UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
DEPARTAMENTO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GABRIEL CYPRIANO SACA

USANDO JADE PARA EVOLUIR A INFRA-ESTRUTURA DE APOIO À CONSTRUÇÃO DE AGENTES PARA ATUAREM EM ODE

VITÓRIA
2008

GABRIEL CYPRIANO SACA

USANDO JADE PARA EVOLUIR A INFRA-ESTRUTURA DE APOIO À CONSTRUÇÃO DE AGENTES PARA ATUAREM EM ODE

Monografia apresentada ao Curso de Ciência da Computação do Departamento de Informática do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação, na área de concentração de Sistemas de Informação.

Orientador: Prof. Dr. Ricardo de Almeida Falbo

VITÓRIA
2008

GABRIEL CYPRIANO SACA

USANDO JADE PARA EVOLUIR A INFRA-ESTRUTURA DE APOIO À CONSTRUÇÃO DE AGENTES PARA ATUAREM EM ODE

COMISSÃO EXAMINADORA

_____

Prof. Ricardo de Almeida Falbo, D. Sc.
Universidade Federal do Espírito Santo (UFES)
Orientador

_____

Prof. Giancarlo Guizzardi, PhD.
Universidade Federal do Espírito Santo (UFES)

_____

Renata Silva Souza Guizzardi, PhD.
Universidade Federal do Espírito Santo (UFES)

Vitória, 17 de Julho de 2008.

A minha família, por todo o apoio.

A Sara, pelo companheirismo e compreensão.

A Ricardo Falbo, por me guiar pelo caminho.

RESUMO

USANDO JADE PARA EVOLUIR A INFRA-ESTRUTURA DE APOIO À CONSTRUÇÃO DE AGENTES PARA ATUAREM EM ODE

GABRIEL CYPRIANO SACA

Julho de 2008

Orientador: Prof. Ricardo de Almeida Falbo

Agentes de software provêem soluções para problemas computacionais complexos e, portanto, Ambientes de Desenvolvimento de Software (ADSs) podem se beneficiar dessa tecnologia. Neste trabalho foi proposta a evolução de AgeODE, uma infra-estrutura para apoiar a construção de agentes para atuarem em ODE (*Ontology-based software Development Environment*), um ADS desenvolvido com base em ontologias.

Chegou-se à conclusão que o *framework* para a construção de agentes no qual estava fundamentada a primeira versão de AgeODE apresentava diversos problemas, sendo o mais grave o fato do projeto que o desenvolvia ter sido desativado. Na busca por soluções, chegou-se ao *framework* JADE, provavelmente o *framework* para construção de agentes dominante atualmente.

Esta monografia apresenta a nova versão de AgeODE, que teve origem com o levantamento de oportunidades de melhoria da versão anterior. A nova versão adaptou a abordagem para tratar os padrões de agentes da versão anterior, definiu uma nova arquitetura para os agentes e um padrão para se criar comportamentos reutilizáveis de agentes. Outras contribuições deste trabalho foram uma abordagem para tratar ontologias no contexto de JADE e a proposta para se apresentar as sugestões dos agentes na própria ferramenta de trabalho. Para avaliar a nova infra-estrutura, um sistema multiagente foi construído para apoiar a Gerência de Riscos.

Palavras-chave: Agentes, Sistemas Multiagente, Infra-estruturas de Agentes, Ambientes de Desenvolvimento de Software.

ABSTRACT

USING JADE TO EVOLVE THE INFRASTRUCTURE TO SUPPORT THE DEVELOPMENT OF AGENTS EMBEDDED IN ODE

Gabriel Cypriano Saca

July, 2008

Supervisor: Prof. Ricardo de Almeida Falbo

Since software agents provide solutions to complex computational problems, Software Engineering Environments (SEEs) can benefit themselves from this technology. This work presents an evolution of AgeODE, the infrastructure to support the development of agents embedded in ODE - Ontology-based software Development Environment.

The first version of AgeODE was built on top of the JATLite framework, which unfortunately had no longer received support. Therefore we proposed a shift in AgeODE's infrastructure, adopting JADE in place of JATLite as the base framework. JADE is probably the dominant agent development framework nowadays.

This work presents AgeODE's new version. It had its origins with the identification of problems in the first version. AgeODE's new version has adapted the way it approaches agent patterns, favoring composition of behaviours over inheritance. Moreover, AgeODE defines a new agent architecture, and a new design pattern to develop reusable agents' behaviours. Other major contributions were a preliminary approach for building JADE ontologies from domain ontologies, and a new approach for agent suggestions which aims at presenting suggestions embedded in the tool the user is currently working on. Finally, a multiagent system was built using AgeODE to support Risk Management in ODE.

Keywords: Agents, Multiagent Systems, Agent Infrastructures, Software Engineering Environments.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

Capítulo 1

Introdução

A utilização de agentes de software na solução de problemas computacionais complexos tem sido um assunto bastante explorado nas últimas décadas. Muito mais que uma nova tecnologia, a orientação a agentes tem se mostrado uma nova forma de pensar estratégias de resolução de problemas (SCHWAMBACH, 2004).

De forma sucinta, a orientação a agentes adota o conceito de agente para caracterizar uma unidade autônoma de resolução de problemas (WOOLDRIDGE et al., 1995), a partir da qual soluções são criadas por meio da organização de agentes trabalhando cooperativamente, cada um atuando como responsável pela resolução de parte do problema, caracterizando, assim, um sistema distribuído para resolução de problemas, denominado sistema multiagente (JENNINGS *et al.*, 1998).

A capacidade de agentes de software para, de forma autônoma, executar ou apoiar a realização de algumas tarefas do processo de software e, de forma pró-ativa, disseminar conhecimento tem estimulado bastante o uso dessa tecnologia em Ambientes de Desenvolvimento de Software (ADSs) (PEZZIN, 2004). ADSs são sistemas que buscam combinar técnicas, métodos e ferramentas para apoiar o Engenheiro de Software na construção de produtos de software, abrangendo todas as atividades do processo de software ou pelo menos porções significativas dele (FALBO, 1998) (HARRISON *et al.*, 2000).

Apesar de existirem vários *frameworks* destinados à construção de agentes (JEON *et al.*, 2000) (BELLIFEMINE *et al.*, 2007), de modo geral, eles não se destinam a construir agentes que atuarão no contexto de um ADS. Desse modo, surge a necessidade de se criar infra-estruturas que definam abordagens específicas para a criação de agentes que atuarão em ADSs (PEZZIN, 2004).

1.1. Contexto e Objetivo do Trabalho

Este trabalho está inserido no contexto do Projeto ODE (*Ontology-based software Development Environment*) (FALBO *et al.*, 2003), desenvolvido no Laboratório de Engenharia de Software da Universidade Federal do Espírito Santo (LabES) e que tem sido foco de pesquisas de diversos alunos de graduação e mestrado. O principal objetivo deste projeto é o desenvolvimento de um ambiente integrado, centrado em processos, para apoiar organizações de software em seus esforços de desenvolvimento e manutenção de software.

O ambiente ODE possui uma infra-estrutura para apoiar a construção de agentes autônomos para atuarem em ODE, desenvolvida originalmente em (PEZZIN, 2004), denominada AgeODE. Pezzin (2004) definiu: (i) alguns tipos específicos de agentes úteis para ADSs, (ii) como se deveria dar a comunicação entre agentes, (iii) como eles teriam acesso aos objetos do domínio do problema para compor sua base de conhecimento e (iv) como seria a arquitetura interna dos mesmos.

Entretanto, em 2006, o ambiente ODE passou por uma reestruturação em sua arquitetura para torná-lo multi-usuário, tendo sido a sua camada de persistência alterada. Essa reestruturação trouxe impactos em diversas aplicações do ambiente, dentre elas AgeODE. Devido a essa manutenção, aproveitou-se para avaliar a infra-estrutura e identificar oportunidades de melhoria. Nessa ocasião, detectou-se que JAT*Lite* (JEON *et al.*, 2000) (JATLITE, 2004), o *framework* para a construção de agentes no qual estava fundamentada a primeira versão de AgeODE, havia sido desativado e, por conseguinte, o mesmo estava defasado em relação a outros *frameworks*.

Desse modo, propôs-se a mudança do *framework* base de AgeODE para JADE (BELLIFEMINE *et al.*, 2007), provavelmente o *framework* para construção de agentes dominante atualmente. Essa mudança traz diversos benefícios imediatos à infra-estrutura, como, por exemplo, o aumento do desempenho, que antes estava abaixo do esperado, mas também trouxe novos desafios.

O objetivo principal deste trabalho é evoluir AgeODE, de forma a torná-la uma infra-estrutura mais flexível e poderosa, no sentido de aumentar a simplicidade para se construir sistemas multiagente e, ao mesmo tempo, proporcionar mais recursos aos construtores de agentes, de modo a viabilizar a criação de sistemas que apóiem as atividades de ODE de maneira efetiva.

1.2. Histórico do Trabalho

O começo deste trabalho, a contar pelos primeiros estudos sobre ADSs, o ambiente ODE e gerência de conhecimento, ocorreu em um projeto de iniciação científica realizado entre 2006 e 2007. Neste projeto, o foco principal da pesquisa foi voltado à reintegração de AgeODE ao ambiente. No entanto, no decorrer deste projeto chegou-se à conclusão de que era mais interessante reformular AgeODE antes da reintegração, visto que sua base no *framework* JAT*Lite* estava fadada a tornar-se defasada, já que o projeto JAT*Lite* havia sido descontinuado. Assim, os principais produtos desse projeto foram a identificação de oportunidades de melhoria detectadas na versão original de AgeODE, além de uma proposta para se trocar o *framework* base para JADE.

A pesquisa teve continuidade com mais um projeto de iniciação científica, que teve início em meados de 2007 e segue até o presente momento. Estudou-se mais a fundo o *framework* JADE e foram definidas diretrizes de estudo, detectando quais eram as peças-chave necessárias para se construir uma infra-estrutura baseada em JADE. Em paralelo à construção de um protótipo da nova versão de AgeODE, foi concebido um sistema multiagente para apoiar a Engenharia de Requisitos em ODE, projeto que utilizou diferentes metodologias de desenvolvimento orientado a agentes e que ainda está em curso.

Com o início deste projeto de graduação, procurou-se avançar no caminho evolutivo de AgeODE, criando-se soluções para problemas não abordados em seu protótipo inicial, com ênfase maior na utilização de ontologias e linguagens de conteúdo. Uma análise crítica das abordagens de JADE levou a novas propostas de melhoria, nem sempre acatando as sugestões do *framework*. Além disso, o contato contínuo com AgeODE e com JADE proporcionou que se pudesse abstrair os principais elementos envolvidos na construção de agentes, dando origem à definição de uma nova arquitetura para os agentes construídos no contexto de AgeODE.

1.3. Organização do Trabalho

Esta monografia encontra-se organizada na presente *Introdução* e em mais quatro outros capítulos.

O Capítulo 2 – *Ambientes de Desenvolvimento de Software e Sistemas Multiagente* – apresenta a tecnologia de agentes e como ela vem sendo abordada no contexto do ambiente

ODE. O texto aborda, ainda, o *framework* JADE e as especificações de FIPA (FIPA, 2008), em que ele se baseia.

No Capítulo 3 – *Evolução de AgeODE* – é apresentada a principal contribuição deste trabalho, a nova versão de AgeODE. Seguiu-se um caminho evolutivo, apresentando-se, em um primeiro momento, a versão original de AgeODE, depois as oportunidades de melhoria detectadas e, por fim, a nova versão da infra-estrutura.

O Capítulo 4 – *Estudo de Caso: Gerência de Riscos Apoiada por um Sistema Multiagente* – apresenta a aplicação da infra-estrutura AgeODE para a construção de um sistema multiagente que objetiva disseminar conhecimento de forma pró-ativa no contexto da ferramenta de Gerência de Riscos do ambiente ODE.

Finalmente, o Capítulo 5 – *Considerações Finais* – contém as conclusões sobre o trabalho desenvolvido, evidenciando suas contribuições e perspectivas de trabalhos futuros.

Capítulo 2

Ambientes de Desenvolvimento de Software e Sistemas Multiagente

O paradigma de agentes é um novo e importante paradigma de desenvolvimento de software, que está começando a migrar de universidades e laboratórios de pesquisa para aplicações comerciais e industriais. Adotar tal abordagem para a resolução de problemas tem sido um assunto muito explorado nas últimas duas décadas. Inúmeros trabalhos têm apresentado conceituações, formalizações, protocolos, técnicas e métodos para aplicação desse paradigma no desenvolvimento de software. Isso tem acontecido pelo fato da abordagem multiagente possuir algumas características que viabilizam a resolução de problemas de outra forma que não a tradicional, adequando-se a problemas complexos e de natureza descentralizada. Esse paradigma adota o conceito de agente para caracterizar uma unidade autônoma de resolução de problemas. A partir disso, uma solução é criada por meio do agrupamento de agentes trabalhando cooperativamente, cada um deles resolvendo parte do problema. A este agrupamento é dado o nome de sistema multiagente (PEZZIN, 2004).

A capacidade de agentes de software para, de forma autônoma, executar ou apoiar a realização de algumas tarefas do processo de software tem estimulado bastante o uso dessa tecnologia em Ambientes de Desenvolvimento de Software (ADSs) (PEZZIN, 2004).

ADSs são sistemas que objetivam fornecer apoio ao processo de software, provendo um conjunto de ferramentas e facilidades integradas que sejam capazes de apoiar cada uma das atividades desse complexo processo, ou pelo menos porções significativas dele (HARRISON *et al.*, 2000). Claramente, construir um ambiente que atenda a esse objetivo não é algo trivial. Há anos, pesquisadores, universidades e até mesmo a indústria de software vêm investindo neste intento, com algumas tentativas frustradas e outras que fizeram com que a pesquisa progredisse, incorporando novas características e evoluindo esses ambientes (RUY, 2006).

Este capítulo discute a tecnologia de agentes e como ela vem sendo abordada no contexto do ambiente ODE (*Ontology-based software Development Environment*) (FALBO *et al.*, 2003). Para isso, o capítulo está organizado da seguinte forma: a Seção 2.1 – Agentes e Sistemas Multiagente – apresenta os principais conceitos da tecnologia de agentes; a Seção 2.2 – Engenharia de Software Orientada a Agentes e a Metodologia OplA – mostra sucintamente que a tecnologia de agentes, como um novo paradigma de desenvolvimento, requer que seja abordada como uma disciplina de engenharia, além de apresentar a metodologia OplA, utilizada neste trabalho; a Seção 2.3 – JADE e FIPA – apresenta o *framework* JADE, utilizado neste trabalho, além da organização padronizadora FIPA, na qual JADE se baseia; a Seção 2.4 – Ambientes de Desenvolvimento de Software e o Ambiente ODE – discute o que são ADSs e apresenta sucintamente o ambiente ODE, ADS no contexto do qual este trabalho foi desenvolvido; a Seção 2.5 – AgeODE – apresenta AgeODE, a infra-estrutura de apoio à construção de agentes do ambiente ODE e que é o foco principal da pesquisa desenvolvida neste trabalho; por fim, a Seção 2.6 apresenta as considerações finais e conclusões do capítulo.

2.1. Agentes e Sistemas Multiagente

Agentes podem ser vistos como sistemas computacionais capazes de ações autônomas em algum ambiente, a fim de atingirem seus objetivos de projeto. Um agente tipicamente sente seu ambiente (por meio de sensores) e tem disponível um repertório de ações que podem ser executadas para modificar o ambiente, o qual pode responder não-deterministicamente à execução de suas ações (WOOLDRIDGE, 1999). Dessa forma, pode-se dizer que um agente é tudo aquilo que pode ser visto como percebendo um ambiente e agindo sobre ele em busca de um conjunto de objetivos, como esquematizado na Figura 2.1.

Figura 2.1 – Um Agente interagindo com seu Ambiente
Fonte: Wooldridge (1999)

Assim, há diferenças significativas entre agentes e objetos (WOOLDRIDGE, 1999). Uma delas está no grau no qual agentes e objetos são autônomos. Um objeto pode exibir autonomia sobre seu estado, ou seja, ele pode controlar o acesso a suas informações (atributos), mas um objeto não exibe controle sobre seu comportamento. Agentes, por outro lado, têm também controle sobre seu comportamento. É melhor pensar, então, em agentes não como invocadores de métodos, mas como requisitantes de ações a serem executadas. Um agente pode requisitar que outro execute uma ação, mas este outro pode ou não executar a ação. No caso da orientação a objetos, a decisão está no objeto que invoca o método. No caso da orientação a agentes, a decisão está com o agente que recebe a requisição.

Outra diferença a se destacar entre um sistema baseado em objetos e um sistema baseado em agentes é que o último é eminentemente *multi-threaded* (possui múltiplas linhas de controle), sendo que se supõe que cada agente tem sua própria linha de controle. No modelo de objetos padrão, há uma única linha de controle do sistema. Essa linha de controle vai passando de objeto para objeto (PEZZIN, 2004).

Todavia, a maioria dos sistemas baseados em agentes não é composta apenas de agentes. Os agentes não existem por si só, eles atuam em um ambiente onde cumprirão seus objetivos de projeto e esse ambiente é, normalmente, construído usando outras tecnologias, como a orientação objetos (SCHWAMBACH, 2004). Assim, um sistema multiagente compreende múltiplos agentes e objetos, que são abstrações distintas usadas para modelar diferentes entidades do domínio do problema (GARCIA *et al.*, 2002).

De fato, nosso mundo é composto de entidades ativas e de entidades passivas. Por exemplo, em uma organização, um empregado é uma entidade ativa, enquanto os recursos que ele usa para seu trabalho são entidades passivas. Assim, nós podemos conceber um sistema como composto de entidades ativas (agentes) que manipulam uma certa quantidade de entidades passivas (objetos) a fim de realizar suas tarefas (GUIZZARDI, 2006).

Dessa forma, é importante destacar as principais características básicas de agentes, que não são encontradas em objetos (WOOLDRIDGE, 1999):

- Autonomia – controle sobre suas ações e sobre seu estado interno, além da capacidade de agir sem intervenção direta de humanos ou outros agentes;
- Reatividade – habilidade de perceber e reagir a mudanças em seu ambiente;
- Proatividade – propriedade de agir guiado por objetivos, a partir de iniciativa própria;

- Sociabilidade – potencialidade de se comunicar com outros agentes para que possa atingir seus objetivos.

A questão da sociabilidade se refere ao fato de que, dado um problema, uma solução pode ser criada por meio do agrupamento de agentes trabalhando cooperativamente, cada um deles resolvendo parte do problema. A esse agrupamento é dado o nome de sistema multiagente. Cada agente desempenha um ou mais papéis específicos em um sistema multiagente. Define-se como papel aquilo que é esperado que o agente faça dentro da organização, ou seja, um conjunto de responsabilidades bem definido dentro do contexto global do sistema que o agente pode cumprir com um certo grau de autonomia (ZAMBONELLI *et al.*, 2000).

Pode-se dizer também que agentes trabalham juntos em um sistema multiagente para solucionar problemas que estão acima de suas capacidades e conhecimentos individuais (JENNINGS *et al.*, 1998). Visto que cada agente tem informação ou capacidade incompleta para solucionar o problema, ressaltam-se algumas características de sistemas multiagente (JENNINGS *et al.*, 1998):

- cada agente tem um ponto de vista limitado;
- não há controle global do sistema;
- os dados estão descentralizados;
- a computação é assíncrona.

A natureza complexa desses problemas abre espaço para que os agentes tenham objetivos em comum ou mesmo objetivos conflitantes. Dessa forma, agentes em um mesmo sistema podem se comunicar de forma direta, por meio de comunicação e negociação, ou mesmo de forma indireta, agindo sobre o ambiente em que eles se situam (BELLIFEMINE *et al.*, 2007).

Além disso, para viabilizar a comunicação direta entre agentes, cada agente tem uma identidade que o distingue de outros agentes no mesmo sistema e é expressa em termos de um nome, um endereço e uma descrição dos serviços que ele fornece. A identidade de um agente pode ser comunicada para outros agentes e agentes podem livremente aparecer e desaparecer no sistema ou mudar sua localização (BRUGALI *et al.*, 2000).

A definição de estratégias que conciliem os interesses individuais de cada agente para que as atividades relacionadas desenvolvam-se de modo coordenado é um dos aspectos fundamentais a serem considerados no projeto de sistemas multiagente (SEGHROUCHNI, 1996). Assim, uma importante faceta de um ambiente de comunicação é a coordenação. A

coordenação é o processo pelo qual um agente raciocina sobre suas ações locais e as ações (antecipadas) de outros agentes com o objetivo de garantir que a comunidade funcione de maneira coerente (JENNINGS, 1996).

Existem diversos motivos pelos quais agentes em um mesmo sistema devem estar sujeitos à coordenação, incluindo (BELLIFEMINE *et al.*, 2007):

- os objetivos dos agentes podem causar conflito entre as suas ações;
- os objetivos dos agentes podem ser interdependentes;
- agentes podem ter diferentes habilidades e conhecimento diferenciado; e
- os objetivos dos agentes podem ser mais facilmente atingidos se diversos agentes trabalharem em busca deles.

Porém, a coordenação não é a única forma de interação em um sistema multiagente. Podem-se listar três tipos de interação entre agentes (GUIZZARDI, 2007):

- Coordenação, na qual os agentes devem gerenciar as dependências entre diferentes atividades, de forma que as ações dos diferentes agentes sejam sincronizadas e o trabalho dobrado seja evitado;
- Cooperação, quando os agentes devem trabalhar juntos em busca de um objetivo comum;
- Negociação, quando os agentes devem chegar a um acordo que seja aceitável para todas as partes envolvidas.

Como apontado anteriormente, em um sistema multiagente, os agentes precisam se comunicar a fim de alcançar seus objetivos e/ou objetivos da sociedade na qual existem. A comunicação entre agentes é um problema quádruplo, envolvendo: (i) protocolo de interação, (ii) linguagem de comunicação, (iii) linguagem de conteúdo e (iv) ontologias.

Os protocolos de interação se referem à estratégia de alto nível perseguida por um agente de software e que governa sua interação com outros agentes. Geralmente, mensagens não são trocadas entre agentes de forma isolada, mas como parte de uma seqüência de interação. Assim, protocolos de interação descrevem um padrão de comunicação como uma seqüência permitida de atos comunicativos entre os agentes (PEZZIN, 2004). Por exemplo, seguindo as especificações de FIPA (FIPA, 2008), o protocolo de interação de requisição (*fipa-request*) descreve um agente fazendo uma requisição a outro, que, por sua vez, deve responder com um ato comunicativo de acordo (*agree*) ou recusa (*refuse*).

As linguagens de comunicação se baseiam na teoria de atos de fala (AUSTIN, 1955) (SEARLE, 1969), também chamados de performativas ou atos comunicativos no âmbito da

tecnologia de agentes. A teoria de atos de fala defende que os atos de fala implicam em ações e que, dessa forma, podem surtir efeitos no modelo mental e nas atitudes de uma comunidade (BELLIFEMINE *et al.*, 2007). Um exemplo simples poderia ser: "Eu aceito me casar com você", no qual o ato de fala implica a ação de um casamento matrimonial que pode causar efeitos diversos na sociedade em que os indivíduos estão situados. Alguns dos atos de comunicação mais comuns em sistemas multiagente são informar (*inform*), requisitar (*request*), concordar (*agree*), não entendido (*not understood*) e recusar (*refuse*). As linguagens de comunicação visam a classificar de forma explícita qual é a ação correspondente a uma mensagem. Por exemplo, utilizando-se a terminologia da linguagem de comunicação FIPA-ACL (FIPA, 2008), é possível que se determine explicitamente que o ato comunicativo de uma mensagem é requisitar (*request*) ou concordar (*agree*).

De forma mais genérica, as linguagens de comunicação visam a permitir a anotação do conteúdo da mensagem com diversas informações, tais como: qual é o ato comunicativo, qual é a ontologia sobre a qual o conteúdo da mensagem discursa, quem é o remetente e quais são os receptores da mensagem. Dada a sua importância, algumas iniciativas têm procurado estabelecer padrões para linguagens de comunicação entre agentes, com destaque para FIPA-ACL e KQML (FININ *et al.*, 1995).

Já linguagens de conteúdo são linguagens formais que indicam qual é a sintaxe segundo a qual o conteúdo das mensagens é expresso. Agentes tipicamente se comunicam utilizando expressões complexas e, por isso, é necessário que haja uma sintaxe bem definida para que os agentes receptores possam analisar o conteúdo da mensagem (BELLIFEMINE *et al.*, 2007).

Além disso, o vocabulário comum a ser utilizado nessas mensagens é definido por meio de ontologias. Assim, as ontologias associam significado ao conteúdo das mensagens. Segundo Huhns *et al.* (1997), uma ontologia fornece um mundo virtual compartilhado, no qual cada agente pode fundamentar suas crenças e ações, dando aos agentes um domínio de discurso mais rico e mais útil.

Dessa forma, agentes só podem se comunicar efetivamente se seguirem um protocolo de interação conhecido por ambos e se compartilharem uma linguagem de comunicação, uma linguagem de conteúdo e um vocabulário comum, dado por ontologias (BRUGALI *et al.*, 2000).

Há várias razões para interconectar agentes computacionais, tais como (HUHNS *et al.*, 1999): (i) possibilitar que eles cooperem para solucionar problemas, (ii) para compartilhar conhecimento, (iii) para trabalhar em paralelo para resolver problemas em comum, (iv) para

que sistemas possam ser desenvolvidos e implementados de forma modular, (v) para que sistemas sejam tolerantes a falhas por meio de redundância, (vi) para representar múltiplos pontos de vista e conhecimento de especialistas e (vii) para serem reutilizáveis.

Segundo Juchen (2002), a aplicação de agentes em sistemas de informação se justifica quando observadas as seguintes características:

- O domínio envolve uma inerente distribuição de dados, capacidades para resolução de problemas e responsabilidades;
- Há a necessidade de se manter a autonomia das sub-partes envolvidas, bem como a estrutura organizacional;
- As interações são razoavelmente sofisticadas, incluindo negociação, compartilhamento de informações e coordenação;
- A solução para o problema não pode ser inteiramente descrita do princípio ao fim, devido às diversas mudanças que podem ocorrer no ambiente do sistema, além da imprevisibilidade e dinamicidade dos processos de negócio e da necessidade de capacidade pró-ativa.

2.2. Engenharia de Software Orientada a Agentes e a Metodologia OpIA

A maioria das metodologias de engenharia de software recentes é definida para uma abordagem orientada a objetos (OO) e a maioria dos sistemas multiagente, até um futuro próximo, será implementada com as tecnologias baseadas em objetos e componentes, a menos que surja uma linguagem de programação de agentes extremamente aceita. Assim, visto no nível de implementação, um agente é um objeto ou componente relativamente complexo. Entretanto, isto é como considerar que uma casa é uma pilha de tijolos, ao passo que é mais conveniente vê-la em termos de conceitos de mais alto nível, tais como sala de estar, cozinha e banheiro. Mesmo que o nível de agente seja o nível natural de abstração para descrever um sistema multiagente, a falta de notações diagramáticas aceitas e ferramentas de projeto e implementação pode impedir a arquitetura multiagente de explorar seus benefícios (SCHWAMBACH, 2004).

Quando um agente é visto de um nível mais alto de abstração, estruturas possuem focos que não são encontrados em objetos ou componentes convencionais. A orientação a agentes (OA) é, então, um paradigma para análise, projeto e organização do sistema e,

portanto, a engenharia de sistemas multiagente requer metodologias de engenharia de software orientada a agentes. Tais metodologias devem adotar uma linguagem de modelagem orientada a agentes capaz de prover primitivas para descrever essas estruturas de mais alto nível (CAIRE *et al.*, 2001).

Todavia, inicialmente, métodos e técnicas adequados a outros paradigmas, sobretudo orientados a objetos, foram empregados no desenvolvimento de sistemas baseados em agentes. Tratar o paradigma OA como extensão do paradigma OO pode ser vantajoso, porque padrões e técnicas de modelagem já bastante conhecidos podem ser aplicados, como é o caso da UML. Entretanto, deve-se realçar que agentes possuem características, como autonomia, não pertinentes ao paradigma de objetos e, por isso, existe a necessidade de definições próprias para eles. Além disso, agentes não existem sozinhos em um sistema. Eles precisam se integrar ao ambiente em que atuam e, normalmente, em sistemas de software, esse ambiente é modelado e implementado segundo outro paradigma, tal como o orientado a objetos. Deste modo, a integração do paradigma de agentes com outros paradigmas é fator determinante para o sucesso de sua utilização no desenvolvimento de sistemas de software (SCHWAMBACH, 2004).

Com o objetivo de avançar na Engenharia de Software para sistemas de agentes, diversas metodologias e linguagens de modelagem têm sido propostas. Algumas têm influência da orientação a objetos e métodos formais, como Gaia (WOOLDRIDGE *et al.*, 2000), ROADMAP (JUAN *et al.*, 2002), OperA (DIGNUM, 2004) e Prometheus (PADGHAM *et al.*, 2002); outras têm influência de análise organizacional, como Tropos (BRESCIANI *et al.*, 2004); existem propostas baseadas em UML, como AORML (WAGNER, 2003), MessageUML (CAIRE *et al.*, 2001) e AUML (ODELL *et al.*, 2000); e, por fim, existem metodologias com influência de Inteligência Artificial e sistemas baseados em conhecimento, como Mas-CommonKADS (IGLESIAS *et al.*, 1998).

Além disso, pode-se notar que algumas delas são inovadoras, no sentido de sugerir seus próprios modelos e linguagem de modelagem, tal como Tropos; outras são extensões de metodologias já conhecidas, tal como MAS-CommonKADS; além de existirem aquelas que combinam abordagens previamente existentes, como ARKnowD (GUIZZARDI, 2006), que combina Tropos e AORML.

2.2.1 – OplA

Este trabalho utiliza a metodologia OplA (*Objects plus Agents oriented methodology*), proposta em (SCHWAMBACH, 2004). OplA visa a apoiar o desenvolvimento de sistemas baseados em agentes e objetos, guiando o engenheiro de software nas principais atividades do processo de desenvolvimento de software.

Para cada uma das fases propostas, OplA define sub-atividades e os artefatos que devem ser gerados por elas, muitas vezes sugerindo modelos de documentos. A integração dos dois paradigmas, o orientado a objetos e o orientado a agentes, é feita de forma explícita, principalmente nas fases de análise e projeto, possibilitando uma implementação menos traumática. Além disso, o uso da UML como linguagem de modelagem, com extensões definidas usando apenas os mecanismos de extensão da própria UML, torna OplA uma metodologia fácil de ser assimilada e trabalhada usando as ferramentas CASE existentes (SCHWAMBACH, 2004).

A Tabela 2.1 apresenta as principais atividades propostas por OplA e os artefatos a serem produzidos nas fases de Especificação de Requisitos, Análise e Projeto. A descrição completa de cada uma das atividades, inclusive com diretivas para as fases de Implementação e Testes, pode ser encontrada em (SCHWAMBACH, 2004).

Tabela 2.1 – Resumo das Fases de OplA.

Fase	**Atividade**	**Sub-Atividade**	**Artefatos a serem produzidos**
Especificação de Requisitos			Documento de Especificação de Requisitos contendo um Modelo de Casos de Uso
Análise	Refinamento de Casos de Uso		Modelo de Casos de Uso identificando as formas de atuação dos agentes nos casos de uso
	Análise Estrutural	Identificação de Subsistemas	Diagramas de Pacotes
		Modelagem do Ambiente	Diagramas de Classes de Objetos
		Modelagem Estrutural dos Agentes	Diagramas de Classes de Agentes

Tabela 2.1 – Resumo das Fases de OplA (Continuação).

Fase	Atividade	Sub-Atividade	Artefatos a serem produzidos
Análise	Análise Estrutural	Modelagem do Conhecimento dos Agentes	Diagramas de Classes de Agentes e Objetos
		Modelagem Estática das Interações entre Agentes	Diagramas de Instâncias de Agentes (Diagramas de Objetos da UML)
Análise	Análise Comportamental	Modelagem de Estados de Agentes e Objetos	Diagramas de Estado
		Modelagem Dinâmica da Interação entre Agentes	Diagramas de Colaboração mostrando a interação entre Agentes
		Modelagem Dinâmica das Trocas de Mensagens	Diagramas de Seqüência
Projeto	Projeto Arquitetural	Definição da Arquitetura do Sistema	Diagrama de Pacotes
	Projeto Estrutural	idem Análise Estrutural, mas agora considerando a plataforma de implementação	idem Análise Estrutural
	Projeto Comportamental	idem Análise Comportamental, considerando a plataforma de implementação	idem Análise Comportamental

2.3. JADE e FIPA

Vê-se que a maioria das metodologias foca suas definições apenas nas fases de análise e projeto, o que provoca uma indesejável distância conceitual (*gap*) com as demais fases técnicas do processo de desenvolvimento. Esse *gap* só é percebido pelos desenvolvedores

durante a realização das atividades de implementação e testes de um sistema (SCHWAMBACH, 2004).

Todavia, como apontado anteriormente, a maioria dos sistemas multiagente tem sido implementada com as tecnologias baseadas em objetos e componentes e para apoiar essa implementação, abstrações de objetos tradicionais têm sido enriquecidas com a incorporação de novas características, tais como linhas de controle internas, tratamento de eventos e dependência de contexto (PEZZIN, 2004).

Entretanto, isso não quer dizer que agentes não adicionam novas facilidades para ajudar a resolver problemas; muito pelo contrário. O fato de objetos e agentes estarem de alguma forma convergindo, sobretudo em uma perspectiva de implementação, pode ser tomado como uma evidência de que as abstrações introduzidas pela tecnologia de agentes são adequadas para o desenvolvimento de sistemas de software complexos (PEZZIN, 2004).

Porém, conforme apontado por Kendall *et al.* (2000), a maioria dos projetos envolvendo agentes tem sido conduzida com muita dificuldade e de forma independente por cada grupo de desenvolvimento, o que leva a problemas, tais como: falta de uma definição padrão, isto é, agentes construídos por diferentes grupos têm diferentes capacidades; duplicação de esforços, uma vez que tem havido pouco reúso de arquiteturas, projetos ou componentes de agentes; e incapacidade para satisfazer potenciais requisitos industriais, dadas as dificuldades de integrar agentes a infra-estruturas computacionais e de software já existentes.

Frameworks para construção de agentes têm sido propostos com o objetivo de tratar alguns dos problemas relativos ao desenvolvimento de agentes, tais como a falta de padronização e uniformidade na construção dos agentes e a falta de uma abordagem de reúso sistemática. Um *framework* desse tipo permite que agentes sejam desenvolvidos de uma forma mais fácil e padronizada, fazendo com que eles compartilhem diversas funcionalidades, tal como a forma de comunicação, evitando, assim, a duplicação de esforços e abrindo espaço para um reúso mais sistemático (PEZZIN, 2004). Um *framework*, neste contexto, é um projeto reutilizável de parte de um sistema, que é representado por um conjunto de classes abstratas e o modo como suas instâncias interagem (JOHNSON, 1997).

Segundo Sycara *et al.* (2003), tem havido considerável pesquisa na formulação de teorias, princípios e diretivas para a construção de sistemas multiagente, mas há relativamente pouca experiência em se construir agentes e colocá-los em uso. Portanto, o desenvolvimento de um sistema multiagente é, ainda, extremamente desafiante. Para esses autores, a pesquisa em sistemas multiagente não atingirá seu verdadeiro potencial até que se tenha uma grande

quantidade de sistemas, componentes e serviços em funcionamento. Para alcançar este objetivo, é crucial se ter um *framework* para apoiar a construção de sistemas multiagente que seja estável, amplamente usado, amplamente acessível e extensível.

Várias corporações de padronização, tal como a FIPA (FIPA, 2008), estão tentando definir padrões para vários aspectos de *frameworks* para sistemas multiagente, tais como padrões referentes à comunicação entre agentes, conforme discutido anteriormente.

Este trabalho utiliza o *framework* JADE (BELLIFEMINE *et al.*, 2007). Visto que JADE é, de certa forma, uma implementação das especificações de FIPA, o *framework* é altamente dependente das idéias expressas nas especificações de FIPA. Assim, a seção 2.3.1 apresenta FIPA e, logo após, a seção 2.3.2 apresenta JADE.

2.3.1 – FIPA

Nesta seção, apresenta-se, de forma sucinta, um subconjunto das especificações de FIPA que são relevantes a JADE e a este trabalho, além de linhas gerais a respeito da organização. As especificações de FIPA são classificadas com relação ao seu status, que pode ser: preliminar, experimental, padrão ou obsoleta. Além disso, todas as especificações estão publicamente disponíveis em seu *website* (FIPA, 2008).

FIPA (*The Foundation for Intelligent Physical Agents*) foi criada em 1996 como uma associação internacional sem fins lucrativos para desenvolver um conjunto de padrões para a tecnologia de agentes. Além disso, em meados de 2005, FIPA foi incorporada à IEEE (BELLIFEMINE *et al.*, 2007).

FIPA baseia-se no princípio de que apenas o comportamento externo dos componentes de sistemas de agentes devem ser especificados, deixando a arquitetura interna e detalhes de implementação para os desenvolvedores das diferentes plataformas. Isso garante interoperabilidade entre as plataformas condizentes com as especificações. Essa interoperabilidade entre plataformas foi testada nos testes de interoperabilidade organizados por FIPA em 1999 e 2001, além do projeto de larga escala Agentcities (WILLMOTT *et al.*, 2001), que integrou várias plataformas distribuídas pelo globo (BELLIFEMINE *et al.*, 2007).

Para atingir interoperabilidade entre as diversas plataformas, FIPA especifica abstrações bem definidas independentes de tecnologia, além de definir mapeamentos para tecnologias comumente utilizadas que visam à interoperabilidade, como CORBA e JINI (BELLIFEMINE *et al.*, 2007).

