DIEGO AZEVEDO CÉSAR

INCORPORAÇÃO DE PROPRIEDADES TRANSVERSAIS EM SISTEMAS MULTIAGENTES UTILIZANDO PROGRAMAÇÃO ORIENTADA A ASPECTOS

Dissertação apresentada à Universidade Federal de Viçosa, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, para obtenção do título de *Magister Scientiae*.

VIÇOSA
MINAS GERAIS - BRASIL
2008

Aos meus pais José Geraldo e Maria das Graças,

ao meu orientador Alcione de Paiva Oliveira,

aos amigos e professores do Departamento de Informática - DPI - UFV.

AGRADECIMENTOS

- Agradeço primeiramente a Deus, sábio mentor e fiel companheiro, pela força e sabedoria para vencer cada novo desafio.

- Aos meus pais, José Geraldo e Maria das Graças, por todo amor e ensinamentos que sempre me deram. O apoio incondicional foi de grande importância para o término dessa etapa da caminhada. Obrigado por fazerem dos meus sonhos, seus sonhos, e da minha vida, suas vidas.

- Ao meu irmão caçula, Dione, que mesmo a distância, trouxe incentivo, e compartilhou de momentos de alegria e dificuldades.

- Em especial, ao meu orientador, Professor Alcione de Paiva Oliveira, pelos ensinamentos, paciência, atenção e disponibilidade em sempre ajudar.

- Aos professores do Departamento de Informática - DPI, que contribuíram com a minha formação acadêmica e amadurecimento pessoal.

- Aos funcionários do DPI, em especial, ao secretário da pós-graduação, Altino Alves de Souza Filho, pela grande competência e dedicação; à chefe de expediente, Eliana Ferreira Rocha, pela presteza e amizade.

- À minha grande amiga Ariana Priscila da Silva e sua família; Pelo acolhimento e convivência fraterna. A eles dedico, com carinho especial, as conquistas alcançadas nessa fase.

- Aos meus amigos Vítor Vita Martins e Carina Cristina Teixeira pela segura amizade e suporte emocional durante o desenvolvimento desse trabalho; Sou grato a eles de uma forma incomparável.

- Aos colegas de mestrado, pelo apoio, incentivo e contribuições acadêmicas. Em especial, à minha amiga Deisymar Botega Tavares, pela grande atenção e confiança em sempre ajudar.

- A todos que, de algum modo, contribuíram para a conclusão dessa etapa da minha vida. Tenho consciência de que, se cheguei até aqui, é porque tive a oportunidade crescer e aprender ao lado de pessoas muito importantes.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE ABREVIAÇÕES

ACL	Agent Comunication Language
AID	Agent Identifier
AJDT	AspectJ Development Tools
AMS	Agent Management System
API	Application Programming Interface
DF	*Directory Facilitator*
DPI	*Dots per Inch*
EPL	*Eclipse Public License*
FIPA	*Foundation for Intelligent Physical Agents*
IA	Inteligência Artificial
IAD	Inteligência Artificial Distribuída
IDE	*Integrated Development Environment*
JADE	*Java Agent DEvelopment framework*
JDT	*Java Developer Toolkit*
JESS	*Java Expert System Shell*
MLP	*Multi-Layer Perceptron*
OA	Orientação a Aspectos
PCA	*Principal Components Analysis*
PDE	*Plugin Development Environment*
POA	Programação Orientada a Aspectos
RNA	Redes Neurais Artificiais
SMA	Sistema Multiagente
SWT	*Standard Widget Toolkit*
UML	*Unified Modeling Language*
XML	*EXtensible Markup Language*
XSL	*Extensible Stylesheet Language*

RESUMO

CÉSAR, Diego Azevedo, M.Sc., Universidade Federal de Viçosa, outubro de 2008. **Incorporação de propriedades transversais em Sistemas Multiagentes utilizando Programação orientada a Aspectos**. Orientador: Alcione de Paiva Oliveira. Co-orientadores: José Luis Braga e Vladimir Di Iorio.

O desenvolvimento de Sistemas Multiagentes (SMAs) é uma tarefa complexa pois envolve a especificação e implementação de diversas propriedades relacionadas com os agentes, tais como interação, adaptação, autonomia, dentre outros. Diversas dessas propriedades podem ser consideradas transversais, ou seja, encontram-se espalhadas por todo o código do programa, se misturando com as funcionalidades básicas dos agentes e são difíceis de serem representadas de forma modular. Em geral, esses interesses transversais afetam diversos módulos e não podem ser isoladas explicitamente por técnicas baseadas em abstrações orientadas a objetos. Além disso, as linguagens orientadas a aspectos usualmente tem uma sintaxe complexa e conceitos de difícil entendimento para os desenvolvedores de *software*. Este trabalho apresenta um estudo da utilização de Programação Orientada a Aspectos (POA) para promover o isolamento dos comportamentos inteligentes dos agentes como módulos do sistema. É proposta uma ferramenta denominada CM2JADE, que auxilia o desenvolvimento dos Sistemas Multiagentes com base em abstrações orientadas a aspectos. Com a utilização da ferramenta, as propriedades dos agentes são capturadas e mantidas separadas do corpo do agente, modelado como objeto, e suas propriedades modeladas como relacionamentos aspectuais. O CM2JADE apresenta diversos benefícios na geração de código e modelagem das propriedades transversais dos agentes na fase de desenvolvimento. É apresentado um estudo de caso com os resultados e comentários da

aplicação da programação orientada a aspectos em Sistemas Multiagentes utilizando a ferramenta proposta.

ABSTRACT

CÉSAR, Diego Azevedo, M.Sc., Universidade Federal de Viçosa, october of 2008. **Crosscutting requirements in Multi-agent Systems using Aspect-Oriented Programming**. Adviser: Alcione de Paiva Oliveira. Co-Advisers: José Luis Braga and Vladimir Di Iorio.

The development of Multi-Agent Systems (MASs) involves special concerns, such as interaction, adaptation, autonomy, among others. Many of these concerns are overlapping, crosscut each other and the agents basic functionality and cannot be represented in a modular fashion. In general, they inherently affect several system modules and cannot be explicitly captured based on existing software engineering abstractions. Therefore, aspect oriented languages usually have complex syntax and complex concepts which make them difficult to become popular for developers. This work presents a study on the use of Aspect Oriented Programming (AOP) to isolate the agent intelligent behaviour like completely separated module. In order to tackle these problems, we propose a tool called CM2JADE, which basically provides assistance for the development of multi-agent systems based on aspect oriented abstractions. As the tool is targeted to aspect-based development, agency properties are captured and kept separated from object functionality by means of aspectual relationships. The generative approach brings several benefits to the code generation and modeling of agent crosscutting features in development stage. A sample case study is presented with the results and comments about the application of the aspect-based programming to multi-agent systems using CM2JADE.

Capítulo 1

Introdução

"*The standard definition of AI is that which we don't understand*"(Bill Joy).

A área de Sistemas Multiagentes (SMA), sub-área da Inteligência Artificial (IA), estuda a coletividade ao invés do indivíduo e emprega noções de sociedade e de comportamento social de outras áreas, como a Sociologia e a Psicologia. O comportamento social apresentado pelos agentes, a troca de informações entre eles, bem como a associação para alcançar objetivos comuns é a base para a inteligência desse tipo de sistema.

O estudo de agentes tornou-se importante na IA e em muitos campos da Ciência da Computação em função de permitir a apresentação de um comportamento complexo a partir da interação de unidades básicas (WOOLDRIDGE; JENNINGS, 1995). Estes agentes também são investigados em diversas áreas, tais como Economia, Filosofia, Lógica, Ecologia e Ciências Sociais (WOOLDRIDGE, 1999). Na engenharia de *software*, a utilização de SMAs pode representar uma decisão de projeto para abordar problemas complexos, de grande porte ou que suportem execução em ambiente distribuído. Apresentam contribuições, ainda, para o entendimento de sociedades, por meio de simulações ou predições de comportamento humano. Assim, ajudam a definir e a melhorar processos que envolvem a interação de entidades inteligentes.

Os agentes podem colaborar ou competir com outros agentes, formando Sis-

temas Multiagentes. A exemplo das sociedades humanas, ora competitivas e ora cooperativas, um agente com comportamento definido pode trazer vários benefícios, através da interação com os demais, para tomada de decisão em problemas com múltiplos objetivos, mesmos que estes sejam conflitantes. Isso ocorre porque cada agente é capaz de buscar a satisfação de um objetivo particular para o qual foi projetado ou abandonar, temporariamente, esse objetivo em função de alcançar um objetivo sistêmico.

Os Sistemas Multiagentes estão fortemente relacionados com o campo da IA pois estes desempenham um comportamento global inteligente alcançado a partir do comportamento individual dos agentes (WOOLDRIDGE, 1999). Por apresentarem atividades sociais como cooperação e competição, que requerem um certo grau de deliberação, controle de objetivos múltiplos e comportamento inteligente, esses tipos de sistemas, suas características e aplicações são objetos de investigação da IA, que é uma área que tem experimentado um ressurgimento nos últimos anos.

A construção de um Sistema Multiagente (SMA) é complexa por envolver um grande número de características intrínsecas tais como seus interesses, metas, objetivos, e outros requisitos extrínsecos que tratam a forma de interação entre esses agentes, seus grupos e sociedades. Em geral, essas características interagem entre si de forma diferente para cada problema abordado e também variam segundo as atividades desempenhadas pelos agentes. Outro motivo que torna difícil o projeto desse tipo de sistema é o comportamento não-determinístico e autônomo dos agentes (CHEN; SADAOUI, 2005).

No sentido de diminuir a dificuldade de construção de sistemas complexos, a engenharia de *software* (BOEHM, 2002) propõe a divisão do problema em módulos, reduzindo o acoplamento entre as partes do sistema. Dessa forma, a maximização da modularidade do sistema e da independência entre os pacotes torna mais legível e compreensível o Sistema, além de facilitar sua manutenção (PRESSMAN, 2001).

A modularização de sistemas é muito importante em sistemas de informação e algumas soluções tais como utilização de Interfaces de Programação de Aplicações

(do inglês *Application Programming Interface* - APIs) orientadas a objetos, padrões de projeto (GAMMA et al., 1995) e extração de Aspectos (KICZALES et al., 1997) são empregadas para minimizar os efeitos indesejáveis de espalhamento e intrusão de código em módulos diversos destes sistemas. Diversos ambientes de desenvolvimento de *software* suportam a utilização dessas soluções e auxiliam os desenvolvedores na construção de sistemas complexos a partir delas.

A aplicação de técnicas de modularização dos elementos de um sistema de *software* desempenham um papel importante para entendimento do mesmo. Existem requisitos, denominados requisitos transversais, que geralmente ficam entrelaçados com outras funcionalidades, em diversas partes do sistema, e são mais difíceis de modularizar diante do paradigma de programação orientada a objetos. Em Sistemas Multiagentes, a utilização destas metodologias de essa modularização são bem-vindas porque o número de requisitos transversais é muito grande. Um simples agente inclui múltiplas propriedades transversais, incluindo autonomia, interação, adaptação, colaboração, aprendizagem e mobilidade (GARCIA; LUCENA, 2007). Os interesses transversais não podem ser especificados somente a partir de abstrações e mecanismos de decomposição da orientação a objetos (UBAYASHI; TAMAI, 2001a; GARCIA et al., 2004). Uma alternativa para modularizar requisitos transversais é a abstração orientada a Aspectos.

A Orientação a Aspectos (OA) (KICZALES et al., 1997) é uma das abordagens para a separação avançada de interesses. Aspectos são usados como abstrações capazes de capturar de forma modular os interesses transversais nos sistemas. Existem diversas implementações do conceito de aspectos, dentre elas se destaca *AspectJ* (KICZALES et al., 2001b). Essa implementação é uma extensão orientada a aspectos da linguagem de programação Java. *AspectJ* é uma linguagem orientada a aspectos que possui uma grande quantidade de recursos para declarar, com clareza, os pontos do código que serão afetados pelo aspecto e as ações que deseja-se executar nesses pontos.

Porém existe uma certa inércia quanto ao aprendizado de técnicas que modularizam requisitos tranversos em Sistemas de computação. Essa inércia pode ser

atribuída à sintaxe não-usual de linguagens de programação que se propõem a modularizar requisitos transversais. Outra dificuldade existente diz respeito às ferramentas de construção de aspectos integradas aos ambientes de desenvolvimento de *softwares*. Essa ferramentas geralmente não disponibilizam recursos gráficos para que usuários especifiquem de forma fácil os pontos de código que desejam afetar, as ações a serem executadas nestes pontos e consequentemente os interesses de software que deseja-se tratar de forma modular.

A separação de interesses é um ponto central no desenvolvimento de sistemas de *software* baseados em agentes porque os requisitos transversais impactam diretamente no modelo dos agentes (UBAYASHI; TAMAI, 2001a; GARCIA et al., 2004). A modelagem de elementos internos dos agentes, tais como seus objetivos, metas e planos recebem influência de diversos outros interesses/módulos do sistema. Caso haja uma modularização correta dos requisitos que influenciam nestes e em outros pontos do agente, a compreensão, adaptação e manutenção do sistema torna-se mais fácil e o ciclo de vida do sistema pode ser prolongado.

1.1 O problema e sua importância

Existem algumas dificuldades na construção de Sistemas Multiagentes. Do ponto de vista do sistema, a criação de mecanismos de controle genéricos para a coordenação dos diversos agentes apresenta-se como um grande desafio. Na outra extremidade do modelo, um outro problema é a especificação do próprio agente, que envolve a representação do seu conhecimento sobre o meio e sobre os demais agentes, de seus objetivos e de como será possível atingí-los.

Um sistema baseado em agentes apresenta um número considerável de requisitos não-ortogonais, ou seja, módulos do sistema interagem entre si. do agente, suas atividades sociais, objetivos, metas e planos de ação. Estes módulos representam as características Além disso, o relacionamento entre as propriedades de um agente dependem da complexidade do mesmo e das características do ambiente no qual o

agente é projetado para atuar. A metodologia de construção de um SMA influencia diretamente no entendimento do mesmo e na produtividade de desenvolvimento.

A construção de Sistemas multiagentes utilizando a abordagem orientada a objetos é de modelagem complexa, de difícil entendimento e, geralmente, produz um código que dificulta a manutenção desse tipo de sistema. Além disso, geralmente, a orientação a objetos não evita, nesse tipo de sistema, o espalhamento das funções transversais por diversos módulos e objetos distintos. No caso de agentes de *software*, esse espalhamento ocasiona problemas na eficiência e na manutenibilidade.

Os Sistemas Multiagentes desenvolvidos com técnicas de orientação a objetos apresentam efeitos indesejados de intrusão e espalhamento de código. Isso significa que existem fragmentos de código indevidamente espalhados por módulos do sistema destinados a tratar de outros interesses específicos. Nesse sentido, pode ser uma decisão de projeto, que visa minimizar o espalhamento de código, a utilização da programação orientada a aspectos.

Apesar das abstração orientada a aspectos ter sido concebida para ser utilizada em conjunto com a programação orientada a objetos, os desenvolvedores das linguagens de *software* orientado a objetos não estão habituados com a sintaxe das linguagens que implementam o conceito de Aspectos. O aprendizado de uma nova linguagem pode apresentar uma longa curva de aprendizado e acarretar em gastos com treinamento pessoal em ambientes corporativos (GAZOLLA, 2008).

Este trabalho se propõe a responder a seguinte questão: que contribuição(ões) a programação orientada à aspectos pode trazer para a implementação de Sistemas Multiagentes em que as funcionalidades básicas dos agentes estejam entrelaçadas com código referente ao tratamento dos interesses transversais extrínsecos e intrínsecos do agente?

1.2 Hipótese

A abstração orientada a aspectos contribui para a modularização de requisitos intrínsecos e extrínsecos de sistemas baseados em agentes e, por conseqüência, favorece o entendimento e a manutenção desse tipo de sistema.

1.3 Objetivos

O objetivo geral do projeto é obter um mecanismo para adicionar propriedades transversais a uma ou mais classes de agentes de um Sistema Multiagente, utilizando recursos de programação orientada a aspectos de forma fácil e transparente para o usuário. Especificamente, pretende-se:

- Construir uma interface gráfica de inserção de instruções transversais em Sistemas Multiagentes.
- Possibilitar a integração do mecanismo a um ambiente de programação que suporte a construção de *software* e compatibilidade com um *framework* de desenvolvimento de Sistemas Multiagentes.
- Disponibilizar ao projetista do sistema multiagente, usuário da ferramenta proposta, recursos para estudar o comportamento do sistema antes e após a inserção das propriedades do agente.
- Apresentar um estudo de caso do uso da ferramenta para modificar o comportamento de um Sistema Multiagente específico, bem como os benefícios alcançados com a utilização desta.

1.4 Organização do Texto

Neste capítulo abordam-se alguns conceitos iniciais de Sistemas Multiagentes e Programação Orientada a Aspectos. O uso de Aspectos nesse tipo de sistema é o principal

foco deste trabalho. A motivação desta iniciativa é facilitar a adição de propriedades transversais em sistemas baseados em agentes, mesmo que essas propriedades sejam características desse mesmo agente ou desse grupo de agentes.

Este trabalho está organizado em 6 capítulos, a saber: No Capítulo 2 é apresentada a revisão bibliográfica, na qual são descritos os Sistemas Multiagentes, bem como fundamentos da abstração orientada a aspectos; o Capítulo 3 descreve a metodologia de concepção do mecanismo proposto para adição de propriedades transversais em sistemas baseados em agentes; o Capítulo 4 apresenta a ferramenta construída, seus benefícios e limitações; o Capítulo 5 contém um estudo de caso com aplicação do mecanismo em um Sistema Multiagente; o Capítulo 6 encerra o trabalho apresentando as conclusões finais e direções para trabalhos futuros.

Capítulo 2

Aspectos teóricos e tecnológicos

"*Inside every large progam there is a small program trying to get out*"(Tony Hoare).

"Inteligência Artificial (IA) é a área da ciência da computação orientada ao entendimento, construção e validação de sistemas ou entidades inteligentes, isto é, que exibem, de alguma forma, características associadas ao que chamamos inteligência"(RICH; KNIGHT, 1990). A IA esforça-se para construir e entender essas entidades inteligentes, estejam elas desempenhando alguma atividade racional ou baseada em procedimentos humanos (RUSSELL; NORVIG, 2002).

Existem duas abordagens dos pesquisadores aos estudos de Inteligência Artificial. A primeira visa a construção, estudo e análise de sistemas computacionais que agem racionalmente, tomam decisões baseadas em regras, fatos e experiência. A segunda abordagem considera que entidades inteligentes se baseiam na performance humana para realizar alguma tarefa.

A IA é também organizada em sub-campos de pesquisa, como Inteligência Artificial Simbólica, Evolucionária, Conexionista, dentre outras. Porém, a abordagem utilizada neste trabalho é a Inteligência Artificial Distribuída, que propõe a resolução de problemas estudados em quaisquer outros ramos da IA fisicamente distribuída. Essa abordagem visa a construção de sistemas de entidades inteligentes que interagem produtivamente com outras para resolver problemas. Na Inteligência Artificial

Distribuída (IAD), são estudados os sistemas baseados em agentes, coordenação, estruturas e linguagens para agentes inteligentes.

2.1 Agentes inteligentes e Sistemas Multiagentes

Os agentes inteligentes artificiais estão presentes como área de estudo e também aplicações práticas de na resolução dos problemas reais da indústria, na robótica e engenharia (JENNINGS, 1994; ZAKI et al., 2007; PARUNAK, 1987; NEAGU et al., 2006). Segundo JENNINGS; WOOLDRIDGE (1998) algumas razões para o crescimento desse interesse em pesquisas com Sistemas Multiagentes são:

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

Essas características incentivam a utilização de sistemas baseados em agentes em diversas áreas. Porém, existe uma questão polêmica no campo de estudo da Inteligência Artificial, que há anos é discutida, é a definição de "agente". Huhns e Singh (1998) apresentam a seguinte discussão sobre o tema:

> Existem dois pontos-de-vistas em extremos opostos sobre os agentes. Uma bem estabelecida tradição considera os agentes como entidades essencialmente cognitivas e conscientes, que possuem sentimentos, percepção e emoção exatamente como os seres humanos. Sob este ponto-de-vista, todo o trabalho computacional feito sobre agentes atualmente é inerentemente inadequado. O outro ponto-de-vista assume que agentes são meramente autômatos e se comportam exatamente como foram projetados ou programados.

> Este ponto de vista admite uma grande variedade de sistemas computacionais, incluindo agentes computacionais. Uma preocupação é que ele pode ser muito permissivo (HUHNS; SINGH, 1998) (Tradução do autor).

Neste sentido, existem inúmeras definições para "agentes". FRANKLIN; GRAESSER (1997) apresentam uma coleção dessas definições, das quais as mais relevantes estão resumidas abaixo.

- "Um agente é qualquer coisa que pode ser vista como percebendo seu ambiente através de sensores e agindo sobre este ambiente através de efetuadores."(RUSSELL; NORVIG, 2002);
- "Agentes Autônomos são sistemas computacionais que habitam algum ambiente dinâmico e complexo, percebem e atuam autonomamente neste ambiente e, fazendo isto, atingem um conjunto de objetivos ou tarefas para os quais foram projetados."(MAES, 1994);
- "Um agente é definido como uma entidade de *software* persistente dedicada a um propósito específico."(SMITH; CYPHER; SPOHRER, 1994); "Agentes inteligentes realizam continuamente três funções: percebem as condições dinâmicas em um ambiente; agem para afetar as condições do ambiente; e raciocinam para interpretar as percepções, resolver problemas, realizar inferências e determinar ações."(HAYES-ROTH; BROWNSTON; SINCOFF, 1995);
- "... um sistema computacional baseado em *hardware* ou (mais habitualmente) em *software* que possui as seguintes propriedades: autonomia, habilidade social, reatividade e pró-atividade."(WOOLDRIDGE; JENNINGS, 1995)
- "Agentes Autônomos são sistemas capazes de ações autônomas e propositadas no mundo real."(FRANKLIN, 1995).

Finalmente, RUSSELL; NORVIG (2002) afirmam que "a noção de um agente deve servir como uma ferramenta para a análise de sistemas e não uma caracterização absoluta que divide o mundo em duas categorias: a dos agentes e a dos não agentes"(RUSSELL; NORVIG, 2002).

Conclui-se essa discussão com uma definição que contém alguns conceitos mais habitualmente aceitos: "Agentes são componentes (de *software*) ativos e persistentes que percebem o mundo, raciocinam, agem e se comunicam"(HUHNS, 2003) (HUHNS, 1998). Considera-se que um agente é inteligente se ele é capaz de apresentar autonomia e reúne três características de flexibilidade (WOOLDRIDGE, 1999):

- Reatividade: agentes inteligentes estão aptos a perceber seus ambientes, e responder no tempo correto para mudá-los para satisfazer seus objetivos.
- Autonomia: agentes inteligentes estão aptos à demonstrar comportamento direcionado à metas e ter iniciativas para satisfazer seus objetivos. A autonomia normalmente significa que um agente tem controle sobre suas ações e pode agir de forma independente de outros. Em outras palavras, o requisito de autonomia realiza o gerenciamento de objetivos de agente. Para ser autônomo, o agente deve ser capaz de instanciar e alcançar seus objetivos.
- Habilidade social: agentes inteligentes são capazes de interagir com outros agentes (e possivelmente humanos) para satisfazer seus objetivos. Para que haja interação entre entidades autônomas deve existir um protocolo de comunicação e uma linguagem de comum acordo. Deve existir um compromisso entre a autonomia dos agentes e a socialização dos mesmos com a comunidade para que os objetivos sistêmicos sejam alcançados. Se um agente rejeita todas as solicitações de serviço encaminhadas a ele em função dos seus objetivos individuais o ambiente não pode ser caracterizado cooperativo e as metas globais podem ser prejudicadas.

Para agentes membros de Sistemas Multiagentes, características como sociabilidade, autonomia social e interações são bem-vindas ou necessárias. Assim, normalmente os agentes são vistos como autônomos e tipicamente heterogêneos.

Um Sistemas Multiagentes é um sistema computacional criado a partir de entidades autônomas (agentes) que interagem entre si e com o meio ambiente no qual

estão inseridos. Cada agente possui características e capacidades específicas, além de objetivos próprios, o que torna a interação essencial para que um agente satisfaça seus objetivos individuais e sistêmicos. Para tornar isso possível, mecanismos de interação e coordenação devem ser implementados.

Ao especificar como deve funcionar um agente ou um grupo de agentes, deve-se planejar como serão baseadas suas decisões, metas, desejos e comunicação. As decisões de um agente podem ser baseadas em um conjunto de regras de decisão que especificam sua forma de raciocínio. Suas metas e desejos variam de acordo com o domínio do problema e por isso necessitam ser estudadas para cada problema em particular.

Um sistema baseado em agentes é constituído de diversas propriedades importantes para que essas entidades sejam consideradas inteligentes. Essas propriedades podem ser intrínsecas como autonomia, adaptação, interação, mobilidade e colaboração ou extrínsecas no nível do sistema como a dependência, persistência e levantamento de exceção (GARCIA; LUCENA; COWAN, 2004).

Uma entidade inteligente age racionalmente e intencionalmente com respeito aos seus próprios objetivos e o estado do seu conhecimento em um determinado momento (DEMAZEAU; MÜLLER, 1990). Esses autores propuseram um agente genérico que deve possuir no mínimo algumas características para interagir em um mundo Multiagente. Para um Sistema multi-agente algumas características são necessárias (DEMAZEAU; MÜLLER, 1990):

- conhecimento: é a representação do mundo e do problema que o agente tem que solucionar. Pode ser implícito ou explícito, inato ou aprendido através de comunicação e percepção;
- objetivos: como o conhecimento, pode ser implícito, explícito, inato ou aprendido;
- capacidades de raciocínio: que podem derivar soluções possíveis para o problema do agente. O forma de raciocínio dos agentes pode ser associada à verificação

de regras ou baseada em casos (LEAKE; PLAZA, 1997).

- capacidades de decisão: para definir uma escolha entre as soluções possíveis para o problema do agente. A tomada de decisão dos agentes pode ser auxiliada por uma máquina de inferência que atua sobre a base de conhecimento do agente e derivar novas sentenças ou um conjunto de ações mais indicadas para o estado atual do ambiente.

A forma de estruturação dessas quatro características no projeto de um agente influenciam diretamente na complexidade do SMA. Também deve-se considerar como variáveis de influência na complexidade de um sistema baseado em agentes a sua estrutura organizacional, a forma pela qual seus comportamentos são estruturados, e o conjunto de sentenças que expressa suas crenças, desejos e intenções.

A maneira pela qual é feita a tomada de decisão, em um Sistema Multiagente, depende das regras de comportamento definidas para a sociedade e para o agente em uma implementação específica e pode ser, por exemplo, o resultado de uma competição entre agentes. Desse modo, as regras de comportamento definem como um ou mais agentes terão acesso a alocação dos recursos.

O projetista de um sistema, após definir o domínio da aplicação e os seus propósitos, necessita implementar os agentes autônomos e definir os outros aspectos do sistema que definem a sociedade, como a estrutura de autoridade, as regras de comportamento e uma linguagem de comunicação.

Questões referentes ao modo de programação, decomposição dos agentes em relação ao problema abordado são pertinentes durante o projeto de um grupo de agentes e podem variar de complexidade de acordo com a natureza do problema abordado.

A comunicação entre os agentes e detalhes referentes a interação dos mesmos requerem protocolos claros e bem definidos para que não haja interpretação indevida do conteúdo das mensagens envidadas por um agente. Essas linguagens de comunicação, em geral, devem tratar de questões como identificação dos agentes emissores e

receptores, conteúdo da mensagem e performativas de comunicação.

O mecanismo de construção dos agentes deve permitir que agentes individuais atuem e raciocinem sobre ações, planos e conhecimento sobre outros agentes com a finalidade de coordená-los. Além de proporcionar a cada agente a capacidade de raciocínio individual, os agentes devem ser capazes de identificar e reconciliar pontos de vistas muito diferentes e intenções conflitantes em uma coleção de agentes que tentam coordenar suas ações através de mensagens de solicitações de serviços independente da localização dos agentes.

A programação de agentes móveis é uma tarefa que requer, para garantir o envio de mensagens para o destinatário correto, de um recurso de localização do ambiente de execução do agente. Ou seja, como equilibrar efetivamente a computação local e a comunicação em uma solução genérica para construção de Sistemas Multiagentes?

Para gerenciar a interação entre os agentes, a sociedade é controlada por uma política que envolve regras de comportamento e uma estrutura de autoridade. Esta política possibilita aos agentes decidir qual deles deve possuir o controle de um recurso em um determinado momento.

Os comportamentos que agentes podem exibir em um Sistema Multiagentes são definidos por (DEMAZEAU; MÜLLER, 1990):

- coabitação: um agente precisa atingir um objetivo e consegue fazer isso sozinho;
- cooperação: um agente precisa atingir um objetivo, mas não consegue fazer isso sozinho, tendo que cooperar com outros agentes;
- colaboração: alguns agentes podem alcançar sozinhos objetivos que dizem respeito a todos os agentes, e apenas um agente é escolhido para realizar as tarefas necessárias;
- distribuição: somente vários agentes trabalhando juntos conseguem atingir alguns objetivos comuns.

A estruturação dos comportamentos dos agentes dependem do nível de complexidade das tarefas para as quais o agente foi projetado para realizar. Em (RUSSELL; NORVIG, 2002) são apresentados cinco níveis de complexidade comportamental para agentes artificiais. Dentre eles se encontram os agentes reativos simples, baseados em modelos, em objetivos, os agentes orientados por utilidade e por fim os agentes com aprendizado. Tais níveis de agentes diferenciam em complexidades de comportamento.

O primeiro tipo de agente, e também o mais simples, é o agente reativo. Esse tipo de agente seleciona ações baseado apenas na percepção atual. Ou seja, o histórico de percepções não é componente ativo na tomada de decisão. Nesses sistemas, os agentes não armazenam representações sobre o meio ambiente, sobre outros agentes ou sobre suas próprias ações passadas. O comportamento de cada agente pode ser implementado como um autômato finito simples, com regras que mapeiam dados obtidos do ambiente diretamente em ações (estímulo $\longrightarrow$ resposta).

A proposta dos sistemas que empregam apenas agentes reativos é obter um comportamento inteligente a partir da interação de um grande número de agentes simples. Esse enfoque possui grande influência da entomologia [1]. As principais características deste tipo de agente e dos sistemas onde eles se encontram são destacadas a seguir e podem ser encontradas mais detalhadamente em (BOND; GASSER, 1991).

- Não há representação explícita do conhecimento: o conhecimento dos agentes é implícito (as suas regras de comportamento) e sua manifestação se externa através do seu comportamento em interação com os demais agentes;
- Não há representação interna do ambiente: o comportamento (resposta) de cada agente é baseado no que ele percebe (estímulo) a cada instante. Não há uma representação interna explícita do ambiente;
- Não há memória das ações: os agentes reativos não mantém nenhum tipo de

[1] Entomologia é a ciência que estuda os insetos. Os SMAs reativos procuram reproduzir o comportamento observado em colônias de insetos onde, apesar de os indíviduos possuírem capacidades limitadas, a sociedade apresenta um comportamento bastante complexo.

histórico de suas ações, ou seja, o resultado de uma determinada ação passada não influencia diretamente na decisão de uma ação futura;

- Organização etológica: a forma de organização dos SMAs reativos é similar a observada por animais que vivem em grandes comunidades;
- Grande número de membros: em geral, os SMAs reativos possuem um grande número de agentes, com populações que podem chegar a ordem de milhares de membros.

