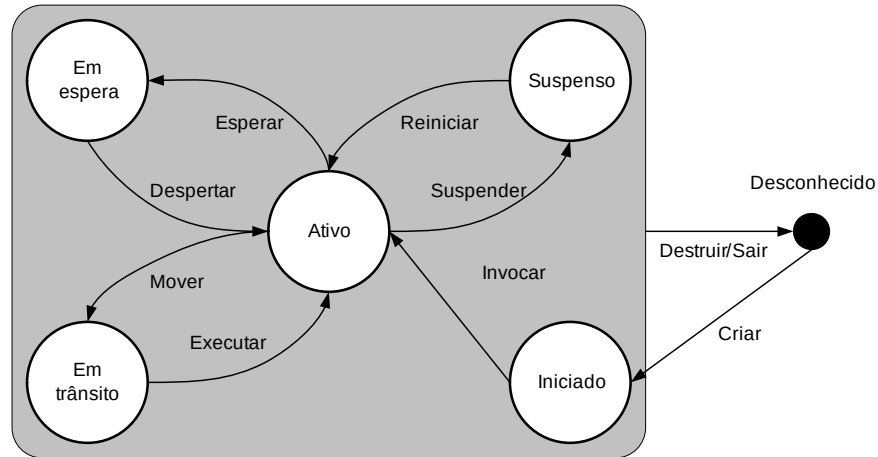


Figura 4.3: Ciclo de vida de um agente definido pela FIPA

A troca de estados dos agentes realizada por JADE para gerenciar o ciclo de vida dos agentes, conforme apresentado na Figura 4.3, pode ser feita através da chamada a métodos públicos disponibilizados na classe *Agent*, são eles:

- **public void doActivate ()**: realiza a transição do estado suspenso para um dos estados: ativo ou em espera (estado no qual o agente estava quando *doSuspend()* foi executado). Esse método é chamado da plataforma de agentes e reinicia a execução do agente. Chamar *doActivate()* em um agente que não esteja suspenso não faz efeito algum;
- **public void doSuspend ()**: faz a transição do estado ativo ou em espera para o estado de suspenso. O estado original do agente é armazenado e será restaurado pela chamada ao método *doActivate()*. Esse método pode ser chamado pela plataforma de agentes ou pelo próprio agente que interrompe todas suas atividades. Mensagens que chegam para um agente suspenso são armazenadas pela plataforma e são entregues assim que o agente reinicie suas atividades. Chamar *doSuspend ()* de um agente suspenso não faz efeito algum;
- **public void doWait ()**: realiza a transição do estado ativo para o estado em espera. Esse método pode ser chamado pela plataforma de agentes ou pelo próprio agente que causa um bloqueio dele mesmo, interrompendo todas as suas atividades até que alguns eventos ocorram. Um agente em estado de espera sai desse estado quando método *doWake ()* é chamado ou quando alguma mensagem é recebida. Chamar o método *doWait ()* em agentes suspensos ou em espera não faz efeito algum;
- **public void doWake ()**: faz a transição do estado em espera para o estado ativo. Esse método é chamado pela plataforma de agentes e reinicia a execução do agente. Chamadas ao método *doWake()* em agentes que não estiverem em estado de espera não fazem efeito.
- **public void doDelete ()**: faz a transição dos estados ativo, suspenso ou em espera para o estado AP_DELETED. Esse ato causa a destruição do agente e pode ser chamado pela plataforma de agentes ou pelo próprio agente. Chamar *doDelete()* em um agente já destruído não causa efeito algum;
- **public void doMove (Location destino)**: realiza a transição do estado ativo para o estado em trânsito. Esse método tem o objetivo de dar suporte à mobilidade de agentes e é chamado pela plataforma de agentes ou pelo próprio agente para iniciar o processo de migração;
- **public void doClone (Location destino, java.lang.String nomeClone)**: faz a transição do estado ativo para o estado AP_COPY. Esse método tem o objetivo de dar suporte à mobilidade

de agentes e é chamado pela plataforma de agentes ou pelo próprio agente para iniciar o processo de clonagem.

Os agentes em JADE só têm permissão para executar seus comportamentos quando estiverem no estado ativo. Se o agente estiver em qualquer estado e um de seus comportamentos chamar o método *doWait ()*, além do agente ficar no estado de espera, ele e todas suas atividades serão bloqueadas, não só o comportamento que chamou este método. O método *block ()* da classe *Behaviour* pode ser chamado para evitar o bloqueio de todos os comportamentos porque, além de permitir o desbloqueio do agente caso uma mensagem seja recebida, realiza a suspensão de apenas um comportamento.

4.2.3 Comportamentos de Agentes

As ações ou comportamentos que um agente desempenha dentro de um SMA são fundamentais para que o objetivo final do sistema seja alcançado. JADE contém comportamentos específicos para a maioria das tarefas comuns na programação de agentes, tais como envio e recebimento de mensagens, bem como para construção de tarefas mais complexas. No desenvolvimento orientado a agentes com JADE, o agente deve obrigatoriamente ser uma instância direta ou indireta da classe *Agent* e implementar tarefas específicas escrevendo um ou mais comportamentos através das subclasses de *Behaviour*, instanciando-as e adicionando-as ao agente.

A estrutura dos comportamentos de JADE é suportada por um escalonador, interno à superclasse *Agent* e transparente para o desenvolvedor, que gerencia automaticamente o escalonamento desses comportamentos usando a política de escalonamento *Round-Robin* não-preemptivo em todos os comportamentos disponíveis na fila de pronto [Bellifemine05]. 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.

Do ponto de vista de programação concorrente, um agente é um objeto ativo com uma *thread* de controle interna. JADE usa o modelo de uma *thread* por agente ao invés de uma *thread* por tarefa ou conversação, com o objetivo de manter um número pequeno de *threads* necessárias para executar a plataforma de agentes.

A classe *Behaviour*

Behaviour é uma classe abstrata de JADE disponível no pacote *jade.core.behaviours* e que pode ser usada para modelar uma tarefa genérica do agente. Por ser abstrata é uma classe que possui alguns métodos implementados e outros não [Deitel03], desta forma, a classe *Behaviour* tem como finalidade prover a estrutura de comportamentos que podem ser implementados para execução de agentes em JADE.

Os dois métodos principais da classe *Behaviour* são abstratos: *action()* e *done()*. O método *action()* deve ser implementado para realizar as tarefas ou ações que serão desempenhadas pelo agente através deste comportamento. O método *done ()* 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, fazendo com que o método *action()* seja novamente executado.

Diversos outros métodos estão disponíveis para classe *Behaviour*, dentre os quais podemos citar:

- ***onEnd()***: é um método invocado apenas uma vez, após o comportamento ter sido encerrado;
- ***onStart()***: é um método executado uma única vez também, mas antes do início da execução do comportamento do agente;
- ***void block()***: é um método utilizado para bloquear o comportamento. É possível passar como parâmetro desse método um tempo de bloqueio em milissegundos;
- ***void restart()***: reinicia um comportamento bloqueado;
- ***void reset()***: restaura o estado inicial do comportamento;
- ***boolean isRunnable()***: retorna se o comportamento está bloqueado ou não.

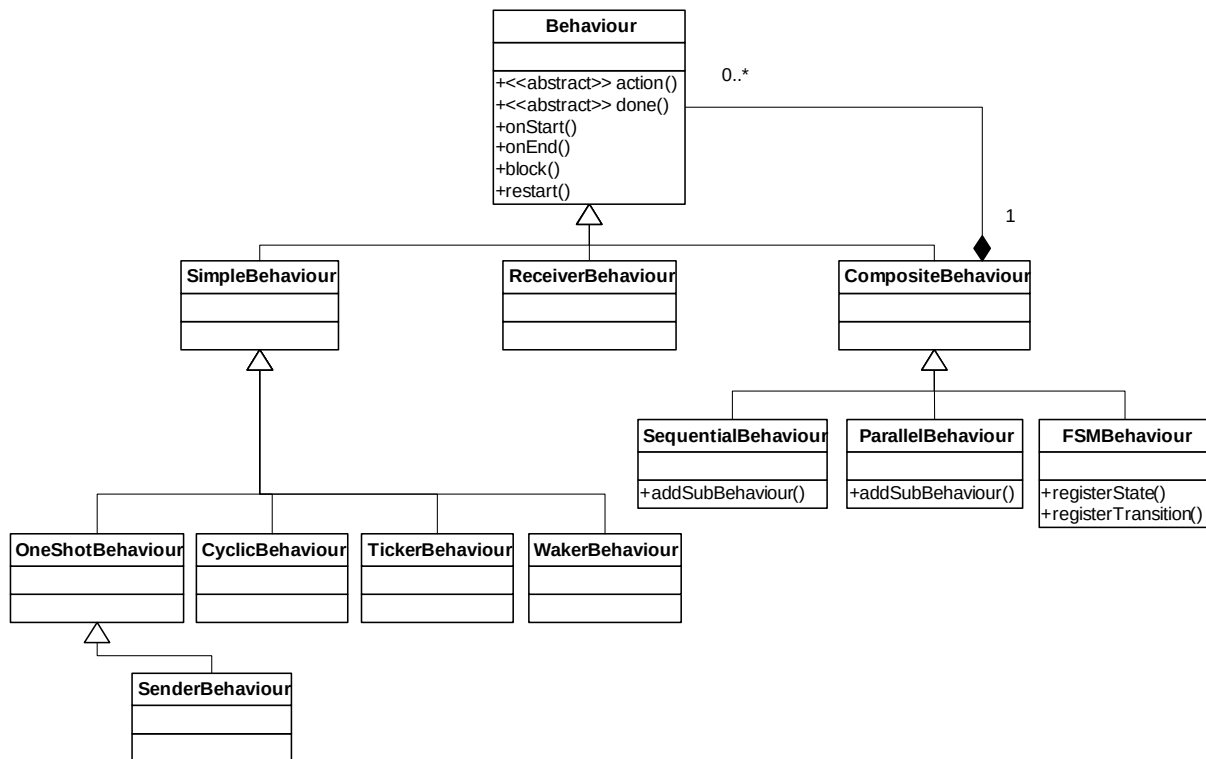


Figura 4.4: Hierarquia das classes de comportamentos em JADE derivadas de *Behaviour*

JADE possui várias classes de comportamentos prontas para uso pelo desenvolvedor adequando-as de acordo com a necessidade específica do agente. A Figura 4.4 apresenta a hierarquia de classes que derivam de *Behaviour*.

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

SimpleBehaviour, por sua vez, é uma classe abstrata que deve ser utilizada para modelar comportamentos que não pode ter sub-comportamentos. Isto é, modela comportamentos que são feitos para serem únicos, monolíticos, e que não podem ser interrompidos. Possui quatro classes herdadas disponibilizadas pelo JADE. São elas:

- ***jade.core.behaviours.CyclicBehaviour***: define um comportamento simples que deve ser executado sempre. É uma classe abstrata que pode ser herdada para criação de comportamentos cuja execução será contínua. Essa característica deste tipo de comportamento faz com que ele se repita como se estivesse em um *loop* infinito, porque o método *done()* herdado da super classe *Behaviour*, sempre retorna o valor falso para comportamentos cíclicos;
- ***jade.core.behaviours.TickerBehaviour***: representa comportamentos que precisam ser executados periodicamente para realizar tarefas específicas. *TickerBehaviour* é uma classe abstrata que implementa um comportamento cujo objetivo é executar periodicamente um pedaço de código definido pelo desenvolvedor em uma determinada frequência de repetições. O método *onTick()* deve ser redefinido para incluir o trecho de código que representa as ações que devem ser executadas periodicamente. O intervalo entre uma repetição e outra deve ser passado em milisegundos no construtor da classe;
- ***jade.core.behaviours.OneShotBehaviour***: modela uma tarefa atômica. Essa classe abstrata pode ser herdada para a criação de comportamentos para tarefas simples que precisam ser executadas apenas uma única vez (seu método *done()* retorna *true*). 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. Seu funcionamento restringe-se a realizar o comportamento de enviar determinada mensagem ACL e logo em seguida ser finalizado. A mensagem ACL a ser enviada é passada como parâmetro no construtor da classe;
- ***jade.core.behaviours.WakerBehaviour***: representa um comportamento simples que deve ser executado depois de determinado tempo expirado. Em outras palavras, é uma classe abstrata que uma tarefa “*OneShot*” que 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.

A classe ***CompositeBehaviour*** é uma classe abstrata para realizar a modelagem de comportamentos que representam tarefas complexa, ou seja, tarefas que são compostas por outras tarefas. Ela controla internamente seus sub-comportamentos seus intervalos de execução de acordo com alguma política de escalonamento. *CompositeBehaviour* possui três classes filhas disponibilizadas por JADE, são elas:

- ***jade.core.behaviours.FSMBehaviour***: representa um comportamento composto que possui o escalonamento de seus sub-comportamentos de acordo com uma máquina de estados finitos (*Finite State Machine*) definida pelo desenvolvedor. Nessa estrutura, cada sub-comportamento representa um estado na máquina de estados finitos. Esta classe de comportamento possui métodos para registrar estados e transições que definem como será realizado o escalonamento desses sub-comportamentos. Um *FSMBehaviour* deve ser definido a partir dos seguintes passos:

1) Determinar um único comportamento como estado inicial, usando o *registerFirstState()* e passando como parâmetros o comportamento e o nome do estado;

- 2) Determinar um ou mais comportamentos como estados finais através do método *registerLastState()*;
- 3) Determinar um ou mais comportamentos como estados intermediários através do método *registerState()*;
- 4) Determinar, para cada estado, as transições deste com os outros estados através do método *registerTransition()*.

- ***jade.core.behaviours.ParallelBehaviour***: define um comportamento composto com escalonamento concorrente de sub-comportamentos. Sua finalização dá-se quando alguma condição com relação à execução de seus sub-comportamentos é atendida. Essas condições são definidas passando-se como parâmetro no construtor da classe *ParallelBehaviour* um valor inteiro ou uma das constantes: *WHEN_ALL* ou *WHEN_ANY*. O valor inteiro é utilizado para definir o número de sub-comportamentos finalizados que representa a condição para finalização do comportamento paralelo. As constantes, por sua vez, são empregadas para definir que todos os comportamentos ou algum dos comportamentos, respectivamente, precisam ser encerrados para que encerre a execução da instância de *ParallelBehaviour*;
- ***jade.core.behaviours.SequentialBehaviour***: representa um comportamento composto com escalonamento seqüencial de sub-comportamentos. Ou seja, os seus sub-comportamentos são executados numa ordem seqüencial e o comportamento que foi decomposto apenas pode ser encerrado quando todos os seus sub-comportamentos tiverem finalizado sua execução. Para adicionar sub-comportamentos a comportamentos compostos do tipo *ParallelBehaviour* e *SequentialBehaviour* devemos chamar o método *addSubBehaviour()*.

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()* de comportamentos. Isso porque enquanto o método *action()* de algum comportamento estiver sendo executado, nenhum outro comportamento deste mesmo agente poderá ser executado até que este método finalize sua execução. Não é possível, por exemplo, interromper um comportamento durante a execução de um *action()*, passar o controle para outros comportamentos e voltar para o comportamento original de local onde tinha sido interrompido [Bellifemine04].

4.2.4 Mensagens em JADE

Sistemas baseados em agentes possuem intrinsecamente uma arquitetura *peer-to-peer*, onde cada agente é um software executando em um nó da rede. Os agentes necessitam iniciar comunicação com outros agentes, sendo capazes de prover serviços a eles. Isso torna o papel da comunicação em sistemas baseados em agentes muito importante. O modelo de comunicação em JADE é fundamentado em três características principais [Bellifemine04]:

- Agentes são entidades ativas, elas podem responder “não” e são fracamente acopladas. O grau de dependência entre os agentes é minimizado porque a plataforma permite que o agente escolha de forma flexível os destinatários das mensagens que deseja enviar. Além do mais, cada agente decide que mensagens ele quer interpretar ou descartar, também podendo, por exemplo, controlar o seu ciclo de execução para aguardar respostas a mensagens que enviou.
- Agentes realizam ações e a comunicação é considerada com apenas mais um tipo de ação. Isso quer dizer que os agentes tanto podem realizar ações físicas quanto ações comunicativas e, com respeito a essas ações, também podemos ter o planejamento da comunicação. Tornar possível este planejamento traz à tona a necessidade de definir claramente efeitos e pré-condições associados a cada comunicação possível.

- A comunicação traz consigo um significado semântico. Quando um agente é objeto de uma ação comunicativa recebendo uma mensagem, ele deve ser capaz de compreender o significado desta ação e qual a intenção comunicativa do agente que enviou a mensagem. Por isso a necessidade de uma semântica universal e a necessidade de um padrão de comunicação.

JADE é compatível com o padrão da FIPA definido para resolver a necessidade de comunicação e sociabilidade dos agentes, a Linguagem de Comunicação de Agentes (ACL – *Agent Communication Language*) [FIPA00]. O padrão FIPA-ACL é baseado na teoria dos atos de fala [Russel02] e nessas três características descritas acima.

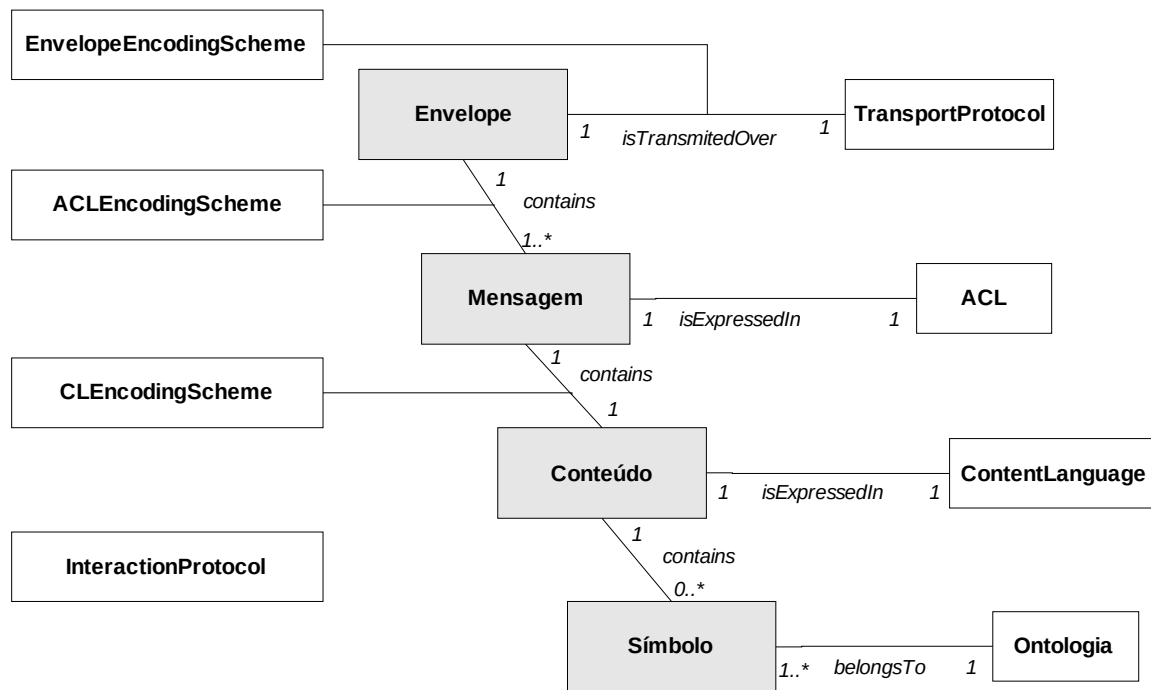


Figura 4.5: Componentes do modelo de comunicação da FIPA

Diversos padrões de comunicação comuns foram definidos pela FIPA e são denominados de protocolos de interação. Esses protocolos podem ser utilizados para conseguir realizar tarefas comuns, como por exemplo delegar uma ação, realizar uma chamada de propostas, entre outras. A Figura 4.5 representa o modelo de comunicação definido pela FIPA e adotado por JADE, como também os relacionamentos existentes entre os elementos que compõe este modelo.

Na comunicação entre agentes na plataforma JADE, os agentes se comunicam enviando mensagens individuais uns para os outros através de passagem assíncrona de mensagens ACL. Os agentes interpretam as mensagens como atos de fala, e não como chamadas, bem como JADE codifica as mensagens trocadas pelos agentes de forma transparente para o desenvolvedor. O padrão FIPA define que somente mensagens ACL são transportadas na plataforma, porém não define nenhum padrão para o conteúdo das mensagens.

A classe *ACLMessage* representa as mensagens seguindo as especificações da FIPA [FIPA00]. Na Figura 4.6 é apresentada uma estrutura de mensagem ACL. Nela, todos os parâmetros são pares representados por uma palavra-chave e um valor. Os valores dos parâmetros podem ser inseridos através dos métodos *set()* e podem ser lidos através de métodos *get()*, como exemplo podemos utilizar os métodos *setContent()* e *getContent()* para inserir ou capturar o valor do parâmetro *content*, respectivamente.

Todas as performativas da FIPA para atos de comunicação são identificadas na mensagem ACL por constantes, por exemplo: *ACCEPT_PROPOSAL*, *AGREE*, *CANCEL*, *CFP*, *CONFIRM*, etc. Essas constantes são referenciadas no construtor da classe com o objetivo de definir o tipo de mensagem.

PERFORMATIVA	{	QUERY-IF
ENDEREÇAMENTO	{	:sender agencia@G6C15:1099/JADE
		:receiver empregador@G6C15:1099/JADE
		:protocol fipa-query
COMUNICAÇÃO	{	:conversation_id C2471722_1067382505429
		:reply_with
		:reply_by
		:language fipa-s10
		:ontology employment-ontology
MENSAGEM	{	:content ((TRABALHA-PARA
		(PESSOA :nome "Maria"
		:idade 23
		:endereco (ENDERECO
		:rua "Rua da Aurora"
		:numero 15
		:cidade Recife))
		(EMPRESA :nome "BOMPREGO"
		:endereco (ENDERECO
		:rua "17 de Agosto"
		:numero 200
		:cidade Recife))))

Figura 4.6: Exemplo de mensagem de acordo com o padrão ACL

Para que um agente envie mensagens a outros, o desenvolvedor deve instanciar um objeto da classe *ACLMessage*, preencher a mensagem com as informações necessárias em cada campo correspondente e chamar o método *send()* da classe *Agent*. Quando o agente quiser receber as mensagens, deve chamar o método *receive()* ou *blockingReceive()* da classe *Agent*. Outra forma de enviar ou receber mensagens em JADE é através do uso das classes de comportamentos próprios para ações comunicativas: *SenderBehaviour* e *ReceiveBehaviour*. O uso dessas duas classes torna possível que as trocas de mensagens possam ser escalonadas como atividades independentes.

A seguir são listados alguns métodos importantes da classe *ACLMessage*:

- *public void addReceiver(AID idAgente)*: adiciona o AID de um agente como receptor ou destinatário da mensagem ACL, determinando quem receberá a mensagem;
- *public void removeReceiver(AID idAgente)*: remove um determinado AID da lista de receptores;
- *public ACLMessage createReply()*: cria uma nova mensagem ACL em resposta à determinada mensagem;
- *public static java.lang.String[] getAllPerformativeNames()*: retorna um array de strings com todas as performativas que podem ser utilizadas em JADE;

- `public Iterator getAllReceiver()`: retorna um *iterator* contendo todos os AIDs dos agentes receptores da mensagem;
- `public java.lang.String getContent()`: retorna uma *string* contendo o conteúdo da mensagem.
- `public void setContent(java.lang.String conteudo)`: define o conteúdo da mensagem a ser enviada.
- `public void setOntology(java.lang.String ontologia)`: define a ontologia que a mensagem respeita;
- `public void setPerformative(int performativa)`: define a constante recebida como parâmetro para ser a performativa da mensagem ACL.

JADE também permite que não só *strings* sejam transmitidas no conteúdo de mensagens, mas também outros objetos Java. O método `setContentObject()` da classe `ACLMessage` permite definir como conteúdo de uma mensagem um objeto Java do tipo `java.io.Serializable`. Este objeto pode ser recuperado no recebimento da mensagem através do método `getContentObject()`. Contudo, esses métodos não estão em conformidade com as especificações da FIPA, uma vez que ela estabelece que todas as trocas de mensagens entre agentes devem estar baseadas em texto.

4.2.5 Protocolos de interação

O padrão FIPA-ACL [FIPA00] em sua visão fundamental especifica que cada mensagem representa um ato de comunicação, representando uma possível ação que o agente pode realizar. Para cada ato de comunicação, a FIPA estabelece um conjunto de pré-condições possíveis e um conjunto de efeitos racionais. Em resumo, são especificadas as condições que precisam ser verdadeiras antes do agente executar uma ação (antes de enviar a mensagem) e o efeito esperado da ação (a razão pela qual a mensagem é enviada). Também é especificado pelo padrão que, mesmo quando o agente transmissor tenha executado o ato (enviado a mensagem), ele não precisa acreditar que o efeito racional necessariamente acontecerá. Por exemplo, dada sua autonomia, o agente receptor pode simplesmente decidir ignorar a mensagem recebida. Isto não é desejável na maioria das aplicações porque aumenta o não-determinismo no ambiente.

Por esta razão, ao invés de unicamente enviar mensagens, é interessante que os agentes utilizem protocolos de interação. A utilização de protocolos de interação, iniciados por um determinado agente, permite que ele possa verificar se o efeito racional esperado para aquele ato de comunicação foi atingido ou não.

A FIPA especifica um conjunto padronizado de protocolos de interação que podem ser usados como modelos para especificar interações entre agentes. Para cada conversa entre os agentes, JADE distingue os papéis de iniciador (*Initiator*) e respondedor (*Responder*). O primeiro representa o agente que inicia a conversa, enquanto que o último representa o agente engajado em uma conversa após ser acionado por algum outro agente. JADE provê classes prontas com o comportamento de ambos os papéis em conversações seguindo a maioria dos protocolos de interação da FIPA. Estas classes podem ser encontradas no pacote `jade.proto`. Elas oferecem um conjunto de métodos de chamada para manipular os estados dos protocolos com uma API homogênea. Além disso, JADE oferece no pacote `jade.proto.states` implementações para alguns estados genéricos cujo emprego pode ser útil para protocolos de interação.

Todos os comportamentos específicos para iniciadores terminam e são removidos da fila de tarefas de agentes tão logo elas alcancem qualquer estado final do protocolo de interação. A fim de permitir o reuso dos objetos de Java para representar estes comportamentos sem precisar recriar novos objetos, todos os iniciadores incluem um número de métodos *reset* com os argumentos apropriados. Além disso, todos os iniciadores, exceto o *FipaRequestInitiatorBehaviour*, são 1:N, ou seja, podem manipular vários respondedores ao mesmo tempo. Os respondedores, ao contrário, são cíclicos e são re-agendados assim que alcançam qualquer estado final do protocolo de interação. Esta característica permite ao desenvolvedor limitar o número máximo de respondedores que o agente pode executar em paralelo, através da adição de sub-comportamentos dentro do método *setup()* do agente.