FIPA define especificações nos âmbitos de comunicação entre agentes, gerência de agentes e arquitetura de plataformas, que funcionam como normas para que os agentes possam existir, operar e se comunicar (BELLIFEMINE *et al.*, 2007). Visto que as especificações relacionadas à gerência de agentes e arquitetura de plataformas são mais direcionadas para desenvolvedores de plataformas que desejam ser condizentes com FIPA e que neste trabalho utilizamos a plataforma JADE, daremos maior ênfase às especificações relacionadas à comunicação.

Segundo Bellifemine *et al.* (2007), em 1997 surgiu o primeiro conjunto de especificações de FIPA. Sob uma perspectiva de comunicação, FIPA decidiu adotar ARCOL (SADEK, 1991) da France Télécom como a base para uma linguagem de comunicação, que mais tarde seria conhecida com FIPA-ACL, ou apenas ACL. A decisão de adotar ARCOL surgiu de um debate intenso e controverso sobre os méritos de ARCOL sobre KQML. Ao final do debate, ARCOL foi a escolhida por ser fundamentada por uma semântica formal. A seguir, FIPA decidiu adotar a linguagem SL, também conhecida como FIPA-SL, como um padrão instrutivo para a expressão do conteúdo das mensagens trocadas entre agentes. Além disso, também foram adotados vários protocolos de cooperação, conhecidos em FIPA como protocolos de interação, oriundos também da France Télécom.

Uma mensagem FIPA-ACL contém um conjunto de um ou mais parâmetros. Quais parâmetros são necessários para uma comunicação efetiva entre agentes varia de acordo com a situação. O único parâmetro obrigatório em todas as mensagens ACL é o *performative* (ato de fala), porém espera-se que a maioria das mensagens ACL também contenha os parâmetros *sender* (remetente), *receiver* (receptor) e *content* (conteúdo). Alguns dos parâmetros definidos por FIPA relevantes para este trabalho são:

- *performative* – tipo do ato comunicativo da mensagem
- *sender* – identidade do remetente da mensagem
- *receiver* – identidade dos receptores da mensagem
- *content* – conteúdo da mensagem
- *language* – linguagem na qual o conteúdo está expresso
- *ontology* – referência a uma ontologia que dá sentido aos símbolos do conteúdo da mensagem
- *protocol* – protocolo de interação utilizado para estruturar a interação
- *conversation-id* – identificador único para todo o conjunto de mensagens de uma interação

- *reply-by* – Uma data/hora indicando até quando uma resposta poderá ser recebida

Além desses parâmetros, é possível que o usuário defina um parâmetro e o inclua na mensagem ACL precedido de "X-". Segue um exemplo simples de uma mensagem FIPA-ACL com a performativa *request* (requisição).

```
(request
     :sender (agent-identifier :name alice@dominio.com)
     :receiver (agent-identifier :name bob@dominio.com)
     :ontology assistente-viagens
     :language FIPA-SL
     :protocol fipa-request
     :content
       ""((action
            (agent-identifier :nome bob@dominio.com)
             (reservar-hotel :chegada 01/08/2008
                             :saída 06/08/2008 ... )
       ))""
)
```

Como apontado anteriormente, FIPA-ACL define a comunicação em termos de um ato comunicativo, ou performativa. Assim, FIPA padroniza um conjunto de atos comunicativos (FIPA, 2008), fornecendo para cada um, uma descrição textual, um modelo formal expresso em lógica modal e um exemplo. Segue uma descrição sucinta de alguns atos comunicativos relevantes para este trabalho:

- *Agree* – A ação de concordar em executar alguma ação, possivelmente no futuro;
- *Failure* – A ação de contar a um outro agente que se tentou realizar uma ação, mas ocorreu alguma falha;
- *Inform* – O remetente informa ao receptor que uma dada proposição é verdadeira;
- *Not Understood* – O remetente do ato (por exemplo, *i*) informa ao receptor (por exemplo, *j*) que ele percebeu que *j* realizou alguma ação, mas que *i* não entendeu o que *j* fez. Um caso comum é quando *i* conta a *j* que *i* não entendeu a mensagem que *j* acabou de lhe enviar;
- *Refuse* – A ação de se recusar a executar uma dada ação e explicar a razão da recusa;

- *Request* – O remetente requisita que o receptor execute alguma ação. Uma importante classe de usos do ato comunicativo *request* é a requisição para que o receptor execute um outro ato comunicativo.

Uma das vantagens em se explicitar qual é a performativa de uma mensagem é a possibilidade de se especificar seqüências pré-definidas de mensagens que podem ser aplicadas em diversas situações que compartilham um mesmo padrão de comunicação independente de domínio (BELLIFEMINE *et al.*, 2007). Essas seqüências pré-definidas de mensagens são os protocolos de interação. FIPA especifica vários protocolos de interação e opta por descrevê-los utilizando AUML, como indicado por Odell (2001). A Figura 2.2 apresenta o Protocolo de Interação FIPA *Request* (FIPA, 2008), identificado neste trabalho simplesmente por *fipa-request*, assim como deve ser descrito no parâmetro *protocol* de uma mensagem ACL.

Como mostra a figura, o protocolo *fipa-request* permite que um agente, o *Initiator* (Iniciador), requisite um outro agente, o *Participant* (Participante), a realizar uma ação. O agente Participante processa a requisição e decide se deve aceitar ou recusar a requisição. Se o Participante decide recusar a requisição, então *refused* é verdadeiro e o agente deve realizar um ato comunicativo *refuse*; caso contrário, *agreed* é verdadeiro. Se a condição indicar que um acordo explícito é requerido (ou seja, "*notification necessary*" é verdade), então o agente Participante realiza um ato comunicativo *agree*. Esse acordo explícito é opcional e depende de circunstâncias, como, por exemplo, se a ação requisitada pode ser executada com certa rapidez e ser finalizada antes de um horário especificado no campo *reply-by* da mensagem ACL. Desde que o agente Participante concorde em executar a requisição, ele deve responder com uma das três seguintes opções: (i) um *failure*, se ele falha em sua tentativa de executar a requisição; (ii) um *inform-done*, se ele executa com sucesso a requisição e apenas deseja indicar que a finalizou; ou (iii) um *inform-result*, se o agente deseja indicar tanto que finalizou a requisição quanto informar ao agente Iniciador os resultados da requisição.

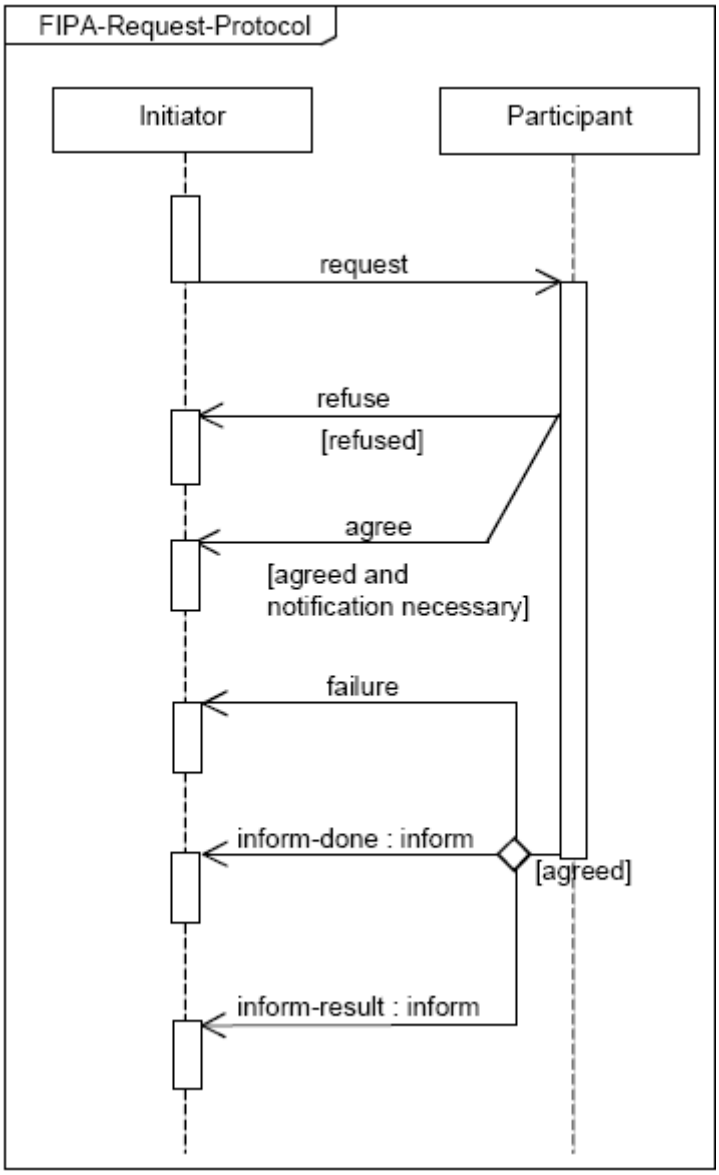

Figura 2.2 – Protocolo de Interação FIPA *Request*
Fonte: FIPA (2008)

Qualquer interação entre agentes, além de utilizar um protocolo de interação de FIPA, deve ser identificada, utilizando-se o campo *conversation-id* de uma mensagem ACL, com um identificador único e não nulo. Dessa forma, os agentes envolvidos devem identificar todas as mensagens da interação pelo mesmo identificador de interação. Isso facilita que cada agente possa gerenciar suas estratégias e suas ações. Por exemplo, isso permite que um agente identifique cada interação individualmente e, assim, permite que ele possa raciocinar utilizando todo o histórico de mensagens trocadas em cada interação.

Além disso, a qualquer momento em um protocolo de interação, o receptor de um ato comunicativo pode informar ao remetente que ele não entendeu a mensagem recebida. Isso é

feito realizando-se um ato comunicativo *not-understood*. O envio de um *not-understood* no contexto de uma interação pode acarretar o término do protocolo de interação e, dessa forma, o término da interação implica que qualquer compromisso feito durante a interação é nulo.

No que toca linguagens de conteúdo, FIPA não impõe que uma linguagem específica deva ser utilizada, porém define e recomenda a linguagem SL. FIPA define três subconjuntos da linguagem SL (SL0, SL1 e SL2), que divergem em termos de quais operadores são suportados. SL1 estende a forma representacional mínima de SL0, adicionando conectivos boleanos para representar expressões proposicionais, assim como *not*, *and* e *or*. SL2 estende SL1 permitindo lógica de predicados de primeira ordem e lógica modal, mas é restrita para garantir decidibilidade. Deve ser notado que cada ato comunicativo requer o uso de diferentes subconjuntos de FIPA-SL. Por exemplo, *request* requer apenas o uso do operador *done*, que é definido em SL0. Mais detalhes sobre FIPA-SL podem ser encontrados nas especificações de FIPA (FIPA, 2008).

Segundo Bellifemine *et al.* (2007), além de SL ser a linguagem de conteúdo recomendada por FIPA, é provavelmente a linguagem de conteúdo mais difundida na comunidade envolvida com agentes de software. Dessa forma, na medida em que houver a necessidade de que agentes desenvolvidos no contexto de um sistema possam se comunicar com outros agentes desenvolvidos por equipes diferentes e utilizando *frameworks* potencialmente diferentes, SL deve ser a principal candidata à linguagem de conteúdo.

Já sob uma perspectiva de gerência de agentes, é importante destacar que FIPA define um agente AMS (*Agent Management System*), um agente gerenciador, que, entre outras coisas, é o responsável por prover o serviço de páginas brancas, onde um agente pode obter o endereço de outro agente por meio de seu nome. Além disso, FIPA define um agente DF (*Directory Facilitator*), que tem como objetivo prover o serviço de páginas amarelas, com o qual agentes podem tornar públicas para outros agentes as linguagens de conteúdo e ontologias com as quais são capazes de se comunicar, além da descrição dos serviços que é capaz de realizar. Dessa forma, por meio de um agente DF, agentes podem descobrir em tempo de execução quais outros agentes podem lhe auxiliar a executar uma determinada tarefa, pedindo ao agente DF os agentes que são capazes de realizar um determinado serviço e com os quais são capazes de se comunicar.

2.3.2 – JADE

Dentre as plataformas em conformidade com as especificações de FIPA, encontra-se JADE (*Java Agent DEvelopment framework*) (BELLIFEMINE *et al.*, 2007), provavelmente o *framework* para construção de agentes dominante atualmente. O *framework* facilita a criação de sistemas multiagente, dispondo de um ambiente de execução que implementa a gerência de ciclo de vida dos agentes, oferecendo algum apoio para a implementação da lógica interna dos agentes, além de dispor de um conjunto de ferramentas gráficas para facilitar a depuração e monitoramento dos agentes. JADE foi inicialmente desenvolvido pela Telecom Italia e mais tarde, em 2000, foi lançado como software livre de código aberto.

No que diz respeito às especificações de FIPA, JADE oferece apoio a todo o conjunto de especificações que tenham atingido o status de padrão. Entre elas, destaca-se no contexto deste trabalho o apoio que JADE oferece aos serviços de páginas brancas e de páginas amarelas, além do apoio à linguagem de comunicação FIPA-ACL, à linguagem de conteúdo FIPA-SL e à maioria dos protocolos de interação de FIPA. Além disso, existem ainda vários componentes do *framework* que vão além das especificações de FIPA, entre eles: entrega persistente de mensagens, ferramentas gráficas de depuração e monitoramento e apoio à definição de ontologias.

Visto que JADE é um *framework* construído utilizando-se a linguagem de programação Java, uma classe de agente é representada por uma classe Java, que deve herdar da classe `Agent`, disponível no *framework*. Porém, ao contrário de objetos comuns, que são manipulados utilizando-se referências, um agente é sempre instanciado pelo ambiente de execução de JADE e sua referência não é conhecida por nenhum outro objeto Java (a menos que o agente explicitamente passe sua referência como parâmetro). Dessa forma, agentes nunca interagem com outros agentes via chamada de métodos, mas apenas por envio de mensagens assíncronas, o que está de acordo com o conceito de que agentes devem ter autonomia. Para que um agente possa referenciar um outro agente em uma mensagem ACL, deve ser utilizado o identificador do agente, uma instância da classe `AID`, também fornecida pelo *framework*.

Um outro ponto que reforça a autonomia de agentes em JADE é que cada agente em execução no ambiente tem a sua própria linha de execução, que é implementada utilizando-se *threads* de Java. Não obstante, JADE abstrai para o desenvolvedor todos os detalhes de programação concorrente, decorrentes do uso de *threads*.

Em JADE há uma separação entre a entidade agente e as entidades que representam as tarefas a serem realizadas pelos agentes, ou seja, seus planos, chamados em JADE de

comportamentos. Os comportamentos dos agentes são implementados como subclasses da classe `Behaviour`. JADE oferece, assim, um mecanismo semelhante à composição, por meio do qual comportamentos podem ser adicionados a agentes a qualquer momento via o método `addBehaviour()` da classe `Agent`. O escalonamento e a execução dos comportamentos são completamente gerenciados por JADE, que adota um esquema de escalonamento cooperativo, portanto, não-preemptivo. Dessa forma, comportamentos devem ser modulares, no sentido de que deve-se executar parte de um comportamento e dar vez para que parte de outro comportamento execute, caso contrário, um comportamento que tenha uma execução demorada deixaria os outros comportamentos do mesmo agente em longo período de espera. Um ponto forte de JADE é que, quando não há nenhum comportamento a ser executado, a *thread* do agente fica em modo *sleep* para que não consuma processador do sistema de computação desnecessariamente.

JADE fornece várias classes reutilizáveis de comportamento para que o usuário possa desenvolver os planos dos agentes utilizando tanto abstrações simples, como comportamentos cíclicos e temporais, além de abstrações mais complexas, como composição de comportamentos, comportamentos seqüenciais, paralelos e baseados em máquinas de estado finito. Para utilizá-las, basta que o desenvolvedor de agentes crie classes de comportamento que herdam dessas classes reutilizáveis, adicionando a lógica dependente de domínio.

Porém, o ponto forte de JADE no que diz respeito ao reúso de comportamentos de agentes é que o *framework* também fornece comportamentos reutilizáveis que implementam os protocolos de interação de FIPA. Dessa forma, os protocolos de interação de FIPA não são apenas uma sugestão de como trocar mensagens que representam atos comunicativos em uma interação, eles são realmente executados no contexto de interações entre agentes. Além disso, o programador não precisa implementar todos os detalhes relacionados ao fluxo de mensagens quando dois ou mais agentes interagem seguindo um protocolo de interação padrão. Isso é feito nos comportamentos reutilizáveis fornecidos por JADE. Tudo o que o programador tem que fazer é redefinir alguns métodos desses comportamentos, inserindo a lógica específica de domínio, que será o plano do agente no contexto da interação.

Por exemplo, no caso do protocolo de interação *fipa-request*, no caso de o agente Participante responder ao agente Iniciador com um *agree*, então será o próprio *framework* JADE que irá verificar se o ato comunicativo posteriormente realizado pelo Participante no contexto dessa mesma interação será um *failure* ou um *inform*.

As classes de comportamento relativas aos protocolos de interação de FIPA são classificadas de acordo com o papel que o agente realizará na interação, ou seja, se o agente

iniciar a interação, o nome da classe de comportamento será finalizada por *Initiator*, caso contrário, será finalizada por *Responder*.

Além disso, classes de comportamento que iniciam uma interação suportam tanto interações um-para-um quanto interações um-para-muitos. Já classes de comportamento que respondem a um estímulo de interação são disponibilizadas em duas versões, uma versão que executa o comportamento de forma cíclica (ou seja, pode responder a vários estímulos de interação, um de cada vez) e outra versão que executa o comportamento uma só vez.

Outro ponto a ser discutido é que JADE fornece apenas uma classe para protocolos que seguem o mesmo padrão de interação. Por exemplo, os protocolos de interação *fipa-request* e *fipa-query* têm a mesma estrutura de interação e as classes de comportamento que JADE fornece para se executar ambos os protocolos de interação são `AchieveREInitiator`, para o iniciador da interação, e `AchieveREResponder`, para o agente que irá responder ao estímulo de interação, onde RE é uma sigla para *rational effect*.

No entanto, existem casos em que, para que um agente formule uma resposta a um estímulo, ele deve iniciar um outro protocolo de interação com outros agentes. Dessa forma, têm-se dois (ou mais) protocolos aninhados. No entanto, não há como executar um comportamento dentro de um método (neste caso, dentro do método do comportamento mais abrangente que será chamado por JADE). Assim, JADE oferece apoio a esse tipo de interação, permitindo que se registre um comportamento aninhado a ser executado no lugar do método correspondente do comportamento mais abrangente.

No contexto de um protocolo de interação, várias mensagens são trocadas entre agentes (classe `ACLMessage`). Assim, deve ser notado que o conteúdo de uma mensagem ACL contém símbolos independentes de domínio (as construções da linguagem de conteúdo) e símbolos dependentes de domínio (que são definidos em uma ontologia compartilhada pelos agentes). Dessa forma, o apoio a linguagens de conteúdo em JADE e o apoio a ontologias estão intimamente relacionados.

Em geral, mensagens ACL trocadas entre agentes são dispostas em uma forma textual, assim como apresentado na seção anterior. Adicionando isso ao fato de que os agentes de JADE são baseados em Java, há de se convir que, para que dois agentes troquem uma mensagem, é necessário que: (i) o agente remetente converta a sua representação interna, potencialmente sob a forma de objetos Java, em uma representação textual na sintaxe de uma determinada linguagem de conteúdo; (ii) o agente receptor faça a conversão inversa; e (iii) o agente receptor precise ter uma forma de associar os termos da mensagem com os conceitos

de uma ontologia compartilhada entre os agentes, o que Bellifemine *et al.* (2007) chamam de verificação semântica.

O apoio a ontologias e linguagens de conteúdo de JADE busca automatizar todas essas conversões e verificações semânticas, como mostra a Figura 2.3. Isso permite aos desenvolvedores de agentes manipular toda a informação no contexto de um agente utilizando objetos Java, sem que seja necessário realizar manipulação de *strings* para se fazer conversões ou mesmo verificações semânticas.

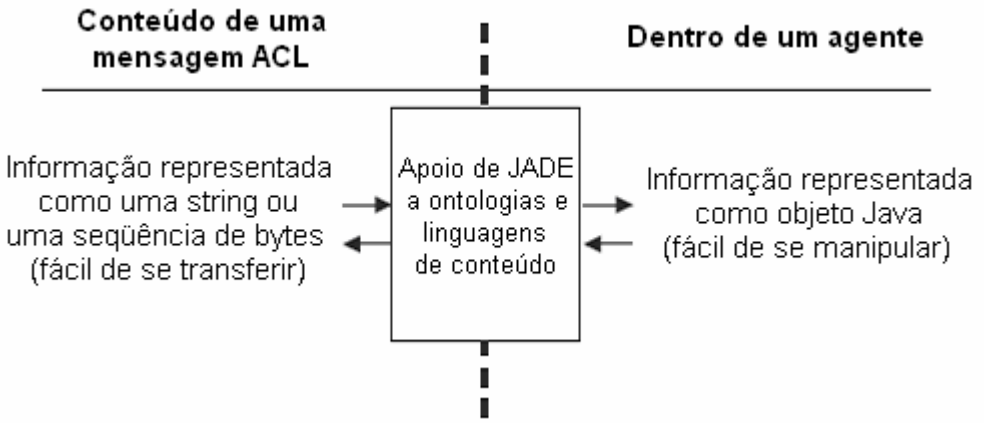

Figura 2.3 – Apoio de JADE a ontologias e linguagens de conteúdo
Fonte: Bellifemine *et al.* (2007)

O componente responsável por fazer essas conversões e verificações semânticas em JADE é o Gerenciador de Conteúdo. Cada agente em JADE tem uma referência a um Gerenciador de Conteúdo, instância da classe `ContentManager`. Apesar de o Gerenciador de Conteúdo prover uma interface de acesso a essas funcionalidades de conversão, ele, na verdade, delega essas operações para uma ontologia (instância da classe `Ontology`) e para um *codec* de linguagem de conteúdo (instância da interface `Codec`). Mais especificamente, a ontologia é a responsável por fazer a verificação semântica, enquanto o *codec* é o responsável por fazer a conversão entre objetos Java e sua representação textual, segundo a sintaxe de uma linguagem de conteúdo. Vale lembrar que JADE oferece apoio à linguagem de conteúdo FIPA-SL e, portanto, provê a implementação de um *codec* para essa linguagem, por meio da classe `SLCodec`.

Dessa forma, para que se utilize efetivamente o apoio a ontologias e linguagens de conteúdo oferecido por JADE, deve-se: (i) definir uma ontologia; (ii) desenvolver classes Java referentes aos elementos da ontologia, se essas já não existirem; e (iii) registrar a ontologia

definida e o *codec* da linguagem de conteúdo utilizada juntamente ao Gerenciador de Conteúdo que será utilizado para mediar a comunicação.

A definição de uma ontologia em JADE se dá através da criação de uma subclasse da classe `Ontology`. A partir daí, deve-se declarar quais são os conceitos, predicados e ações de agentes presentes na ontologia, ou seja, todo o universo de símbolos específicos de um domínio que podem ser encontrados no conteúdo de mensagens. Conceitos são, como usual, os conceitos do domínio que a ontologia se propõe a definir. Predicados são expressões que, quando utilizadas junto a conceitos, podem ser verdadeiras ou falsas. Ações de agentes são conceitos especiais que indicam ações que podem ser realizadas por agentes.

Em um domínio imaginário de agentes assistentes de viagens, um exemplo de conceito é *hotel*, que pode ter o atributo *nome*, do tipo string. Já *funciona-no-inverno* pode ser um predicado que tem valor verdadeiro quando aplicado a um hotel que funciona no inverno e valor falso quando aplicado a um hotel que não funciona no inverno. Além disso, *reservar-hotel* pode ser uma ação de agente, visto que um agente pode requisitar que outro agente reserve um hotel. Atributos de *reservar-hotel* podem ser, por exemplo, *hotel*, *chegada* e *saída*, os dois últimos do tipo data.

As classes referentes aos conceitos, predicados e ações de agentes devem seguir algumas regras, a saber: (i) devem implementar uma interface (sem métodos) correspondente, por exemplo, conceitos devem implementar a interface `Concept`, predicados devem implementar a interface `Predicate` e ações de agentes devem implementar a interface `AgentAction`; (ii) devem espelhar as heranças especificadas na ontologia; (iii) devem ter os atributos especificados na ontologia e devem ser JavaBeans, ou seja, devem ter métodos de acesso para cada atributo. Além disso, vale destacar que JADE é um *framework* Java que também pode ser utilizado em dispositivos móveis. Assim, atributos cuja relação na ontologia tem cardinalidade maior que 1 devem ser do tipo `jade.util.leap.List`, uma lista de JADE correspondente à lista padrão de Java `java.util.List`, porém com aspectos que a fazem ser eficiente em dispositivos móveis.

Uma outra funcionalidade referente à definição de ontologias em JADE é a combinação de ontologias. Ou seja, é possível fazer com que uma nova ontologia estenda ontologias previamente definidas. Para isso, deve-se especificar as ontologias a serem estendidas no construtor da ontologia derivada e, a partir daí, todos os conceitos, predicados e ações de agentes definidos nas ontologias base podem ser reutilizados na ontologia derivada.

É interessante destacar que existe o *plugin* beangenerator (BEANGENERATOR, 2007) para o editor de ontologias Protégé (PROTÉGÉ, 2008), que permite que sejam gerados, a partir de uma ontologia definida no Protégé, tanto a definição da ontologia em JADE quanto a definição das classes correspondentes aos conceitos, predicados e ações de agentes.

2.4. Ambientes de Desenvolvimento de Software e o Ambiente ODE

Este trabalho está inserido no contexto de um Ambiente de Desenvolvimento de Software (ADS). ADSs são sistemas que buscam combinar técnicas, métodos e ferramentas para apoiar o Engenheiro de Software na construção de produtos de software, abrangendo todas as atividades do processo de software ou pelo menos porções significativas dele (FALBO, 1998) (HARRISON *et al.*, 2000).

Com o objetivo de fazer com que os ADSs apóiem as atividades, segundo um processo de software estabelecido, surge uma linha de pesquisa com foco na integração de processos, a de ADSs Centrados em Processos (ADSCP). ADSCPs podem ser vistos como a automatização de um processo de software e têm a responsabilidade de (CHRISTIE, 1995): (i) guiar a seqüência de atividades definida; (ii) gerenciar os produtos que estão sendo desenvolvidos; (iii) executar ferramentas necessárias para a realização das atividades; (iv) permitir comunicação entre as pessoas; (v) colher dados de métricas automaticamente; (vi) reduzir erros humanos; e (vii) prover controle do projeto à medida que este vai sendo executado.

Este trabalho se situa no contexto do Projeto ODE (*Ontology-based software Development Environment*) (FALBO *et al.*, 2003) (FALBO *et al.*, 2005b), em curso desde 2001 e que visa ao desenvolvimento de um Ambiente de Desenvolvimento de Software Centrado em Processo (ADSCP).

ODE possui várias ferramentas, dentre elas: de apoio à definição de processos de software (BERTOLLO *et al.*, 2006), acompanhamento de projetos (ControlPro) (DAL MORO *et al.*, 2005), gerência de recursos humanos (GerênciaRH), realização de estimativas (EstimaODE) (CARVALHO *et al.*, 2006) e gerência de riscos (GeRis) (FALBO *et al.*, 2004b).

A arquitetura física de ODE é composta por quatro pacotes, a saber: Componente de Domínio do Problema (cdp), Componente de Gerência de Tarefas (cgt), Componente de Gerência de Dados (cgd) e Componente de Gerência de Visão (cgv). Os pacotes cdp e cgt

estão relacionados com a lógica de negócio, sendo que o primeiro trata o domínio do problema, enquanto o segundo trata de funcionalidades, implementando casos de uso. O pacote cgd contém as classes relacionadas com persistência de objetos. Por fim, o pacote cgv contém classes relacionadas com a lógica de apresentação. Nesse pacote há dois tipos de classes: controladores e componentes de interface. Vale destacar que algumas ferramentas de ODE as tratam em pacotes separados, a saber: Componente de Controle de Interação (cci) e Componente de Interação Humana (cih). Dessa forma, pode-se fazer referência ao padrão MVC (Modelo – Visão – Controlador) (GAMMA *et al.*, 1995), de modo que os pacotes cdp e cgd correspondem ao Modelo, os pacotes cgt e cci correspondem ao Controlador e o pacote cih corresponde à Visão, assim como apresenta a Figura 2.4.

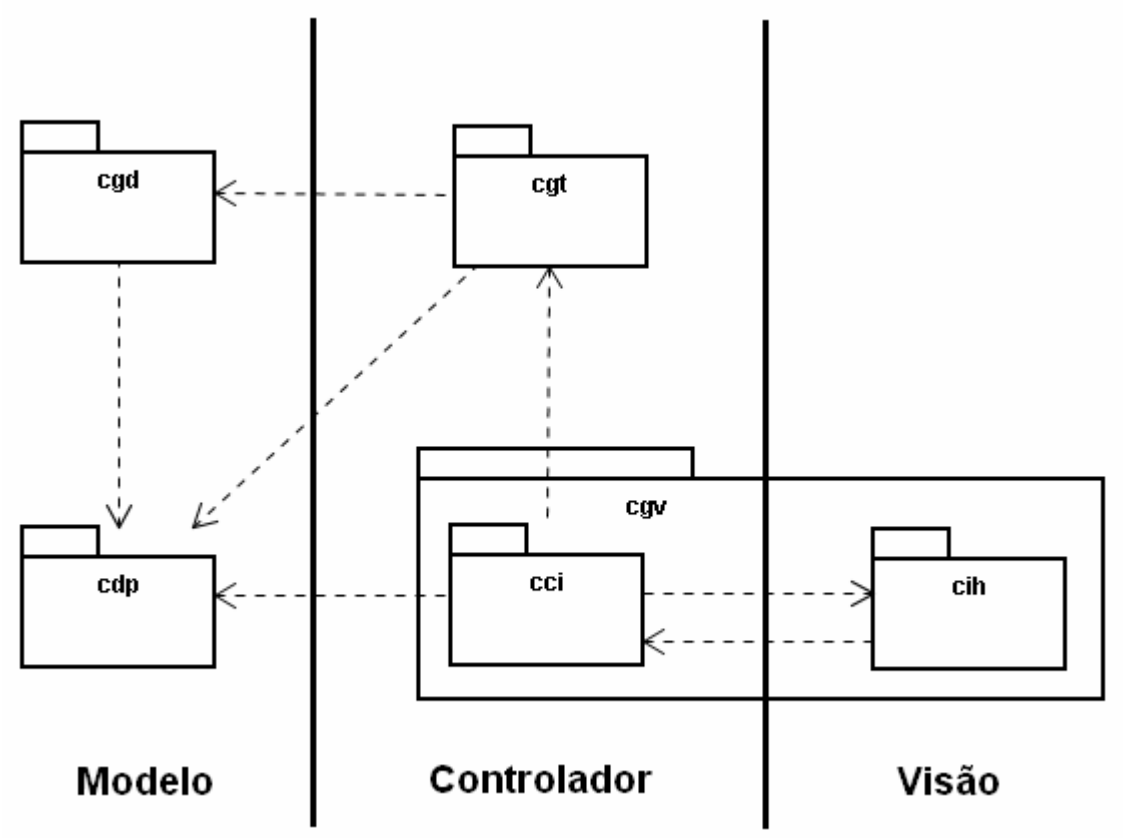

Figura 2.4 – Arquitetura física de ODE

É interessante ressaltar, também, uma outra visão da arquitetura em quatro pacotes de ODE, na qual os pacotes cdp e cgt estão relacionados com a lógica de negócio, o pacote cgv contêm classes relacionadas com a lógica de apresentação e o pacote cgd contém classes relacionadas com a lógica de acesso aos dados.

Além disso, há outras pesquisas relacionadas ao ambiente ODE que merecem destaque no contexto deste trabalho. São elas sua infra-estrutura de gerência de conhecimento e sua base ontológica.

Infra-Estrutura de Gerência de Conhecimento

É importante notar que organizações de software devem ser conduzidas como organizações que aprendem continuamente. O desenvolvimento de software é um esforço coletivo, complexo e criativo e, para desenvolver produtos de software de qualidade, as organizações de software devem utilizar seu conhecimento organizacional (RUY, 2006). Assim, destaca-se a Gerência de Conhecimento, que envolve a administração, de forma sistemática e ativa, dos recursos de conhecimento de uma organização, utilizando tecnologia apropriada (LIMA, 2004).

O´Leary *et al.* (2001) afirmam que, para ser eficaz, a gestão do conhecimento deve estar embutida nos processos de trabalho. Assim, os Ambientes de Desenvolvimento de Software podem ser vistos como os ambientes propícios para a introdução da abordagem de Gerência do Conhecimento na Engenharia de Software. Neste contexto, são relevantes conhecimentos sobre a própria área de Engenharia de Software, sobre a organização em que o ADS está implantado e sobre o domínio de aplicação em questão (NATALI *et al.*, 2002).

Em ODE, essa sinergia entre ADSCP e Gerência de Conhecimento tem sido bastante trabalhada (NATALI *et al.*, 2002) (NATALI *et al.*, 2003) (FALBO *et al.*, 2004a) (FALBO *et al.*, 2004b) (FALBO *et al.*, 2005a) (FALBO *et al.*, 2005b) (NUNES *et al.*, 2006) (CARVALHO *et al.*, 2006), explorando-se o uso de Gerência de Conhecimento para apoiar, de forma automatizada, algumas atividades do processo de software, tais como estimativas, gerência de riscos e alocação de recursos.

A abordagem de Gerência de Conhecimento de ODE (NATALI *et al.*, 2003) posiciona a memória organizacional no centro de sua infra-estrutura, apoiando o compartilhamento e reúso de conhecimento. Em torno da memória organizacional, se localizam os serviços que apóiam cada uma das atividades do processo de gerência de conhecimento, a saber, coleta, busca, disseminação e apoio ao reúso e à manutenção do conhecimento armazenado, como mostra a Figura 2.5.

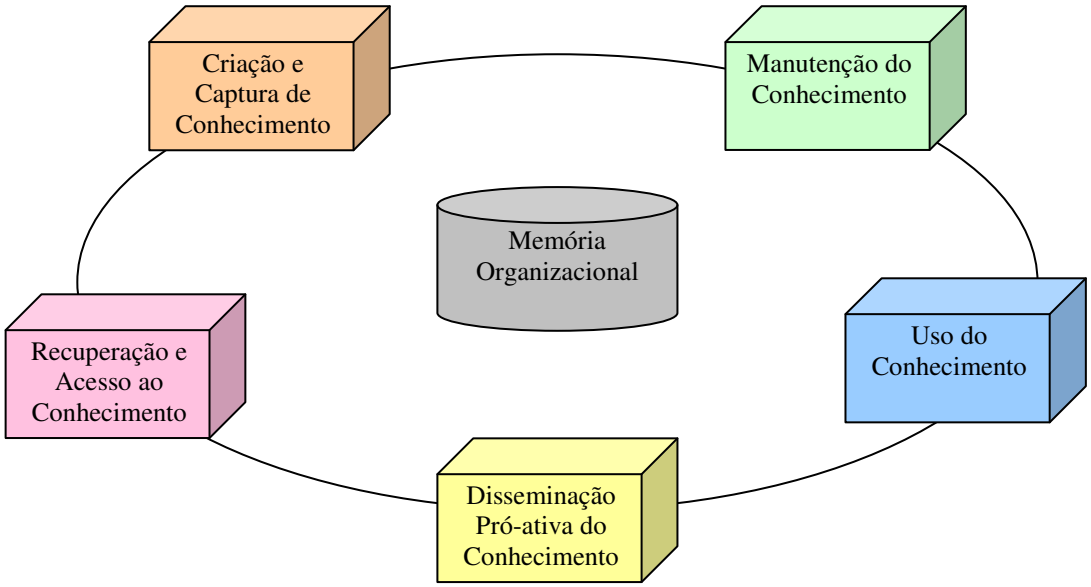

Figura 2.5 – Infra-estrutura de Gerência de Conhecimento Proposta para ODE.
Fonte: Natali *et al.* (2003)

Dentre as atividades da Gerência de Conhecimento (GC), uma das principais é a disseminação de conhecimento. Ela pode se dar de duas formas: passiva, quando os engenheiros de software buscam o conhecimento de que necessitam; ou ativa, quando o sistema de GC reconhece as necessidades do engenheiro de software e pró-ativamente apresenta itens de conhecimento relevantes para a realização da tarefa em curso. Conforme apontado em (ABECKER *et al.*, 1998) e (HOLZ, 2003), a primeira abordagem é insuficiente para organizações de software, porque muitas vezes os engenheiros de software não estão cientes de que existem itens de conhecimento relevantes ou porque estão tão ocupados que não têm tempo de procurar por eles. Neste caso, a disseminação pró-ativa é fundamental para o sucesso da GC e, neste contexto, agentes desempenham um importante papel. Na medida em que a disseminação pró-ativa diz respeito à apresentação de conhecimento relevante por iniciativa do sistema, a tecnologia de agentes emerge como uma abordagem potencial para lidar com esse problema (NATALI *et al.*, 2002).

Dessa forma, agentes de software são utilizados para ligar os membros da organização ao conhecimento disponível, auxiliando não apenas na busca por conhecimento, mas também no filtro de conhecimento relevante e sua disseminação. Em um processo de trabalho definido, agentes atuam de maneira pró-ativa, buscando e oferecendo itens de conhecimento que podem ser relevantes para a tarefa que o usuário está executando (SCHWAMBACH, 2004).