Um agente baseado em modelos mantém um tipo de estado interno que permite a percepção de em que parte do mundo externo o agente habita, mesmo que esse não conheça todo o ambiente *a priori*.

Dado que conhecer o estado atual nem sempre é suficiente para a tomada de decisão, o agente baseado em objetivos tem um mecanismo de armazenamento de informações das situações desejáveis. Um escalonamento de objetivos que selecione o objetivo mais importante também é necessário caso os objetivos sejam conflitantes.

Os agentes baseados em utilidade têm uma função que mapeia um estado (ou uma seqüência de estados) em um número real, que descreve o grau de satisfação associado.

O último nível de agente, e também o mais complexo, é o que provê mecanismo de aprendizado. Para aperfeiçoamento de suas ações, ele tem uma função de desempenho que permite a percepção e análise das ações anteriores bem ou mal sucedidas e seleção das ações melhor empregadas. Os sistemas cognitivos, por sua vez, normalmente empregam poucos agentes, que possuem uma complexidade considerável e apresentam as características mencionadas anteriormente. Uma característica importante de um agente cognitivo é sua capacidade de tomada de decisão.

Em geral, os SMAs cognitivos são baseados em modelos de organização social, de sociedades humanas: grupos, hierarquias, mercados, etc. Esses Agentes Cognitivos, segundo (BOND; GASSER, 1991), possuem uma representação explícita do ambiente

e dos membros da comunidade e podem raciocinar sobre as ações tomadas no passado e planejar as futuras ações. Os Agentes Cognitivos podem ainda interagir com os demais membros da comunidade através de linguagens e protocolos de comunicação complexos, estratégias sofisticadas de negociação.

Esse tipo de agente é o mais complexo da escala de agentes definida em (RUSSELL; NORVIG, 2002) e deve prover de uma representação explícita do ambiente e dos outros agentes da sociedade. Caso o ambiente seja mutável, o agente deve perceber as mudanças a partir da propriedade de aprendizado.

Além da representação do ambiente, esse tipo de agente pode manter um histórico das interações e ações passadas e, graças a esta memória, são capazes de planejar suas ações futuras. As decisões baseadas em histórico de ações permitem ao agente não escalonar uma ação não-favorável, para um determinado objetivo, presente no histórico do agente.

Agentes cognitivos provêm de mecanismo de controle deliberativo. Os agentes cognitivos raciocinam e decidem em conjunto sobre quais ações devem ser executadas, que planos seguir e que objetivos devem alcançar. Uma análise de prioridade deve ser realizada em casode objetivos conflitantes.

Por fim, os modelos de organização dos SMA cognitivos, em geral, são modelos sociológicos, como, por exemplo, as organizações humanas e contêm, usualmente, poucos agentes.

De forma geral, pode-se dizer que os agentes cognitivos, diferentemente dos reativos, possuem uma representação explícita e detalhada, além de possuir um histórico no qual ele se baseia para tomar suas decisões. Por outro lado, os agentes reativos, diferentemente dos cognitivos, possuem uma comunicação indireta e um controle não deliberativo, fazendo com que seja necessário uma quantidade considerável de agentes no sistema para produzir, de forma emergente, um comportamento complexo.

À medida que o nível de agentes é incrementado, a complexidade do sistema baseado nesses mesmos agentes também cresce. O agente com aprendizagem provê

um módulo no sistema que, muitas das vezes, acrescenta ao sistema um número considerável de linhas de código que atuam diretamente com muitos módulos do sistema. Isso dificulta o entendimento do SMA e também aumenta o acoplamento dos módulos deste mesmo sistema.

As linguagens de programação de Sistemas Multiagentes, as plataformas e ferramentas de desenvolvimento são componentes importantes que podem afetar a difusão e utilização dos agentes como tecnologia nos diversos domínios de aplicação. De fato, o sucesso dos Sistemas Multiagentes é largamente dependente da acessibilidade e facilidade de uso das tecnologias apropriadas (linguagens de programação, bibliotecas de *software*, e ferramentas de desenvolvimento) que permitem uma implementação mais direta dos conceitos e técnicas que formam a base dos Sistemas Multiagentes (BELLIFEMINE; CAIRE; GREENWOOD, 2007). A seção 2.1.1 apresenta uma tecnologia de apoio a construção de agentes inteligentes, suas características e recursos.

2.1.1 *Java Agent Development Framework - JADE*

JADE (*Java Agent DEvelopment framework*) é um ambiente para desenvolvimento de aplicações baseado em agentes conforme as especificações da *Foundation for Intelligent Physical Agents* (FIPA) para interoperabilidade entre Sistemas Multiagentes totalmente implementado em Java. Foi desenvolvido e suportado pelo *Centro Studi e Laboratori Telecomunicazioni S.p.a.* (CSELT) da Universidade de Parma na Itália. O principal objetivo do *JADE* é simplificar e facilitar o desenvolvimento de Sistemas Multiagentes garantindo um padrão de interoperabilidade entre os mesmos através de um abrangente conjunto de agentes de serviços de sistema.

Um agente implementado em JADE deve estender a classe *jade.core.Agent*. A classe *Agent* fornece funcionalidades essenciais para que os agentes possam interagir com um plataforma, tais como métodos de registro, configuração e gerenciamento remoto do agente. A classe *Agent* possui o método *setup()*, onde é implementada a configuração inicial do agente, como a definição dos comportamentos do agente.

Os comportamentos de um agente são representados por instâncias de classes que implementam a classe *jade.core.behaviours.Behaviour*. A classe *Behavior* representa uma tarefa ou funcionalidade do agente. O método *action()* da classe *Behaviour* descreve as ações que são executadas por aquele comportamento do agente. A Figura 2.1(a) representa um agente *MyAgent* e seu comportamento estendendo as classes *Agent* e *Behaviour* respectivamente.

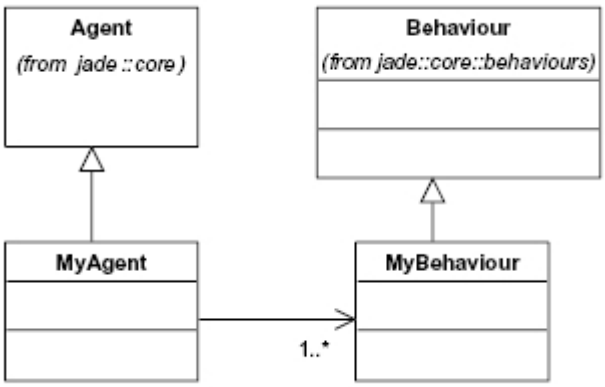
Em *JADE*, cada serviço/funcionalidade de um agente deve ser implementado como um ou mais comportamentos, estendendo classes e implementando métodos específicos do *framework*. Os comportamentos de JADE modelam arquiteturas reativas e a abstração de comportamento do modelo do agente de JADE permite a integração de *softwares* externos para enriquecer a arquitetura do agente. Como exemplo, têm-se os agentes reativos-deliberativos com JADE e máquina de inferência JESS.

Esse *framework* gerencia a instanciação de agentes e seu ciclo de vida e fornece um identificador globalmente único denominado AID (*Agent Identifier*) para que seja realizada também a troca de mensagens. Esse identificador possui uma sintaxe que inclui parâmetros como o nome do *host* no qual o agente se hospeda, o nome do agente naquele *host*, e o número da porta de comunicação.

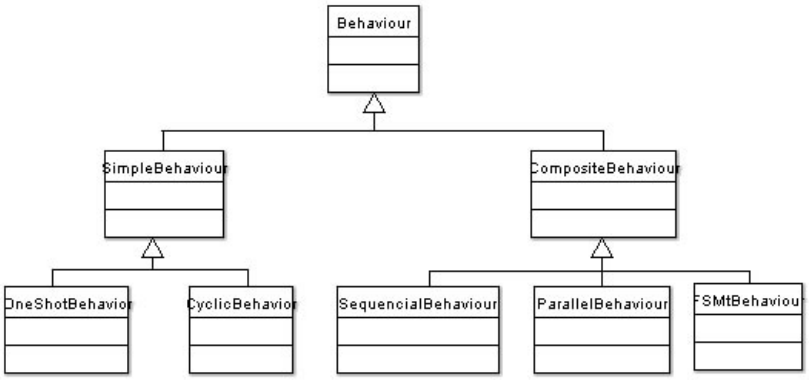
Para especificar a(s) tarefa(s) do agente, estas devem ser criadas como subclasses da classe *Behaviour*. E para executar um comportamento basta adicioná-lo a lista de comportamentos do agente através do método *addBehaviour()*. Assim um agente pode executar uma série de comportamentos cíclicos, seqüenciais ou paralelos.

Existem alguns tipos de comportamentos implementados no *framework JADE* importantes para a definição das ações de um agente. Esses tipos de comportamentos são apresentados na forma de diagrama de classes da UML (BOOCH; RUMBAUGH; JACOBSON, 1999) na figura 2.1(b).

Cada agente executando em uma plataforma é identificada por um "identificador de agente"(ou AID de *Agent Identifier*) que é representado por uma instância da classe *jade.core.AID*. O método *getAID()* da classe *Agent* permite obter o identi-



(a) Agentes e seus comportamentos no *JADE*



(b) Hierarquia de comportamentos e subclasses da classe Behaviour no *framework JADE*

Figura 2.1: Classes construídas no *framework JADE*

ficador do agente. Um objeto do tipo *AID* inclui um nome único global seguido de uma quantidade de endereços. O nome global de um agente tem o formato *<nome do agente>@<nome da máquina>:<porta>/JADE*, assim um agente chamado Rafael que está executando em uma máquina chamada *R1* terá *Rafael@R1:1099/JADE3* como seu nome global. Os endereços incluídos no *AID* são os endereços da plataforma onde o agente está executando. Estes endereços são usados quando um agente precisa comunicar-se com outros agentes de diferentes plataformas.

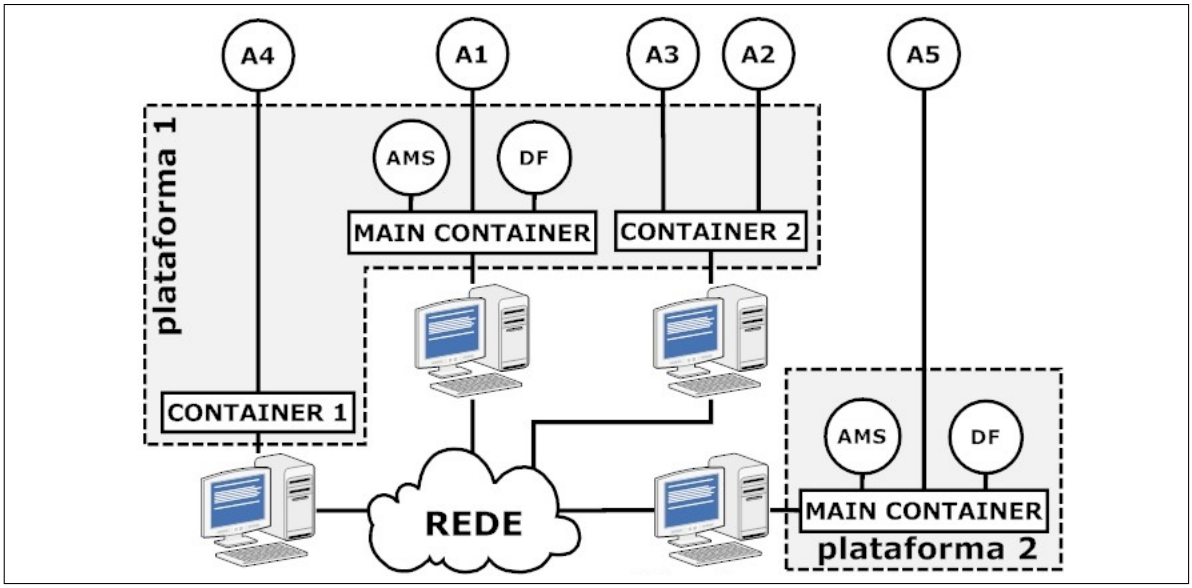


Figura 2.2: *Containers* e plataformas no JADE (OLIVEIRA, 2006)

Agentes são fundalmentalmente uma forma de distribuir processos. Sendo assim um sistema baseado em agentes deve seguir a noção clássica dos modelos de computação distribuída dividida em duas partes: componentes e conectores (BELLIFEMINE; CAIRE; GREENWOOD, 2007). Nesse sentido, *JADE* provê uma biblioteca de classes que facilita a comunicação entre os agentes. A comunicação é uma propriedade necessária para permitir a colaboração, negociação e cooperação entre entidades independentes (OLIVEIRA, 2006). O mecanismo de comunicação entre agentes é complexo porque além de autônomos e móveis, os agentes estão inseridos em sociedades. Nesse sentido, deve haver uma semântica clara, definida e aceita pelas entidades envolvidas. Esse *framework* para Sistemas baseados em agentes implementa as especificações da FIPA para comunicação de agentes. A esse conjunto

de especificações chama-se *FIPA Agent Comunication Language* (FIPA-ACL). Ela fornece performativas de comunicação e a possibilidade de associar uma ontologia para entendimento dos agentes do contexto da mensagem.

Os agentes implementados em *JADE* utilizam as especificações FIPA-ACL para comunicação. Isso implica que um agente não pode manipular diretamente a base de conhecimento de outro agente. Para que um agente influencie na decisão de outro agente ou de um grupo de agentes, o primeiro envia a mensagem para os de interesse, e estes por sua vez atualizam sua base de conhecimento ou atuam segundo sua vontade. Da mesma forma uma mensagem de solicitação deve ser encaminhada para ter acesso a determinado recurso ou informação.

Estes agentes de serviços de sistema tanto facilitam quanto possibilitam a comunicação entre agentes, de acordo com as especificações da FIPA: serviço de nomes (*naming service*) e páginas amarelas (*yellow-page service*), transporte de mensagens, serviços de codificação e decodificação de mensagens e uma biblioteca de protocolos de interação (padrão FIPA) pronta para ser usada. Toda sua comunicação entre agentes é feita via troca de mensagens. Além disso, os agentes de serviço lidam com todos os aspectos que não fazem parte do agente em si e que são independentes das aplicações tais como transporte de mensagens, codificação e interpretação de mensagens e ciclo de vida dos agentes. Em outras palavras, uma plataforma de agentes em complacência com a FIPA é um pacote, leia-se bibliotecas, para desenvolvimento de agentes em Java.

De acordo com BELLIFEMINE et al. (2003), o *JADE* foi escrito em Java devido a características particulares da linguagem, como por exemplo, a programação orientada a objetos em ambientes distribuídos e heterogêneos. Foram desenvolvidos tanto pacotes Java com funcionalidades prontas para uso, quanto interfaces abstratas para se adaptarem de acordo com a funcionalidade da aplicação de agentes.

Além disso, o *JADE* possui uma série de características que facilita a programação de Sistemas Multiagentes como (BELLIFEMINE; CAIRE; GREENWOOD, 2007):

- Plataforma distribuída de agentes - *JADE* pode ser dividida em vários "*hosts*"ou máquinas (desde que eles possam ser conectados via RMI). Apenas uma aplicação Java e uma *Java Virtual Machine* é executada em cada host. Os agentes são implementados como *threads* Java e inseridos dentro de repositórios de agentes chamados de *containeres* (*Agent Containers*) que provêm todo o suporte para a execução do agente.

- *Graphical User Interface* (GUI) - Trata-se de uma interface visual que gerencia vários agentes e *containeres* de agentes inclusive remotamente.

- Ferramentas de depuração - Esse *framework* contém ferramentas que ajudam o desenvolvimento e depuração de aplicações Multiagentes baseadas em *JADE*.

- Suporte à mobilidade de agentes. Os agentes podem, com certas restrições, migrar entre processos e máquinas. A migração do agente é realizada de forma transparente para a comunicação a interação entre esse agente e os outros continue acontecendo mesmo durante a migração.

- Suporte a execução de múltiplas, paralelas e concorrentes atividades de agentes - através dos modelos de comportamentos (*Behaviours*).

- Ambiente de agentes complacentes a FIPA - No qual incluem o sistema gerenciador de agentes (*Agent Management System* - AMS), o diretório facilitador (*Directory Facilitator* - DF) e o canal de comunicação dos agentes (*Agent Communication Channel* - ACC). Todos esses três componentes são automaticamente carregados quando o ambiente é iniciado.

- Suporte para ontologias e linguagens de contexto. Os programadores podem implementar um novo contexto que especifica o contexto dos requisitos da aplicação.

- Transporte de mensagens - Transporte de mensagens no formato FIPA-ACL dentro da mesma plataforma de agentes. Os agentes podem enviar mensagens de forma assíncrona sem a preocupação de detalhes sobre o envio.

- Biblioteca de protocolos FIPA - Para interação entre agentes *JADE*, dispõe de uma biblioteca de protocolos prontos para serem usados.

- Automação de registros - Registro e cancelamento automático de agentes com o Sistema de Gerenciamento de Agentes (do inglês *Agent Management System* - AMS) fazendo com que o desenvolvedor não se preocupe com isso.

- Serviços de Nomes (*Naming Service*) em conformidade aos padrões FIPA: na inicialização dos agentes, estes obtêm seus *Globally Unique Identifier* (GUID) da plataforma que são identificadores únicos em todo o ambiente.

- Integração - Mecanismo que permite que aplicações externas carreguem agentes autônomos *JADE*.

Além das características acima citadas, que por si só já facilitam muito o desenvolvimento de Sistemas Multiagentes, o *JADE* possui também ferramentas muito úteis que simplificam a administração da plataforma de agentes e o desenvolvimento de aplicações.

Em relação a FIPA, o *JADE* diminui para o programador muitas das especificações da FIPA, como:

- Não há a necessidade de implementar a plataforma de agentes: o sistema gerenciador de agentes (AMS), o diretório facilitador (DF) e o canal de comunicação dos agentes (ACC) são carregados na inicialização do ambiente.

- Não há a necessidade de implementar um gerenciamento de agentes: um agente é registrado na plataforma no seu próprio construtor, recebendo nome e endereço, sem falar na classe *Agent* que oferece acessos simplificados a serviços no DF.

- Não há necessidade de implementar transporte de mensagens e analisar gramatical das mensagens (do inglês *parsing*): isto é automaticamente feito pelo ambiente na troca de mensagens.

Cada instância de um ambiente de execução do *JADE* é chamado de um *container*, já que ele contém vários agentes. O conjunto de *containers* ativos é chamado de uma plataforma. Um *container* especial chamado de *Main Container* deve estar sempre ativo em uma plataforma e todas os outros *containers* devem se registrar junto a ele logo que são iniciados. Em outras palavras, o primeiro *container* de uma plataforma deve ser o *Main Container* e todos os demais devem ser *containers* "normais"e devem saber onde encontrar seu *Main Container* para se registrarem nele.

A Figura 2.2 ilustra os conceitos de *containers* e plataformas apresentando um cenário de duas plataformas compostas por 3 e 1 *containers*. Os agentes são identificados por um nome único e podem se comunicar independentemente de sua localização. Um agente pode se comunicar com outro em um mesmo *container*, como os agentes A2 e A3; em diferentes *containers* da mesma plataforma, como A1 e A2; ou em diferentes plataformas, como A4 e A5.

2.2 *Concerns* de Agente

A engenharia de *software* tem derivado progressivamente um melhor entendimento de características dos Sistemas computacionais (WOOLDRIDGE, 1999). O estudo dos sistemas baseado em agentes dentro da Engenharia de *Software* é relevante pois esse tipo de sistema geralmente apresenta arquitetura mais complexa e que exige um projeto de software atento às características inerentes aos próprios agentes. Isso ocorre porque os SMAs contêm muitos componentes que interagem dinamicamente, cada um com suas linhas de execução (*threads* de controle) e requerem um coordenado protocolo de comunicação entre os módulos autônomos.

A engenharia de *software* baseada em agentes (GARCIA et al., 2006; JENNINGS, 1999, 2000JENNINGS, 2000; LUCENA et al., 2004) é uma área emergente cujo objetivo é oferecer suporte ao desenvolvimento de Sistemas Multiagentes. Metodologias e arquiteturas de agentes são propostas e estudadas afim de tornar a construção de agentes inteligentes mais clara para os desenvolvedores e de forma mais prática. Os conceitos e arquiteturas podem variar porque a idéia dos agentes de *software* e os conceitos relacionados é muito vaga dentro da comunidade de agentes (EKDAHL, 2001). Cada pesquisador possui suas próprias definições e, portanto, projeta ou implementa os Sistemas Multiagentes a seu modo.

Um sistema baseado em agentes é composto por um conjunto de entidades que incorporam diferentes tipos de agentes e objetos que são inseridos em algum ambiente (JENNINGS, 1999). Um agente é visto como uma extensão de um objeto (GARCIA et al., 2003; GUESSOUM; BRIOT, 1999b; SHOHAM, 1993). No entanto, os objetos são entidades não-autônomas que representam elementos de sistemas passivos. Um agente é uma entidade interativa, adaptativa e autônoma que age no ambiente e manipula objetos (BRIOT, 1998; BASILI; SELBY; HUTCHENS, 1986; GARCIA; LUCENA, 2008). Somente sistemas interativos, adaptativos e autônomos são agentes (JENNINGS, 1999; BENEDICENTI, 2000). Além da autonomia, os agentes inteligentes possuem uma série de outras propriedades que o caracterizam, e que atuam direta-

mente na complexidade do sistema.

Entender os propriedades essenciais (*concerns*) que caracterizam um Sistema Multiagente e impulsionam o projeto e o desenvolvimento do mesmo é um grande desafio (EKDAHL, 2001). É muito difícil caracterizar e comparar arquiteturas e métodos orientados a agentes e promover efetivamente a separação de concerns de agentes. Além disso, ainda não há uma compreensão clara da interação entre as noções de agentes e objetos a partir de uma perspectiva da engenharia de *software* (BRIOT, 1998; HENDERSON-SELLERS, 2002; GARCIA et al., 2003; WOOLDRIDGE; CIANCARINI, 2001).

GARCIA (2004) descreve os interesses de um sistema baseado em agentes. A tabela 2.1 apresenta os interesses de um Sistema Multiagente divididos em três grupos(GARCIA, 2004): Requisitos fundamentais, de agência e adicionais. Essa tabela apresenta, também, uma breve descrição de cada requisito.

Os requisitos adicionais e alguns requisitos de agência, em uma abordagem orientada a aspectos, podem representar os aspectos do sistema (GARCIA, 2004). E os outros são modelados como classes, da mesma forma que a abstração orientada a objetos propõe. Dentre as propriedades funcionais que são mapeadas como classes do sistema, a de maior ênfase neste trabalho é a classe de Agente.

Um agente é composto de conhecimento e de um conjunto de propriedades, chamadas propriedades de agente ou propriedades de agência. As propriedades de agente são características comportamentais que podem ser incorporadas por um agente (GARCIA, 2004). Podem ser classificadas como propriedades de agência as características como autonomia, interação e adaptação, enquanto a colaboração, os papéis, a aprendizagem e a mobilidade não são condições necessárias, porém podem compor adicionalmente o compo do agente.

Diversos autores têm identificado que algumas propriedades de agentes são frequentemente transversais, tais como mobilidade (UBAYASHI; TAMAI, 2001b), interação (GARCIA, 2004; GARCIA et al., 2004), aprendizado (D'HONDT; GYBELS;

Requisito	Definição
Requisitos Fundamentais	
Ambiente	A entidade que contém os objetos e os agentes de sistema
Objeto	Uma entidade passiva que oferece serviços a outras entidades
Tipo de objeto	Uma categoria específica de objeto no sistema
Agente	Uma entidade interativa, adaptativa e autônoma que oferece serviços a outras entidades
Tipo de agente	Uma categoria específica de agente no sistema
Evento	Um evento é tudo que acontece para alterar o ambiente ou tudo aquilo com que o agente deve ser preocupar
Requisitos de Agência	
Conhecimento intrínseco	A funcionalidade básica do agente (ou seja os serviços básicos) disponibilizada para outras entidades do ambiente
Interação	Um agente se comunica com outras entidades por meio de sensores e efetores
Adaptação	Um agente adapta/modifica seu conhecimento e comportamento de acordo com os eventos observados
Autonomia	Um agente é capaz de agir sem intervenção externa direta; ele possui sua própria *thread* de controle, aceita ou recusa uma solicitação de serviço e age proativamente
Requisitos adicionais	
Aprendizagem	Um agente pode aprender com base em experiências anteriores enquanto interage com seu ambiente e colaborando com outros agentes
Mobilidade	Um agente é capaz de se transportar de um ambiente em uma rede para outro
Colaboração	Um agente pode cooperar com outros agentes a fim de alcançar seus objetivos e os objetivos do sistema
Função	Agrega uma parte do conhecimento do agente (conhecimento extrínseco), que é necessária em contextos colaborativos específicos

Tabela 2.1: Uma visão geral dos requisitos/características do Agente (GARCIA, 2004)

JONCKERS, 2004), autonomia (GUESSOUM; BRIOT, 1999a; AMANDI; PRICE, 1998), e colaboração (KENDALL, 1999; UBAYASHI; TAMAI, 2001b). Apesar de termos apresentado cada interesse (*concern*) separadamente, os requisitos do agente não são ortogonais - em geral, eles interagem entre si (AMANDI, 1997; AMANDI; PRICE, 1998; GUESSOUM; BRIOT, 1999b; PACE; CAMPO; SORIA, 2004).

Além disso, algumas propriedades de agente são sobrepostas, como a interação e a colaboração (GARCIA, 2004). A colaboração é vista como uma forma de interação mais sofisticada, porque incorpora comunicação e coordenação.

Os relacionamentos entre *concerns* do agente dependem da complexidade do agente e do tipo do mesmo (AMANDI, 1997; AMANDI; PRICE, 1998; GUESSOUM; BRIOT, 1999b; PACE; CAMPO; SORIA, 2004). A figura 2.3 mostra um exemplo de interligação entre os *concerns* de um Sistema Multiagente.

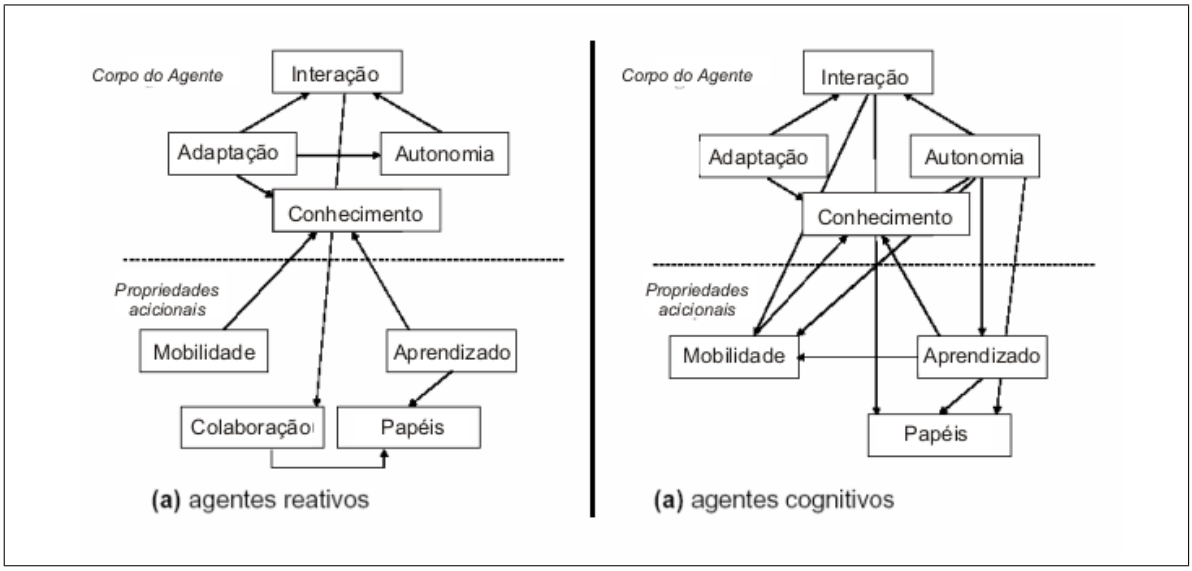


Figura 2.3: Relacionamento entre *concerns* do agente (GARCIA, 2004)

Nota-se, pela figura 2.3 que o acoplamento e dependência entre os módulos em sistemas baseados em agentes é alto e mesmo as suas propriedades adicionais são fortemente ligadas. Essa é uma característica que incentiva a utilização de técnicas de abstração que modularizem esses requisitos de maneira a desacoplar os *concerns* do sistema.

Abstrações orientadas a aspectos para capturar interesses tranversais dos agentes que são difíceis de modularizar com abstrações orientadas a objeto e especificações orientadas a agentes (GARCIA et al., 2005).

Vários benefícios podem ser alcançados em um Sistema Multiagente com o uso de uma técnica de modularização como a programação orientada a aspectos.

- Separação de *concerns*; A abstração orientada a objetos nem sempre é capaz de modularizar os requisitos transversais de um sitema baseado em agentes. Os módulos de um SMA ficam, geralmente, fortemente acoplados e entrelaçados e dificultam a separação das classes e métodos do projeto do agente.

- Transparência; Aspectos podem ser usados para adicionar funcionalidades aos comportamentos de um agente ou a uma sociedade de agentes. Assim sendo, o comportamento básico do agente não faz referência as funcionalidades acrescidas utilizando recursos de abstração orientada a aspectos.

- Facilidade de evolução; Utilizando abstrações orientadas a aspectos, novas regras do SMA podem ser inseridas com mais facilidade. Para isso basta adicionar novos conjuntos de junção (*pointcuts*) para acoplar novas funcionalidades.

- Minimização da esquizofrenia do agente. O problema de separar o agente que deve ser um único objeto refere-se à "esquizofrenia de agente"(CZARNECKI; EISENECKER, 2000). O uso de delegação no projeto orientado a objetos oferece suporte a uma maior flexibilidade para separar e compor interesses (AMANDI, 1997; AMANDI; PRICE, 1998). Diferente da composição baseada apenas em herança, a delegação oferece suporte à separação de interesses usando árvores de herança. Cada árvore de herança deve tratar de um determinado interesse do agente (GARCIA, 2004). No entanto, ela tem algumas desvantagens. A separação explícita de interesses não é alcançada. Por exemplo, em um agente cognitivo, as chamadas aos mecanismos de aprendizagem estão entrelaçadas com as funcionalidades básicas do agente. Além disso, é necessária a criação de algumas classes adicionais para tratar da composição dos interesses. Em uma

arquitetura de *software* orientada a aspectos esse problema é minimizado, ou seja, o agente é representado por uma única instância da classe de agente no sistema.

- Redução da replicação de código. Ao usar herança para o desenvolvimento de agentes, o comportamento comum de qualquer tipo de agente pode ser agrupado em uma classe de Agente. Podemos produzir diferentes tipos de agentes usando a especialização dessa classe. No entanto, se a herança for a única técnica de projeto usada, o projeto do agente apresenta vários problemas (AMANDI, 1997; AMANDI; PRICE, 1998; GUESSOUM; BRIOT, 1999b). O principal deles é a presença de código replicado ao longo da árvore de herança (AMANDI, 1997; AMANDI; PRICE, 1998). Ademais, muitas outras classes devem ser criadas para oferecer suporte à composição de vários concerns do agente (GARCIA, 2004).

2.3 Programação Orientada a Aspectos

Em Ciência da Computação, as técnicas de programação têm experimentado inúmeras transformações, evoluindo desde construções de baixo nível - como as linguagens de montagem - até abordagens de alto nível - como a Programação Orientada a Objetos (POO) (ELRAD; FILMAN; BADER, 2001; ELRAD et al., 2001).