AchieveRE (*Achieve Rational Effect*)

Os diversos protocolos de interação padronizados pela FIPA, tais como *FIPA-Request*, *FIPA-Query*, *FIPA-Propose*, *FIPA-Request-When*, *FIPA-Recruiting*, *FIPA-Brokering*, permitem ao agente iniciador da conversação verificar o efeito racional esperado do ato de comunicação realizado de acordo com um protocolo. Devido ao compartilhamento de praticamente a mesma estrutura por esses protocolos mais simples, JADE provê as classes *AchieveREInitiator* e *AchieveREResponder*. Estas classes apresentam uma maneira de realizar uma implementação homogênea de todos estes tipos de protocolos de interação.

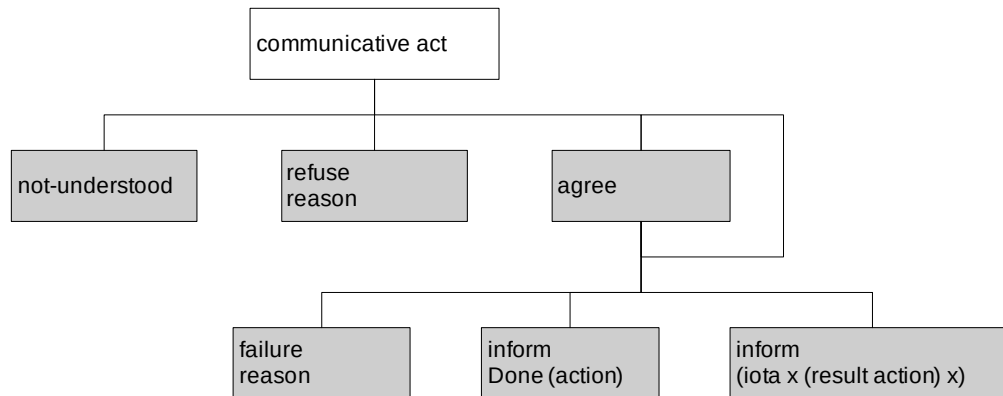


Figura 4.7: Estrutura homogênea dos protocolos de interação em JADE

A Figura 4.7 mostra a estrutura abstrata destes protocolos de interação. O *Initiator* envia uma mensagem, através da realização de um ato comunicativo. O *Responder* pode então responder enviando uma mensagem indicando que não entendeu a mensagem que recebeu (*NOT-UNDERSTOOD*), uma mensagem de recusa (*REFUSE*) para atingir o efeito racional do ato comunicativo, ou também uma mensagem de aceitação (*AGREE*) para comunicar o acordo para executar o ato comunicativo, como mostrado na primeira linha das caixas sombreadas. O respondedor executa a ação e, finalmente, deve responder com um informe (*INFORM*) do resultado da ação que foi realizada ou com uma falha (*FAILURE*) caso algo de errado tenha acontecido. A mensagem de *agree* pode ser omitida e o agente já enviar diretamente mensagens de falha ou de informação que a ação foi realizada com sucesso.

As classes que JADE provê para implementação dos papéis de iniciador respondedor são as seguintes:

- **jade.proto.AchieveREInitiator**: esta classe deve ser utilizada para implementação de um papel de iniciador de conversação utilizando-se protocolos de comunicação. Instâncias desta classe podem ser construídas, passando como argumento do seu construtor a mensagem usada para iniciar o protocolo. É importante que esta mensagem tenha o valor correto no campo da mensagem *ACLMessage* como definido pelas constantes na interface *FIPAProtocolNames*;
 - **jade.proto.SimpleAchieveREInitiator**: esta classe é uma implementação mais simples do papel do iniciador. A principal diferença entre o *AchieveREInitiator* e o *SimpleAchieveREInitiator* é que esta versão do protocolo não permite ao programador registrar comportamentos específicos da aplicação para manipular os estados do protocolo;
- **jade.proto.AchieveREResponder**: esta classe serve para implementação do papel de respondedor. É muito importante passar o modelo de mensagem correto como argumento para o seu construtor, pois ele é usado para selecionar qual mensagem ACL recebida deverá ser tratada. O método *createMessageTemplate* pode ser usado para criar um modelo de mensagem para um determinado protocolo de interação, mas também modelos mais seletivos podem ser úteis em alguns casos. Por exemplo, podemos ter uma instância da classe *MessageTemplate* para cada possível agente transmissor. Esta classe pode facilmente ser estendida através da sobrecarga de um ou mais dos seus métodos (*prepareResponse* ou *prepareResultNotification*), os quais provêem maneiras de manipular os estados do protocolo e, em particular, preparar as mensagens de resposta;
 - **jade.proto.SimpleAchieveREResponder**: esta classe é uma implementação mais simples da classe *AchieveREResponder*. A principal diferença é que esta versão não permite ao programador registrar outros comportamentos como manipuladores dos estados do protocolo.

FIPA-Contract-Net

O protocolo de interação *Contract-Net* é um dos mais conhecidos no paradigma de orientação a agentes. Ele permite ao agente iniciador da conversação enviar uma chamada de propostas (*Call for Proposal – CFP*) para um conjunto de respondedores, avaliar suas propostas e então aceitar a que ele preferir (ou mesmo rejeitar todas elas). O protocolo de interação *Contract-Net* é profundamente descrito nas especificações da FIPA [FIPA04]. A Figura 4.8 apresenta uma visão abstrata e simples deste protocolo.

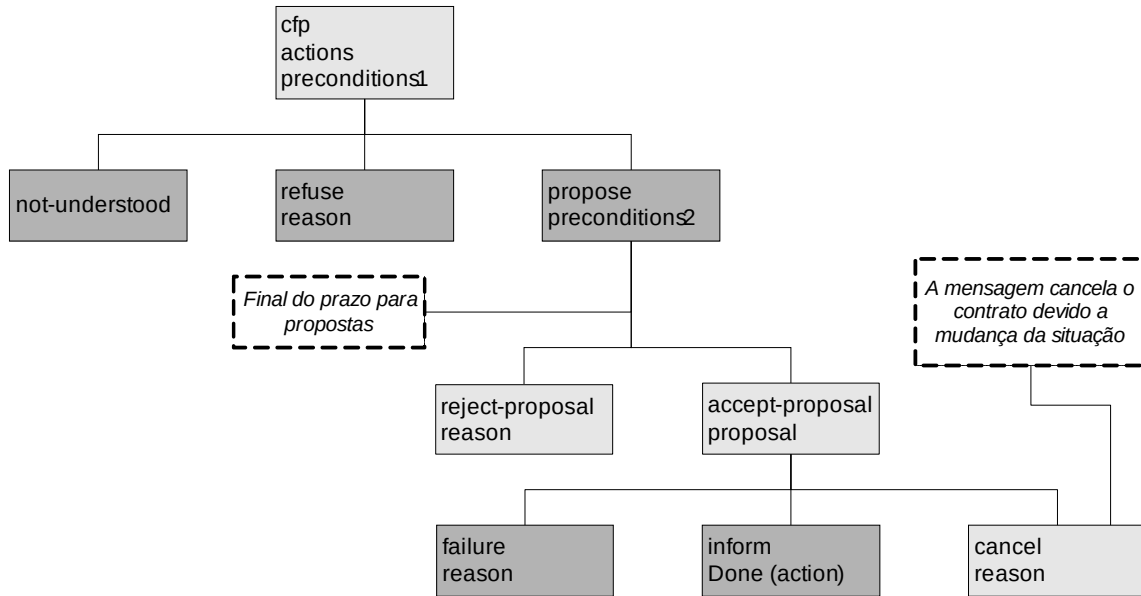


Figura 4.8: Protocolo de interação FIPA-Contract-Net

O iniciador solicita propostas de outros agentes enviando uma mensagem *CFP* que especifica a ação a ser executada e, se necessário, condições em torno da sua execução. Os agentes respondedores podem então responder enviando uma mensagem *PROPOSE* incluindo as pré-condições que eles configuraram para a ação, por exemplo o custo ou o tempo da sua proposta. Alternativamente, os agentes respondedores podem enviar uma mensagem *REFUSE* para recusar a proposta ou, eventualmente, uma mensagem *NOT-UNDERSTOOD* para comunicar problemas de comunicação.

O Iniciador pode então avaliar todas as propostas recebidas e fazer sua escolha de quais propostas de agentes serão aceitas e quais serão rejeitadas. Uma vez que os respondedores que tiveram suas propostas aceitas (através do recebimento de uma mensagem do tipo *ACCEPT-PROPOSAL*) tenham completado suas tarefas, eles podem, finalmente, responder com um *INFORM* indicando o resultado da realização das ações da proposta aceita, ou com uma *FAILURE* se alguma procedimento deu errado.

As classes oferecidas por JADE para implementação do protocolo FIPA-Contract-Net são:

- **jade.proto.ContractNetInitiator**: este comportamento implementa o protocolo de interação fipa-contract-net do ponto de vista do agente iniciando o protocolo, que é o agente que envia a mensagem de CFP;
- **jade.proto.ContractNetResponder**: esta classe de comportamento implementa o protocolo de interação FIPA-Contract-Net do ponto de vista do respondedor de uma mensagem de chamada de propostas. É muito importante passar o modelo de mensagem correto como argumento do seu construtor.

4.2.6 Ontologias em JADE

Quando um agente X se comunica com um outro agente Y, determinada quantidade de informação I é transferida de X para Y através de uma mensagem ACL. Dentro da mensagem ACL, I é representada

como uma expressão de conteúdo, consistente com uma linguagem de conteúdo apropriada, por exemplo a FIPA-SL [FIPA04], e codificada em um formato apropriado, tal como *string*. Tanto X quando Y possuem sua própria forma (possivelmente diferente) de representar I internamente.

Levando-se em consideração que a maneira como o agente representa internamente um pedaço de informação deve permitir uma fácil manipulação desta informação, é bastante claro que a representação usada numa expressão de conteúdo não é adequada para o interior do agente. Por exemplo, a informação que existe uma pessoa cujo nome é Maria e que possui 23 anos na expressão de conteúdo ACL pode ser representada por: (Pessoa :nome Maria :idade 23).

Armazenar esta informação dentro de um agente simplesmente como uma variável do tipo *string* não é uma forma adequada para manipular essa informação, por exemplo, para capturar a idade de Maria, todas as vezes seria necessário realizar uma verificação semântica na *string*. Entretanto, considerando agentes de software escritos em Java, a informação pode ser convenientemente representada dentro do agente como objetos Java. Por exemplo, representar a informação acima sobre Maria como uma instância de uma classe de aplicação específica, inicializada com o valor "Maria" para o atributo *nome* e o valor 23 para o atributo *idade*, pode facilitar a manipulação dessa informação.

```
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 a) {idade = a; }
    ...
}
```

Mesmo assim, é claro que se por um lado a manipulação da informação torna-se mais fácil, por outro lado, cada vez que o agente X enviar um pedaço de informação I para o agente Y, têm-se dois problemas, um de conversão e outro de checagem. Primeiramente, X precisaria converter sua representação interna de I para a representação da expressão de conteúdo ACL correspondente, e Y precisaria realizar a conversão oposta. Além disso, Y também deveria realizar checagem semântica para verificar se I é um pedaço de informação significativa, ou seja, que está compatível com as regras da ontologia através da qual X e Y atribuíram um significado apropriado para I.

O *framework* JADE provê um suporte para linguagens de conteúdo e ontologias projetado de forma a realizar automaticamente as conversões e checagem das operações citadas acima, conforme apresentado na Figura 4.9. Desta forma, JADE permite que os desenvolvedores manipulem informações dentro de seus agentes como objetos de Java, sem a necessidade de qualquer esforço extra.

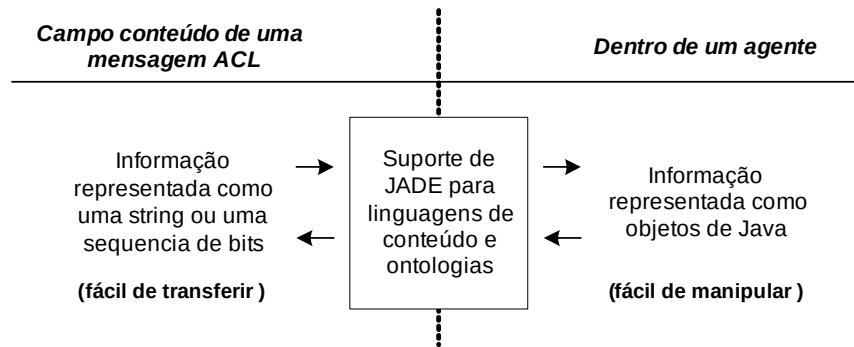


Figura 4.9: Conversão realizada pelo suporte de JADE para linguagens de conteúdo e ontologias

O Modelo de Referência de Conteúdo

Um dos ganhos obtidos quando utilizamos a infra-estrutura para construção de ontologias de JADE é a possibilidade de realizar checagens semânticas adequadas em uma determinada expressão de conteúdo. Essas checagens são possíveis porque JADE classifica todos os elementos do domínio de discurso (elementos que podem aparecer dentro de uma sentença válida enviada por um agente como conteúdo de uma mensagem ACL) de acordo com características semânticas genéricas. Essa classificação é derivada da definição da FIPA para estrutura de mensagens ACL [FIPA00], que requer que o conteúdo de cada mensagem ACL possua semântica própria de acordo com a performativa que está associada à mensagem.

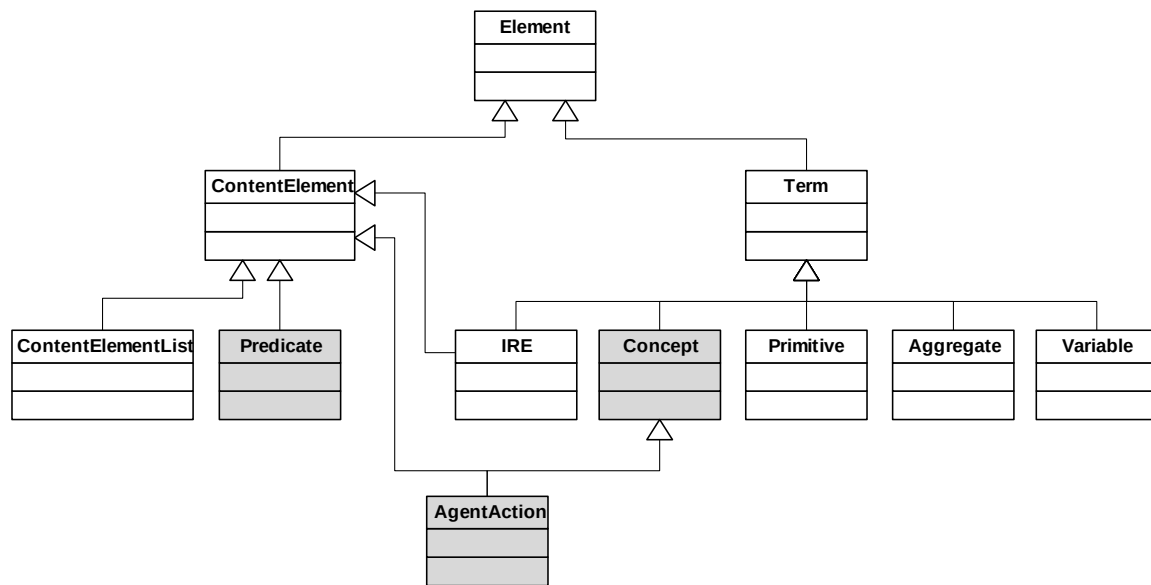


Figura 4.10: O modelo de referência de conteúdo de JADE

A Figura 4.10 apresenta o modelo de referência de conteúdo de JADE para definição de elementos de uma ontologia. Uma linguagem de conteúdo completamente expressiva deve ser capaz de representar e distinguir entre todos esses tipos de elementos. A seguir são descritos os elementos que compõe este modelo:

- **Predicados** (classe *Predicate*) são expressões que dizem algo a respeito do estado do mundo, podendo assumir um valor verdadeiro ou falso. Por exemplo: `(Trabalha-para (Pessoa :nome Maria) (Empresa :nome Bompreço))` está questionando se “a pessoa Maria trabalha para empresa Bompreço”. Os predicados podem ser usados significativamente como conteúdo de mensagens com as performativas `INFORM` ou `QUERY-IF`, enquanto que não faz muito sentido se for utilizado como conteúdo de uma mensagem `REQUEST`.
- **Termos** (*Term*) são expressões que identificam entidades (abstratas ou concretas) que “existem” no mundo e que os agentes podem conversar e raciocinar sobre elas. Os termos são classificados em mais detalhes como conceitos, ações, primitivas, IREs, agregados e variáveis.
- **Conceitos** (classe *Concept*) são expressões que indicam entidades com uma estrutura complexa que pode ser definida em termos de *slots*. Como exemplo temos a estrutura *pessoa*, que pode ser definida com os *slots* *nome* e *idade*: `(Pessoa :nome Paulo :idade 55)`. Geralmente, não faz sentido usar conceitos diretamente como conteúdo de uma mensagem ACL. Em geral eles são referenciados dentro de predicados ou outros conceitos, como por exemplo em `(Livro :título "O Alquimista" :author (Pessoa :nome Paulo))`.
- **Ações** de agentes (classe *AgentAction*) são conceitos especiais que indicam ações que podem ser realizadas por determinados agentes, por exemplo a ação de vender um livro para uma pessoa pode ser representada da seguinte forma: `(Venda (Livro :título "O Alquimista") (Pessoa :nome Maria))`. É útil tratar ações de agentes separadamente uma vez que ao invés de conceitos “normais”, elas têm conteúdos significativos para certos tipos de mensagens ACL, tais como aquelas com performativa do tipo `REQUEST`. Atos de comunicação (ou seja, mensagens ACL) são por si só ações de agentes.
- **Primitivas** (*Primitive*) são expressões que indicam entidades atômicas, tais como *strings* e inteiros. Primitivas podem ser expressões agregadas, expressões referenciais de identificação ou expressões variáveis.
- **Agregados** (*Aggregate*) são expressões que indicam entidades que são agrupamentos de outras entidades, por exemplo `(sequence (Pessoa :nome Maria) (Pessoa :nome Paulo))`.
- **Expressões Referenciais de Identificação (IRE)** são expressões que identificam a entidade (ou entidades) para as quais um determinado predicado é verdadeiro. Por exemplo `(all ?x (Trabalha-para ?x (Empresa :nome Bompreço)))` identifica todos os elementos “x” para os quais o predicado `(Trabalha-para x (Empresa :nome Bompreço))` é verdadeiro, isto é, todas as pessoas que trabalham para empresa Bompreço. Essas expressões geralmente são usadas em consultas (como por exemplo conteúdos de mensagens `QUERY_REF` e requerem o uso de variáveis).
- **Variáveis** (*Variable*) são expressões geralmente utilizadas em consultas, que indicam um elemento genérico ainda desconhecido.

O modelo de referência de conteúdo de JADE ainda inclui mais dois tipos de elementos: *ContentElement* e *ContentElementList*. Esses elementos são introduzidos considerando-se que apenas predicados, ações de agentes, IREs e listas de elementos desses três tipos (*ContentElementList*) são conteúdo significativo de pelo menos uma mensagem ACL. Todos eles são especializações de *ContentElement*.

Estrutura de uma ontologia em JADE

Uma ontologia em JADE é uma instância da classe *jade.content.onto.Ontology* para a qual um conjunto de esquemas é adicionado com o objetivo de definir a estrutura dos predicados, ações de agentes e conceitos (basicamente seus nomes e seus campos) que são pertinentes ao domínio que está sendo

modelado através da ontologia. Os esquemas são instâncias das classes *PredicateSchema*, *AgentActionSchema* e *ConceptSchema*, que estão incluídas no pacote *jade.content.schema*. Essas classes possuem métodos com os quais se torna possível declarar os campos que definem a estrutura de cada tipo de predicado, ação ou conceito.

Como uma ontologia é basicamente uma coleção de esquemas que tipicamente não evolui durante o ciclo de vida do agente, é uma boa prática declarar uma ontologia com um objeto *singleton* e definir uma classe (que estenda *jade.content.onto.Ontology*) com um método estático para acessar esse objeto *singleton* [Gamma95]. Desta forma, é possível compartilhar, entre os diferentes agentes contidos em uma mesma JVM, o mesmo objeto que define a ontologia e, conseqüentemente todos os esquemas que estão incluídos nele.

Cada esquema adicionado na ontologia é associado à uma classe Java, por exemplo, o esquema do conceito CD é associado à classe *CD.java*. Enquanto usarem a ontologia definida, expressões indicando CDs serão instanciadas da classe *CD*. Cada campo de um esquema tem um nome e um tipo, isto é, valores para este campo devem estar compatíveis com o esquema dado. Um campo pode ser declarado como opcional (*OPTIONAL*), indicando que o seu valor pode ser *null*, ou mandatório (*MANDATORY*). Se um valor nulo for encontrado para um campo mandatório durante a validação de uma expressão de conteúdo, é lançada uma exceção do tipo *OntologyException*. Além disso, um campo pode ter cardinalidade maior que um (>1), ou seja, valores para aquele campo são agregados de elementos de um determinado tipo. Por exemplo, o campo *tracks* dentro do esquema do conceito CD pode conter 1 ou mais elementos do tipo *Track*.

Um esquema pode conter outros super esquemas. Isso permite definir relacionamentos de especialização ou extensão entre conceitos. Por exemplo, o esquema de CD tem o esquema de Item como um super tipo, significando que CD é um tipo de Item e portanto cada instância de CD é também uma instância de Item.

Gerenciamento do conteúdo

A conversão e checagem de operações é realizada por um objeto gerenciador de conteúdo, que é uma instância da classe *jade.content.ContentManager*. Cada agente JADE tem embutido um gerenciador de conteúdo que está acessível através do método *getContentManager()* da classe *Agent*. A classe *ContentManager* provê todos os métodos para transformar objetos Java em *strings* (ou seqüências de *bytes*) e para inseri-los no campo de conteúdo de instâncias de *ACLMessage* e vice-versa.

O gerenciador de conteúdo provê interfaces convenientes para acessar a funcionalidade de conversão, contudo, na versão de JADE que utilizamos, apenas delega a conversão e checa as operações para uma ontologia (uma instância da classe *jade.content.onto.Ontology*) e para um codificador de linguagem de conteúdo (uma instância da interface *jade.content.lang.Codec*). De forma mais específica, a ontologia valida a informação a ser convertida de um ponto de vista semântico, enquanto que o codificador realiza a tradução para *strings* (ou seqüência de *bytes*) de acordo com as regras sintáticas da linguagem de conteúdo relacionada. A não ser que seja necessário utilizar características avançadas sobre ontologias, não é necessário se aprofundar nos detalhes do uso de codificadores.

Outras Características

Além da estrutura básica de JADE para tratar com ontologias, alguns aspectos avançados também são considerados. É possível realizar, por exemplo, a combinação de ontologias, a criação de um padrão de interface para vocabulários, o uso de operadores para linguagens de conteúdo, a criação de consultas sobre a ontologia, entre outras coisas.

Com relação ao apoio ferramental para definição de ontologias, podemos utilizar a ferramenta Protegé [Noy2001] de maneira muito prática. Protegé foi desenvolvida como uma ferramenta de código livre, permitindo assim que possa ser realizada sua customização de modo a estar compatível com outras ferramentas para criação de ontologias, bem como a geração automática de código. Isso é possível porque podemos definir em Protegé uma ontologia em um formato abstrato que pode então ser convertido para linguagens específicas, através de um plug-ins. Atualmente, Protegé é a única ferramenta através da qual podemos gerar o código de ontologias em JADE automaticamente, usando como formato abstrato na ferramenta o modelo de ontologias de JADE.

4.2.7 Ferramentas de Gerenciamento da Plataforma JADE

O *framework* JADE garante um padrão de interoperabilidade entre SMAs através de um abrangente conjunto de agentes de serviços de sistema disponível através da plataforma. Estes agentes de serviços são ferramentas de suporte ao desenvolvedor, os quais tanto facilitam como possibilitam a comunicação entre agentes de acordo com as especificações da FIPA (*Foundation for Intelligent Physical Agents*) [FIPA04].

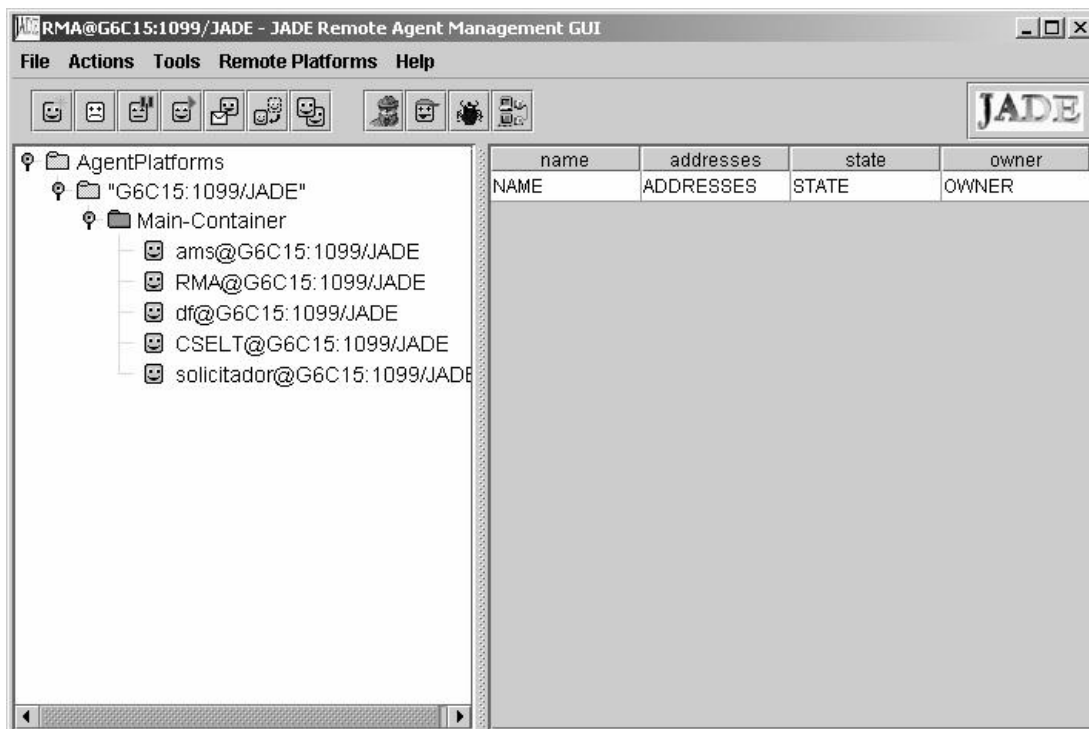


Figura 4.11: Tela da ferramenta Remote Monitoring Agent

Os agentes de serviços de sistema lidam com os aspectos que não fazem parte do agente em si e que são independentes das aplicações, tais como: transporte de mensagens, codificação e interpretação de mensagens, ciclo de vida dos agentes, páginas brancas e páginas amarelas. As principais ferramentas que JADE disponibiliza para os desenvolvedores de agentes podem ser encontradas no pacote *jade.tools*, são elas: RMA, *Sniffer Agent*, *Introspector Agent*, *Dummy Agent*, e DF GUI.