Base Ontológica

Em ODE, parte-se do pressuposto que, se as ferramentas de um ADS são construídas baseadas em ontologias, a integração delas pode ser facilitada, pois os conceitos envolvidos são bem definidos pelas ontologias (FALBO *et al.*, 2003). Uma ontologia é a especificação de uma conceituação (GRUBER, 1995), que consiste, basicamente, de conceitos e relações, suas definições, propriedades e restrições, descritos na forma de axiomas. Uma ontologia não é somente uma hierarquia de termos, mas uma teoria completa axiomatizada de um domínio (GUIZZARDI *et al.*, 2001).

Dentre as ontologias que compõem a base ontológica de ODE, tem-se as ontologias de processos de software (BERTOLLO, 2006), de qualidade de software (DAL MORO, 2008), de artefatos de software (NUNES, 2005), de gerência de configuração de software (ARANTES *et al.*, 2007), de riscos de software (FALBO *et al.*, 2004b) e de requisitos (NARDI *et al.*, 2006).

Por ser baseado em ontologias, ODE realiza grande parte de suas tarefas utilizando e respeitando os modelos e restrições impostos pelas ontologias sobre as quais se fundamenta. Assim, tem-se tentado implementar essa estreita relação entre ambiente e ontologias de uma forma natural, uma vez que há uma enorme dependência do ambiente em relação às ontologias. Dado que ODE é desenvolvido usando a tecnologia de objetos, optou-se por realizar um mapeamento de ontologias para modelos de objetos, aplicando-se parcialmente a abordagem sistemática de derivação de infra-estruturas de objetos a partir de ontologias, descrita em (FALBO *et al.*, 2002). Essa abordagem permite que os elementos definidos em uma ontologia (conceitos, relações, propriedades e restrições definidas como axiomas) sejam mapeados para um modelo de objetos que é, então, integrado como parte fundamental da estrutura do ambiente.

Desse modo, a instanciação das ontologias de ODE dá origem a uma parte importante do "conhecimento" do ambiente. Esse conhecimento é usado para apoiar o usuário em diversas tarefas, tais como definição de processos, alocação de recursos, avaliação de qualidade, gerenciamento de riscos etc (RUY, 2006). No contexto deste trabalho, deve-se ressaltar que as ontologias também definem o vocabulário comum a ser utilizado nas mensagens trocadas entre agentes que atuam no ambiente e, portanto, adicionam significado ao conteúdo das mensagens.

2.5. AgeODE

Em um primeiro momento, ODE possuía alguns agentes de software desenvolvidos para apoiar algumas atividades do processo de software, atuando em ferramentas específicas (PEZZIN, 2002) (SCHWAMBACH, 2002) (NATALI, 2003). Alguns problemas foram encontrados no desenvolvimento desses agentes em ODE, a saber: (i) agentes não possuíam autonomia, uma vez que eram tratados como objetos e, portanto, não possuíam uma linha de controle própria nem eram tratados como processos separados; (ii) cada agente era construído de uma maneira diferente por cada um dos desenvolvedores. Não havia padronização nem uniformidade na construção dos agentes, o que provocava problemas de integração; (iii) finalmente, cada desenvolvedor partia do zero na árdua tarefa de se construir agentes, não havendo, deste modo, nenhuma forma de reúso (PEZZIN, 2004).

Ainda que existam vários *frameworks* de apoio à construção de agentes (COLLIS, 1999) (COST *et al.*, 1999) (JEON *et al.*, 2000) (BELLIFEMINE *et al.*, 2001) (DING *et al.*, 2001) (JACK, 2004) (BIGUS, 2000) (RETSINA, 2004) (AGENTBUILDER, 2004), de modo geral, eles não se destinam a ser especificamente um *framework* para construção de agentes que atuarão em um ADS e, conseqüentemente, não tratam aspectos relacionados à integração de agentes em um ADS (PEZZIN, 2004).

Buscando preencher essa lacuna, foi desenvolvida, originalmente em (PEZZIN, 2004), AgeODE, uma infra-estrutura para apoiar a construção de agentes para atuarem no ambiente ODE. Uma infra-estrutura de agentes pode ser vista como um *framework* para a construção de agentes de software. Assim, pode-se definir uma infra-estrutura para construção de agentes como sendo um arcabouço (de classes) por meio do qual se podem desenvolver agentes heterogêneos capazes de se comunicar, colaborar e trocar serviços para alcançar os objetivos do sistema multiagente no qual estarão inseridos.

Deve ser ressaltado que, assim como *frameworks* ou infra-estruturas, padrões de agentes também são úteis na construção de agentes, por introduzirem, também, uma forma de reúso. Em Engenharia de Software, padrões de software são tentativas de descrever soluções de sucesso para problemas recorrentes. Desenvolvedores não inventam padrões de software; ao invés disso, eles descobrem padrões por meio de experiências em sistemas práticos construídos (DEVEDZIC, 1999). Assim, um padrão é uma solução comprovada para um problema num contexto (COPLIEN, 1995).

Padrões e *frameworks* tratam de reúso, mas são bastante diferentes. Um padrão descreve um problema a ser resolvido, uma solução e o contexto no qual a solução funciona.

Frameworks, por outro lado, são mais que idéias, eles são também código e um único *framework*, geralmente, contém muitos padrões (JOHNSON, 1997a) (JOHNSON, 1997b).

Desse modo, AgeODE foi definida a partir da análise de padrões de agentes e do levantamento dos requisitos de uma infra-estrutura para construção de agentes em um ADS, buscando contemplar os seguintes requisitos (PEZZIN, 2004):

- por agentes poderem apresentar características e interesses diferentes uns dos outros dentro do ambiente onde atuam, é interessante que uma infra-estrutura para construção de agentes proveja diferentes classes de agentes. Tais classes de agentes devem levar em conta as particularidades do ambiente em que os agentes atuarão, no caso, um ADS;
- definir como se dará a comunicação entre agentes;
- ainda que diferentes agentes tenham características diferentes, visando integração, uniformidade e reúso, deve ser definida uma arquitetura básica interna;
- uma vez que os agentes atuarão em um ADS, parte de suas bases de conhecimento será constituída de objetos do próprio ADS e, portanto, deve-se definir como agentes terão acesso a esses objetos do ambiente.

A partir dos requisitos identificados, a infra-estrutura foi definida, levando-se em conta vários aspectos, entre eles: como se dá a comunicação entre os agentes, como os agentes têm acesso aos objetos do domínio do problema para compor sua base de conhecimento e como é a arquitetura interna dos agentes. Foram definidos, também, alguns tipos específicos de agentes úteis para ADSs, a saber: (i) agentes de interface, (ii) agentes de usuário, (iii) agentes de informação e (iv) agentes coordenadores (PEZZIN, 2004).

Agentes de Interface

Uma das formas mais básicas de se monitorar as ações de um usuário em um ADS é observar o que ele faz, monitorando os eventos solicitados por meio da interface com o usuário. Assim, surge a necessidade de um agente de interface. Por exemplo, pode-se querer saber qual das ferramentas do ambiente foi aberta pelo usuário ou qual atividade ele selecionou para trabalhar. Neste caso, monitorando as interações ocorridas na interface do ADS, um agente pode capturar essas ações (PEZZIN, 2004).

Um agente de interface deve fornecer ao usuário do ADS uma interface mais amigável, com características pró-ativas e que detecte as ações do usuário quanto ao uso da interface. Ao gerenciar interfaces gráficas com o usuário, esse agente deve transformar

comandos do usuário em objetivos, realizar ações e exibir os resultados (BRUGALI *et al.*, 2000).

Desse modo, sua tarefa principal é monitorar as ações do usuário. Um agente de interface deve ser capaz de observar as ações que o usuário realiza na interface do ambiente e de capturá-las, sabendo, a partir delas, como agir diante de uma situação. Além disso, ele deve ser capaz de adaptar a interface às necessidades identificadas pelo perfil do usuário, estabelecendo padrões de uso mais adequados (KENDALL *et al.*, 1998).

Agentes de Usuário

Uma vez que um ADS é usado por vários usuários, é importante que o ambiente possua uma forma de distingui-los de acordo com suas características, ou seja, de acordo com os papéis que desempenham, tais como gerente de projeto, engenheiro de software, gerente de configuração etc, e seus interesses de uso do ambiente. Por exemplo, se um usuário é um gerente de projeto, seria interessante que lhe fossem fornecidas pró-ativamente informações sobre o projeto que ele gerencia e lhe fossem dadas sugestões para apoiar o seu gerenciamento. Isso poderia ser feito por meio de um agente de usuário (PEZZIN, 2004).

Um agente de usuário deve usar o conhecimento que possui sobre determinado usuário para permitir ou não a execução de atividades por ele. Ele deve ser capaz de estabelecer o perfil do usuário, buscando informações pertinentes ao usuário no contexto do ADS ou de um projeto específico. Um agente de usuário pode interagir com o usuário que ele representa, auxiliando-o na realização de atividades e na tomada de decisões. É como se esse agente incorporasse o papel do usuário, passando a realizar algumas de suas atividades e o ajudando na realização de outras (PEZZIN, 2004).

Agentes de Informação

Agentes de informação ou de domínio são responsáveis pela realização de alguma funcionalidade do sistema. Sua tarefa principal é buscar informações e realizar tarefas dentro do domínio de problemas do ADS, visando satisfazer necessidades de processamento do sistema. Muitas vezes essas tarefas são bastante complexas e uma abordagem de solução distribuída de problemas pode ser útil, necessitando-se, assim, de agentes para realizá-las (PEZZIN, 2004). Para tal, pode ser necessário guardar a origem da informação, tal como um banco de dados ou um servidor *web* (BRUGALI *et al.*, 2000).

Por exemplo, quando se adota uma abordagem baseada em gerência de conhecimento, muitas vezes, parte importante do processamento diz respeito à recuperação de projetos similares a um projeto dado. Para um caso como esse, pode-se ter um agente de informação que tenha essa responsabilidade (PEZZIN, 2004).

Agentes Coordenadores

Como em grupos de pessoas, em que, muitas vezes, há a necessidade de se ter um coordenador do grupo, distribuindo tarefas e, posteriormente, cobrando e reunindo os resultados, num sistema multiagente também existe tal necessidade. Principalmente quando se tem uma tarefa complexa que pode ser dividida em sub-tarefas, que, por sua vez, são distribuídas para diferentes agentes, agentes coordenadores são necessários. Mas não basta só distribuir as sub-tarefas aos agentes. Deve-se, também, conhecer quais agentes são capazes de executar determinadas sub-tarefas. De fato, em sistemas multiagente, muitas das informações parciais ou específicas decorrentes da atuação dos agentes está dispersa na sociedade. Cada um dos agentes envolvidos na resolução do problema possui um conjunto de resultados parciais de seu próprio processamento (PEZZIN, 2004).

Assim, um agente coordenador deve ser capaz de coordenar as tarefas a serem executadas em um dado contexto do ADS, distribuí-las, quando necessário, aos agentes capazes de executá-las, consolidar determinado conjunto de resultados recuperados de um ou mais agentes dispersos na sociedade (JUCHEM *et al.*, 2002) e conhecer agentes com capacidades/habilidades específicas que estão sob seu domínio de coordenação. De fato, ele oferece uma maneira de definir a coordenação de múltiplas tarefas a serem executadas, promovendo reúso de tarefas, designação dinâmica de tarefas a agentes e, ainda, a composição de tarefas (ARIDOR *et al.*, 1998).

No contexto de ADSs, ferramentas que impõem um certo passo-a-passo para a realização de uma atividade e que têm diferentes agentes atuando em cada um desses passos necessitam de um agente coordenador para coordenar suas tarefas e distribuí-las aos demais agentes da ferramenta (PEZZIN, 2004). Isso ocorre, por exemplo, com a ferramenta de gerência de riscos de ODE, GeRis (SCHWAMBACH, 2004), na qual diferentes agentes atuam em diferentes sub-atividades da gerência de risco, a saber: identificação de riscos, análise de riscos, definição dos riscos a serem gerenciados e planejamento de ações de mitigação e de contingência.

2.6. Conclusões do Capítulo

Por serem sistemas de software complexos, ADSs são potenciais beneficiários da tecnologia de agentes. Como um exemplo da utilidade dessa tecnologia, pode-se citar a disseminação de conhecimento na gerência de conhecimento. Assim, em ODE agentes são utilizados para aperfeiçoar algumas das funcionalidades do ambiente.

Deve-se ressaltar que a integração é uma questão fundamental para ADSs (HARRISON *et al.*, 2000), (TRAVASSOS, 1994), (FALBO *et al.*, 2003) e, portanto, a construção de agentes para atuarem no contexto de ADSs deve considerar que os agentes desenvolvidos devem apresentar homogeneidade na forma de apresentação, comunicação e atuação (PEZZIN, 2004).

Pensando nisso, foi construída uma infra-estrutura de apoio à construção de agentes em ODE, chamada AgeODE (PEZZIN, 2004). AgeODE busca a integração, uma vez que todos os agentes compartilham a mesma forma de comunicação com outros agentes, a mesma forma de acesso aos objetos que compõem o ambiente e uma arquitetura interna básica (PEZZIN, 2004).

Em 2006, o ambiente ODE passou por uma reestruturação em sua arquitetura para torná-lo multi-usuário, tendo sido a sua camada de persistência alterada. Essa reestruturação trouxe impactos em diversas aplicações do ambiente, dentre elas AgeODE. Por conta dessa manutenção, aproveitou-se para evoluir essa infra-estrutura, de modo a torná-la mais flexível e poderosa. A evolução de AgeODE é a principal contribuição deste trabalho e é discutida no próximo capítulo.

Capítulo 3

Evolução de AgeODE

O ambiente de desenvolvimento ODE (*Ontology-based software Development Environment*) (FALBO *et al.*, 2003) possui uma infra-estrutura para apoiar a construção de agentes autônomos para atuarem em ODE, desenvolvida originalmente em (PEZZIN, 2004), denominada AgeODE. AgeODE define alguns tipos específicos de agentes úteis para ADSs, como se dá a comunicação entre eles, como os agentes têm acesso aos objetos do domínio do problema para compor sua base de conhecimento e como é a arquitetura interna dos mesmos.

A primeira versão de AgeODE foi desenvolvida sobre o *framework* para construção de agentes JAT*Lite* (JEON *et al.*, 2000) (JATLITE, 2004) e apresentava alguns problemas, com destaque para o desempenho do ambiente quando executado com agentes.

Na busca por soluções para os problemas detectados, chegou-se à conclusão que JAT*Lite*, o *framework* para a construção de agentes no qual estava fundamentada a primeira versão de AgeODE, apresenta diversos problemas, sendo o mais grave o fato do projeto que desenvolvia o *framework* ter sido desativado e, por conseguinte, o mesmo estava defasado em relação a outros *frameworks*. Uma pesquisa foi feita, então, buscando uma alternativa de *framework*, chegando-se ao *framework* JADE.

Constatou-se que diversos serviços poderiam ser muito aperfeiçoados, na medida em que JADE é um *framework* muito mais poderoso que JAT*Lite*. Dessa forma, foi feita uma reengenharia de AgeODE, de modo a incorporar os principais recursos de JADE, além de corrigir algumas falhas da versão original e adaptar as suas principais idéias para a nova versão.

A organização deste capítulo se dá da seguinte maneira: a seção 3.1 – Desenvolvimento da Versão Original de AgeODE – apresenta a versão original da infra-estrutura, que tinha por base o *framework* JAT*Lite*; a seção 3.2 – Oportunidades de Melhoria Identificadas na Versão Original – se volta para identificar problemas e oportunidades de melhoria na versão original, visando a evoluir AgeODE; a seção 3.3 – A Nova Versão de

AgeODE – discute as abordagens adotadas para se adotar JADE como novo *framework* base, levando-se em conta as oportunidades de melhoria anteriormente detectadas; por fim, a seção 3.4 apresenta as considerações finais e conclusões do capítulo.

3.1. Desenvolvimento da Versão Original de AgeODE

A construção de uma infra-estrutura para construção de agentes não é uma tarefa fácil. Portanto, originalmente optou-se por construir AgeODE como uma camada sobre JAT*Lite* (JEON *et al.*, 2000) (JATLITE, 2004), uma vez que esta possui uma implementação simples e de fácil entendimento, contemplando a comunicação entre os agentes usando KQML (FININ *et al.*, 1995), o que permitiu não se partir do zero para se desenvolver a infra-estrutura (PEZZIN, 2004).

JAT*Lite* (*Java Agent Template, Lite*) (JEON *et al.*, 2000) (JATLITE, 2004) é um *framework* para construção de agentes que oferece um conjunto de pacotes escritos na linguagem Java, permitindo criar agentes de software. Esses agentes se registram junto a um roteador de mensagens e, a partir daí, enviam e recebem mensagens usando a linguagem de comunicação KQML. Para isso, o *framework* utiliza as construções de *multithreading* e *sockets* oferecidas pela linguagem Java.

JAT*Lite* organiza as classes do *framework* em camadas, conforme mostra a Figura 3.1, de modo que os desenvolvedores possam escolher quais camadas seus agentes implementarão. As camadas de JAT*Lite* são:

- *Camada Abstrata*: fornece uma coleção de classes abstratas necessárias para a implementação de JAT*Lite*;
- *Camada Básica*: fornece a comunicação básica baseada em TCP/IP e na camada abstrata;
- *Camada KQML*: efetua o armazenamento e *parsing* de mensagens KQML, bem como os serviços de registro, conexão e desconexão;
- *Camada Roteadora*: fornece registro de nomes e roteamento e enfileiramento de mensagens para os agentes;
- *Camada de Protocolo*: dá suporte a vários serviços de *Internet*, tais como SMTP, FTP, POP3, HTTP etc, tanto para aplicações *stand-alone* quanto para *applets*.

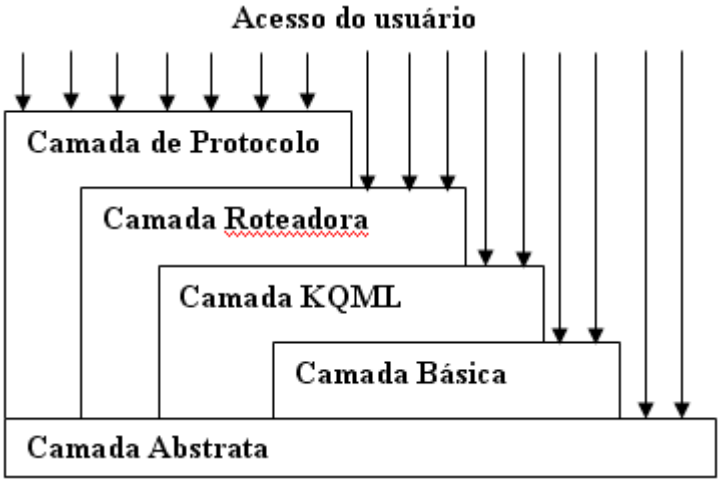

Figura 3.1 – Camadas de JAT*Lite*

Para implementar a versão original de AgeODE, foram usadas apenas as quatro primeiras camadas. Não houve necessidade de se usar a camada de Protocolo, uma vez que havia sido usado o protocolo padrão de comunicação, já que, naquele momento, ODE era uma aplicação mono-usuário. Entretanto, naquela época já havia perspectivas de torná-lo uma aplicação multi-usuário, o que fez Pezzin (2004) apontar que seria necessário posteriormente incorporar essa última camada a AgeODE.

A idéia de construir AgeODE como uma camada sobre JAT*Lite*, em que classes de AgeODE herdam de classes de JAT*Lite*, mesmo que não acrescentando nenhuma funcionalidade específica nas sub-classes, visa a criar uma camada, ou fachada, mantendo isolamento entre JAT*Lite* e AgeODE e fazendo com que outros desenvolvedores de agentes em ODE não precisem conhecer detalhes da arquitetura interna de JAT*Lite*. Além disso, caso alguma manutenção se fizesse necessária, poderia ser realizada apenas nas classes de AgeODE (PEZZIN, 2004).

JAT*Lite* separa os agentes entre clientes e roteadores, de forma que o roteador é responsável por receber mensagens de agentes clientes e por direcioná-las para os destinatários. Assim, o agente roteador necessita de um repositório de mensagens para salvá-las e de um mecanismo de roteamento para enviar as mensagens, assim que haja o estabelecimento da conexão com o destinatário.

Entretanto, JAT*Lite* não possui uma classificação para agentes. Além disso, JAT*Lite* não contempla acesso a objetos (nem tampouco trata de como é constituída a base de conhecimento dos agentes) e não aponta muitos detalhes sobre a arquitetura dos mesmos. Assim, a parte de comunicação de JAT*Lite*, no que se refere ao roteamento de mensagens

entre agentes, endereçamento e registro de agentes, foi aproveitada integralmente na versão original de AgeODE, enquanto outros aspectos tiveram que ser incorporados à infra-estrutura (PEZZIN, 2004).

Dessa forma, percebeu-se que a construção de uma infra-estrutura destinada a apoiar a construção de agentes em ODE traz consigo alguns problemas inerentes. Tendo em vista que as soluções da versão original de AgeODE para esses problemas são importantes no contexto deste trabalho, discutiremos algumas delas, a saber: (i) arquitetura de AgeODE; (ii) a comunicação entre agentes; (iii) o acesso a objetos do ambiente; (iv) a captura de eventos da interface com o usuário; e (v) a apresentação de sugestões dos agentes.

Arquitetura de AgeODE

A comunicação entre os agentes de JAT*Lite* se baseia em um modelo cliente-servidor, em que agentes clientes usam o serviço de roteamento oferecido por um agente servidor, denominado roteador. Assim, ligadas às classes de JAT*Lite*, havia duas classes de AgeODE: *AgCliente* e *AgRoteador* (PEZZIN, 2004).

AgCliente incorporava a AgeODE características de agentes clientes de JAT*Lite*, oferecendo serviços para trocar mensagens entre agentes. A idéia era aproveitar o que JAT*Lite* fornecia em termos de comunicação, a saber, JAT*Lite* permite a troca de informações sem que seja necessário se preocupar com a comunicação propriamente dita, que fica a cargo de um agente roteador. Assim, os agentes clientes, ao interagirem entre si, não se reportavam diretamente uns aos outros, o faziam sempre por meio do agente roteador. A classe *AgRoteador*, por sua vez, herdava da classe *RouterAction* de JAT*Lite*, que funciona como um roteador de mensagens e que aguarda que agentes clientes se conectem a ele.

Dessa forma, um agente cliente podia se registrar juntamente a um agente roteador e trocar mensagens com outros agentes clientes, enquanto um agente roteador possuía uma tabela de endereços de agentes clientes que usavam seus serviços de roteamento e uma tabela das conexões de agentes ativas num dado momento.

Conforme discutido anteriormente, é importante que os agentes clientes de AgeODE compartilhem uma arquitetura básica. Assim, foi definido que os agentes clientes deveriam possuir, entre outros (PEZZIN, 2004):

- Atributos gerais, obrigatórios para todos os agentes clientes, o que incluía: nome e endereço;

- Outros atributos, que podiam ser objetos de ODE de interesse do agente, uma vez que estes constituem parte de sua base de conhecimento;
- Operações, representando atividades dos agentes;
- Protocolos, que definiam como um agente pode interagir com outros agentes, especificando entre quais agentes ocorre a comunicação, qual o conteúdo dessa comunicação, qual a ontologia, qual a performativa etc. Toda comunicação entre agentes clientes requeria a execução do protocolo básico *enviarMensagem*;
- Capacidade de observar o ambiente de forma autônoma, quando pertinente, sendo que, para tal, podia ser necessário capturar eventos da interface com o usuário por meio de observadores, como discutido mais à frente;
- Autonomia para tomar decisões junto ao ADS no qual atuam. Para isso, cada agente era implementado como um processo independente.

Uma vez que diferentes agentes podem apresentar formas de atuação bastante diferentes, é interessante que AgeODE proveja um conjunto básico de classes de agentes abrangendo as formas de atuação mais comuns em um ADS. Especializando *AgCliente*, como mostra a Figura 3.2, utilizando as extensões de UML definidas em OplA (SCHWAMBACH, 2004), definiu-se esses tipos de agentes. Vale destacar que a versão original de AgeODE também possuía o tipo roteador (*AgRoteador*). Contudo, esse tipo de agente estava associado ao funcionamento da infra-estrutura e, de maneira geral, era transparente para os seus usuários (PEZZIN, 2004).

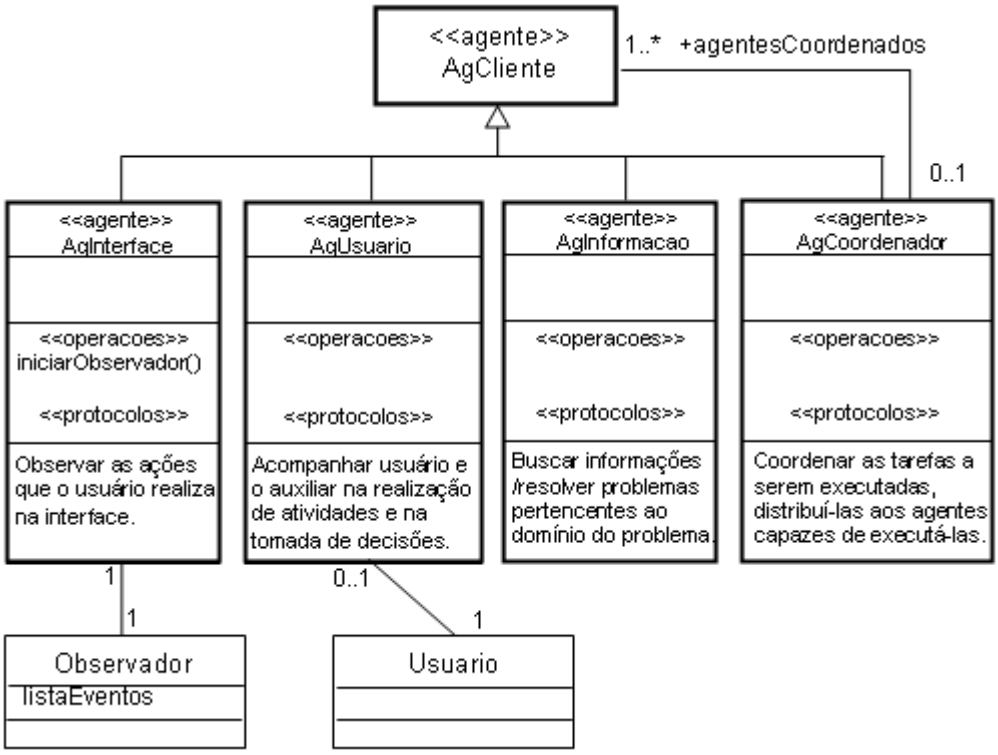

Figura 3.2 – Diagrama de Classes dos Tipos de Agentes da versão original de AgeODE
Fonte: Pezzin (2004)

Os agentes de usuário (*AgUsuario*) acompanham um usuário específico de ODE, auxiliando-o na realização de atividades e na tomada de decisões. Assim, eles tinham uma referência a um objeto do tipo *Usuário*, classe do ambiente ODE.

Já os agentes de interface (*AgInterface*), por necessitarem capturar eventos da interface com o usuário, tinham que ter associado a eles um objeto *Observador*. O observador agia como se fosse o mecanismo de percepção de um agente de interface no ambiente ODE, interceptando os eventos de interface. Dessa forma, quando um evento de interface era interceptado pelo observador, ele avisava ao agente, para que esse pudesse ficar a par do que estava acontecendo e, assim, pudesse tomar decisões.

Os agentes de informação (*AgInformacao*) eram responsáveis por buscar informações e realizar tarefas nos domínios de problema das ferramentas de ODE. Uma vez que são muito distintas as tarefas que potencialmente podem ser resolvidas por agentes desse tipo, sua estrutura interna era a mesma dos agentes clientes (*AgCliente*), contendo apenas os mecanismos básicos de comunicação e registro.

Finalmente, os agentes coordenadores (*AgCoordenador*) são responsáveis por coordenar as tarefas a serem executadas por um conjunto de agentes, ditos agentes coordenados, em um dado contexto de ODE. Tipicamente, são úteis para coordenar os

trabalhos em tarefas que possuem um processo inerente a elas, enquanto outros agentes ficam responsáveis por realizar certos passos desse processo.

Quando um agente assumia o perfil de mais de um tipo de agente, ele devia herdar de mais de uma classe de tipo de agente de AgeODE. Por exemplo, um agente pode ser dos tipos de usuário e de informação e, portanto, teria que herdar de *AgUsuario* e de *AgInterface*. Contudo, vale lembrar que, como os agentes eram implementados em Java, uma linguagem que não suporta herança múltipla, esses agentes tinham que herdar apenas da classe que representava seu tipo principal e deviam implementar as interfaces de seus demais tipos. Assim, além das classes mostradas na Figura 3.2, cada tipo de agente tinha uma interface associada.

Comunicação entre Agentes

Uma vez que, na versão original de AgeODE, a parte relativa à comunicação utilizava o *framework* JAT*Lite*, as mensagens KQML em AgeODE, assim como em JAT*Lite*, eram implementadas da seguinte forma: cada mensagem era uma estrutura que possuía vários campos do tipo string, um para cada parâmetro da mensagem. Para compor uma mensagem KQML, um agente devia preencher os campos correspondentes da mensagem e enviá-la. Ao receber uma mensagem de outro agente, o agente receptor devia interpretá-la conforme a indica a sintaxe de KQML.

O conteúdo das mensagens era escrito na linguagem XML (*Extensible Markup Language*). XML é uma meta-linguagem de marcação de dados, que provê um formato para descrever dados estruturados. Optou-se por utilizar XML na versão original de AgeODE por ser uma linguagem padrão, com marcações que permitem adicionar informação sobre o conteúdo de uma mensagem, o que facilitava o tratamento das mensagens pelos agentes (PEZZIN, 2004).

Além disso, como ODE é um ADS baseado em ontologias, a versão original de AgeODE definia que as mensagens KQML tinham que descrever o campo referente a ontologias, informando as ontologias envolvidas na comunicação. Assim, para que um agente pudesse se comunicar efetivamente com outro, eles deviam conhecer as ontologias em questão (PEZZIN, 2004).

Dessa forma, utilizando o exemplo de um agente perguntando a outro qual é o projeto corrente (o projeto aberto no ambiente ODE), são descritos, a seguir, alguns dos passos realizados pelos agentes para que pudessem efetivamente realizar a interação (PEZZIN, 2004):

(i) Ao receber uma mensagem, o agente receptor tinha que executar sua operação *interpretarMensagem*, usando, como parâmetro, a mensagem recebida. Em um primeiro momento, essa operação verificava qual era a performativa da mensagem. Se a performativa fosse, por exemplo, *ask-one*, ou seja, uma pergunta, a operação verificava se o agente podia respondê-la, comparando a ontologia da mensagem com as ontologias que o agente conhecia e verificando se o agente tinha em sua base de conhecimento uma resposta para a pergunta. Ou seja, a operação procurava o conteúdo da mensagem na lista de questões às quais o agente podia responder. Por exemplo, "Qual é o projeto?" era escrita em XML como `<Ode.Controle.Cdp.Projeto/>`, ou seja, o nome completo da classe *Projeto*.

(ii) Caso o agente soubesse responder à pergunta, o método *interpretarMensagem* formulava uma resposta ao agente remetente utilizando-se a performativa `reply` e a enviava. Caso o agente não conseguisse ou não quisesse respondê-la, ele enviava uma mensagem de desculpas, utilizando a performativa *sorry*.

(iii) Conforme dito anteriormente, uma vez que o conteúdo das mensagens era escrito em XML, era necessário que o agente remetente descrevesse a pergunta, que, na versão original de AgeODE, se tratava de uma indicação do objeto solicitado, referindo-se ao caminho completo da classe de que ele era instância. Além disso, era necessário que o agente receptor da mensagem formulasse uma resposta, utilizando como conteúdo da mensagem o identificador do objeto correspondente, de modo que o agente remetente, ao receber e interpretar a resposta, pudesse obter, a partir do banco de dados, o objeto em questão por meio de seu identificador.

Acesso a Objetos do Ambiente

A comunicação entre agentes e objetos, ao contrário da comunicação entre agentes, não exigia o que Pezzin (2004) e Schwambach (2004) chamavam de um protocolo de comunicação. No caso da comunicação entre agentes e objetos, agentes invocavam métodos dos objetos diretamente, abrindo um canal de comunicação entre um agente e ODE, utilizando *sockets*. Além disso, é importante frisar que objetos do ambiente não tinham acesso aos agentes, de modo a preservar a autonomia dos mesmos.

Um *socket* pode ser visto como um "bocal virtual", onde agentes e objetos podem se conectar para introduzir e retirar bits, sendo que existe um *socket* nos dois pontos finais de uma conexão. O uso de *sockets* em AgeODE se dava tanto na comunicação entre agentes, quanto na comunicação entre agentes e objetos. Mas isso era transparente para um

desenvolvedor de agentes usando AgeODE, pois ele só precisava utilizar os métodos para enviar mensagens entre agentes e os métodos que obtinham objetos de ODE (PEZZIN, 2004).

Deve ser salientado que é um requisito de ODE que o usuário possa escolher se deseja executar o ambiente com ou sem o apoio de agentes. Assim, esse controle era feito por meio da classe *AplControlarAgentes*, que também se encarregava de controlar os acessos de agentes aos objetos de ODE. Portanto, a *AplControlarAgentes* era responsável por controlar as conexões dos agentes junto a ODE (que eram implementadas via *sockets*), receber os pedidos de acesso aos objetos de ODE, solicitar aos objetos em questão que os métodos pedidos fossem executados e enviar o retorno dos métodos aos agentes solicitantes. Assim, deve-se frisar que, na versão original de AgeODE, cabia ao desenvolvedor do agente codificar o método do objeto a ser acessado, especificando o nome completo da classe do objeto, além do nome do método (PEZZIN, 2004).

Além disso, a classe *AplControlarAgentes* tinha outras responsabilidades, entre elas, iniciar os objetos observadores dos agentes de interface e iniciar uma janela padrão de sugestões de agentes, conforme discutido a seguir.

Captura de Eventos da Interface com o Usuário

Conforme apontado anteriormente, agentes de interface tinham que ter associado a eles um objeto `Observador`, que age como seu mecanismo de percepção do ambiente. Para ilustrar a observação de um agente de interface na versão original de AgeODE, seja o seguinte exemplo: quando um usuário iniciava uma sessão de ODE usando agentes, um Agente Assistente Pessoal (*AgAssistentePessoal*) era iniciado e passava a monitorar a interface do ambiente, interceptando algumas ações que o usuário realizava no ambiente. Sendo assim, o Agente Assistente Pessoal era um agente de interface e, portanto, tinha um observador associado a ele, denominado *ObsAgAssistentePessoal*.

O Agente Assistente Pessoal necessitava ser notificado quando ocorriam alguns eventos na interface de ODE, tal como, o clique do botão OK da Janela Abrir Projeto. Assim sendo, logo após um usuário iniciar uma sessão de ODE, o observador *ObsAgAssistentePessoal* obtinha a classe do botão OK da janela por meio de reflexão computacional e instalava um interceptador, que o notificava sempre que houvesse um clique nas instâncias desse botão. Se o botão fosse acionado, o observador enviava uma mensagem, via *socket*, ao Agente Assistente Pessoal informando que um projeto foi aberto (PEZZIN, 2004).

Como citado anteriormente, é um requisito de usabilidade de ODE poder ser executado sem agentes, uma vez que cabe ao usuário decidir se é de seu interesse receber ajuda pró-ativa dos agentes ou não. Se um usuário de ODE decidir executar o ambiente sem o apoio de agentes, ODE terá apenas suas funcionalidades habituais. Caso ele decida usar agentes, ODE contará com a ajuda dos mesmos para apoiar a realização de algumas tarefas no ambiente. Assim, somente os agentes têm acesso aos objetos de ODE. O contrário não é verdade. Agentes atuam dentro do ambiente ODE sem que o próprio ambiente tenha conhecimento disso (PEZZIN, 2004).

Entretanto, esse fato implicava diretamente na implementação dos agentes (PEZZIN, 2004). Podia ser necessário fazer uma atualização no código de um agente, caso a implementação de alguma das classes do domínio do problema a que ele tinha acesso fosse alterada (o que era visto como uma coisa natural, já que parte de sua base de conhecimento estava sendo alterada). Além disso, é importante ressaltar que uma alteração em um agente nunca implicava uma alteração em uma das classes de ODE.

Apresentação de Sugestões dos Agentes

A versão original de AgeODE tinha uma interface padrão para que os usuários de ODE pudessem visualizar as sugestões dos agentes, permitindo que os agentes interagissem com os usuários sem ter que conhecer ou interferir diretamente nas interfaces gráficas de ODE (PEZZIN, 2004).

A janela exibida na Figura 3.3 era, de fato, uma janela genérica de sugestões de agentes. Sempre que ODE fosse executado com agentes, essa janela se abria logo após a entrada do usuário no ambiente. A partir daí, um agente que quisesse enviar uma mensagem de aviso ou sugestão para o usuário devia publicá-la nessa interface. Na mesma figura, pode-se perceber, ainda, um ícone com uma "carinha" no canto superior direito da janela principal de ODE. Sempre que houvesse sugestões de agentes ainda não lidas, esse ícone ficava amarelo e sorrindo. Quando todas as sugestões já tivessem sido lidas, o ícone ficava cinza e adormecido (PEZZIN, 2004).

Além disso, caso o usuário de ODE recebesse uma sugestão de um agente e a aceitasse, ou seja, clicasse no botão OK da janela de sugestões, uma mensagem era enviada ao agente, informando que o usuário aceitou sua sugestão (PEZZIN, 2004).