Por influência das pesquisas na área de Engenharia de *Software* para o projeto de linguagens de programação, os projetos e programas têm sido idealizados de forma mais próxima do raciocínio dos desenvolvedores devido a adoção de abstrações mais elevadas na modelagem dos sistemas. Essas pesquisas têm sido uma fonte valiosa para a implementação de conceitos e abstrações importantes que hoje são comuns nas técnicas de programação. Uma questão importante, abordada nesse trabalho, para a contrução mais clara de sistemas de computação e de informação é a separação de interesses.

O princípio da separação de interesses defende que, para superar a comple-

xidade, deve-se resolver uma questão importante por vez (DIJKSTRA, 1976). Na Engenharia de *Software*, esse princípio está relacionado à modularização e à decomposição de sistemas (PARNAS, 1972). Os sistemas de software complexos devem ser decompostos em unidades modulares menores e claramente separadas, cada uma lidando com um único requisito (PARNAS, 1972). Se bem realizada, a separação de interesses pode oferecer muitos benefícios cruciais (DIJKSTRA, 1976; TARR et al., 1999). Ela oferece suporte a uma melhor manutenibilidade com alterações aditivas, em vez de invasivas, melhor compreensão e redução da complexidade, melhor reusabilidade e uma integração de componentes simplificada.

Sem os meios apropriados para a separação e a modularização, os requisitos transversais tendem a ficar espalhados e entrelaçados com outros interesses. As conseqüências naturais são uma menor compreensibilidade, uma menor manutenibilidade e menos reusabilidade dos artefatos do *software*. A Programação Orientada a Aspectos (POA)(KICZALES et al., 1997) é uma tecnologia em evolução que oferece suporte a um novo tipo de separação de interesses no nível do código-fonte.

O conceito de interesse (*concern*) foi introduzido por DIJKSTRA (1976) há quase trinta anos, e tem sido utilizado em Engenharia de *Software* para se referir tanto a atributos de alto nível de um determinado *software* quanto a funcionalidades de baixo nível (ELRAD; FILMAN; BADER, 2001).

De maneira geral, um interesse pode indicar tanto um requisito funcional, quanto não-funcional de um sistema. Em um projeto de *software* - para fins de simplicidade, legibilidade, e conseqüente facilidade na manutenção e maior potencial para reutilização - é importante que os vários interesses relevantes para um sistema estejam localizados em módulos separados. De forma geral, todas as técnicas de programação oferecem suporte a esse tipo de separação de interesses[2](DIJKSTRA,

[2]O princípio da separação de interesses tem como objetivo dividir um domínio de conhecimento em partes menores, para poder entender objetivamente cada uma delas. Portanto, a implementação de um sistema particionado em módulos que contêm cada um dos interesses é uma conseqüência da utilização desse princípio.

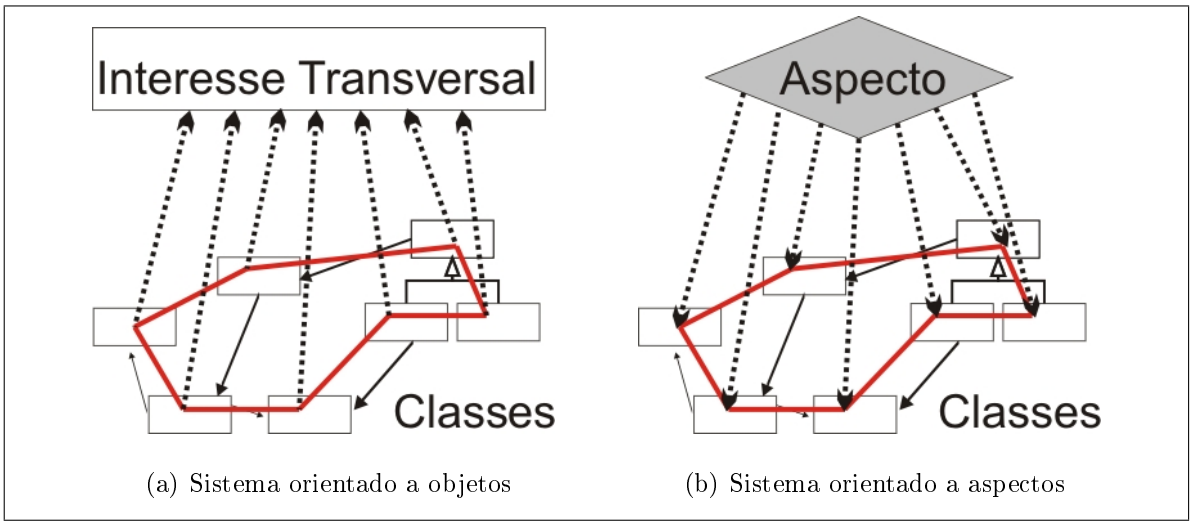
1976), porém cada uma a sua maneira (utilizando sub-rotinas, procedimentos, funções, classes, APIs) e em graus diferentes.

Com o intuito de resolver esse problema, foi proposta a Programação Orientada a Aspectos (POA), uma tecnologia de suporte para a implementação dos interesses transversais de maneira localizada. A POA é resultado de vários estudos em metaprogramação (BAKER; HSIEH, 2002), programação orientada a assuntos (HARRISON; OSSHER, 1993), filtros de composição (BERGMANS; AKSIT, 2001), programação adaptativa (LIEBERHERR, 1996) e outros. O termo aspecto refere-se aos interesses transversais que podem ser implementados em módulos separados (KICZALES et al., 2001c, 1997KICZALES et al., 1997).

KICZALES et al. (1997) definiram dois importantes termos básicos: o componente, que encapsula as propriedades do sistema através de uma abordagem estruturada ou orientada a objetos, e o aspecto, que permite encapsular, por meio de uma abordagem orientada a aspectos, propriedades que essas abordagens não são capazes de tratar. Os aspectos tratam elementos que representam propriedades que precisam ser satisfeitas em vários componentes de um sistema, como configuração de alguma característica do sistema que envolve vários componentes, ou que não fazem parte dos requisitos funcionais dos componentes, por exemplo, monitoração.

Aspecto é o termo usado para denotar a abstração cujo objetivo é oferecer suporte a um melhor isolamento de requisitos transversais. Eles são unidades modulares de requisitos transversais associados a um conjunto de classes e objetos. Um aspecto pode afetar uma ou mais classes e/ou objetos de várias formas. Ele pode alterar a estrutura estática (*static crosscutting*) ou a dinâmica (*dynamic crosscutting*) de classes e objetos (KULESZA; SANT'ANNA; LUCENA, 2005). Ou seja, os aspectos precisam saber os pontos onde os interesses transversais devem se relacionar com os interesses de negócio.

A Figura 2.4(a) ilustra a estrutura de um típico sistema orientado a objetos. Já na Figura 2.4(b), o interesse transversal é modularizado em um aspecto, onde se pode perceber a existência da inversão de controle. Nota-se que em um sistema orientado a



(a) Sistema orientado a objetos (b) Sistema orientado a aspectos

Figura 2.4: Inversão de controle e redução do espalhamento de código resultantes da aplicação da abstração orientada a aspectos (KULESZA; SANT'ANNA; LUCENA, 2005)

Aspectos os interesses transversais são tratados de forma modular. Posteriormente os fragmentos de código são inseridos nos devidos locais e compor o programa executável. Esse processo pode ser dividido em três etapas bem definidas.

Na fase de decomposição aspectual (*Aspectual Decomposition*), etapa primária no desenvolvimento orientado a aspectos, os interesses do sistema abordado são decompostos em interesses de negócios: requisitos funcionais e interesses transversais.

Na abstração orientada a aspectos, os requisitos funcionais devem ser implentados separadamente dos interesses transversais (LADDAD, 2003). Os primeiros são implementados nas classes que compõem o sistema. Já os interesses transversais são modelados como aspectos do sistema. Essa fase é chamada de implementação dos interesses (*Concern Implementation*).

A terceira e última etapa é a Recomposição Aspectual (*Aspectual Recomposition*). Nessa etapa, as regras de recomposição devem ser especificadas nos aspectos. O processo de recomposição (*weaving*) se utiliza dessas regras para compor o sistema final. Cada linguagem de programação que implementa o conceito de Aspectos utiliza de recursos variados para auxiliar o usuário na realização dessas etapas.

De acordo com CHAVES(2004), um sistema construído com POA é composto de: (*i*) uma linguagem de componentes; (*ii*) uma (ou mais) linguagem(ns) de aspectos; (*iii*) código de programas de componentes; (*iv*) código de programas de aspectos; e (*v*) um compilador de aspectos (ou combinador de aspectos, do inglês *aspect weaver*), que é capaz de combinar o código dos programas de componentes e de aspectos de forma a gerar um programa final.

A POA oferece mecanismos para que os aspectos possam ser construídos em módulos separados e provê meios para a definição de pontos do programa onde esses aspectos possam definir comportamento. A partir daí, um programa executável pode ser gerado, combinando os módulos básicos com os aspectos. Dessa forma, a POA pretende dar suporte aos interesses transversais assim como a POO tem dado suporte aos objetos (KICZALES et al., 2001c). Um exemplo do esquema da abstração orientada a aspectos é apresentado na figura 2.5.

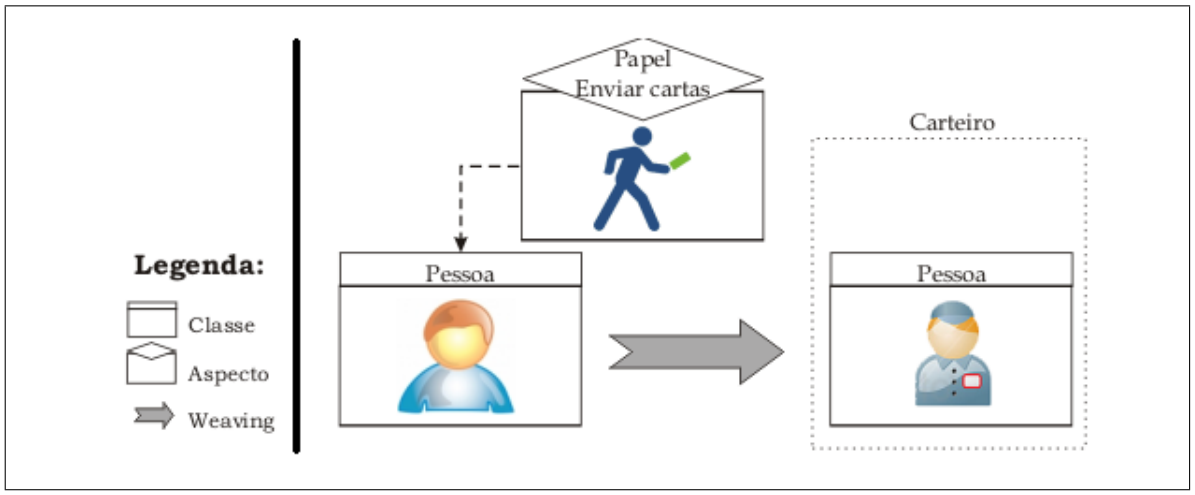


Figura 2.5: Esquema do uso de aspectos no acréscimo de funcionalidades usando o exemplo do papel carteiro

A Figura 2.5 mostra um exemplo ilustrativo de como funciona a adição de aspectos em um dado sistema. Nesse exemplo inicialmente tem-se a classe *Pessoa*. Essa classe possui atributos e métodos comuns para desempenhar esse papel. Declara-se um aspecto responsável pela agregação das funções necessárias para transformar uma pessoa em um carteiro e vincula-se esse aspecto à classe *Pessoa* através dos pontos

de junção. Por um processo de transformação chamado de Recomposição Aspectual (*Weaving*) a classe *Pessoa* agrega novas funcionalidades codificadas no aspecto *Papel_ enviar_ cartas* e então é competente para realizar as funções de carteiro.

Desde a primeira proposta de POA (KICZALES et al., 1997), muitas pesquisas têm sido feitas nesse tema, principalmente no que diz respeito a linguagens de suporte e a aplicações. Como essa técnica é relativamente nova, ainda existem muitos campos pouco explorados, e alguns deles estão começando a ser abordados. Exemplos desses assuntos são: uma teoria para a POA, abordando essa técnica a partir de perspectivas formais (WALKER; ZDANCEWIC; LIGATTI, 2003); estudos empíricos para a avaliação da efetividade da técnica (MURPHY et al., 2001; WALKER; BANIASSAD; MURPHY, 1999); projeto de programas orientados a Aspectos (STEIN; HANENBERG; UNLAND, 2002; CLARKE; WALKER, 2002; ZHOU DEBRA RICHARDSON, 2004); e verificação, validação e teste de programas orientados a Aspectos (XIE et al., 2006; XU; XU, 2006; LEMOS et al., 2007).

A Programação Orientada a Aspectos atualmente é vista como uma evolução das técnicas citadas anteriormente e procura facilitar a resolução de problemas relacionados à modularização do código de interesses transversais, dando assim suporte a implementação de requisitos de adaptação encontrado em determinadas aplicações.

Assim, o objetivo de POA é promover a especificação separada de interesses não-transversais (os componentes) e transversais (os aspectos) de sistemas de modo bastante simples, e ainda compô-los, em momento apropriado, nos pontos de junção adequados, e por fim, construir o sistema desejado.

A principal vantagem do uso de POA é que a separação de interesses permite a facilidade para o processo de criação, entendimento, evolução e futura manutenção do *software*.

Isso é possível devido os interesses transversais não aparecerem no código dos componentes, ficando corretamente encapsulados nos aspectos. Os componentes e aspectos ficam mais legíveis individualmente, além da forma como eles se compõem.

Além disso, na Programação Orientada a Aspectos o reuso de código é sensivelmente mais efetivo (CHAVES, 2004), uma vez que o componente torna-se reutilizável em uma maior variedade de situações, devido a inexistência de dependências relacionadas a requisitos implementados por aspectos, que normalmente representam requisitos secundários e não relacionados com a lógica de negócio da aplicação.

Entre as desvantagens da adoção de POA, podemos citar: (*i*) dificuldade inicial de aprendizado de um novo paradigma (por exemplo, familiaridade com os novos conceitos e construções de uma linguagem de aspectos); (*ii*) baixa maturidade dos ambientes de desenvolvimento que pode levar a problemas como tempo de compilação (ou combinação entre componentes e aspectos), tamanho e desempenho na execução do código final gerado na compilação; e (*iii*) problemas relacionados à composição de aspectos inerentes da orientação a aspectos (por exemplo, aspectos que se compõem podem ser incompatíveis e aspectos que implementam propriedades que possuem precedência entre si). Com exceção do item (*i*), a maioria dos problemas já são satisfatoriamente resolvidos com a evolução das ferramentas disponíveis atualmente.

2.3.1 Linguagem *AspectJ*

AspectJ (KICZALES et al., 2001a, 2001bKICZALES et al., 2001b) é a uma linguagem orientada a aspectos utilizada por várias soluções de *software* da indústria, como por exemplo, o servidor de aplicação *JBoss*. Inicialmente criada no *Xerox Palo Alto Research Center*, hoje é mantida pelo projeto de código aberto *Eclipse*.

A linguagem de aspectos *AspectJ* é uma extensão orientada a aspectos da linguagem de programação Java (GOSLING et al., 2005), o que permite o uso do paradigma POA de forma eficiente em aplicações Java puramente orientadas a objetos. Com *AspectJ* os aspectos e seus elementos são combinados com o código Java (o programa de componentes) através da utilização de um compilador especial ou combinador, chamado de *weaver*. O resultado final desse processo é um código Java comum (*bytecode*), que torna possível a execução da aplicação com aspectos total-

mente portável em qualquer ambiente de execução da plataforma Java.

Além da modularização das funcionalidades que implementam interesses transversais, *AspectJ* torna possível o desenvolvimento independentes dessas partes do sistema, através da associação entre aspectos e componentes de uma forma que apenas o aspecto referencia o componente. É possível, por exemplo, que os aspectos que implementam requisitos não-funcionais detectem eventos ou afetem o comportamento de componentes que implementam requisitos funcionais, sem estes não terem conhecimento daqueles requisitos.

Como uma extensão da linguagem Java, *AspectJ* acrescenta alguns novos conceitos e construções associadas. Esses conceitos e construções são: aspecto, ponto de junção (*join point*), um ponto bem definido da execução dinâmica de um programa em Java, conjuntos de junção (*pointcut*), adendos (*advice*) e declaração intertipos (*inter-type declaration*), que serão explicados adiante.

Aspecto representa a unidade de modularidade para interesses transversais em *AspectJ*. Eles efetivamente podem alterar componentes sobre os quais atuam, encapsulando uma funcionalidade de natureza estática (estrutural) e/ou dinâmica (comportamental) que entrecorta outras classes do programa. Um aspecto é similar a uma classe Java e, além de conter métodos, atributos e relacionamento com outros aspectos, é definido em termos de conjuntos de junção (*pointcuts*), adendos (*advices*) e declarações intertipos. Instâncias de um aspecto podem ser associados a um objeto, mas também com outros elementos de tempo de execução da linguagem Java. A instancia é automaticamente criada quando o alvo da associação (elemento de tempo de execução) é ativado. Os aspectos podem ser flexivelmente incluídos e/ou excluídos do contexto de componentes de produção, assim como serem utilizados apenas na fase de desenvolvimento. Dentre os recursos da linguagem *AspectJ* destacam-se os pontos de junção (*join points*), os conjuntos de junção (*pointcuts*), os adendos (*advices*) e os objetos com reflexão.

Os Pontos de Junção (*Join Point*) são pontos bem definidos na execução de um programa. A linguagem *AspectJ* contém recursos para definir construções que

referenciam pontos do código-fonte como chamada de um método específico de uma classe, ou todos os métodos de uma classe ou um conjunto de classes, dentre outros. Por exemplo, se um ponto de junção declara *call(protected void Agent.setup())*, o ponto de código referenciado pelo aspecto é a chamada do método *setup()* da classe *Agent*, com retorno tipo *void* e com modificador de acesso *protected*.

Os Conjuntos de Junção (*Pointcuts*) definem uma coleção de pontos de junção, que são elementos que especificam como as classes e os aspectos estão relacionados. Além disso, descrevem e dão nome a conjuntos de pontos de junção. As definições de pointcuts são especificadas utilizando designadores (pré-definidos na linguagem ou elaborado pelo programador a partir dos existentes) que podem representar: chamada de métodos, execução de métodos, criação de objeto, atribuição de referência, lançamento de exceção e etc. Os conjuntos de junção podem ser combinados utilizando operadores lógicos, além da possibilidade de utilizar curingas (do inglês *wild cards*) para construções mais avançadas.

O adendo (*Advice*) é um mecanismo análogo a um método onde se define qual é o trecho de código que deve ser executado quando um conjunto de junção (*pointcut*) é atingido. *AspectJ* suporta adendos dos tipos *before*, *after* e *around*, que determinam o tempo que o seu código é executado. Os *advices before* e *after* definem uma execução antes e depois, respectivamente, de um *pointcut*. Já o *advice around*, além de definir a execução de um código quando um *pointcut* é atingido, ele permite o controle sobre a computação de cada ponto de junção definido (por exemplo, a substituição total de uma execução de um método).

Os Objetos com Reflexão são utilizados para se ter acesso, quando um ponto de junção é acionado, às informações do seu contexto a partir dos objetos *thisJoinPoint*, *thisJoinPointStaticPart* e *thisEnclosingJoinPointStaticPart*.

Adicionalmente, um aspecto pode introduzir métodos, atributos, relacionamento entre interfaces e classes usando a construção chamada de declaração intertipos. De forma simplificada, podemos ver *pointcuts* e *advices* como a representação da parte dinâmica da linguagem *AspectJ*. Já as construções de declaração de intertipos,

permite a mudança na estrutura estática de um programa, por exemplo, adicionando campos e/ou métodos a classes alterando o relacionamento entre classes.

Os recursos previstos na linguagem *AspectJ* possibilitam a inserção de um mesmo adendo em diversos pontos de código de diversas classes do sistema. Sendo assim, um requisito arquitetural ou transversal é implantado, na fase de recomposição aspectual, em todos os pontos do código que se enquadrem nas especificações do conjunto de junção declarado no aspecto. Essa possibilidade é interessante, tanto no campo da engenharia de *software* quanto no de linguagens de programação pois permite a construção de um código reutilizável, e de fácil migração.

2.4 Eclipse

O *Eclipse* (ECLIPSE, 2008) é uma plataforma que disponibiliza um ambiente de desenvolvimento integrado, do inglês IDE (*Integrated Development Environment*), que fornece um conjunto de funcionalidades que dão suporte ao ciclo de desenvolvimento de aplicações, tais como: editores, compiladores, depuradores, modelagem, refatoração, distribuição, testes automatizados e geração de código. Esses recursos tornam o desenvolvimento mais rápido e prático. Dentre as linguagens suportadas pelo Eclipse, pode-se citar: *C*, *C++*, *PHP*, *Ruby*, dentre outras. Porém, o seu uso é maior em aplicações Java.

Em virtude do uso da tecnologia de *plugins*, o Eclipse permite personalizar o ambiente de trabalho do desenvolvedor de acordo com o projeto que está sendo desenvolvido, seja ele, um simples projeto com páginas HTML estáticas, até aplicações com uso de EJBs (Enterprise Java Beans), frameworks diversos ou J2ME (Java to MicroEdition) [SUN, 2005]. Além disso, a tecnologia de *plugin* possibilita a criação de seus próprios *plugins* (DAI; MANDEL; RYMAN, 2007).

O projeto *Eclipse* surgiu a partir de um consórcio de empresas lideradas pela IBM em novembro de 2001. No início de 2004, ele se tornou um *software* gratuito, livre e de código aberto sob a licença EPL (*Eclipse Public License*). Desde então,

o Eclipse vem evoluindo com o apoio da comunidade de colaboradores e de grandes empresas e instituições. Essa evolução distribuída só é possível porque esse projeto é efetivamente bem coordenado e organizado.

O *Eclipse* possui uma arquitetura baseada em *plugins*. Um *plugin* é uma funcionalidade integrada a um ambiente de desenvolvido com a finalidade de facilitar o trabalho de seu usuário. No *Eclipse*, praticamente tudo é *plugin*, com exceção da Plataforma de Inicialização (*Platform Runtime*), que é o seu núcleo. Este é responsável por iniciar o ambiente e carregar todos os outros *plugins* e componentes. Esse modelo de arquitetura permite que o desenvolvedor personalize o ambiente de trabalho com os *plugins* desejados de acordo com o perfil do projeto em desenvolvimento. Existe também um ambiente, chamado de PDE (*Plugin Development Environment*), que fornece meios para a criação de seus próprios *plugins*.

Um *plugin* desenvolvido para auxiliar a programação orientada a Aspectos no ambiente Eclipse é o *AspectJ Development Tools - AJDT*. O AJDT é o projeto da organização Eclipse que permite o uso de AOP em Java, inclui o AspectJ e permite controle e produtividade no seu projeto. Trata-se de um conjunto de ferramentas de visualização de aspectos, conjuntos de junção e o mecanismo de recomposição aspectual. Esse *plugin* é um projeto Eclipse de código aberto que contém uma série de mecanismos necessários para desenvolver e executar aplicações *AspectJ*.

Além da integração fácil de *plugins*, como o AJDT, o Eclipse possui uma variedade de recursos inovadores é oferecida, proporcionando mais produtividade. Mecanismos avançados de refatoração, ferramentas de geração de código, recursos que auxiliam na escrita de código e atalhos são alguns exemplos. Além disso, o ambiente gráfico do *Eclipse* é construído com uma biblioteca de componentes própria, a SWT (*Standard Widget Toolkit*). Ela oferece uma série de componentes gráficos e interfaces de usuário que, entretanto, não comprometem a seu desempenho.

O conjunto de características aqui discutidas sobre o *Eclipse* o credenciou como uma IDE bastante utilizada por desenvolvedores Java. Isso demonstra que existe um esforço de transparência e colaboração para incrementar em recursos o projeto *Eclipse*.

2.4.1 O Ambiente de desenvolvimento de *plugins* do *Eclipse* - PDE

O PDE (*Plugin Development Environment*) (PDE, 2008) é uma extensão do Eclipse que fornece um ambiente completo para a construção, compilação, depuração, teste e empacotamento de um *plugin*. No *Eclipse*, um *plugin* é um programa escrito em Java que respeita uma série de padrões e especificações, que permitem utilizar os recursos da plataforma e se comunicar com ela. Dentre esses recursos, podemos citar:

- *Standard Widget Toolkit* - SWT: é uma biblioteca de componentes gráficos para interface gráfica que permitem à plataforma Eclipse ter funcionalidades nativas de modo independente de sistema operacional.
- o JFace; e o
- *Java Development Tools* - JDT .

O SWT oferece uma biblioteca de componentes gráficos, como botões, campos e tabelas. O JFace trabalha em conjunto com SWT e trata de operações rotineiras relacionadas às interfaces de usuário. Já o JDT dá suporte ao desenvolvimento de aplicações ou *plugins* Java.

A idealização de um *plugin* pode ser dividida em três partes: onde, como e o que fazer. A primeira está relacionada ao local onde uma funcionalidade será disponibilizada na bancada de trabalho (*workbench*), chamado de ponto de extensão. Por exemplo, isso pode ser feito através de um ícone na barra de ferramentas ou de uma nova opção de menu. A segunda parte começa depois do acionamento da funcionalidade, onde pode ser necessária a exibição de interfaces gráficas contendo informações, campos, botões e opções para configurar o que, por fim, na terceira parte, irá acontecer. O resultado final pode ser uma geração de código, uma refatoração, uma formatação de classes ou a configuração de alguma variável do ambiente.

Com o PDE, qualquer programador pode desenvolver seu próprio *plugin* e disponibilizá-lo para a comunidade, o que gera um ciclo virtuoso de auto-ajuda, pois na

maioria das vezes o *plugin* com a funcionalidade que se precisa já existe. Baseado em uma filosofia de cooperação, grande parte dos *plugins* têm seu código-fonte disponível para que a comunidade ajude a incorporar novas funcionalidades e a melhorá-los. Dessa forma, o ambiente se torna cada vez mais rico à medida que novas extensões são adicionadas ou aperfeiçoadas. O PDE contribui, com seus recursos, para o sucesso do Eclipse.

2.5 Trabalhos relacionados

Nesta subseção, são apresentados alguns trabalhos que influenciaram no amadurecimento da idéia do mecanismo proposto, que visa simplificar o gerenciamento de interesses transversais em Sistemas Multi-agentes. Existem diversos propostas de trabalhos em que a aplicação de técnicas transversais em Sistemas Multiagentes é constatada. No decorrer da seção são apresentados alguns trabalhos que utilizam abstrações orientadas a aspectos para modularizar esses requisitos em agentes.

Em (MEHMOOD; ASHRAF; RASHEED, 2005) é proposto um *framework* orientado a aspectos que visa separar os parâmetros e intruções que caracterizam o requisito funcional "performance"do sistema multiagente. A proposta dos autores é tratar de forma modular tais variáveis em relação às características que compõem o corpo do agente, os requisitos funcionais e demais requisitos não-funcionais de SMAs. Dentre essas variáveis que envolvem a performance de um sistema pode-se citar: o tempo de resposta dos agentes, o número máximo de repetições, os intervalos de espera por mensagens, o tempo de execução, dentre outras. As propriedades de mobilidade, interação, autnomia, adaptação, aprendizado e colaboração são classificadas como interesses intrínsecos ao agente ou de agência. Porém os interesses de sincronização, performance, controle de segurança e persistência são tratadas como não-funcionais. O *framework* proposto contém um conjunto de interfaces categorizadas em "interfaces normais"e "interfaces transversais". As interfaces normais somente prestam serviços para outras e são utilizadas para as classes que as implementem codificar as outras

tarefas que um agente deve executar. As interfaces transversais especificam quando e como os aspectos arquiteturais afetam outros componentes do sistema.

De forma semelhante ao trabalho anterior, em (LOBATO et al., 2006) é apresentado um *framework* de construção de sistemas multi-agentes, chamado AspectM, que modulariza o interesse de mobilidade. Nesse trabalho, uma classe de agente estacionário recebe inserção de código, por meio de aspectos, e assim torna-se capaz de mover-se entre plataformas e nestas realizar suas tarefas. Uma comparação baseada na modularização do interesse de mobilidade é realizada com outros *frameworks* orientados a objetos como Aglets (LANGE; MITSURU, 1998), JADE (BELLIFEMINE; POGGI; RIMASSA, 1999) e RoleEP/Epsilon (UBAYASHI; TAMAI, 2001a). Os autores apresentam ainda uma integração do *framework* proposto com a plataforma de mobilidade do JADE.

GARCIA et al. (2001) propõe o uso de aspectos e de reflexão computacional para separar os requisitos intrínsecos e extrínsecos de sistemas baseados em agentes. Aspectos são utilizados nesse trabalho para modularizar os requisitos de agência do sistema, ou seja, os requisitos extrínsecos. Os autores sugerem que o comportamento do agente e seus estados sejam modelados com o uso de classes e herança.

Uma outra abordagem, visando uma modularização adequada é a separação dos interesses dos Sistemas Multiagentes por papéis de agente. Como exemplo tem-se o trabalho de CABRI; LEONARDI; ZAMBONELLI (2002), no qual os autores visam a separação do interesse de interação entre os agentes, os dados de cada papel de agente necessário para o sistema são persistidos em arquivos XML e posteriormente transformados em interfaces do sistema. É proposta um modelo de três camadas para tratar das regras da aplicação, dos papéis de agentes representados pelas interfaces e a última camada que é a de interação. A interação entre os agentes é modularizada utilizando a filosofia orientada a aspectos.

Em (KULESZA et al., 2004), abordou-se a geração de código em sistemas multiagentes partindo de uma especificação primária. Foi definida pelos autores uma linguagem específica de domínio chamada Agent-DSL. Essa linguagem é utilizada

para especificar as propriedades de agência dos agentes como conhecimento, interação, adaptação, autonomia e colaboração e porterimente auxiliar na geração das classes de agentes. Um *framework* orientado a aspectos fornece uma família de interfaces para o desenvolvedor que as utiliza para codificar as instruções específicas dos agentes. Os aspectos gerados abstratos e apontam para as interfaces que representam os requisitos de agência dos agentes. Porém nenhum recurso de programação orientada a aspectos é fornecida ao desenvolvedor para compor o código do corpo dos agentes. Além disso, como as classes são geradas utilizando recursos desde a especificação do projeto o sistema proposto não pode ser utilizado para acrescentar propriedades transversais em Sistemas Multiagentes que tenham sido desenvolvidos sem os artefatos de especificação.

Já em (GAZOLLA, 2008) é apresentado um mecanismo visual de modularização de *logging* em sistemas computacionais. O mecanismo é baseado na filosofia orientada a aspectos para separar requisitos transversais e trata de forma transparente ao usuário a modularização do interesse em questão. O desenvolvedor não necessita de aprender quaisquer técnicas de modularização de requisitos transversais e nem conhecer a sintaxe de *AspectJ*. O mecanismo proposto persiste os dados coletados das interfaces em XML e, utilizando JAXB, realiza a geração dos aspectos envolvidos na modularização do interesse *loggging*.

Percebe-se que os estudos da abstração orientada a aspectos, de técnicas de modelagem para Sistemas Multiagentes, e de ferramentas que auxiliam o desenvolvimento de sistemas recebem grande dedicação da comunidade científica. Porém existe a necessidade de uma ferramenta que auxilie o desenvolvedor de SMAs a tratar as propriedades de agentes de forma modular, sem que seja necessário o conhecimento da sintaxe de linguagens voltadas para a modularização de requisitos transvesais. É desejável que as ferramentas que sirvam a esse propósito disponibilizem ao desenvolvedor uma forma de estudar o comportamento do Sistema Multiagente com e sem as propriedades em questão.