A ferramenta *Remote Monitoring Agent* (RMA), como apresentado na Figura 4.11, funciona como uma console gráfica para gerenciamento e controle da plataforma JADE. Ela é usada para monitorar e administrar o *status* de todos os componentes da plataforma distribuída, incluindo agentes e *containers*. O RMA é um objeto Java, instanciado da classe *jade.tools.rma.rma*, que pode ser carregado através das seguintes linhas de comando: *java jade.Boot -gui* ou *java jade.Boot meuComputador:jade.tools.rma.rma*. Com o RMA podemos controlar o ciclo de vida dos agentes (inclusive os que estão distribuídos), conexões com plataformas remotas e a instalação de protocolos de transporte de mensagens. Além disso, o RMA serve como um controle principal onde estão centralizadas algumas funcionalidades como, por exemplo, a inicialização das demais ferramentas de JADE.

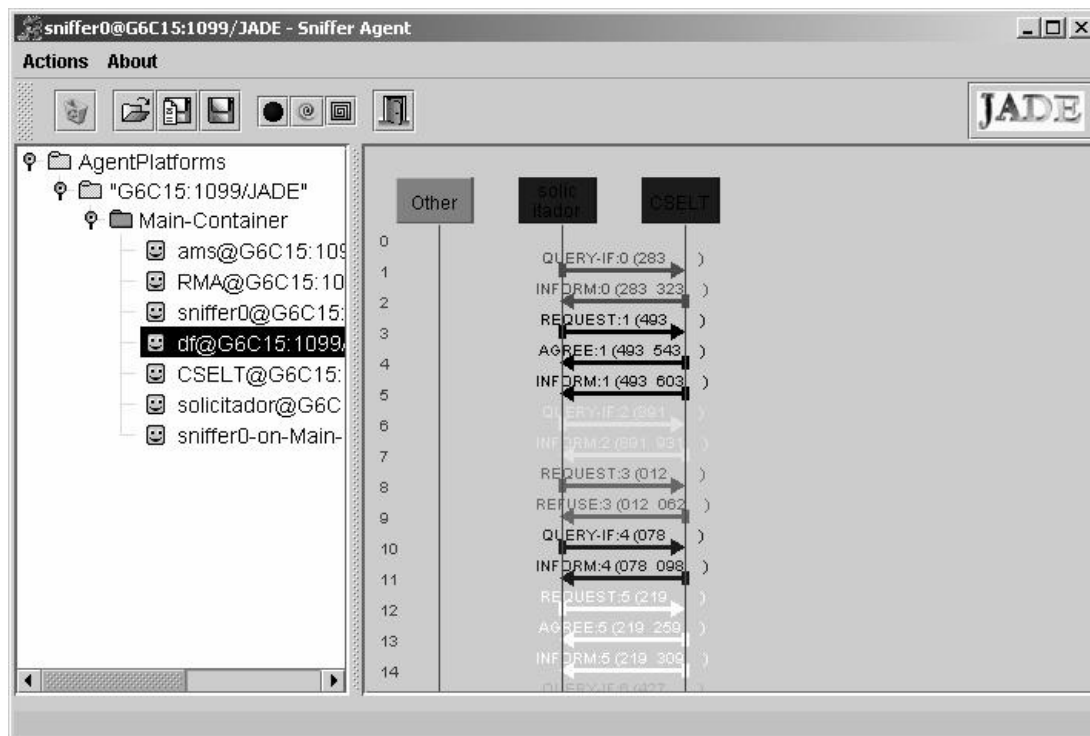


Figura 4.12: Tela da ferramenta *Sniffer Agent*

A Figura 4.12 apresenta a tela da ferramenta *Sniffer Agent*. Ela pode ser utilizada para interceptar as mensagens ACL e exibir a conversação através de uma notação similar ao diagrama de sequência da UML. Esta ferramenta é extremamente útil para o rastreamento e a depuração de conversações entre agentes, permitindo também que as mensagens trocadas pelos agentes sejam salvas ou carregadas a partir de arquivos.

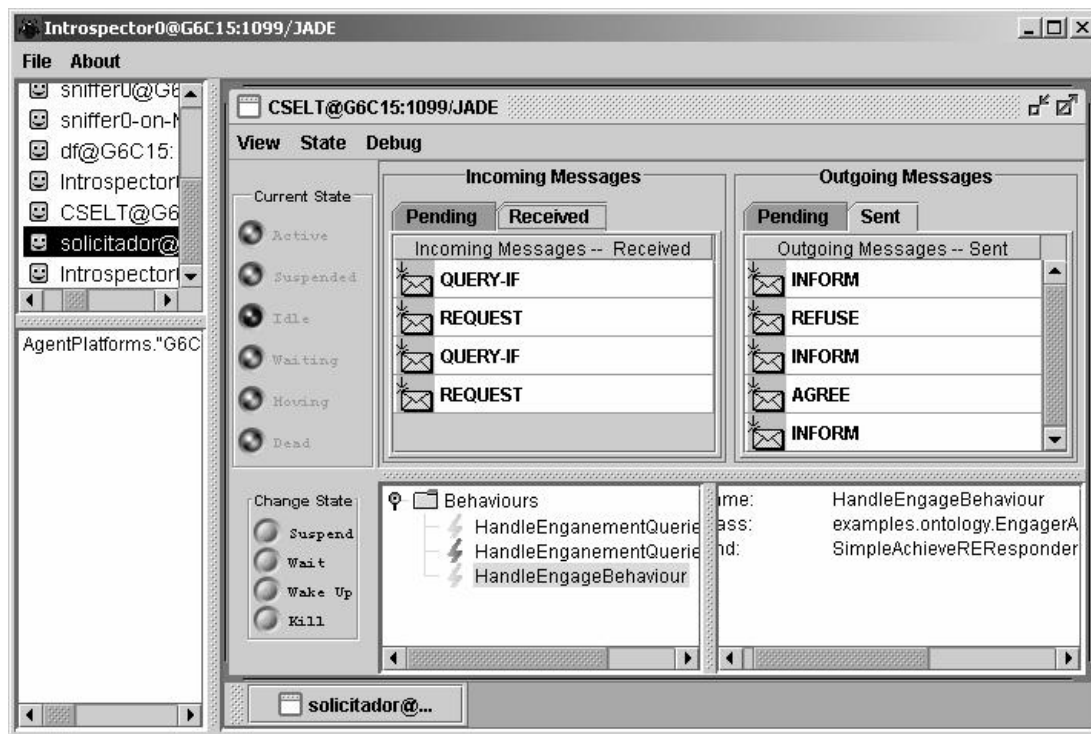
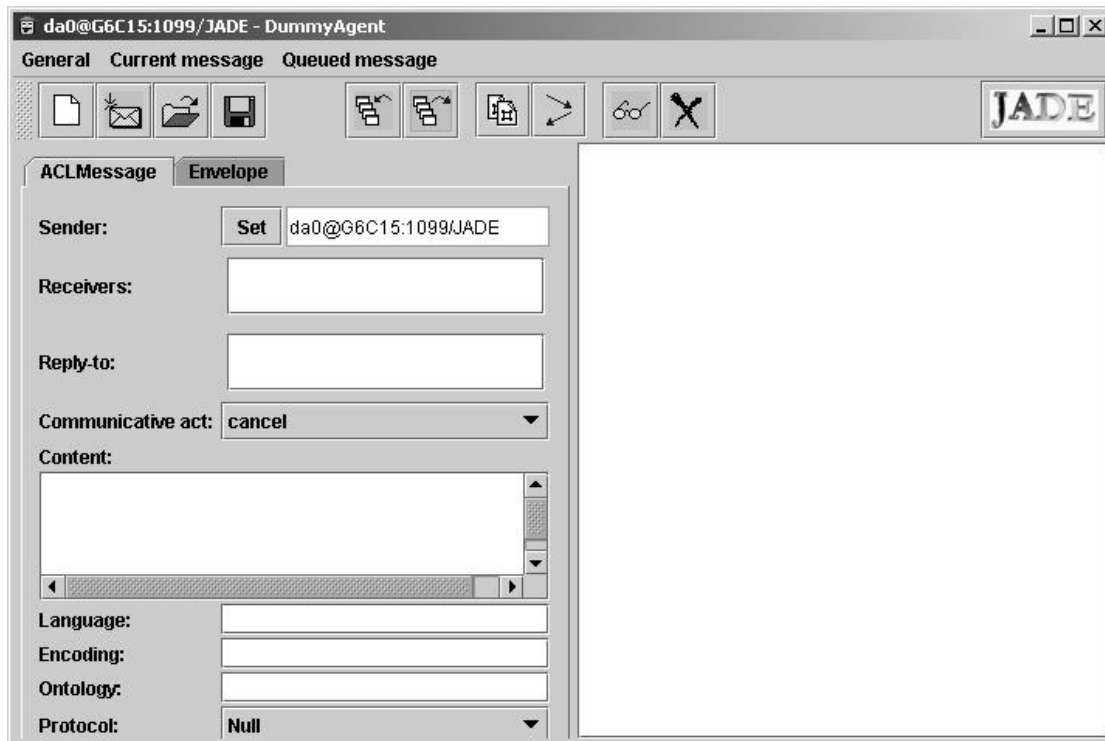


Figura 4.13: Tela da ferramenta *Introspector Agent*

O Introspector Agent (Figura 4.13) é uma ferramenta que pode ser utilizada para monitorar vários agentes ao mesmo tempo, sendo possível observar o seu ciclo de vida, suas mensagens ACL trocadas e seus comportamentos em execução.

Para compor e enviar mensagens ACL para outros agentes, bem como para exibir o conteúdo de todos os campos da mensagem recebida, podemos utilizar a ferramenta *Dummy Agent*. Esse agente de serviço, apresentado na Figura 4.14, possui também a função de armazenar mensagens ACL.

Por fim temos o DF GUI, uma ferramenta que provê o serviço de páginas amarelas da plataforma, funcionando como uma interface para o facilitador de diretórios e representando o FIPA-DF. O DF GUI permite que sejam realizadas as operações de registro, cancelamento de registro, alteração de registro e busca por agentes e serviços. Através dela também podemos criar confederações de facilitadores de diretório e realizar propagação de busca através de domínios e sub-domínios.

Figura 4.14: Tela da ferramenta *Dummy Agent*

4.3 Considerações Finais

Este capítulo apresentou a estrutura geral da plataforma de desenvolvimento de agentes JADE. Dentre os ambientes para implementação de agentes descritos no capítulo 2, nós escolhemos JADE como uma plataforma adequada para implementação de SMA desenvolvidos usando a metodologia Tropos. A seguir apresentamos critérios e as características de JADE utilizados para esta escolha:

- JADE é distribuída completamente como software livre, sem custos e sem data de expiração;
- A linguagem de programação suportada por JADE é Java, que é portátil e possui versões dedicadas a configurações de hardware mínimas [Deitel03];
- As especificações de JADE são compatíveis com os padrões da FIPA [FIPA04]. JADE oferece uma biblioteca de classes de protocolos de interação padronizados pela FIPA, prontas para serem instanciadas ou estendidas;
- Não há necessidade de implementar a plataforma de agentes, as funcionalidades e a ontologia de gerenciamento de agentes, nem os mecanismos de transporte e *parsing* de mensagens;
- JADE oferece suporte a usuários, tais como: uma lista de problemas relatados, documentação para programadores e administradores, fóruns e lista de discussão ativos;
- JADE oferece um transporte eficiente de mensagens FIPA-ACL [FIPA00] entre agentes.

Capítulo

5 Do Projeto Detalhado em Tropos à Implementação com JADE

Este capítulo apresenta a proposta de adaptação da fase de projeto detalhado da metodologia Tropos para auxiliar o desenvolvimento de SMA implementados utilizando a plataforma JADE.

5.1 Introdução

Um agente pode ser útil como uma entidade *stand-alone* que delega tarefas particulares de interesse de um usuário, por exemplo: um assistente digital pessoal ou um filtro de emails. Entretanto, na grande maioria dos casos, agentes existem em ambientes que contém outros agentes, tais ambiente podem ser considerados sistemas multi-agentes.

Em sistemas orientados a agentes, o comportamento global dos agentes deriva da interação entre os agentes que constituem a sociedade: eles cooperam, coordenam ou negociam uns com os outros [Silva04]. Um SMA então pode ser visto como uma sociedade de componentes de software (agentes) autônomos, colaborativos e dirigidos a objetivos, bem como uma organização social. Cada papel que um agente pode executar tem o seu conjunto bem definido de responsabilidades, atingidas por meio das habilidades dos agentes e por suas capacidades de interação com outros agentes.

Arquiteturas de sistemas multi-agentes sugerem sistemas mais robustos, flexíveis e modulares do que sistemas tradicionais que são simplesmente orientados a objetos. Eles tendem a ser mais abertos e dinâmicos no sentido que consideram estar situados em um ambiente organizacional mutante, ou seja, um ambiente operacional onde novos componentes podem ser adicionados, modificados ou removidos a qualquer momento.

Modelar sistemas orientados a agentes é uma tarefa que exige que sejam levados em consideração alguns aspectos relacionados à sociabilidade de um sistema multi-agentes:

- Objetivar um bom casamento entre as arquiteturas de um sistema e seu ambiente organizacional e operacional;
- Considerar que a autonomia de um agente deve refletir a natureza social e descentralizada de sistemas empresariais modernos, que são operados por diferentes *stakeholders*;
- Compatibilizar a maneira como os agentes operam flexivelmente para atingir seus objetivos com as situações dinâmicas e não-determinísticas nas quais os softwares de negócios são esperados para lidar.

A UML como linguagem de modelagem de sistemas pode ser adaptada para que possa dar cobertura a novos domínios de aplicação ou novos paradigmas de desenvolvimento de software, pois, embora esta linguagem contemple grande parte das necessidades de modelagem de sistemas, não atende a

todas elas. Os mecanismos que permitem essa adaptação são denominados mecanismos de extensão da UML [Lima05]. Há três tipos de mecanismos de extensão: estereótipos, valores etiquetados e restrições. Um estereótipo é um modelo definido dentro do próprio modelo. A UML possui vários estereótipos já definidos, que fazem parte de sua especificação formal e ajudam a compor o seu padrão de modelagem. Entretanto, esses elementos não são suficientes para modelar de maneira mais explícita o nosso projeto detalhado.

Um estereótipo pode estar baseado em qualquer elemento da UML, representando uma subclasse desse elemento (herdando os mesmos atributos e relacionamentos), mas com diferentes funcionalidades que podem ser inseridas através de restrições extras ou valores etiquetados que agreguem mais informações ao elemento que foi estereotipado. Os estereótipos são mostrados no diagrama na forma de texto antecedido e sucedido pelo símbolo de aspa angular dupla (<< >>) perto do símbolo do elemento base do modelo, mas podem possuir também um ícone particular ou ter sua aparência modificada sendo associado à uma imagem. Exemplos de estereótipos padrão da UML são: <<interface>>, que é baseado no elemento *class* e <<process>>, baseado no elemento *activity*.

Etiqueta é o nome de alguma propriedade que contém um valor especificado para um determinado elemento [Booch99]. Os valores etiquetados podem ser especificados usando tipo de dados padrão da UML para definir seus próprios tipo de dados e servem para guardar uma peça de informação sobre um elemento.

Atualmente, Tropos possui para fase de projeto detalhado um conjunto de estereótipos, etiquetas (*tagged values*) e restrições definidos para acomodar seus conceitos dentro da UML [Mylopoulos01]. Estas extensões podem ser usadas na modelagem de diagramas de classes e demais diagramas, dentre elas podemos citar <<i* actor>> e <<i* dependency>>, para representar, respectivamente, atores e dependências do *framework i**.

Alguns diagramas da UML e da AUML [Bauer01] têm sido utilizados na fase de projeto detalhado de Tropos. Podemos projetar agentes de software através de diagramas de agentes, comportamentos através de diagramas de estado e a dinâmica da comunicação entre agentes usando-se diagramas de sequência. O que diferencia em grande parte a proposta que estaremos apresentando neste capítulo, da forma como Tropos tem dado suporte ao projeto detalhado de agentes, é o fato de a conceitualização utilizada, mesmo visando a implementação em JADE, é mais abstrata do que a atualmente oferecida por Tropos nesta etapa. Consequentemente, a metodologia Tropos ganha a possibilidade de tratar arquiteturas de agentes mais genéricas em nível de projeto detalhado.

5.2 O Projeto Detalhado para JADE

Como visto no Capítulo 3, a fase de projeto detalhado de Tropos está preocupada em introduzir detalhes adicionais para cada componente arquitetural do sistema, principalmente no que diz respeito às peculiaridades dos agentes que envolvem a comunicação e seus comportamentos. O projeto detalhado de Tropos deve ser feito recebendo como entrada as especificações resultantes da fase de projeto arquitetural da metodologia, de maneira que os elementos modelados nesta fase possam ser rastreados por todas as fases antecedentes [Bresciani04].

Além disso, geralmente existe uma forte dependência entre a modelagem do projeto detalhado e determinadas características da linguagem de programação de agentes e plataforma de desenvolvimento adotada [Bresciani04], o que relaciona estritamente o projeto detalhado com as escolhas de implementação. Por exemplo, para implementar SMA em JADE devemos estar preocupados em quatro aspectos principais: agentes, comportamentos de agentes, mensagens ACL e ontologias; enquanto que a implementação do SMA em um modelo BDI faz com que nos preocupemos com agentes, crenças (*beliefs*), desejos (*desires*) e intenções (*intentions*).

Desenvolvemos nossa proposta a partir do trabalho apresentado em [Kolp05], que sugere um *framework* conceitual para modelar e investigar padrões sociais, baseado em cinco dimensões complementares de modelagem de Tropos. Contudo, o foco do nosso trabalho é bem diferente deste, que está relacionado apenas a arquiteturas do tipo BDI e a agentes implementados em JACK. A Tabela 5.1 resume o relacionamento entre essas dimensões e suas representações.

Tabela 5.1: Dimensões de modelagem de padrões de projeto em Tropos

Dimensão	Pergunta	Representação	Notação
Social	“Quais os agentes, suas necessidades e dependências?”	Modelo SD do i*	i*
Intencional	“O que cada serviço faz?”	Tabelas	-
Estrutural	“Como cada serviço é operacionalizado?”	Diagrama de classes adaptado	UML
Comunicativa	“Como é realizada a comunicação entre os agentes?”	Diagrama de sequência adaptado	UML
Dinâmica	“Como são representadas as capacidades de um agente?”	Diagrama de atividades adaptado	UML

Cada dimensão reflete um aspecto particular da arquitetura de um SMA. A dimensão social identifica os agentes relevantes e suas interdependências intencionais. Em seguida, a dimensão intencional identifica e formaliza serviços providos por agentes para realizar as intenções identificadas pela dimensão social, independentemente dos planos que implementarão esses serviços. Esta dimensão responde à pergunta: “O que cada serviço faz?”. A primeira dimensão utiliza os conceitos do *framework* i*, enquanto que a dimensão intencional apenas relaciona em tabelas as descrições dos serviços que correspondem aos planos dos agentes.

A dimensão estrutural está preocupada com a operacionalização dos serviços identificados pela dimensão intencional em termos de conceitos orientados a agentes BDI, tais como *beliefs*, *events*, *plans* e seus relacionamentos. Esta dimensão responde à pergunta: “como cada serviço é operacionalizado?”. A dimensão comunicativa modela as trocas temporárias de eventos entre os agentes, enquanto que a dimensão dinâmica modela os mecanismos de sincronização entre eventos e planos. As dimensões social e intencional são específicas de SMAs. As três últimas dimensões da arquitetura, também são relevantes para sistemas tradicionais (não orientados a agentes), mas elas foram adaptadas e entendidas para os conceitos da orientação a agentes [Kolp05].

Sugerimos neste trabalho adaptações na fase de projeto detalhado de Tropos para incluir alguns modelos novos para dar suporte à implementação de sistemas utilizando a plataforma JADE. O desenvolvedor, a partir dos modelos especificados no projeto detalhado, poderá implementar mais facilmente as estruturas que forem pré-definidas. Dentre as cinco dimensões apresentadas por [Kolp05], estaremos trabalhando com as três últimas, relativas à modelagem de características estruturais, comunicativas e dinâmicas. Em nossa proposta, cada uma das dimensões apresentará um modelo básico e outro auxiliar para serem utilizados no projeto detalhado de sistemas orientados a agentes, totalizando a adoção de seis diagramas de projeto.

Nosso modelo está direcionado para implementação de agentes na plataforma de desenvolvimento JADE e expande o conjunto de extensões da UML através do uso de estereótipos para algumas classes base da UML. A Tabela 5.2 apresenta a listagem dos estereótipos e as suas descrições. Eles serão utilizados em alguns dos diagramas dos modelos que apresentaremos nas seções seguintes.

Tabela 5.2: Estereótipos sugeridos para o projeto detalhado para JADE

Estereótipo	Descrição	Classe Base da UML
<<agent>>	Identifica um agente a ser modelado como extensão da classe <i>Agent</i>	Class
<<behaviour>>	Identifica um comportamento para ser modelado como extensão da classe <i>Behaviour</i>	Class
<<message>>	Identifica uma mensagem que pode ser modelada com um a instância de <i>MessageTemplate</i> ou uma <i>ACLMessage</i>	Class
<<ontology>>	Identifica uma ontologia que deve ser definida estendendo a classe <i>Ontology</i>	Class
<<concept>>	Indica um conceito que será representado na ontologia, a partir da classe <i>Concept</i>	Class
<<predicate>>	Indica um predicado que será representado na ontologia, a partir da classe <i>Predicate</i>	Class
<<action>>	Identifica uma ação que será representada na ontologia a partir da classe <i>AgentAction</i>	Class
<<has>>	Associa um agente a um comportamento, indicando que ele pode executar comportamentos daquele tipo	Association
<<communicates>>	Associa um agente a outro agente, indicando que eles precisam se comunicar para realizar tarefas	Association
<<understands>>	Associa um agente a uma ontologia, indicando que o agente compreende a estrutura da ontologia e pode interpretá-la	Association

5.2.1 Projeto da dimensão estrutural

A *dimensão estrutural* está preocupada com o detalhamento dos atores que foram modelados nas fases anteriores de desenvolvimento de Tropos. Esta dimensão é modelada utilizando diagramas de classe da UML [Booch99, Lima05] estendidos para considerar as estruturas de programação suportadas por JADE. O projeto da dimensão estrutural do sistema está dividido em dois modelos: o modelo de relacionamentos e o modelo de ontologias.

Através do primeiro modelo podemos representar os conceitos e relacionamentos fundamentais para construção da dimensão estrutural dos agentes. Nele podemos visualizar os relacionamentos entre agentes, comportamentos, mensagens e ontologias. Cada um desses conceitos define um *template* para classes de um SMA a ser implementado com JADE. Por sua vez, o modelo de ontologias para o projeto da dimensão estrutural foi proposto para auxiliar a modelagem da comunicação realizada através das interações entre os agentes. A principal intenção deste modelo é projetar uma ou mais ontologias a serem utilizadas dentro dos parâmetros das mensagens trocadas pelos agentes em suas interações, visando padronizar os vocabulários dos domínios de problema que os agentes tratam em seus ambientes de execução.

Modelo de Relacionamentos

Os modelos de relacionamentos devem ser construídos a partir de diagramas de classes da UML, onde poderemos representar as classes utilizando os seguintes estereótipos: `<<agent>>`, `<<behaviour>>`, `<<message>>` e `<<ontology>>`. Cada uma das classe estereotipadas deve ser implementada como uma das entidades seguintes suportadas pela plataforma JADE, respectivamente: *Agent*, *Behaviour*, *ACLMessage* ou *Ontology*.

Além dessas representações, podemos utilizar os seguintes estereótipos para relacionar estas classes, conforme descrição na Tabela 5.2: `<<has>`, `<<communicates>>` e `<<understands>>`. Esses estereótipos indicam relacionamentos de agentes com outros agentes, comportamentos ou ontologias.

Para comportamentos que aparecem listados dentro da classe do agente, a implementação deste deve ser realizada como classes declaradas dentro da classe do agente. Ao contrário, se o comportamento está associado ao agente através de uma ligação com o estereótipo `<<has>>`, o agente possui a classe de comportamento declarada dentro de sua classe.

Modelo de Ontologias

Visto que é interessante e útil utilizar ontologias para facilitar a comunicação entre os agentes quando estão trocando mensagens, sugerimos a inclusão do modelo auxiliar de ontologias no projeto da dimensão estrutural de sistemas multi-agentes. O principal intuito de modelar uma ou mais ontologias para serem usadas dentro dos parâmetros das mensagens compartilhadas entre agentes em suas interações, é padronizar o vocabulário do domínio do problema que está sendo abordado e, desta forma, facilitar a manipulação e compreensão deste vocabulário pelos agentes.

Como visto na Seção 4.2.6, para implementar ontologias em JADE podemos utilizar as seguintes classes providas por sua API: *Ontology*, *Predicate*, *Concept* e *AgentAction*. Para apresentar a estrutura de ontologias para comunicação dos agentes, também utilizaremos diagramas de classes da UML, estendidos com os estereótipos `<<ontology>>`, `<<predicate>>`, `<<concept>>` e `<<action>>`, modelando em nível de projeto detalhado a representação de ontologias suportadas por JADE.

5.2.2 Projeto da dimensão comunicativa

Devido à grande importância da comunicação inerentemente realizada no paradigma de orientação a agentes, onde os agentes interagem uns com os outros através de trocas de mensagens, também

contribuímos com os modelos da dimensão comunicativa. Esta dimensão representa, de maneira temporal, os eventos relacionados a trocas de mensagens realizada no sistema.