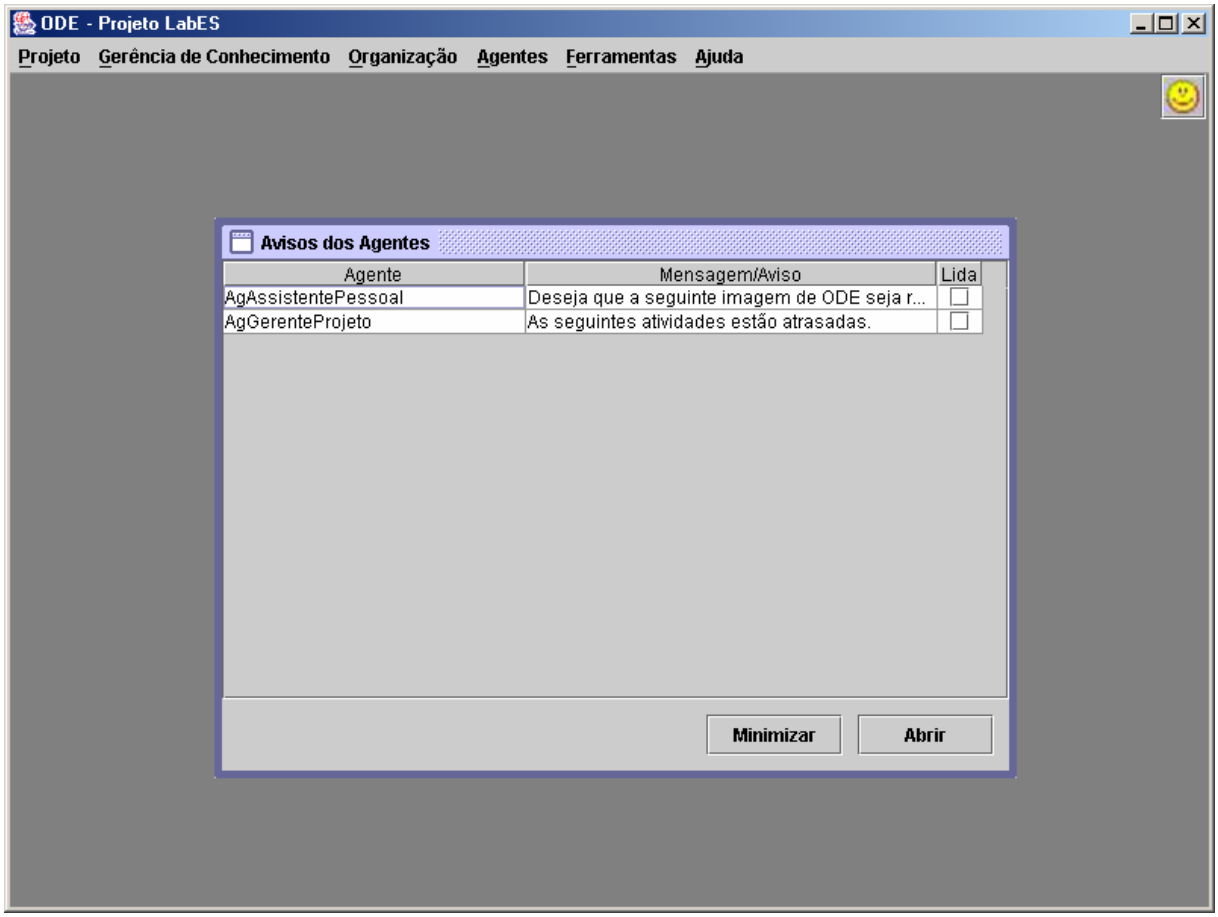

Figura 3.3 – Janela de Sugestões dos Agentes de ODE
Fonte: Pezzin (2004)

Um exemplo de publicação de mensagens na janela de sugestões dos agentes é exibido na Figura 3.4, utilizando-se a aplicação de recuperação da imagem, que antes existia em ODE. Após o usuário entrar em ODE, o Agente Assistente Pessoal recuperava a imagem do ambiente, isto é, o estado em que o ambiente estava na última vez em que o usuário acessou o ambiente (ferramentas e projeto abertos), e publicava um aviso na janela de sugestões, como ilustra a Figura 3.3. Quando o usuário pedisse para visualizar o aviso, a janela padrão de sugestões de agentes, mostrada com os dados do exemplo na Figura 3.4, era exibida.

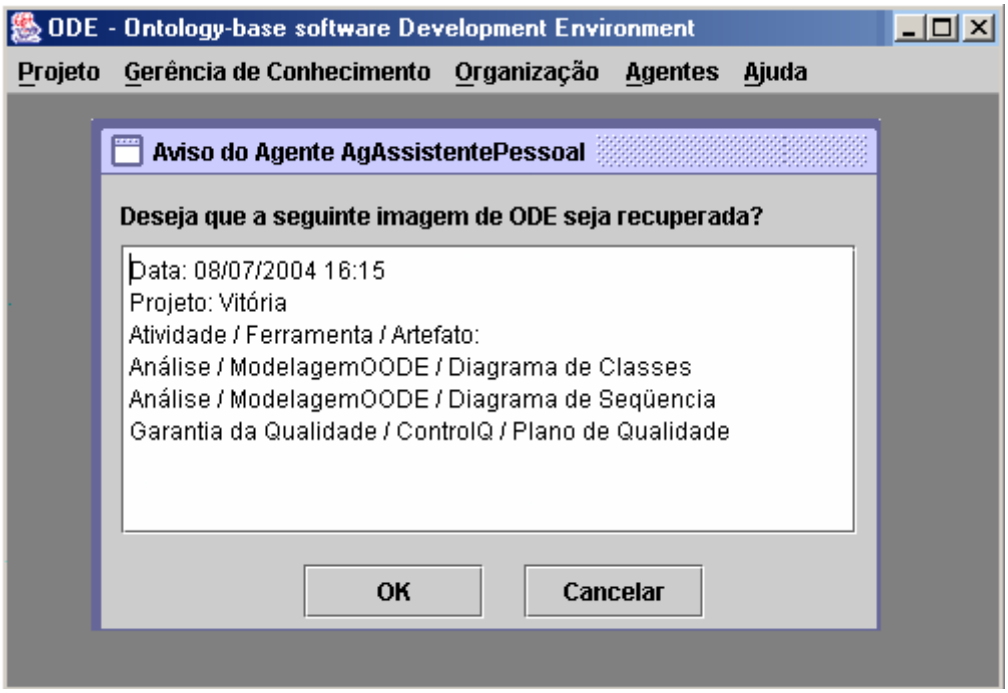

Figura 3.4 – Recuperando a imagem de ODE para o usuário
Fonte: Pezzin (2004)

3.2. Oportunidades de Melhoria Identificadas na Versão Original

Como mencionado no capítulo anterior, o ambiente ODE passou por uma reestruturação em sua arquitetura para torná-lo multi-usuário, tendo sido a sua camada de persistência alterada. Assim sendo, o objetivo principal deste trabalho é reintegrar AgeODE ao ambiente, além de evoluir essa infra-estrutura, de modo a torná-la mais flexível e poderosa. Para tal, a infra-estrutura original foi avaliada e diversos problemas e oportunidades de melhorias foram detectados, a saber:

(OM1) O desempenho da infra-estrutura estava visivelmente abaixo do desejável. Uma das causas desse problema poderia ser o fato de que os agentes, por serem implementados em processos diferentes, não podiam compartilhar recursos, como, por exemplo, a conexão com o banco de dados. Dessa forma, era possível encontrar várias conexões com o banco de dados ativas ao mesmo tempo, como já identificado em (PEZZIN, 2004). Todavia, foi identificado que a real causa do baixo desempenho da infra-estrutura era que os agentes realizavam *busy wait*, ou seja, os agentes competiam entre si e com o ambiente ODE pelo uso do processador do sistema computacional, resultando em baixo desempenho. Um problema secundário dessa abordagem era que os agentes não percebiam eventos externos (mudanças no ambiente ou atos

comunicativos de outros agente) imediatamente, mas só após verificarem a ocorrência deles, o que era realizado de tempos em tempos, utilizando-se um intervalo fixo de tempo.

(OM2) A versão original de AgeODE requeria que, caso fosse utilizado o apoio de agentes, o usuário de ODE iniciasse o processo do Agente Roteador, antes de iniciar o processo de ODE. Além disso, o processo de ODE requeria que o usuário indicasse, via linha de comando, se este deveria ser executado com ou sem o apoio de agentes.

(OM3) AgeODE utilizava *sockets* como meio de se implementar comunicações *peer-to-peer*, entretanto, esse é um mecanismo de baixo nível, que requer, entre outros, gerenciamento de endereços IP, de portas, de conexões etc. Além disso, utilizando-se *sockets*, um agente remetente só podia enviar mensagens a agentes que estivessem *online* e um agente receptor tinha que checar de tempos em tempos se havia novas mensagens.

(OM4) JAT*Lite* não oferecia suporte à análise de mensagens KQML e, portanto, o código referente à análise de mensagens era complexo e relembrava programação estruturada. Era necessário utilizar manipulação de *strings* de forma intensa para fazer a análise das mensagens recebidas. Além disso, existiam várias condições para verificar qual era a performativa da mensagem. O mesmo acontecia para o conteúdo dos outros parâmetros de uma mensagem KQML, além da verificação de condições e manipulação de *strings* para se analisar o conteúdo da mensagem.

(OM5) O conteúdo das mensagens dos agentes era pobre. A versão original de AgeODE utilizava XML como linguagem de conteúdo de mensagens, entretanto, o conteúdo era apenas o caminho completo de uma classe ou o identificador de um objeto no banco de dados. Não havia como os agentes se expressarem de forma mais rica em relação à classe ou ao objeto referenciado na mensagem, a não ser pela escolha da performativa da mensagem KQML.

(OM6) As ontologias nas mensagens KQML adotavam uma "postura diplomática". O campo referente à ontologia era apenas uma string, que simbolizava a ontologia a que ele se remetia, sem de fato referenciar uma ontologia implementada em alguma linguagem. Essa abordagem poderia ser um potencial limitador, à medida que serviços mais robustos fossem requisitados para os agentes de ODE, como, por exemplo, interagir com agentes externos a ODE, potencialmente na *web*.

(OM7) A definição de protocolo adotada por Pezzin (2004) e Schwambach (2004) diferia da definição de protocolo usualmente aceita na literatura. À grosso modo, Pezzin (2004)

e Schwambach (2004) tratam o conceito de protocolo, tanto em AgeODE quanto em OplA, como a especificação de uma interação entre agentes em termos de remetente, destinatário e semântica do conteúdo de uma mensagem. Entretanto, como discutido no capítulo anterior, um protocolo de interação é um padrão de comunicação, descrito como uma seqüência permitida de mensagens (atos comunicativos) trocadas entre agentes. Dessa forma, um protocolo de interação nada diz sobre quem é o remetente de uma mensagem, quem é o destinatário de uma mensagem, nem a qual é a semântica do conteúdo de uma mensagem.

(OM8) Uma classe de agente tinha várias responsabilidades, entre elas: tomar decisões referentes às regras de negócio, gerenciar uma conexão com o banco de dados, gerenciar o envio de mensagens, analisar mensagens recebidas etc. Apesar da separação dos agentes de interface de seus observadores ser uma forma de componentização, notou-se que isso não era o bastante. O desenvolvimento orientado a agentes deve se beneficiar de uma arquitetura em componentes, bem como faz o desenvolvimento orientado a objetos, como exemplificado com a arquitetura de ODE no capítulo anterior.

(OM9) À medida que novos meios de apresentação de aplicações se tornem mais atraentes, o pacote Componente de Gerência de Visão (cgv) do ambiente ODE, apresentado no capítulo anterior, poderá ser muito alterado. De fato, há uma discussão atual com respeito à criação ou mesmo migração de ferramentas de ODE para a *web*. Dessa forma, visando manutenibilidade, os agentes tem que ser independentes das classes do pacote cgv, ou, de forma a minimizar o problema, ter um baixo acoplamento com as mesmas. Isso contrasta com a idéia de que um agente de interface deve monitorar as ações do usuário interceptando eventos de interface.

(OM10) Ainda no que se refere aos observadores dos agentes de interface, identificou-se que, na versão original de AgeODE, eles interceptavam eventos assim que eles fossem ocorrer, ao invés de logo após eles terem ocorrido. Isso é um problema, visto que o usuário pode solicitar ao ambiente que seja executada alguma ação, porém o ambiente pode, por algum motivo (por exemplo, falta do preenchimento de campos obrigatórios, disparo de exceções etc.), negar que a ação seja executada.

(OM11) Como discutido na seção anterior, o fato de os observadores utilizarem reflexão computacional para instalar interceptadores nas classes fazia com que alterações nessas classes pudessem implicar alterações no código dos agentes. Olhando essa questão sob um prisma de manutenibilidade, não é desejável que mudanças simples,

como a renomeação de uma classe de ODE, implicasse manutenção do código de agentes.

(OM12) A definição purista de agentes adotada no ambiente, que indica que os agentes percebem o ambiente sem que esse tenha conhecimento disso, levou à decisão dos observadores dos agentes de interface serem implementados na versão original de AgeODE utilizando reflexão computacional. Entretanto, além de todos os problemas anteriormente citados decorrentes dessa abordagem, notou-se que o código dos observadores era relativamente complexo, o que dificultava a criação e a manutenção do mesmo.

(OM13) Agentes atuavam dentro do ambiente ODE sem que o próprio ambiente tivesse conhecimento disso e, portanto, não podiam interferir nas interfaces de ODE. Isso implicava que os agentes tinham que apresentar suas sugestões de forma totalmente desacoplada à ferramenta relacionada com a sugestão. Assim, existia uma janela padrão de sugestões de agentes, que exibia de uma forma genérica qualquer sugestão de agentes. Isso fazia com que as sugestões fossem apresentadas de uma maneira pobre, exclusivamente em formato textual, sem levar em conta se os dados poderiam ser melhor apresentados de uma maneira diferente, por exemplo, em formato tabular.

(OM14) Conforme citado no item anterior, as sugestões dos agentes eram simplesmente apresentadas. O ideal seria permitir que o usuário pudesse simplesmente dizer que aceita a sugestão do agente e o ambiente acatar a sugestão, ao invés de esperar que o usuário entrasse com os dados sugeridos pelo agente. Em outras palavras, o ambiente não "entendia" o que significavam as sugestões dos agentes.

(OM15) Agentes podiam fazer sugestões de baixa precisão. Por exemplo, um agente destinado a sugerir itens de conhecimento para reúso poderia sugerir itens de conhecimento em excesso, por não conseguir sugerir um subconjunto ótimo de itens a serem reutilizados. Uma das razões dessa imprecisão era o fato de que AgeODE ainda não tratava questões importantes, como capacidade de inferência e capacidade de aprendizado dos agentes, como apontado em (PEZZIN, 2004).

(OM16) O usuário não tinha informações que indicavam o raciocínio (*rationale*) que levou a uma dada sugestão. Isso pode fazer com que, após algumas sugestões não acatadas pelo usuário, ele passe a desacreditar no agente e a ignorar suas sugestões.

(OM17) As sugestões dos agentes não eram flexíveis, no sentido de levar em conta aspectos como a experiência do usuário na ferramenta e aceitação das sugestões por

parte do usuário. Por exemplo, nada impedia que agentes repetissem por completo sugestões parcialmente não acatadas pelo usuário.

3.3. A Nova Versão de AgeODE

Na busca por soluções para os problemas detectados, chegou-se à conclusão que JAT*Lite*, o *framework* para a construção de agentes no qual estava fundamentada a primeira versão de AgeODE, apresentava diversos problemas, sendo o mais grave o fato do projeto que desenvolvia o *framework* ter sido desativado e, por conseguinte, o mesmo estar defasado em relação a outros *frameworks*. Uma pesquisa foi feita, então, buscando uma alternativa de *framework*, chegando-se ao *framework* JADE (BELLIFEMINE *et al.*, 2007).

A subseção seguinte mostra que alguns dos problemas anteriormente detectados, a saber (OM1) a (OM7), são automaticamente solucionados à medida que se passa a utilizar JADE como *framework* base. As subseções subseqüentes apresentam outras propostas para se abordar alguns dos problemas e oportunidades de melhoria identificados.

3.3.1 – Um Novo *Framework* Base

A partir de uma análise do *framework* JADE e de sua fundamentação nas idéias de FIPA, ambos apresentados no capítulo anterior, constatou-se que diversos serviços poderiam ser muito aperfeiçoados, na medida em que JADE é um *framework* muito mais poderoso que JAT*Lite*. Além disso, ao se considerar que AgeODE passa a utilizar JADE como *framework* base, ao invés de JAT*Lite*, percebe-se que diversos dos problemas apresentados na seção anterior podem ser solucionados diretamente, a saber, as oportunidades de melhoria (OM1) a (OM7).

O problema (OM1), relacionado ao desempenho da infra-estrutura, é resolvido considerando-se que JADE implementa cada agente como uma *thread*. Assim, pode-se manter a *thread* de um agente "dormindo" e apenas "acordá-la" quando for necessário algum processamento, não consumindo recursos do sistema de computação desnecessariamente e, por conseguinte eliminando o problema de os agentes realizarem *busy wait*. Além disso, com a implementação baseada em *threads*, os agentes podem compartilhar recursos, como, por exemplo, a conexão com o banco de dados.

A resolução do problema (OM2), referente à inicialização da plataforma de execução dos agentes, é facilitada por JADE, visto que este permite que uma aplicação externa, neste

caso o Ambiente ODE, inicie a plataforma de execução de JADE e, a partir daí, inicie a execução de agentes. Além de facilitar a resolução desse problema, a arquitetura da plataforma JADE se mostra muito mais poderosa que a de JAT*Lite*. Uma plataforma JADE é composta de contêineres de agentes que podem estar distribuídos pela rede. Os agentes de JADE vivem em contêineres, sendo cada contêiner é um processo computacional que proporciona todos os serviços necessários para executar e gerenciar o ciclo de vida dos agentes. Existe um contêiner principal, junto ao qual os outros contêineres se registram, de modo que agentes de diferentes contêineres possam se comunicar entre si e até migrar de um contêiner para outro, viabilizando um novo padrão de agentes em AgeODE, o de agentes móveis. Isso abre espaço para que se criem aplicações em ODE de forma que agentes em diferentes instâncias de ODE (potencialmente operadas por usuários diferentes e em máquinas diferentes) possam se comunicar entre si. A arquitetura da plataforma JADE é exibida na Figura 3.5.

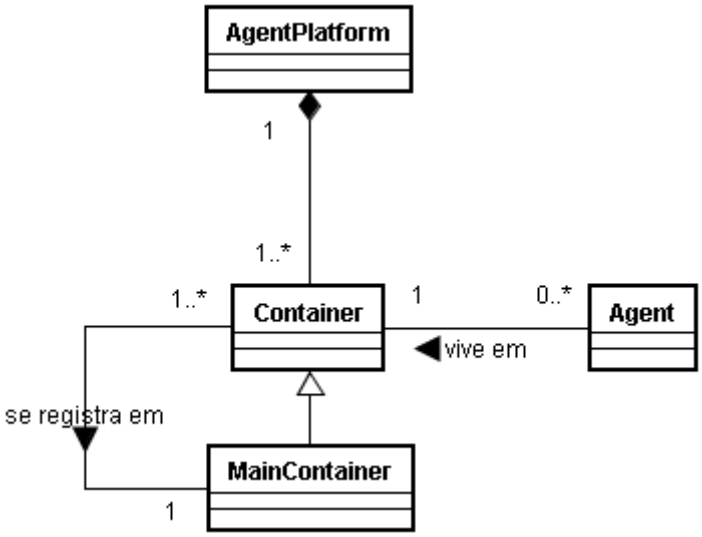

Figura 3.5 – Arquitetura da plataforma JADE
Fonte: Bellifemine *et al.* (2007)

O problema (OM3), uso de *sockets* como meio de se implementar comunicações *peer-to-peer*, é automaticamente resolvido ao se utilizar JADE, visto que o paradigma de comunicação desse *framework* é baseado em passagem assíncrona de mensagens. Mais detalhadamente, cada agente tem uma "caixa de mensagens" (implementada como uma pilha de mensagens), em que o ambiente de execução de JADE posta as mensagens enviadas por outros agentes. Assim que uma nova mensagem entra na "caixa de mensagens" de um agente, ele é notificado e, pode, se quiser e quando quiser, retirar a mensagem da "caixa de mensagens" para que ela seja lida. Porém, o potencial total do mecanismo de passagem de

mensagens em JADE é obtido quando se utiliza o apoio aos protocolos de interação de FIPA, visto que os comportamentos reutilizáveis de JADE se encarregam tanto de receber quanto de enviar as mensagens da interação, cabendo ao construtor do agente apenas sobrescrever métodos para processar as mensagens e métodos para criar as mensagens a serem enviadas. A Figura 3.6 exemplifica o paradigma de passagem assíncrona de mensagens, adotado em JADE.

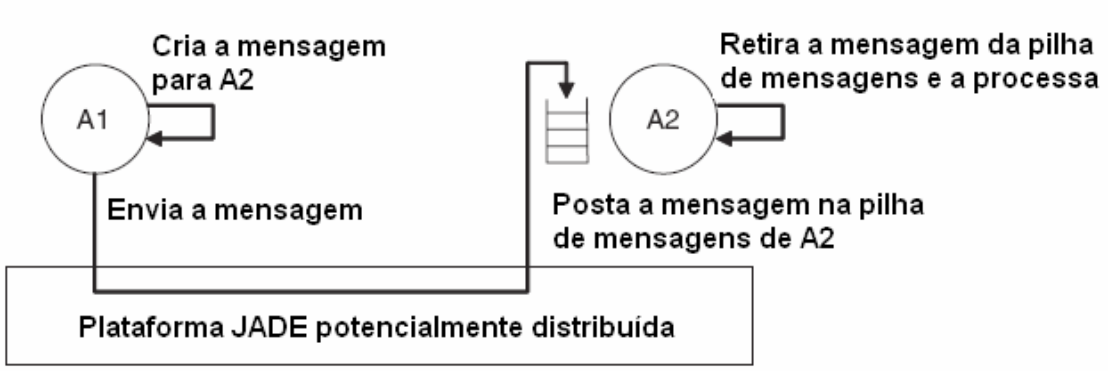

Figura 3.6 – Paradigma de passagem assíncrona de mensagens em JADE
Fonte: Bellifemine *et al.* (2007)

Ao se utilizar o apoio a ontologias e linguagens de conteúdo oferecido por JADE, pode-se minimizar os problemas (OM4), (OM5) e (OM6). De fato, o apoio oferecido pelo *framework* permite que se criem mensagens FIPA-SL a partir de objetos Java e, simetricamente, permite que se obtenha objetos Java a partir de mensagens FIPA-SL. Dessa forma, o problema (OM4), que diz respeito à intensa manipulação de *strings* e verificação de condições para se analisar mensagens recebidas, é minimizado.

Além disso, o uso da linguagem de conteúdo FIPA-SL em JADE resolve o problema (OM5), que identifica falta de expressividade no conteúdo das mensagens trocadas por agentes de AgeODE. Conforme apresentado no capítulo anterior, FIPA-SL é uma linguagem de conteúdo de caráter formal, que oferece ao construtor de agentes vários operadores, oferecendo suporte desde conectivos boleanos até lógica de primeira ordem e lógica modal.

A "postura diplomática" definida para as ontologias na versão original de AgeODE, apontada pelo problema (OM6), também pode ser repensada devido ao apoio a ontologias e linguagens de conteúdo de JADE. Esse *framework* minimiza a limitação antes existente, oferecendo um mecanismo para se implementar ontologias e utilizar a informação de uma mensagem ACL sobre qual é a ontologia em discurso de forma útil, assim como discutido no capítulo anterior. Postergamos uma análise mais detalhada sobre o modo como AgeODE utiliza o apoio de JADE a ontologias e linguagens de conteúdo para a sub-seção 3.3.3.

Outra funcionalidade de JADE que diretamente corrige um problema de AgeODE é o apoio que JADE oferece para se criar interações entre agentes em conformidade com os protocolos de interação especificados por FIPA. Essa funcionalidade, quando utilizada, corrige o conceito de protocolo de interação vigente na versão original de AgeODE, como identifica o problema (OM7). A abordagem de AgeODE em relação ao apoio a protocolos de interação oferecido por JADE é discutido a seguir.

3.3.2 – Arquitetura

Como apontado no capítulo anterior, em JADE há uma separação entre os agentes e seus comportamentos, que devem ser subclasses de *Agent* e de *Behaviour*, respectivamente. Além disso, JADE oferece um mecanismo semelhante à composição, por meio do qual comportamentos podem ser adicionados a agentes a qualquer momento, utilizando-se o método *addBehaviour* da classe *Agent*, e o próprio *framework* se encarrega de executar e escalonar os comportamentos de um agente.

Dessa forma, concluímos que, ao passo que a versão original de AgeODE trabalhava com herança como mecanismo de reutilização, a nova versão de AgeODE pode trabalhar com um mecanismo mais poderoso, a composição (GAMMA *et al.*, 1995).

Assim, torna-se necessário repensar a abordagem de utilização dos padrões de agentes identificados na versão original de AgeODE. Vale lembrar que, na versão original de AgeODE, o desenvolvedor de um novo agente tinha que definir o tipo principal e os possíveis tipos secundários de um agente e, a partir disso, fazer com o que o novo agente herdasse a classe do tipo de agente principal e implementasse as interfaces dos tipos de agente secundários.

Este trabalho propõe que, na nova versão de AgeODE, exista apenas um agente genérico, chamado de *AgOde*, de forma a servir como fachada para *Agent*, fornecido por JADE. Assim, o desenvolvedor deve identificar, para cada novo agente a ser construído, quais são os seus tipos (sem distinção entre tipo principal e tipos secundários) e, a partir daí, saber um conjunto mínimo de tipos de comportamentos que esse agente deve ter. Desse modo, torna-se necessário que, primeiro, se identifiquem quais tipos de comportamentos agentes de AgeODE podem executar para que, então, possam se definir quais tipos de comportamentos são relacionados a quais tipos de agentes.

Para definir tipos de comportamentos, vale lembrar a definição de agente, que diz que um agente é um ser dotado de autonomia e que tipicamente sente seu ambiente por meio de

sensores e atua sobre ele, de forma a modificá-lo para atingir seus objetivos. Mais ainda, um agente é um ser social, que tem capacidades de se comunicar com outros agentes, visando a viabilizar o cumprimento de seus objetivos (WOOLDRIDGE, 1999).

Dessa forma, pode-se perceber que agentes devem ter comportamentos para: (i) observar o ambiente (sensores); (ii) atuar sobre o ambiente (atuadores); e (iii) se comunicar com outros agentes. Verifica-se, também, que comportamentos relacionados à comunicação têm duas preocupações distintas: (i) gerenciar interações inteiras, ou seja, lidar com protocolos de interação; e (ii) gerenciar um passo específico da interação, ou seja, lidar com análise de mensagens, incluindo a manipulação de ontologias e linguagens de conteúdo. Enfim, pode-se extrair a lógica que existe por trás de cada um desses tipos de comportamentos, que consiste de planos que os agentes executam visando a atingir seus objetivos.

Tendo em mente a separação entre agentes e comportamentos e a definição de tipos de comportamentos, é possível esboçar uma arquitetura de agentes, de forma a tratar a oportunidade de melhoria (OM8), que identifica que o desenvolvimento orientado a agentes pode se beneficiar de uma arquitetura em componentes, como faz o desenvolvimento orientado a objetos.

Assim sendo, a Figura 3.7 mostra a arquitetura dos agentes da nova versão AgeODE. Agentes de AgeODE têm um pacote onde fica sua classe principal, ou seja, a classe que herda de *Agent*. Além disso, existe um pacote *Comportamento*, que pode ser visto como um repositório de comportamentos reutilizáveis. Esses comportamentos são divididos em pacotes, de forma a separar comportamentos que visam a: observar o ambiente (*Observador*), atuar sobre o ambiente (*Atuador*), formular os planos do agente (*Lógica*) e se comunicar com outros agentes (*Comunicação*). Além disso, os comportamentos relacionados à comunicação são divididos em dois pacotes, a saber: *Controle de Interação*, responsável por gerenciar protocolos de interação, e *Manipulação de Mensagens*, responsável por gerenciar cada passo de uma interação, ou seja, criar e analisar mensagens, fazendo uso do apoio que JADE oferece à linguagem ACL e a ontologias e linguagens de conteúdo.

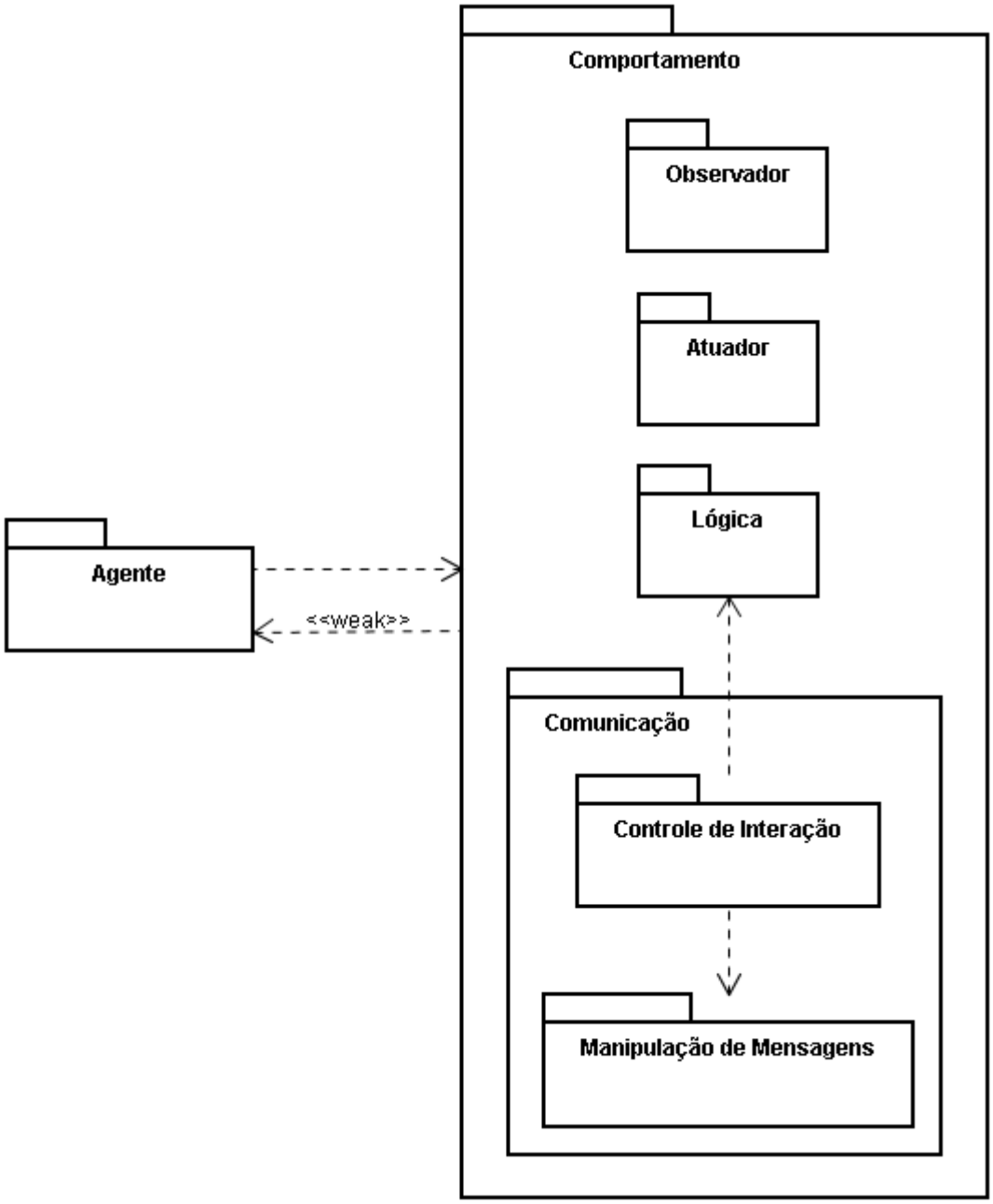

Figura 3.7 – Nova Arquitetura dos Agentes de AgeODE

Dessa forma, define-se que agentes de informação devem ter comportamentos do pacote *Lógica*. Agentes de interface, por observarem o ambiente, devem ter comportamentos do pacote *Observador*. Agentes coordenadores, que pela sua definição coordenam outros agentes, devem ter comportamentos do pacote *Comunicação*. Por fim, agentes de usuário devem ter comportamentos do pacote *Atuador*, pois visam a ajudar os usuários na realização de alguma tarefa.

Contudo, deve ser salientado que os agentes não são restritos a ter comportamentos apenas de um dos tipos citados acima. Por exemplo, qualquer agente que esteja embutido em

uma sociedade e se comunique com outros agentes também terá comportamentos do pacote *Comunicação*. Além disso, é uma boa prática que as decisões tomadas no contexto de uma interação, principalmente quando são complexas, sejam isoladas em classes do pacote *Lógica*.

Também é possível que um agente seja do tipo interface e do tipo usuário ao mesmo tempo. Esse seria, por exemplo, o caso de um agente que atua sozinho, observando o ambiente para acompanhar as ações do usuário e, em momento propício, atua no ambiente, fazendo sugestões para que usuário realize suas tarefas da melhor maneira possível.

Assim, pode-se definir que todo agente, ou pelo menos aqueles que tenham uma estrutura complexa, tem seus planos de alto nível encapsulados em um comportamento do pacote *Lógica*, que pode agir como controlador, gerenciando a execução de outros comportamentos de mais baixo nível dos pacotes *Lógica*, *Comunicação*, *Atuação* e *Observador*. Esse comportamento do pacote *Lógica* é, portanto, um comportamento que utiliza composição para orquestrar a execução de outros comportamentos mais simples.

Visando reutilização, JADE fornece o comportamento abstrato *CompositeBehaviour*, que tem como subclasse concreta o comportamento *ParallelBehaviour*, destinada a comportamentos compostos que executam seus comportamentos agregados em paralelo; e tem como subclasse abstrata o comportamento abstrato *SerialBehaviour*, em que o comportamento composto executa seus comportamentos agregados de forma serial. *SerialBehaviour*, por sua vez, tem como subclasses concretas *SequentialBehaviour*, de forma que o comportamento composto executa seus comportamentos agregados utilizando uma política simples de execução seqüencial; e *FSMBehaviour*, que implementa uma máquina de estados, de modo que o desenvolvedor a define registrando estados e transições, servindo como mecanismo poderoso para orquestrar a execução de comportamentos agregados.

3.3.3 – Protocolos de Interação

Os componentes de um agente de AgeODE que gerenciam protocolos de interação são os comportamentos do pacote *Controle de Interação*. Vale notar que classes desse pacote delegam a criação e a análise de mensagens para classes do pacote *Manipulação de Mensagens* e delegam as decisões tomadas no decorrer da interação para classes do pacote *Lógica*.

Assim, deve ser ressaltado que as classes do pacote *Controle de Interação* herdam da classe *Behaviour*, porém classes do pacote *Manipulação de Mensagens* não herdam dessa classe, pois têm seus métodos invocados apenas por classes controladoras de interação. Isso

pode acontecer também com comportamentos do pacote *Lógica*, entretanto, não obrigatoriamente, como exemplificado na subseção anterior com comportamentos do pacote *Lógica* que agem como orquestradores dos planos de execução do agente. Como diretriz, se um comportamento tem seus métodos invocados pelo ambiente de execução JADE, ele deve herdar de *Behaviour*; caso contrário, não deve herdar de *Behaviour*.

Como discutido no capítulo anterior, um dos pontos fortes de JADE é que o *framework* fornece comportamentos reutilizáveis que implementam os protocolos de interação de FIPA. Assim, tudo o que o programador tem que fazer é redefinir alguns métodos desses comportamentos, inserindo lógica específica de domínio. Essa abordagem vem a corrigir o problema (OM7), que indicava que, na versão original de AgeODE, a definição de protocolo de interação adotada era diferente da aceita por FIPA.

Em vista disso, definiu-se que todas as interações realizadas por dois ou mais agentes construídos no contexto de AgeODE devem seguir os protocolos de interação de FIPA e, portanto, devem ter comportamentos que herdam das classes reutilizáveis de JADE que implementam esses protocolos. Dessa forma, os comportamentos do pacote *Controle de Interação* herdam sempre indiretamente de *Behaviour*, mas, em contraposto aos comportamentos do pacote *Lógica*, herdam de comportamentos que já têm uma estrutura definida (por implementarem os protocolos de interação).

Um padrão de nomenclatura foi adotado, em que os comportamentos desse pacote devem ter seu nome iniciado por *Comp* e finalizado por *Initiator*, caso seja um comportamento que inicia uma interação, ou por *Responder*, caso seja um comportamento que responde a um estímulo de interação. Para exemplificar, suponha que um agente *AgIdentificadorRiscos* inicie uma interação com o agente *AgIdentificadorProjetosSimilares*, seguindo o protocolo de interação *fipa-request*, de forma a requisitar projetos similares. Assim, um nome possível para o comportamento do agente *AgIdentificadorRiscos* é *CompIndentificarProjetosSimilaresInitiator*, enquanto um nome possível para o comportamento do agente *AgIdentificadorProjetosSimilares* é *CompIndentificarProjetosSimilaresResponder*.

Esse tipo de mecanismo pode ser muito poderoso, na medida em que se podem criar bibliotecas de comportamentos reutilizáveis, de forma que várias classes diferentes de agentes possam executar um mesmo tipo de comportamento. Retornando ao exemplo anterior, suponha que exista um terceiro agente, *AgAvaliadorRiscos*, que também inicia uma interação com o agente *AgIdentificadorProjetosSimilares* de modo a requisitar projetos similares. Assim, pode-se promover o reúso de comportamentos se tanto *AgIdentificadorRiscos* quanto

AgAvaliadorRiscos utilizarem a classe de comportamento *CompIndentificarProjetosSimilaresInitiator*.