Capítulo 3

Métodos e soluções adotadas

> "*A common mistake people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools*"(Douglas Adams).

Este capítulo tem por objetivo apresentar, em um nível mais detalhado, a solução proposta para acréscimo de propriedades transversais em Sistemas Multiagentes. A metodologia deste trabalho assume caráter qualitativo com base em um estudo de caso e os objetivos específicos foram apresentados na seção 1.3. A fase inicial do trabalho consistiu em estudar teoricamente e na prática o funcionamento de sistemas baseados em agentes e analisar alguns recursos tecnológicos utilizados para facilitar sua construção. Nessa fase, metodologias de construção de SMAs encontradas em literatura, protocolos de comunicação de agentes, ambientes de desenvolvimento integrados, e *frameworks* para sistemas baseados em agentes foram estudadas e analisadas. Terminado o estudo bibliográfico, optou-se pelos recursos tecnológicos necessários para alcançar os objetivos traçados.

A proposta é desenvolver um mecanismo visual baseado em aspectos que facilite o gerenciamento do interesses transversais em Sistemas Multiagentes. O resultado esperado para sistemas que utilizem o mecanismo proposto é que estes sejam mais fáceis de entender, adaptar, manter, estender e evoluir.

A motivação desse trabalho é o aparente fato de que, atualmente, a técnica de modelagem orientada a objetos é insuficiente para expressar propriedades transversais de um Sistema de forma fácil para compreensão humana. Outro fator motivante é a quantidade de interesses transversais existente em um Sistemas Multiagente e a relação entre o aumento desses interesses com o crescimento da complexidade dos sistemas.

Técnicas que modularizam requisitos tranversos em Sistemas de computação são projetadas, porém existe uma certa inércia quanto ao aprendizado destas técnicas. Essa inércia pode ser atribuída à sintaxe não-usual de linguagens de modelagem desses requisitos espalhados pelo código. Esse também é um fator que justifica o desenvolvimento desse trabalho, pois possibilita a inserção de código utilizando recursos de programação orientada a aspectos sem que o desenvolvedor necessite de familiaridade com a sintaxe de alguma linguagem de modularização de requisitos transversais.

No decorrer desta seção, serão detalhadas as decisões tomadas e os caminhos percorridos durante o desenvolvimento desse trabalho relativos à concretização de cada um dos objetivos específicos traçados. A seção 3.1 apresenta as características iniciais desejadas para a ferramenta e os requisitos principais que o mecanismo deve atender. A seção 3.2 trata de questões específicas de implementação da ferramenta proposta para os requisitos especificados na idealização da ferramentana seção sec:ideal. Porém, para que os requisitos do mecanismo pudessem ser especificados duas decisões de projeto foram tomadas *a priori*: o *framework* de desenvolvimento de Sistemas Multiagentes com o qual a ferramenta será compatível e o ambiente de desenvolvimento integrado para o qual a ferramenta será construída. O mecanismo visual de inserção de propriedades transversais é compatível com o *framework JADE*, abordado na seção 2.1.1, e construído para integrar o ambiente de desenvolvimento *Eclipse 3.0.1*, apresentado na seção 2.4.

Percebeu-se que a utilização do *framework JADE* poderia facilitar a construção dos agentes e que, pela forma em que foi concebido, as propriedades dos agente poderiam ser incorporadas via aspectos.

3.1 Requisitos do mecanismo

O projeto tem como um de seus objetivos a construção de um mecanismo de gerenciamento de propriedades transversais em Sistemas Multiagentes. O primeiro passo, nesse sentido, foi a definição de como essas propriedades deveriam ser adicionadas em Sistemas Multiagentes que utilizam o *framework* JADE. Como um *framework* tem um fluxo de execução predefinido e um conjunto de interfaces, a inserção de código da nova propriedade transversal deve referenciar pontos específicos do código para garantir o funcionamento do sistema.

Observou-se que uma funcionalidade desejável para o sistema deve ser a capacidade de acrescentar antes, durante ou depois de um comportamento adicionado no corpo do método *protected void setup()* de uma classe de agente. Partindo do princípio que as atividades de um agente são modeladas como comportamentos e são explicitamente adicionados no corpo do método *setup()*, um aspecto pode realizar as alterações de fluxo do código para que o agente realize um maior número de comportamentos inteligentes ou simplesmente incorpore novas regras de negócio.

O fluxo de execução do *framework JADE*, que gerencia os agentes instanciados com os serviços apresentados na seção 2.1.1, procura pelo método *protected void setup()* de cada agente. As instruções específicas que compõem as tarefas desejadas para o agente devem ser inseridas no corpo desse método. Sendo assim, considerou-se cada propriedade adicional como sendo um novo comportamento, definido também no *framework* em questão, inserido no método *protected void setup()* de cada agente como mostra a figura 3.1.

Cada propriedade transversal será modelada como um comportamento representado pela classe *Behaviour* de *JADE* e suas sub-classes. Como abordado na seção 2.3 o código é submetido ao recompositor aspectual (*Weaver*) e fragmentos de código são inseridos nessa fase. A figura 3.1 apresenta um esquema de inserção de código transversal entre os demais comportamentos do agente.

Pode-se definir então como requisito primário a adição, via mecanismo de in-

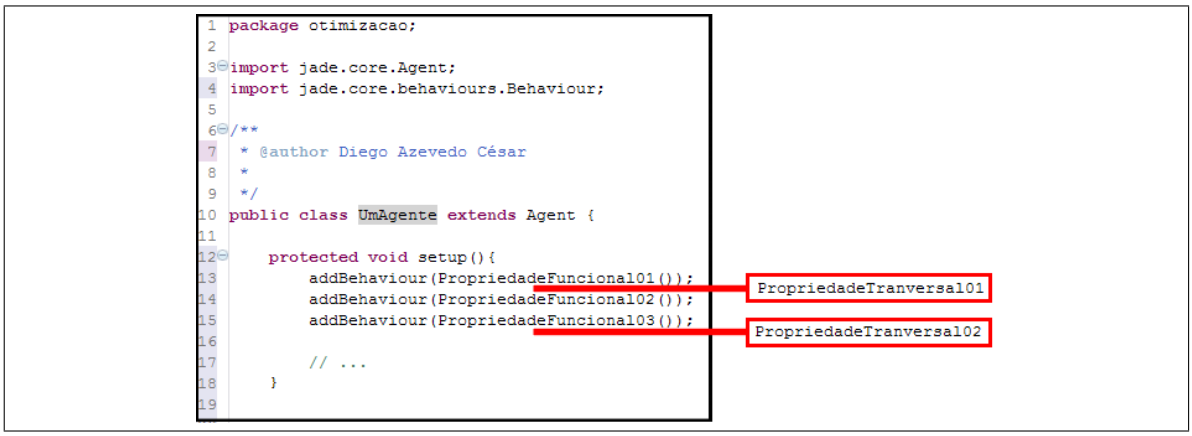


Figura 3.1: Exemplo de código após fase de Recomposição Aspectual (*Weaving*)

terface visual e recursos de programação orientada a aspectos, de comportamentos derivados da classe *jade.core.behaviours.Behaviour*.

Faz-se necessário, para estudo de comportamento dos agentes do sistema, que o mecanismo visual possibilite o gerenciamento das propriedades transversais inseridas no sistema. O objetivo é que uma determinada funcionalidade possa ser habilitada ou desabilitada em função da necessidade do usuário. Cenários de execução diferentes podem ser analisados com facilidade de alternância de comportamentos.

A terceira funcionalidade é a adição, por mecanismo visual, de trechos de código em outras classes do sistema que sejam complementares para a propriedade transversal adicionada. Nesse sentido, *AspectJ* oferece um amplo conjunto de recursos para a definição desses pontos, chamados pontos de junção. Além disso, os adendos permitem definir quando (antes, durante ou depois) e o que fazer no momento em que um desses pontos for executado. As opções de inserir propriedades em uma classe e suas subclasses, ou em uma família de classes, ou até mesmo uma família de métodos deve ser feita de forma gráfica na janela que trata de adição de trechos de código complementares.

A figura 3.2 apresenta o Diagrama de Uso da UML (BOOCH; RUMBAUGH; JACOBSON, 1999) gerado para auxiliar no levantamento de requisitos da ferramenta proposta neste trabalho. O único ator apresentado no diagrama é o Desenvolvedor,

porém esse mesmo ator pode representar toda a equipe de desenvolvimento, ou quaisquer responsáveis pelo desenvolvimento do Sistema Multiagente.

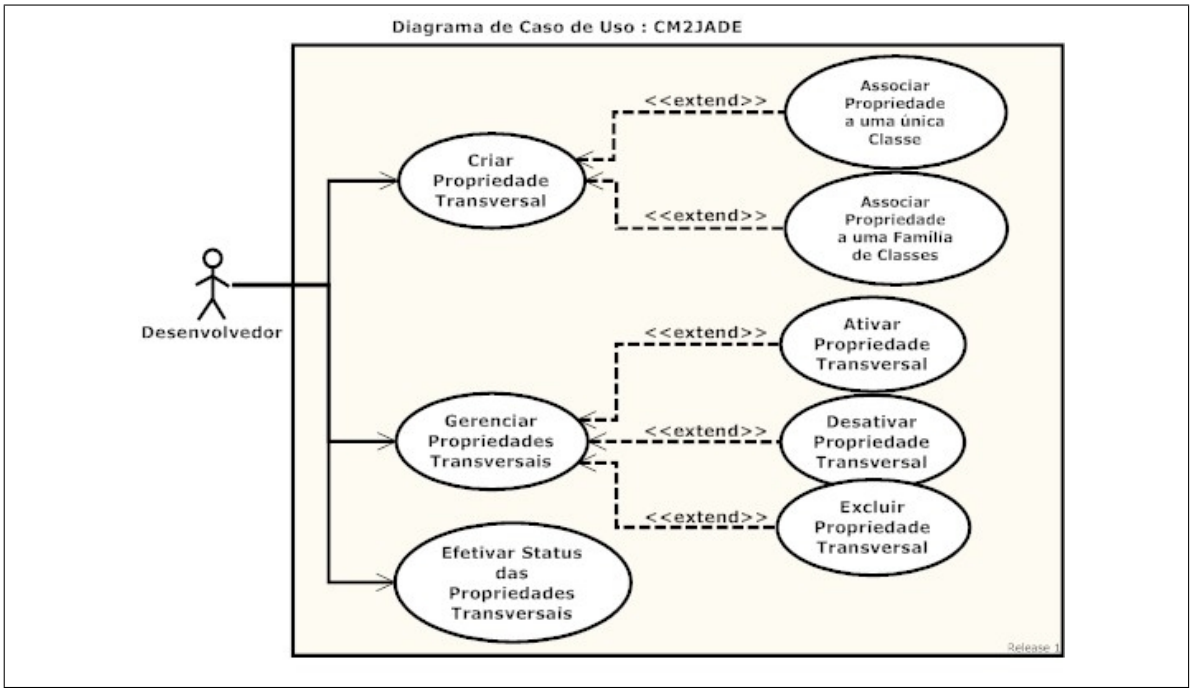


Figura 3.2: Diagrama de Caso de Uso para levantamento de requisitos da ferramenta proposta

Cada caso de uso segue um fluxo de atividades definido que envolve ações esperadas do desenvolvedor e do mecanismo. O caso de uso expandido (WAZLAWICK, 2004) mostra os passos de execução dos casos de uso no fluxo principal e os tratamentos de exceções quando estes são aplicáveis. A figura 3.3 mostra os casos de uso expandidos para o mecanismo idealizado.

É desejável que a ferramenta tenha fácil integração com o ambiente de desenvolvimento integrado. Portanto, a solução deve ser compatível com o Eclipse, que é a IDE escolhida neste trabalho.

Caso de Uso: Criar Propriedades Transversais

Fluxo Principal:

1 O Desenvolvedor seleciona a opção Criar Prop. Transversais
2 O Sistema abre a janela de Criação de Prop. Transversais
3 O Deselvolvedor informa o nome da Propriedade
4 O Deselvovedor informa o ponto do código a ser alterado
5 O Desenvolvedor informa o código a ser inserido no ponto especificado
6 O Desenvolvedor confirma os dados da propriedade
7 O Sistema persiste os dados da propriedade
8 O Sistema fecha a janela de Criação de Prop. Transversais

Caso de Uso: Gerenciar Propriedades Transversais

Fluxo Principal:

1 O Desenvolvedor seleciona a opção Gerenciar Prop. Transversais
2 O Sistema lê a quantidade de propriedades da lista de propriedades
3 Para cada propriedade da lista de propriedades
3.1 O Sistema reúne os parâmetros referentes à propriedade
4 O Sistema exibe a janela de propriedades, seus parâmetros e Status
5 O Desenvolvedor fecha a janela de propriedades

Tratamento de Exceções:

3a O Desenvolvedor deseja ativar uma Propriedade
3a.1 O Desenvolvedor deve informar a Propriedade Transversal a ser Ativada
3a.2 O Sistema verifica o status da propriedade no arquivo de propriedades
3a.3 O Sistema atualiza o status da propriedade para Ativada
3a.4 Retorna ao fluxo principal no passo 5

3b O Desenvolvedor deseja desativar uma Propriedade
3b.1 O Desenvolvedor deve informar a Propriedade Transversal a ser Desativada
3b.2 O Sistema verifica o status da propriedade no arquivo de propriedades
3b.3 O Sistema atualiza o status da propriedade para Desativada
3b.4 Retorna ao fluxo principal no passo 5

3c O Desenvolvedor deseja excluir uma Propriedade
3c.1 O Desenvolvedor deve informar a Propriedade Transversal a ser Excluída
3c.2 O Sistema verifica a existência da propriedade no arquivo de propriedades
3c.3 O Sistema exclui a propriedades da lista de propriedades
3c.4 Retorna ao fluxo principal no passo 5

Caso de Uso: Efetivar Status das Propriedades Transversais

Fluxo Principal:

1 O Desenvolvedor seleciona a opção Efetivar Status das Prop. Transversais
2 O Sistema apaga todos os aspectos de propriedades transversais existentes
3 O Sistema lê os parâmetros armazenados na lista de propriedades
4 O Sistema seleciona a lista de propriedades ativas
5 Para cada propriedade da lista de propriedades ativas
5.1 O Sistema seleciona os parâmetros de criação referentes à propriedade
5.2 O Sistema associa os parâmetros de criação aos seus templates de transfomação
5.3 O Sistema gera os arquivos de aspecto

Figura 3.3: Casos de Uso expandidos com fluxo principal e tratamento de exeções para a ferramenta proposta

3.2 Decisões de projeto do Sistema

Uma vez levantadas as funcionalidades desejadas para o sistema computacional em questão, a próxima fase foi o desenvolvimento do projeto. As decisões estratégicas e tecnológicas para a construção da ferramenta basearam-se na integração com o ambiente de desenvolvimento, usabilidade e persistência.

Para desenvolvimento do projeto utilizou-se o ambiente de desenvolvimento integrado *Eclipse 3.0.1*. Para escrever esse *plugin* foram utilizados recursos como *Standard Widget Toolkit* (SWT), *Swing*, *Plugin Developer Environment* (PDE), JFace e *Java Developer Toolkit* (JDT).

O SWT é um conjunto de componentes gráficos usado para construir botões, imagens, tabelas e *labels*. JFace é um conjuntos de classes utilizado para construir menus e barras de ferramentas, janelas de diálogo, fontes, imagens, arquivos de texto e classes base para *wizards*. As classes que compõem o PDE ajudam com a manipulação de dados e *wizards*.

Para construção das interfaces gráficas, optou-se pelos recursos contidos em *org.eclipse.jface.** e *org.eclipse.swt.**. A construção de caixas de diálogo de entrada e de mensagem, de erro, bem como janelas de assistência (*wizards*) é facilitada pelo pacote *JFace* e objetos gráficos como botões, campos selecionáveis, diferentes tipos de *layouts* podem ser utilizados com a declaração do pacote SWT.

Como opção de persistência, optou-se pelo uso de arquivos escritos na Linguagem de Marcação Extensível - XML (do inglês *Extensible Markup Language*)(BRAY et al., 2006) que armazenam os dados informados pelo usuário das propriedades transversais de agentes. *A Posteriori* esses arquivos servem como parâmetros de entrada para geração dos aspectos escritos em *AspectJ* que efetivaram as propriedades transversais no sistema. Esses arquivos XML têm utilidade, também, para a janela de gerenciamento de funcionalidades transversais prevista na seção 3.1.

Para conversão dos arquivos XML em arquivos de Aspectos de sintaxe correta, utilizou-se um template XSL que associa cada *tag* do arquivo ao seu devido lugar

no código do arquivo *<nomearquivo>.aj*. Para associar o arquivo XML ao *template* XSL é necessária a utilização de um *parser* para esse fim específico. Um esquema da conversão de um arquivo XML gerado por uma aplicação *A* em dados formatados para a aplicação *B* utilizando um transformador XSL é apresentado na figura 3.4.

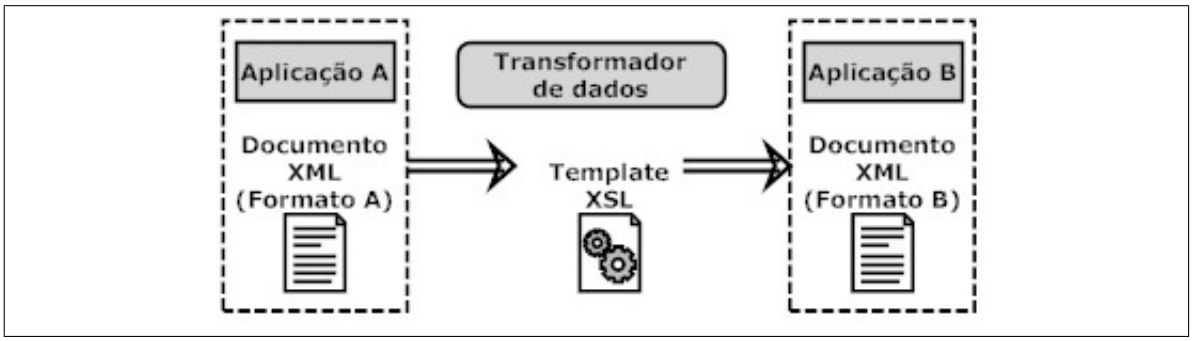


Figura 3.4: Esquema de transformação de dados entre aplicativos com o uso de arquivo XML e transformador XSL

Para facilitar a integração da ferramenta ao ambiente de desenvolvimento, nesse caso, o Eclipse, optou-se por desenvolvê-la como um *plugin*, que, após empacotado, é reconhecido automaticamente pela IDE quando armazenado em uma pasta específica do diretório de instalação do ambiente. Além disso, é desejável que o *plugin* utilize recursos da API Java PDE e não crie dependência com nenhum recurso que envolva direitos de uso e licença.

Sobre as interfaces gráficas, optou-se que o acesso do usuário às janelas de adição de propriedades transversais, gerenciamento da lista de propriedades transversais e adição de fragmentos de códigos de propriedades seja realizado por acesso a botões que deverão estar localizados na barra de ferramenta. Esse recurso agiliza o acesso às janelas, que recebem diversos acessos no caso de estudo de comportamento do sistema com alternância de comportamentos.

A não incorporação dos pacotes que caracterizam o *framework JADE* à biblioteca do *plugin* foi uma decisão de projeto que visa o desprendimento da ferramenta à determinada versão do JADE e também não aumentar, desnecessariamente o tamanho do arquivo .jar que carrega as classes da ferramenta após serem empacotadas. A

mesma decisão de não associação de determinadas tecnologias com o *plugin* desenvolvido foi tomada quanto a versões de *AspectJ* e plugins de aspectos para Eclipse como AJDT.

Capítulo 4

CM2JADE : Ferramenta de adição de propriedades transversais em SMAs

"*The best way to have a good idea is to have lots of ideas*"(Linus Pauling).

Esta seção descreve a construção e o funcionamento da ferramenta proposta na seção 1.3 entitulada CM2JADE (*Concern Management to Java to Java Agent Development*). Questões sobre a forma como as tecnologias envolvidas foram empregadas e contribuiram para o desenvolvimento serão abordadas em um nível mais detalhado. Uma explanação dos diversos subprodutos gerados durante o desenvolvimento como interfaces gráficas, mecanismo de persistência, organização dos pacotes, móulo conversor e analisador de dados, arquivos de Aspectos escritos em linguagem *AspectJ* são discutidos no decorrer da seção.

Entende-se que o CM2JADE deve englobar os requisitos levantados na seção 3.1. Considera-se que (i) uma classe de agente deve herdar a classe *jade.core.Agent*, implementar/sobrepor o método *protected void setup()* declarado na classe *Agent*. (ii) O ponto de junção do Aspecto criado para inserção de propriedades transversais deve ocorrer explicitamente no corpo do método *setup()* e apontar para algum método de assinatura *addBehaviour(Behaviour)*, definido em sub-classe de *jade.core.Agent*.

Como exemplo tem-se uma classe de agente na qual é desejado adicionar uma propriedade transversal após o monitoramento da caixa de mensagens. Uma classe de agente recebe, por herança, os atributos e métodos definidos na classe *Agent*. Para esse exemplo específico deve existir no corpo do método *protected void setup()*, uma chamada de *addBehaviour(new MonitorarMensagem())*, onde *MonitorarMensagem* é uma classe que herda *jade.core.behaviours.Behaviour* e implementa os métodos abstratos declarados nessa mesma classe. A chamada do método de adição de comportamento definido na classe *Agent* faz-se necessária e a assinatura do método é um dos parâmetros do ponto de junção do aspecto gerado.

O esquema de funcionamento do CM2JADE é mostrado na Figura 4.1. Nela, são apresentados seus componentes, e a interação entre as classes que representam as interfaces gráficas da ferramenta, o mecanismo de persistência, e o módulo gerador de códigos, até o produto final da ferramenta que é um conjunto de aspectos gerados. O mecanismo que persiste os dados das propriedades transversais em um conjunto de arquivos XML, e arquivos de propriedades. Com a utilização destes o CM2JADE faz a geração dos aspectos solicitados ou os remove de atividade, segundo o *status* declarado no arquivo de propriedades. Esses arquivos são mantidos atualizados pelas interfaces para gerenciar os aspectos que serão gerados a desejo do usuário. Esses aspectos são o produto final de modularização das propriedades dos agentes e serão discutidos no decorrer da seção.

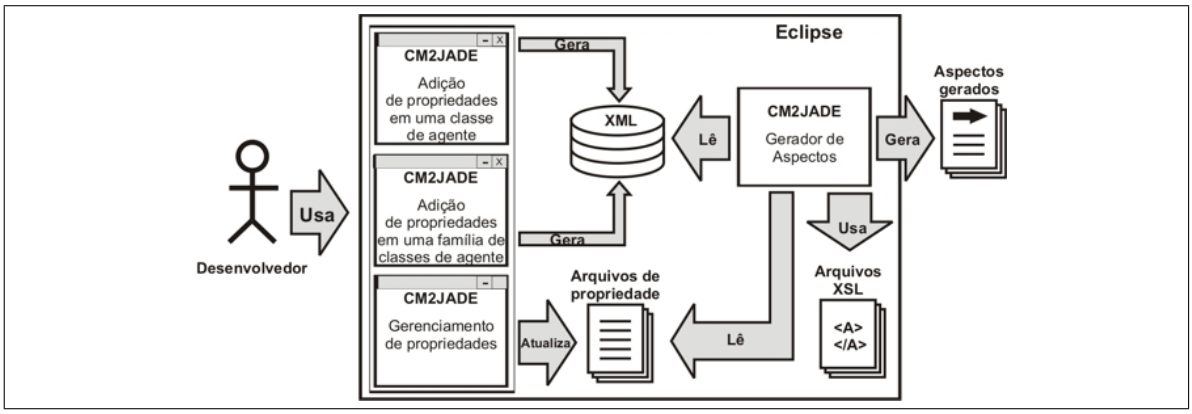


Figura 4.1: Esquema de funcionamento do CM2JADE

Em se tratando da ferramenta construída, os recursos gráficos utilizados foram botões de acesso às janelas com tabelas ou janelas de assistência. Os botões estão localizados na barra de ferramentas da área de trabalho do Eclipse. Uma vez que o ambiente de desenvolvimento Eclipse é iniciado, a ferramenta está pronta para uso. A figura 4.2 mostra os botões adicionados e parte da barra de ferramentas.

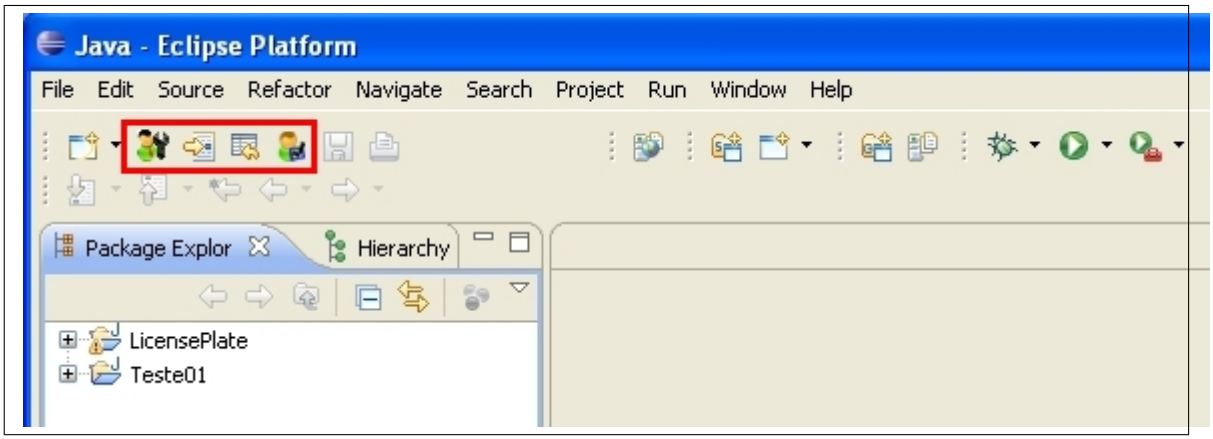


Figura 4.2: Botões de acesso às janelas de gerenciamento de propriedades transversais do CM2JADE

Os botões apresentados na figura 4.2 representam respectivamenta e a adição de propriedades transversais em uma única classe de agente, o gerenciamento e visualização das propriedades transversais declaradas, adição de propriedades transversais em uma família de classes de agente e o botão acionado para efetivar as propriedades ativas. Cada um desses recursos serão abordados no decorrer da seção e são apresentados nas figuras 4.3, 4.6 e 4.7 respectivamente.

A figura 4.3 representa uma janela de assistência com campos editáveis nos quais o usuário fornece os parâmetros e diretivas para a construção da propriedade transversal desejada. O primeiro parâmetro é o *Container* em que os arquivos de persistência serão criados. O segundo parâmetro diz respeito às diretivas de nomeação de arquivos. O campo de *Nome de propriedade* existe para que o usuário informe um nome intuitivo que referencie a propriedade adicionada. No campo de Tipo de *link*, o usuário informa ao *plugin*, através da interface, se a execução do código-fonte presente no aspecto será executada antes, durante ou depois do conjunto de junção. A opção

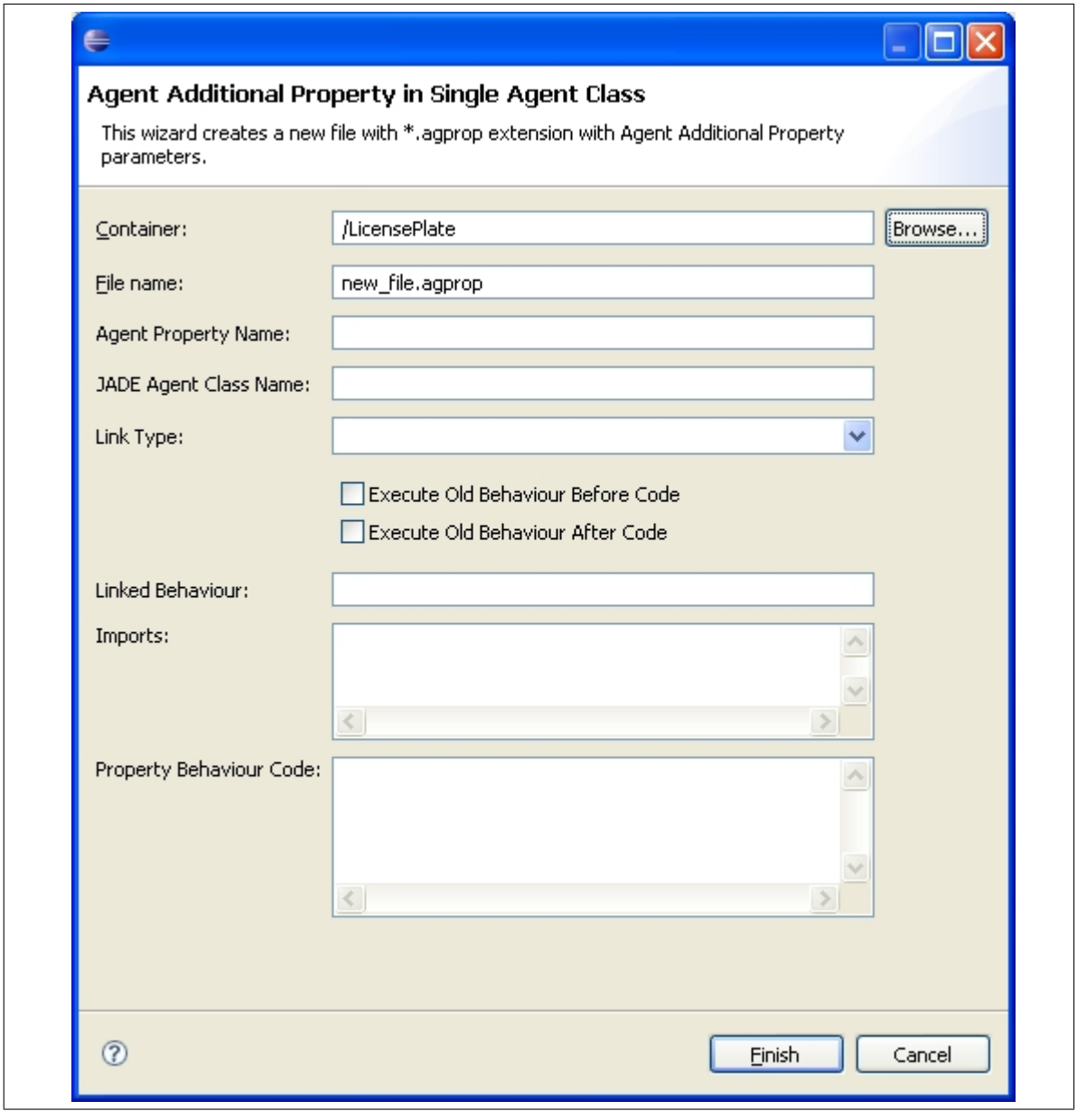


Figura 4.3: Interface gráfica de adição de propriedades transversais

de executar o código durante a execução possibilita ao usuário substituir o comportamento apontado pelo conjunto de junção. Caso não seja de interesse sobrescrever o método, o usuário pode marcar a opção de executar o comportamento afetado pelo conjunto de junção antes ou após as instruções inseridas. Uma terceira opção é a de especificar o ponto do código informado pelo usuário que deseja-se executar o comportamento apontado pelo conjunto de junção. Neste caso o desenvolvedor escreve no código a ser inserido a instrução *ExecuteLinkedBehaviout()* com os parâmetros esperados pelo método. O próximo parâmetro, o *Linked Behaviour* representa o tipo de comportamento que será atingido pelo aspecto. Se o usuário deseja adicionar uma propriedade após a chamada do método *addBehaviour(new UmComportamento())*, o usuário informa em *JADE Agent Style Point* a opção *after call of* e em *JADE Agent Class Point* o nome do comportamento, ou seja, *UmComportamento()*. Em seguida o desenvolvedor informa as bibliotecas de aplicação que deseja importar para escrever o código. O último campo de edição tem a função de capturar o código fonte que será executado quando o novo comportamento for efetivado.