Os dois modelos desta dimensão são o modelo de interações e o modelo de mensagens. O modelo de interações está preocupado com os protocolos que serão utilizados nas conversações entre os agentes, bem como com a estrutura dessas interações. Utilizamos diagramas de sequência da UML estendidos com novos estereótipos. O modelo de mensagens, entretanto, é um modelo auxiliar proposto nesta dissertação. Ele pode ser usado com o propósito de definir a composição detalhada de uma mensagem que pode ser enviada por agentes, levando em consideração a proposta de especificação de mensagens ACL da FIPA. Para cada evento representado em diagramas de sequência do modelo de interações, podemos detalhar a mensagem que está sendo trocada utilizando o modelo de mensagens.

Apesar de em trabalhos anteriores Tropos sugerir a adoção de qualquer linguagem para comunicação de agentes pré-existente [Bresciani04], nossa proposta está focada na implementação de mensagens utilizando a linguagem FIPA-ACL, uma vez que esta é a linguagem adotada por JADE e está de acordo com os padrões da FIPA.

Modelo de Interação

O modelo de interação utiliza diagramas de sequência da UML para apresentar a comunicação realizada através da troca de mensagens entre os agentes. A diferença existente entre os diagramas de sequência já aplicados nos demais estudos de caso da metodologia Tropos é que os agentes que trocam mensagens podem ser estereotipados com <<agent>>. Além disso, podemos utilizar o os eventos de trocas de mensagens podem indicar a mensagem que está sendo detalhada no modelo de mensagens, assim como o protocolo que está sendo usado.

Modelo de Mensagens

O modelo de mensagens é um modelo auxiliar proposto nesta dissertação para ser utilizado com a finalidade de definir de forma detalhada as mensagens trocadas pelos agentes, usando a estrutura proposta pela FIPA para especificação de mensagens ACL [FIPA00]. Para cada evento representado em um determinado diagrama de sequência do modelo de interação, podemos detalhar a mensagem que está sendo trocada através do modelo de mensagens. A Tabela 5.3: *Template* para definição de mensagens ACL apresenta um template para preenchimento dos campos da mensagem ACL que será modelada.

Tabela 5.3: *Template* para definição de mensagens ACL

Mensagem ACL		
Performativa FIPA	PERFORMATIVA	
Endereçamento	<u>sender</u>	Identificação do agente transmissor
	<u>receiver</u>	Identificação do agente receptor
Comunicação	<u>:protocol</u>	Nome do protocolo de interação
Mensagem	<u>:language</u>	Linguagem de conteúdo
	<u>ontology</u>	Ontologia da mensagem
	content	Conteúdo da mensagem

5.2.3 Projeto da dimensão dinâmica

A dimensão dinâmica modela as capacidades e possíveis comportamentos necessários para que o agente realize suas tarefas e atinja seus objetivos. Os dois modelos propostos para esta dimensão são o modelo de capacidades e o o modelo de comportamentos. O modelo de capacidades da dimensão dinâmica serve para auxiliar a distribuição dos comportamentos através das capacidades de cada agente. Ele associa agentes a capacidades e capacidades a comportamentos.

Outro modelo proposto para ser utilizado no projeto detalhado de Tropos é o modelo comportamental. Sua principal função é projetar tarefas e intenções dos agentes que serão implementadas utilizando a estrutura de comportamentos oferecida por JADE. Nesta dimensão podemos utilizar tabelas para representar o modelo de capacidades e diagramas de atividades da UML para especificar os comportamentos dos agentes.

Modelo de Comportamentos

O o modelo de comportamentos deve ser usado servirá para projetar as tarefas ou intenções dos agentes. Comportamentos em JADE são unidades lógicas de atividade, que podem ser compostas de várias maneiras para atingir padrões de execução complexos, e que podem ser executadas concorrentemente [Bellifemine04]. A forma com que os desenvolvedores codificam os comportamentos dos agentes determina as operações que eles podem realizar, enquanto que o fluxo de execução de um agente é definido através das interconexões dos comportamentos.

Para Tropos, os atores possuem dependências entre si que podem ser representadas direta ou indiretamente através de comportamentos que os agentes em JADE podem executar. A ativação desses comportamentos pode ser realizada através de trocas de mensagens entre os agentes, fazendo com que comportamentos realizem tarefas, forneçam recursos, ou atendam a metas e metas-soft.

Assim como em [Castro02a], diagramas de atividades da UML [Booch99] podem ser utilizados para especificar os comportamentos internos dos agentes mais detalhadamente. Cada plano de uma gente na arquitetura BDI que é considerada nos trabalhos anteriores de Tropos pode ser representado por um comportamento modelado através de diagramas de atividades. Com essa representação, podemos modelar como o agente executará suas ações e responderá às mensagens que recebeu de outros agentes ou a eventos do ambiente.

Modelo de Capacidades

Este é um modelo auxiliar para relacionar em uma tabela os serviços que os agentes são capazes de fazer [Kolp05]. Essas capacidades devem ser inicialmente identificadas a partir da modelagem da arquitetura realizada na fase de projeto arquitetural, e em seguida este modelo pode ser utilizado para relacionar estes serviços a agentes e comportamentos do sistema. A identificação desses serviços no projeto detalhado é útil também para modelar os serviços dos agentes que podem ser registrados nas páginas amarelas da plataforma.

A tabela a seguir apresenta um template para definição dos serviços que agentes são capazes de executar e os comportamentos que estão associados a esses serviços.

Tabela 5.4: Template para definição do modelo de capacidades

Capacidade	Descrição	Agente
<i>Nome do serviço</i>	<i>Descrição do serviço</i>	<i>Agente capaz de fornecer o serviço</i>
...		

5.3 A implementação em JADE a partir de Tropos

Estamos sugerindo nesta seção alguns passos para realizar a implementação dos sistemas em JADE a partir do projeto detalhado utilizando a proposta descrita neste capítulo. No entanto, precisamos validar melhor a aplicação da proposta para só então propor diretrizes mais consistentes de como fazer a ligação do projeto detalhado de Tropos desde o projeto da arquitetura até a implementação em JADE. A seguir são apresentadas algumas considerações.

Atores identificados na fase de projeto arquitetural de acordo com a metodologia Tropos utilizando-se a notação do framework i* podem se representados com classes ou componentes de software no sistema a ser desenvolvido em JADE. Esses elementos podem ser representados por classes comuns de JAVA.

Por sua vez, os atores que forem representados como agentes na modelagem do projeto arquitetural podem ser representados em nosso modelo através de classes utilizando o estereótipo <<agent>>. Todo agente em JADE, como visto no Capítulo 4, possui alguns métodos básicos que precisam ser implementados. Estes métodos e alguns atributos são representados no diagrama de relacionamentos dentro da classe do agente e podem ser implementados como na orientação a objetos. Um agente que estiver relacionado com um comportamento através de associações do tipo <<has>> irá adicionar este comportamento em sua estrutura quando for realizada a sua implementação.

As dependências de tarefa e de meta do framework i* sugerem a declaração de comportamentos para realizá-las. Além disso, dependências sugerem trocas de mensagens entre os agentes e conseqüentemente uso da ontologia para esta comunicação. Quando um agente solicitar que outro realize uma tarefa, podemos utilizar o conceito de <<action>> para modelar a ação que precisa ser realizada sobre algum conceito da ontologia. Se uma meta precisar ser atingida, naturalmente podemos utiliza predicados da ontologia para realizar consultas a respeito do alcance ou não da meta e receber respostas lógicas em relação ao questionamento que foi realizado.

Dependências de recurso estão associadas a conceitos da ontologia e a comportamentos para fornecer recursos. Esses comportamentos por sua vez precisam considerar a ontologia que define o conceito do recurso para que o agente ou ator que depende dele possa compreender o envio da mensagem de fornecimento.

Podemos observar ainda que os links de decomposição e meios-fins sugerem composições de comportamentos em JADE. A natureza da dependência que estiver sendo decomposta ou fornecendo alternativas para que seja realizada é que vai indicar qual das estruturas comportamentais de JADE será utilizada.

É importante observar os sentidos das dependências no diagrama arquitetural em i^* , para poder identificar que agentes realizarão a dependência através de seus comportamentos. Naturalmente, o agente que depende de outro pode atuar como iniciador da conversação que visa o cumprimento do objetivo ou tarefa, realizando solicitações através de mensagens.

Com relação à implementação em JADE, a partir dos modelos de projeto construídos podemos seguir os seguintes passos:

- Implementar as classes de agentes,
- implementar seus comportamentos básicos, que correspondem às suas intenções;
- Para cada agente, implementar, de acordo com a estrutura dos comportamentos associados a ele, os comportamentos de mais baixo nível e menor granularidade primeiro, fazendo sua composição com outros comportamentos em seguida.
- Definir em detalhe as ações dos comportamentos do agente;
- Implementar a ontologia para comunicação dos agentes;
- Executar a plataforma JADE e verificar os fluxos de execução dos agentes, depurando seu funcionamento e suas interações.

5.4 Considerações Finais

A ausência de notações aceitáveis para serem utilizadas efetivamente com intuito de modelar SMAs faz com que o potencial do paradigma de agentes não seja totalmente explorado. A comunidade de orientação a agentes vem tentando resolver este problema investigando a possibilidade de estender a UML para dar suporte à modelagem de conceitos básicos do paradigma, tais como agentes, crenças, ontologias e comportamentos [Bergenti00, Mylopoulos01, Odell01]. Mesmo assim, a maioria desses trabalhos não cobre toda extensão do ciclo de desenvolvimento. Por isso é fundamental o uso de metodologias de desenvolvimento que forneçam mecanismos e estruturações que facilitem a transição de conceitos e abstrações através das fases de desenvolvimento, desde os requisitos até a implementação.

Neste capítulo foi apresentada nossa proposta de complementação da fase de projeto detalhado de Tropos. Foram propostos dez novos estereótipos para serem utilizados nos diagramas da modelagem do projeto de SMAs sendo desenvolvidos com a metodologia Tropos e a plataforma JADE. Os diagramas propostos estão distribuídos por três dimensões: estrutural, comunicativa e dinâmica. Cada uma está preocupada em visualizar o projeto detalhado sob um aspecto diferente.

Esperamos complementar nossa proposta no futuro, para incluir outras características importantes do desenvolvimento de sistemas orientados a agentes que não fizeram parte do escopo desta dissertação. Entretanto, o conjunto de elementos sugeridos e o grau de abstração em que foram modelados apresenta-se aplicável ao projeto detalhado de sistemas que obedeçam ao paradigma de orientação a agentes.

Capítulo

6 Estudo de Caso

Este capítulo apresenta um estudo de caso desenvolvido para capturar aspectos dos agentes que precisam ser levados em consideração quando desejamos implementar um SMA em JADE, com o propósito de aplicar nossa proposta de complementação da fase de projeto detalhado da metodologia Tropos. A aplicação escolhida é um sistema de gerenciamento de conteúdo que utiliza agentes para simular o processo de publicação de notícias na *internet*.

6.1 Introdução

A *internet* 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 atualização do 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. 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 no desenvolvimento de um CMS que utiliza agentes para simular o processo de publicação de notícias em um site da *internet*.

6.2 O Smart Journal

O sistema *Smart Journal* é uma ferramenta de gerenciamento de conteúdo, desenvolvida para sistematizar o processo de publicação de notícias *on-line*, através da simulação do processo de funcionamento de uma redação de jornal. Este sistema ainda não foi implementado completamente, mas já se apresenta suficientemente estruturado para servir como estudo de caso desta dissertação.

O *Smart Journal* está voltado para a publicação de conteúdo (informações adicionadas de contexto, estruturadas e formatadas) na *internet*. Num jornal, notícias e fotografias podem ser consideradas conteúdos de publicação. Cada edição possui notícias associadas. Estas notícias podem possuir fotos relacionadas e são categorizadas e distribuídas através de editorias, cada qual sob responsabilidade de um determinado editor. Por exemplo, notícias relacionadas a esportes são entregues por repórteres, ou obtidas através de agências de notícias, para o editor de esportes. Os repórteres e fotógrafos recebem pautas que devem ser cumpridas para que sejam providenciadas notícias e fotografias. A pauta geral da edição é decidida numa reunião, onde editores e o chefe de redação discutem e avaliam a necessidade de se investigar um determinado assunto para que sejam publicadas notícias a respeito dele.

Para garantir a interoperabilidade do sistema com fornecedores reais de notícias, o *Smart Journal* obedece ao padrão NITF (*News Industry Text Format*) [IPTC04]. Este padrão oferece um vocabulário XML aberto e bem documentado, que foi desenvolvido pelo IPTC (*International Press Telecommunications Council*) para definir conteúdo e estrutura de artigos de notícias, e atualmente é o mais comumente utilizado entre empresas que publicam ou fornecem notícias. Alguns dos meios de comunicação que atualmente utilizam este padrão são: a agência de notícias AFP (*Agence France Press*) e o jornal *The New York Times*, entre outros.

6.3 Modelagem com Tropos

A modelagem do *Smart Journal* foi completamente realizada através da proposta Tropos, sua especificação através das quatro fases da metodologia está apresentada nesta seção.

6.3.1 Requisitos Iniciais

Uma redação de jornal pode ser vista como uma organização dinâmica onde várias pessoas colaboram, cada uma com suas habilidades, para produção de edições de jornais a serem veiculados em diferentes tipos de mídia. Comumente, essa organização está situada dentro de um ambiente maior, uma empresa que explora vários meios de comunicação (rádio, televisão, etc.).

Através da análise deste ambiente organizacional, realizada através de pesquisas e de algumas visitas feitas a uma redação de jornal real, 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 *internet* 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 6.1 apresenta o modelo i* de dependência estratégica da fase de Requisitos Iniciais deste trabalho.

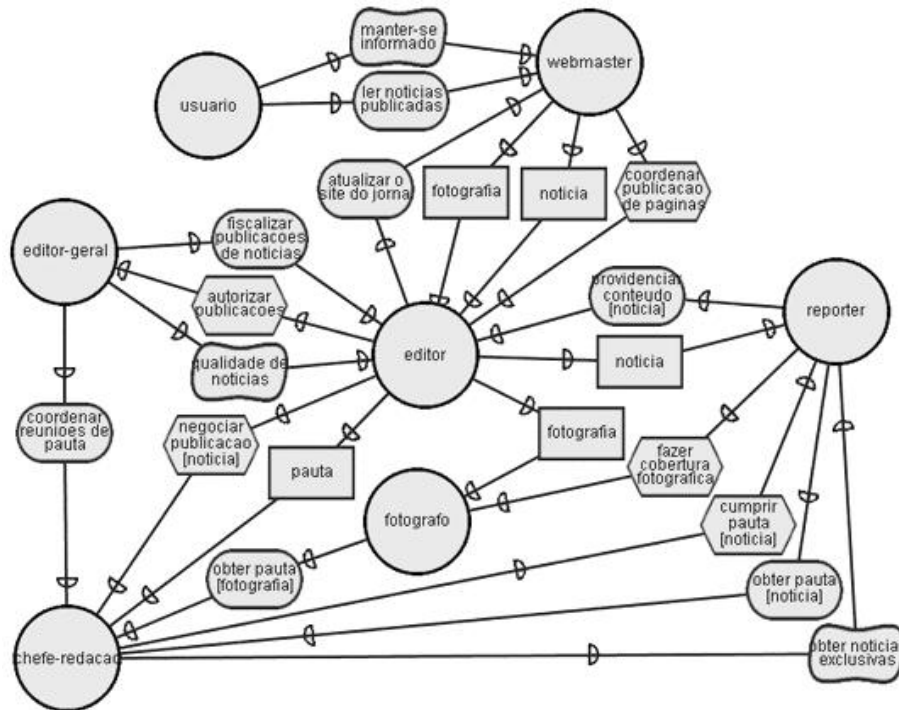


Figura 6.1: Modelo i* de Dependência Estratégica dos Requisitos Iniciais

O *usuário* é o ator que representa o leitor do conteúdo publicado no site do jornal. Ele depende do *webmaster* para completar seu objetivo *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. 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* seja satisfeito e do *fotógrafo* para que a tarefa *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. Ao mesmo tempo, 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* 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 razão estratégica permite um refinamento das descrições dos processos descrevendo os relacionamentos intencionais que são internos aos atores. A Figura 6.2 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*.

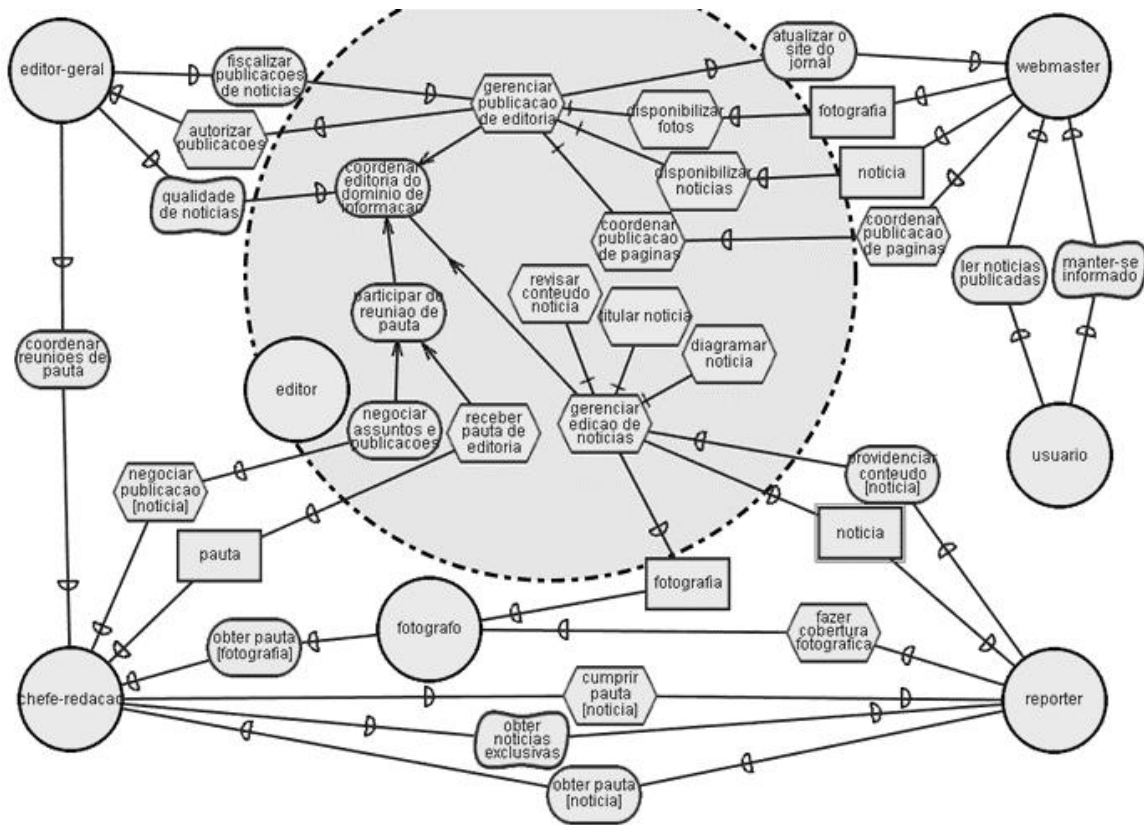


Figura 6.2: Modelo i* de Razão Estratégica de uma redação de jornal para o ator *editor*

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 *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*. Para atender às dependências em relação ao ator *chefe-redação*, o *editor* pode atingir a meta *participar de reunião de pauta* de 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*.

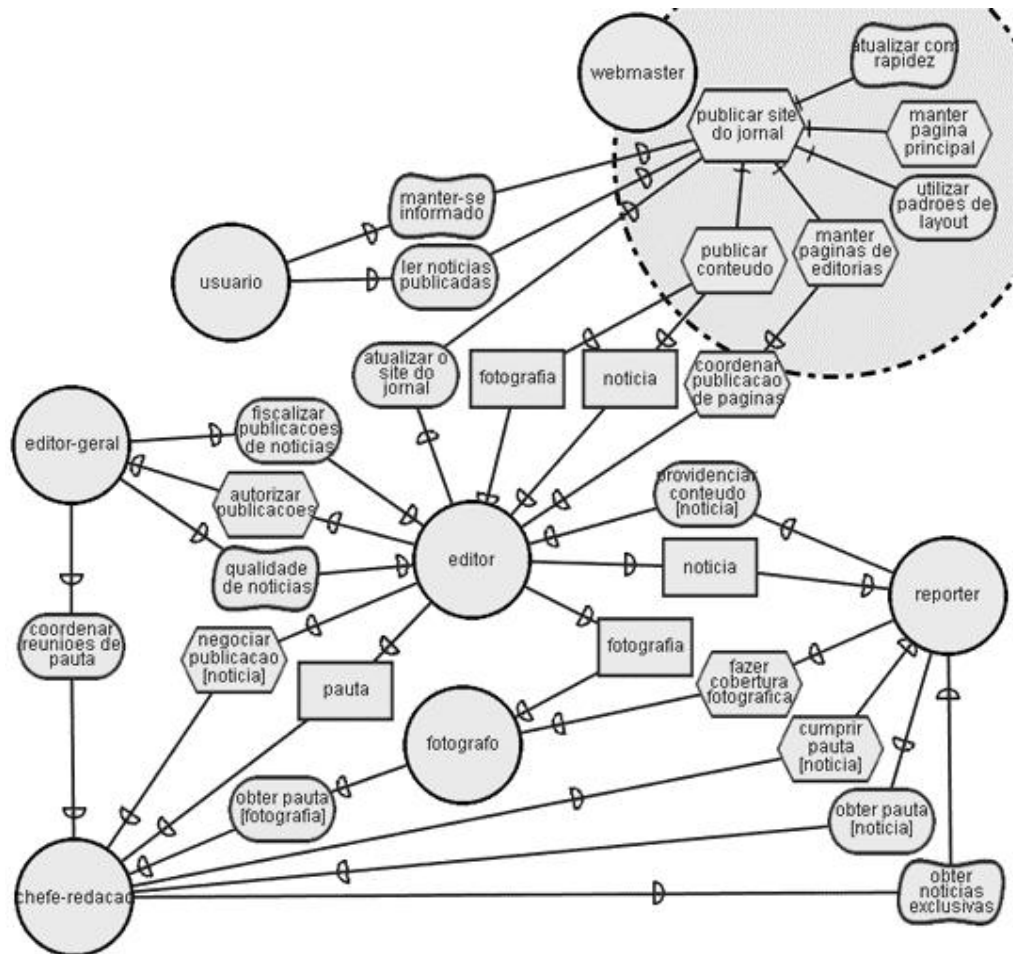


Figura 6.3: Modelo i* de Razão Estratégica de uma redação de jornal para o ator *webmaster*

As intenções do ator *webmaster*, apresentadas na Figura 6.3, 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*.

A Figura 6.4 apresenta o modelo de razão estratégica dos requisitos iniciais com a expansão do ator *repórter*. Este ator 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 *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*. Um *furo de reportagem* acontece quando o repórter obtém uma notícia muito importante que não estava programada na pauta de assuntos daquela edição. 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*.

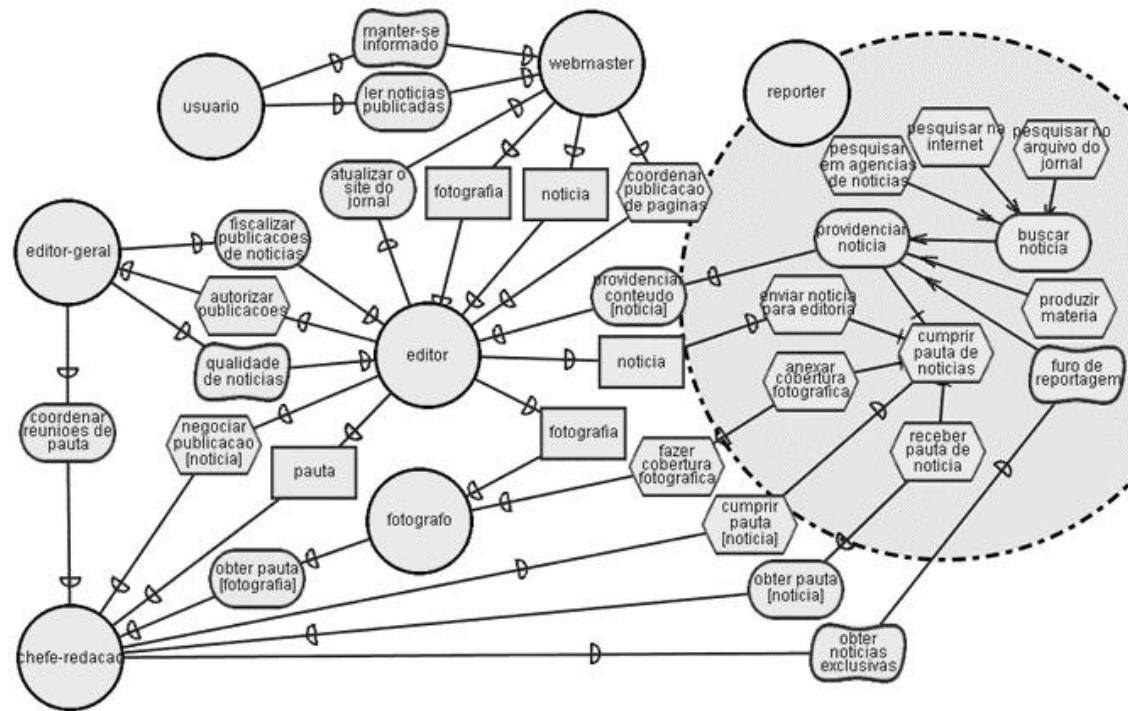


Figura 6.4: Modelo i* de Razão Estratégica dos Requisitos Iniciais para o ator repórter

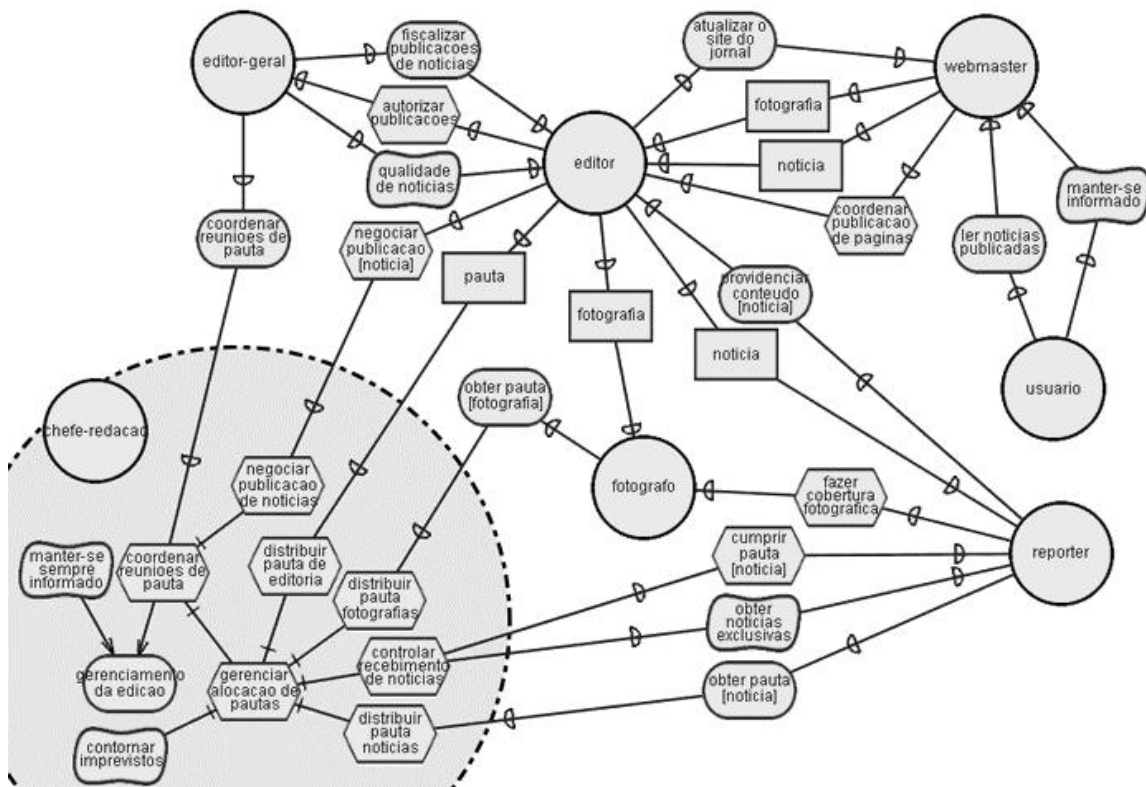


Figura 6.5: Modelo i* de Razão Estratégica de uma redação de jornal para o ator chefe-redação

A Figura 6.5 apresenta a expansão do ator *chefe-redação* no modelo de razão estratégica dos requisitos iniciais. O principal objetivo do ator *chefe-redação* 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*.