No entanto, como apontado em (BELLIFEMINE *et al.*, 2007), é considerada uma boa prática que os comportamentos dos agentes sejam implementadas como classes internas da classe do agente que irá executá-lo. Essa prática contrasta com a idéia de comportamentos reutilizáveis, chamados em (BELLIFEMINE *et al.*, 2007) de comportamentos de propósito geral, e, portanto, não foi adotada em AgeODE. Dessa forma, propõe-se que classes de comportamentos de agentes não sejam classes internas.

Contudo, a questão da reutilização de classes de comportamentos não é tão simples assim. A classe *Behaviour* tem uma referência à classe *Agent*, que serve, entre outros, como meio para que o comportamento possa informar o resultado de suas ações ao agente que o está executando. Todavia, a classe *Agent* não contém métodos específicos do domínio do agente e do comportamento em questão e, portanto, como prática adotada em exemplos que vêm junto com *framework* JADE, o comportamento faz um *casting* na referência do agente, de modo que a referência seja de uma subclasse de *Agent* que tenha métodos específicos de domínio. Entretanto, essa solução não nos atende, visto que, ao efetuar um *casting*, o comportamento estaria preso a um determinado tipo de agente, não podendo ser reutilizado por várias classes de agentes.

Além disso, um comportamento também pode ser iniciado por um outro comportamento que já se encontra em execução no contexto de um agente. Esse é o caso de protocolos aninhados, em que, para que um agente formule uma resposta a uma pergunta, ele deve iniciar um outro protocolo de interação com outros agentes. Dessa forma, pode-se perceber que devemos criar uma comunicação de baixo acoplamento entre um comportamento e a entidade à qual ele informa os resultados obtidos, e que essa entidade pode ser qualquer classe de agente ou qualquer classe de comportamento. Assim, preferimos o termo “comportamentos reutilizáveis” ao termo “comportamentos independentes de agente”. Essa dependência de baixo acoplamento que pode existir entre o comportamento e o agente em que ele se contextualiza é representada na Figura 3.7 utilizando-se o estereótipo <<*weak*>>.

Pensando em uma solução para que se possam construir comportamentos de agentes efetivamente reutilizáveis, chegou-se a um padrão em que esses comportamentos que necessitam informar o resultado de suas ações à entidade que os iniciaram devem ter associados a si uma interface, que deve ser implementada pelas classes que obterão os resultados obtidos pelo comportamento. Dessa forma, o construtor do comportamento, além

de receber como parâmetro um agente do tipo *Agent*, recebe uma classe (de agente ou de comportamento) que implementa a interface com que o comportamento está associado e que receberá os resultados obtidos pelo comportamento.

A Figura 3.8 ilustra a instanciação desse padrão para o exemplo anterior, onde os agentes *AgIdentificadorRiscos* e *AgAvaliadorRiscos* implementam a interface *IndentificarProjetosSimilaresInitiator* e reutilizam a classe de comportamento *CompIndentificarProjetosSimilaresInitiator*, que os informará dos resultados via chamada ao método *dispararProjetosSimilaresIdentificados*.

Como se pode notar na Figura 3.8, utiliza-se o estereótipo <<*behaviour*>> para indicar uma relação de execução de comportamento entre um agente e um comportamento. Além disso, como define OplA (SCHWAMBACH, 2004), as classes de agentes são representadas como classes ativas da UML 2, também chamadas de objetos ativos na UML 1.5. Além da diferença na nomenclatura entre as diferentes versões da UML, também existe uma diferença na sintaxe, onde uma classe ativa na UML 2 tem linhas verticais adicionais nas laterais, enquanto na UML 1.5 tem a borda mais grossa (FOWLER, 2003). Deve-se realçar que a ferramenta CASE utilizada neste trabalho suporta apenas a sintaxe de classes ativas da versão 1.5 da UML.

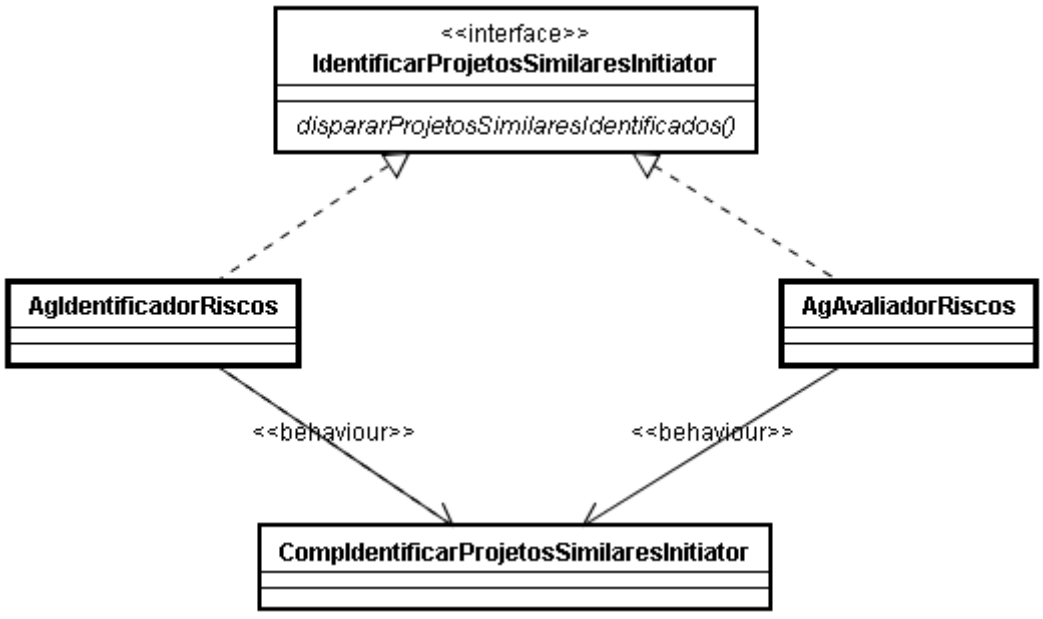

Figura 3.8 – Aplicação do padrão para se criar Comportamentos reutilizáveis

Vale ressaltar que JADE fornece um mecanismo alternativo para que se possam criar comportamentos reutilizáveis. Cada comportamento tem disponível um *DataStore*, que pode ser compartilhado pelas classes dos diferentes comportamentos no contexto de um agente,

além da classe do próprio agente. *DataStore* é um mapa, ou seja, armazena pares [chave,valor] e age como se fosse um repositório de variáveis globais, que pode ser compartilhado pelos vários comportamentos de um agente, além do próprio agente. Entretanto, entendemos que o uso do *DataStore* quebra o encapsulamento, um dos principais avanços do paradigma orientado a objetos em relação ao paradigma estruturado e, portanto, não deve ser utilizado.

3.3.4 – Apoio a Ontologias e a Linguagens de Conteúdo

O apoio a ontologias e a linguagens de conteúdo oferecido por JADE permite proporcionar grandes avanços em AgeODE, tratando as oportunidades de melhoria (OM6) e (OM5), respectivamente, que eram grandes limitações da versão original.

De acordo com a nova arquitetura de agentes de AgeODE, são os comportamentos do pacote *Manipulação de Mensagens* que incorporam os recursos que JADE oferece para a manipulação de ontologias e linguagens de conteúdo. Conforme discutido no capítulo anterior, esse apoio visa a permitir aos desenvolvedores de agentes manipular toda a informação no contexto de um agente utilizando objetos Java, sem que seja necessário convertê-los em *strings* a serem embutidas em alguma linguagem de conteúdo, nem que seja necessário fazer a operação inversa, ou seja, receber uma mensagem e obter objetos Java manualmente a partir de seu conteúdo.

Além disso, JADE faz o que Bellifemine *et al.* (2007) chamam de verificação semântica, ou seja, JADE associa os termos de uma mensagem com os conceitos de uma ontologia compartilhada entre os agentes no contexto da interação, de forma que sejam conferidas todas as restrições impostas na ontologia. Vale ressaltar que restrições mais avançadas podem ser descritas nas ontologias por meio de *Facets* (CAIRE *et al.*, 2004).

Para viabilizar essa verificação semântica, é necessário que antes se defina uma ontologia, o que é feito por meio da criação de uma subclasse da classe *Ontology*. A partir daí, deve-se declarar quais são os conceitos, predicados e ações de agentes presentes na ontologia.

Todavia, em contraposto à definição de que uma ontologia de domínio descreve o vocabulário de um domínio genérico e pode ser instanciada por uma classe de aplicações (GUARINO, 1998), JADE impõe que todos os símbolos utilizados em interações entre agentes, específicos da aplicação em questão ou não, sejam definidos nas ontologias, de modo a viabilizar um mapeamento direto entre os símbolos da ontologia e as classes da aplicação.

Um exemplo disso são os predicados e as ações de agentes, que são dependentes de aplicação e, portanto, não estariam presentes em uma ontologia de domínio.

Além disso, quando se analisa a dimensão temporal da utilização de ontologias em sistemas de informação (GUARINO, 1998), percebe-se que as ontologias nas quais ODE se baseia são utilizadas em tempo de desenvolvimento, ao passo que as ontologias de JADE são utilizadas em tempo de execução. Contudo, não se pode confundir a implementação (em uma linguagem propícia para implementação de ontologias) das ontologias de domínio nas quais ODE se baseia, como é feito na Infra-estrutura Semântica de ODE (RUY, 2006) (PIANISSOLLA, 2007); com a implementação (em Java, feito por meio de herança da classe *Ontology*) das ontologias que JADE requer para que se possa automatizar a construção e a análise de mensagens de agentes. Ou seja, os estudos para se utilizar ontologias em ODE em tempo de execução, da forma como definido em (GUARINO, 1998), estão no contexto da Infra-estrutura Semântica de ODE, ao passo que as ontologias implementadas neste trabalho visam apenas a suprir essa necessidade do *framework* JADE.

Para que se perceba essa diferença de forma mais explícita, basta mostrar que, ao se desenvolver uma ferramenta de ODE, as classes do diagrama de classes da fase de análise instanciam os conceitos das ontologias, inserindo detalhes dependentes de aplicação. Em um segundo momento, esse diagrama de classes incorpora detalhes específicos da tecnologia de implementação, portanto, gerando um diagrama de classes de projeto. Por fim, uma ontologia de JADE incorpora detalhes de comunicação entre agentes, como predicados e ações de agentes, derivados de um diagrama de classes de projeto. Essa relação entre uma ontologia de JADE e modelos de projeto é necessária, visto que JADE necessita fazer um mapeamento direto entre suas ontologias e as classes da aplicação, de modo que seja viável fazer a conversão entre objetos Java e símbolos contidos em mensagens trocadas por agentes.

Outra diferença entre ontologias de domínio e ontologias de JADE é que ontologias de JADE não podem ter relacionamentos com navegabilidade dupla. Ao se converter um objeto Java para embuti-lo em uma mensagem SL, também se devem converter todos os seus atributos que estão presentes na ontologia de JADE. Dessa forma, sejam duas classes A e B, que têm um relacionamento de navegabilidade dupla. Então, para que JADE insira A em uma mensagem SL, deve converter A e também seu atributo B em uma *string*, na sintaxe de SL. Porém, para converter B em uma *string* na sintaxe de SL, JADE deve converter também seu atributo A em uma *string*, e assim por diante. Ou seja, relacionamentos com navegabilidade dupla em ontologias de JADE levam o *framework* a um laço infinito. O mesmo pode ser dito

para alguns conjuntos de associações que formem um ciclo, de forma que o *framework* tente converter os objetos, mas acabe entrando em laço infinito.

Com base nas diferenças entre ontologias de domínio e ontologias-JADE, fez-se, então, uma distinção também na nomenclatura, de modo que as ontologias no contexto de JADE sejam chamadas de ontologias-JADE. Essa nomenclatura é usada daqui em diante neste trabalho.

Outro ponto a ser discutido sobre a abordagem de JADE para tratar ontologias é que é defendido como boa prática que só sejam incluídos em uma ontologia-JADE conceitos, predicados e ações de agentes que fazem parte do vocabulário contido em mensagens trocadas por agentes (NIKRAZ *et al.*, 2006). Entretanto, ontologias-JADE definidas no contexto de AgeODE não devem seguir essas diretrizes.

Como citado anteriormente, ontologias-JADE derivam de diagramas de classes de projeto de ODE, que, por sua vez, derivam de diagramas de classes de análise, que derivam de ontologias. Dessa forma, pelo menos indiretamente, uma ontologia-JADE deriva de uma ontologia de domínio em ODE. Assim, propõe-se que uma ontologia-JADE contenha todos os conceitos da ontologia de domínio, porém, descritos da forma como eles são tratados no diagrama de classes de projeto. Além disso, uma ontologia-JADE deve, sempre que possível, eliminar os conceitos e atributos dependentes de aplicação (inseridos no diagrama de classes de análise) e de tecnologia (inseridos no digrama de classes de projeto). Assim, um conceito de uma ontologia-JADE é potencialmente mais simples que a sua classe correspondente, visto que a última contém detalhes de aplicação e de tecnologia não necessariamente presentes na ontologia-JADE.

Outro ponto a ser destacado é que as ontologias-JADE definidas no contexto de AgeODE têm potencialmente mais conceitos do que as ontologias-JADE construídas utilizando-se as práticas apresentadas em (NIKRAZ *et al.*, 2006) e, assim, se tornam mais fáceis de serem mantidas, pois, à medida que o sistema cresce, os agentes potencialmente passam a ter necessidade de discursar sobre outros conceitos do domínio sobre os quais antes não discursavam.

Além de conceitos, AgeODE define que uma ontologia-JADE pode conter, também, predicados e ações de agentes específicos da aplicação em questão. Neste trabalho apresenta-se uma ontologia-JADE e se discute como ela foi definida a partir de uma ontologia de domínio e de um diagrama de classes de projeto no Capítulo 4, no contexto de uma ferramenta de apoio à Gerência de Riscos em ODE.

Vale lembrar que a infra-estrutura AgeODE é definida no contexto de ODE e, portanto, deve levar em conta suas peculiaridades. Ainda que seja corriqueiro se conceituar ODE como um ADS centrado em processo e baseado em ontologias, pode-se também dizer que ODE é um sistema de informação multi-usuário com um banco de dados central. Dessa forma, deve-se levar em conta que agentes que atuam no contexto ODE, apesar de potencialmente formarem um sistema multiagente distribuído, atuam sobre um sistema que tem seus dados centralizados em um servidor de banco de dados.

Entretanto, pode-se notar que o apoio de JADE a ontologias e linguagens de conteúdo foi projetado de forma a ser adequado para que agentes de diferentes sistemas se comuniquem entre si, ou mesmo para integrar diversos módulos distribuídos de um sistema, cada qual com sua base de dados. Muito também se fala de aplicações da tecnologia de agentes na *Web* Semântica (BERNERS-LEE *et al.*, 2001). De fato, esse é realmente o objetivo de FIPA: promover a interoperabilidade. Dessa forma, o apoio a ontologias e linguagens de conteúdo de JADE transforma objetos manipulados por um agente A1, em uma mensagem SL (utilizando o mapeamento da ontologia-JADE para as classes de aplicação de A1) e depois converte essa mensagem em objetos Java que serão manipulados por um agente A2 (utilizando o mapeamento da ontologia-JADE para as classes de aplicação de A2). Assim, além do fato de que os dois mapeamentos utilizados serem potencialmente diferentes (pelo fato de que as classes da aplicação de A1 serem potencialmente diferentes das classes da aplicação de A2), cada uma dessas duas aplicações tem sua própria base de dados.

Claramente, não é isso o que acontece em ODE. Agentes de AgeODE compartilham um mesmo mapeamento entre ontologias-JADE e classes de aplicação, além de terem um banco de dados único. Assim, mecanismos mais eficientes podem ser utilizados no contexto de AgeODE.

Como mencionado anteriormente, agentes não discursam apenas sobre os conceitos do domínio em questão, mas também sobre alguns conceitos dependentes de aplicação, como pode ser o caso do identificador de objetos no banco de dados. Como os agentes de ODE interagem discursando sobre os mesmos objetos (pois estão disponíveis em um banco de dados compartilhado), eles devem ter esse poder de expressividade no conteúdo de suas mensagens e, portanto, entende-se que deve-se inserir o conceito de objeto persistente nas ontologias-JADE construídas no contexto de AgeODE. *ObjetoPersistente* é uma classe de ODE que tem um atributo *id*, do tipo *string*, que visa justamente a identificar objetos do ambiente no banco de dados. Dessa forma, todas as classes do domínio do problema de ODE (classes do pacote cdp) devem herdar de *ObjetoPersistente*. Assim sendo, todas as ontologias-

JADE construídas no contexto de AgeOde devem incluir o conceito *ObjetoPersistente*, definido na ontologia-JADE de Persistência, apresentada na Figura 3.9.

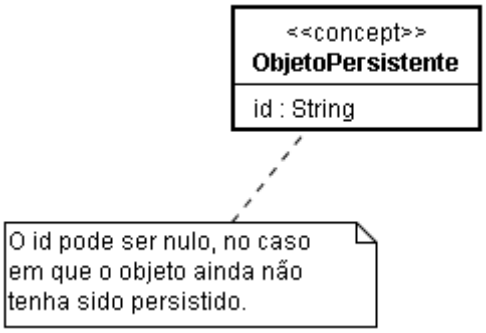

Figura 3.9 – Ontologia-JADE de Persistência

Um outro item em destaque no que diz respeito ao apoio a ontologias e linguagens de conteúdo oferecido por JADE é que JADE indica que, após a criação das ontologias-JADE, deve-se dar início à construção das classes de aplicação que se referem aos elementos da ontologia-JADE, ou seja, conceitos, predicados e ações de agente. Além disso, essas classes devem implementar interfaces oferecidas por JADE, segundo o tipo de elemento que elas representam na ontologia. Assim, sistemas desenvolvidos dessa maneira são centrados em agentes e se supõe que o sistema não existe sem os agentes. Todavia, ODE é mais uma vez peculiar, no sentido de que é um sistema que já existe (portanto, as classes que vão ser mapeadas nas ontologias já existem) e que deve ser apto a executar sem agentes.

Além disso, JADE impõe regras estruturais para classes que representam um elemento de uma ontologia-JADE. Uma delas é que essas classes devem ter métodos *get* e *set* para cada atributo, o que é um problema, pois, em alguns casos, classes de ODE contêm métodos *obter* e *atribuir* ao invés de *get* e *set*. Outra regra que não se aplica a ODE é que atributos cuja relação na ontologia-JADE tem cardinalidade maior que 1 devem ser do tipo `jade.util.leap.List`, uma lista de JADE correspondente à lista padrão de Java `java.util.List`, porém com aspectos que a fazem ser eficiente em dispositivos móveis.

Em outras palavras, as classes de domínio do problema de ODE não aderem, nem podem aderir, às regras impostas por JADE para classes que representam elementos de uma ontologia-JADE.

Mais ainda, considera-se que não é uma boa prática poluir classes de domínio do problema com detalhes relativos a um *framework*. Isso leva a diminuir a capacidade de reutilização das classes de domínio do problema (que podem ser consideradas um *framework*

vertical), pois faz com que qualquer alteração no mecanismo de manipulação de ontologias-JADE que possa vir a ser introduzida em uma versão futura implique em necessidade de manutenção nas classes de domínio do problema.

Assim sendo, sugere-se a aplicação do padrão de projeto Adaptador (GAMMA *et al.*, 1995), de forma que se criem adaptadores que adaptam as classes de domínio do problema de ODE, visando a satisfazer os requisitos de JADE. Assim, os elementos das ontologias-JADE terão mapeamento para os adaptadores, e não para as classes de domínio do problema. Dessa forma, pode-se fazer com que os adaptadores cumpram todas as regras ditadas pelo *framework*, de modo que as classes de domínio do problema não necessitem ser alteradas. Neste trabalho esses adaptadores são denominados adaptadores-JADE.

Seja o diagrama de classes de projeto da Figura 3.10 e a ontologia-JADE da Figura 3.11, gerada a partir do diagrama de classes. Note que uma ontologia-JADE, as associações têm sempre navegabilidade simples e multiplicidades explícitas apenas na extremidade navegável. A Figura 3.12 exemplifica como foi aplicado o padrão de adaptadores-JADE, apresentando um adaptador-JADE para o conceito *KRisco* da ontologia-JADE da Figura 3.11. Vale notar que a classe *Conhecimento* não entra na ontologia-JADE e, portanto, não precisa de um adaptador-JADE.

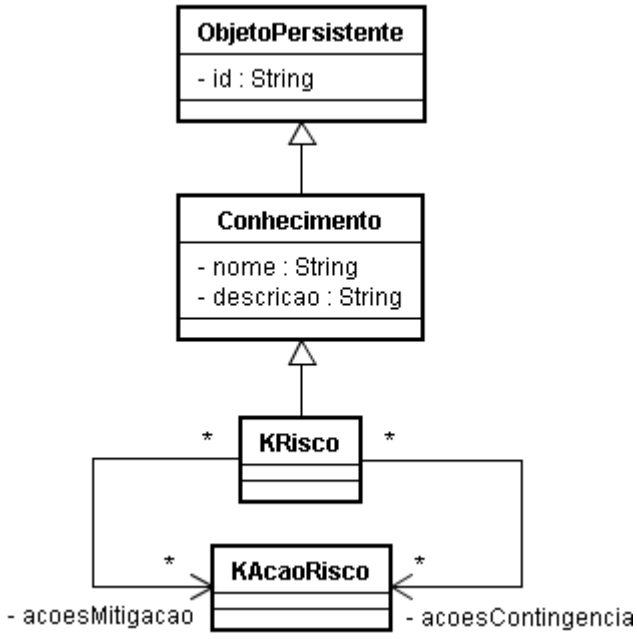

Figura 3.10 – Exemplo de diagrama de classes de Projeto

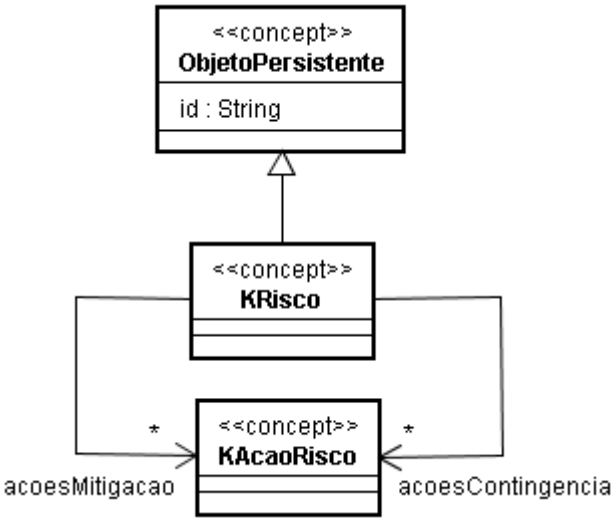

Figura 3.11 – Exemplo de ontologia-JADE

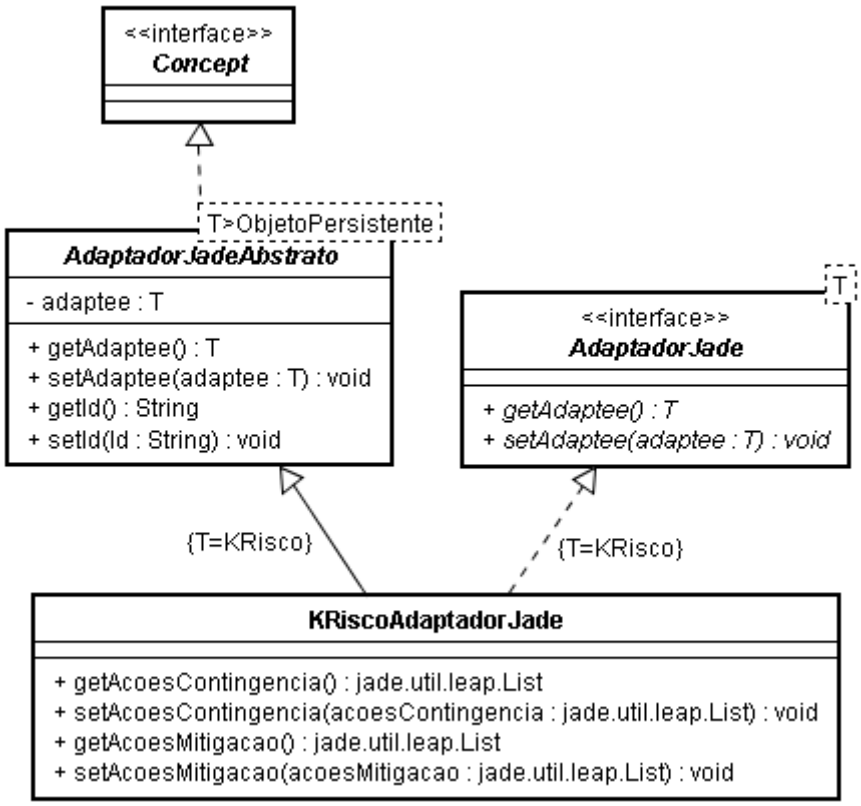

Figura 3.12 – Exemplo da criação de um adaptador-JADE para o conceito KRisco

Um adaptador-JADE deve implementar a interface parametrizada *AdaptadorJade*, que requer métodos de acesso ao objeto adaptado. Dessa forma, o *framework* JADE, após verificar o mapeamento entre o conceito de uma ontologia e um adaptador-JADE, invoca os métodos *setters* do adaptador-JADE, de modo a preencher seus atributos (que devem espelhar

os atributos do conceito na ontologia-JADE). Após essa inicialização, um adaptador-JADE deve estar apto a retornar o objeto real do domínio do problema, ou seja, seu *adaptee*.

Além disso, por clareza, omitimos na Figura 3.12 outros métodos de *AdaptadorJade*, que incluem métodos de conversão entre uma lista Java e uma lista JADE (ambas contendo elementos do tipo adaptado), e vice-versa, além de métodos de conversão entre um conjunto Java e uma lista JADE, e vice-versa. Esses quatro métodos de conversão são necessários porque, como já dito, ao se converter um objeto Java para embuti-lo em uma mensagem SL, também se devem converter todos os seus atributos que estão presentes na ontologia de JADE. Assim, um adaptador de um conceito que tenha um relacionamento de cardinalidade maior que 1 na ontologia-JADE deve pedir ao adaptador do outro conceito participante do relacionamento para que converta a lista (ou conjunto) de objetos adaptados. O inverso acontece ao se criar objetos a partir de uma mensagem SL, ou seja, devem ser criados objetos Java a partir de listas de JADE contendo adaptadores-JADE, que por fim encapsulam objetos do domínio de ODE.

De forma a promover o reúso, a classe abstrata parametrizada *AdaptadorJadeAbstrato* foi criada, de modo que todos os adaptadores de conceitos que herdam de *ObjetoPersistente* devem ser suas subclasses. Portanto, *AdaptadorJadeAbstrato* implementa o atributo *id* (identificador do objeto no banco de dados) e também implementa, parcialmente, os métodos conversores que são úteis na conversão de mensagens SL em objetos, deixando a cargo de suas subclasses apenas um método responsável por recuperar os objetos do banco de dados, a partir do *id*. Essa limitação se deve ao fato de que esse adaptador genérico não sabe qual classe do pacote Componente de Gerência de Dados (cgd) de ODE deve ser instanciada para se ter acesso ao objeto no banco de dados.

Além disso, nem todos os objetos obtidos por meio de mensagens SL estarão persistidos no banco de dados. No caso em que um agente A1 cria um objeto O1 e expressa na mensagem SL que esse objeto é um desejo dele (ou seja, uma situação do mundo que ele quer que se torne verdadeira), então um agente A2 irá receber a mensagem e não terá como obter O1 a partir do banco de dados, porque esse objeto simplesmente não está lá. Nesse caso, o *id* desse objeto terá o valor nulo.

Assim, adaptadores que receberem um *id* válido do *framework* JADE deverão recuperar seu objeto adaptado do banco de dados, ao passo que adaptadores que receberem um *id* nulo não deverão recuperar o objeto no banco de dados.

Pode-se relacionar o padrão adaptador-JADE com o *pipeline* de conversão de JADE, mostrado na Figura 3.13. A partir do conteúdo de uma mensagem, JADE utiliza um *parser*

para a linguagem de conteúdo e depois utiliza um *parser* para a ontologia (que faz as verificações semânticas), obtendo assim, um adaptador-JADE. Assim, é como se o adaptador-JADE fosse mais um *parser/encoder* na figura, que traduz adaptadores-JADE em objetos do domínio do problema de ODE. Desse modo, é possível observar que os adaptadores-JADE têm dois papéis: (i) servir como objetos que têm uma estrutura especificada por FIPA; e (ii) converter objetos do domínio do problema em adaptadores-JADE e vice-versa.

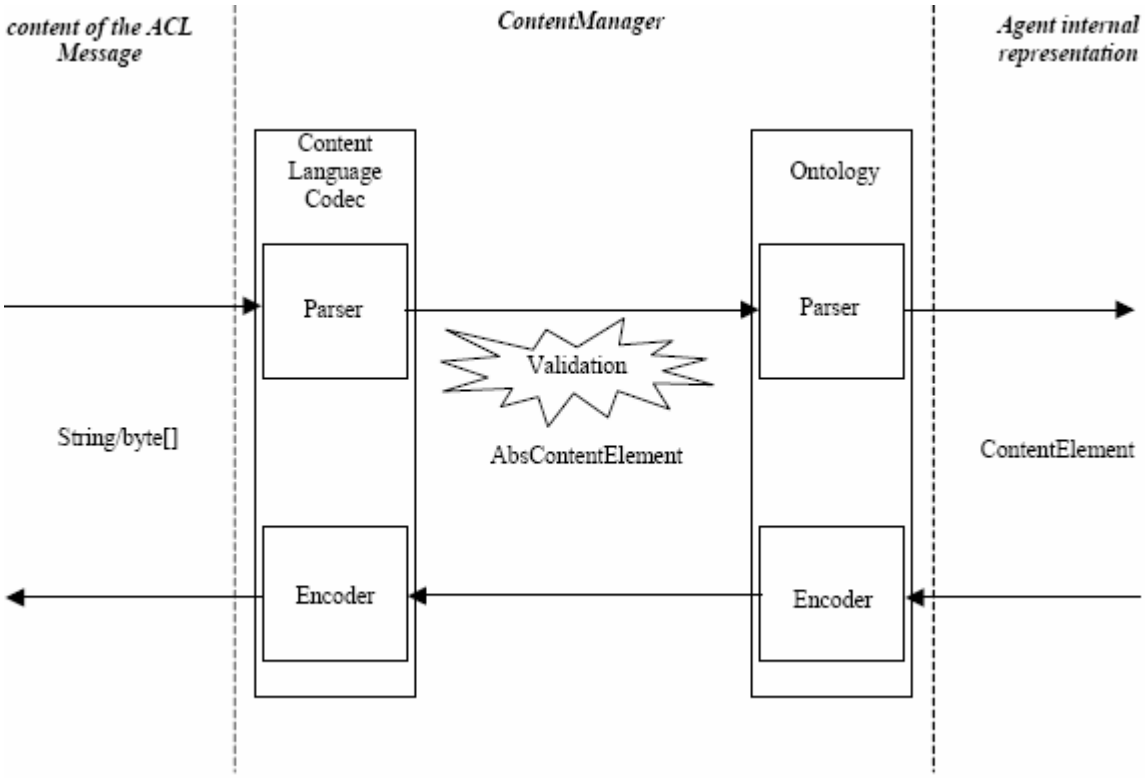

Figura 3.13 – *Pipeline* de conversão de JADE
Fonte: Caire *et al.* (2004)

Vale ressaltar que os adaptadores-JADE e as classes que representam as ontologias-JADE fazem parte do pacote *Ontologia-JADE*, que não faz parte da arquitetura dos agentes, pois são classes externas a um agente. Vale notar, também, que existem várias dependências entre os pacotes de AgeODE e os pacotes de ODE. Por exemplo, o pacote *Ontologia-JADE* depende dos pacotes cdp e cgd e os pacotes *Atuador* e *Observador* dependem dos pacotes cgt e cgv. No entanto, nenhum pacote de ODE depende de um pacote de AgeODE, caso contrário, ODE não poderia executar sem o apoio de agentes, que é um requisito de usabilidade do ambiente.

Um recurso bastante importante de JADE no que diz respeito ao apoio a ontologias é a possibilidade de se combinar ontologias-JADE. Para citar um exemplo de uso desse recurso, a ontologia-JADE de Persistência, definida nesta mesma seção, nunca é utilizada de forma isolada e deve ser reutilizada em todas as outras ontologias-JADE no contexto de AgeODE.

Defende-se que, ao se definir uma ontologia-JADE, é uma boa prática criar uma interface que tenha constantes que definem o vocabulário descrito pela ontologia (BELLIFEMINE *et al.*, 2007). Assim, essas constantes podem ser reutilizadas em outras ontologias-JADE que potencialmente podem ser combinadas com esta, de modo a facilitar a implementação de novas ontologias. Para isso, Bellifemine *et al.* (2007) definem um padrão, chamado *Vocabulary Interface Pattern*, apresentado na Figura 3.14, no qual a ontologia-JADE *ExtendedOntology* combina as ontologias-JADE *Base1Ontology* e *Base2Ontology* e ainda pode adicionar seus próprios conceitos, predicados e ações de agente.

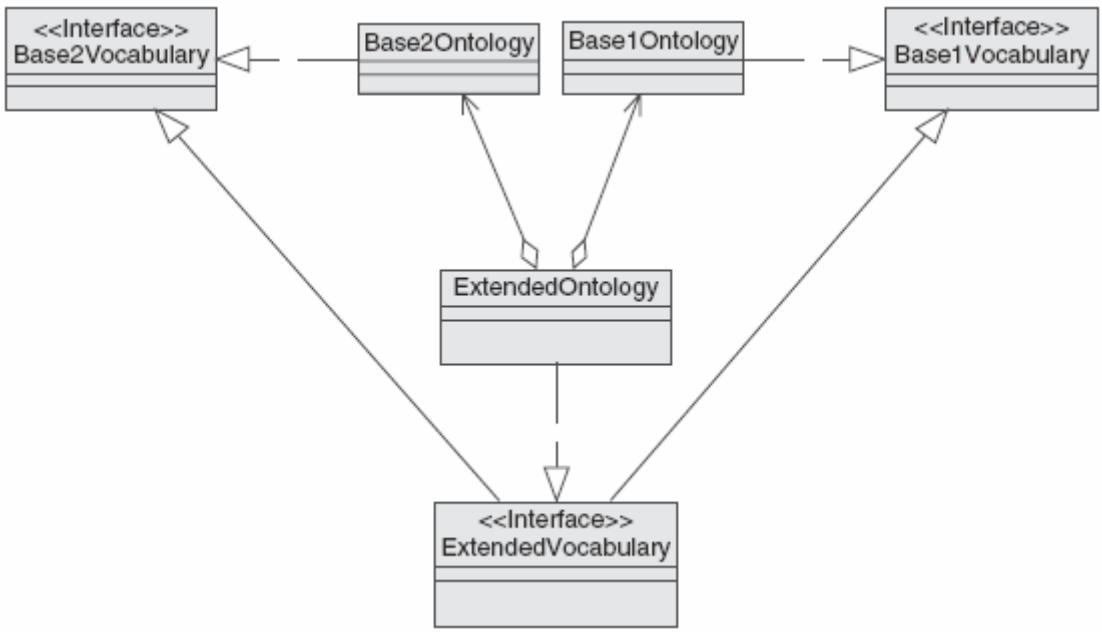

Figura 3.14 – *Vocabulary Interface Pattern*
Fonte: Bellifemine *et al.* (2007)

Entretanto, não é considerado boa prática definir uma interface de constantes e implementá-la nas classes em que se deseja utilizar essas constantes (BLOCH, 2001). Para corrigir essa prática antiga, foi incluído, no Java versão 5, um recurso chamado *static import*, em que se pode importar constantes estáticas de uma classe (ou interface) e utilizá-las como se elas tivessem sido declaradas localmente. Assim, é tido como prática atual que se deve

definir uma classe abstrata com construtor privado e inserir lá as constantes. Dessa forma, pode-se utilizar composição para o reúso de constantes, ao invés de herança.

Mas a combinação de ontologias-JADE é ainda um recurso com uma limitação considerável. Não é possível que se reutilize um único elemento (ou um subconjunto de elementos) de uma ontologia-JADE. Ou seja, ou uma ontologia-JADE reutiliza todos os elementos de outra ontologia-JADE ou não reutiliza nenhum.

Conforme citado no capítulo anterior, existe o *plugin* beangenerator (BEANGENERATOR, 2007) para o editor de ontologias Protégé (PROTÉGÉ, 2008), que permite definir uma ontologia no Protégé e, a partir dela, gerar automaticamente a classe da ontologia, bem como as classes de conceitos, predicados e ações de agente. Apesar de esse ser um recurso interessante, no contexto de AgeODE, não é possível gerar classes de conceitos, pois essas são implementadas como adaptadores-JADE. Mesmo que não existissem os adaptadores-JADE, esse reúso não seria aproveitado, visto que as classes de domínio do problema de ODE já existem.

Aparte as abordagens discutidas de AgeODE para se utilizar o apoio a ontologias de JADE, a nova versão de AgeODE não define nenhuma diretriz especial para se utilizar o apoio a linguagens de conteúdo. Ainda assim, vale salientar que, além dos benefícios de FIPA-SL apresentados no capítulo anterior, o uso da linguagem se mostra interessante, visto que se pode utilizar, em um primeiro momento, apenas o subconjunto SL0, ou mesmo SL1, e, na medida em que os agentes de ODE passem a modelar seus estados mentais utilizando lógica modal, como é feito, por exemplo, quando se utiliza o modelo de agentes BDI (*belief, desire, intention*) (RAO *et al.*, 1995), pode-se utilizar SL2, de maneira a desfrutar do potencial total de FIPA-SL.

É importante notar, no entanto, que a conversão entre objetos Java e mensagens SL nem sempre é automática e, à medida que se utilizam recursos mais poderosos de FIPA-SL, a conversão se torna mais manual (CAIRE *et al.*, 2004).