O resultado da confirmação do usuário após o fornecimento dos parâmetros nessa interface é a geração de um arquivo escrito em linguagem XML contendo os dados dos campos editáveis separados em *tags* de marcação. Um exemplo típico de arquivo XML gerado após o uso da interface de adição de propriedades transversais é apresentado na figura 4.4(a). O arquivo trata a adição da propriedade Mobilidade em agentes da classe *FileAnalyserAgent*.

Cada unidade de marcação (*tag*) do arquivo armazena itens importantes para a conversão dos dados fornecidos pela janela gráfica para um aspecto sintaticamente correto. Os ítens do arquivo são:

- Cabeçalho. As linhas 01 e 02 do arquivo XML apresentado na figura 4.4(a) declaram informações de codificação de texto e fazem referência à pasta de estilo utilizada.
- Campo *AspectConcern*. Essa *tag* engloba todas as outras do arquivo e contém propriedades como o nome da propriedade transversal em questão. Essa pro-

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="LO2AgentClassSchema.xsl" type="text/xsl"?>
<AspectConcern name="Mobility">
    <Package>management.properties</Package>
    <Imports>
        <Import>jade.core.Agent</Import>
        <Import>jade.core.behaviours.*</Import>
        <Import>jade.lang.acl.ACLMessage</Import>
        <Import>jade.core.Location</Import>

    </Imports>
    <Classname>FileAnalyserAgent</Classname>
    <Advice>after</Advice>
    <Advicereturn></Advicereturn>
    <AdviceproceedBefore></AdviceproceedBefore>
    <AdviceproceedAfter></AdviceproceedAfter>
    <BehaviourLink>AnalysingDocuments</BehaviourLink>

    <Source>
        ACLMessage reqMess=agent.receive();
        if(reqMess!=null)
        {
            String req_Content= reqMess.getContent();
            Location dest = MessageAnalyser.getLocation(req_Content);
            agent.doMove(dest);
        }
    </Source>
</AspectConcern>
```

(a) Um arquivo XML gerado para a propriedade transversal de mobilidade de agentes

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text"/>
    <xsl:template match="/">
        package <xsl:value-of select="//Package"/>;

        <xsl:apply-templates select="//Import"/>

        public aspect
        <xsl:value-of select="AspectConcern/@name"/>Concern {

        pointcut pointcut<xsl:value-of select="AspectConcern/@name"/>
        (Behaviour msg, <xsl:value-of select="//Classname"/> agent):
        call(void Agent.addBehaviour(Behaviour)) &amp;&amp;
        this(agent) &amp;&amp;
        withincode(protected void setup()) &amp;&amp;
        args(msg);

        <xsl:apply-templates select="//Advicereturn"/><xsl:text> </xsl:text>
        <xsl:apply-templates select="//Advice"/>
        (Behaviour msg, <xsl:value-of select="//Classname"/> agent) :
        pointcut<xsl:value-of select="AspectConcern/@name"/>(msg, agent){
        if(msg instanceof <xsl:apply-templates select="//BehaviourLink"/>){
            <xsl:apply-templates select="//AdviceproceedBefore"/>
            <xsl:apply-templates select="//Source"/>
            <xsl:apply-templates select="//AdviceproceedAfter"/>
        }
        }

        }
    </xsl:template>
```

(b) Arquivo XSL utilizado no processo de conversão de dados em XML para código *AspectJ*

Figura 4.4: Estrutura de arquivo XML e *template* XSL envolvidos na geração de aspectos pelo CM2JADE

priedade serve como identificador único de uma propriedade e é utilizada na lógica do sistema para habilitar ou desabilitar uma propriedade.

- *Package*: menciona o pacote onde o arquivo *AspectJ* será gerado posteriormente.
- *Import*: tem a função de declarar os pacotes utilizados para reescrever o código da propriedade.
- *Classname*: representa o nome da classe de agente que será referenciada pelo aspecto gerado. No caso da figura 4.4(a) existe uma classe de agente *FileAnalyserAgent* em que se tem o interesse em alterar o comportamento. Portanto, a *tag* armazena o nome da classe *<Classname>FileAnalyserAgent</Classname>*.
- *Advice*. Essa *tag* pode receber os valores *before*, *around* ou *after*, dependendo do fluxo de interesse que cerca o local do código apontado pelo conjunto de junção.
- *Advicereturn*. Uma vez que se utilize o adendo *around*, o método que pretende-se modificar o comportamento pode retornar um valor do tipo primitivo ou um objeto. Essa *tag* é utilizada para que o aspecto seja escrito sintaticamente correto retornando o valor esperado na classe de Agente. No caso do método substituído ter retorno do tipo *void*, a *tag* é declarada sem valor algum.
- *AdviceProceedBefore* e *AdviceProceedAfter*. Ao utilizar o recurso adendo *around*, um método é substituído. O método *proceed()* (prosseguir), que é uma palavra-chave do AspectJ, executa o método substituído. As *tags AdviceProceedBefore* e *AdviceProceedAfter* tratam a inserção da chamada de *proceed()* no início ou no fim do corpo do código do adendo.
- *BehaviourLink*. Essa *tag* representa o comportamento apontado pelo conjunto de junção que é chamado no método *setup()* da classe apontada em *<Classname>*. *Source*. Representa o código-fonte que será executado quando o aspecto realizar a inserção de código.

Uma vez gerado o arquivo XML com as informações corretas em cada *tag* o usuário efetiva as mudanças e então, por um processo de transformação que envolve

o template XSL, um arquivo de aspectos é gerado. Um exemplo da transformação XML $\overset{XSL}{\rightarrow}$ código *AspectJ* é apresentado na figura 4.9.

```
1  package invokatron.aspecto;
2
3  import jade.core.Agent;
4  import jade.core.behaviours.*;
5  import jade.lang.acl.ACLMessage;
6  import jade.core.Location;
7
8  public aspect MobilityConcern {
9
10     pointcut pointcutMobility(Behaviour msg, FileAnalyserAgent agent):
11         call(void Agent.addBehaviour(Behaviour)) &&
12         this(agent) &&
13         withincode(protected void setup()) &&
14         args(msg);
15
16     after(Behaviour msg, FileAnalyserAgent agent) :
17         pointcutMobility(msg, agent){
18         if(msg instanceof AnalysingDocuments){
19
20             ACLMessage reqMess=agent.receive();
21             if(reqMess!=null)
22             {
23                 String req_Content= reqMess.getContent();
24                 Location dest = MessageAnalyser.getLocation(req_Content);
25                 agent.doMove(dest);
26             }
27
28         }
29     }
30
31 }
```

Figura 4.5: Código *AspectJ* resultante da transformação de um arquivo XML

A linguagem de estilo extensível permite que a informação armazenada em um formato seja associada com os elementos de outro formato e através de um mecanismo de análise de estrutura gramatical produza um documento em outro formato. Para cada tipo de transformação é necessário um XSL específico. Uma fonte de dados única, neste caso o arquivo XML pode ser convertida em diversos formatos pela simples submição a um parser associado a um XSL específico que descreva o formato final desejado. A figura 4.9 mostra, em destaque, as *tags* do arquivo XML da figura 4.4(a) submetida ao estilo XSL do arquivo apresentado na figura 4.4(b). Percebe-se que uma *tag* XML, como por exemplo *AspectConcern*, é utilizada mais de uma vez para a especificação do *template* XSL. Nesse caso, a *tag AspectConcern* foi utilizada para compor o nome do aspecto, o nome do conjunto de junção, e repetida na declaração do adendo para especificar a qual conjunto de junção o adendo se refere. Da mesma forma a *tag Classname* é utilizada em dois pontos no código gerado, nas linhas 10 e

16, representados pela palavra *FileAnalyserAgent*.

O aspecto *MobilityConcern*, apresentado na figura 4.9, contém um conjunto de junção e um adendo. O conjunto de junção *pointcutMobility* captura dois parâmetros, *msg* e *agent*, que representam o comportamento afetado pelo aspecto e a classe de agente em questão respectivamente. O objetivo desse conjunto de junção é apontar para todas as chamadas do método *addBehaviour(obj)*, onde *obj* é um objeto do tipo *Behaviour*. A expressão da linha 12 declara que o aspecto só tem interesse em atingir as chamadas que ocorrem dentro da classe do tipo do objeto *agent*, que nesse caso são classes do tipo *FileAnalyserAgent*. Está declarado na linha 13 que só são objetivadas as chamadas que ocorrerem dentro do método *protected void setup()*. O parâmetro *args* é enviado, na linha 14, ao adendo para um teste de tipo, na linha 18.

O fluxo dos dados informados pelo usuário, iniciado no preenchimento dos campos editáveis da interface gráfica, persistidos em XML, e posteriormente transformados em aspectos abordam a funcionalidade levantada na seção 3.1. Uma vez que é possível adicionar propriedades transversais e gerar código em *AspectJ* para o domínio abordado, é necessária a existência de um mecanismo que liste os dados relevantes para manutenção das propriedades e que permita habilitar ou desabilitar determinada propriedade. Esse mecanismo é previsto como funcionalidade desse projeto e está disponível na barra de ferramentas do *plugin*, e é apresentado na figura 4.6.

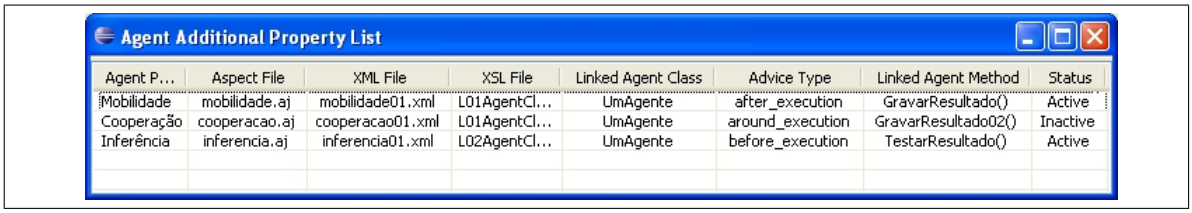


Figura 4.6: Interface gráfica de gerenciamento de propriedades transversais

A janela gráfica apresentada na figura 4.6 mostra as propriedades transversais adicionadas ao Sistema Multiagente, os arquivos necessários para sua manutenção e o *status* de cada propriedade. Para ativar ou desativar uma determinada propriedade, basta selecionar a propriedade e escolher o botão *Enable* ou *Disable*. O *status* de

uma propriedade é armazenado em um arquivo de extensão *agprop*. A utilidade dessa janela gráfica é possibilitar a alternância de comportamentos transversais declarados. Cada propriedade assume o status de habilitada ou desabilitada, segundo a vontade do usuário. Sendo assim, a execução do sistema pode ser estudada em 2^n cenários direrentes, onde n é a quantidade de propriedades transversais adicionadas com o CM2JADE.

As informações apresentadas pelo gerenciador de propriedades transversais são apresentadas na forma de colunas. Cada coluna pode ser ordenada de forma alfabética crescente ou decrescente. Os ítens da lista de propriedades são:

- Nome da propriedade de Agente. É desejável que o nome da propriedade seja intuitivo para o usuário.
- Arquivo *AspectJ*. Cada propriedade habilitada e efetivada contém um arquivo de aspecto correspondente escrito em linguagem *AspectJ*.
- Arquivo XML. Os dados informados pelo usuário e utilizados para gerar os arquivos em *AspectJ* são armazenados em arquivos XML. Existe um arquivo XML de nome único para cada propriedade. Como uma propriedade pode ter um aspecto principal e arquivos de aspectos secundários com fragmentos espalhados de código por outros agentes que dizem respeito a uma mesma propriedade um índice numérico foi acrescido no fim do nome do arquivo.
- Arquivo XSL. Cada XML gerado associado a um arquivo XSL específico gera um arquivo de aspectos. A adição de propriedades transversais e de fragmentos de propriedades transversais usam diferentes arquivos XSL. Essa informação é apresentada ao usuário nessa janela gráfica.
- Classe de agente referenciada. Como abordado anteriormente, cada propriedade transversal adicionada menciona uma classe de agente. Essa classe será alterada após a fase de *weaving*.
- Tipo de Adendo. Essa coluna apresenta o tipo de inserção de códido utilizada

pelo aspecto. Os valores válidos para essa coluna são *before call*, *around call* e *after call*.

- Comportamento afetado. É apresentado o nome do método que é apontado pelo conjunto de junção.
- Status da Propriedade.

Quando uma propriedade de agente é criada, com o uso da ferramenta, seu *status* inicial é de habilitada. Porém, o fato de que uma propriedade está habilitada, não significa que o aspecto correspondente a ela está criado. Os arquivos de aspectos são criados quando o usuário efetiva as habilidades transversais na barra de trabalho da IDE.

Pode ocorrer a necessidade, para que uma propriedade transversal englobe todas as ações que a caracterizam, da inserção de fragmentos de código em outros agentes. Por exemplo, se uma propriedade de agente depende de troca de mensagens entre agentes de papéis diferentes o comportamento inserido não só deve fornecer ao agente de papel A a capacidade de enviar mensagens para o agente B, mas também ao agente B a função de monitorar e entender a solicitação do agente A. Portanto, para uma única propriedade, pode haver a necessidade de construção de mais de um aspecto ou de que um único conjunto de junção tenha mais de um adendo.

Para promover a inserção complementar de código em agentes, para tratar de uma propriedade transversal já declarada, o CM2JADE apresenta uma terceira interface gráfica. Essa interface tem a função de adicionar fragmentos de código em quaisquer agentes declarados num projeto. É necessário o vínculo explícito com uma propriedade já existente.

O propósito dessa interface é ser a mais genérica possível para englobar os casos de inserção de códigos transversais que a interface da figura 4.3 não engloba. Na interface de adição de propriedades, por visar comportamentos específicos que sejam chamados no corpo do método *setup()* de uma única classe, grande parte do poder da linguagem *AspectJ* não é aproveitado, como abordar métodos de um conjunto de

classes. Para aproveitar esses recursos disponíveis na linguagem *AspectJ* e não limitá-los pelo uso da ferramenta a interface de adição de fragmentos, apresentada na figura 4.7, e rotinas que envolvem a lógica de construção de aspectos a partir dela foram idealizadas no projeto.

A estrutura dos campos editáveis da janela gráfica apresentada na figura 4.7 é semelhante a primeira interface gráfica, figura 4.3. Isso acontece porque a finalidade das duas interfaces é semelhante: inserir código em pontos específicos de determinados métodos para classe(s) de agente(s). A diferença é que a interface de adição de propriedade vincula o conjunto de junção ao método de adicão de comportamento(*addBehaviour*) implementado na classe *Agent*. Já o propósito geral dessa interface, figura 4.7, é adicionar em um grupo de classes, especificados através de um filtro de nomes, na chamada de um determinado método que pode ocorrer em uma família de métodos informados pelo usuário. Da mesma forma que na primeira interface, é necessário informar o tipo de adendo que se deseja gerar. Os ítens editáveis da interface são:

- Nome da propriedade de agente. Como abordado anteriormente essa interface é para inserção de código referente a interfaces já declaradas no sistema com uso da tela de adição de propriedades transversais. Esse campo serve para vincular o aspecto gerado a um nome propriedade. Se essa mesma propriedade for desabilitada, todos os aspectos gerados referentes a ela não serão construídos, e portanto, não executados.

- Nome da Classe ou das Classes. No caso dos agentes a serem apontados pelo aspecto serem de uma classe única o campo deve receber o nome da classe. Caso os agentes sejam de várias classes deve-se obter um filtro de nomes que represente somente aquele conjunto de classes. Por exemplo, em um sistema em que tem-se as classes *FileAnalyserAgent*, *FileCounterAgent*, *FileTransferAgent*, *FileCreatorAgent*, *PersistenceAgent* e *DirectoryCreatorAgent*, e pretende-se modificar algum comportamento dos agentes manipuladores de arquivo, deve-se especificar nesse campo a expressão *File*Agent*.

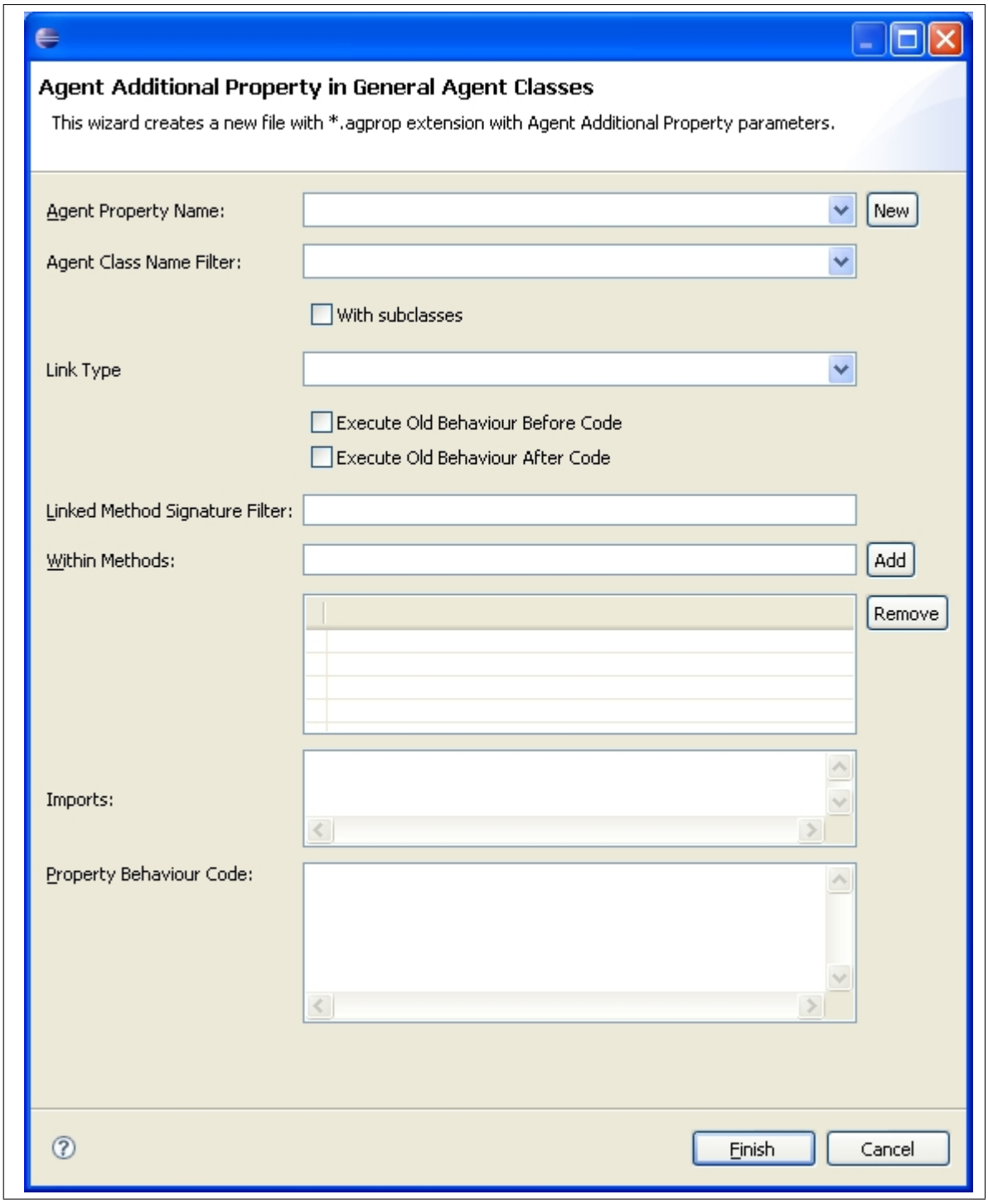


Figura 4.7: Interface gráfica de adição fragmentos de propriedades transversais

- Opção de marcação de subclasses. Caso o interesse seja na classe e nas subclasses de um agente, por exemplo em *DirectoryCreatorAgent*, o mecanismo de geração de aspectos insere no campo de nome da classe o símbolo '+', que especifica em AspectJ a classe de nome especificado e suas subclasses. Nesse caso o filtro seria *DirectoryCreatorAgent+*

- O método para o qual o aspecto vai apontar e mudar o comportamento a redor. Diferentemente da interface da figura 4.3 que recebe o argumento do método *addBehaviour*, esse campo deve receber a assinatura completa do método de interesse.

- Os itens de biblioteca de *software* a serem importados.

- O corpo dos métodos que serão investigados. O método apresentado no item anterior pode ser chamado em diversos lugares no código, porém nem todos podem compor o conjunto de interesses do comportamento declarado. Esse campo tem a função de recolher do usuário os métodos que devem ser analisados procurando o método cuja assinatura deve ser informada pelo usuário no item anterior.

- O código a ser executado. Essa janela serve para que o usuário informe o conjunto de instruções que deseja que seja realizado quando um ponto do código atender as restrições do conjunto de junção.

Uma vez que são confirmados os dados solicitados pela interface, um arquivo XML semelhante apresentado na figura 4.4(a) é gerado. Posteriormente, quando o usuário efetivar as mudanças realizadas com o uso do CM2JADE, um aspecto será gerado para adicionar as mudanças no momento de execução do sistema.

Abordadas as funcionalidades e características da ferramenta perante o usuário, a figura 4.8 apresenta o diagrama de classes da UML (BOOCH; RUMBAUGH; JACOBSON, 1999) da ferramenta em alto nível. São especificados os pacotes e classes do sistema e alguns métodos implementados nas classes.

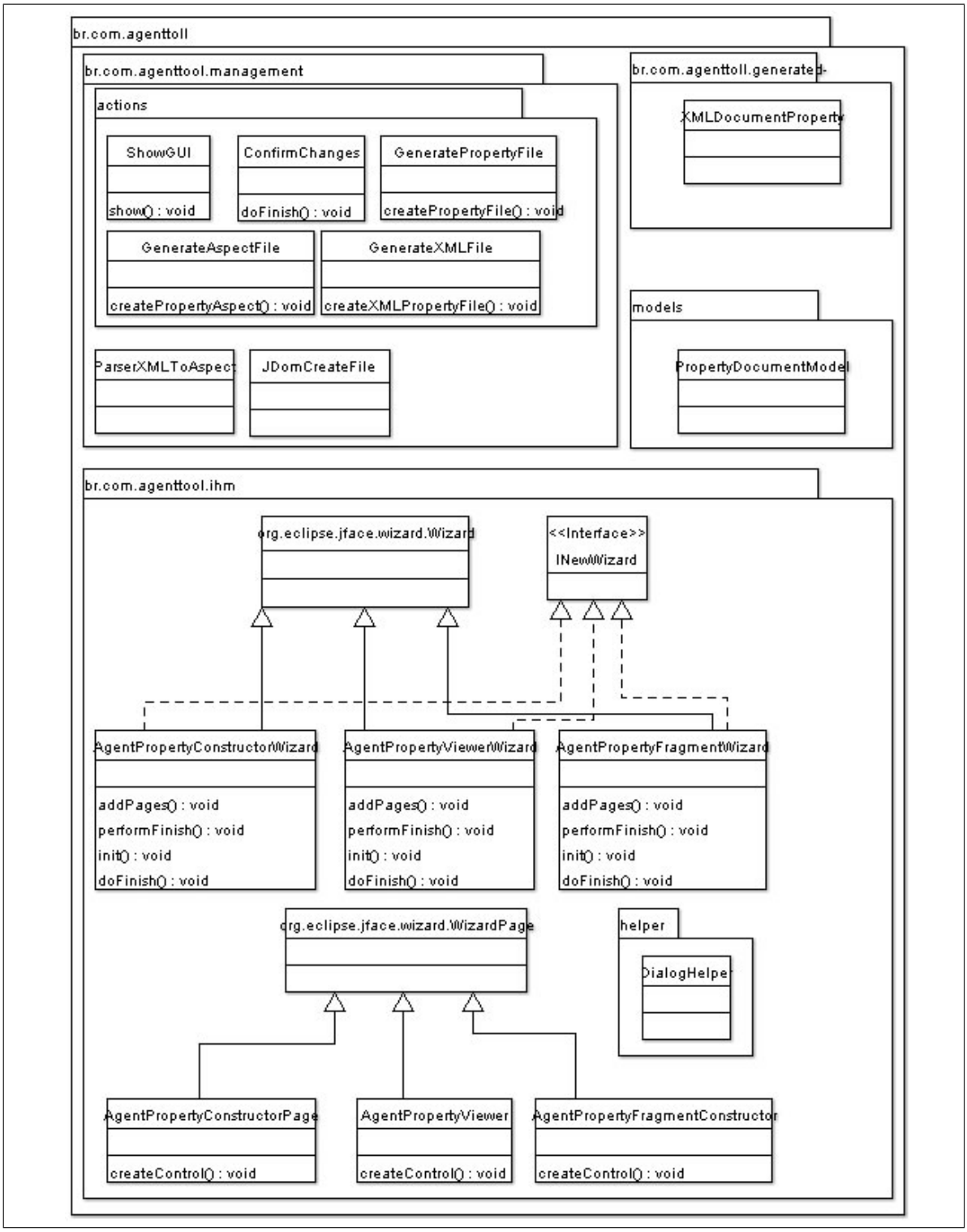


Figura 4.8: Diagrama de classes do CM2JADE (alto nível)

Os arquivos XML gerados, os arquivos template XSL, os aspectos e os arquivos *.properties* são armazenados no pacote *br.com.agenttool.generated*. As classes que representam as interfaces do sistema ficam no pacote *br.com.agenttool.ihm* e as regras de negócio para geração dos arquivos de aspectos foram empacotados em *br.com.agenttool.management*.

Essa seção apresentou as funcionalidades da ferramenta proposta nesse trabalho, o processo de geração dos arquivos envolvidos na adição de propriedades transversais em sistemas Multiagente e a organização de classes da ferramenta. As seções 4.1 e 4.2 discutem os benefícios da ferramenta proposta e suas limitações.

4.1 Benefícios

Um Sistema Multiagente engloba uma série de características intrínsecas e extrínsecas. Essas características são aprendizado, cooperação, mobilidade, concorrência, conhecimento, interação, autonomia e adaptação, dentre outros. Como abordado como problemática desse trabalho, na seção 1.1, construir um agente contendo esses módulos é uma tarefa difícil e requer gerenciamento de diversos módulos do sistema. Outra característica desses requisitos intrínsecas e extrínsecas é a distribuição dos mesmos pelo sistema.

A ferramenta construída ajuda no entendimento e manutenção do sistema, dado que se um ou mais requisitos que compõem o corpo do agente puderem ser isolados com o uso da mesma, o sistema diminui o índice de espalhamento e entrelaçamento de código pelo uso de recursos de programação orientada a aspectos.

A automatização dos aspectos proporcionada pelo CM2JADE é importante para que a mesma tenha ciclo de uso durável. Como existe uma certa inércia em aprender novas técnicas para modularização de requisitos transversais e poucos usuários familiarizam-se com a sintaxe de *AspectJ* o uso de aspectos para resolver problemas de requisitos transvesais em Sistemas Multiagentes diminuiria caso a ferramenta não se trata a adição de aspectos de forma gráfica.

A possibilidade do uso de máscaras para filtrar nomes de classes favorece o uso do mesmo código de aspecto para mais de uma classe no sistema. Também é possível abordar o mesmo comportamento de uma classe e suas subclasses especificando no campo editável de assinatura do método. A tabela 4.1 apresenta exemplos de assinaturas de métodos e os métodos das classes especificados. É importante notar que os aspectos podem ser gerados para desempenhar comportamentos diferentes para métodos de mesmo nome, porém com variação de argumentos, modificadores de acesso ou tipos de retorno.

Assinatura do método	**Métodos afetados**
public void Analyser.doAction()	O método *doAction()* da classe *Analyser* que tem acesso *public*, retorno *void* e não tem argumentos
public void Analyser.*()	Todos os métodos da classe *Analyser* que tem acesso *public*, retorno *void* e não têm argumentos
public * Analyser.*()	Todos os métodos da classe *Analyser* que tem acesso *public* e não têm argumentos
public * Analyser.*(..)	Todos os métodos da classe *Analyser* que têm acesso *public* e com quaisquer tipos de argumentos
*** Analyser.*(..)**	Todos os métodos da classe *Analyser* com quaisquer tipos de argumentos
*** Analyser+.*(..)**	Todos os métodos da classe *Analyser* e suas subclasses com quaisquer tipos de argumentos
*** Analyser.add*Listener (EventListener+)**	Quaisquer métodos cujo nome inicia com *add* e termina com *Listener* da classe *Analyser* com argumento que sejam da classe *EventListener* ou suas subclasses

Tabela 4.1: Exemplo de assinaturas de métodos

O estudo experimental do comportamento do sistema alternando entre propriedades é importante para determinadas áreas. Em caso de simulações que envolvem comportamentos sociais, resultados podem ser medidos ao variar o grau de concorrên-

cia de determinados indivíduos ou até mesmo incorporar características cooperativas em determinados papéis de uma sociedade.

Por fim, o benefício de uma ferramenta gráfica na produção de código é o aumento de produtividade no desenvolvimento de sistemas.

4.2 Limitações

A Adição de uma propriedade simples em um Sistema Multiagente com o uso da ferramenta é conseguida ao apontar para chamadas, dentro do método *protected void Setup()*, do método *addBehaviour* com um parâmetro específico. Porém, caso seja uma decisão de desenvolvimento construir o corpo do agente de outra forma, os ganhos na tentativa de utilização da ferramenta serão poucos ou nenhum.

Como caso típico em que o CM2JADE não tem utilidade é apresentado na figura 4.9(a). Isso ocorre porque o agente, nesse caso, não foi construído com a utilização do método *addBehaviour*. Essa construção é tipicamente utilizada quando se deseja codificar um agente simples, de comportamento previsível e com poucas linhas de código. Para que a ferramenta desempenha uma ajuda maior ao usuário, o agente deveria ter sido estruturado, para realizar o mesmo conjunto de instruções, da forma apresentada na figura 4.9(b).

Torna-se favorável, para a utilização da ferramenta proposta, que haja um esforço de programação para utilizar o método *addBehaviour* implementado na classe *Agent* do *framework JADE*. Esse método pode adicionar uma ampla gama de comportamentos cíclicos, contínuos ou com limite de iterações e realizar o mesmo conjunto de instruções, como mostra a figura 4.9(b).

Quando o código do comportamento que se deseja acrescentar é informado na janela gráfica, os recursos de correção automática de classes não-declaradas no ambiente integrado de programação não são aplicados, como o editor de código-fonte do *Eclipse* possui. Isso exige do usuário uma programação mais atenta e uma possível

```java
package invokatron.teste;

import jade.core.Agent;

public class AgenteO1 extends Agent {

    private static final long serialVersionUID = 1L;

    protected void setup(){

        System.out.println(getAID().getName() + "iniciou");
        // Source code
        doDelete();

    }
}
```

(a) Código em que não é possível acrescentar propriedades com o CM2JADE

```java
package invokatron.teste;

import jade.core.Agent;
import jade.core.behaviours.Behaviour;

public class AgenteO1 extends Agent {

    private static final long serialVersionUID = 1L;

    protected void setup(){
        addBehaviour(UmComportamento());
    }
    private Behaviour UmComportamento() {

        return new Behaviour(){

            public void action() {
                System.out.println(getAID().getName() + "iniciou");
                // Source code
                doDelete();

            }

            public boolean done() {
                return false;
            }
            private static final long serialVersionUID = 1L;
        };
    }
}
```

(b) Código em que é possível acrescentar propriedades com o CM2JADE

Figura 4.9: Comparativo entre classes de agentes segundo a possibilidade de uso do CM2JADE

revisão para que o código esteja sintaticamente correto e faça referência a classes que foram ou que serão implementadas no sistema.