6.3.2 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.

Dois novos atores foram inseridos nesta fase, são eles: o *sistema gerenciador de notícias* e a *agência de notícias*. Além disso, como a intenção do sistema *Smart Journal* é automatizar parte do processo de publicação de notícias, 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. A principal motivação para esta decisão é porque as responsabilidades antes atribuídas a esses atores são agora atribuídas ao sistema conforme será apresentado no modelo SR desta fase.

Modelo de Dependência Estratégica

O *Smart Journal* está representado no modelo SD descrito na Figura 6.6 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*, que 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 *internet*, 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 ator sistema possui relacionamentos de dependência com os atores *editor* e *agência de notícias* para serem fornecidos os recursos *notícia* e *fotografia*. 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*. Por sua vez, o *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. O sistema também depende do editor para que seja realizado o objetivo atualizar o site do jornal.

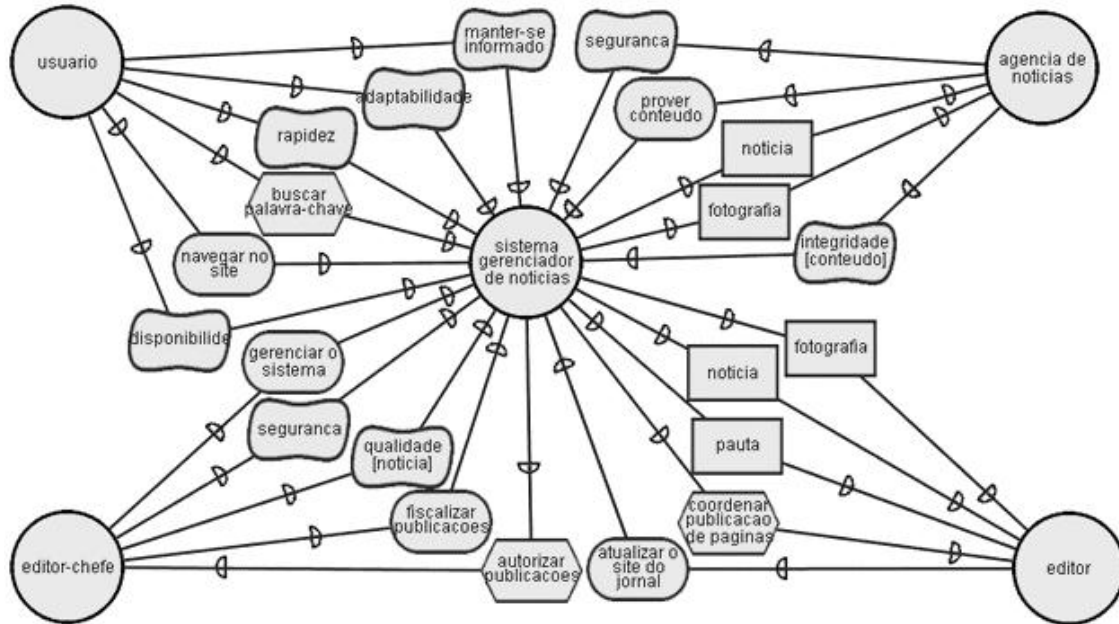


Figura 6.6: Modelo i* de Dependência Estratégica dos Requisitos Finais

O editor-chefe depende do sistema para atingir as metas de *gerenciar o sistema* e *fiscalizar publicações* de notícias, enquanto que o sistema gerenciador de notícias depende do editor-chefe para realização da tarefa *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.

Modelo de Razão Estratégica

Depois que os atores principais e suas dependências estratégicas com relação ao sistema foram identificados, o modelo SR apresenta como os objetivos podem ser atingidos através das contribuições dos atores. A Figura 6.7 apresenta o modelo SR da fase de Requisitos Finais, com o ator sistema gerenciador de notícias expandido para apresentar as razões que estão por trás de suas dependências com os demais atores. Podemos visualizar os principais objetivos identificados para o sistema, que são: *atualizar site*, *capturar profile*, *gerenciar site*, *controlar recebimento de conteúdo* e *coordenar editoriais*. Esses objetivos são apresentados numa estrutura de decomposição e finalidades, através de ligações do tipo meios-fins e ligações de decomposição de tarefas.

A meta do sistema para *capturar profile* de usuário pode ser realizada através de ambas ou de uma das tarefas *consultar notícia* e *pesquisar conteúdo*. A pesquisa por conteúdo das publicações é decomposta e realizada através das tarefas *consultar base de dados* e *consultar agência de notícias*. Estas tarefas estão relacionadas com as dependências externas em relação ao usuário de *navegar no site* (meta) e *buscar palavra-chave* (tarefa).

O sistema possui três meios para atingir a finalidade de *gerenciar o site*: através da meta *atualizar o site*; da tarefa *produzir estatísticas* ou de outra tarefa para *gerenciar a publicação de conteúdo*. A

meta *atualizar o site* pode ser atingida através da realização de uma das três tarefas: *publicar conteúdo*, *configurar layout*, ou *atualizar editoria*. O gerenciamento do conteúdo das publicações é uma tarefa mais complexa do que a atualização no site, sendo decomposta em outras três tarefas e duas metas que estão sob responsabilidade do ator *sistema gerenciador de notícias*. Estas tarefas consistem em *autorizar a publicação de conteúdo*, *armazenar o conteúdo* de notícias e fotografias, e *gerenciar pautas*. As duas sub-metas resultantes da decomposição da tarefa de gerência do conteúdo da publicação são *coordenar editorias* e *controlar recebimento de conteúdo*. Com relação a esta última meta, há duas tarefas que são meios para atingi-la: *receber conteúdo de agência* ou *receber conteúdo produzido pelo editor*.

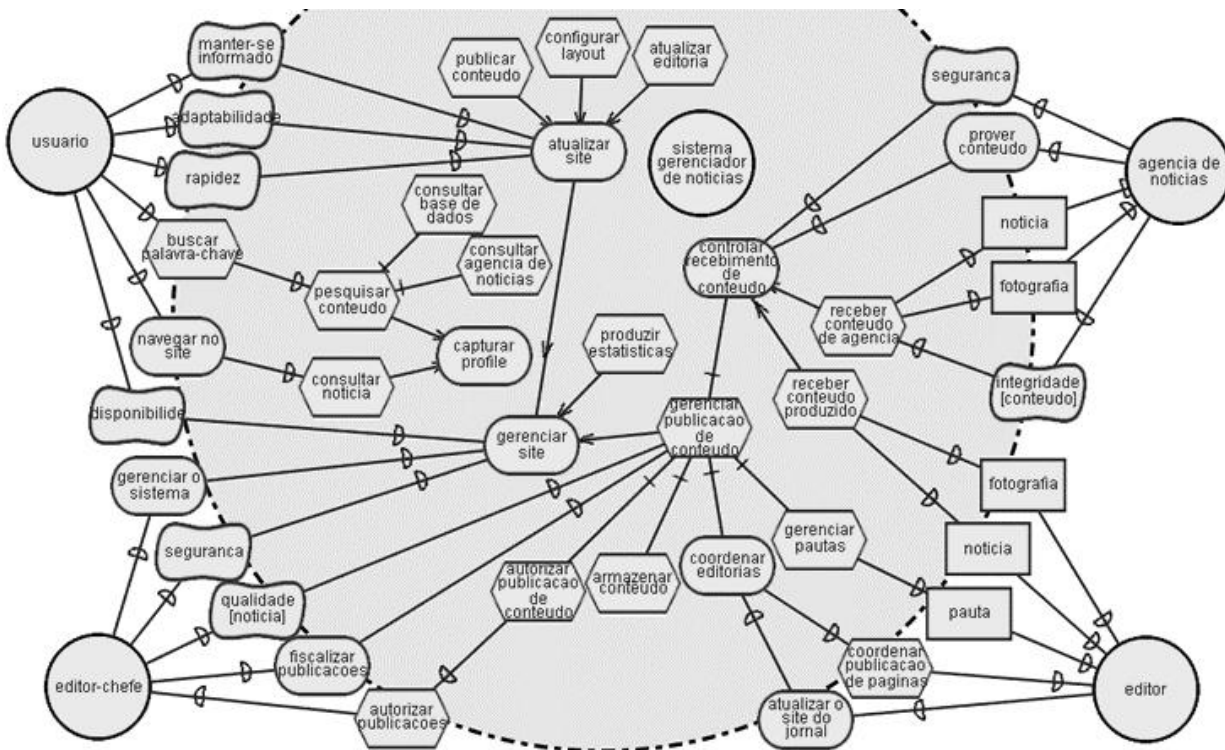


Figura 6.7: Modelo i* de Razão Estratégica dos Requisitos Finais

Neste modelo SR de requisitos finais, foram identificadas as metas-soft mais que mais representam restrições com relação ao sistema, são elas: *disponibilidade*, *segurança* e *adaptabilidade*. Elas serão usadas na próxima fase para auxiliar na escolha do estilo arquitetural mais adequado para o *Smart Journal*, conforme veremos na seção a seguir.

6.3.3 Projeto Arquitetural

A fase de Projeto Arquitetural é iniciada através da atividade de escolha do estilo arquitetural organizacional mais adequado para o sistema. Esta atividade possui como elementos fundamentais para realização do processo de escolha os atributos de qualidade identificados na fase de Requisitos Finais. Para conduzir tal análise, Tropos utiliza o *framework* NFR [Chung00]. Em seguida, é realizada a modelagem da arquitetura do sistema baseando-se no estilo arquitetural mais fortemente sugerido

através da primeira atividade. Por fim, podemos aplicar padrões sociais na estruturação da arquitetura do sistema a fim de usufruir das vantagens oferecidas pelos padrões de projeto sugeridos para Tropos [Do03].

Escolha do estilo arquitetural

O refinamento dos principais atributos de qualidade identificados na fase de Requisitos Finais é realizado por meio da construção de um grafo SIG do NFR [Chung00], com o auxílio do catálogo de correlação entre os estilos arquiteturais e estes atributos (Tabela 3.2). Neste grafo, os atributos de qualidade e suas sub-metas são representados como nuvens de contorno simples e os estilos arquiteturais aparecem no grafo como nuvens de borda espessa, conforme notação explicada na Seção 3.3.3.

Na Figura 6.8, podemos verificar as interdependências e o refinamento dos três atributos de qualidade mais importantes identificados para o sistema: *disponibilidade*, *segurança* e *adaptabilidade*. A *disponibilidade* e a *adaptabilidade* do sistema são qualidades requeridas pelo ator *usuário* em relação ao sistema *Smart Journal*., enquanto que a *segurança* é dependida tanto pela *agência de notícias* quanto pelo *editor-chefe*. Por se tratar de um sistema de serviços para Web, a *disponibilidade* e a *segurança* foram consideradas como características prioritárias no nosso sistema, por isso estão identificadas no grafo com o sinal de exclamação (!).

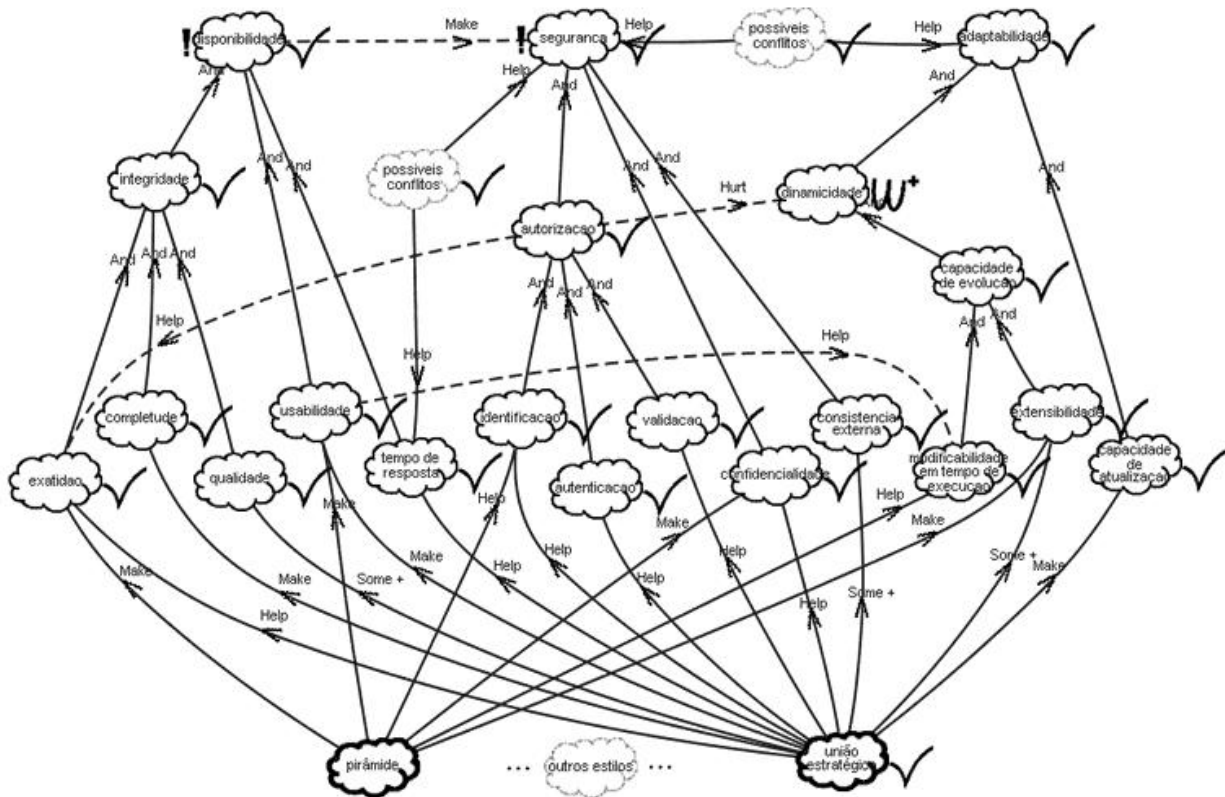


Figura 6.8: Grafo de interdependências de metas-soft do *Smart Journal*

O sistema *Smart Journal* precisa estar disponível para acesso levando-se em consideração preocupações relacionadas à integridade de dados, facilidade de uso, e eficiência no acesso às informações disponibilizadas para o usuário. O atributo de qualidade *disponibilidade* foi refinado através de sua decomposição em três sub-metas: *integridade*, *usabilidade* e *tempo de resposta*. A *integridade* requerida envolve outras qualidades, sendo refinada em *exatidão*, *completude* e *qualidade*.

Uma aplicação para *Web* segura deve garantir alguns aspectos importantes como a realização de transações de dados consistentes (no nosso caso, o fornecimento de notícias). Além disso, mecanismos de autenticação de usuários e proteção de acesso, ao processamento e armazenamento dos dados, precisam estar minimamente sujeitos à falhas. Na nossa análise dos atributos de qualidade, *segurança* está decomposta em *autorização*, *confidencialidade* e *consistência externa*. Para *autorização*, o refinamento resultante é composto pelos requisitos não-funcionais *identificação*, *autenticação* e *validação*.

A adaptabilidade lida com a maneira como o sistema pode ser projetado usando mecanismos genéricos para permitir, por exemplo, que páginas Web sejam alteradas dinamicamente. A divulgação e apresentação das notícias do site do *Smart Journal* devem ser constantemente atualizadas pelo sistema. Por isso, temos que o atributo de qualidade *adaptabilidade* pode ser subdividido em *dinamicidade* e *capacidade de atualização*. A *dinamicidade* envolve a *capacidade de evolução* do sistema, para qual podemos considerar as sub-metas *modificabilidade em tempo de execução* e *extensibilidade*.

Quando identificamos as interdependências positivas e negativas dos atributos de qualidade, constatamos que pode haver possíveis conflitos entre as metas-soft *segurança* e *tempo de resposta*, uma vez que os mecanismos utilizados para tratar a autorização e confidencialidade do sistema podem demandar processamentos adicionais que comprometam o seu tempo de resposta. Também consideramos possível a existência de conflitos entre a *segurança* e a *adaptabilidade* do sistema. Ainda constatamos que o requisito *disponibilidade* contribui de maneira suficientemente positiva (*make*) para *segurança* do sistema, enquanto a *autorização* contribui negativamente (*hurt*) para *dinamicidade* e positivamente para *exatidão*. Além disso, a *usabilidade* contribui de modo parcialmente positivo (*help*) para a *modificabilidade em tempo de execução* do sistema.

Após analisarmos o grafo SIG apresentado na Figura 6.8, avaliando os estilos arquiteturais com relação ao cumprimento das metas refinadas, optamos por utilizar o estilo arquitetural organizacional *União Estratégica*, apresentado na Figura 6.8 como uma meta operacionalizada marcada com um “√”.

Modelagem da arquitetura

A arquitetura organizacional do sistema *Smart Journal* é decomposta em três atores que representam os parceiros principais e um ator que representa uma gerência comum. Os parceiros principais identificados foram: *coletor*, *apresentador* e *publicador*. A gerência comum será representada pelo *gerenciador*. Contudo, futuramente podem ser acrescentados parceiros secundários. Cada parceiro pode gerenciar e controlar a si próprio em uma dimensão local e pode interagir diretamente com os outros parceiros para trocar recursos, tais como dados e conhecimento. A Figura 6.9 apresenta uma sugestão de atribuição de responsabilidades para o sistema baseando-se no estilo *União Estratégica* [Do03].

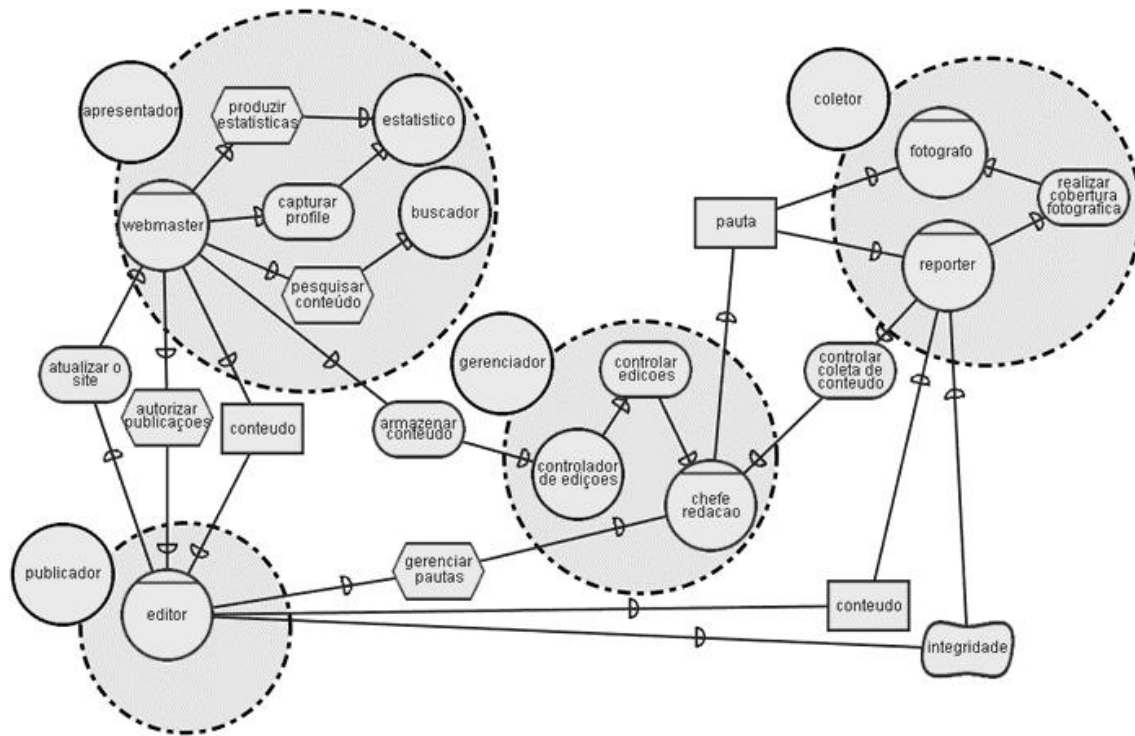


Figura 6.9: Arquitetura do *Smart Journal* baseada no estilo União Estratégica

O preenchimento das obrigações dos atores é realizado através da decomposição dos parceiros principais e da gerência comum em outros atores e agentes. Considerando que o sistema a ser desenvolvido trata-se de um simulador, dentre os atores que fazem parte da representação arquitetural, identificamos também alguns agentes escolhidos para fazer parte da arquitetura. Esses agentes são equivalentes a atores descobertos nas fases iniciais do desenvolvimento, que no mundo real realizariam as tarefas e teriam objetivos semelhantes a atingir, e serão tratados como agentes de software.

Visualizando a arquitetura de maneira bem abstrata, observamos que o subsistema *coletor* é responsável por suprir o *publicador* com notícias e fotografias que, por sua vez, são solicitadas pelo *gerenciador* e disponibilizadas no site pelo módulo *apresentador*.

Quando um usuário deseja ler as notícias publicadas pelo *Smart Journal*, ele acessa o *website* mantido pelo parceiro *apresentador* através do agente *webmaster*, que é responsável pela atualização e disponibilização do conteúdo publicado no site. A informação a ser publicada pode ser provida por editores e repórteres do mundo real (através da interface do site), ou obtida das agências de notícias que forem fornecedoras de conteúdo para o sistema. A coleta dessas notícias é realizada pelo parceiro principal *coletor*, através de seus sub-atores *repórter* e *fotógrafo*. O *gerenciador* controla a coleta de notícias e o ciclo de vida da publicação dessas notícias, além de gerenciar a distribuição de responsabilidades para *editores*, *repórteres* e *fotógrafos*.

O *gerenciador* depende do *coletor* para ter o conteúdo (notícias e fotografias) obtido de acordo com as pautas da edição enviadas por ele. Desta forma, é necessário manter a consistência entre o que é solicitado do *coletor* e o que é passado para o *publicador* de maneira que as editorias sejam

bem-definidas, uma vez que cada agente *editor* é responsável por uma determinada categoria de notícias. Por exemplo, um editor pode ser responsável por notícias de esportes, enquanto outro é responsável por notícias internacionais.

O gerenciador seleciona um ou mais *repórteres* e *fotógrafos* para encontrar, respectivamente, as *notícias* e *fotografias* que devem obedecer à determinada *pauta*. Esse conteúdo deve ser entregue depois ao editor correspondente, que validará ou não sua publicação. O agente *chefe-redação* controla o cumprimento das notícias que foram coletadas e repassadas aos *editores*, conseguindo desta forma gerenciar a abertura e fechamento das edições do jornal.

Aplicação dos padrões sociais

Considerando a natureza do sistema e as relações existentes entre os agentes que compõe sua estrutura, analisamos algumas possibilidades de aplicação dos padrões sociais apresentados em [Kolp05] na arquitetura do *Smart Journal*.

Entretanto, não estaremos aplicando nossa proposta com relação a padrões sociais, por isso apenas realizamos uma análise prévia de que padrões poderão ser utilizados numa versão futura do nosso estudo de caso. A seguir são listadas alguma dessas possibilidades:

- Utilizar o padrão *Wrapper* para permitir que os agentes repórteres e fotógrafos realizem interações com os sistemas de software que fornecem as notícias das agências (que podem ser sistemas legados), obtendo as notícias e repassando-as aos agentes responsáveis respeitando os protocolos de comunicação.
- Aplicar o padrão *Mediator* para auxiliar na distribuição das tarefas que são coordenadas ou delegadas pelo agente chefe de redação, tais como a distribuição e gerenciamento de pautas. No padrão *Mediator*, um agente mediador atua como intermediário em interações entre agentes distintos. Um iniciador endereça o mediador no lugar de endereçar diretamente um outro colega, o executor. O mediador tem modelos de conhecimento dos colegas e coordena a cooperação entre eles. Inversamente, cada executor tem um modelo de conhecimento do mediador.
- Um agente *Matchmaker* pode ser adicionado ao sistema a fim de guardar a localização de um determinado provedor de serviços (como por exemplo, coleta de notícias ou fotografias). No padrão *Matchmaker*, um agente localiza um provedor correspondente a uma solicitação de serviço de um cliente, e então direciona o consumidor para se comunicar diretamente com o provedor escolhido. O facilitador de diretórios (DF) de JADE já realiza este papel.
- O agente *chefe de redação* pode trabalhar como um agente *Broker*, selecionando um ou mais repórteres e fotógrafos para achar conteúdos para cobrir uma pauta específica. No padrão *Broker*, o agente é um árbitro que realiza a intermediação do acesso a serviços providos por um agente (provedor) para satisfazer a solicitação de um consumidor. O agente *Broker* localiza provedores correspondentes a solicitação do cliente por um serviço. Ele requisita e recebe o serviço dos provedores de serviço e então encaminha o serviço para o cliente.

6.3.4 Projeto Detalhado

Apresentamos nesta seção alguns diagramas do projeto detalhado do sistema *Smart Journal* utilizando os estereótipos que foram definidos na proposta apresentada no Capítulo 5, de acordo com as visões estabelecidas em cada dimensão.