Em AgeODE, a manipulação de mensagens ACL e, portanto, a manipulação de ontologias e linguagens de conteúdo é concentrada no pacote *Manipulação de Mensagens*. Esse pacote obteve um dos mais altos graus de reúso na infra-estrutura, visto que operações de criação e análise de mensagens são, de certo modo, uniformes.

3.3.5 – Observação do Ambiente

De acordo com a nova arquitetura de agentes de AgeODE, a observação do ambiente é de responsabilidade dos comportamentos do pacote *Observador*. Esse pacote foi um dos principais focos da evolução de AgeODE neste trabalho, visto que diversas oportunidades de melhoria foram levantadas em seu âmbito, a saber: oportunidades de melhoria (OM9), (OM10), (OM11) e (OM12).

Para minimizar o problema (OM9), que indica que o pacote *Componente de Gerência de Visão* (cgv) de ODE pode ser alterado à medida que aumentam os estudos para se migrar algumas ferramentas de ODE para a *web*, definiu-se como diretriz que os agentes de interface (ou seus observadores) devem observar a ocorrência de eventos do pacote Componente de Gerência de Tarefas (cgt), ao invés de observar a ocorrência de eventos do pacote cgv, desde que isso seja possível. Em alguns casos, um evento de interface dispara uma chamada a um método de uma classe do pacote cgt, ou seja, representa a realização de um caso de uso. Nesses casos, o agente de interface deve interceptar esses eventos a partir de classes do pacote cgt. Se esse não for o caso, não há outra alternativa e os agentes de interface têm que interceptar eventos a partir de classes controladoras do pacote cgv.

Como abordagem para solucionar o problema (OM10), que se refere à interceptação de eventos, definiu-se como diretriz que os observadores devem interceptar eventos logo após eles terem ocorrido, e não antes deles ocorrerem. Vale a pena frisar que, nos casos em que os agentes de interface tenham que interceptar eventos do pacote cgv, eles não devem nunca interceptar eventos a partir de classes de componentes de interface como, por exemplo, um botão contido em uma janela, pois, assim, não seria possível saber se a ação que o usuário tinha a intenção de executar ao se pressionar o botão foi executada com sucesso pelo ambiente ou não. Dessa forma, nesses casos, o agente deve interceptar eventos a partir de classes controladoras do pacote cgv.

Resolvidas essas questões, ainda há o problema da relativa complexidade do código dos observadores (OM11), o que dificultava a criação e a manutenção dos mesmos. Assim, optou-se por adotar uma postura menos purista, na qual, ao invés de desaclopamento total entre agentes e ambiente, existe um baixo acoplamento. A idéia é utilizar o padrão de projeto Observador (GAMMA *et al.*, 1995), de forma que as próprias classes a serem observadas acionam os observadores dos agentes de interface, de modo semelhante ao descrito em (NIKRAZ *et al.*, 2006). Isso é feito por meio de um mecanismo de cadastro de "ouvintes" de notificações eventos, de forma que, quando o evento ocorre, todos os “ouvintes” cadastrados são notificados de sua ocorrência. Dessa forma, tanto a oportunidade de melhoria (OM12)

quanto a (OM11) são tratadas. Além disso, considera-se essa solução como de baixo acoplamento, pois a ferramenta de ODE observada não conhece quem são os "ouvintes" de notificações de eventos (no caso, agentes, mas poderiam haver outros) e, caso o ambiente fosse executado sem o apoio de agentes, a ferramenta poderia ser executada sem problema algum.

Para exemplificar a aplicação do padrão de projeto Observador nos observadores de agentes de interface, suponha o seguinte caso: um agente de interface A1 deseja ser notificado da ocorrência do evento "usuário definiu os riscos a serem gerenciados em um projeto" e um agente de interface A2 deseja ser notificado da ocorrência do evento "usuário planejou ações de mitigação e contingência para um risco em um projeto". Dessa forma, o observador de cada agente de interface deve pedir para receber notificações de eventos da classe *AplGerenciarRiscos*, do pacote cgt, que, assim que concluir com sucesso uma das operações *definirGerenciamentoRiscos* ou *definirAcoesPlanejadas*, aciona os seus observadores indicando qual é o projeto em questão e quais foram os riscos escolhidos para gerenciamento ou quais foram as ações de mitigação e contingência planejadas para um determinado risco, de acordo com a operação realizada. Como mostra a Figura 3.15, que omite parâmetros e tipo de retorno de métodos por simplicidade, os observadores dos agentes, *ObservadorRiscosGerenciados* e *ObservadorAcoesPlanejadas*, herdam de um observador abstrato, *ObservadorAplGerenciarRiscos*, e sobrescrevem os métodos correspondentes aos eventos de que desejam receber notificações de sua ocorrência, de forma a avisar os seus agentes de interface do evento ocorrido. Além disso, cada um dos observadores deve se cadastrar junto a *AplGerenciarRiscos* e deve pedir para ser removido do conjunto de observadores assim que seu agente de interface indicar que não necessita mais ser notificado da ocorrência dos eventos.

Afora abordagens referentes às oportunidades de melhoria identificadas na versão original de AgeODE, esse pacote recebeu outras contribuições, levando-se em conta a arquitetura dos agentes definida neste trabalho e a utilização do *framework* JADE.

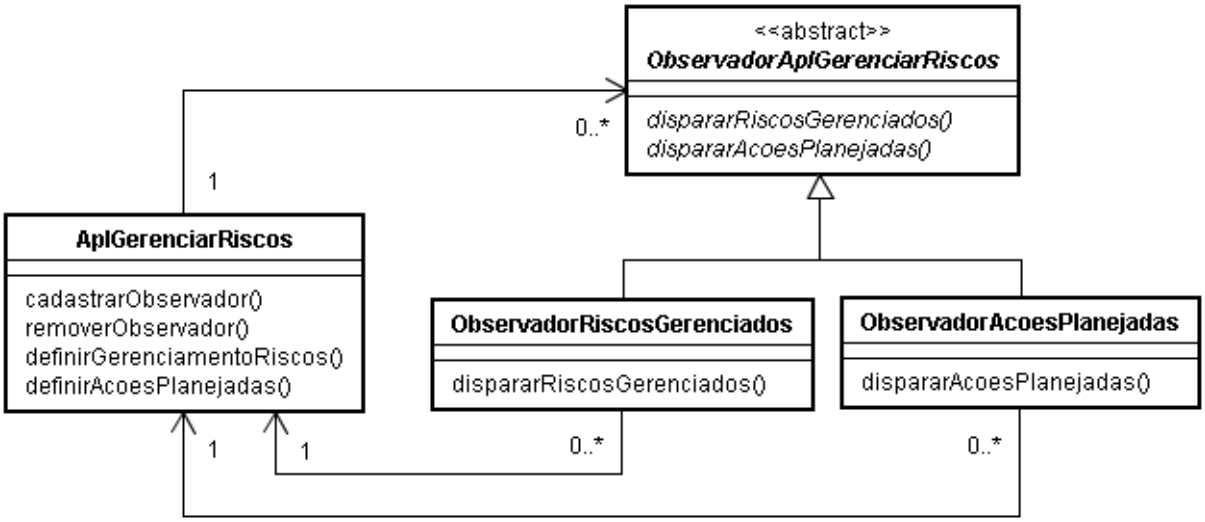

Figura 3.15 – Observadores de Agentes de Interface

De acordo com a nova arquitetura de agentes de AgeODE, a observação do ambiente é de responsabilidade dos comportamentos do pacote *Observador*. Esses comportamentos não herdam de *Behaviour*, visto que eles devem esperar que notificações de eventos sejam disparadas por objetos do ambiente e, portanto, não têm seus métodos invocados pelo *framework* JADE. Além dessas classes não terem seus métodos invocados por JADE, elas têm seus métodos invocados pelo ambiente em que se situa o agente, e não pelo próprio agente, como acontece com as classes do pacote *Manipulação de Mensagens* e algumas classes do pacote *Lógica*. Assim, levando-se em consideração que cada agente de JADE tem uma *thread* e, portanto, todos os comportamentos desse agente têm que executar nessa mesma *thread*, o observador, ao receber uma notificação de ocorrência de evento, deve iniciar um comportamento do agente para que esse processe a notificação. Dessa forma, o próprio *framework* JADE faz com que esse novo comportamento execute na *thread* do agente. Vale notar que, se isso não fosse feito, parte dos comportamentos do pacote *Observador* de um agente seriam executados na *thread* do próprio ambiente, ao invés de serem executados na *thread* do agente.

Tendo isso em mente, vale a pena refinar a diretriz da subseção 3.3.3, que passa a tomar a seguinte forma: “Se um comportamento tem seus métodos invocados por JADE, ele deve herdar de *Behaviour*; se ele tem seus métodos invocados por comportamentos de um agente ou pelo próprio agente, ele não deve herdar de *Behaviour*; e se algum método do comportamento for invocado pelo ambiente em que ele se situa (como é o caso dos observadores), esse método deve iniciar um novo comportamento a ser executado pelo agente, de forma a não utilizar a *thread* do ambiente.”

3.3.6 – Apresentação de Sugestões

O pacote *Atuador* da nova arquitetura de agentes de AgeODE é o responsável pela apresentação de sugestões, visto que esse é um modo de atuar sobre o ambiente, de forma a modificá-lo. A abordagem utilizada no pacote *Atuador* com respeito à apresentação de sugestões dos agentes se mostrou muito importante, considerando-se que as oportunidades de melhoria (OM13) a (OM17) são todas relacionadas à apresentação de sugestões.

Como apontado por O´Leary *et al.* (2001), para ser eficaz, a gestão do conhecimento deve estar embutida nos processos de trabalho. Assim, as ferramentas de ODE podem ser vistas como os locais propícios para a exibição de sugestões dos agentes. Pensando nisso, foi proposto criar um baixo acoplamento entre os agentes e o ambiente, de forma que o agente indique para a ferramenta de ODE qual é a sugestão e essa ferramenta a apresente ao usuário da melhor forma possível. É claro, essa apresentação deve seguir um padrão de interface único para todas as ferramentas, de modo a fazer com que o usuário saiba intuitivamente que aquilo se trata de uma sugestão de agentes. Para isso, sugere-se que as sugestões sejam apresentadas de alguma forma destacadas de vermelho, além de haver uma mensagem que informe isso ao usuário. Mais uma vez, considera-se essa solução de baixo acoplamento, visto que, caso o ambiente seja executado sem o apoio de agentes, a ferramenta pode ser executada sem problema algum. Um exemplo prático dessa solução é apresentado no Capítulo 4, aplicando-se AgeODE para construir um sistema multiagente que apóia a disseminação pró-ativa de conhecimento em GeRis (FALBO et al., 2004b), ferramenta que apóia a Gerência de Riscos em ODE.

Além dessa abordagem de baixo acoplamento resolver o problema (OM13), ela facilita para que seja trabalhada a oportunidade de melhoria (OM14), visto que a ferramenta passa a "entender" qual é a sugestão do agente e pode, além de apresentá-la da melhor forma possível, definir mecanismos para acatar a sugestão do agente automaticamente, caso o usuário permita.

Fazendo referência a JADE e à arquitetura de agentes definida neste trabalho, pode-se perceber que comportamentos do pacote *Atuador* que visam a fazer sugestões aos usuários tipicamente implementam o comportamento reutilizável *OneShotBehaviour*, que, por definição, é um comportamento que é executado apenas uma vez e, então, é finalizado.

No entanto, é provável que os agentes queiram saber até que ponto suas sugestões foram acatadas pelo usuário, assim como já acontecia na versão original de AgeODE. Assim, deve-se utilizar o mecanismo de observação do ambiente, discutido na subseção anterior, de modo que o ambiente notifique ao agente qual foi a ação tomada, para que este possa

comparar essas ações com as ações sugeridas. Em vista disso, um comportamento do pacote *Atuador* que visa a apresentar sugestões ao usuário herda do comportamento de JADE *OneShotBehaviour* e implementa duas operações, a saber: (i) informa à ferramenta do ambiente qual é a sugestão do agente; e (ii) cria um comportamento do pacote *Observador* (que não herda de *Behaviour*), de forma a esperar notificações das ações tomadas pelo usuário.

Além disso, outros pontos fracos identificados na versão original de AgeODE com relação às sugestões dos agentes apontavam que agentes: podiam fazer sugestões com baixa precisão, como, por exemplo, sugerir em excesso itens de conhecimento a serem reutilizados (OM15); não tinham mecanismos para expressar o raciocínio seguido para se chegar a uma sugestão (OM16); e não eram flexíveis ao propor sugestões, no sentido de levar em conta aspectos como a experiência do usuário na ferramenta e a aceitação das sugestões por parte do usuário (OM17).

A curto prazo, identificou-se que é possível tornar mais precisas as sugestões dos agentes, ainda que usando apenas programação imperativa (Java), sobretudo no sistema multiagente que apóia a disseminação pró-ativa de conhecimento em GeRis (SCHWAMBACH, 2004), que foi reformulado devido à reengenharia de AgeODE e é apresentado no Capítulo 4. Além disso, sugere-se que informações relacionadas a como os agentes chegam à sugestão sejam incorporadas à interface em que a sugestão é apresentada. Como diretriz, essa interface deve ter um campo de mensagem em sua parte inferior, que indica o que é a sugestão do agente e como ela é obtida. Isso pode ser facilmente incorporado às ferramentas que utilizam a Interface de Fluxo de Tarefas (COELHO, 2007), como é o caso de GeRis, visto que essa interface reutilizável define um painel de mensagem ao usuário.

Pensando em longo prazo, vale notar que, posteriormente à criação da versão original de AgeODE, uma Infra-estrutura Semântica (RUY, 2006) (PIANISSOLLA, 2007) foi incorporada ao ambiente ODE, que objetiva ampliar as capacidades semânticas do ambiente, representando ontologias por meio da linguagem OWL (*Ontology Web Language*) (MCGUINNESS *et al.*, 2004) e as manipulando utilizando Jena (JENA, 2008), uma biblioteca dotada de um motor de inferência. Dessa forma, a Infra-estrutura Semântica de ODE pode ser integrada a AgeODE, de forma a incorporar capacidade de inferência aos agentes.

Dotar agentes com uma máquina de inferência ataca diretamente as oportunidades de melhoria relacionadas ao aumento da precisão das sugestões dos agentes e à de exibição da linha de raciocínio por trás de cada sugestão, visto que agentes terão à sua disposição mecanismos muito mais poderosos para se chegar às sugestões, a partir de passos lógicos

realizados por um motor de inferência, e que podem ainda servir como mecanismo de exibição do raciocínio que leva a cada sugestão.

Além disso, à medida que se evolui AgeODE, pode-se vislumbrar a adoção de uma arquitetura conceitual para os agentes, visando a facilitar a incorporação de capacidade de aprendizado a respeito dos usuários e de outros agentes do ambiente. Um exemplo de arquitetura conceitual de agentes é o modelo BDI (*belief, desire, intention*) (RAO *et al.*, 1995), que modela atitudes mentais dos agentes por meio de crenças, desejos e intenções, com embasamento em lógica modal.

Entende-se que, com um modelo conceitual que incorpora modelagem de estados mentais de agentes, assim como o modelo BDI, a construção de sugestões personalizadas, levando-se em conta o conhecimento que o agente tem do usuário, se tornará mais natural, uma vez que um usuário pode ser visto como um agente humano (em contraposto aos agentes artificiais de AgeODE) e, portanto, agentes de ODE poderiam modelá-los segundo suas crenças, desejos e intenções.

Vale lembrar que a modelagem de usuários em ADSs já havia sido proposta anteriormente (PEZZIN, 2004) (SILVA, 2001) (REIS *et al.*, 2001), com ênfase maior em agentes de usuário, que podem ser usados para estabelecer o perfil de um usuário, monitorando-o e capturando ações que ele faz com uma certa freqüência.

3.4. Considerações Finais

A primeira versão de AgeODE foi desenvolvida sobre o *framework* para construção de agentes JAT*Lite* (JEON *et al.*, 2000) (JATLITE, 2004) e apresentava diversos problemas, com destaque para o desempenho do ambiente quando executado com agentes.

Tendo em vista os problemas e oportunidades de melhoria identificados, uma nova versão de AgeODE foi desenvolvida, dessa vez utilizando JADE como *framework* base. Constatou-se que diversos serviços poderiam ser muito aperfeiçoados, na medida em que JADE é um *framework* muito mais poderoso que JAT*Lite*. Dessa forma, buscou-se avançar no caminho evolutivo de AgeODE, propondo soluções nos âmbitos de: (i) arquitetura dos agentes; (ii) protocolos de interação; (iii) ontologias e linguagens de conteúdo; (iv) observação do ambiente; e (v) apresentação de sugestões.

Até o início deste trabalho, AgeODE havia sido usada para a construção de um sistema multiagente para a disseminação de conhecimento em duas atividades apoiadas por

ferramentas de ODE: alocação de recursos e gerência de riscos (FALBO *et al.*, 2005a). Entretanto, essa nova versão de AgeODE implica em uma reengenharia dos sistemas multiagente antes existentes. Desse modo, o próximo capítulo discute como aplicar a nova versão de AgeODE na construção de um sistema multiagente no contexto da ferramenta de gerência de riscos.

Capítulo 4

Estudo de Caso: Gerência de Riscos Apoiada por um Sistema Multiagente

Conforme discutido no Capítulo 3, AgeODE, a infra-estrutura para construção de agentes em ODE, foi completamente reestruturada, tendo sido trocado o seu *framework* base, o que leva à necessidade de se validar a sua nova versão. Fez-se, então, uma reengenharia de um sistema multiagente para apoiar a Gerência de Riscos em ODE, originalmente construído em (SCHWAMBACH, 2004) utilizando-se a versão antiga de AgeODE.

Segundo o padrão IEEE Std 1540-2001 (IEEE, 2001), o propósito da Gerência de Riscos é identificar potenciais problemas de cunho técnico ou gerencial antes que eles ocorram, de forma que ações possam ser tomadas a fim de reduzir ou eliminar a probabilidade e o impacto desses problemas.

Gerenciar riscos em um projeto de software é uma tarefa complexa, que requer profissionais de muita experiência. Dessa forma, o ideal é que um gerente de projetos experiente seja alocado para essa atividade, mas, infelizmente, muitas vezes organizações se deparam com situações em que não têm profissionais com esse perfil disponíveis para todos os projetos (FALBO *et al.*, 2004b).

Assim, é de suma importância que organizações possam gerenciar seu conhecimento no contexto da gerência de riscos e, portanto, torna-se útil oferecer apoio automatizado a essa atividade, sobretudo quando integrado a um ADS, provendo disseminação pró-ativa de conhecimento.

Este capítulo está organizado da seguinte maneira: a seção 4.1 – Gerência de Riscos – discute brevemente os principais aspectos da atividade de gerência de riscos; a seção 4.2 – Gerência de Riscos em ODE – discute como a atividade de gerência de riscos é apoiada no ambiente de desenvolvimento ODE (FALBO *et al.*, 2003) (FALBO *et al.*, 2005b), sendo apresentada sua ferramenta CASE de apoio à gerência de riscos; a seção 4.3 – Um Sistema

Multiagente para apoiar a Gerência de Riscos – apresenta a reengenharia do sistema multiagente originalmente construído em (SCHWAMBACH, 2004), utilizando-se, dessa vez, a nova versão de AgeODE; por fim, a seção 4.4 apresenta as considerações finais do capítulo.

4.1. Gerência de Riscos

Diversos fatores contribuem para o aumento da complexidade no desenvolvimento de software, tais como inovações tecnológicas e prazos de entrega cada vez mais apertados. Isso faz com que atividades relacionadas ao desenvolvimento e manutenção de software sejam consideradas atividades de risco e, portanto, gerenciar riscos em projetos de software é essencial (FALBO *et al.*, 2004b).

Um risco é um problema potencial, ou seja, a probabilidade de ocorrência de um evento, perigo, ameaça ou situação e suas conseqüências indesejáveis (IEEE, 2001). É a probabilidade de alguma circunstância adversa ocorrer, ameaçando o projeto, o software que está sendo desenvolvido ou a organização (SOMERVILLE, 2003). Os riscos sempre envolvem duas características: incerteza - o evento que caracteriza um risco pode ou não acontecer, e perda - se um risco se tornar realidade, conseqüências indesejáveis vão ocorrer (PRESSMAN, 2006).

Uma implementação bem sucedida de um processo de gerência de riscos, como o sugerido pelo padrão IEEE Std 1540-2001, leva a importantes resultados, dentre eles: (i) riscos são identificados; (ii) a probabilidade e as conseqüências desses riscos são entendidas; (iii) a ordem de prioridade na qual riscos devem ser tratados é estabelecida; (iv) alternativas adequadas de tratamento de riscos (ações de mitigação e contingência) são estabelecidas; e (v) as ações adequadas são selecionadas para riscos que estiverem em um nível acima do limiar aceitável.

Assim, um processo de gerência de riscos geralmente inclui as seguintes atividades (FALBO *et al.*, 2004b):

- Identificação de Riscos: tenta apontar ameaças (riscos) ao projeto. Seu objetivo é identificar o que pode acontecer de errado.
- Avaliação de Riscos: preocupa-se em avaliar o grau de exposição dos riscos identificados, ou seja, estimar sua probabilidade de ocorrência e seu impacto;

- Definição de Riscos a serem Gerenciados: visa a definir prioridades entre os riscos identificados. O objetivo é alocar recursos apenas para os riscos mais importantes, deixando de gerenciar riscos com baixa probabilidade e baixo impacto;
- Planejamento de Ações: propõe-se a planejar ações de mitigação e contingência para os riscos gerenciados (aqueles com maior prioridade). Ações de mitigação visam a reduzir a probabilidade ou o impacto de um risco antes que ele ocorra. Já ações de contingência são ações planejadas para serem executadas quando um risco ocorre, assumindo que as ações de mitigação falharam;
- Monitoração de Riscos: após o início de um projeto, os riscos gerenciados devem ser monitorados. Graus de exposição dos riscos podem ser modificados, novos riscos podem surgir e outros anteriormente identificados podem perder sua relevância. Portanto, é necessário controlar riscos gerenciados, identificar novos riscos, além de executar ações necessárias e avaliar os resultados.

Desse modo, gerenciar riscos em projetos de software permite evitar problemas. Já ignorar riscos é perigoso e pode levar a diversas conseqüências, dentre elas, atraso do projeto, aumento de seus custos ou mesmo o seu cancelamento. Todavia, apesar de sua importância em projetos de software, muitas organizações apresentam dificuldades em se adotar um processo de Gerência de Riscos (FALBO *et al.*, 2004b). De acordo com resultados da pesquisa em Qualidade e Produtividade no Setor de Software Brasileiro realizada em 2001 (PBQP, 2002), apenas 11,8% das organizações de software brasileiras realizavam um processo de Gerência de Riscos.

Além disso, a Gerência de Riscos é uma atividade complexa e de conhecimento intenso. Dessa forma, requer gerentes de projeto experientes, que nem sempre estão disponíveis para todos os projetos. Assim, é de suma importância que organizações gerenciem seu conhecimento a respeito de riscos, de forma que gerentes de projeto novatos possam executar essa atividade, apoiados pelo conhecimento organizacional.

Assim sendo, ferramentas de software que apóiem uma abordagem sistemática de gerência de riscos e que gerenciem o conhecimento organizacional a respeito de riscos são de extrema importância.

4.2. Gerência de Riscos em ODE

Para apoiar a Gerência de Riscos no contexto do ambiente ODE, desenvolveu-se a ferramenta GeRis (FALBO *et al.*, 2004b), que usa a infra-estrutura de Gerência de Conhecimento de ODE (NATALI, 2003) para promover aprendizagem organizacional para a realização das atividades de Gerência de Riscos. Como GeRis foi desenvolvida baseada numa ontologia de riscos (FALBO *et al.*, 2004b), os conceitos relacionados a esse domínio de estudo estão bem definidos.

O planejamento de riscos em GeRis é realizado segundo o processo descrito na subseção anterior. Quando o gerente de projetos inicia a elaboração do plano de riscos, ele tem como primeira tarefa a identificação dos potenciais riscos associados ao projeto. De posse dos riscos identificados, o gerente realiza a análise dos riscos, quando estimativas de impacto e probabilidade de ocorrência são estabelecidas, sendo computado um valor de grau de exposição (GE) pelo produto de impacto e probabilidade. Uma vez definidos seus impactos e probabilidades, é necessário definir, ainda, um ponto de corte, definindo preliminarmente quais riscos serão gerenciados – aqueles que têm grau de exposição acima do ponto de corte serão gerenciados. Logo após, o gerente de projetos pode refinar o conjunto de riscos que serão monitorados, indicando quais riscos efetivamente serão gerenciados ou não durante o desenvolvimento do sistema. Finalmente, para os riscos a serem gerenciados, são definidas ações de contingência e mitigação a serem oportunamente aplicadas.

Como resultado desse processo, é gerado um plano de riscos. Além disso, na atividade de monitoração dos riscos, novas versões do plano de riscos podem ser criadas, sempre se tomando como base a última versão. Desse modo, a monitoração dos riscos é uma atividade periódica, de forma que todo o histórico é mantido nas várias versões do plano de riscos de um projeto.

Como apontado no Capítulo 3, no contexto de AgeODE, a ontologia de domínio em que ODE se baseia é um dos insumos necessários para se construir a ontologia-JADE correspondente. No caso de GeRis, essa ontologia é a ontologia de riscos (FALBO *et al.*, 2004b), cujo modelo conceitual é apresentado na Figura 4.1.

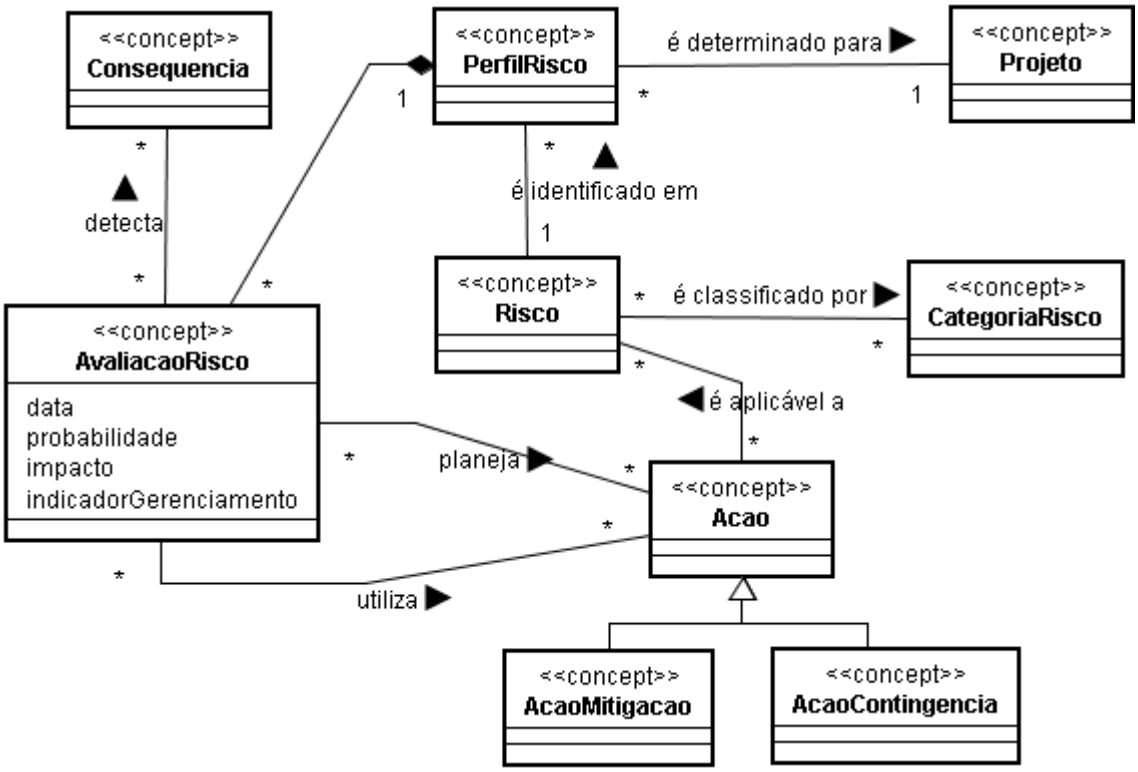

Figura 4.1 – Ontologia de Riscos
Fonte: Falbo *et al.* (2004b)

Uma vez que GeRis oferece apoio baseado em gerência de conhecimento, é importante que essa ferramenta ofereça assistência pró-ativa na disseminação de conhecimento em cada uma das sub-atividades da gerência de riscos (SCHWAMBACH, 2004). Assim, no contexto deste trabalho, o sistema multiagente que visa a disseminar pró-ativamente conhecimento no contexto de GeRis, originalmente construído em (SCHWAMBACH, 2004), foi reincorporado ao ambiente ODE. A seguir, é apresentado o desenvolvimento desse sistema, que foi construído utilizando-se a nova versão de AgeODE e a metodologia OplA (SCHWAMBACH, 2004).

4.3. Um Sistema Multiagente para Apoiar a Gerência de Riscos

A gerência de riscos é uma atividade complexa que, quando apoiada por agentes de software, pode se tornar mais simples de ser realizada, pois o gerente de projetos pode contar com a ajuda dos agentes como disseminadores de conhecimento. O conhecimento armazenado no sistema na forma de experiências em projetos anteriores forma a base de conhecimento a partir da qual os agentes atuam para alcançar seus objetivos, que podem ser

diversos, dependendo da sub-atividade da gerência de riscos, entre eles: identificação, avaliação e definição dos riscos a serem gerenciados e monitorados (SCHWAMBACH, 2004).

Dessa forma, decidiu-se reincorporar o sistema multiagente originalmente construído em (SCHWAMBACH, 2004), de forma a avaliar preliminarmente a nova versão da infra-estrutura de construção de agentes em ODE, proposta neste trabalho.

O sistema multiagente integrado à GeRis é parcialmente descrito a seguir, com destaque para a fase de Projeto, na qual OplA define que deve-se levar em conta a tecnologia utilizada para implementação. Portanto, como houve uma reengenharia de AgeODE, essa fase do desenvolvimento foi revisada.

4.3.1 – Especificação de Requisitos e Análise

Na fase de Especificação de Requisitos, OplA sugere o uso de modelos de casos de uso para capturar e especificar os requisitos funcionais do sistema a ser construído. Deve-se realçar, entretanto, que a identificação dos agentes que atuam na execução dos casos de uso, apesar de fazerem parte do documento que descreve casos de uso, só é feita na fase de análise (SCHWAMBACH, 2004).

Como não houve contribuições deste trabalho na fase de Especificação de Requisitos, o modelo de casos de uso da fase de análise, já com a identificação dos agentes que atuam na execução dos casos de uso, é apenas mostrado na Figura 4.2. OplA sugere que esse diagrama de casos de uso revisado deve ser produzido na atividade de Análise de Casos de Uso, de modo a apresentar as formas de atuação dos agentes. Uma forma de atuação representa um papel exercido pelo agente na execução do caso de uso. Um agente pode atuar de diversas formas num mesmo caso de uso ou em casos de uso distintos. OplA define que as formas de atuação dos agentes são modeladas como casos de uso de extensão com o estereótipo <<atuação do agente>>.

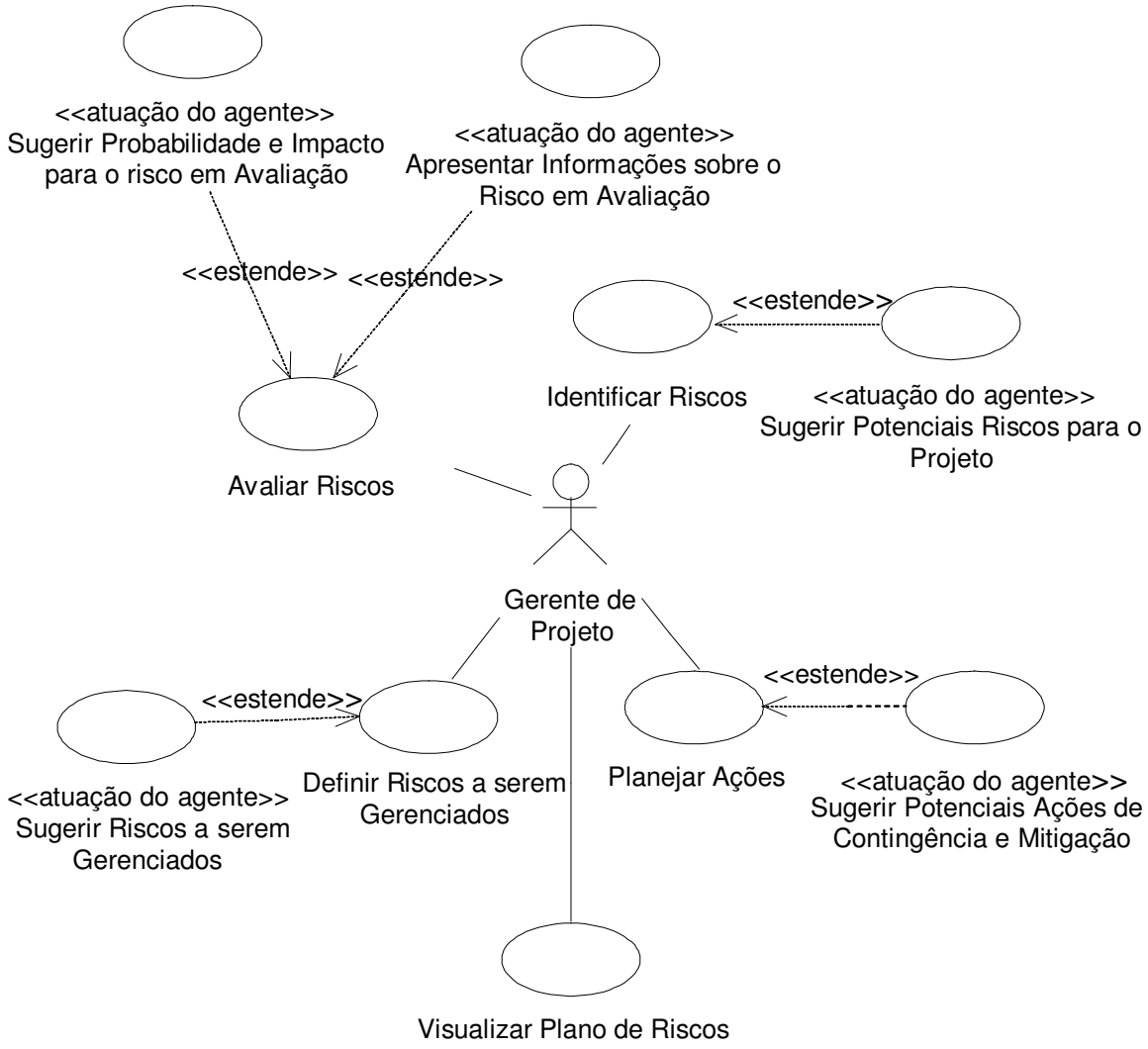

Figura 4.2 – Diagrama de Casos de Uso com as Atuações de Agentes
Fonte: Schwambach (2004)

Além disso, OplA sugere que, no documento de especificação de requisitos revisado, os agentes devem ser identificados por um nome e se deve descrever suas formas de atuação. Os documentos de Especificação de Requisitos e de Análise deste sistema multiagente podem ser encontrados em (SCHWAMBACH, 2004).

Entretanto, há outra atividade definida por OplA para fase de Análise que é pertinente a este trabalho, a saber, a Análise Comportamental. Como proposto por OplA, nessa fase deve-se criar, entre outros, um diagrama de colaboração, de forma a mostrar a interação existente entre os agentes durante sua existência no sistema.

Como discutido no capítulo anterior, OplA tem uma definição do conceito de protocolo diferente da adotada neste trabalho. Portanto, o diagrama de colaboração apresentado na Figura 4.3 não leva em conta protocolos de interação de FIPA.

A partir da Figura 4.3, pode-se perceber que o sistema multiagente concebido por Schwambach (2004) contém dois agentes que atuam de forma global em ODE, *AgAssistentePessoal* e *AgIdentificadorProjetosSimilares*, além de quatro agentes específicos do sistema multiagente que apóia a gerência de riscos, *AgGerenciadorRiscos*, *AgIdentificadorRiscos*, *AgAvaliadorRiscos* e *AgAvaliadorAcoes*. A seguir, encontra-se uma descrição desses agentes, já incorporando melhorias propostas neste trabalho, a saber o uso de um ponto de corte para definir preliminarmente os riscos a serem gerenciados:

- *AgAssistentePessoal*: agente geral atuando no ambiente ODE, responsável por traçar o perfil do usuário no sistema e por iniciar os sistemas multiagentes que atuam em ODE, de acordo com a tarefa que o usuário esteja executando. Portanto, esse agente inicia o sistema multiagente de GeRis;
- *AgIdentificadorProjetosSimilares*: agente que atende a requisições de qualquer agente de ODE que necessite identificar projetos similares. Ele cumpre sua tarefa por meio da Infra-estrutura de Caracterização de ODE (CARVALHO, 2006);
- *AgGerenciadorRiscos*: agente coordenador que coordena as atividades dos demais agentes de seu sistema multiagente. Dessa forma, necessita ser sensível ao contexto de GeRis, no sentido de que deve conhecer as sub-atividades da gerência de riscos e saber qual está em execução;
- *AgIdentificadorRiscos*: agente que tem como objetivo apresentar uma lista de potenciais riscos para o projeto;
- *AgAvaliadorRiscos*: agente que, de posse de um determinado risco, (i) obtém informações relevantes, tais como impacto, probabilidade e índice de ocorrência, (ii) sugere valores para impacto e probabilidade e (iii) sugere, por meio de uma análise dos pontos de corte de planos de riscos de projetos similares e dos grau de exposição dos riscos, quais devem ser gerenciados;
- *AgAvaliadorAcoes*: agente que tem como objetivo sugerir potenciais ações de contingência e mitigação para o risco que foi avaliado.