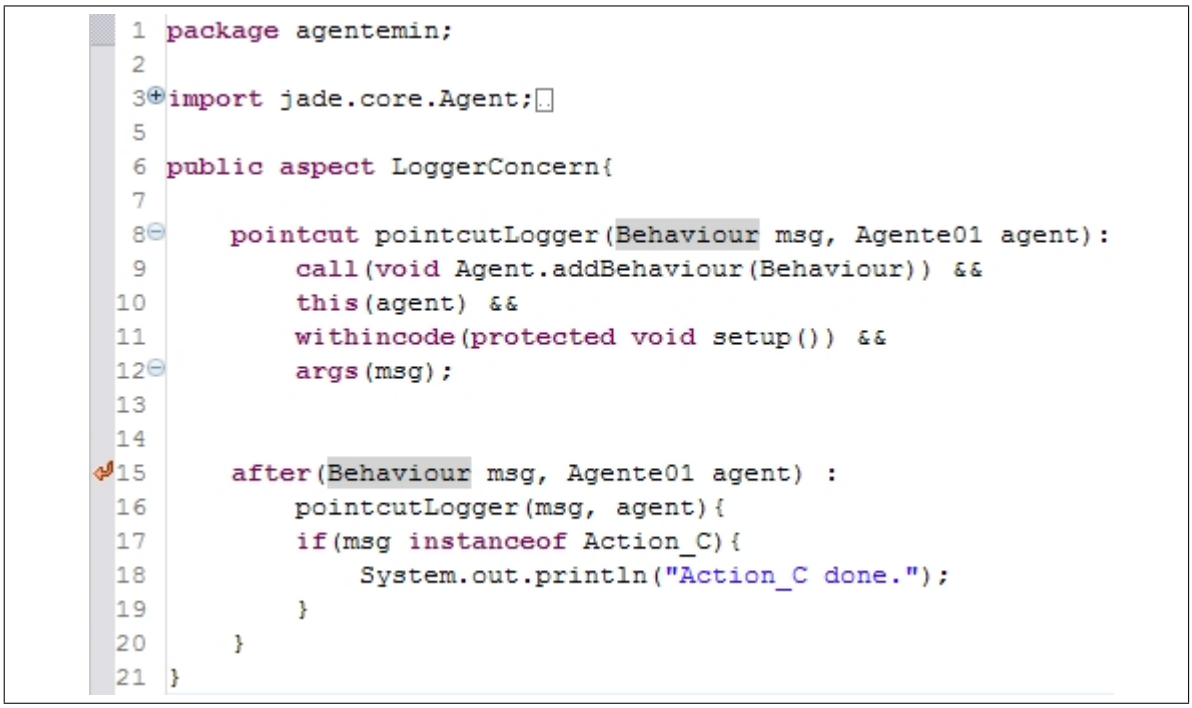
Outro efeito indesejado resultante da metodologia utilizada é cosntatado após a fase de recomposição aspectual (*weaving*). Para discutir esse efeito indesejado a figura 4.10 apresenta o corpo de um agente contruído em *JADE* que faz uso do método de adição de comportamentos *addBehaviour* da classe *Agent*. Esse agente *Agente01* possui três comportamentos *Action_A*, *Action_B* e *Action_C*.

```
1  package agentemin;
2
3  import jade.core.Agent;
4
5  public class Agente01 extends Agent {
6
7      protected void setup(){
8          addBehaviour(new Action_A(this));
9          addBehaviour(new Action_B(this));
10         addBehaviour(new Action_C(this));
11     }
12
13     // ...
14
15
```

Figura 4.10: Exemplo de um classe de agente JADE

Após a utilização da ferramenta proposta para acrescentar um comando de impressão após a execução do comportamento *Action_C*, um aspecto foi gerado como mostra a figura 4.11. Nota-se que, apesar do aspecto mudar o comportamento somente do comportamento para o qual foi proposto, o conjunto de junção aponta para todas as chamadas de *addBehaviour* para quaisquer parâmetros do tipo *Behaviour*. Isso ocorre porque a assinatura do método de adição de comportamento do *framework JADE* solicita um parâmetro do tipo *jade.core.behaviours.Behaviour*. Porém, o comportamento de interesse *Action_C* pode ser isolado pelo teste da linha 17 da figura 4.11.

A ferramenta consegue atingir um comportamento específico de interesse do desenvolvedor, porém, o conjunto de junção também aponta para os demais compor-



```
1  package agentemin;
2
3  import jade.core.Agent;
5
6  public aspect LoggerConcern{
7
8      pointcut pointcutLogger(Behaviour msg, Agente01 agent):
9          call(void Agent.addBehaviour(Behaviour)) &&
10         this(agent) &&
11         withincode(protected void setup()) &&
12         args(msg);
13
14
15     after(Behaviour msg, Agente01 agent) :
16         pointcutLogger(msg, agent){
17         if(msg instanceof Action_C){
18             System.out.println("Action_C done.");
19         }
20     }
21 }
```

Figura 4.11: Aspecto gerado pela CM2JADE para acréscimo de código após o comportamento *Action_ C*

tamentos adicionados no corpo do método *setup()*. O teste da linha 17, apresentado na figura 4.11, é realizado em tempo de execução. Portanto o código gerado pelo aspecto será inserido, neste caso, após todas as chamadas do método *addBehaviour* realizadas dentro do método *setup()*. Esse efeito é indesejado pois o tamanho do arquivo *byte-code* torna-se maior comparado a um outro código-fonte em que o comando de impressão foi inserido diretamente no código sem levar em consideração quaisquer diretivas de modularização. O código *byte-code* resultante da aplicação do aspecto da figura 4.11 na classe de agente *Agente01* após a fase de recomposição aspectual é apresentada na figura 4.12.

As limitações apresentadas nessa seção são contornáveis e podem ser trabalhados posteriormente. Ao avaliar a criticidade dos tópicos discutidos, optou-se por adiar, por momento, as correções necessárias para minimizar ou eliminar as limitações do sistema. Essas e demais modificações serão tratadas na seção 6.

```
1   protected void setup()
2   {
3       this;
4       Action_A action_a = new Action_A(this);
5       action_a;
6       addBehaviour();
7       break MISSING_BLOCK_LABEL_28;
8       Throwable throwable;
9       throwable;
10      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_a, this);
11      throw throwable;
12      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_a, this);
13      this;
14      Action_B action_b = new Action_B(this);
15      action_b;
16      addBehaviour();
17      break MISSING_BLOCK_LABEL_68;
18      Throwable throwable1;
19      throwable1;
20      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_b, this);
21      throw throwable1;
22      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_b, this);
23      this;
24      Action_C action_c = new Action_C(this);
25      action_c;
26      addBehaviour();
27      break MISSING_BLOCK_LABEL_111;
28      Throwable throwable2;
29      throwable2;
30      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_c, this);
31      throw throwable2;
32      LoggerConcern.aspectOf().ajc$after$agentemin_LoggerConcern$1$f1307d73(action_c, this);
33      return;
34  }
35}
```

Figura 4.12: Código *byte-code* resultante da aplicação do aspecto gerado pelo CM2JADE para acréscimo de código após o comportamento $Action\_C$

Capítulo 5

Resultados e discussões

"*Testing can show the presence of errors, but not their absence*"(E. Dijkstra).

Essa seção apresenta uma aplicação do CM2JADE, que é a ferramenta proposta nesse trabalho para adição de uma propriedade transversal em um Sistema Multiagente. Inicialmente é descrito um problema e uma solução que envolve mecanismos estudados na Inteligência Artificial e Processamento Digital de Imagens. É realizada, então, a adição de uma propriedade transversal de agente utilizando o *CM2JADE*. Por fim, uma comparação entre resultados obtidos antes e após a adição desse comportamento é realizada.

5.1 Estudo de caso - Sistema de Reconhecimento de Caracteres

Esse sistema tem como propósito geral identificar caracteres, letras e dígitos, numa imagem que contenha uma placa de automóvel afim de armazená-la para posterior verificação dos dados do veículo e identificar possíveis roubos, situações irregulares, controle de tráfego em estacionamentos ou falsificações. Diversos sistemas computacionais foram criados com o intuito de realizar a segmentação e reconhecimento cor-

reto de placas de automóveis (CARMEN - *Adaptive Recognition Hungary*, NRS - *Computer Recognition Systems*, IMPS(TM) - *Optasia Systems Pte Ltd.*) e diversos estudos acadêmicos têm sido realizados nesse sentido (BRUGGE et al., 1999; SIAH et al., 1996; TSENG et al., 2008; LI; ZENG; ZHOU, 2008; LICENSE..., 1999). Nessa seção serão abordados os requisitos levantados, primeiramente uma solução baseada em agentes e uma posterior alteração dessa solução utilizando a ferramenta apresentada nesse trabalho, bem como algumas considerações sobre o funcionamento desta para o caso abordado. Apresenta-se a solução que diz respeito a coleta da imagem, seu processamento inicial e classificação. A persistência dos dados não será discutida.

Considera-se que as imagens coletadas têm tamanho fixo de 800 *pixels* de largura por 400 *pixels* de altura, resolução de 72 pontos por polegadas (dpi - *Dots per Inch*). Além disso, a placa do veículo pode estar localizada em qualquer região da imagem. O cenário de coleta da imagem pode ser descrito como um local público de passagem de veículos em que a imagem é capturada frente à placa do veículo. A posição da câmera é fixa para todas as amostras coletadas. Um veículo só é liberado para prosseguir quando o sistema armazena a imagem da placa.

A princípio, é coletada uma imagem e enviada aos agentes de pré-processamento. Como exemplo ilustrativo, no decorrer deste capítulo, será utilizada a imagem apresentada na figura 5.1.

A solução proposta é um Sistema Multiagente em que cada grupo de agentes é responsável por diferentes partes do tratamento das imagens coletadas e se comunicam para que a tarefa de tratamento e classificação dos objetos contidos na imagem seja concluída. Cada papel de agente possui autonomia de funcionamento e repete sua função quantas vezes for necessário até que todas as partes da imagem estejam totalmente classificadas em caractere ou não-caractere.

Para desenvolvimento do sistema utilizou-se a linguagem Java, o ambiente integrado de desenvolvimento Eclipse, o *framework* de desenvolvimento de Sistemas Multiagentes *JADE* e o CM2JADE para acréscimo de propriedades transversais em sistemas baseados em agentes. Tipicamente, esse é um Sistema Multiagente devido

Figura 5.1: Imagem de exemplo do estudo de caso - Placa de veículo

ao número de agentes e papéis de agentes envolvidos na resolução do problema. A figura 5.2 apresenta um diagrama de classes do sistema proposto como solução para o problema em questão.

Um agente declarado segundo o *framework JADE* deve extender a classe *jade.core.Agent* e implementar o método *protected void Setup()*. O agente Genesis do pacote *licenseplate.starting* é chamado quando o sistema entra em execução a partir do *framework* pela linha de comando *java jade.Boot -gui genesis:licenseplate.starting.Genesis*. Realizada sua função esse agente deixa de existir no ambiente do sistema. A linha de comando especifica que o ambiente deve inicializar em modo gráfico (opção *-gui*) e que deve lançar, na inicialização, o agente *Genesis*. A única função desse agente é criar os outros agentes do sistema, sejam eles de quaisquer pacotes.

O agente *PersistenceAgent* é responsável pela comunicação com o sistema de banco de dados para armazenamento da imagem coletada e dos dígitos reconhecidos. Os outros agentes do sistema dividem as tarefas de tratamento, pré-processamento e reconhecimento de imagem. Podem variar em número e habitar *containers* de diferentes máquinas nas quais o *framework JADE* estiver em execução.

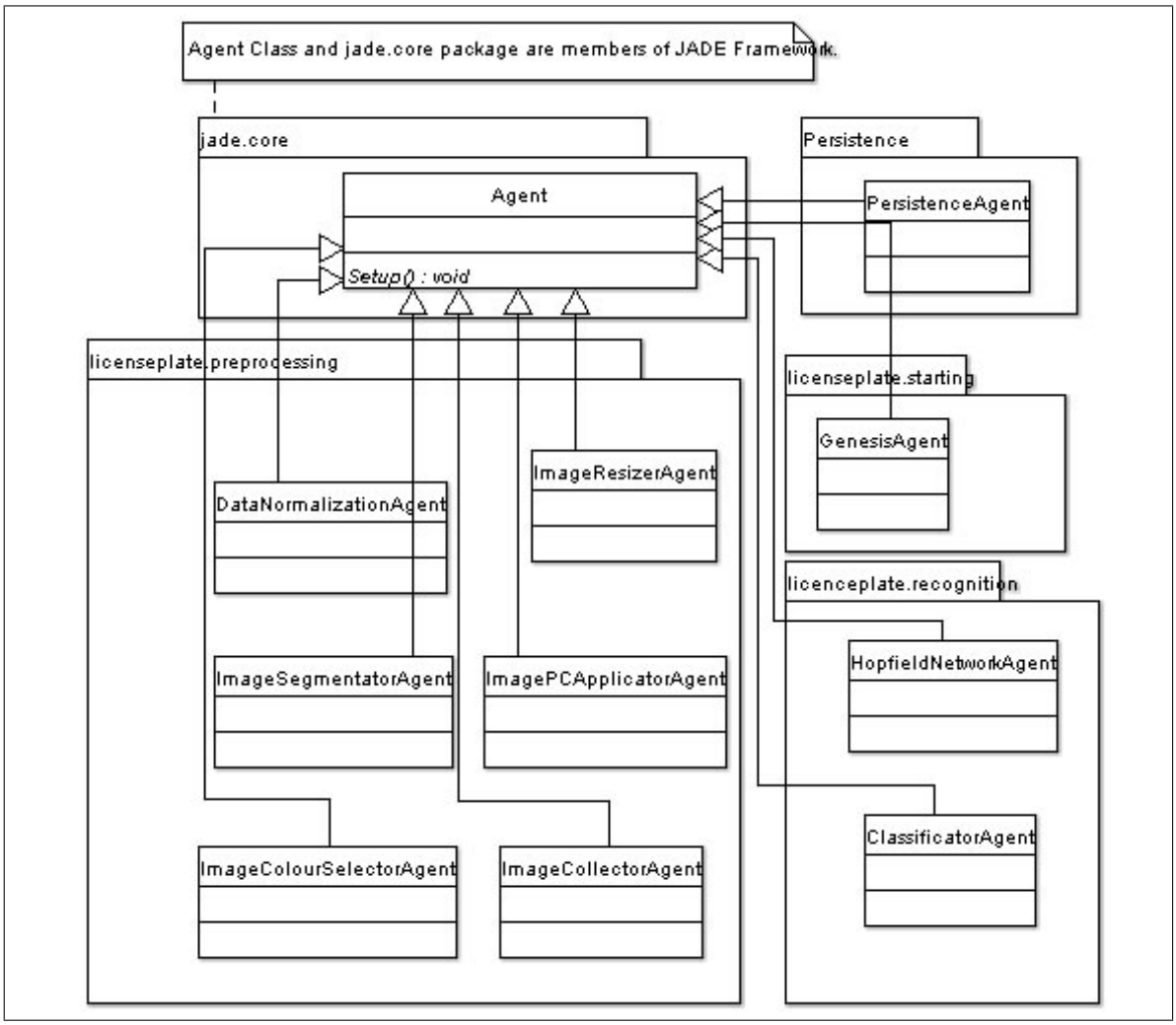


Figura 5.2: Diagrama de classes do Sistema Multiagente de Reconhecimento de placa de veículos

O agente *ImageColourSelectorAgent* é responsável por tratamentos de cor na imagem coletada. Uma de suas principais funções é a binarização do objeto imagem. Ele converte uma imagem em uma imagem binária utilizando um limiar de decisão. Esse agente produz imagens binarizadas a partir de imagens RGB ou tons de cinza. O resultado é uma imagem bidimensional que tem valores 1 (um) (branco) para todos os *pixels* de uma imagem de entrada com luminância maior que um nível e 0 (zero) para todos os outros. Um resultado típico de binarização de imagem realizado por um dos agentes do papel *ImageColourSelectorAgent* é apresentado na figura 5.3.

Após a binarização da imagem completada o agente *ImageColourSelectorAgent*

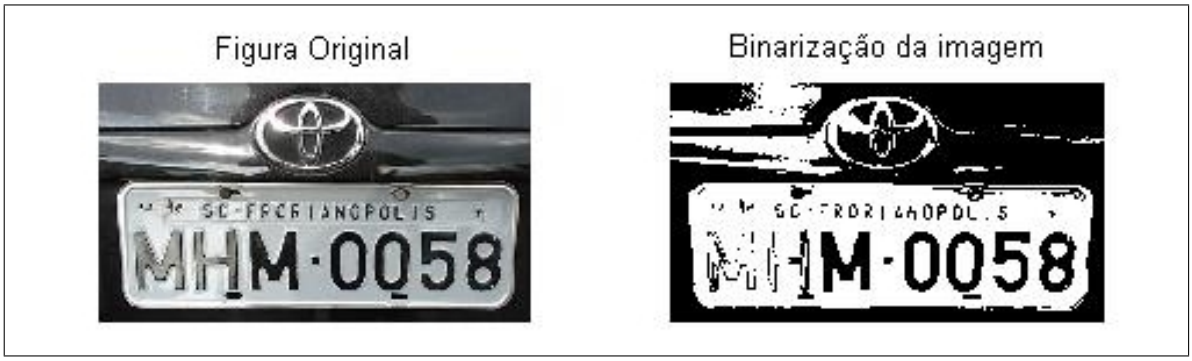


Figura 5.3: Comparativo entre a imagem original e a imagem binarizada por um agente ImageColourSelectorAgent

envia mensagens, definidas segundo o protocolo da classe *ACLMessage* de *JADE*, para os agentes *ImageCollectorAgent* informando que a imagem está pronta para procura de quaisquer objetos. Cada *ImageCollectorAgent a priori* seleciona um pixel aleatório de luminância alta, ou seja, valor 1 (um) e também uma amplitude máxima de busca. A partir desses parâmetros, cada agente percorre as quatro direções a partir do *pixel* inicial tentando recrutar algum objeto contido na imagem. Um agente *ImageCollectorAgent* ao escolher a posição inicial (221,328) e amplitude de busca de 72 pixels inicia a busca em cada direção a partir do ponto apresenta área total de busca como mostra a figura 5.4. A partir de um pixel inicial o agente busca por pixels vizinhos de características próximas, nesse caso, pixels de cor preta. A figura mostra que à medida que as iterações do agente avançam um objeto pode ser formado, caso a região escolhida favoreça a formação de um objeto.

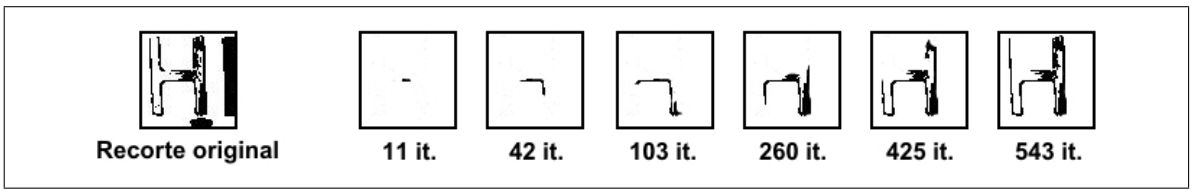


Figura 5.4: Exemplo de busca de objetos por um agente da classe *ImageCollectorAgent*

Após terminada a busca por objetos na parte da imagem em questão, o agente *ImageCollectorAgent* envia mensagem para algum agente responsável por segmentar a imagem. Esse papel de agente recebe o nome de *ImageSegmentationAgent* nesse sis-

tema. A segmentação da imagem selecionada é uma tarefa que consiste na eliminação de colunas e linhas inteiras da matriz-imagem que estejam sem informação relevante, ou seja, nesse caso serão eliminadas as colunas e linhas que estiverem totalmente em branco.

Uma vez que a imagem é submetida a um processo de segmentação, a dimensão de uma imagem em relação a outra pode apresentar diferença. Faz-se necessário, portanto, reajustar o tamanho da imagem antes de aplicar as técnicas de reconhecimento. O redimensionamento da imagem é realizado pelo papel de agente *ImageResizerAgent*. Sua tarefa consiste em retornar uma imagem, a partir da imagem original, com um número específico de linhas e colunas. O método de interpolação utilizado foi o de interpolação bilinear. Essa etapa é necessária porque o mecanismo de reconhecimento utilizado necessita de um número determinado de *pixels*. Um exemplo de três imagens que contêm diferentes objetos de diferentes tamanhos são apresentados antes e após o redimensionamento realizado por agentes do tipo *ImageResizerAgent*.

A imagem coletada passa por uma série de transformação, na fase de pré-processamento, para facilitar o reconhecimento dos objetos contidos na imagem. A imagem da placa do automóvel foi binarizada, segmentada e redimensionada. Porém, possivelmente, um grande número de *pixels* ainda compõe a imagem. Quanto maior a quantidade de dados de entrada submetidos ao mecanismo de reconhecimento, mais tempo esse mecanismo se dedica a classificação do objeto. Para diminuir o número de pontos submetidos ao mecanismo de reconhecimento e classificação, optou-se pela aplicação da técnica de análise de componentes principais (PCA - *Principal Components Analysis*)(MOORE, 1981; LOS, 1989).

Muitos dos algoritmos clássicos de processamento de sinal recorrem, de uma ou de outra forma, a métodos de descorrelação de dados. As vantagens de tais técnicas baseiam-se num princípio de que ao descorrelacionar os dados, parte da informação redundante em cada dimensão é eliminada.

O objetivo da análise em componentes principais é encontrar uma transformação mais representativa e geralmente mais compacta das observações. O método

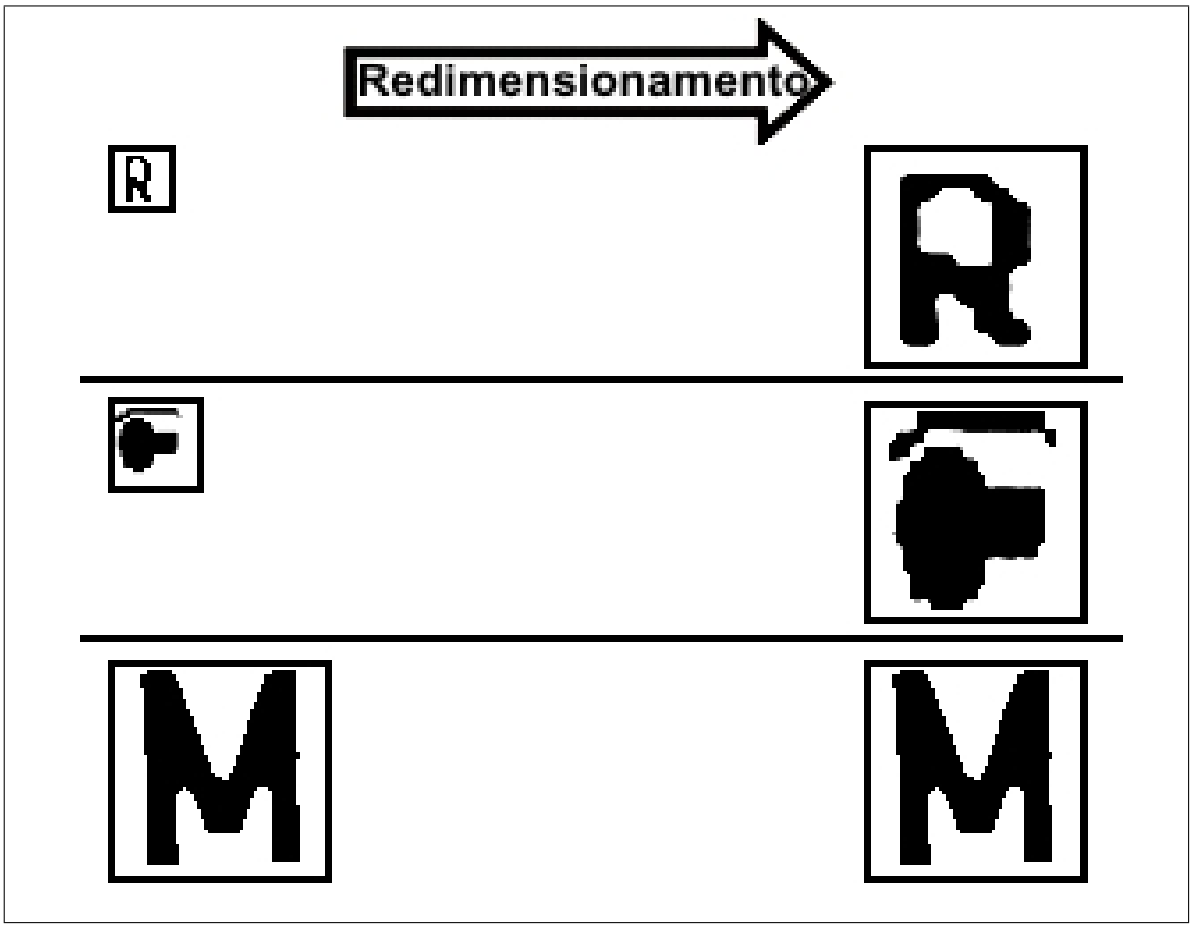


Figura 5.5: Exemplo de redimensionamento de objetos por um agente da classe *ImageResizerAgent*

de ACP transforma um vetor aleatório $\mathbf{x} \in R^m$ noutro vetor $\mathbf{y} \in R^n$ (para $n \leq m$) projetando $\mathbf{x}$ nas n direções ortogonais de maior variância - as componentes principais. Essas componentes são individualmente responsáveis pela variância das observações, e, neste sentido, representam-nas mais claramente. No sistema computacional em questão a aplicação da técnica de análise de componentes principais é função do agente *PCAApplicatorAgent*. Optou-se, através do método de tentativa e erro, pela utilização de vinte e sete componentes principais. Isso significa que ao receber uma imagem do agente *ImageResizerAgent*, o agente *PCAApplicatorAgent* realiza a análise dos componentes e apresenta os vinte e sete componentes para o agente responsável pelo reconhecimento e classificação.

O agente *ClassifierAgent*, ao receber uma mensagem segundo o protocolo *ACLMes-*

sage, obtém os pontos resultantes da análise de componentes principais e os submete para reconhecimento. Optou-se como mecanismo de classificação a utilização de uma rede neural artificial multicamadas (*Multilayer Perceptron*)(RUCK et al., 1990; ATTALI; PAGÈS, 1997) e função de ativação tangente sigmoidal nas camadas intermediária e de saída e de arquitetura $27 \times 9 \times 6$. Isso significa que a rede neural é composta de vinte e sete entradas, nove neurônios na camada intermediária e seis neurônios na camada de saída. A tabela 5.1 apresenta as saídas desejadas para cada caractere reconhecido.

Para realização da fase de treinamento, utilizou-se um conjunto de duas mil amostras uniformemente distribuídos entre os padrões de entrada. Para validação e para teste final, utilizou-se um total de setessentas amostras em cada. Essa divisão do conjunto de amostras em três partes é necessária devido às características do algoritmo de treinamento da rede neural. O algoritmo de treinamento utilizado foi o algoritmo multiobjetivo (TEIXEIRA et al., 2000; TEIXEIRA, 2001). Como resultado, obteve-se uma rede neural com 97.2% de acerto para dados de teste final.

O agente *ClassifierAgent* submete os dados recebidos ao mecanismo de classificação, que neste caso é a rede neural, e obtém uma resposta entre as marcadas na tabela 5.1. Essa resposta é enviada para o agente responsável pela persistência dos dados e para o agente *ImageCollectorAgent*.

A persistência é necessária uma vez que um caractere é encontrado na placa, outros que estejam posicionados antes ou após sejam também encontrados e uma sequência correta de dígitos seja formada. Se não houver a persistência correta dos dados, uma placa de veículo, por exemplo, com sequência MHM-0058 pode ser reconhecida como HMM-0580.

O agente *ImageCollectorAgent* recebe a resposta se a imagem que ele enviou para os outros agentes contém um caractere válido para que seja avaliada a posição de varredura de *pixels* na imagem e a amplitude de busca. Uma vez que a resposta do agente classificador seja que a imagem submetida contém um caractere válido, o agente coletor reinicia a busca numa região próxima e com amplitude semelhante.

	Neurônios de Saída							Neurônios de Saída					
Caractere	**01**	**02**	**03**	**04**	**05**	**06**	**Caractere**	**01**	**02**	**03**	**04**	**05**	**06**
0	-1	-1	-1	-1	-1	-1	**J**	-1	1	-1	-1	1	1
1	-1	-1	-1	-1	-1	1	**K**	-1	1	-1	1	-1	-1
2	-1	-1	-1	-1	1	-1	**L**	-1	1	-1	1	-1	1
3	-1	-1	-1	-1	1	1	**M**	-1	1	-1	1	1	-1
4	-1	-1	-1	1	-1	-1	**N**	-1	1	-1	1	1	1
5	-1	-1	-1	1	-1	1	**O**	-1	1	1	-1	-1	-1
6	-1	-1	-1	1	1	-1	**P**	-1	1	1	-1	-1	1
7	-1	-1	-1	1	1	1	**Q**	-1	1	1	-1	1	-1
8	-1	-1	1	-1	-1	-1	**R**	-1	1	1	-1	1	1
9	-1	-1	1	-1	-1	1	**S**	-1	1	1	1	-1	-1
A	-1	-1	1	-1	1	-1	**T**	-1	1	1	1	-1	1
B	-1	-1	1	-1	1	1	**U**	-1	1	1	1	1	-1
C	-1	-1	1	1	-1	-1	**V**	-1	1	1	1	1	1
D	-1	-1	1	1	-1	1	**W**	1	-1	-1	-1	-1	-1
E	-1	-1	1	1	1	-1	**X**	1	-1	-1	-1	-1	1
F	-1	-1	1	1	1	1	**Y**	1	-1	-1	-1	1	-1
G	-1	1	-1	-1	-1	-1	**Z**	1	-1	-1	-1	1	1
H	-1	1	-1	-1	-1	1	**Não-dig**	1	-1	-1	1	-1	-1
I	-1	1	-1	-1	1	-1	**Escuro**	1	-1	-1	-1	-1	1

Tabela 5.1: Parâmetros do mecanismo de reconhecimento

É importante notar que nesse cenário, cada agente de mesmo papel trabalha isoladamente, e que as mensagens trocadas entre eles servem apenas para garantir que dois ou mais agentes de mesmo papel não investiguem o mesmo conjunto de *pixels*. Uma situação típica com o funcionamento do sistema concebido dessa forma é apresentada para posterior comparação realizada com o uso da ferramenta construída nesse trabalho.

Para fins práticos denomina-se o funcionamento do sistema com as características apresentadas acima de cenário individual. Cada agente coletor comunica com os demais agentes coletores informando a sua posição na imagem para que não exista retrabalho de agentes numa mesma região. E cada agente coletor solicita serviços para agentes de outros papéis.

Após ser coletada a imagem é binarizada, como mostra a figura 5.3 e apresentada aos agentes coletores. Cada agente coletor seleciona uma área de busca e encaminha para outros agentes e espera a resposta, para decidir se a próxima área de busca será ou não próxima da atual.

Para efeito comparativo cada iteração será contada desde a escolha de cada agente coletor por uma região até o término da classificação de todas as imagens submetidas dessas mesmas regiões. Só então acontece uma nova busca por objetos na imagem. A figura 5.6 mostra a evolução da busca dos agentes em seis pontos de verificação. Cada ponto de verificação representa um conjunto de iterações realizadas e a área explorada pelos agentes coletores.