Dimensão Estrutural (Modelo de relacionamentos)

O modelo de relacionamentos apresentado na Figura 6.10 é resultante das dependências existentes entre os agentes que foram definidos no modelo arquitetural representado na Figura 6.9. As dependências entre esses agentes indicam que eles precisam se comunicar para atingir seus objetivos, obter recursos ou solicitar a realização de tarefas.

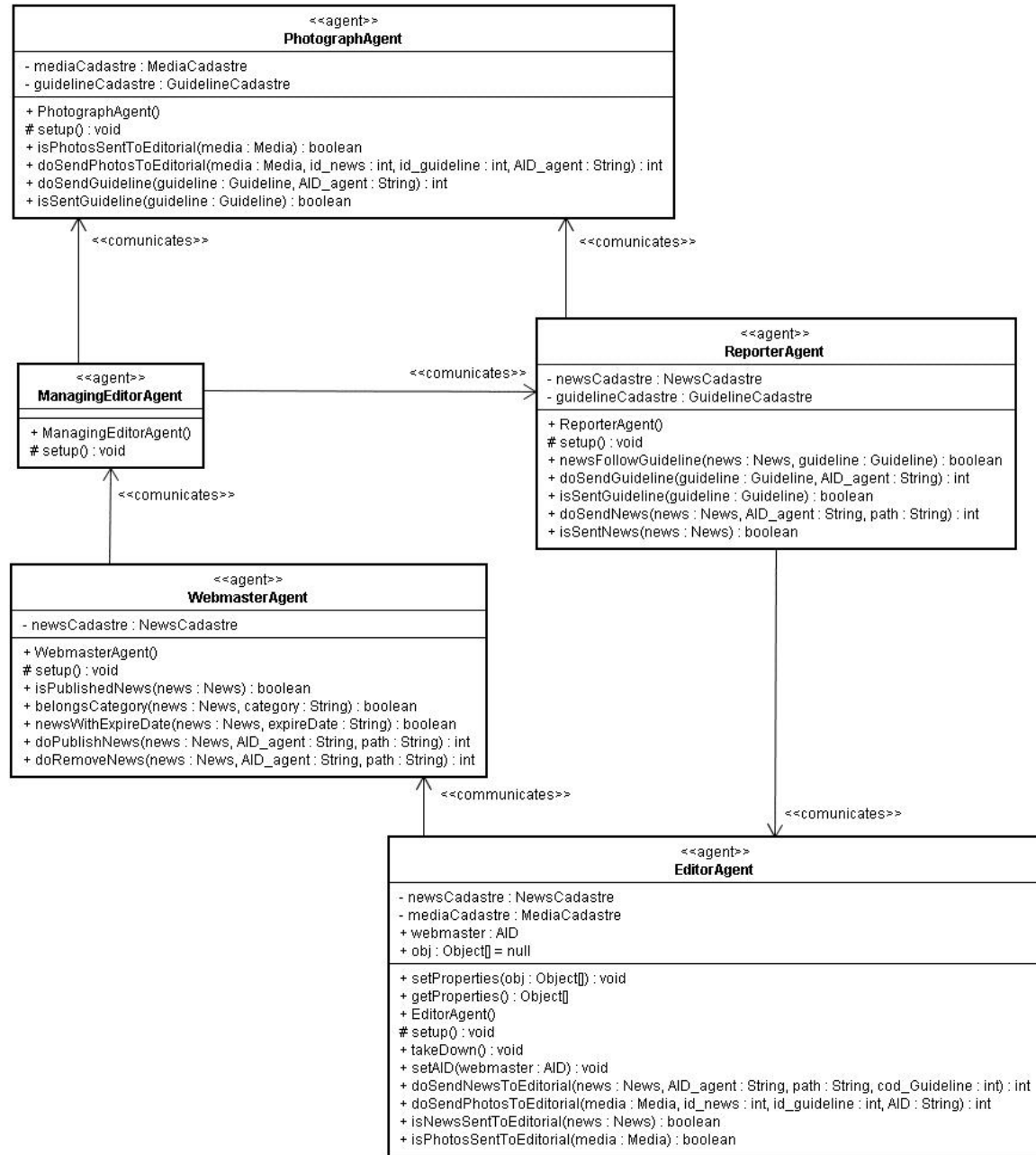


Figura 6.10: Diagrama de relacionamentos para os agentes da aplicação

Dimensão estrutural (Modelo de Ontologias)

Os principais conceitos da ontologia foram identificados através dos recursos representados na modelagem da arquitetura utilizando o *framework* i*: São eles: notícias (*News*), pautas (*Guideline*) e fotografias (*Media*). Além desses conceitos, devido à estrutura definida pelo padrão NITF [IPTC04], outras classes também foram definidas em nossa ontologia. Na Figura 6.11 podemos visualizar a estrutura de relacionamento dos conceitos identificados.

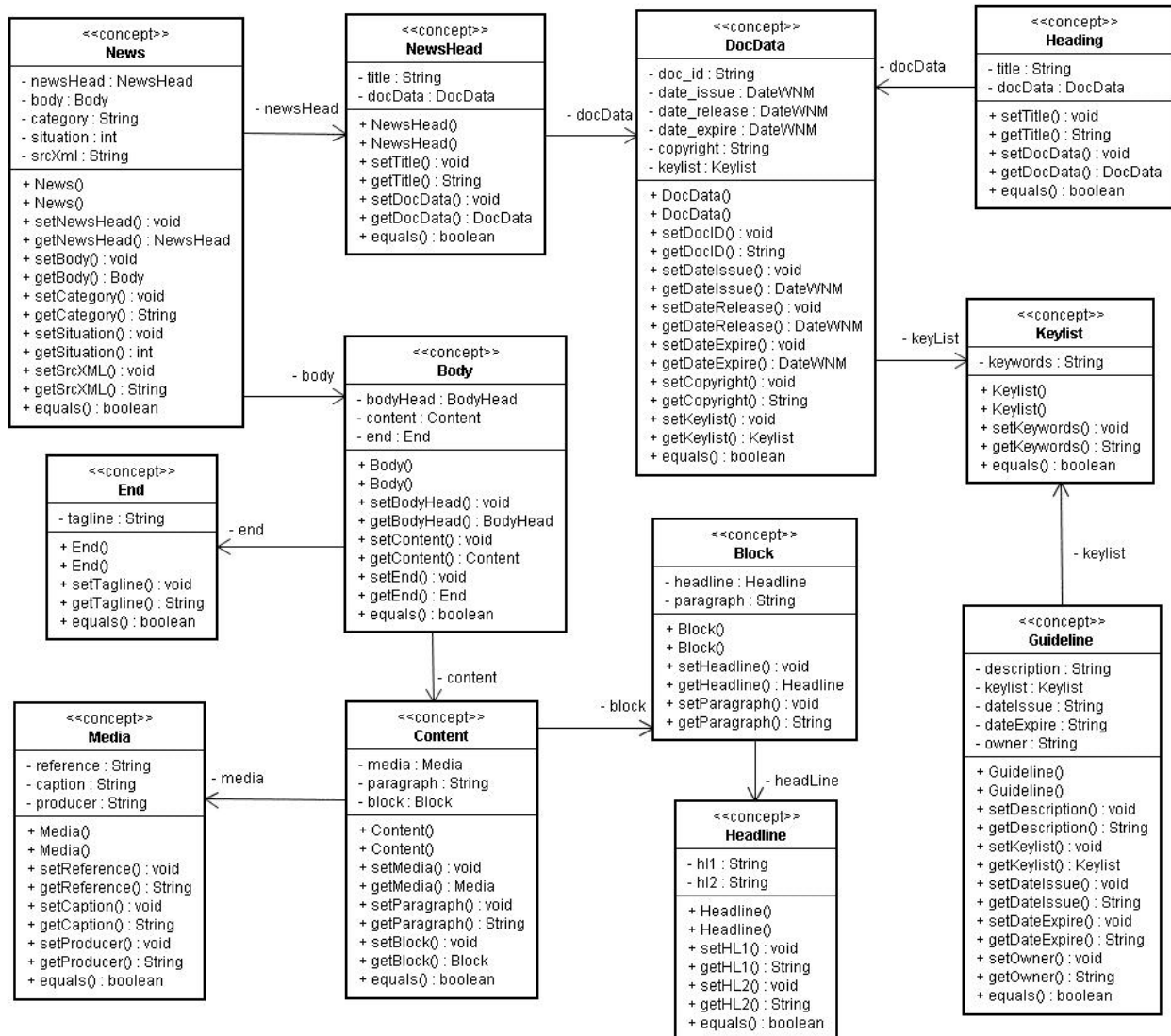


Figura 6.11: Visão parcial do diagrama de ontologias para classes de conceitos

As figuras Figura 6.12 e Figura 6.13 apresentam, respectivamente, a estrutura parcial dos conceitos de predicados e ações de agentes da ontologia modelada para o *Smart Journal*. A partir desses diagramas do projeto detalhado da ontologia é possível montar o seu código em JADE, criando uma classe para representar toda ontologia e adicionando os esquemas para os conceitos, predicados e ações de agentes, de forma a compor a estrutura geral da ontologia. O código desta classe pode ser encontrado no Apêndice B.

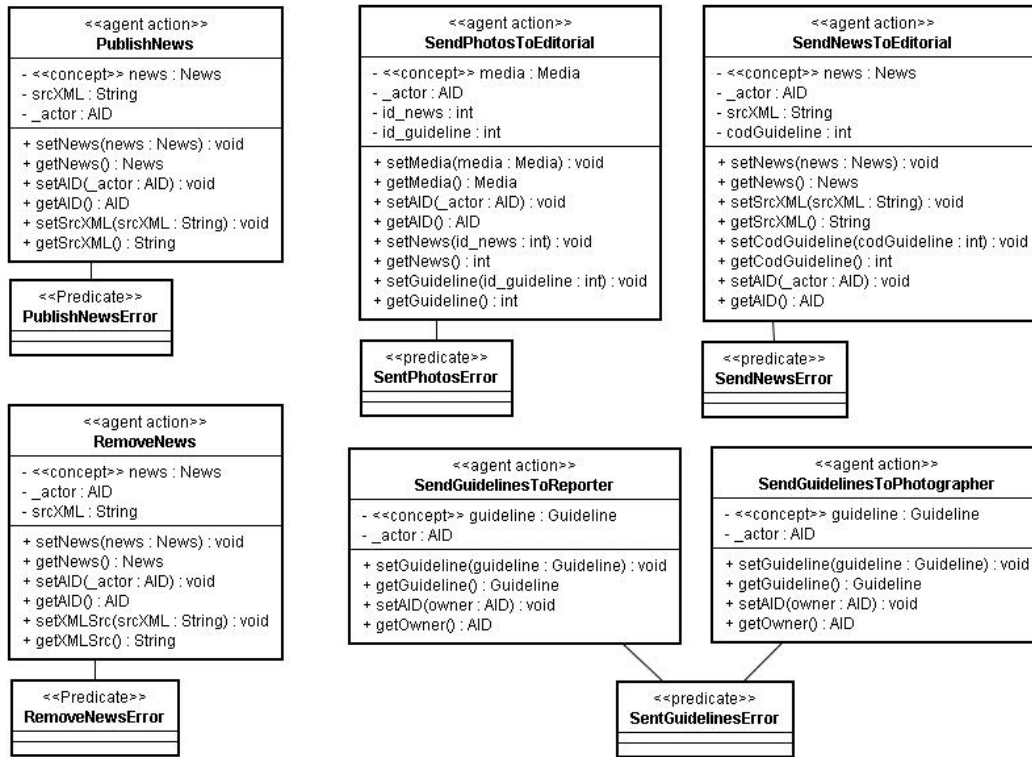


Figura 6.12: Visão parcial do diagrama de ontologias para classes de ações de agentes

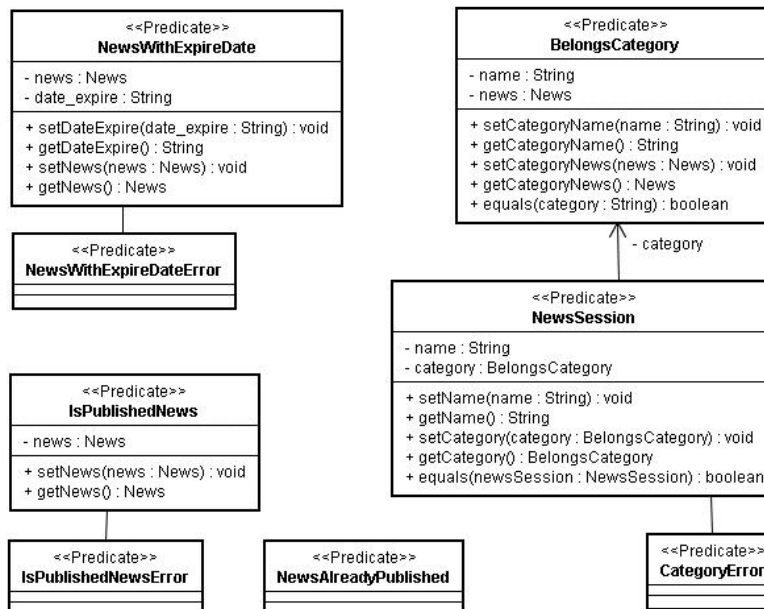


Figura 6.13: Visão parcial do diagrama de ontologias para classes de predicados

Dimensão Comunicativa (Modelo de Interações)

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.

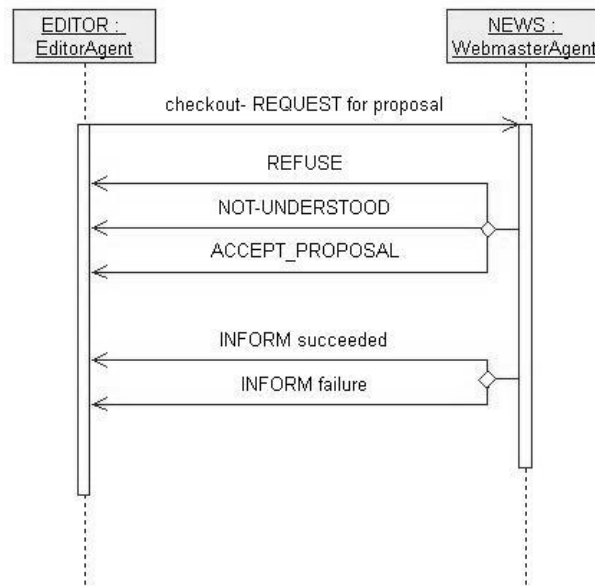


Figura 6.14: Diagrama de interação para o comportamento PublishNews

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.

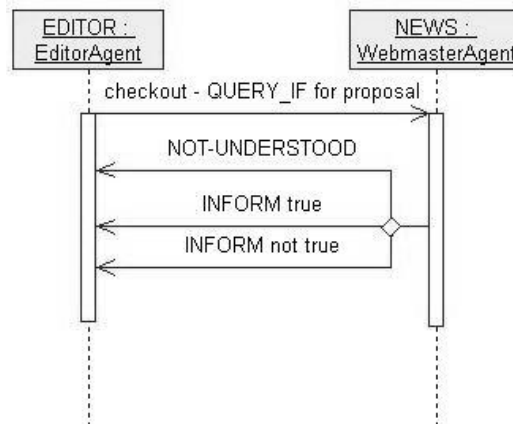


Figura 6.15: Diagrama de interação para o comportamento CheckAlreadyPublish

6.4 Implementação em JADE

A implementação em JADE está sendo realizada a partir de um protótipo construído para investigar o desenvolvimento de sistemas multi-agentes na plataforma. Para cada agente identificado no projeto detalhado, foram criados as classes correspondentes derivadas de *Agent*. Elas foram declaradas no pacote *smagnw.agent*. Os comportamentos foram implementados em um pacote separado (*smagnw.behaviours*) para que possam ser reutilizados por agentes que tenham capacidades semelhantes.

As tabelas a seguir apresentam a estrutura da classe de agente *WebmasterAgent* e um de seus comportamentos, o *HandlePublishBehaviour*.

Tabela 6.1: Estrutura da classe *smagnw.agent.WebmasterAgent*

```
package smagnw.agent;

import jade.core.*;
import jade.domain.*;
import jade.content.lang.sl.*;
import jade.lang.acl.MessageTemplate;
import java.util.Vector;

import smagnw.behaviour.*;
import smagnw.ontology.*;
import smagnw.ontology.concept.*;
import smagnw.ontology.util.news.*;

/** This agent is able to handle the publication of web pages by requesting a webmaster agent
to do that. */

public class WebmasterAgent extends Agent {

    // AGENT LOCAL VARIABLES
    private NewsCadastre newsCadastre;

    // AGENT CONSTRUCTOR
    public WebmasterAgent () {
        super();
        newsCadastre = new NewsCadastre();
    }

    // AGENT SETUP
    protected void setup() {
        System.out.println("This is the WebmasterAgent representing the on-line journal!\n");

        // Register the codec for the SL0 language
        getContentManager().registerLanguage(new SLCodec(), FIPANames.ContentLanguage.FIPA_SL0);

        // Register the ontology used by this application
        getContentManager().registerOntology(NewsOntology.getInstance());

        // Create and add the behaviour for handling QUERIES using the news-ontology
        addBehaviour(new HandleBelongsCategoryQueriesBehaviour(this));
        addBehaviour(new HandleIsPublishedNewsQueriesBehaviour(this));
        addBehaviour(new HandleNewsWithExpireDateQueriesBehaviour(this));

        // Create and add the behaviour for handling REQUESTS using the news-ontology
        MessageTemplate mt = MessageTemplate.and(
            MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_REQUEST),
            MessageTemplate.MatchOntology(NewsOntology.ontologyName));

        HandleBelongsCategoryQueriesBehaviour b1 = new
        HandleBelongsCategoryQueriesBehaviour(this);
        HandleIsPublishedNewsQueriesBehaviour b2 = new
        HandleIsPublishedNewsQueriesBehaviour(this);
        HandleNewsWithExpireDateQueriesBehaviour b3 = new
        HandleNewsWithExpireDateQueriesBehaviour(this);
        HandlePublishBehaviour b4 = new HandlePublishBehaviour(this);
    }
}
```

```

        addBehaviour(b1);
        addBehaviour(b2);
        addBehaviour(b3);
        addBehaviour(b4);
    }

    // AGENT METHODS
    public boolean isPublishedNews(News news) {
        boolean ispublishedNews = false;
        try {
            ispublishedNews = this.newsCadaastre.hasNews(news.getSrcXML());
        } catch (NullPointerException npe) {
            ispublishedNews = false;
        }
        return ispublishedNews;
    }

    public boolean belongsCategory(News news, String category) {
        boolean belongsCategory = false;
        if (news.getCategory().equalsIgnoreCase(category)) {
            belongsCategory = true;
        }
        return belongsCategory;
    }

    public boolean newsWithExpireDate(News news, String expireDate) {
        boolean isNewsWithExpireDate = false;
        if ((news.getNewsHead().getDocData().getDateExpire().toString()).equalsIgnoreCase(expireDate))
        {
            isNewsWithExpireDate = true;
        }
        return isNewsWithExpireDate;
    }

    public int doPublishNews(News news, String AID_agent, String path) {
        int retorno = -1;
        Vector newsVector = null;
        try {
            newsVector = this.newsCadaastre.getNews("");
            if (newsVector.isEmpty()) {
                this.newsCadaastre.insertNews(news, 0, AID_agent, path);
                retorno = 1;
            } else {
                int vectorSize = newsVector.size();
                while (vectorSize > 0) {
                    if (news.equals((News) newsVector.get(vectorSize-1))) {
                        retorno = -1;
                    } else {
                        this.newsCadaastre.insertNews(news, 0, AID_agent, path);
                        retorno = 1;
                        vectorSize = 0;
                    }
                }
            }
        } catch (NullPointerException npe) {
            this.newsCadaastre.insertNews(news, 0, AID_agent, path);
            retorno = 1;
        }
        return retorno;
    }

    public int doRemoveNews(News news, String AID_agent, String path) {
        Vector newsVector = this.newsCadaastre.getNews("");
        if (!newsVector.isEmpty()) {
            if (!newsVector.contains(news)) {
                return (-1);
            } else {
                this.newsCadaastre.insertNews(news, 0, AID_agent, path);
                return (1);
            }
        } else {
            this.newsCadaastre.insertNews(news, 0, AID_agent, path);
            return (1);
        }
    }

```

```

    }
} // End of the class WebmasterAgent

```

Tabela 6.2: Estrutura da classe *smagnw.behaviours.HandlePublishBehaviour*

```

package smagnw.behaviour;

/**
 * <p>Title: SMAGNW</p>
 * <p>Description: Sistema Multi-Agentes para Gerenciamento de Noticias na Web</p>
 * <p>Copyright: Copyright (c) 2004</p>
 * <p>Company: CIN/UFPE</p>
 * @author [igls, bss]@cin.ufpe.br
 * @version 1.0
 */

import jade.proto.SimpleAchieveREResponder;

import jade.core.*;

import jade.domain.*;

import jade.content.*;
import jade.content.onto.basic.*;

import jade.lang.acl.ACLMessage;
import jade.lang.acl.MessageTemplate;

import jade.content.onto.basic.*;

import smagnw.agent.*;
import smagnw.ontology.predicate.*;
import smagnw.ontology.concept.*;
import smagnw.ontology.agentAction.*;

/**
 * This behaviour handles a single publish action that has been
 * requested following the FIPA-Request protocol
 */
public class HandlePublishBehaviour extends SimpleAchieveREResponder {

    Agent myAgent;

    /**
     * Constructor for the HandlePublishBehaviour class.
     * @param myAgent The agent owning this behaviour
     * @param requestMsg The ACL message by means of which the engagement
     * action has been requested
     */
    public HandlePublishBehaviour(Agent myAgent) {
        super(myAgent, MessageTemplate.MatchProtocol(
            FIPANames.InteractionProtocol.FIPA_REQUEST));
        this.myAgent = myAgent;
    }

    /**
     * This method implements the FipaRequestResponderBehaviour.Factory interface.
     * It will be called within a FipaRequestResponderBehaviour
     * when an publication action is requested to instantiate a new
     * HandlePublishBehaviour handling the requested action
     * @param msg The ACL message by means of which the engagement
     * action has been requested
     */

    /**
     * This method actually handles the publication action
     */

    public ACLMessage prepareResultNotification(ACLMessage request, ACLMessage response) {
        // Prepare a dummy ACLMessage used to create the content of all reply messages
        ACLMessage msg = request.createReply();
    }
}

```

```

    try {
        System.out.println("entrou no handlePublishBehaviour");
        // Get the requested action
        Action action = (Action)this.myAgent.getContentManager().extractContent(request);
        PublishNews publishNews = (PublishNews) action.getAction();
        News news = publishNews.getNews();
        String AID_agent = this.myAgent.getAID().getName();
        String path = publishNews.getSrcXML();

        // Check if news is already published.
        // If it's true REFUSE and exit, else PUBLISH
        // Perform the publishing action
        int result = ((WebmasterAgent) this.myAgent).doPublishNews(news, AID_agent, path);

        // Reply according to the result
        if (result > 0){
            // OK --> INFORM action done
            Done d = new Done();
            d.setAction(action);
            this.myAgent.getContentManager().fillContent(msg, d);
            msg.setPerformative(ACLMessage.INFORM);
        } else{
            // NOT OK --> FAILURE
            ContentElementList l = new ContentElementList();
            l.add(action);
            l.add(new PublishNewsError());
            this.myAgent.getContentManager().fillContent(msg, l);
            msg.setPerformative(ACLMessage.FAILURE);
        }

    } catch (Exception fe){
        System.out.println(fe.getMessage().toString());
    }

    return msg;
} // END method prepareResultNotification

public ACLMessage prepareResponse(ACLMessage request) {
    // Prepare a dummy ACLMessage used to create the content of all reply messages
    ACLMessage temp = request.createReply();

    try {
        // Get the requested action.
        Action action = (Action)this.myAgent.getContentManager().extractContent(request);
        PublishNews publish = (PublishNews) action.getAction();
        News news = publish.getNews();
        String AID_agent = this.myAgent.getAID().getName();
        String path = publish.getSrcXML();

        // Check if the news is already published
        // If it's true REFUSE and exit, else AGREE
        if (!((WebmasterAgent) myAgent).isPublishedNews(news)) {
            System.out.println("AGREE");
            // AGREE to accomplish the publish action without any
            // special condition.
            ContentElementList l = new ContentElementList();
            l.add(action);
            l.add(new TrueProposition());
            this.myAgent.getContentManager().fillContent(temp, l);
            temp.setPerformative(ACLMessage.AGREE);
        } else {
            System.out.println("REFUSE");
            ContentElementList l = new ContentElementList();
            l.add(action);
            l.add(new NewsAlreadyPublished());
            this.myAgent.getContentManager().fillContent(temp, l);
            temp.setPerformative(ACLMessage.REFUSE);
        }

    } catch (Exception fe) {
        fe.printStackTrace();
        System.out.println(this.myAgent.getName() + ": Error handling the publish action.
2");
        System.out.println(fe.getMessage().toString());
    }
}

```

```

    return temp;
  } // END method prepareResponse
} // END class HandlePublishBehaviour

```

A implementação da ontologia pode ser encontrada no Apêndice B.

6.4.1 Estrutura geral do sistema

Conforme apresentado na Figura 6.16, o sistema pode ser dividido em quatro camadas principais: interface, comunicação, negócio e persistência. A camada responsável pela interface com o usuário (GUI) está sendo implementada utilizando as tecnologias JSP, HTML e XML (para armazenamento e exibição das notícias). A partir dessa camada, temos a exibição do conteúdo das notícias e a interface para administração do sistema. Os agentes são inicializados através da interface de administração, bem como a plataforma JADE (através do agente RMA), para que a troca de mensagens entre os agentes possa ser visualizada.

A camada de comunicação, por sua vez, está subdividida em 2 outras camadas de comunicação. A segunda camada tem como função servir de canal de interação entre a GUI e a aplicação em JADE. 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 submetê-lo ao sistema através de um *Servlet* [Deitel03]. Sua outra subcamada é definida e disponibilizada pela plataforma JADE e serve para realizar a comunicação entre os agentes que estiverem situados em cada um dos módulos da camada de negócios.