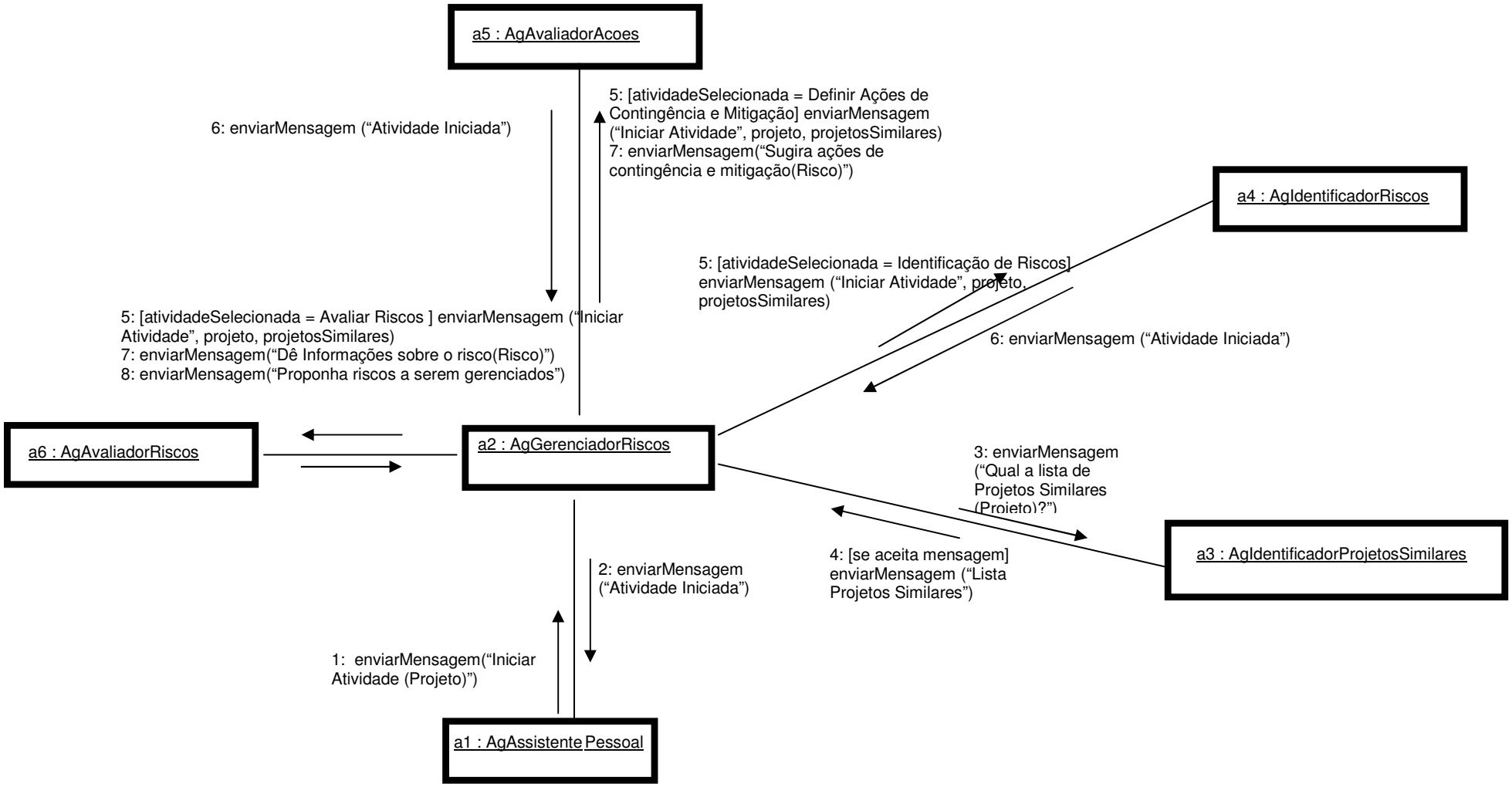

Figura 4.3 – Diagrama de Colaboração dos Agentes que compõem o sistema
Fonte: Schwambach (2004)

4.3.2 – Projeto

De acordo com OplA, a fase de projeto engloba toda a definição feita nas fases anteriores de especificação de requisitos e análise, porém levando-se em consideração a tecnologia a ser utilizada. Assim, o objetivo da fase de projeto é produzir um modelo ou representação de uma entidade que será construída mais tarde. Dessa forma, no contexto deste sistema multiagente, deve-se levar em conta a linguagem Java e a infra-estrutura AgeODE.

Uma das atividades da fase de Projeto é a Modelagem do Ambiente, que tem por objetivo modelar o ambiente no qual os agentes atuam, por meio de um diagrama de classes de objetos. Dessa forma, esse diagrama é o diagrama de classes de GeRis, criado na fase de projeto do desenvolvimento orientado a objetos e apresentado na Figura 4.4.

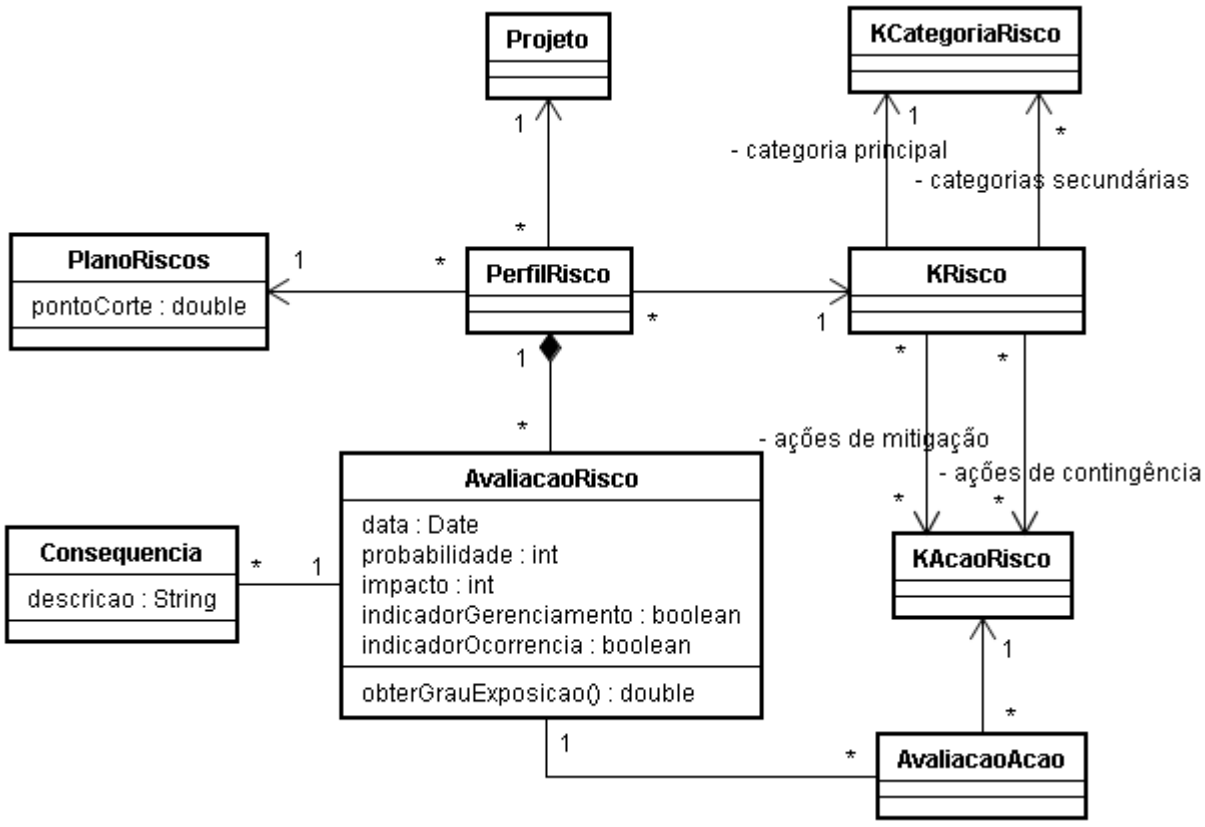

Figura 4.4 – Diagrama de Classes de Objetos

Pode-se perceber que há diferenças entre a ontologia de riscos, apresentada na Figura 4.1, e o diagrama de classes de objetos. Essas diferenças são de caráter da aplicação (uma ontologia de domínio visa a descrever um modelo consensual de uma classe de aplicações) e de caráter tecnológico (o diagrama de classes apresentado leva em conta, por exemplo, que suas classes serão implementadas na linguagem Java).

Além disso, duas diferenças entre o diagrama de classes de objetos da fase de projeto apresentado em (SCHWAMBACH, 2004) e esse são a definição de uma categoria principal para riscos e a adição do atributo *pontoCorte* na classe *PlanoRiscos*.

A primeira alteração permite que se organizem os riscos segundo uma categoria, o que é usual na prática. Já a segunda alteração se baseia no fato de que, tipicamente, os riscos com graus de exposição maiores são gerenciados, enquanto os de graus de exposição menores são, de certa forma, descartados. Dessa forma, o ponto de corte é um limiar, acima do qual estão os riscos identificados para um projeto que têm mais chances de serem gerenciados. Esse valor pode auxiliar gerentes de projetos novatos, de forma que pontos de corte de projetos similares sirvam como conhecimento organizacional.

A partir daí, deve-se revisar o diagrama de classes de agentes definido na fase de análise. Entretanto, identificou-se que é necessário uma revisão dos diagramas de classes de agentes de OplA, de forma a corrigir o conceito de protocolo vigente. No entanto, propor melhorias em OplA não está no escopo deste trabalho e, portanto, somente uma descrição textual dessa atividade é apresentada.

Uma das principais mudanças na nova versão de AgeODE é que os padrões de agentes identificados em (PEZZIN, 2004) não são mais representados por classes abstratas de agentes. Como discutido no Capítulo 3, a nova versão de AgeODE propõe que a identificação dos tipos de cada agente leve a indicações de quais tipos de comportamentos ele deve executar, visto que a nova versão de AgeODE propõe a composição de comportamentos como mecanismo de reúso, em contraposição à herança, como era feito na versão antiga.

Assim sendo, identificaram-se os tipos de cada agente do sistema multiagente proposto para GeRis. *AgGerenciadorRiscos* é claramente um agente coordenador, pois coordena os demais agentes. Além disso, uma vez que ele observa o ambiente e, com base nas ações do gerente de projetos, requisita serviços ao outros agentes, ele também é um agente de interface. Também se pode considerar que o *AgGerenciadorRiscos* é um agente de usuário, visto que ele apresenta sugestões que visam a apoiar os gerentes de projetos a tomar decisões. Os demais agentes, *AgIdentificadorRiscos*, *AgAvaliadorRiscos* e *AgAvaliadorAcoes*, são todos agentes de informação, pois trabalham informações de suas bases de conhecimento para atingir seus objetivos de projeto.

Desse modo, *AgGerenciadorRiscos* deve ter comportamentos do pacote *Comunicação* (por ser um agente coordenador), do pacote *Observador* (por ser um agente de interface) e do pacote *Atuador* (por ser um agente de usuário).

Já os agentes *AgIdentificadorRiscos*, *AgAvaliadorRiscos* e *AgAvaliadorAcoes* devem ter comportamentos do pacote *Lógica* (por serem agentes de informação). Entretanto, como já apontado no Capítulo 3, esse é apenas um indicativo do conjunto mínimo de comportamentos que esses agentes devem ter. De fato, todos esses três agentes de informação também devem ter comportamentos do pacote *Comunicação*, visto que ficam à espera de requisições do *AgGerenciadorRiscos* para que prestem seus serviços.

Além disso, os agentes gerais de ODE utilizados por esse sistema multiagente também tiveram seus tipos identificados. *AgAssistentePessoal* é um agente de interface, pois observa as ações do usuário no sistema e, de acordo com o contexto, inicia sistemas multiagente que visam a apoiar o usuário na realização de suas tarefas. Além disso, ele também é um agente de usuário, visto que foi originalmente concebido para traçar o perfil do usuário no sistema. Entretanto, essa segunda responsabilidade de *AgAssistentePessoal* não é tratada neste trabalho. *AgIdentificadorProjetosSimilares* é um agente de informação, pois apenas fica à espera de requisições para que identifique projetos similares.

Dessa forma, *AgAssistentePessoal* deve ter comportamentos do pacote *Observador* (por ser um agente de interface) e do pacote *Atuador* (por ser um agente de usuário). Já o *AgIdentificadorProjetosSimilares* deve ter comportamentos do pacote *Lógica*, que o permite acessar a infra-estrutura de caracterização do ambiente e identificar quais são os projetos similares a um dado projeto, potencialmente levando-se em consideração uma determinada atividade (por exemplo, pode-se buscar projetos similares levando-se em conta características de um projeto relevantes para a atividade de Análise de Riscos). Além disso, deve ter comportamentos do pacote *Comunicação*, pois responde a requisições de outros agentes.

No que tange a protocolos de interação, identificou-se que todas as interações entre o *AgGerenciadorRiscos* e os seus agentes coordenados seguem o protocolo de interação *fipa-request*, pois ele faz requisições de: (i) sugestão de potenciais riscos para um projeto, (ii) sugestão de impacto para um risco em um projeto, (iii) sugestão de probabilidade para um risco em um projeto, (iv) sugestão de ponto de corte relativo ao grau de exposição para definir riscos a gerenciar, (v) sugestão de riscos a gerenciar, (vi) sugestão de potenciais ações de mitigação, (vii) sugestão de potenciais ações de contingência e (viii) apresentação de informações de um risco. A interação entre o *AgGerenciadorRiscos* e o *AgIdentificadorProjetosSimilares* também segue o protocolo de interação *fipa-request*, pois se trata de uma requisição de identificação de projetos similares.

Conforme apontado no Capítulo 2, JADE tem um recurso interessante, que provê aos agentes um serviço de páginas amarelas, com o qual agentes podem tornar pública para outros

agentes a descrição dos serviços que é capaz de realizar. Dessa forma, agentes podem descobrir, em tempo de execução, quais outros agentes podem lhe auxiliar a executar uma determinada tarefa. Assim, foi definido que o *AgIdentificadorProjetosSimilares* registra seu serviço nas páginas amarelas, de forma que haja um baixo acoplamento entre ele e os agentes que requisitam seu serviço. Em contraposição, o *AgGerenciadorRiscos* e seus agentes coordenados utilizam o serviço de páginas brancas, ou seja, o *AgGerenciadorRiscos* conhece todos os seus subordinados e os contata diretamente.

4.3.2.1 – Ontologias-JADE

Definidos o ambiente, os agentes e suas interações, devem-se criar as ontologias-JADE que darão significado às mensagens trocadas entre os agentes. Deve-se notar que OplA não define essa atividade para a fase de Projeto, visto que, quando originalmente concebida, não levava em conta o *framework* JADE. Entretanto, como é um requisito da nova versão de AgeODE para que os agentes possam se comunicar, definimos algumas ontologias-JADE, a saber: ontologia-JADE de Persistência, ontologia-JADE de Organização e ontologia-JADE de Riscos.

A ontologia-JADE de Persistência foi apresentada na Figura 3.9 e serve de base para todas as outras ontologias-JADE, pois, como discutido no Capítulo 3, todos os conceitos de ontologias-JADE devem herdar do conceito *ObjetoPersistente*, definido na ontologia-JADE de Persistência.

A ontologia-JADE de Organização, apresentada na Figura 4.5, visa a descrever o vocabulário necessário para se comunicar com o *AgIdentificadorProjetosSimilares*.

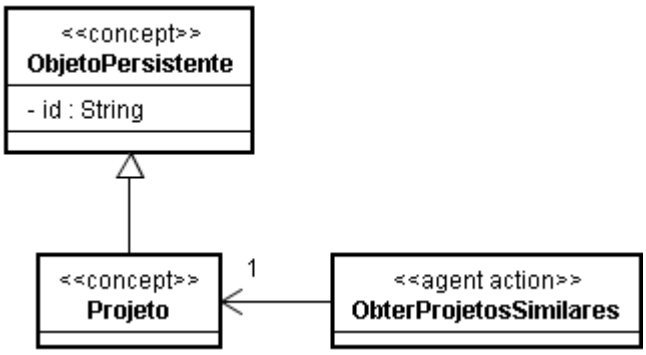

Figura 4.5 – Ontologia-JADE de Organização

Deve ser notado que a ontologia-JADE de Organização, como utiliza o conceito *ObjetoPersistente* da ontologia-JADE de Persistência, depende da mesma. Além disso, como o conceito *Projeto* é utilizado na ontologia-JADE de Riscos, ela dependerá da ontologia-JADE de Organização, além de depender da ontologia-JADE de Persistência, pois contém conceitos, que, no contexto de AgeODE, sempre herdam de *ObjetoPersistente*. Assim, a Figura 4.6 apresenta o diagrama de pacotes que representa a combinação de ontologias-JADE no contexto do sistema multiagente de apoio à gerência de riscos.

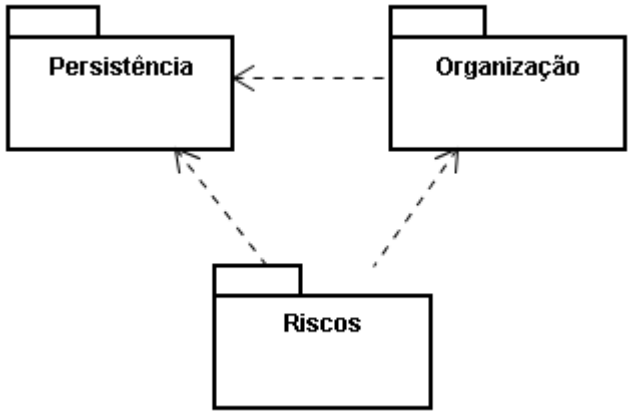

Figura 4.6 – Combinação de Ontologias-JADE

A ontologia-JADE de Riscos, por ser relativamente grande, foi desmembrada em dois diagramas. O primeiro diagrama contém apenas os conceitos da ontologia-JADE e é apresentado na Figura 4.7, omitindo-se, por clareza, o fato de que todos os conceitos herdam de *ObjetoPersistente*.

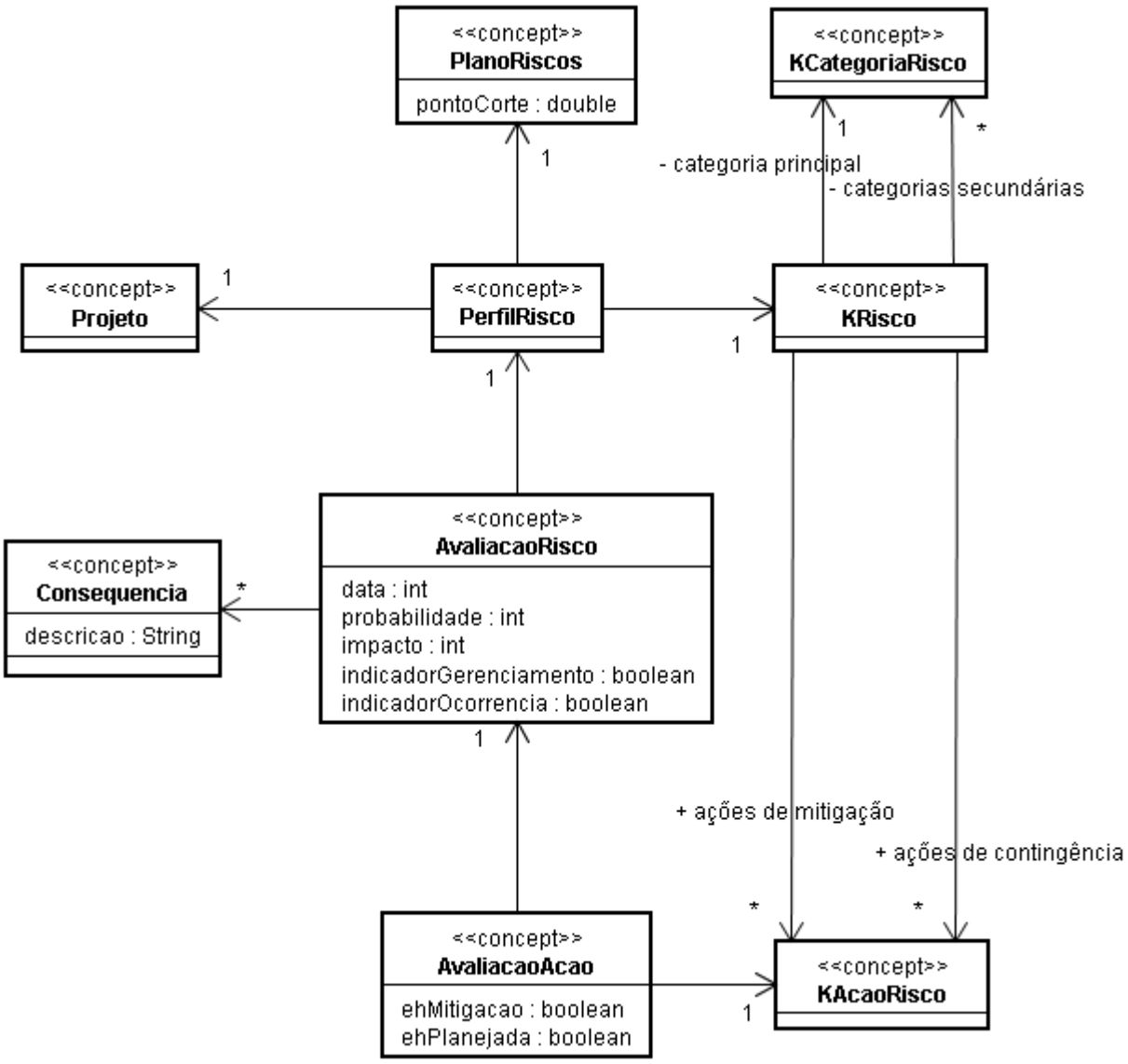

Figura 4.7 – Ontologia-JADE de Riscos: Modelo Conceitual

Como discutido no Capítulo 3, não é possível ter relacionamentos com navegabilidade dupla em ontologias-JADE. Dessa forma, o relacionamento de navegabilidade dupla entre *AvaliacaoAcao* e *AvaliacaoRisco*, exibido no diagrama de classes da Figura 4.4, foi transformado em um relacionamento de navegabilidade simples. O mesmo ocorre com os relacionamentos entre *AvaliacaoRisco* e *PerfilRisco* e entre *AvaliacaoRisco* e *Consequencia*. Deve ser notado que, conforme discutido no Capítulo 3, ciclos formados por associações também podem levar JADE a um laço infinito, no entanto, não se tem esse caso nessa ontologia-JADE.

Além disso, adaptadores-JADE foram criados para cada conceito das ontologias-JADE, de modo a isolarem as classes do pacote Componente de Domínio do Problema (cdp) de ODE do *framework* JADE, conforme discutido no Capítulo 3. Os predicados e ações de

agentes não precisam ter adaptadores-JADE associados, visto que não existem classes correlatas no ambiente ODE para serem adaptadas.

A outra parte da ontologia-JADE de Risco é apresentada na Figura 4.8. Esse diagrama contém as ações de agentes e os conceitos associados a elas. Vale destacar que não foi necessário criar predicados na ontologia-JADE de Riscos. Além disso, deve ser ressaltado que, se fossem acatadas as sugestões de Nikraz *et al.* (2006), o diagrama da Figura 4.8 seria toda a ontologia-JADE de Riscos. No entanto, como apontado no Capítulo 3, AgeODE sugere que as ontologias-JADE espelhem o máximo possível as ontologias de domínio e, portanto, devem ser integrados, também, os conceitos presentes no diagrama da Figura 4.7 e não presentes no diagrama da Figura 4.8.

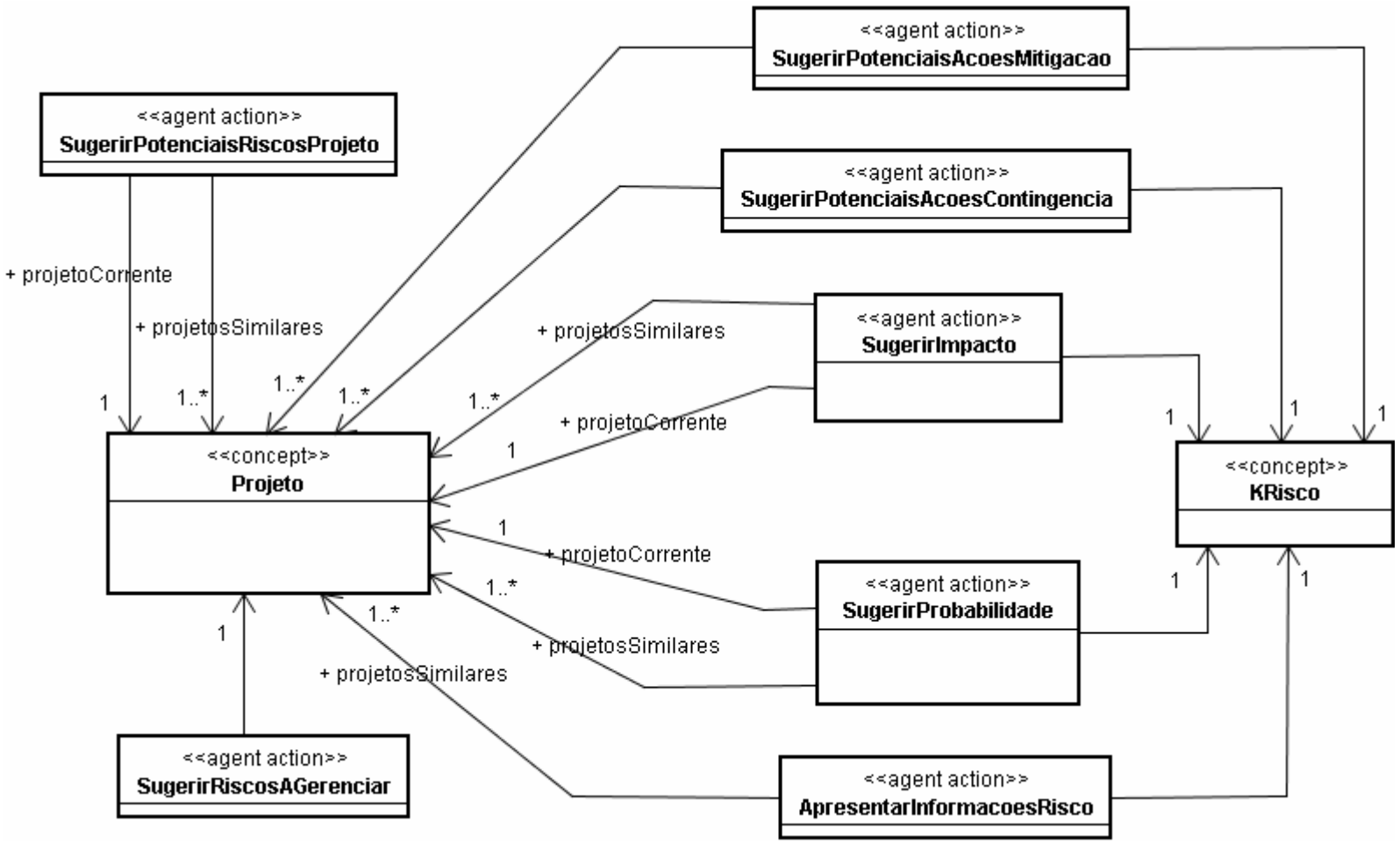

Figura 4.8 – Ontologia-JADE de Riscos: Ações de Agentes

4.3.2.2 – Lógica dos Agentes

Como discutido no Capítulo 3, identificou-se que é possível tornar mais precisas as sugestões dos agentes, ainda que usando apenas programação imperativa (Java). Por exemplo, na atividade de Identificação de Riscos, os agentes não mais sugerem todos os riscos identificados em projetos similares, mas apenas riscos identificados em pelo menos 50% dos projetos similares.

Na atividade de Avaliação de Riscos, tanto para a sugestão da probabilidade quanto para sugestão do impacto, fez-se uma média ponderada combinando-se o grau de similaridade entre os projetos e os valores obtidos de probabilidade e impacto nos projetos similares. Desse modo, projetos que tenham um grau de similaridade maior influenciam mais na sugestão dos agentes.

Procedimento similar foi utilizado na atividade de Definição de Ponto de Corte para a Gerência de Riscos. Além disso, na atividade Refinar Riscos Gerenciados, os riscos acima do ponto de corte sugerido são os riscos sugeridos pelos agentes.

Na atividade de Planejamento de Ações utilizou-se uma abordagem similar à da atividade de Identificação de Riscos. Ou seja, ações planejadas como ações de contingência para pelo menos 50% dos projetos similares são sugeridas como ações de contingência. Procedimento análogo é feito para ações planejadas como ações de mitigação.

4.3.3.3 – Apresentação de Sugestões

Como discutido no Capítulo 3, abandonou-se a abordagem de uma interface padrão para apresentação de sugestões de agentes. A nova versão de AgeODE define que as sugestões devem estar embutidas na própria ferramenta de trabalho do usuário. Além disso, as apresentações de sugestões devem seguir um padrão de interface único para todas as ferramentas, de modo a fazer com que o usuário saiba intuitivamente que aquilo se trata de sugestões de agentes. Para isso, AgeODE define como diretriz que as sugestões sejam apresentadas de alguma forma destacadas de vermelho, além de haver uma mensagem que informe isso ao usuário.

Dessa forma, a Figura 4.9 apresenta a tela em que o gerente de projetos identifica riscos para um projeto em GeRis. Pode-se notar que dois dos riscos estão destacados em vermelho e que a mensagem no canto inferior da tela informa ao usuário que isso se trata de sugestões dos agentes.

De modo similar, a Figura 4.10 apresenta a tela de GeRis em que o gerente de projetos refina a seleção dos riscos a serem identificados, após uma seleção preliminar ter sido feita definindo-se o ponto de corte do plano de riscos. Pode-se notar que os riscos com graus de exposição (G.E.) mais altos foram sugeridos pelos agentes para serem gerenciados.

Figura 4.9 – Apresentação de Sugestão dos Agentes na etapa Identificar Riscos

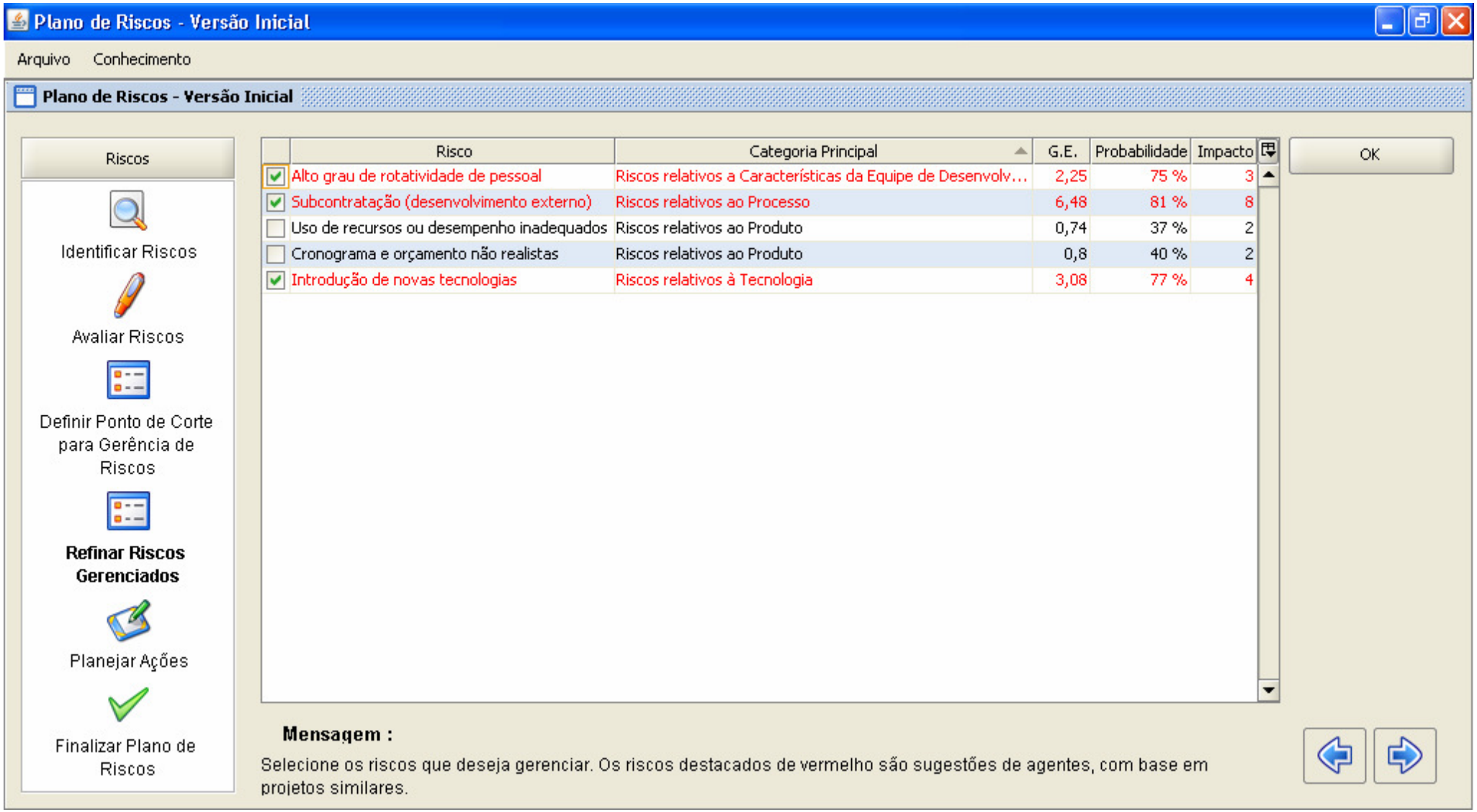

Figura 4.10 – Apresentação de Sugestão dos Agentes na etapa Refinar Riscos Gerenciados

4.4 – Conclusões do Capítulo

A gerência de riscos tem tido sua importância cada vez mais reconhecida como um fator essencial para o sucesso do desenvolvimento profissional de software. Entretanto, é um processo complexo, que requer profissionais experientes e uma abordagem sistemática a ser seguida.

Esse problema pode ser minimizado utilizando-se apoio automatizado, como o fornecido por GeRis, de forma que o conhecimento organizacional seja disseminado, fazendo com que gerentes de projeto novatos possam realizar a gerência de riscos mais facilmente. Pensando nisso, foi construído um sistema multiagente que acessa as infra-estruturas de Caracterização e de Gerência de Conhecimento do ambiente ODE, de forma a apoiar os gerentes de projetos com conhecimento obtido a partir de projetos anteriores.

A construção desse sistema multiagente também proporcionou que se avaliasse, de forma preliminar, a nova versão da infra-estrutura AgeODE, proposta no Capítulo 3. A nova versão oferece ao desenvolvedor mais recursos, como protocolos de interação, ontologias-JADE, uma linguagem de conteúdo expressiva e um serviço de páginas amarelas. Além disso, percebeu-se que, além de ser possível construir sistemas mais poderosos e flexíveis, o desenvolvimento se tornou mais simples e fácil, visto que o novo *framework* base automatiza parcialmente algumas tarefas que antes eram feitas manualmente, como a criação e análise de mensagens trocadas por agentes. Além disso, as novas abordagens de AgeODE para a construção dos observadores e para a apresentação de sugestões são muito mais simples de serem desenvolvidas, mas ainda mantêm a premissa básica de que o ambiente ODE deve poder ser executado sem o apoio de agentes. Um outro ponto de destaque é que a nova abordagem proposta para apresentação de sugestões tem se mostrado mais efetiva, no sentido de que agora os usuários do ambiente recebem sugestões embutidas na própria ferramenta de trabalho.

Capítulo 5

Considerações Finais

Diante da crescente demanda de produtos de software, cada vez é mais importante para as organizações de software produzir com agilidade e qualidade. Crescente também é a complexidade dos sistemas nos dias atuais. Para atender esses requisitos do mercado, é fundamental a utilização de ferramentas adequadas e meios de se aproveitar, ao máximo, o capital intelectual existente na organização (RUY, 2006).

Com o intuito de atender a essa demanda, esforços têm sido despendidos no sentido de se criar Ambientes de Desenvolvimento de Software com Gerência de Conhecimento, de forma a oferecer ferramentas que apóiem as mais diversas atividades envolvidas no processo de software, fazendo com que elas trabalhem em conjunto e apóiem a organização a gerenciar seu conhecimento.

Neste contexto, a tecnologia de agentes emerge com grande potencial, no sentido de se criar novos caminhos para que algumas tarefas do processo de software sejam apoiadas pelo ADS de maneira autônoma. Além disso, a natureza pró-ativa dos agentes tem estimulado bastante o seu uso para disseminar conhecimento, de forma a elevar o aprendizado organizacional.

Entretanto, essa é uma tecnologia nova, que ainda não tem ferramentas com a mesma maturidade das ferramentas de apoio à orientação a objetos. Linguagens de programação orientada a agentes se mostraram necessárias, de modo a prover os desenvolvedores com primitivas e conceitos da tecnologia de agentes de forma nativa. Entretanto, enquanto não surgem linguagens de programação orientada a agentes que sejam amplamente aceitas e que provejam interoperabilidade com sistemas orientados a objetos, uma alternativa para se desenvolver sistemas multiagente é utilizar *frameworks* que criam um novo nível de abstração, fornecendo conceitos da orientação a agentes por meio de uma linguagem com primitivas da orientação a objetos.

Como este trabalho se situa no contexto de ODE, um ADS construído utilizando-se a linguagem de programação Java, é natural que se busque soluções com *frameworks* basedos em Java, tais como JAT*Lite* ou JADE, para apoiar a construção de agentes.

Assim, foi construída, originalmente em (PEZZIN, 2004), AgeODE, uma infra-estrutura para a construção de agentes em ODE que se baseava em JAT*Lite*. Este trabalho dá origem a uma nova versão de AgeODE, propondo, entre outros, a mudança de seu *framework* base para JADE, um *framework* que se mostrou ser muito mais poderoso que JAT*Lite*.

Este capítulo apresenta as considerações finais a respeito do trabalho desenvolvido. Na seção 5.1 são apresentadas as conclusões e principais contribuições do trabalho e na seção 5.2, são discutidas perspectivas futuras para dar continuidade ao trabalho de pesquisa.

5.1. Conclusões

Este trabalho teve início com o levantamento de algumas oportunidades de melhoria em relação à primeira versão de AgeODE, as quais apontam para um possível caminho evolutivo para AgeODE. A troca do *framework* base de JATLite para JADE foi um marco importante em AgeODE. Ela tratou diversos problemas detectados, tal como o problema do desempenho da versão anterior, fazendo com que usuários de ODE possam efetivamente utilizar o ambiente com apoio de agentes. O uso de JADE também fez com que se pesquisasse as especificações de FIPA, levando à detecção da necessidade de se corrigir o conceito de protocolo de interação vigente na versão anterior de AgeODE.

Entretanto, a adoção de JADE impôs novos desafios. Foi necessário repensar a forma com que AgeODE lidava com os padrões de agentes identificados em (PEZZIN, 2004). A necessidade de haver uma "biblioteca" de comportamentos reutilizáveis deu início a esforços no sentido de se poder construir comportamentos executáveis por qualquer classe de agentes, resultando em um padrão para se construir comportamentos reutilizáveis e em uma nova arquitetura para os agentes. Essa nova arquitetura faz com que os desenvolvedores de agentes no contexto de AgeODE tenham que mudar a forma de pensar, passando a utilizar um mecanismo de composição de comportamentos para se poder desenvolver os serviços dos agentes.

O modo com que JADE trabalha com ontologias, diferente da forma com que ODE já vinha as utilizando, também deu início a novas pesquisas. Em um primeiro momento, fez-se uma distinção na nomenclatura, chamando as ontologias no contexto de JADE de ontologias-

JADE. A partir daí, definiu-se uma abordagem para se construir ontologias-JADE, de forma que os agentes pudessem discursar sobre os objetos a que tinham acesso utilizando-se o banco de dados. Pesquisas nesse âmbito ainda deram fruto a um padrão de projeto para se isolar as classes de ODE do *framework* JADE.

Novas propostas relativas à observação do ambiente e à apresentação de sugestões dos agentes também foram foco de pesquisa, visando a aumentar a usabilidade da infra-estrutura, ou seja, proporcionar meios para que o desenvolvimento de agentes e de sistemas multiagente seja mais simples. Além disso, a nova abordagem para apresentação de sugestões se mostrou mais efetiva, não desviando a atenção do usuário para janelas específicas de sugestões.

O trabalho discutiu, ainda, a reengenharia do sistema multiagente para apoiar a Gerência de Riscos, originalmente construído em (SCHWAMBACH, 2004). Utilizou-se desse sistema para avaliar a nova versão da infra-estrutura AgeODE e, como efeito colateral, identificou-se algumas oportunidades de melhoria na metodologia OplA (SCHWAMBACH, 2004). Além disso, propostas, ainda que simples, para melhorar a eficiência das sugestões no contexto desse sistema multiagente foram feitas.

Diante dos pontos discutidos, é possível apontar como as principais contribuições deste trabalho:

- Troca do *framework* base de AgeODE, tratando o problema do desempenho e criando oportunidades para muitos outros serviços, visto que JADE é um *framework* muito mais poderoso que JAT*Lite*;
- Adaptação da forma como AgeODE lida com os padrões de agentes, levando-se em conta o *framework* JADE, passando a trabalhar com composição de comportamentos ao invés de herança de sub-classes de agentes e implementação de interfaces de agentes;
- Definição da arquitetura dos agentes, de forma a adotar uma organização baseada em componentes;
- Correção do conceito de Protocolo de Interação vigente na versão original de AgeODE;
- Padrão para se criar comportamentos reutilizáveis, sem que seja necessário abrir mão do encapsulamento;
- Definição de uma abordagem preliminar para se desenvolver ontologias-JADE no contexto de AgeODE, propondo uma forma dos agentes poderem discursar sobre

objetos compartilhados e definindo o uso de adaptadores-JADE, que permitem que as classes de ODE não dependam do *framework* JADE;

- Simplificação da abordagem para a construção de observadores de agentes de interface;
- Abordagem mais efetiva para se apresentar sugestões de agentes aos usuários de ODE;
- Reengenharia do sistema multiagente para apoiar a Gerência de Riscos.

São visíveis algumas vantagens da nova versão de AgeODE. JADE por si só proporcionou várias melhorias. Além dessas, outras foram desenvolvidas neste trabalho, visando a tornar AgeODE uma infra-estrutura mais flexível, como, por exemplo, a definição de uma arquitetura para os agentes, o padrão para se criar comportamentos reutilizáveis e a nova abordagem para se apresentar sugestões de agentes. Contudo, sabe-se que essa versão de AgeODE é apenas mais um passo em seu caminho evolutivo. Assim, a próxima seção apresenta perspectivas futuras de evolução deste trabalho.

5.2 – Perspectivas Futuras

Buscando-se melhorar e expandir a infra-estrutura proposta, algumas perspectivas de trabalhos futuros podem ser destacadas. Algumas delas representam apenas melhorias funcionais e correção de pontos falhos do trabalho. Outras devem ser trabalhadas em um âmbito maior e representam evoluções deste trabalho.

Inicialmente, cabe listar alguns pontos fracos do trabalho. A maioria deles referente ao sistema multiagente de apoio à Gerência de Riscos, que, após as experimentações, pôde evidenciar algumas de suas falhas, e à infra-estrutura AgeODE, que ainda tem muito a evoluir.

O primeiro item é relativo a melhorias no sistema multiagente desenvolvido neste trabalho. Percebeu-se que, ao sugerir riscos a serem gerenciados para o projeto, agentes levam em conta apenas os pontos de corte adotados nos projetos similares. Entretanto, pode ser que haja riscos específicos que, mesmo que tenham grau de exposição baixo, sejam, na maioria das vezes, monitorados nos projetos.

Além disso, a Infra-estrutura de Caracterização de ODE (CARVALHO, 2006) permite que sejam identificados projetos similares potencialmente levando-se em consideração uma determinada atividade (por exemplo, pode-se buscar projetos similares levando-se em conta

características de um projeto relevantes para a atividade de Análise de Riscos). Dessa forma, na medida em que esse sistema multiagente se torne mais robusto, pode ser que cada agente subordinado do *AgGerenciadorRiscos* queira identificar projetos similares sob um ponto de vista diferente. Caberá, então, analisar se é o *AgGerenciadorRiscos* que deve identificar os projetos similares e informá-los aos seus subordinados, como é feito atualmente, ou se é melhor deixar a solicitação de identificação de projetos similares a cargo de cada um de seus agentes subordinados.

No que tange ao objetivo final de se construir AgeODE, ou seja, apoiar o ambiente ODE na solução de problemas de forma autônoma e pró-ativa, é necessário que se criem outros sistemas multiagente, de modo que alguns resultados possam ser vistos, mesmo que não se tenha uma infra-estrutura avançada. Ferramentas como as de Engenharia de Requisitos (ReqODE) (MARTINS et al., 2006) e de Alocação de Recursos Humanos (AlocaODE) (COELHO, 2007) podem ser apoiadas por novos sistemas multiagente.

Ainda no que toca à construção de sistemas multiagente, deve-se discutir maneiras de se projetar interfaces com o usuário que facilitem que uma sugestão seja apresentada de forma a permitir que o usuário simplesmente possa acatá-la. A nova abordagem para apresentação de sugestões foi um caminho nessa direção, porém ainda não atingiu todo seu potencial. Criar projetos de interface com o usuário que facilitem a exibição da linha de raciocínio do agente também foi um item não abordado neste trabalho.

A respeito do *framework* JADE e da infra-estrutura AgeODE, detectou-se que as regras impostas por JADE para a implementação das classes que são referenciadas pelas ontologias-JADE[1] são definidas por uma subclasse de *Introspector*. JADE permite que, para cada ontologia-JADE definida, seja associado um *Introspector* (CAIRE *et al.*, 2004). Dessa forma, criando-se um *Introspector* diferente do padrão fornecido por JADE, é possível que se defina as próprias regras de AgeODE para se implementar as classes que são referenciadas pelas ontologias-JADE. Por exemplo, pode-se fazer com que o próprio *Introspector* obtenha os objetos das mensagens FIPA-SL no banco de dados, ao invés de ser necessário fazer com que cada adaptador-JADE obtenha seu objeto adaptado no banco de dados, caso ele seja um objeto persistido.

É válido também criar sistemas multiagente distribuídos. Tomando-se como exemplo o sistema multiagente para apoiar a Gerência de Riscos, *AgIdentificadorProjetosSimilares* não é um agente específico desse sistema multiagente e, portanto, não necessita estar atrelado à

1 No caso de AgeODE, essas classes são os adaptadores-JADE, as classes de ações de agentes e as classes de predicado.

instância de ODE em que GeRis está em execução. Em outras palavras, pode-se implantar o *AgIdentificadorProjetosSimilares* em uma espécie de servidor de agentes genéricos, que pode ser inclusive a mesma máquina que atua como servidor de banco de dados na organização. Dessa forma, vários sistemas multiagente de propósito diferente, espalhados pelas instâncias de ODE, poderiam buscar no serviço de páginas amarelas por agentes que prestassem o serviço de identificação de projetos similares e, após a descoberta, poderiam pedir o serviço a esse agente, que estaria em outra máquina. De fato, podem existir vários agentes do tipo *AgIdentificadorProjetosSimilares* no servidor, fazendo com que nenhum agente fique sobrecarregado. Isso é possível graças à arquitetura da plataforma JADE, que tem como mecanismo central um contêiner principal[2], que seria uma espécie de servidor, onde os outros contêineres se registram. Dessa forma, cada instância de ODE teria um contêiner que se registraria no contêiner principal implantado no servidor.

Além disso, o trabalho de SCHWAMBACH (2004) realizou uma proposta de criação de uma metodologia para apoiar a construção de sistemas multiagente, mas com algumas considerações feitas neste trabalho, as pesquisas feitas podem evoluir. A saber, para apoiar a construção de sistemas multiagente distribuídos, um diagrama de implantação como o descrito em (NIKRAZ *et al.*, 2006) pode ser adicionado, de forma a apoiar na decisão de onde se deve implantar cada agente, talvez até adicionando informações sobre a decisão de se registrar os serviços de um agente em páginas amarelas ou não.

Além disso, devem ser incorporados os protocolos de interação de FIPA em OplA. Na fase de Análise, ao se dizer que uma interação segue o protocolo *fipa-request*, por exemplo, deve-se entender que uma interação equivalente ocorre. Isso não é dependente de tecnologia, visto que FIPA define seus protocolos de interação por meio de modelos abstratos de interação entre agentes. Já na fase de Projeto, caso se esteja utilizando AgeODE ou JADE, pode-se supor que , para o mesmo exemplo, os comportamentos *AchieveREInitiator* e *AchieveREResponder*, que implementam esse protocolo, serão utilizados.

Estudos com vistas a se tentar relacionar o diagrama de base de conhecimento de agentes, definido por OplA, e a criação de uma ontologia-JADE podem ser feitos. Em um primeiro momento, pode-se dizer que, enquanto a base de conhecimento de um agente descreve o mundo sobre o qual ele pode discursar, uma ontologia-JADE descreve um conjunto de elementos comuns entre as bases de conhecimento de uma comunidade de agentes.

---

2 Apesar de JADE tipicamente trabalhar com um contêiner principal, pode-se criar um mecanismo tolerante a falhas, onde existe mais de um contêiner principal (BELLIFEMINE *et al.*, 2007).

Ainda com relação ao diagrama de base de conhecimento dos agentes definido por OplA, deve-se estudar mais a fundo se é útil modelar, também, a base de conhecimento dos comportamentos de um agente. Talvez seja interessante pensar na possibilidade de se tratar o agente como um todo na fase de Análise e na fase de Projeto, levando-se em consideração AgeODE ou JADE, modelar a base de conhecimento de um agente por meio das bases de conhecimento de seus comportamentos.

Vale destacar que combinar metodologias também pode ser proveitoso, tal como faz ARKnowD (GUIZZARDI, 2006). De fato, um sistema multiagente para apoiar ReqODE está sendo desenvolvido utilizando tanto OplA quanto ARKnowD. Assim, ao final desse trabalho, poder-se-á propor combinações entre as metodologias ou mesmo a incorporação de novas facilidades a OplA.

Ponto também importante é aprofundar os estudos para se melhorar a precisão das sugestões dos agentes. Trabalhos que visem a unir AgeODE com a Infra-estrutura Semântica de ODE (RUY, 2006) (PIANISSOLLA, 2007), de modo a dotar agentes de mecanismos de inferência podem avançar nessa área de estudo.

Perspectivas dessa união fazem com que se possa identificar uma limitação de JADE e, portanto, de AgeODE. No contexto da Infra-estrutura Semântica, as ontologias de domínio de ODE são implementadas utilizando-se OWL (*Ontology Web Language*) (MCGUINNESS *et al.*, 2004), uma linguagem que vem despontando como padrão no contexto da *Web* Semântica (BERNERS-LEE *et al.*, 2001). Entretanto, AgeODE utiliza ontologias-JADE, uma forma de se implementar ontologias de que só sistemas construídos utilizando-se JADE se beneficiam. À medida que JADE passe a suportar um padrão aberto, como OWL, agentes construídos utilizando-se AgeODE passarão a se beneficiar das pesquisas realizadas nessa área. Além disso, a integração entre AgeODE e a Infra-estrutura Semântica seria mais natural.

Ainda no que tange a ontologias, pode-se vislumbrar um método sistemático para se criar ontologias-JADE a partir de ontologias de domínio. Se isso realmente for possível, pode-se vislumbrar, também, a criação automática de ontologias-JADE a partir das ontologias descritas no contexto da Infra-estrutura Semântica.

Em um âmbito mais geral de AgeODE, vale também destacar algumas das perspectivas deixadas por este trabalho. Muito foi discutido em prover apoio pró-ativo para disseminação de conhecimento utilizando-se os agentes construídos com AgeODE. No entanto, AgeODE é uma infra-estrutura que, de maneira mais geral, apóia a construção de agentes para atuarem em ODE. Ou seja, podem-se construir agentes com objetivos que não a disseminação de conhecimento.

Por exemplo, organizações de software tipicamente têm um Gerente de Desenvolvimento que coordena os vários projetos da organização, entre outros, alocando Gerentes de Projetos para cada projeto. Os Gerentes de Projetos, além de terem outras responsabilidades, são encarregados de alocar analistas, projetistas e programadores para seus projetos. Essa questão de alocação de recursos humanos é complexa, pois envolve negociação entre as partes envolvidas, prioridades de projetos etc. Um sistema multiagente para apoiar essa atividade poderia ser concebido por agentes que representassem cada recurso humano envolvido, defendendo seus interesses e buscando um objetivo em comum, a prosperidade da organização. Esse é um sistema multiagente distribuído e que não envolve, pelo menos em princípio, a disseminação de conhecimento.

Entretanto, esse é um desafio complexo e deve ser almejado em longo prazo. Entre outros, esses agentes devem modelar os usuários e devem estar aptos a negociar com outros agentes. Como apontado no Capítulo 3, entende-se que os meios para realizar essas tarefas podem se embasar em um modelo conceitual de agentes, como o modelo BDI (*belief, desire, intention*) (RAO *et al.*, 1995). Uma implementação do modelo BDI que permite integração com JADE é Jadex (BRAUBACH *et al.*, 2005) (POKAHR *et al.*, 2005).

Outras evoluções são experimentar outros protocolos de interação, tal como *fipa-contract-net* (rede de contratos), que pode ser utilizado no exemplo citado anteriormente sobre alocação de recursos humanos; e a utilização mais extensa das potencialidades da linguagem de conteúdo FIPA-SL, que são vastas, inclusive com suporte à lógica modal, que é importante quando se utiliza o modelo BDI.

Portanto, muito trabalho ainda deve ser realizado. Os resultados e contribuições apresentados nesta monografia representam um pequeno passo diante de um mundo de possibilidades. De qualquer forma, acredita-se que as contribuições realizadas neste trabalho possam intensificar a criação de sistemas multiagente que sejam capazes de apoiar efetivamente os usuários de ODE na realização de suas tarefas.

Referências Bibliográficas

ABECKER, A., BERNARDI, A., HINKELMANN, K., KÜHN, O., SINTEK, M., "*Towards a Technology for Organizational Memories*", IEEE Intelligent Systems, 40-48, May/June 1998.

AGENTBUILDER, "*An Integrated Toolkit for Constructing Intelligent Software Agents - User's Guide Version 1.3 Rev. 0*", April 30, 2000, Reticular Systems, Inc. Disponível em <http://www.agentbuilder.com>. Acesso em 23 set. 2004.

ARANTES, L.O., FALBO, R.A., GUIZZARDI, G., "*Evolving a Software Configuration Management Ontology*", Second Workshop on Ontologies and Metamodeling Software and Data Engineering – WOMSDE´2007, XXI Simpósio Brasileiro de Engenharia de Software – SBES´2007, João Pessoa. 2007.

ARIDOR, Y., LANGE, D. B., "*Agent Design Patterns: Elements of Agent Application Design*". Proceedings of the second international conference on Autonomous agents May 1998.

AUSTIN, John L., "*How To Do Things with Words*", 1955. Oxford, Oxford University Press. 1955.

BEANGENERATOR. "*BeanGenerator*" <http://protege.cim3.net/cgi-bin/wiki.pl?OntologyBeanGenerator>. 2007. Acesso em 4 jul. 2008

BELLIFEMINE, F., CAIRE, G., GREENWOOD, D., "*Developing Multi-Agent Systems with JADE*." John Wiley & Sons. New York, USA. 2007.

BELLIFEMMINE, F., POGGI, A., RIMASSA, G., "*JADE - A FIPA2000 Compliant Agent Development Environment*". AGENTS'01, Montréal, Quebec, Canada. 2001

BERNERS-LEE, T., HENDLER, J., LASSILA, O., "*The Semantic Web*". Scientific American Magazine. May 2001.

BERTOLLO, G., "*Definição de Processos em um Ambiente de Desenvolvimento de Software*". Dissertação de Mestrado, Mestrado em Informática, UFES, Vitória, Maio 2006.

BERTOLLO, G., FALBO R. A., "*Definição de Processos em um Ambiente de Desenvolvimento de Software Baseado em Ontologias*". Anais do V Simpósio Brasileiro de Qualidade de Software, p. 72-86, Vila Velha, Brasil, Maio 2006.

BIGUS, J. P.: "*The Agent Building and Learning Environment*". Agents 2000, Barcelona, Spain. 2000.

BLOCH, J., "*Effective Java Programming Language Guide*". Item 17. Addison-Wesley. 2001.

BRAUBACH, L., POKAHR, A., LAMERSDORF, W., "*Jadex: a BDI-Agent System Combining Middleware and Reasoning*". In Walliser, M., Brantschen, S., Calisti, M. and Hempfling, T. (eds), Whitestein Series in Software Agent Technologies, Birkh¨auser-Verlag, Springer Science+Business Media, Berlin, New York, 2005.

BRESCIANI, P., GIORGINI, P., GIUNCHIGLIA, F., MYLOPOULOS, J., PERINI, A., "*Tropos: An Agent-Oriented Software Development Methodology*". International Journal of Autonomous Agents and Multi Agent Systems, 8(3):203–236. 2004.

BRUGALI, D., SYCARA, K., "*Towards Agent Oriented Application Frameworks*". ACM, 2000.

CAIRE, G., CABANILLAS, D., "*JADE Tutorial: Application-Defined Content Languages and Ontologies*". 2004. Disponível em: < http://jade.tilab.com/doc/tutorials/CLOntoSupport.pdf>. Acesso em: 8 jul. 2008.

CAIRE, G., LEAL, F., CHAINHO, P., EVANS, R., GARIJO, F., GOMEZ, J., PAVON, J., KEARNEY, P., STARK, J., MASSONET, P., "*Agent Oriented Analysis using MESSAGE/UML*". AOSE 2001.

CARVALHO, V.A., "*Gerência de Conhecimento e Decisão em Grupo: um Estudo de Caso na Gerência de Projetos*". Dissertação de Mestrado, UFES, Vitória, Brasil, 2006.

CARVALHO, V.A., ARANTES, L.O., FALBO R.A., "*EstimaODE: Apoio a Estimativas de Tamanho e Esforço no Ambiente de Desenvolvimento de Software ODE*". Anais do V Simpósio Brasileiro de Qualidade de Software, p. 12-26, Vila Velha, Brasil, Maio 2006.

CHRISTIE, A. M., "S*oftware Process Automation – The Technology and its Adoption*", Peittsburghm Pennsylvannia, Springer-Verlag Berlin Heidelberg, 1995.

COELHO, A.G.N., "*Apoio à Gerência de Recursos em ODE*". Projeto de Graduação, Curso de Ciência da Computação, UFES, Vitória, 2007.

COLLIS, J., NDUMU, D., "*The ZEUS Agent Building Toolkit*" – ZEUS Technical Manual, September, 1999.

COPLIEN, J. O., "*Software Patterns*". Disponível em <http://hillside.net/patterns/definition.html>. 1995. Acesso em 4 jul. 2008.

COST, R. S., FININ, T., LABROU, Y., LUAN, X., PENG, Y., SOBOROFF, I., "*Agent Development with Jackal*". Autonomous Agents '99 Seattle WA USA. 1999.

DAL MORO, R., "*Avaliação e Melhoria de Processos de Software: Conceituação e Definição de um Processo para Apoiar a sua Automatização*". Dissertação de Mestrado, UFES, Vitória, Brasil, 2008.

DAL MORO, R., NARDI, J.C., FALBO, R.A., "*ControlPro: Uma Ferramenta de Acompanhamento de Projetos Integrada a um Ambiente de Desenvolvimento de Software*". XII Sessão de Ferramentas do Simpósio Brasileiro de Engenharia de Software, SBES'2005, Uberlândia, Brasil, Outubro 2005.

DEVEDZIC , V., "*Ontologies: Borrowing From Software Patterns*". Intelligence ACM - September 1999. Volume 10 Issue 3. 1999

DIGNUM, V., "*A Model for Organizational Interaction: Based on Agents, Founded in Logic*". Tese de D. Sc., Utrecht University, Holanda. 2004.

DING, Y., MALAKA, R., KRAY, C., SCHILLO, M., "*RAJA - A Resource-Adaptive Java Agent Infrastructure*". International Conference on Autonomous Agents, AGENTS'01, 2001.

FALBO, R.A., "I*ntegração de Conhecimento em um Ambiente de Desenvolvimento de Software*", Tese de D. Sc., COPPE/UFRJ, Rio de Janeiro, Brasil, 1998.

FALBO, R.A., ARANTES, D.O., NATALI, A.C.C., "*Integrating Knowledge Management and Groupware in a Software Development Environment*", 5th International Conference on Practical Aspects of Knowledge Management, Vienna, Austria, 2004a.

FALBO, R.A., GUIZZARDI, G., DUARTE, K.C., "*An Ontological Approach to Domain Engineering*". Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering, SEKE'2002, 351-358, Ischia, Italy, 2002.

FALBO, R.A., NATALI, A. C. C., MIAN, P. G., BERTOLLO, G., RUY, F. B., "*ODE: Ontology-based software Development Environment*". IX Congreso Argentino de Ciencias de la Computación, 1124-1135, La Plata, Argentina, Outubro 2003.

FALBO, R.A., PEZZIN, J., SCHWAMBACH, M., "*A Multi-Agent System for Knowledge Delivery in a Software Engineering Environment*", Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering, Taipei, China, p. 253 – 258, 2005a.

FALBO, R.A., RUY, F.B., BERTOLLO, G., TOGNERI, D.F., "*Learning How to Manage Risks Using Organizational Knowledge*". Proceedings of the 6th International Workshop on Advances in Learning Software Organizations, LSO'2004, pp. 7-18, Banff, Canada, June 2004b.

FALBO, R.A., RUY, F.B., DAL MORO, R., "*Using Ontologies to Add Semantics to Software Engineering Environments*". 17th International Conference on Software Engineering and Knowledge Engineering, SEKE'2005, 151-156, Taipei, China, July 2005b.

FININ, T., LABROU, Y., MAYFIELD, J., "*KQML as an agent communication language*", Disponível em http://www.cs.umbc.edu/~finin/papers/. September, 1995.

FIPA, "*The Foundation for Intelligent Physical Agents*". <http://www.fipa.org>. Acesso em: 4 jul. 2008.

FOWLER, M., "*UML Distilled: A Brief Guide to the Standard Object Modeling Language*", Third Edition. Addison-Wesley. 2003.

GAMMA, E., HELM, H., JOHNSON, R., VLISSIDES, J., "*Design Patterns: Elements of Reusable Object-Oriented Software*". Addison-Wesley, 1995.

GARCIA, A., SILVA, V., CHAVEZ, C., LUCENA, C., "*Engineering Multi-Agent Systems with Aspects and Patterns*". Journal of the Brazilian Computer Society – Special issue on Databases / Software Engineering – Número 1, Vol 8, Julho 2002.

GRUBER, T.R.: "*Towards principles for the design of ontologies used for knowledge sharing*". Int. J. Human-Computer Studies, v. 43, n. 5/6 , 1995.

GUARINO, N., "*Formal Ontology and Information Systems*". In N. Guarino, editor, Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy, pages 3-- 15. IOS Press, June 1998.

GUIZZARDI, G., FALBO, R. A., PEREIRA FILHO, J. G.: "*Using Objects and Patterns to Implement Domain Ontologies*". Anais do XV Simpósio Brasileiro de Engenharia de Software, Outubro de 2001.

GUIZZARDI, R.S.S., "*Agent-oriented Constructivist Knowledge Management*". Tese de D. Sc., University of Twente, Enschede, Holanda, 2006.

GUIZZARDI, R.S.S., Inteligência Artificial - Notas de Aula. Universidade Federal do Espírito Santo. 2007.

HARRISON, W., OSSHER, H., TARR, P., "*Software Engineering Tools and Environments: A Roadmap*". Proceedings of the Conference on the Future of Software Engineering - International Conference on Software Engineering, 261-277, Limerick, Ireland, 2000.

HOLZ, H., "*Process-Based Knowledge Management Support for Software Engineering*", Doctoral Dissertation, University of Kaiserslautern, dissertation.de Online-Press, 2003.

HUHNS, M. N., SINGH, M. P., "*Ontologies for Agents*". http://computer.org/internet/ - nov. – dec. 1997.

HUHNS, M. N., STEPHENS, L. M., "*Multiagent Systems and Societies of Agents*". In: WEISS, G., "*Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*", London, The MIT Press, 1999, p. 79-120.

IEEE, "*IEEE Standard for Software Life Cycle Processes – Risk Management*", IEEE Std 1540-2001. 2001.

IGLESIAS, C. A., GARIJO, M., GONZALES, J. C., VELASCO, J. R., "*Analysis and Design of Multiagent Systems Using MAS-CommonKADS*". In Singh, M., Rao, A., and Wooldridge, M., editors, "*Intelligent Agents*" IV, volume 1365 of LNAI, pages 313–326. Springer-Verlag, Berlin, Germany. 1998.

JACK, "*JACK Intelligent Agents*". <http://www.agent-software.com>. Acesso em 23 set. 2004.

JENA. "*Jena – A Semantic Web Framework for Java*". Disponível em: <http://jena.sourceforge.net/>, acesso em: 7 jul. 2008.

JENNINGS, N. R., "*Coordination Techniques for DAI*". In: O'HARE, Greg; JENNINGS, Nicholas (Eds.). "*Foundations of Distributed Artificial Intelligence*". [S.1.]: John Wiley and Sons, 1996. cap.6.

JENNINGS, N. R., WOOLDRIDGE, M., "*Applications of Intelligent Agents*". In: JENNINGS, N. R., WOOLDRIDGE, Michael (Eds.), "*Agent technology: foundations, applications, and markets*". Heidelberg, Germany: Springer-Verlag, 1998. p.3-28.

JEON, H., PETRIE, C., CUTKOSKY, M. R., "*JATLite: A Java Agent Infrastructure with Message Routing*". IEEE Internet Computing, Mar/Apr 2000.

JOHNSON, R. E., "*Components, frameworks, patterns*". ACM SIGSOFT Software Engineering Notes, Proceedings of the 1997 symposium on Symposium on software reusability May 1997. Volume 22 Issue 3. 1997a.

JOHNSON, R. E., "*Frameworks = (components + patterns)*". Communications of the ACM October 1997. Volume 40 Issue 10. 1997b.

JUAN, T., PEARCE, A., STERLING, L., "*ROADMAP: Extending the Gaia Methodology for Complex Open Systems*". Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS'02), pages 3–10, New York, USA. ACM Press. 2002.

JUCHEM, M., "*Projeto de Sistemas Multiagentes em Organizações Empresariais*". Tese de Mestrado. Porto Alegre. Janeiro de 2002.

KENDALL, E. A., KRISHNA, P. V. M., PATHAK, C. V., SURESH, C. B., "*Patterns of Intelligent and Mobile Agents*". Proceedings of the second international conference on Autonomous agents May 1998.

KENDALL, E. A., KRISHNA, P. V. M., SURESH, C. B. , PATHAK, C. V., "*An Application Framework for Intelligent and Mobile Agents*". ACM Computing Surveys, Vol. 32, No. 1es, March 2000.

LIMA, K.V.C., "*Definição e Construção de Ambientes de Desenvolvimento de Software Orientados a Organização*". Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, 2004.

MARTINS, A.F., NARDI, J.C., FALBO, R.A., "*ReqODE: Uma Ferramenta de Apoio à Engenharia de Requisitos Integrada ao Ambiente ODE*", Sessão de Ferramentas do XX Simpósio Brasileiro de Engenharia de Software – SBES´2006, Florianópolis, Brasil, Outubro, 2006.

MCGUINNESS, D.L., HARMELEN, F.V., "*OWL Web Ontology Language Overview*", W3C Recommendation, 10 February 2004, Disponível em: <http://www.w3.org/TR/owl-features> Acesso em: 7 jul. 2008.

NARDI, J. C.; FALBO, R.A., "*Uma Ontologia de Requisitos de Software*". IX Workshop Iberoamericano de Ingeniería de Requisitos y Ambientes de Software, La Plata, Argentina, Abril 2006.

NATALI, A.C.C., FALBO, R.A., "*Gerência de Conhecimento em ODE*", Anais do XVII Simpósio Brasileiro de Engenharia de Software, 270-285, Manaus, Brasil, Outubro, 2003.

NATALI, A.C.C., FALBO R.A., "*Knowledge Management in Software Engineering Environments*", Proc. of the 16th Brazilian Symposium on Software Engineering, Gramado, Brazil, 2002.

NIKRAZ, M., CAIRE, G., BAHRI, P.A., "*A methodology for the analysis and design of multi-agent systems using JADE*". International Journal of Computer Systems Science and Engineering. 2006.

NUNES, B. V., "*Integrando Gerência de Configuração de Software, Documentação e Gerência de Conhecimento em um Ambiente de Desenvolvimento de Software*". Dissertação de Mestrado, UFES, Vitória, Brasil, 2005.

NUNES, V.B., FALBO R.A., "*Uma Ferramenta de Gerência de Configuração Integrada a um Ambiente de Desenvolvimento de Software*". Anais do V Simpósio Brasileiro de Qualidade de Software, p. 231-246, Vila Velha, Brasil, Maio 2006.

ODELL, J., PARUNAK, H. V. D., BAUER, B., "*Extending UML for Agents*". Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National conference on Artificial Intelligence, pages 3–17, Austin, TX, USA. 2000.

ODELL, J., PARUNAK, H. V. D., BAUER, B., "*Representing Agent Interaction Protocols in UML*". In: Agent-Oriented Software Engineering, Ciancarini, P. and Wooldridge, M., Eds., Springer, pp. 121- 140, Berlin, 2001. Disponível em < http://www.fipa.org/docs/input/f-in-00077>. Acesso em: 4 jul. 2008.

O'LEARY, D. E., STUDER, R., "*Knowledge Management: An Interdisciplinary Approach*", IEEE Intelligent Systems, v. 16, n. 1, pp. 24-25, Jan/Feb. 2001.

PADGHAM, L., WINIKOFF, M., "*Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents*". Proceedings of the workshop on Agent-oriented methodologies at OOPSLA'02, Seattle, USA. 2002.

PEZZIN, J., "*AgeODE: Uma Infra-estrutura para Apoiar a Construção de Agentes para Atuarem em um Ambiente de Desenvolvimento de Software*". Dissertação de Mestrado, Mestrado em Informática, UFES, Vitória, Outubro 2004.

PIANISSOLLA, T.L., "*Uso de Serviços Semânticos para Apoiar a Identificação de Recursos Humanos Baseada em Competências*". Projeto de Graduação, Curso de Ciência da Computação, UFES, Vitória, 2007.

PBQP (Programa Brasileiro da Qualidade e Produtividade, Subcomitê Setorial da Qualidade e Produtividade em Software), "*Qualidade e Produtividade no Setor de Software Brasileiro – Pesquisa 2001*". 2002.

POKAHR, A., BRAUBACH, L., LAMERSDORF, W., "*A Flexible BDI Architecture Supporting Extensibility*". In Skowron, A., Barthes, J.-P., Jain, L., Sun, R., Morizet-Mahoudeaux, P., Liu, J. and Zhong, N. (eds), 2005 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-2005), pp. 379–385, IEEE Computer Society, 2005.

PRESSMAN, R.S., "*Engenharia de Software*", Mc Graw Hill, 6a edição. 2006.

PROTÉGÉ. "*Protégé – Ontology Editor and Knowledge Acquisition System*". <http://protege.stanford.edu>. Acesso em: 4 jul. 2008.

RAO, A.S., GEORGEFF, M., "*BDI Agents: from Theory to Practice*". In Proceedings of the 1st International Conference on Multi-Agent Systems, pp. 312–319, San Francisco, CA, 1995.

REIS, C. A. L., REIS, R. Q., SOUZA, A. L. R., "*Interação Humana Durante Execução de Processos de Software: Classificação e Exemplos*". Relatório de Pesquisa, UFRGS, Instituto de Informática, Programa de Pós-graduação em Ciência da Computação - Porto Alegre, Junho/2001.

RETSINA. "*RETSINA*" <http://www-2.cs.cmu.edu/~softagents/index.html>. Acesso em 23 set., 2004.

RUY, F.B., "*Semântica em um Ambiente de Desenvolvimento de Software*". Dissertação de Mestrado, Mestrado em Informática, UFES, Vitória, Maio 2006.

SADEK, M.D., "*Attitudes Mentales et Interaction Rationnelle: Vers une Th´eorie Formelle de la Communication*". Thèse de Doctorat Informatique, Université de Rennes I, France, 1991.

SCHWAMBACH, M. M., "*OplA: Uma Metodologia para Desenvolvimento de Sistemas Orientados a Objetos e Agentes*". Dissertação de Mestrado, Mestrado em Informática, UFES, Vitória, Outubro 2004.

SEARLE, J, "*Speech Acts*", Cambridge, MA, Cambridge University Press, 1969.

SEGHROUCHNI, A. F.. "*Rational Agent Cooperation through Concurrent Plan Coordination*". Proceedings of the Iberoamerican Workshop on Artificial Intelligence and Multi-Agent Systems, 1996, Mexico. Anais. Lania, Mia Universidad Veracruzana, 1996. p.162-171.

SILVA, F. A. D., "*Um Modelo de Simulação de Processos de Software Baseado em Conhecimento para o Ambiente PROSOFT*". Dissertação de Mestrado. UFRGS. Porto Alegre, RS. Janeiro de 2001.

SOMMERVILLE, I., "*Engenharia de Software*", Addison-Wesley, 6ª edição. 2003.

SYCARA, K., PAOLUCCI, M., VAN VELSEN, M., GIAMPAPA,. J.: The RETSINA MAS Infrastructure. in the special joint issue of Autonomous Agents and MAS, Volume 7, Nos. 1 and 2, July, 2003. Disponível em <http://www-2.cs.cmu.edu/~softagents/publications.html>. Acesso em: 4 jul. 2008.

TRAVASSOS, G.H., "*O Modelo de Integração de Ferramentas da Estação TABA*". Tese de Doutorado, COPPE/UFRJ, Rio de Janeiro, 1994.

WAGNER, G., "*The Agent-Object-Relationship Meta-Model: Towards a Unified View of State and Behavior*". Information Systems, 28(5):475–504. 2003.

WILLMOTT, S., DALE, J., BURG, B., CHARLTON, P., O'BRIEN, P., "*Agentcities: A Worldwide Open Agent Network*". AgentLink News (8). pp. 13-15. 2001.

WOOLDRIDGE, M., "*Intelligent Agents*". In: WEISS, G., "*Multiagent Systems - A Modern Approach to Distributed Artificial Intelligence*". London, The MIT Press, April 1999, p. 27-77.

WOOLDRIDGE, M., JENNINGS, N. R., KINNY, D., "*The Gaia Methodology for Agent-Oriented Analysis and Design*", Journal of Autonomous Agents and Multi-Agent Systems, 2000.

ZAMBONELLI, F., JENNINGS, N. R., WOOLDRIDGE, M.. "*Organizational Abstractions for the Analysis and Design of Multi-Agent Systems*". Proceedings of the 1st International

Workshop on Agent-Oriented Software Engineering, 2000, Limerick, Ireland. Anais. 2000. P.127-141.