Na figura 5.6, os pontos de verificação 01, 02, 03, 04, 05 e 06, as iterações de número 10, 20, 30, 40, 50 e 60, respectivamente. Foram destinados cinco agentes coletores para esse trabalho, um agente de tratamento de cor e o mesmo número de cinco agentes foi mantido para segmentadores, redimensionadores, analisadores de componentes principais e agentes classificadores.

É provavel que o sistema execute corretamente sua função de classificação de placas de veículos e percorra toda a imagem em busca de novos caracteres. Porém,

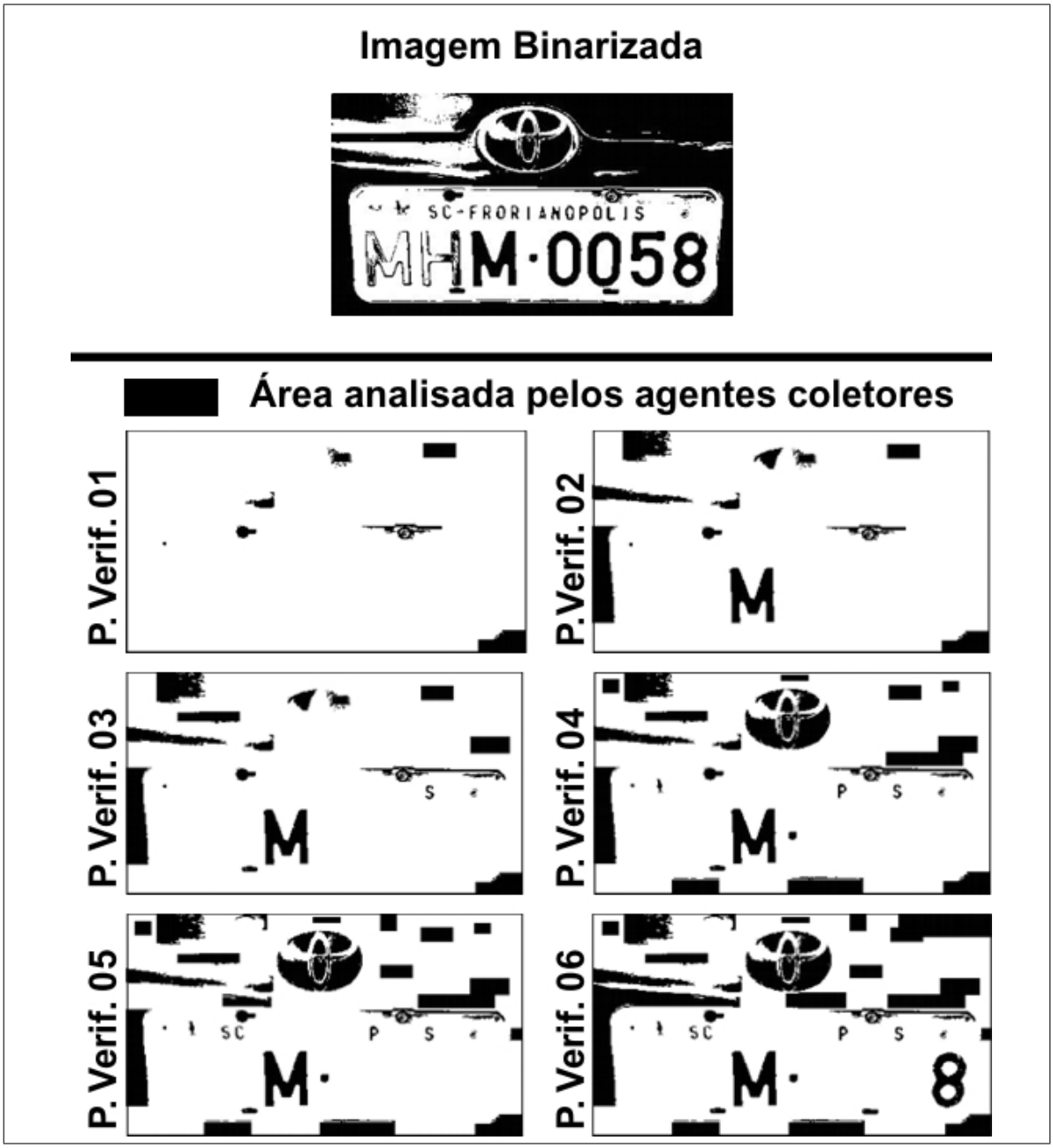


Figura 5.6: Exemplo de um caso típico de varredura dos agentes coletores no cenário individual

é viável que o sistema encontre os caracteres o quanto antes possível e que o sistema incorpore características sociais como cooperação, ajuda mútua e decisões coletivas. Uma proposta de adicionar o requisito de colaboração nesse Sistema Multiagente é apresentado na seção 5.1.1. O uso do CM2JADE é apresentado, bem como os resultados dessa para o sistema computacional baseado em agentes em questão.

5.1.1 Adição de propriedade transversal de Colaboração utilizando o CM2JADE

A seção anterior mostra um Sistema Multiagente que *a priori* desempenha o papel para o qual foi construído sem nenhuma propriedade transversal declarada através do uso da ferramenta desenvolvida nesse trabalho ou qualquer código escrito na linguagem *AspectJ*. É caracterizado um Sistema Multiagente porque dois ou mais agentes desempenham papéis diferentes e trocam mensagens entre si para um propósito geral.

Os agentes coletores apresentam comportamentos individuais no sistema de reconhecimento de dígitos em placas de automóveis, como abordado anteriormente. Essa categoria de agentes recebe mensagens para realizar sua tarefa e envia mensagens solicitando serviços de outros agentes para que a tarefa seja completada.

A figura 5.7 parte do código dos agentes coletores. O método de interesse, neste caso é o *protected void Setup()* que deve ser implementado por quaisquer agentes *JADE*, ou seja, quaisquer classes que herdem a classe *jade.core.Agent*.

Basicamente, o agente aguarda que a lista de agentes prevista para o correto funcionamento do sistema esteja completa e começa seu ciclo de atividades. Cada atividade é modelada como um comportamento da classe *jade.core.behaviours. Behaviour* do *framework JADE*. Inicialmente ele verifica se o agente responsável pela binarização da imagem disponibilizou a imagem para que a coleta tenha início. Se a binarização foi realizada, um objeto imagem é disponibilizado e então o fluxo de atividades continua. Os passos a seguir são:

```
1  package licenseplate.preprocessing;
2
3  import jade.core.AID;
10
11 public class ImageCollectorAgent
12 extends Agent{
13
14     private int type = AIDItem.COLLECTOR;
15     public ImagePart image;
16     public ImagePart[] histImagePart;
17     public boolean[] ImagePartClassification;
18
19
20     protected void setup(){
21         System.out.println("ImageCollectorAgent");
22         AIDList.addAIDItem(getAID(),"ImageCollectorAgent", type);
23
24         while(!AIDList.isComplete()){
25             doWait(100);
26         }
27         addBehaviour(new CheckingMessageFromColourAgent(this));
28         if(image != null){
29             addBehaviour(new CheckingMessageFromClassificator(this));
30             addBehaviour(new ImagePixelChoice(this));
31             addBehaviour(SelectingArea());
32             addBehaviour(SendingAreaToSegmentation());
33             addBehaviour(new CheckingMessageFromClassificator(this));
34         }
35     }
36     //...
```

Figura 5.7: Parte do código-fonte da classe *ImageCollectorAgent*

- Verificar a existência de mensagens do agente classificador. Na primeira coleta, não existem mensagens do agente classificador pois nenhuma imagem foi submetida ainda para classificação. Porém, esse comportamento é importante da segunda vez em diante para que o agente coletor reajuste, se necessário, os parâmetros de posicionamento e de amplitude de busca. A adição desse comportamento é realizada na linha 29 do código apresentado na figura 5.7.

- Escolher uma posição inicial na imagem para busca. A princípio essa escolha é feita aleatoriamente. Após a primeira iteração do sistema, a escolha é feita por critérios de decisão que envolvem a resposta do agente classificador. Se a região anterior foi favorável a localização de dígitos válidos, uma área próxima é escolhida para nova busca. Caso contrário, o agente opta por uma área aleatória da imagem. O método *addBehaviour(new ImagePixelChoice())* é responsável

pelas regras de decisão de localização e amplitude de busca de caracteres.

- Realizar varredura bidirecional para formar algum objeto possível. Um exemplo da execução do método *addBehaviour(SelectionArea())* é apresentado na figura 5.8.

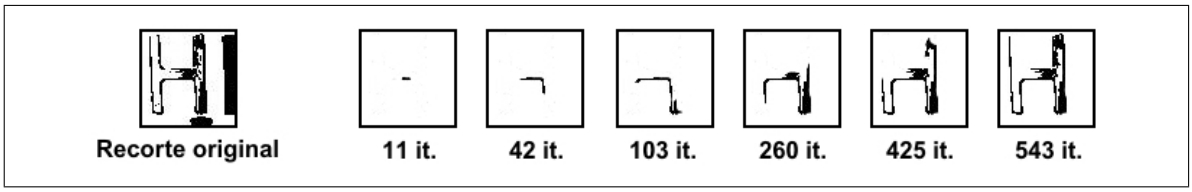


Figura 5.8: Exemplo de busca de objetos por um agente da classe *ImageCollectorAgent*

- Enviar a figura formada para segmentação. O método *action()* do comportamento *SendingAreaToSegmetation()* faz o envio, utilizando protocolo de mensagem ACL, para algum agente segmentador continuar a tarefa.

- Aguardar a mensagem do classificador para retomar o fluxo de atividades. O agente espera até que a imagem que foi enviada para os outros agentes seja classificada como um dos dígitos válidos ou não-dígito para então continuar a procurar. Isso é possivel com a chamada do método *doWait()*, herdado da classe *Agent*. Após o processo de reconhecimento, o agente classificador envia a mensagem-resposta e chama o método *doNotify()* para o agente coletor retornar a buscar objetos em novas regiões.

A nova propriedade transversal irá afetar o corpo do método *protected void Setup()*. A proposta é que cada agente Coletor, ao receber uma mensagem positiva do agente Classificador sobre uma determinada região, envie mensagens para os outros agentes coletores posicionarem suas buscas próximas da área desse agente, e com amplitudes semelhantes. Esta estratégia de localização de caracteres baseia-se na idéia de que, em uma dada placa, os dígitos são igualmente espaçados e estão próximos uns dos outros, e dificilmente estão localizados por toda a foto, dependendo da forma de coleta. Uma vez que um dígito é encontrado, existe uma chance de que os outros dígitos estejam próximos do primeiro, ou pela esquerda ou pela direita.

A propriedade de Colaboração é adicionada utilizando recursos gráficos presentes no CM2JADE que é objeto de estudo desse trabalho. Informando alguns parâmetros de entrada para geração do código da propriedade, um arquivo XML é gerado contendo os dados da propriedade e *a posteriori* são convertidos em um código *AspectJ* que representa a propriedade inserida como um Aspecto no sistema computacional.

O uso da ferramenta inicia-se quando o usuário usa um dos três recursos gráficos inseridos na barra de ferramentas do *Eclipse* com um dos propósitos abaixo:

- Adição de propriedade transversal em um Sistema Multiagente;
- Gerenciamento de propriedades tranversais;
- Adicionar outros comportamentos em classes de agentes do sistema de propriedades transversais que já foram declaradas.

A figura 5.9 apresenta a tela de adição de propriedades transversais com os dados da propriedade Colaboração que afeta a chamada do comportamento *addBehaviour(new CheckingMessageFromClassificator(this))* do método *protected void Setup()* da classe *ImageCollectorAgent*.

Os parâmetros fornecidos na interface indicam que é desejada a inserção de uma propriedade transversal de nome *Collaboration* para agentes da classe *ImageCollectorAgent*, situado no *container LicensePlate*. O novo comportamento será executado após a chamada do método que adiciona comportamentos da classe *CheckingMessageFromClassificator*. As ações para a propriedade tranversal de Cooperação fornecidas na interface caminham no sentido de incorporar ao agente em questão a capacidade de enviar mensagens para os outros agentes coletores a posição de busca do agente, a amplitude de busca e a resposta do agente classificador para a área da imagem. A propriedade transversal aponta para a linha 29 do código apresentado na figura 5.7.

Quando a operação é confirmada, um arquivo escrito em linguagem XML representa os dados fornecidos nessa interface gráfica. A conversão dos dados armazenados no XML em um aspecto escrito na linguagem *AspectJ* é realizada por um *template*

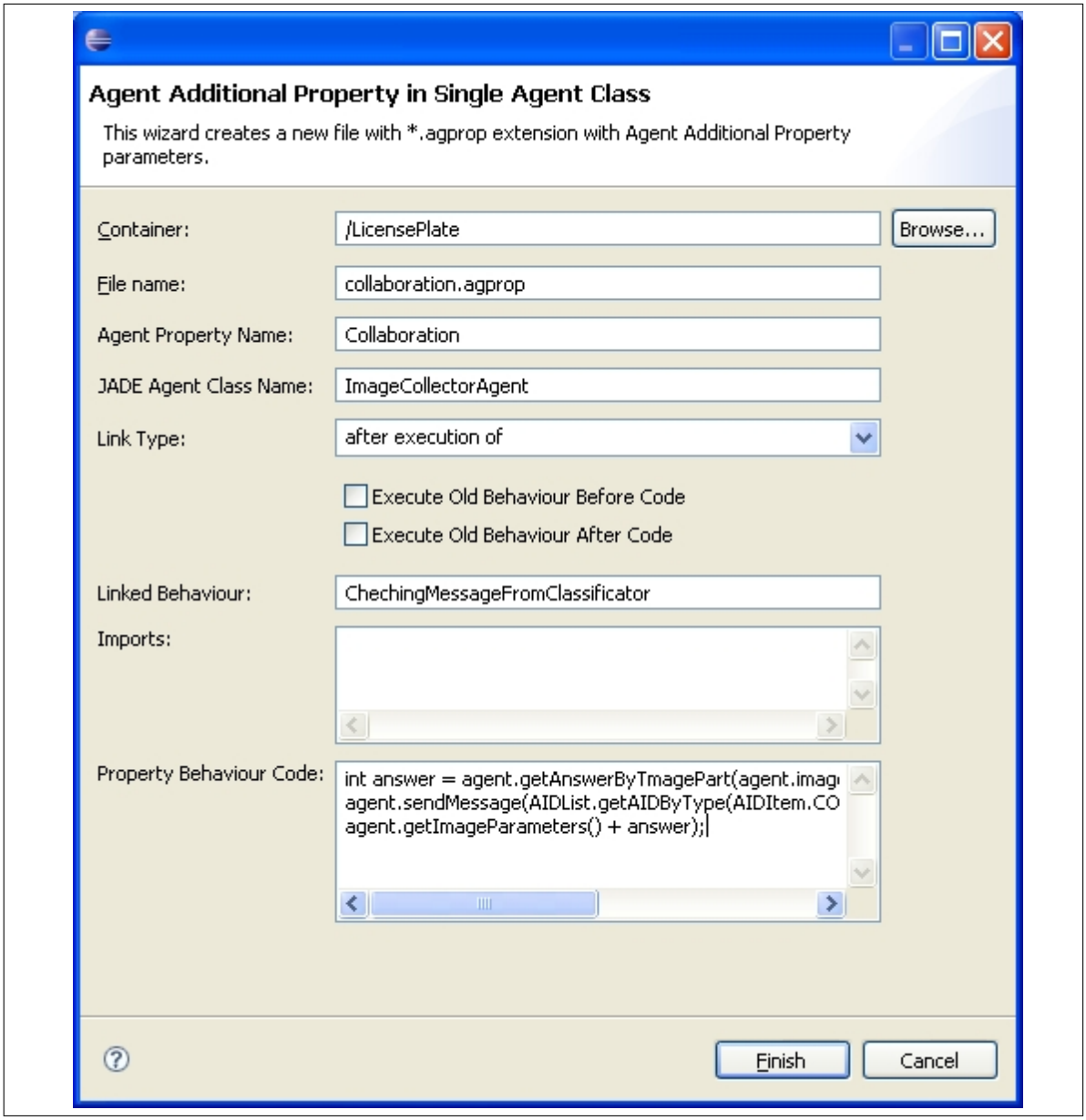


Figura 5.9: Interface de adição de propriedades tranversais em uma Classe de Agente - propriedade Colaboração

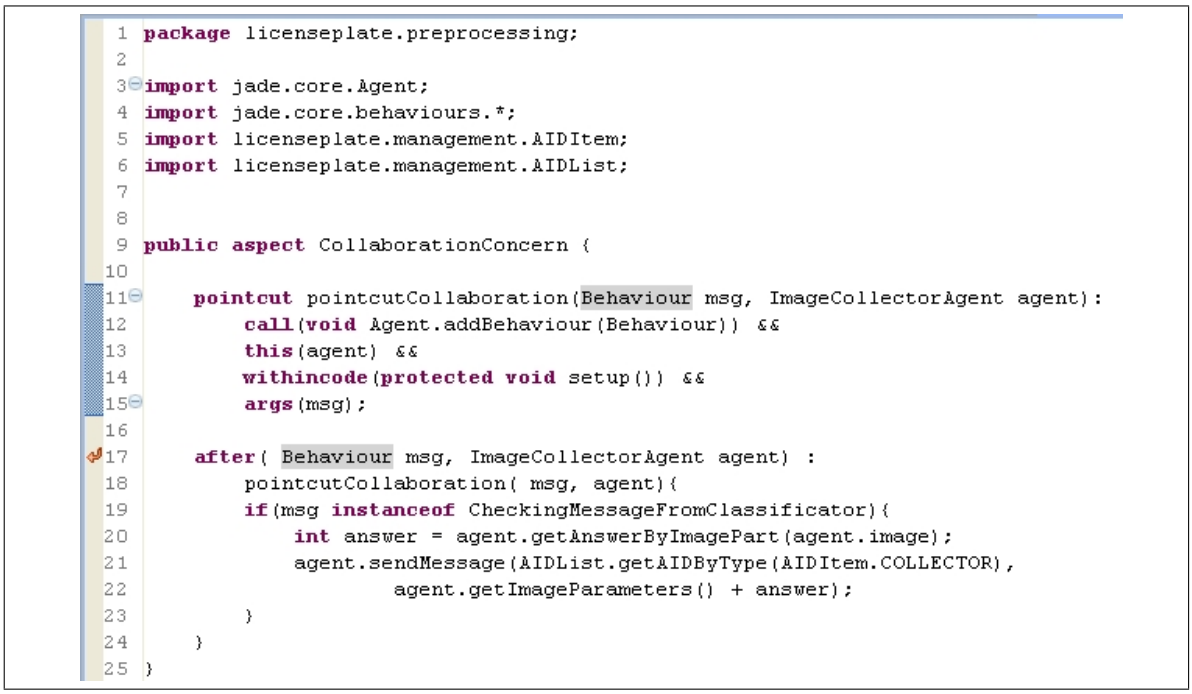
XSL, como abordado na seção 3. O arquivo XML gerado a partir dos dados fornecidos na interface da figura 5.9 é apresentado na figura 5.10.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet href="LO2AgentClassSchema.xsl" type="text/xsl"?>
3  <AspectConcern name="CollaborationConcern">
4      <Package>licenseplate.preprocessing</Package>
5      <Imports>
6          <Import>jade.core.Agent</Import>
7          <Import>jade.core.behaviours.*</Import>
8      </Imports>
9      <Classname>ImageCollectorAgent</Classname>
10     <Advice>after</Advice>
11     <Advicereturn></Advicereturn>
12     <AdviceproceedBefore></AdviceproceedBefore>
13     <AdviceproceedAfter></AdviceproceedAfter>
14     <BehaviourLink>CheckingMessageFromClassificator</BehaviourLink>
15
16     <Source>
17         int answer = agent.getAnswerByImagePart(agent.image);
18         agent.sendMessage(AIDList.getAIDByType(AIDItem.COLLECTOR),
19         agent.getImageParameters() + answer);
20     </Source>
21 </AspectConcern>
```

Figura 5.10: Arquivo XML gerado a partir dos parâmetros da interface de adição de propriedades tranversais em SMAs - propriedade Colaboração

Por fim, um arquivo de código na linguagem *AspectJ* é gerado e representa, efetivamente, a adição da propriedade transversal no Sistema Multiagente no momento da execução do programa computacional. O aspecto aponta, inicialmente para todas as adições de comportamento do método *protected void Setup()* do Agente em questão. Porque a assinatura do método *addBehaviour* da classe *Agent* é *addBehaviour(Behaviour)*. Porém, na codificação do aspecto *ColaborationConcern* é realizado um teste de tipo de comportamento. Portanto consegue-se selecionar os comportamentos do tipo *CheckingMessageFromClassificator*. A figura 5.11 apresenta o código *AspectJ* que representa o aspecto que adiciona a propriedade transversal apresentada na interface 5.9.

O ponto de junção apresentado na linha 11 da figura 5.11 representa todas as chamadas do método *addBehaviour* herdados da classe *jade.core.Agent* que tenham como parâmetro um objeto do tipo Behaviour, que sejam chamados dentro do corpo do



```
1  package licenseplate.preprocessing;
2
3  import jade.core.Agent;
4  import jade.core.behaviours.*;
5  import licenseplate.management.AIDItem;
6  import licenseplate.management.AIDList;
7
8
9  public aspect CollaborationConcern {
10
11     pointcut pointcutCollaboration(Behaviour msg, ImageCollectorAgent agent):
12         call(void Agent.addBehaviour(Behaviour)) &&
13         this(agent) &&
14         withincode(protected void setup()) &&
15         args(msg);
16
17     after( Behaviour msg, ImageCollectorAgent agent) :
18         pointcutCollaboration( msg, agent){
19         if(msg instanceof CheckingMessageFromClassificator){
20             int answer = agent.getAnswerByImagePart(agent.image);
21             agent.sendMessage(AIDList.getAIDByType(AIDItem.COLLECTOR),
22                     agent.getImageParameters() + answer);
23         }
24     }
25 }
```

Figura 5.11: Código *AspectJ* gerado a partir dos parâmetros da interface de adição de propriedades tranversais em SMAs - propriedade Colaboração

método *protected void setup()*. Um teste posterior realizado no adendo *after* verifica se o comportamento passado como parâmetro na chamada *addBehaviour(param)* é do tipo *CheckingMessageFromClassificator*.

A vantagem do uso do CM2JADE e, conseqüentemente, da programação orientada a aspectos nesse sistema é que o código da classe *ImageCollectorAgent* permance inalterado. O comportamento dos agentes pode ser estudado em cenários com alternância de comportamentos tranversais. Nesse caso, é possível medir o benefício da adição da propriedade transversal de colaboração simplesmente ativando ou desativando a propriedade transversal por meio de recursos gráficos.

Para que os agentes coletores recebam as mensagens de um agente coletor e a partir dos parâmetros transmitidos na mensagem pudessem ajustar suas posições e amplitudes de busca, outra parte dessa propriedade foi inserida de forma semelhante à primeira, utilizando recursos gráficos e gerando o aspecto automaticamente. A figura 5.12 apresenta a tela de adição de propriedades transversais com os dados

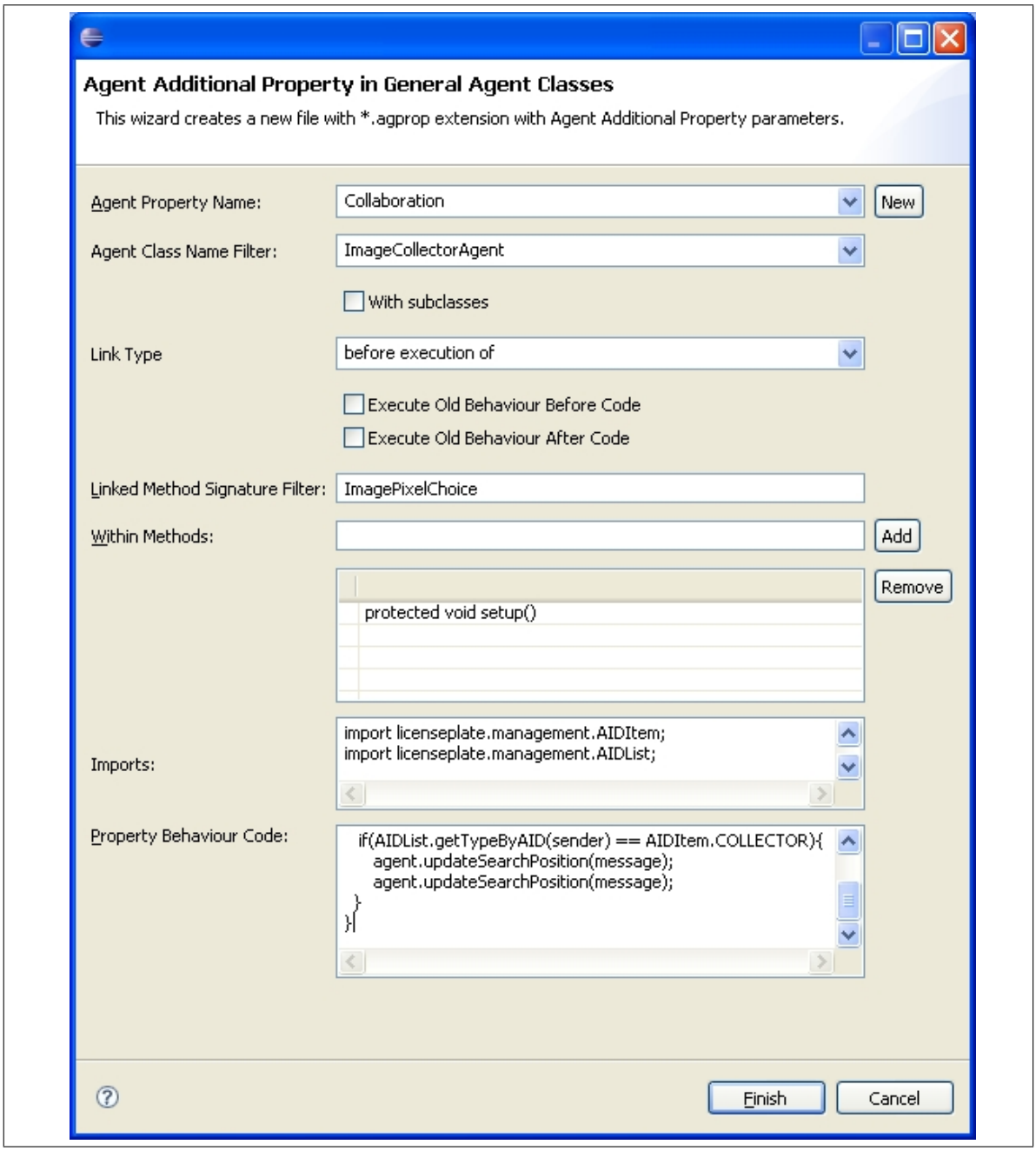


Figura 5.12: Interface de adição de propriedades transversais em uma Família de Classes de Agentes - propriedade Colaboração

da propriedade Colaboração. O aspecto gerado para que os agentes fossem capazes de atualizar suas posições e amplitudes de busca em função da resposta dos outros agentes coletores é apresentado na figura 5.13.