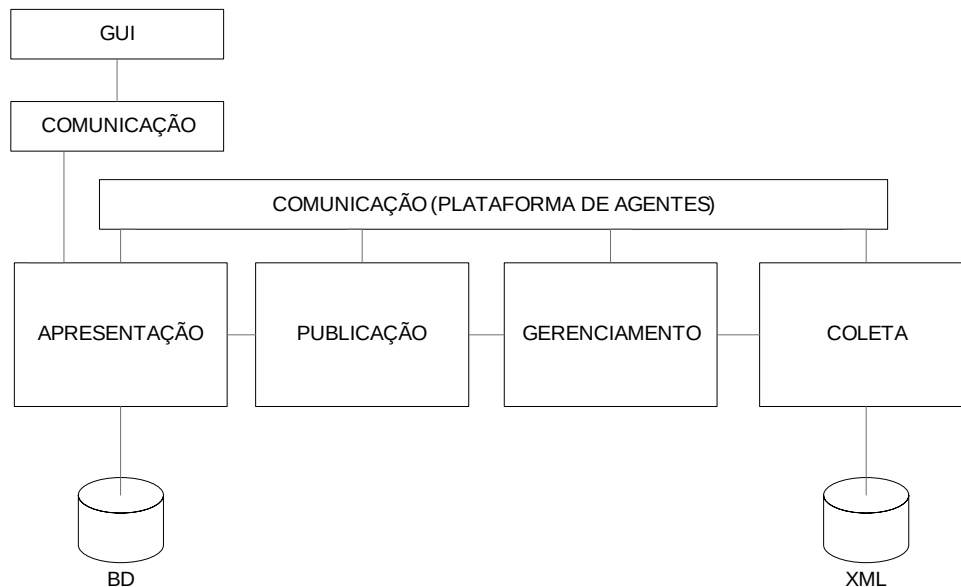


Figura 6.16: Visão arquitetural em camadas do *Smart Journal*

A camada de negócios pode ainda ser dividida conforme as funcionalidades identificadas na arquitetura do sistema resultante da fase de projeto detalhado de Tropos, apresentando quatro módulos diferentes: apresentação, publicação, gerenciamento e coleta. Além disso, nesta camada está

definida toda ontologia que será utilizada pelos agentes. Na camada de persistência estão sendo armazenados os arquivos com as notícias e aqueles que controlam o andamento das publicações.

Interfaces do sistema Smart Journal

O sistema deverá possuir duas interfaces principais, uma para disponibilização do conteúdo para usuários leitores do jornal e outra para administração e gerenciamento deste conteúdo. Na interface principal para o leitor, 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, além de um campo de login para a área restrita (administração do sistema).

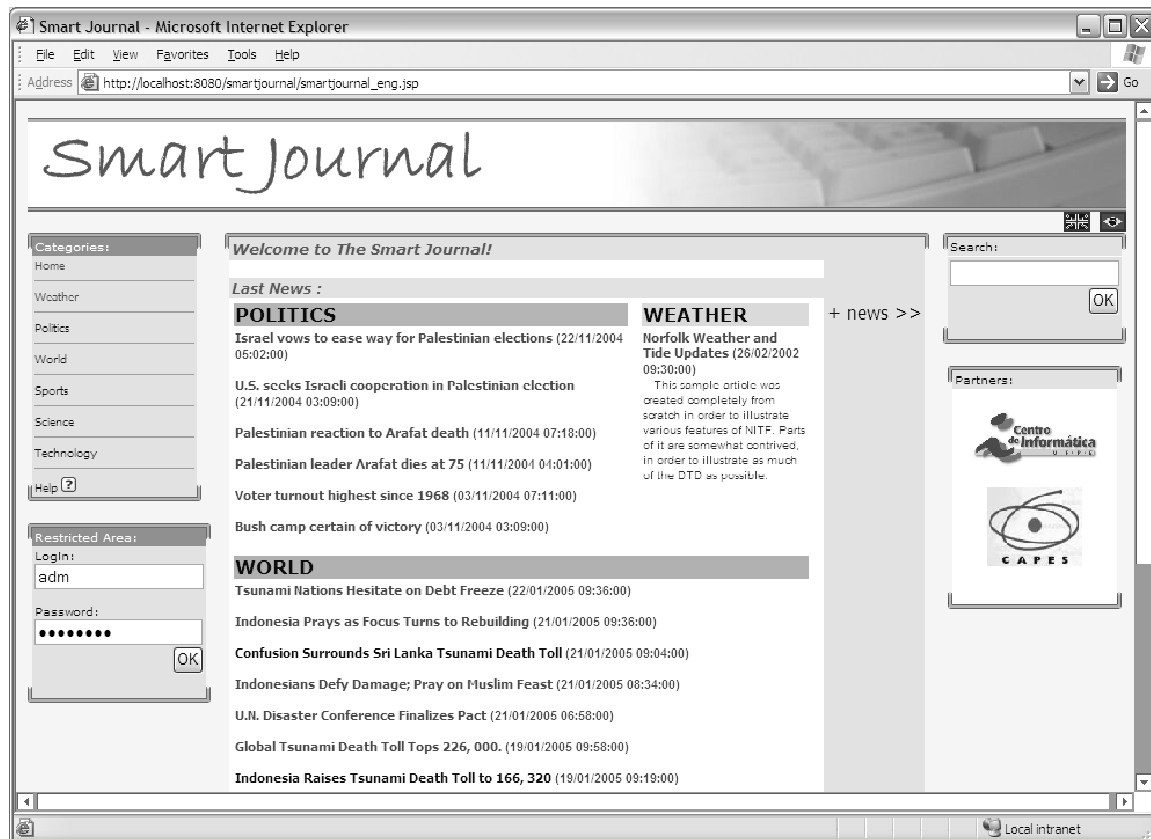


Figura 6.17: Tela principal do sistema *Smart Journal*

Toda gerência de conteúdo é realizada dentro da área restrita do sistema, onde o processo de simulação dos agentes e gerenciamento das publicações pode ser realizado. Neste módulo de interface são apresentados links para 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 será iniciada. As demais telas do sistema podem ser encontradas no Apêndice A.

Estruturação das notícias no *Smart Journal*

O conceito de notícia, além de estar representado como uma classe de conceitos na ontologia que foi definida em JADE, também está sendo armazenado pelo sistema como um arquivo XML seguindo a estrutura proposta pelo padrão NITF [IPTC04]. As notícias no *Smart Journal* são representadas em um arquivo XML, cuja estrutura simplificada é apresentada na

Tabela 6.3.

Tabela 6.3: Representação de uma notícia de acordo com 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>
```

A apresentação das notícias do sistema é realizada via navegador, uma vez que quando os agentes autorizam sua publicação, sua referência fica disponível na página principal ou na seção da qual a notícia faz parte. Assim, o arquivo XML que representa a notícia pode ser acessado e disponibilizado pelo sistema.

6.5 Considerações Finais

Este capítulo apresentou o estudo de caso produzido para esta dissertação. Toda modelagem utilizando a metodologia Tropos e as melhorias propostas no projeto detalhado foi realizada para o desenvolvimento de um sistema gerenciador de notícias denominado *Smart Journal*.

A plataforma JADE se mostrou eficiente para o desenvolvimento de sistemas modelados dentro do contexto da metodologia Tropos. Contudo, ainda encontramos algumas deficiências com relação à implementação. Um dos motivos para isso é a falta de outros estudos de caso que utilizem ambas as tecnologias. Apesar de toda bibliografia consultada, a experimentação realizada no estudo de caso

desta dissertação foi fundamental para compreensão dos mecanismos de funcionamento da plataforma JADE.

Uma vez que a abordagem que utilizamos foi de primeiramente implementar um protótipo do sistema *Smart Journal* na ferramenta JADE, para poder identificar as estruturas que precisavam ser fundamentalmente modeladas pelo projeto detalhado de Tropos, e só em seguida efetivamente realizar a modelagem do projeto detalhado, a evolução de nosso protótipo ainda está em andamento, principalmente para as camadas de interface com o usuário e funcionamento real na internet.

Capítulo

7 Conclusões e Trabalhos Futuros

Este capítulo relata as conclusões obtidas no desenvolvimento deste trabalho, assim como as principais contribuições que ele fornece para Engenharia de Software Orientada a Agentes. São apresentados também alguns trabalhos relacionados, bem como possíveis trabalhos futuros que podem ser realizados a partir deste. Por fim, apresentamos as considerações finais desta dissertação.

7.1 Principais Contribuições

Apresentamos nesta dissertação uma proposta de melhoria da fase de projeto detalhado da metodologia Tropos. O modelo proposto foi baseado nos resultados apresentados em [Kolp05], é concentrado em três dimensões de modelagem, e está direcionado para implementação de sistemas multi-agentes usando a plataforma de desenvolvimento JADE. Cada uma das dimensões apresenta um modelo básico e outro auxiliar para serem utilizados no projeto detalhado de sistemas orientados a agentes. As dimensões de modelagem que foram exploradas em nossa proposta estão preocupadas com características estruturais, comunicativas e dinâmicas.

A **dimensão estrutural** está preocupada com o detalhamento dos atores que foram modelados nas fases anteriores de desenvolvimento de Tropos. Esta dimensão é modelada utilizando diagramas de classe da UML [Booch99, Lima05] estendidos para considerar as estruturas de programação suportadas por JADE. O projeto da dimensão estrutural do sistema está dividido em dois modelos: o **modelo de relacionamentos** e o **modelo de ontologias**. Através do primeiro modelo podemos representar os conceitos e relacionamentos fundamentais para construção da dimensão estrutural dos agentes. Nele podemos visualizar os relacionamentos entre agentes, comportamentos, mensagens e ontologias. Cada um desses conceitos define um *template* para classes de um SMA a ser implementado com JADE. Por sua vez, o modelo de ontologias para o projeto da dimensão estrutural foi proposto para auxiliar a modelagem da comunicação realizada através das interações entre os agentes. A principal intenção deste modelo é projetar uma ou mais ontologias a serem utilizadas dentro dos parâmetros das mensagens trocadas pelos agentes em suas interações, visando padronizar os vocabulários dos domínios de problema que os agentes tratam em seus ambientes de execução.

Devido à grande importância da comunicação inerentemente realizada no paradigma de orientação a agentes, onde os agentes interagem uns com os outros através de trocas de mensagens, também contribuímos com os modelos da **dimensão comunicativa**. Esta dimensão representa, de maneira temporal, os eventos relacionados a trocas de mensagens realizada no sistema. Os dois modelos desta dimensão são o **modelo de interações** e o **modelo de mensagens**. O modelo de interações está preocupado com os protocolos que serão utilizados nas conversações entre os agentes, bem como com a estrutura dessas interações. Utilizamos diagramas de sequência da UML

estendidos com novos estereótipos. O modelo de mensagens, entretanto, é um modelo auxiliar proposto nesta dissertação. Ele pode ser usado com o propósito de definir a composição detalhada de uma mensagem que pode ser enviada por agentes, levando em consideração a proposta de especificação de mensagens ACL da FIPA. Para cada evento representado em diagramas de seqüência do modelo de interações, podemos detalhar a mensagem que está sendo trocada utilizando o modelo de mensagens.

A **dimensão dinâmica** modela as capacidades e possíveis comportamentos necessários para que o agente realize suas tarefas e atinja seus objetivos. O **modelo de capacidades** da dimensão dinâmica serve para auxiliar a distribuição dos comportamentos através das capacidades de cada agente. Ele associa agentes a capacidades e capacidades a comportamentos. O último modelo proposto para ser utilizado no projeto detalhado de Tropos é o **modelo comportamental**. Sua principal função é projetar tarefas e intenções dos agentes que serão implementadas utilizando a estrutura de comportamentos oferecida por JADE. Nesta dimensão podemos utilizar tabelas para representar o modelo de capacidades e diagramas de atividades da UML para especificar os comportamentos dos agentes.

Além das melhorias propostas particularmente para fase de projeto detalhado de Tropos, esta dissertação fornece várias contribuições para a área de Engenharia de Software Orientado a Agentes. Dentre as contribuições mais significativas podemos citar:

- Desenvolvimento de um estudo de caso inédito dentro do projeto Tropos, desde os requisitos até a implementação, especialmente para pesquisa e validação de nossa proposta de dissertação. Este estudo de caso, devido à sua complexidade e potencial adequação ao paradigma de orientação a agentes, deverá ser aperfeiçoado e aproveitado em outros trabalhos do nosso grupo de pesquisa;
- Representação dos artefatos de projeto detalhado utilizando a linguagem UML [Lima05], com o auxílio de seus mecanismos de extensão. Desta forma foi possível prover uma semântica adequada para orientação a agentes, baseada na semântica de orientação a objetos que a UML provê e já é bem aceita na indústria;
- Investigação do desenvolvimento de sistemas orientados a agentes de acordo com a plataforma JADE [JADE05], seguindo as especificações da FIPA [FIPA04], extraindo elementos genéricos de abstração que são adotados por JADE e precisam ser levados em consideração pelas metodologias de desenvolvimento de SMAs.

7.2 Trabalhos Relacionados

Neste trabalho apresentamos a aplicação da metodologia Tropos e da plataforma de desenvolvimento JADE, no desenvolvimento de sistemas multi-agentes. Contudo, diversos trabalhos têm sido desenvolvidos para melhoria do projeto detalhado para implementação de sistemas orientados a agentes, envolvendo metodologias de desenvolvimento e plataformas de implementação.

O trabalho desenvolvido por [Moraitis04] integra a plataforma JADE e a metodologia baseada em agentes chamada Gaia [Wooldridge00], mostrando o mapeamento que pode ser realizado entre os conceitos básicos propostos por Gaia, para especificação e interações entre agentes, e os conceitos

providos por JADE para implementação de agentes. Esta proposta também sugere uma pequena modificação nos modelos de conhecimento de Gaia. Devido ao fato dos conceitos básicos de Gaia serem distintos dos conceitos de Tropos, este mapeamento não pode ser reusado de maneira direta para aplicação no trabalho de integração de Tropos com JADE.

A proposta apresentada em [Kolp05] conceitualiza um *framework* para explorar os padrões apresentados por [Kolp02], que apresenta um catálogo preliminar de padrões sociais. Em particular, cinco dimensões complementares são requeridas para descrever os modelos de projeto detalhados refletindo aspectos particulares das arquiteturas de SMAs, tais como as dimensões sociais, intencionais, estruturais, comunicativas e dinâmicas. As últimas três dimensões estão relacionadas com o projeto detalhado de agentes, sendo realizadas especificamente para padrões de projeto. Elas são descritas usando extensões de UML [Booch99] e as duas primeiras dimensões são descritas usando o *framework* i* [Yu95]. A arquitetura de agentes focada por este *framework* é a arquitetura deliberativa conhecida como BDI (*belief-desire-intention*) [Rao95]. O ambiente alvo escolhido para o desenvolvimento orientado a agentes é JACK [Coburn01], pelo fato deste ambiente dar suporte à implementação de arquiteturas BDI. Este trabalho também introduz a geração de código de determinadas especificações de agentes para a plataforma JACK. Entretanto, esta abordagem não apresenta um método explícito para aplicar os padrões durante o desenvolvimento de sistemas multi-agentes. Os padrões são aplicados de uma maneira *ad hoc* e suas escolhas não são claramente justificadas.

Outro trabalho relacionado importante é apresentado em [Silva05], que propõe um esboço de um processo que facilita a escolha e a aplicação de padrões sociais ao projeto arquitetural detalhado de SMA. Este processo adota a plataforma de desenvolvimento JADE pelo fato de que ela está habilitada para implementação de mais de um tipo de arquitetura de agentes. Embora JADE suporte uma codificação direta da arquitetura de agentes reativos, uma extensão para suportar a implementação da arquitetura de agentes BDI é proposta e é chamada de JADEX [Braubach04]. A abordagem deste trabalho também inclui um modelo para especificar os padrões sociais a fim de promover um melhor entendimento dos mesmos, este modelo visa dar suporte ao desenvolvimento de sistemas em JADE e é composto por três diagramas estendidos da UML para capturar, respectivamente, o comportamento, a estrutura e a colaboração de cada padrão social. O foco da proposta destes diagramas é a implementação em JADE visando a geração automática de código a partir do projeto detalhado dos padrões.

7.3 Trabalhos Futuros

Muito trabalho ainda há por ser feito na fase de projeto detalhado para prover uma especificação apropriada e suficiente de todas as características do projeto de SMA. Tais características incluem, por exemplo, a especificação de como os agentes de um SMA podem, entre outras coisas, mudar ou adaptar dinamicamente os papéis que eles assumem na sociedade de agentes. Ainda podemos mencionar como trabalhos futuros as seguintes atividades:

- Complementar e finalizar a implementação do SMA *Smart Journal*, corrigindo as deficiências encontradas durante o desenvolvimento dessa dissertação que não puderam se resolvidas por limitações de tempo;

- Propor diretrizes bem definidas para implementar sistemas em JADE a partir do projeto detalhado de Tropos, procurando diminuir a distância entre o projeto e a implementação de tais sistemas;
- Aperfeiçoar nossa proposta e aplicá-la na melhoria da descrição dos padrões sociais já existentes para o paradigma de orientação a agentes [Kolp05];
- Melhorar nossa proposta para incluir características de mobilidade dos agentes entre as plataformas, além de considerar com mais ênfase, características de interoperabilidade de agentes em sociedades diferentes;
- Abordar na fase de projeto detalhado a modelagem de características computacionais inteligentes refletidas nas arquiteturas internas dos agentes, ou seja, auxiliar a modelagem de agentes BDI, de aprendizagem, deliberativos ou otimizadores, entre outras arquiteturas de agentes [Russel02];
- Acrescentar em nossa proposta a modelagem em nível de projeto detalhado de características advindas dos atributos de qualidade utilizados para escolha da arquitetura, como por exemplo negociação, coordenação e cooperação;
- Especificar os agentes do sistema multi-agentes *Smart Journal* usando a arquitetura BDI (Belief-Desire-Intention) [Rao95] e implementá-los em JADEX [Braubach04];
- Desenvolver uma ferramenta CASE para suportar tanto a especificação de agentes como também a sua posterior geração de código em JADE [Bellifemine05] e JADEX;
- Desenvolver outros estudos de caso, como por exemplo um sistema de gerenciamento de conferência [Zambonelli03]. Além disso, pretendemos validar nossa proposta em um projeto de grande porte, preferivelmente uma aplicação industrial, que tenha sociedades de agentes maiores e mais complexas;

7.4 Considerações Finais

Uma abordagem orientada a agentes para modelar o mundo deve ser capaz de capturar a complexidade de aspectos sociais não considerada pelas metodologias de desenvolvimento de software tradicionais, objetivando conduzir mais naturalmente a análise, projeto e implementação de sistemas orientados a agentes.

O desenvolvimento do Tropos é um importante passo para viabilizar a adoção do novo paradigma de desenvolvimento orientado a agentes. Particularmente, as melhorias contínuas de Tropos permitirão que se utilizem práticas mais sofisticadas da engenharia de software, que infelizmente não têm sido adotadas pela comunidade orientada a agente. Portanto, estaremos contribuindo para uma grande melhoria da qualidade do desenvolvimento e a conseqüente redução dos custos das aplicações orientadas a agentes. Desta forma, esperamos evoluir o *framework* Tropos de forma que ele permita que os projetistas de SMAs alcancem os benefícios prometidos por este paradigma.

Muito em breve Tropos oferecerá uma metodologia confiável e completa, na qual modelos, métodos e ferramentas CASE suportarão as várias fases do ciclo de desenvolvimento orientado a agentes. Esperamos que Tropos se torne um processo popular e largamente aplicado na indústria de sistemas multi-agentes. Através dos novos mecanismos de estruturação, tornaremos os modelos do Tropos mais simples e fáceis de entender quando aplicados a domínios complexos.

Com as nossas contribuições, objetivamos que projetistas de sistemas multi-agentes possam construir software de forma totalmente sistemática, de modo que os modelos baseados em agentes sejam efetivamente utilizados para dar suporte ao desenvolvimento do sistema durante todo o seu ciclo de vida e evolução.

8 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).
- [Bastos04] Bastos, L. R. D., Castro, J. B.: **Systematic Integration between Requirements and Architecture**. In: Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications ed. : Springer Verlag: LNCS, 2004, v.0, p. 1-18.
- [Bauer01] Bauer, B., Müller, J. P., and Odell, J.: **Agent UML: A formalism for specifying multiagent software systems**. International Journal of Software Engineering and Knowledge Engineering, vol. 11, no. 3, pp. 207–230, 2001.
- [Bellifemine04] Bellifemine, F., Caire, G., Poggi, A., Rimassa, G.: **JADE - A White Paper**. Special issue on JADE of the TILAB Journal "EXP - in search of innovation", (2003).
- [Bellifemine05] Bellifemine, F., Caire, G., Trucco, T., Rimassa, G.: **Jade Programmer's Guide**. JADE 3.3. <http://jade.tilab.com/doc/programmersguide.pdf>, (2005).
- [Bergenti00] Bergenti, F., Poggi, A.: **Exploiting UML in the Design of Multi-Agent Systems**. In Engineering Societies in the Agent World, First International Workshop - ESAW2000 (106-113), Berlin, Germany, August 2000.
- [Braubach04] Braubach, L., Pokahr, A. and Lamersdorf, W.: **Jadex: A Short Overview**. In: Main Conference Net.ObjectDays 2004, AgentExpo, (2004).
- [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.
- [Booch94] Booch, G.: **Object-Oriented Analysis and Design** (second edition). Addison-Wesley: Reading, MA, 1994.
- [Booch99] Booch, G., Rumbaugh, J., Jacobson, I.: **The Unified Modeling Language User Guide**, The Addison Wesley Object Technology Series, Addison Wesley, (1999).
- [Borland05] Borland Software Corporation. **Borland Together**. Informações no site: (<http://www.borland.com/together/>) - Consultado em 5/1/2005.
- [Castro01a] Castro, J. F. B., Mylopoulos, J., Kolp, M.: **Developing Agent-Oriented Information Systems for the Enterprise**. In: Enterprise Information Systems II ed. Dordrecht : Kluwer Academic Publishers, 2001.
- [Castro01b] Castro, J. F. B., Alencar, F. M. R., Cysneiro Filho, G. A. A., Mylopoulos, J.: **Integrating Organizational Requirements and Object Oriented Modeling**. In: 5th IEEE International Symposium on Requirements Engineering, 2001, Toronto. Proceeding of 5th IEEE International Symposium on Requirements Engineering. IEEE Press, 2001. p.146 – 153
- [Castro02a] Castro, J. Kolp, M., Mylopoulos, J.: **Towards Requirements-Driven Information Systems Engineering: The Tropos Project**. Information Systems Journal, 27: 365-89. 2002.

- [Castro02b] Castro, J. F. B., Paim, F. R.: **Enhancing Data Warehouse Quality with the NFR Framework**. In: V Workshop on Requirements Engineering, 2002, Valencia. Proceedings. Valencia: Universidad Politecnica de Valencia, 2002.
- [Castro02c] Castro, J. F. B., Reis, T.: **Medição da qualidade de aplicações Web na fase de requisitos**. In: XVI Brazilian Symposium on Software Engineering SBES2002, 2002, Gramado. Anais do XVI Brazilian Symposium on Software Engineering SBES2002. SBC, 2002. p.162 – 174
- [Chung00] Chung, L. K., Nixon, B. A., Yu, E., Mylopoulos, J.: **Non-Functional Requirements in Software Engineering**. Kluwer Publishing, (2000).
- [Cimatti00] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri. M.: **NuSMV: A New Symbolic Model Checker**. International Journal on Software Tools for Technology Transfer, 2(4):410--425, 2000.
- [Clarke99] Clarke, E., Grumberg, O., and Peled, D.: **Model Checking**. MIT Press, 1999.
- [Coburn01] Coburn, M.: **Jack intelligent agents: User guide version 2.0**. At <http://www.agent-software.com>, 2001.
- [Collis99] Collis, J. and Ndumu, D.: **Zeus Technical Manual**. Intelligent Systems Research Group, BT Labs, British Telecommunications, (1999).
- [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.
- [Dardenne93] Dardenne, A., Lamsweerde, A. Van, and Fickas, S.: **Goal-directed requirements acquisition**. Science of Computer Programming, 20(1–2): 3–50, 1993
- [Deitel03] Deitel, H. M. & Deitel, P. J.: **Java: Como Programar**. 4ª ed., Bookman, (2003).
- [Do03] Do T. T., Kolp M. & Pirotte A.: **Social patterns for designing multi-agent systems**. In Proc. of the 15th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE'03, (2003).
- [Do04] Do T. T., Kolp M. and Faulkner, S.: **Agent Oriented Design Patterns: The SKwyRL Perspective**. In Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS 2004), Porto, Portugal, April 2004.
- [Figueira00] Figueira Filho, C. and Ramalho, G.: **Jeops – the Java Embedded Object Production System**. In M. Monard e J. Sichman (eds.). LNAI Series, vol. 1952, pp 52-61. London: Springer-Verlag, 2000.
- [Finin97] Finin, T., Labrou, Y., and Mayfield, J.: **KQML as an Agent Communication Language**. In ``Software Agents'', Jeffrey Bradshaw (editor), AAAI/MIT Press, 1997.
- [FIPA00] FIPA specification XC00061: **FIPA ACL Message Structure Specification**. In: <http://www.fipa.org>, (2000).
- [FIPA04] **FIPA (The Foundation for Intelligent Agents)**, Available: <http://www.fipa.org>, 2004.
- [FIPA-OS05] FIPA-OS: **A component-based toolkit enabling rapid development of FIPA compliant agents**. In: <http://fipa-os.sourceforge.net/>, (2005).
- [Fuxman01] Fuxman, A., Pistore, M., Mylopoulos, J., and Traverso, P.: **Model checking early requirements specification in Tropos**. In Proc. of the 5th Int. Symposium on Requirements Engineering, RE'01, Toronto, Canada, 2001.
- [Gamma95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1995.
- [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).
- [Gomes96] Gomes-Casseres, B.: **The alliance revolution: the new shape of business rivalry**. Harvard University Press, (1996).
- [GNU99] The GNU Project. **The GNU General Public License**. At <http://www.gnu.org/copyleft/lesser.html> (1999).
- [Hayden99] Hayden S., Carrick C. and Yang Q.: **Architectural design patterns for multiagent coordination**. In: Proceedings of the 3rd International Conference on Autonomous Agents, Agents'99, Seattle, USA, (1999).
- [IPTC04] IPTC. **NITF: News Industry Text Format**. At <http://www.nitf.org/> (2004).
- [JADE05] **JADE: Java Agent DEvelopment Framework**. At <http://jade.tilab.com/> (2005).
- [JACK05] **JACK Intelligent Agents**. At <http://www.agent-software.com/> (2005).
- [Jennings00] Jennings, N.: **On Agent-Based Software Engineering**. In Bradshaw, J. (ed): Handbook of Agent Technology, AAAI/MIT Press, (2000).
- [Jennings03] Jennings, N. R. **Applying Agent Technology**. Proceedings of the 5th European Agent Systems Spring School. EASSS 2003. Barcelona, Espanha, Fevereiro 2003.
- [Kiczales97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J.: **Aspect-Oriented Programming**. In M. Aksit and S Mat-suoka, editors, European Conference on Object-Oriented Programming, volume 1241 of LNCS, pages 220-242. Springer Verlag, 1997.
- [Kolp01a] 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)
- [Kolp01b] Kolp, M., Giorgini, P., Mylopoulos, J.: **A goal-based organizational perspective on multi-agents architectures**. Proc. of the 8th Int. Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages. ATAL'01, Seattle, USA (2001).
- [Kolp02] Kolp, M. Giorgini, P., Mylopoulos, J.: **Information Systems Development through Social Structures**. 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ishia, 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.
- [Labrou99] Labrou, Y., Finin, T., and Pen Y.: **Agent Communication Languages: The Current Landscape**, Intelligent Systems, Vol. 14, No. 2, March/April 1999, IEEE Computer Society.
- [Lima05] Lima, A. S. **UML 2.0 – Do Requisito à Solução**. 1ª ed., Érica, (2005).
- [Lucena04] Lucena, C., Garcia, A., Romanovsky, A., Castro, J., Alencar, P. (editors). Book: **Software Engineering for Multi-Agent Systems II: Research Issues and Practical Applications**. Lecture Notes in Computer Science - LNCS Vol. 2940, State-of-the-Art Survey . 278 pages. 2004

- [Luck03] Luck, M., McBurney, P., Preist, C.: **Agent Technology: Enabling Next Generation Computing**. AgentLink (2003), ISBN 0854 327886. 94 pages (visit <http://www.agentlink.org/roadmap>).
- [Morabito99] Morabito, J., Sack, I. Bhate, A.: **Organizational modeling: inovative architectures for the 21st century**. Prentice Hall, (1999).
- [Moraitis04] Moraitis, P., Spanoudakis N.: **Combining Gaia and JADE for Multi-Agent Systems Development**. Fourth International Symposium "From Agent Theory to Agent Implementation" (AT2AI'04), Vienna, Austria, (2004).
- [Mylopoulos01] Mylopoulos, J., Kolp, M., and Castro, J.: **UML for Agent-Oriented Software Development: the Tropos Proposal**. In Proceedings of the Fourth International Conference on the Unified Modeling Language (<<UML>> 2001), Toronto, Canada, (2001).
- [Newell93] Newell, A.: **Reflections on the Knowledge Level**. Artificial Intelligence, 59:31--38, 1993.
- [Noy2001] Noy, N., Sintek, M., Decker, S., Crubezy, M., Fergerson, R., and Musen, M.: **Creating Semantic Web Contents with Protege-2000**. IEEE Intelligent Systems, 16 (2), 2001, pp. 60-71.
- [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 17th 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).
- [Pedroza04] Pedroza, F. P., Alencar, F. M. R., Castro, J. F. B., Silva, F. R. C., Santander, V. F. A.: **Ferramentas para Suporte do Mapeamento da Modelagem i* para a UML: eXtended GOOD XGOOD e GOOSE**. In: VII Workshop on Requirements Engineering - WER04, 2004, Tandil. Proceedings of the VII Workshop on Requirements Engineering - WER 04, 2004. p. 164-175.
- [Pedroza05] Pedroza, F. P.: **Automatizando as Regras de Mapeamento entre a Modelagem i* e a Modelagem UML usando XMI para Implementação de um Simulador de Rede Ópticas**. Dissertação de Mestrado. Departamento de Engenharia Elétrica, Universidade Federal de Pernambuco, Fevereiro, 2005.
- [Pressman00] Pressman, R. **Software Engineering: A Practioner's Approach**. McGraw-Hill, 2000.
- [Rao95] Rao, A. S. and Georgeff, M. P.: **BDI agents: from theory to practice**, Technical Note 56, Australian Artificial Intelligence Institute, 1995.
- [Rational99] Rational Software Corp. et al.: **Unified Modelling Languages Semantic, version 1.3**. June 1999. Available at: www.rational.com/uml/index.itmpl
- [Reticular99] Reticular Systems Inc: **AgentBuilder: An Integrated Toolkit for Constructing Intelligent Software Agents**. In: <http://www.agentbuilder.com>, (1999).
- [Russel02] Russel, S. and Norvig, P. **Artificial Intelligence: A Modern Approach** (2nd Edition), 2002, Prentice-Hall.
- [Santander02] Santander, V. F. A., Castro, J. F. B.: **Deriving Use Cases from Organizational Modeling**. In: IEEE JOINT CONFERENCE ON REQUIREMENTS ENGINEERING - RE02, 2002, Essen, Germany. Proceegings of the IEEE JOINT CONFERENCE ON REQUIREMENTS ENGINEERING - RE02. Los Alamitos, California, USA : IEEE, 2002. v. 1. p. 32-39.

- [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.
- [Silva04] Silva, C., Tedesco, P., Castro, J. and Pinto, R.: **Comparing Agent-Oriented Methodologies Using NFR Framework**. Proceedings of SELMAS 2004 - 3rd International Workshop on Software Engineering For Large-Scale Multi-Agent Systems, Edinburgh, (2004).
- [Silva05] Silva, C., Castro, J., Tedesco, P. and Silva, I.: **Describing Agent-Oriented Design Patterns in Tropos**. In Proceedings of the 19th Brazilian Symposium on Software Engineering, Minas Gerais, Brazil (2005). (To appear)
- [Scott98] Scott, W. R.: **Organizations: rational, natural, and open systems**. Prentice Hall, (1998).
- [Sommerville01] Sommerville, I.: **Software Engineering**, Ed.6. Addison Wesley (2001).
- [TILAB05] **TILAB (Telecom Italia Lab)**, Available: <http://www.telecomitalialab.com/>, 2004.
- [Tveit01] Tveit, A.: **A survey of Agent-Oriented Software Engineering**, First NTNU CSGSC, May 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.
- [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).
- [Wooldridge01] Wooldridge, M., Weiss, G., Ciancarini, P. editors: **Agent-Oriented Software Engineering II**. Springer-Verlag Lecture Notes in Computer Science Volume 2222, February 2001.
- [Wooldridge02] Wooldridge, M.: **Introduction to Multi-Agent Systems**. John Wiley and Sons, New York (2002).
- [Yoshino95] Yoshino, M. Y., Rangan, U. S.: **Strategic Alliances: An Entrepreneurial Approach to Globalization**. Harvard Business School Press, (1995).
- [Yu95] Yu, E.: **Modelling Strategic Relationships for Business Process Reengineering**, Ph.D. thesis. Dept. of Computer Science, University of Toronto (1995).
- [Yu01] E. Yu.: **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)**, Project Home Page, <http://www.cs.toronto.edu/km/ome>.
- [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.

Apêndice A

A seguir encontram-se os demais artefatos que foram modelados mas não fazem parte da metodologia Tropos em sua versão atual.

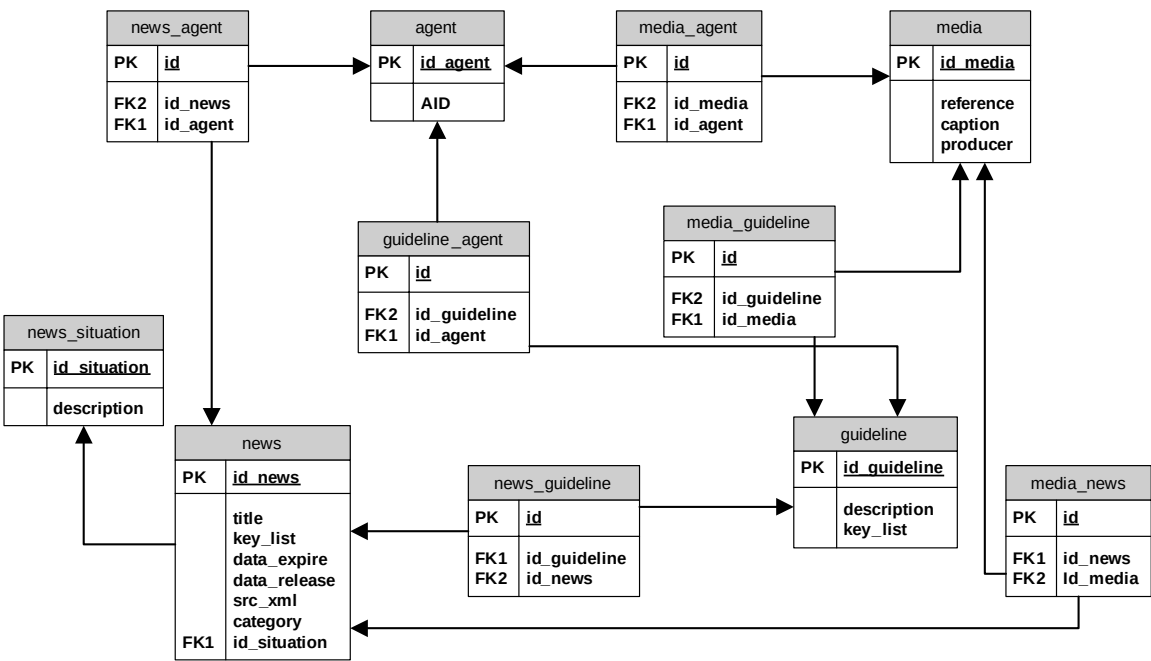


Figura 8.1: Modelo Entidade-Relacionamento do banco de dados definido no *Smart Journal*

Apêndice B

Este apêndice apresenta os códigos em Java das classes do Smart Journal, utilizando a API de JADE. Também são apresentados aqui alguns diagramas correspondentes a esses códigos.

Tabela 8.1: Código em Java da classe *smagnw.ontology.NewsOntology*

```
package smagnw.ontology;

import jade.content.onto.*;
import jade.content.schema.*;
import smagnw.ontology.util.*;
import smagnw.ontology.concept.*;
import smagnw.ontology.predicate.*;
import smagnw.ontology.agentAction.*;

// This class describes the ontology for the news in a context of an on-line journal,
// according to the NITF

public class NewsOntology extends Ontology {

    // ONTOLOGY NAME IDENTIFYING
    public static final String ONTOLOGY_NAME = "news-ontology";

    // Concepts
    public static final String GUIDELINE = "GUIDELINE";
    public static final String GUIDELINE_DESCRIPTION = "description";
    public static final String GUIDELINE_KEY_LIST = "keylist";
    public static final String GUIDELINE_DATE_ISSUE = "dateIssue";
    public static final String GUIDELINE_DATE_EXPIRE = "dateExpire";
    public static final String GUIDELINE_OWNER = "owner";

    public static final String NEWS = "NEWS";
    public static final String NEWS_NEWSHEAD = "newsHead";
    public static final String NEWS_BODY = "body";
    public static final String NEWS_CATEGORY = "category";
    public static final String NEWS_SITUATION = "situation";
    public static final String NEWS_SRCXML = "srcXml";

    public static final String NEWSHEAD = "NEWSHEAD";
    public static final String NEWSHEAD_TITLE = "title";
    public static final String NEWSHEAD_DOC_DATA = "docData";

    public static final String KEYLIST = "KEYLIST";
    public static final String KEYLIST_KEYWORD = "keywords";

    public static final String BODY = "BODY";
    public static final String BODY_BODYHEAD = "bodyHead";
    public static final String BODY_CONTENT = "content";
    public static final String BODY_END = "end";

    public static final String CONTENT = "CONTENT";
    public static final String CONTENT_MEDIA = "media";
```

```

public static final String CONTENT_PARAGRAPH = "paragraph";
public static final String CONTENT_BLOCK = "block";

public static final String MEDIA = "MEDIA";
public static final String MEDIA_REFERENCE = "reference";
public static final String MEDIA_CAPTION = "caption";
public static final String MEDIA_PRODUCER = "producer";
...

// Actions
public static final String PUBLISHNEWS = "PUBLISHNEWS";
public static final String PUBLISH_NEWS = "news";
public static final String PUBLISH_SRC_XML = "srcXML";

public static final String REMOVENEWS = "REMOVENEWS";
public static final String REMOVE_NEWS = "news";

public static final String SENDGUIDELINESTOREPORTER = "SENDGUIDELINESTOREPORTER";
public static final String SEND_GUIDELINES_TO_REPORTER_GUIDELINE = "guideline";
...

// Predicates
public static final String BELONGS_CATEGORY = "BELONGS-CATEGORY";
public static final String BELONGS_CATEGORY_NAME = "categoryName";
public static final String BELONGS_CATEGORY_NEWS = "categoryNews";
public static final String BELONGS_CATEGORY_ERROR = "DOES-NOT-BELONG-CATEGORY";

public static final String IS_PUBLISHED_NEWS = "IS-PUBLISHED-NEWS";
public static final String IS_PUBLISHED_NEWS_NEWS = "news";
public static final String IS_PUBLISHED_NEWS_ERROR = "IS-NOT-PUBLISHED-NEWS";

public static final String NEWS_WITH_EXPIRE_DATE = "NEWS-WITH-EXPIRE-DATE";
public static final String NEWS_WITH_EXPIRE_DATE_NEWS = "news";
public static final String NEWS_WITH_EXPIRE_DATE_DATE = "dateExpire";
public static final String NEWS_WITH_EXPIRE_DATE_ERROR = "VALID-NEWS";

public static final String NEWS_ALREADY_PUBLISHED = "NEWS-ALREADY-PUBLISHED";
public static final String PUBLISH_NEWS_ERROR = "PUBLISH-NEWS-ERROR";
public static final String REMOVE_NEWS_ERROR = "REMOVE-NEWS-ERROR";
public static final String SENT_NEWS_ERROR = "SENT-NEWS-ERROR";
public static final String SENT_PHOTOS_ERROR = "SENT-PHOTOS-ERROR";
public static final String SENT_GUIDELINES_ERROR = "SENT-GUIDELINES-ERROR";
//public static final String SENT_WARNING_ERROR = "SENT-WARNING-ERROR";
...

// The singleton instance of this ontology
private static Ontology theInstance = new NewsOntology();

public static Ontology getInstance() {
    return theInstance;
} // end getInstance()

// Private Constructor
private NewsOntology() {

    // The NewsOntology extends the basic ontology
    super(ONTOLOGY_NAME, BasicOntology.getInstance());

    // Building schemas

```

```

try {

    add(new ConceptSchema(GUIDELINE), Guideline.class);
    add(new ConceptSchema(NEWS), News.class);

    // Heading's News
    add(new ConceptSchema(NEWSHEAD), NewsHead.class);
    add(new ConceptSchema(KEYLIST), Keylist.class);

...

    // Body's News
    add(new ConceptSchema(BODY), Body.class);
    add(new ConceptSchema(CONTENT), Content.class);
    add(new ConceptSchema(MEDIA), Media.class);

...

    // Predicates
    add(new PredicateSchema(BELONGS_CATEGORY), BelongsCategory.class);
    add(new PredicateSchema(BELONGS_CATEGORY_ERROR), CategoryError.class);
    add(new PredicateSchema(IS_PUBLISHED_NEWS), IsPublishedNews.class);
    add(new PredicateSchema(IS_PUBLISHED_NEWS_ERROR), IsPublishedNewsError.class);
    add(new PredicateSchema(NEWS_WITH_EXPIRE_DATE), NewsWithExpireDate.class);
    add(new PredicateSchema(NEWS_WITH_EXPIRE_DATE_ERROR),
NewsWithExpireDateError.class);
    add(new PredicateSchema(NEWS_ALREADY_PUBLISHED), NewsAlreadyPublished.class);
    add(new PredicateSchema(PUBLISH_NEWS_ERROR), PublishNewsError.class);
    add(new PredicateSchema(REMOVE_NEWS_ERROR), RemoveNewsError.class);
    add(new PredicateSchema(SENT_NEWS_ERROR), SendNewsError.class);
    add(new PredicateSchema(SENT_GUIDELINES_ERROR), SentGuidelinesError.class);

...

    // Agent Actions
    add(new AgentActionSchema(PUBLISHNEWS), PublishNews.class);
    add(new AgentActionSchema(REMOVENEWS), RemoveNews.class);
    add(new AgentActionSchema(SENDNEWSTOEDITORIAL), SendNewsToEditorial.class);
    add(new AgentActionSchema(SENDGUIDELINESTOREPORTER),
SendGuidelinesToReporter.class);

...

    /** Concept Schemas */

    // Structure of the schema for the GUIDELINE concept

    ConceptSchema cs = (ConceptSchema) getSchema(GUIDELINE);
    cs.add(GUIDELINE_DESCRIPTION, (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(GUIDELINE_KEY_LIST, (ConceptSchema) getSchema(KEYLIST));
    cs.add(GUIDELINE_DATE_ISSUE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(GUIDELINE_DATE_EXPIRE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(GUIDELINE_OWNER, (PrimitiveSchema) getSchema(BasicOntology.STRING));

    // Structure of the schema for the NEWS concept
    cs = (ConceptSchema) getSchema(NEWS);
    cs.add(NEWS_NEWSHEAD, (ConceptSchema) getSchema(NEWSHEAD));
    cs.add(NEWS_BODY, (ConceptSchema) getSchema(BODY));
    cs.add(NEWS_CATEGORY, (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(NEWS_SITUATION, (PrimitiveSchema) getSchema(BasicOntology.INTEGER));
    cs.add(NEWS_SRCXML, (PrimitiveSchema) getSchema(BasicOntology.STRING));

    // Structure of the schema for the NEWSHEAD concept
    cs = (ConceptSchema) getSchema(NEWSHEAD);
    cs.add(NEWSHEAD_TITLE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
    cs.add(NEWSHEAD_DOC_DATA, (ConceptSchema) getSchema(DOCDATA));

```

```

// Structure of the schema for the KEYLIST concept
cs = (ConceptSchema) getSchema(KEYLIST);
cs.add(KEYLIST_KEYWORD, (PrimitiveSchema) getSchema(BasicOntology.STRING));

// Structure of the schema for the BODY concept
cs = (ConceptSchema) getSchema(BODY);
cs.add(BODY_BODYHEAD, (ConceptSchema) getSchema(BODYHEAD));
cs.add(BODY_CONTENT, (ConceptSchema) getSchema(CONTENT));
cs.add(BODY_END, (ConceptSchema) getSchema(END));

// Structure of the schema for the HEAD concept
cs = (ConceptSchema) getSchema(BODYHEAD);
cs.add(BODYHEAD_HEADLINE, (ConceptSchema) getSchema(HEADLINE));
cs.add(BODYHEAD_NOTE, (ConceptSchema) getSchema(NOTE));
cs.add(BODYHEAD_BY_LINE, (ConceptSchema) getSchema(BYLINE));
cs.add(BODYHEAD_DISTRIBUTOR, (ConceptSchema) getSchema(DISTRIBUTOR));
cs.add(BODYHEAD_DATE_LINE, (ConceptSchema) getSchema(DATELINE));
cs.add(BODYHEAD_ABSTRACT, (PrimitiveSchema) getSchema(BasicOntology.STRING));

// Structure of the schema for the CONTENT concept
cs = (ConceptSchema) getSchema(CONTENT);
cs.add(CONTENT_MEDIA, (ConceptSchema) getSchema(MEDIA));
cs.add(CONTENT_PARAGRAPH, (PrimitiveSchema) getSchema(BasicOntology.STRING));
cs.add(CONTENT_BLOCK, (ConceptSchema) getSchema(BLOCK));

// Structure of the schema for the MEDIA concept
cs = (ConceptSchema) getSchema(MEDIA);
cs.add(MEDIA_REFERENCE, (PrimitiveSchema) getSchema(BasicOntology.STRING));
cs.add(MEDIA_CAPTION, (PrimitiveSchema) getSchema(BasicOntology.STRING));
cs.add(MEDIA_PRODUCER, (PrimitiveSchema) getSchema(BasicOntology.STRING));

...

/** Predicate Schemas */

PredicateSchema ps = (PredicateSchema) getSchema(BELONGS_CATEGORY);
ps.add(BELONGS_CATEGORY_NAME, (PrimitiveSchema) getSchema(BasicOntology.STRING));
ps.add(BELONGS_CATEGORY_NEWS, (ConceptSchema) getSchema(NEWS));

ps = (PredicateSchema) getSchema(IS_PUBLISHED_NEWS);
ps.add(IS_PUBLISHED_NEWS_NEWS, (ConceptSchema) getSchema(NEWS));

ps = (PredicateSchema) getSchema(NEWS_WITH_EXPIRE_DATE);
ps.add(NEWS_WITH_EXPIRE_DATE_NEWS, (ConceptSchema) getSchema(NEWS));
ps.add(NEWS_WITH_EXPIRE_DATE_DATE,
(PrimitiveSchema) getSchema(BasicOntology.STRING));

...

/** Agent Action Schemas */

AgentActionSchema as = (AgentActionSchema) getSchema(PUBLISHNEWS);
as.add(PUBLISH_NEWS, (ConceptSchema) getSchema(NEWS));
as.add(PUBLISH_SRC_XML, (PrimitiveSchema) getSchema(BasicOntology.STRING));

as = (AgentActionSchema) getSchema(REMOVE_NEWS);
as.add(REMOVE_NEWS, (ConceptSchema) getSchema(NEWS));

as = (AgentActionSchema) getSchema(SEND_NEWS_TO_EDITORIAL);
as.add(SEND_NEWS_TO_EDITORIAL_NEWS, (ConceptSchema) getSchema(NEWS));
as.add(SEND_NEWS_TO_EDITORIAL_SRC_XML,
(PrimitiveSchema) getSchema(BasicOntology.STRING));
as.add(SEND_NEWS_TO_EDITORIAL_COD_GUIDELINE,

```

```
(PrimitiveSchema) getSchema (BasicOntology.INTEGER));

    as = (AgentActionSchema) getSchema (SENDPHOTOSTOEDITORIAL);
    as.add (SEND_PHOTOS_TO_EDITORIAL_PHOTOS, (ConceptSchema) getSchema (MEDIA));
    as.add (SEND_PHOTOS_TO_EDITORIAL_NEWS,
(PrimitiveSchema) getSchema (BasicOntology.INTEGER));
    as.add (SEND_PHOTOS_TO_EDITORIAL_GUIDELINE,
{PrimitiveSchema) getSchema (BasicOntology.INTEGER));

    as = (AgentActionSchema) getSchema (SENDGUIDELINESTOREPORTER);
    as.add (SEND_GUIDELINES_TO_REPORTER_GUIDELINE, (ConceptSchema) getSchema (GUIDELINE));

    as = (AgentActionSchema) getSchema (SENDGUIDELINESTOPHOTOGRAPHER);
    as.add (SEND_GUIDELINES_TO_PHOTOGRAPHER_GUIDELINE,
(ConceptSchema) getSchema (GUIDELINE));

    } catch (OntologyException oe) {
        oe.printStackTrace();
    }

    } // end private constructor

} // end class NewsOntology
```

Figura 8.2: Diagrama de ontologias simplificado para classe NewsOntology

<<ontology>> NewsOntology
<pre> + ONTOLOGY_NAME : String = "news-ontology" + GUIDELINE : String = "GUIDELINE" + GUIDELINE_DESCRIPTION : String = "description" + GUIDELINE_KEY_LIST : String = "keylist" + GUIDELINE_DATE_ISSUE : String = "dateIssue" + GUIDELINE_DATE_EXPIRE : String = "dateExpire" + GUIDELINE_OWNER : String = "owner" + NEWS : String = "NEWS" + NEWS_NEWSHEAD : String = "newsHead" + NEWS_BODY : String = "body" + NEWS_CATEGORY : String = "category" + NEWS_SITUATION : String = "situation" + NEWS_SRCXML : String = "srcXml" + NEWSHEAD : String = "NEWSHEAD" + NEWSHEAD_TITLE : String = "title" + NEWSHEAD_DOC_DATA : String = "docData" + KEYLIST : String = "KEYLIST" + KEYLIST_KEYWORD : String = "keywords" + BODY : String = "BODY" + BODY_BODYHEAD : String = "bodyHead" + BODY_CONTENT : String = "content" + BODY_END : String = "end" + CONTENT : String = "CONTENT" + CONTENT_MEDIA : String = "media" + CONTENT_PARAGRAPH : String = "paragraph" + CONTENT_BLOCK : String = "block" + MEDIA : String = "MEDIA" + MEDIA_REFERENCE : String = "reference" + MEDIA_CAPTION : String = "caption" + MEDIA_PRODUCER : String = "producer" + PUBLISHNEWS : String = "PUBLISHNEWS" + PUBLISH_NEWS : String = "news" + PUBLISH_SRC_XML : String = "srcXML" + REMOVENEWS : String = "REMOVENEWS" + REMOVE_NEWS : String = "news" + SENDNEWSTOEDITORIAL : String = "SENDNEWSTOEDITORIAL" + SEND_NEWS_TO_EDITORIAL_NEWS : String = "news" + SEND_NEWS_TO_EDITORIAL_SRC_XML : String = "srcXML" + SEND_NEWS_TO_EDITORIAL_COD_GUIDELINE : String = "codGuideline" + SENDGUIDELINESTOREPORTER : String = "SENDGUIDELINESTOREPORTER" + SEND_GUIDELINES_TO_REPORTER_GUIDELINE : String = "guideline" + BELONGS_CATEGORY : String = "BELONGS-CATEGORY" + BELONGS_CATEGORY_NAME : String = "categoryName" + BELONGS_CATEGORY_NEWS : String = "categoryNews" + BELONGS_CATEGORY_ERROR : String = "DOES-NOT-BELONG-CATEGORY" + IS_PUBLISHED_NEWS : String = "IS-PUBLISHED-NEWS" + IS_PUBLISHED_NEWS_NEWS : String = "news" + IS_PUBLISHED_NEWS_ERROR : String = "IS-NOT-PUBLISHED-NEWS" + NEWS_WITH_EXPIRE_DATE : String = "NEWS-WITH-EXPIRE-DATE" + NEWS_WITH_EXPIRE_DATE_NEWS : String = "news" + NEWS_WITH_EXPIRE_DATE_DATE : String = "dateExpire" + NEWS_WITH_EXPIRE_DATE_ERROR : String = "VALID-NEWS" + NEWS_ALREADY_PUBLISHED : String = "NEWS-ALREADY-PUBLISHED" + PUBLISH_NEWS_ERROR : String = "PUBLISH-NEWS-ERROR" + REMOVE_NEWS_ERROR : String = "REMOVE-NEWS-ERROR" + SENT_NEWS_ERROR : String = "SENT-NEWS-ERROR" + SENT_PHOTOS_ERROR : String = "SENT-PHOTOS-ERROR" + SENT_GUIDELINES_ERROR : String = "SENT-GUIDELINES-ERROR" - theInstance : Ontology = new NewsOntology() </pre>
<pre> + getInstance() : Ontology - NewsOntology() </pre>