```
1  package licenseplate.preprocessing;
2
3  import jade.core.AID;
4  import jade.core.Agent;
5  import jade.core.behaviours.*;
6  import jade.lang.acl.ACLMessage;
7  import licenseplate.management.AIDItem;
8  import licenseplate.management.AIDList;
9
10
11 public aspect CollaborationConcernFragment {
12
13     pointcut pointcutCollaboration(Behaviour msg, ImageCollectorAgent agent):
14         call(void Agent.addBehaviour(Behaviour)) &&
15         this(agent) &&
16         withincode(protected void setup()) &&
17         args(msg);
18
19     before(Behaviour msg, ImageCollectorAgent agent) :
20         pointcutCollaboration( msg, agent){
21         if(msg instanceof ImagePixelChoice){
22             ACLMessage message = agent.receive();
23             if(message != null){
24                AID sender = message.getSender();
25                if(AIDList.getTypeByAID(sender) == AIDItem.COLLECTOR){
26                    agent.updateSearchPosition(message);
27                    agent.updateSearchPosition(message);
28                }
29
30             }
31         }
32     }
33 }
```

Figura 5.13: Código *AspectJ* complementar gerado a partir dos parâmetros da interface de adição de propriedades tranversais em SMAs - propriedade Colaboração

Com a incorporação dos aspectos apresentados nas figuras 5.11 e 5.13 o sistema pode ser executado novamente, com os mesmos parâmetros do cenário individual, para efeito comparativo. O número de agentes de cada categoria foi mantido. Chamar-se-á a execução do sistema com esses parâmetros de cenário cooperativo. A figura 5.14 mostra a evolução da busca dos agentes em seis pontos de verificação. Cada ponto de verificação representa um conjunto de iterações realizadas e a área explorada pelos agentes coletores. Os pontos de verificação representam o mesmo número de iterações do cenário individual.

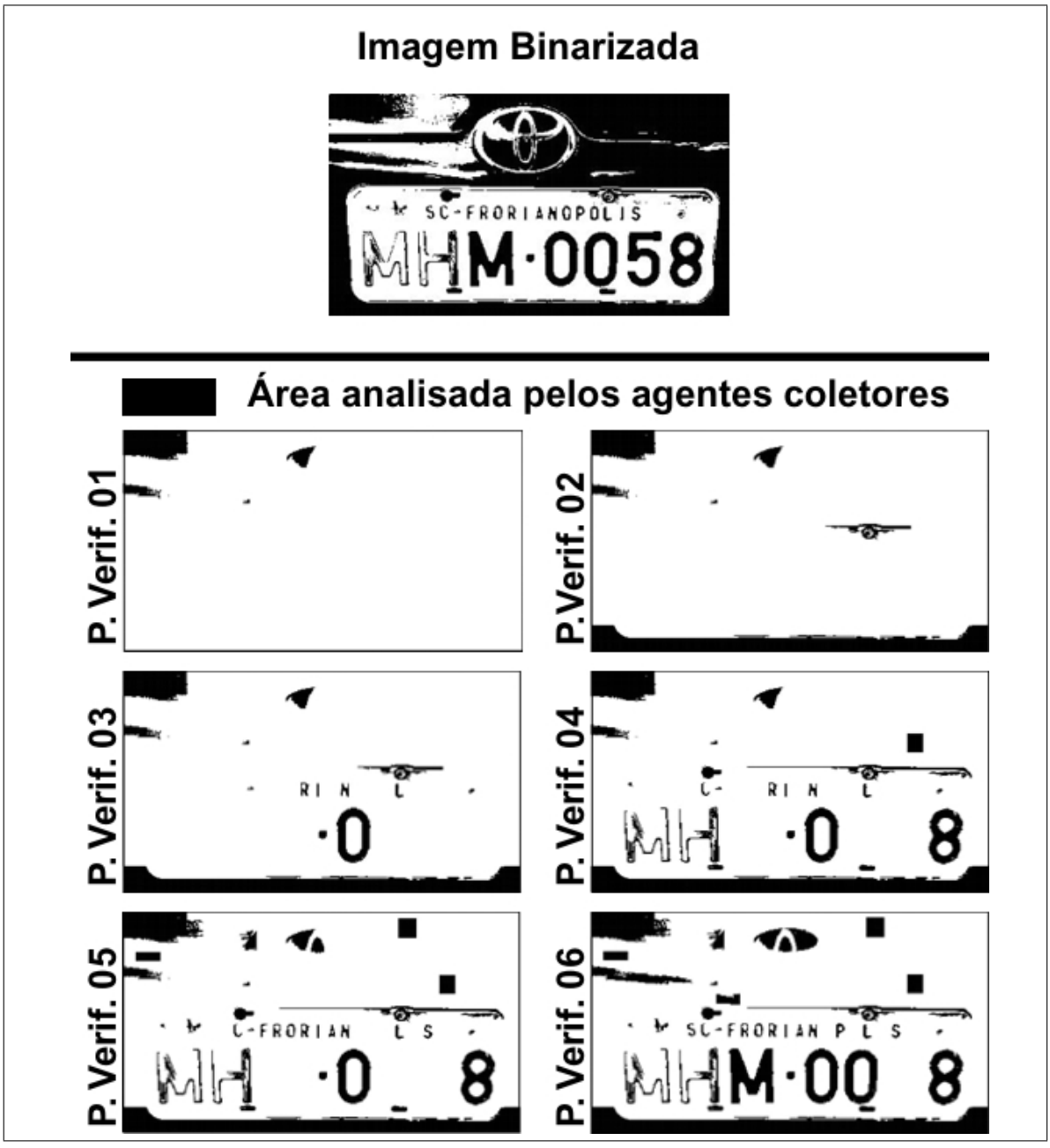


Figura 5.14: Exemplo de um caso típico de varredura dos agentes coletores no cenário cooperativo

Nota-se, na figura 5.14, que a medida que as iterações vão sendo realizadas e que um ou mais agentes encontram regiões com dígitos válidos os outros agentes automaticamente se ajustam para posições próximas dos agentes que realizaram a tarefa com sucesso afim de procurar por dígitos próximos. Isso pode ser observado no ponto de verificação 03, em que um agente acha o dígito 0 e algumas letras são encontrados. A partir desse ponto, os outros agentes coletores ajustaram seus parâmetros para obter os caracteres próximos aos identificados.

A quantidade de dígitos encontrados no ponto de verificação 06 varia para os dois cenários. A contribuição da propriedade Cooperação fez com que os agentes coletores encontrassem a maior parte dos dígitos com menor quantidade de iterações. Maiores detalhes são apresentados na figura 5.15.

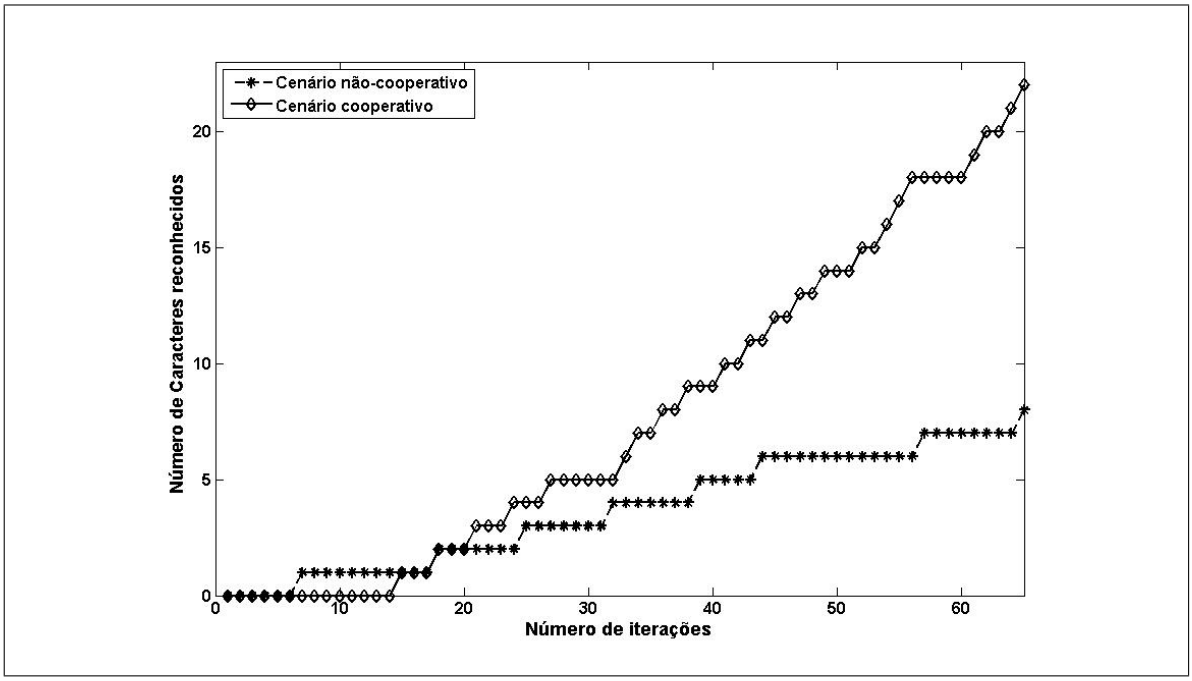


Figura 5.15: Comparação entre cenários baseada no uso de comportamento cooperativo em um Sistema Multiagente

O número máximo de caracteres a ser reconhecido na placa do automóvel é 22. No cenário individual a quantidade de iterações até que todos os caracteres fossem reconhecidos foi de 113 iterações. No cenário cooperativo o sistema completou a tarefa de reconhecimento com 65 iterações. O primeiro caractere válido foi reconhecido na

sétima iteração quando os agentes não cooperam entre si. E no cenário em que existe cooperação foram necessárias 15 iterações até que o primeiro caractere fosse reconhecido. Isso acontece porque nos dois cenários, a escolha da posição de busca na imagem é feita de forma aleatória até que o primeiro objeto válido seja encontrado. Porém, quando os agentes coletores cooperam entre si, a partir do primeiro dígito encontrado, os outros reajustam seus parâmetros para realizar uma busca em regiões mais próximas do dígito reconhecido. Isso resulta, como mostrado no gráfico da figura 5.15, em um reconhecimento dos 22 caracteres com menor quantidade de iterações no cenário cooperativo.

Para estudar o comportamento do sistema com ou sem cooperação, basta mudar o status da propriedade transversal na janela gráfica da ferramenta que exibe os dados da propriedade transversal e os arquivos envolvidos na geração do arquivo que contém o aspecto escrito em linguagem *AspectJ* responsável pela propriedade transversal. A janela gráfica é apresentada contendo a propriedade Cooperação e suas características na figura 5.16.

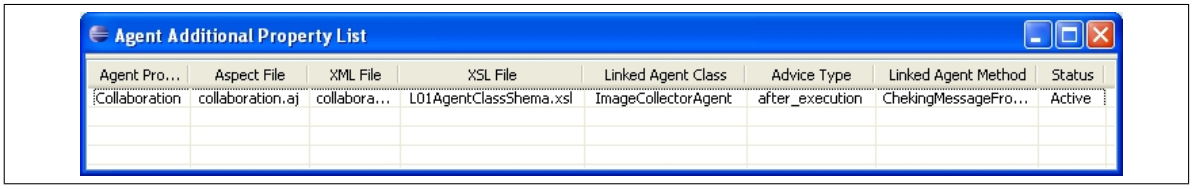


Figura 5.16: Interface de gerenciamento de propriedades tranversais em Sistemas Multiagentes - propriedade Colaboração

5.1.2 Considerações Finais

Percebe-se, pela inclusão de apenas uma propriedade transversal, que o sistema obteve melhorias com o uso da ferramenta desenvolvida. O interesse de Colaboração é adicionado através de recursos gráficos e não requer conhecimento do desenvolvedor da sintaxe de *AspectJ* ou de qualquer outra linguagem de programação orientada a aspectos.

O fato de que o desenvolvedor, usuário do CM2JADE, não necessita de dedicar-se ao aprendizado de qualquer outra linguagem de desenvolvimento, e sua sintaxe é uma característica importante da ferramenta, pois pode diminuir a inércia na utilização de abstrações orientadas as aspectos em aplicações comerciais e que demandam prazos específicos de desenvolvimento e não dispõem de tempo de aprendizado de novas técnicas de modularização de requisitos transversais.

Estudos comparativos com dois cenários diferentes puderam ser realizados, também com a ajuda da ferramenta. A janela gráfica apresentada na figura 5.16 apresenta os dados das propriedades transversais adicionadas com a ferramenta, e a funcionalidade de ativar ou desativar uma propriedade declarada. Nesse caso, a propriedade adicionada é a de Colaboração. Como os dados para geração dos aspectos envolvidos na inserção de uma propriedade são mantidos em XML, os arquivos que contém os aspectos dessa propriedade podem ser destruídos a qualquer momento e gerados novamente, afim de estudar o comportamento dos agentes alternando entre o *status* da propriedade.

O Sistema Multiagente em questão desempenha, após a adição de comportamento colaborativo, a mesma função para a qual foi construído, com menor número de iterações. Isso resulta em economia de tempo de processamento, mesmo que os agentes não tenham sido alocados em *hosts* diferentes.

O CM2JADE apresenta fácil integração com o *Eclipse*, uma vez que foi empacotada como um *plugin*. Portanto é automaticamente reconhecida como parte do *workbench* quando o plugin é adicionado no diretório recipiente de *plugins*.

O estudo de caso apresentado nesse capítulo apresenta os recursos disponíveis no CM2JADE, mecanismos internos de funcionamento e resultados após a inserção da propriedade transersal.

No capítulo seguinte serão apresentadas as conclusões finais do trabalho e perspectivas futuras.

Capítulo 6

Conclusões e perspectivas futuras

A abstração orientada a aspectos contribui para o entendimento e manutenção dos sistemas multiagentes, reduz o espalhamento de código e modularizas propriedades transversais inerentes ao agente. A inversão de controle proporcionada pelas técnicas orientadas a aspectos, permitem que o código transversal antes acoplado ao código, numa abordagem orientada a objetos, seja escrito em arquivos de aspectos, e inserido nos pontos de junção após a fase de recomposição aspectual.

Os pontos de junção gerados a partir dos dados campos editáveis preenchidos nas interfaces gráficas da ferramenta podem referenciar chamadas de um ou mais métodos, de um classe ou de uma família de classes pela especificação do nome das classes, ou até mesmo de uma classe e suas subclasses. Esse recurso é importante para que um comportamento seja especificado uma única vez e seja executado em todas as classes de agente para as quais o comportamento é pertinente.

O mecanismo apresentado nesse trabalho utiliza recursos de abstrações orientadas a aspectos e adiciona comportamento tranversal em um ou mais papéis de agentes de um dado sistema. O sistema muda de comportamento, porém o código construído permanece inalterado. A melhora obtida com a construção da ferramenta na transparência, clareza e entendimento é a mesma proporcionada pelo uso de quaisquer técnicas de modularização de requisitos transversos, porém seu gerenciamento é

tratado de forma gráfica nesse trabalho.

A simulação dos agentes em cenários que prevêm a existência de determinadas propriedades pode ser realizada com o uso do CM2JADE e proporciona ao desenvolvedor a possibilidade de medir ganhos reais com a inserção da(s) propridade(s) adicionadas. Para n propriedades adicionadas ao agente com a ferramenta, existem 2^n cenários a serem estudados.

A independência do mecanismo contruído em relação a determinada versão do *framework* de construção de sistemas multiagentes permite que, caso uma decisão de projeto seja tomada para migrar de versão do *framework JADE*, o mesmo *plugin* seja utilizado até que o método de adição de comportamento da classe *Behaviour* continue ativo.

Uma versão melhorada da ferramenta pode ser construída, em que o corpo principal do agente, na qual as chamadas pelos comportamentos é realizada seja construído de forma gráfica. Isso reduziria as chances do desenvolvedor use o *plugin* e gere pontos de junção não aplicáveis no código. Além do reduzir o tempo de desenvolvimento de sistemas multiagentes e obyenha ganho de produtividade.

O desenvolvedor usuário do *plugin* pode obter, caso o mecanismo apresente futuramente esse recurso de ajuda, uma assistência maior enquanto codifica, na janela gráfica, o código-fonte a ser executado. Assim o usuário tem informações, durante a codificação, sobre as assinaturas dos métodos utilizados.

A ferramenta pode ser aplicada, ainda, em ambientes corporativos para constatar sua usabilidade e seus benefícios em projetos reais de desenvolvimento de *software*. A aplicação de questionários aplicados aos usuários da ferramenta bem como a medição de variáveis de produtividade podem ajudar a medir de forma quantitativa os resultados alcançados com a utilização do mecanismo desenvolvido.

Referências Bibliográficas

AMANDI, A. *Programação de Agentes Orientada a Objetos.* Tese (Doutorado) — Instituto de Informática - Universidade Federal do Rio Grande do Sul (UFRGS), Porto Alegre - RS - Brasil, 1997.

AMANDI, A.; PRICE, A. Building object-agents from a software meta-architecture. In: *SBIA '98: Proceedings of the 14th Brazilian Symposium on Artificial Intelligence.* London, UK: Springer-Verlag, 1998. p. 21–30. ISBN 3-540-65190-X.

ATTALI, J.-G.; PAGÈS, G. Approximations of functions by a multilayer perceptron: a new approach. *Neural Networks*, v. 10, n. 6, p. 1069–1081, 1997.

BAKER, J.; HSIEH, W. Runtime aspect weaving through metaprogramming. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development.* New York, NY, USA: ACM, 2002. p. 86–95. ISBN 1-58113-469-X.

BASILI, V. R.; SELBY, R. W.; HUTCHENS, D. H. Experimentation in software engineering. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 12, n. 7, p. 733–743, 1986. ISSN 0098-5589.

BELLIFEMINE, F. et al. Jade: A white paper. *EXP in search of innovation*, Telecom Italia Lab (TILAB), v. 3, n. 3, p. 6–19, 2003.

BELLIFEMINE, F.; POGGI, A.; RIMASSA, G. JADE - a FIPA-compliant agent framework. In: *Proceedings of the Practical Applications of Intelligent Agents.* [S.l.: s.n.], 1999.

BELLIFEMINE, F. L.; CAIRE, G.; GREENWOOD, D. *Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology)*. [S.l.]: John Wiley & Sons, 2007. ISBN 0470057475.

BENEDICENTI, L. Rethinking smart objects: building artificial intelligence with objects. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 25, n. 3, p. 59–59, 2000. ISSN 0163-5948.

BERGMANS, L.; AKSIT, M. Composing crosscutting concerns using composition filters. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 51–57, 2001. ISSN 0001-0782.

BOEHM, B. W. Software engineering economics. Springer-Verlag New York, Inc., New York, NY, USA, p. 641–686, 2002.

BOND, A. H.; GASSER, L. (Ed.). *Distributed Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1991. ISBN 0-934613-63-X.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *Unified Modeling Language User Guide*. [S.l.]: Addison-Wesley Professional, 1999. ISBN 0321267974.

BRAY, T. et al. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. [S.l.], August 2006.

BRIOT, J.-P. Agents and concurrent objects. *IEEE Concurrency*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 6, n. 4, p. 74–77,81, 1998. ISSN 1092-3063.

BRUGGE, M. H. ter et al. License plate recognition. CRC Press, Inc., Boca Raton, FL, USA, p. 261–296, 1999.

CABRI, G.; LEONARDI, L.; ZAMBONELLI, F. Separation of concerns in agent applications by roles. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2002. p. 430–438. ISBN 0-7695-1588-6.

CHAVES, R. A. *Aspectos e MDA. Criando modelos executáveis baseados em aspectos*. Dissertação (Mestrado) — Universidade Federal de Santa Catarina (UFSC), Florianópolis - SC - Brasil, 2004. Dissertação de Mestrado em Ciência da Computação.

CHEN, B.; SADAOUI, S. A generic formal framework for constructing agent interaction protocols. *International Journal of Software Engineering and Knowledge Engineering*, v. 15, n. 1, p. 61–85, 2005.

CLARKE, S.; WALKER, R. J. Towards a standard design language for aosd. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002. p. 113–119. ISBN 1-58113-469-X.

CZARNECKI, K.; EISENECKER, U. W. *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-30977-7.

DAI, N.; MANDEL, L.; RYMAN, A. *Eclipse web tools platform: developing java™ web applications*. [S.l.]: Addison-Wesley Professional, 2007. ISBN 0321396855.

DEMAZEAU, Y.; MÜLLER, J. *Decentralized Artificial Intelligence*. Amsterdam, Holland: Elsevier Science Publishers B.V., 1990.

D'HONDT, M.; GYBELS, K.; JONCKERS, V. Seamless integration of rule-based knowledge and object-oriented functionality with linguistic symbiosis. In: *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2004. p. 1328–1335. ISBN 1-58113-812-1.

DIJKSTRA, E. W. Book. *A discipline of programming*. [S.l.]: Prentice-Hall, Englewood Cliffs, N.J. :, 1976. xvii, 217 p. ; p. ISBN 013215871.

ECLIPSE. 2008. Disponível em <http://www.eclipse.org>. Acessado em 03/04/2008.

EKDAHL, B. How autonomous is an autonomous agent? In: *Proceedings of the 5th Conference on Systemic, Cybernetics and Informatics (SCI 2001)*. Orlando, Flórida, USA: [s.n.], 2001. p. 22–25.

ELRAD, T. et al. Discussing aspects of aop. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 33–38, 2001. ISSN 0001-0782.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 29–32, 2001. ISSN 0001-0782.

FRANKLIN, S.; GRAESSER, A. Is it an agent, or just a program?: A taxonomy for autonomous agents. In: *ECAI '96: Proceedings of the Workshop on Intelligent Agents III, Agent Theories, Architectures, and Languages*. London, UK: Springer-Verlag, 1997. p. 21–35. ISBN 3-540-62507-0.

FRANKLIN, S. P. *Artificial minds*. Cambridge, MA: MIT Press, 1995.

GAMMA, E. et al. *Design Patterns – Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1995.

GARCIA, A. et al. C.: Promoting advanced separation of concerns in intra-agent and inter-agent software engineering. In: *In: Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems (ASoC) at OOPSLA'2001*. [S.l.: s.n.], 2001.

GARCIA, A. et al. Software engineering for large-scale multi-agent systems - selmas 2006: workshop report. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 31, n. 5, p. 24–32, 2006. ISSN 0163-5948.

GARCIA, A.; LUCENA, C. Taming heterogeneous agent architectures with aspects. *Communications of the ACM*, 2007.

GARCIA, A.; LUCENA, C. Taming heterogeneous agent architectures. *Commun. ACM*, ACM, New York, NY, USA, v. 51, n. 5, p. 75–81, 2008. ISSN 0001-0782.

GARCIA, A. et al. Separation of concerns in multi-agent systems: An empirical study. In: DOE, A. (Ed.). *Software Engineering for Multi-Agent Systems II*. [S.l.]: Springer Berlin / Heidelberg, 2004. p. 49–72.

GARCIA, A. F. *Objetos e Agentes: Uma Abordagem Orientada a Aspectos*. Tese (Doutorado) — PUC-Rio, Rio de Janeiro - RJ - Brasil, 2004.

GARCIA, A. F. et al. Aspects in agent-oriented software engineering: Lessons learned. In: *AOSE*. [S.l.: s.n.], 2005. p. 231–247.

GARCIA, A. F.; LUCENA, C. J. P. de; COWAN, D. D. Agents in object-oriented software engineering. *Software: Practice and Experience*, John Wiley & Sons, Inc., New York, NY, USA, v. 34, n. 5, p. 489–521, 2004. ISSN 0038-0644.

GARCIA, A. F. et al. (Ed.). *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications [the book is a result of SELMAS 2002]*, v. 2603 de *Lecture Notes in Computer Science*, (Lecture Notes in Computer Science, v. 2603). [S.l.]: Springer, 2003. ISBN 3-540-08772-9.

GAZOLLA, P. A. F. M. *Mecanismo visual baseado em aspectos para automatização de logging*. Dissertação (Mestrado) — Universidade Federal de Viçosa, UFV - Viçosa - MG - Brasil, 2008. Dissertação de Mestrado.

GOSLING, J. et al. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. [S.l.]: Addison-Wesley Professional, 2005. ISBN 0321246780.

GUESSOUM, Z.; BRIOT, J.-P. From active objects to autonomous agents. *IEEE Concurrency*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 7, n. 3, p. 68–76, 1999. ISSN 1092-3063.

GUESSOUM, Z.; BRIOT, J. pierre (Ed.). *From Active Objects to Autonomous Agents*, v. 7 de *3*, (3, v. 7). [S.l.]: Springer, 1999. 68–76 p.

HARRISON, W.; OSSHER, H. Subject-oriented programming: a critique of pure objects. In: *OOPSLA '93: Proceedings of the eighth annual conference on*

Object-oriented programming systems, languages, and applications. New York, NY, USA: ACM, 1993. p. 411–428. ISBN 0-89791-587-9.

HAYES-ROTH, B.; BROWNSTON, L.; SINCOFF, E. *Directed Improvisation by Computer Characters*. [S.l.], 1995.

HENDERSON-SELLERS, B. Agent-based software development methodologies, white paper. In: *Summary of Workshop, ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*. [S.l.: s.n.], 2002.

HUHNS, M.; SINGH, M. Agents and multi-agent systems: Themes, approaches, and challenges. *Readings in Intelligent Agents*, Palo Alto: Morgan Kaufmann, 1998.

HUHNS, M. N. Agent foundations for cooperative information systems. In: *In: Proc. s of the Third International Conference on the Practical Applications of Intelligent Agents and Multi-Agent Technology; London 1998; Edited by H.S. Nwana and D.T. Ndumu*. [S.l.: s.n.], 1998.

HUHNS, M. N. Ieee internet computing: Agents on the web - the sentient web. *IEEE Distributed Systems Online*, v. 4, n. 11, 2003.

JENNINGS, N. R. The archon system and its applications. *Second International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, 1994.

JENNINGS, N. R. Agent-oriented software engineering. In: *IEA/AIE '99: Proceedings of the 12th international conference on Industrial and engineering applications of artificial intelligence and expert systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. p. 4–10. ISBN 3-540-66076-3.

JENNINGS, N. R. On agent-based software engineering. *Artificial Intelligence*, v. 117, p. 277–296, 2000.

JENNINGS, N. R.; WOOLDRIDGE, M. A roadmap of agent research and development. *Autonomous Agents and Multi-Agent Systems*, v. 1, p. 7–38, 1998.

KENDALL, E. A. Role model designs and implementations with aspect-oriented programming. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 34, n. 10, p. 353–369, 1999. ISSN 0362-1340.

KICZALES, G. et al. Aspect-oriented programming. In: *In ECOOP'97—Object-Oriented Programming, 11th European Conference, LNCS 1241*. [S.l.: s.n.], 1997. p. 220–242. ISBN 3-540-42206-4.

KICZALES, G. et al. Getting started with aspectj. *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 59–65, 2001. ISSN 0001-0782.

KICZALES, G. et al. An overview of AspectJ. *Lecture Notes in Computer Science*, v. 2072, p. 327–355, 2001.

KICZALES, G. et al. An overview of aspectj. In: *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2001. p. 327–353. ISBN 3-540-42206-4.

KICZALES, G. et al. Aspect-oriented programming. In: AKsIT, M.; MATSUOKA, S. (Ed.). *Proceedings European Conference on Object-Oriented Programming*. Berlin, Heidelberg, and New York: Springer-Verlag, 1997. v. 1241, p. 220–242.

KULESZA, U. et al. A generative approach for multi-agent system development. In: *SELMAS*. [S.l.: s.n.], 2004. v. 3390, p. 52–69. ISBN 0302-9743.

KULESZA, U.; SANT'ANNA, C.; LUCENA, C. Técnicas de projeto orientado a aspectos. In: *: XIX Simpósio Brasileiro de Engenharia de Software (SBES)*. Uberlândia, MG: [s.n.], 2005.

LADDAD, R. *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT, USA: Manning Publications Co., 2003. ISBN 1930110936.

LANGE, D. B.; MITSURU, O. *Programming and Deploying Java Mobile Agents Aglets*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN 0201325829.

LEAKE, D. B.; PLAZA, E. Iccbr '97: Proceedings of the second international conference on case-based reasoning research and development. In: . London, UK: Springer-Verlag, 1997. ISBN 3-540-63233-6.

LEMOS, O. A. L. et al. Control and data flow structural testing criteria for aspect-oriented programs. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 80, n. 6, p. 862–882, 2007. ISSN 0164-1212.

LI, B.; ZENG, Z.-Y.; ZHOU, J.-Z. An algorithm for license plate recognition system for non-stop toll stations. *iih-msp*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 873–876, 2008.

LICENSE Plate Recognition Method for Inclined Plates Outdoors. In: ICIIS '99: Proceedings of the 1999 International Conference on Information Intelligence and Systems. Washington, DC, USA: IEEE Computer Society, 1999. p. 304. ISBN 0-7695-0446-9.

LIEBERHERR, K. J. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. [S.l.]: PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X.

LOBATO, C. et al. A modular implementation framework for code mobility. In: *Mobility '06: Proceedings of the 3rd international conference on Mobile technology, applications & systems*. New York, NY, USA: ACM, 2006. p. 57. ISBN 1-59593-519-3.

LOS, C. The prejudices of least squares, principal components and common factors schemes. *Computers Math. Appl.*, v. 17, p. 1269–1283, 1989.

LUCENA, C. et al. *Software Engineering for Multi-Agent Systems II*. [S.l.]: SpringerVerlag, 2004. ISBN 3540211829.

MAES, P. Agents that reduce work and information overload. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 7, p. 30–40, 1994. ISSN 0001-0782.

MEHMOOD, T.; ASHRAF, N.; RASHEED, K. Framework for separation of performance concerns and improved modularity in multi agent systems using aspects. In: *Software Engineering Research and Practice 2005*. [S.l.: s.n.], 2005. p. 754–757.

MOORE, B. C. Principal component analysis in linear systems: controllability, observability, and model reduction. *IEEE Transactions on Automatic Control*, AC-26, p. 17–32, 1981.

MURPHY, G. C. et al. Does aspect-oriented programming work? *Commun. ACM*, ACM, New York, NY, USA, v. 44, n. 10, p. 75–77, 2001. ISSN 0001-0782.

NEAGU, N. et al. Ls/atn: Reporting on a successful agent-based solution for transport logistics optimization. In: *DIS '06: Proceedings of the IEEE Workshop on Distributed Intelligent Systems: Collective Intelligence and Its Applications*. Washington, DC, USA: IEEE Computer Society, 2006. p. 213–218. ISBN 0-7695-2589-X.

OLIVEIRA, E. J. S. *Comunicação Segura e Confiável para Sistemas Multiagentes Adaptando Especificações XML*. Dissertação (Mestrado) — Universidade Federal do Maranhão, UFMA, 2006. Dissertação de Mestrado em Engenharia de Eletricidade.

PACE, D.; CAMPO, J.; SORIA, M. Architecting the design of multi-agent organizations with proto-frameworks. *Software Engineering for Large-Scale Multi-Agent Systems II Research Issues and Practical Applications Lecture Notes in Computer Science*, Springer, Berlin, Alemania. 2004, v. 2940, p. 165–181, 2004. ISSN 3-540-21182-9.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, ACM, New York, NY, USA, v. 15, n. 12, p. 1053–1058, 1972. ISSN 0001-0782.

PARUNAK, H. V. D. Manufacturing experience with the contract net. In: HUHNS, M. (Ed.). *Distributed Artificial Intelligence*. [S.l.]: Pitman Publishing: London and Morgan Kaufmann: San Mateo, CA, 1987. p. 285–310.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach*. [S.l.]: McGraw-Hill Higher Education, 2001. ISBN 0072496681.

RICH, E.; KNIGHT, K. *Artificial Intelligence*. [S.l.]: McGraw-Hill Higher Education, 1990. ISBN 0070522634.

RUCK, D. W. et al. The multilayer perceptron as an approximation to a bayes optimal discriminant function. *Neural Networks, IEEE Transactions on*, v. 1, n. 4, p. 296–298, 1990.

RUSSELL, J. S.; NORVIG, P. *Artificial Intelligence: A Modern Approach. 2nd edition*. [S.l.]: Pearson, 2002.

SHOHAM, Y. Agent-oriented programming. *Artif. Intell.*, Elsevier Science Publishers Ltd., Essex, UK, v. 60, n. 1, p. 51–92, 1993. ISSN 0004-3702.

SIAH, Y. K. et al. J.p.callan d.d.lewis, r.e.shapire and r.papka. training algorithms for linear text classifiers. In: *In Proc. ACM SIGIR Conf.* [S.l.: s.n.], 1996. p. 298–315.

SMITH, D. C.; CYPHER, A.; SPOHRER, J. Kidsim: programming agents without a programming language. *Commun. ACM*, ACM, New York, NY, USA, v. 37, n. 7, p. 54–67, 1994. ISSN 0001-0782.

STEIN, D.; HANENBERG, S.; UNLAND, R. A uml-based aspect-oriented design notation for aspectj. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2002. p. 106–112. ISBN 1-58113-469-X.

TARR, P. et al. N degrees of separation: multi-dimensional separation of concerns. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999. p. 107–119. ISBN 1-58113-074-0.

TEIXEIRA, R. A. *Treinamento de Redes Neurais Artificiais através da otimização multi-objetivo - Uma nova abordagem para o equilíbrio entre a polarização e a*

variância. Tese (Doutorado) — Universidade Federal de Minas Gerais, Universidade Federal de Minas Gerais, 2001.

TEIXEIRA, R. A. et al. Improving generalization of mlps with multi-objective optimization. *Neurocomputing 35*, v. 1, p. 189–194, 2000.

TSENG, P.-C. et al. Adaptive car plate recognition in qos-aware security network. *ssiri*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 120–127, 2008.

UBAYASHI, N.; TAMAI, T. Separation of concerns in mobile agent applications. In: *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. London, UK: Springer-Verlag, 2001. p. 89–109. ISBN 3-540-42618-3.

UBAYASHI, N.; TAMAI, T. Separation of concerns in mobile agent applications. In: *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. London, UK: Springer-Verlag, 2001. p. 89–109. ISBN 3-540-42618-3.

WALKER, D.; ZDANCEWIC, S.; LIGATTI, J. A theory of aspects. In: *ICFP '03: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2003. p. 127–139. ISBN 1-58113-756-7.

WALKER, R. J.; BANIASSAD, E. L. A.; MURPHY, G. C. An initial assessment of aspect-oriented programming. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999. p. 120–130. ISBN 1-58113-074-0.

WAZLAWICK, R. S. *Análise e Projeto de Sistemas de Informação Orientados a Objetos*. Rio de Janeiro - RJ - Brasil: Campus, 2004. 253 p. ISBN 8535215646.

WOOLDRIDGE, M. *An Introduction to MultiAgent Systems*. [S.l.]: Baffins Lane: LTD, 1999.

WOOLDRIDGE, M.; CIANCARINI, P. Agent-oriented software engineering: the state of the art. In: *First international workshop, AOSE 2000 on Agent-oriented*

software engineering. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2001. p. 1–28. ISBN 3-540-41594-7.

WOOLDRIDGE, M.; JENNINGS, N. R. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, v. 10, n. 2, p. 115–152, 1995.

XIE, T. et al. Detecting redundant unit tests for aspectj programs. In: *ISSRE '06: Proceedings of the 17th International Symposium on Software Reliability Engineering*. Washington, DC, USA: IEEE Computer Society, 2006. p. 179–190. ISBN 0-7695-2684-5.

XU, D.; XU, W. State-based incremental testing of aspect-oriented programs. In: *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2006. p. 180–189. ISBN 1-59593-300-X.

ZAKI, O. et al. Detecting faults in heterogeneous and dynamic systems using dsp and an agent-based architecture. *Eng. Appl. Artif. Intell.*, Pergamon Press, Inc., Tarrytown, NY, USA, v. 20, n. 8, p. 1112–1124, 2007. ISSN 0952-1976.

ZHOU DEBRA RICHARDSON, H. Z. Y. Towards a practical approach to test aspect-oriented software. In: *TECOS 2004: Workshop on Testing Component-Based Systems*. [S.l.: s.n.], 2004.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo