CENTRO ESTADUAL DE EDUCAÇÃO TECNOLÓGICA PAULA SOUZA
MESTRADO EM TECNOLOGIA

WILLIAN FERREIRA PEIXOTO

UMA PROPOSTA PARA CONTROLE DE ACESSO BASEADO EM PAPÉIS NA ARQUITETURA DO AGENTE COM A PROGRAMAÇÃO ORIENTADA A ASPECTOS

SÃO PAULO
NOVEMBRO – 2009

WILLIAN FERREIRA PEIXOTO

UMA PROPOSTA PARA CONTROLE DE ACESSO BASEADO EM PAPÉIS NA ARQUITETURA DO AGENTE COM A PROGRAMAÇÃO ORIENTADA A ASPECTOS

Dissertação apresentada como exigência parcial para obtenção do Título de Mestre em Tecnologia no Centro Estadual de Educação Tecnológica Paula Souza, no Programa de Mestrado em Tecnologia: Gestão Desenvolvimento e Formação, sob orientação da Prof.ª Dr.ª Márcia Ito.

SÃO PAULO
NOVEMBRO – 2009

Peixoto, Willian Ferreira

P379u Uma proposta para controle de acesso baseado em papéis na arquitetura do agente com a programação orientada a aspectos. – São Paulo : CEETEPS, 2009.
115 f.

Dissertação (Mestrado) - Centro Estadual de Educação Tecnológica Paula Souza, 2009.

1. Arquitetura de software . 2. Sistemas multiagentes . 3. Controle de acesso . 4. Programação orientada a aspectos . 5. SMA. 6. RBAC. I. Título.

WILLIAN FERREIRA PEIXOTO

UMA PROPOSTA PARA CONTROLE DE ACESSO BASEADO EM PAPÉIS NA ARQUITETURA DO AGENTE COM A PROGRAMAÇÃO ORIENTADA A ASPECTOS

PROFª. DRª. MÁRCIA ITO

PROF. DR. CARLOS HIDEO ARIMA

PROF. DR. MAURICIO AMARAL DE ALMEIDA

São Paulo, 27 de novembro de 2009

Dedicatória

À minha família que sempre me apoiou.

Agradecimentos

À minha orientadora Prof.ª Dr.ª Marcia Ito, pela dedicação, motivação e confiança.

Aos professores Prof. Dr. Carlos Hideo Arima e Prof. Dr. Maurício Amaral de Almeida pela orientação e apoio para conclusão deste trabalho.

À minha amiga Marie Moritani, que fez a revisão do texto da monografia.

Enfim, a todas as pessoas e entidades que direta ou indiretamente tornaram possível a elaboração deste trabalho.

“Uma biblioteca permite que se procure Marx,
encontre-se Schopenhauer
e se requisite a Bíblia”.
Ernst R. Hauschka

RESUMO

PEIXOTO, W. F. **Uma Proposta para Controle de Acesso Baseado em Papéis na Arquitetura do Agente com a Programação Orientada a Aspectos**. 2009. 115f. Dissertação (Mestrado), Centro Estadual de Educação Tecnológica Paula Souza, São Paulo.

Um agente em geral, não se encontra sozinho num sistema, mas sim como integrante de um Sistema Multiagentes que é composto por um conjunto de diferentes tipos de agentes distribuídos em um ou mais ambientes. Estes agentes desempenham papéis distintos e colaboram entre si para execução de tarefas. Os agentes distribuídos por estes ambientes podem sofrer inúmeras ameaças de segurança onde o acesso não autorizado causado pela ausência ou pelo fraco controle de acesso pode ser a falha de segurança utilizada para uma investida de ataque. Desse modo, o controle de acesso torna-se uma preocupação essencial no desenvolvimento de sistemas orientados a agentes seguros. No entanto, a implementação deste interesse não é uma tarefa trivial, pois devido sua natureza transversal, quando implementado frequentemente causa sintomas indesejáveis que impactam de forma negativa no sistema. Consequentemente, em meio ao desafio de implementar o interesse de controle de acesso nos agentes, os desenvolvedores devem estar atentos também à necessidade de produzir sistemas com um nível avançado de separação de interesses, pois a modularização inadequada de interesses transversais gera esforços indesejáveis na compreensão, reuso e evolução de projetos orientados a agentes. A ideia da separação de interesses é antiga e este princípio é suportado parcialmente por paradigmas de desenvolvimento como a programação estruturada e a programação orientada a objetos. A programação orientada a aspectos é apontada como um paradigma complementar aos existentes, cujo objetivo é prover suporte à separação avançada de interesses transversais em módulos separados, chamados de aspectos. Desta forma, através da utilização dos aspectos e do modelo de controle de acesso RBAC, este trabalho de mestrado propõe uma arquitetura para o controle de acesso baseado em papéis que proporciona além do controle de acesso, a separação avançada deste interesse.

Palavras-chave: Sistemas Multiagentes, SMA, Arquitetura de Software, Controle de Acesso, Autorização, RBAC, Programação Orientada a Aspectos

ABSTRACT

PEIXOTO, W. F. **Uma Proposta para Controle de Acesso Baseado em Papéis na Arquitetura do Agente com a Programação Orientada a Aspectos**. 2009. 115f. Dissertação (Mestrado), Centro Estadual de Educação Tecnológica Paula Souza, São Paulo.

An agent is not commonly found alone in a software system, usually an agent is an element of a Multi-agent System that is composed of various different kinds of agents that run in one or more environments. Agents play different roles and can collaborate with each other to perform some tasks. Once agents are distributed in multiple environments, they can suffer attacks. The unauthorized access caused by the absence or weak authorization can be the security vulnerability used for such attacks. Thus, authorization turns out to be an essential concern in the development of safe agent-oriented systems. However, the implementation of this concern is not a trivial task, due to its crosscutting nature, unwanted symptoms that affect in a negative way in the system are often seen when implemented. Consequently, amidst the challenge of implementing the authorization concern in agents, the developers should be aware of the need to produce systems with an advanced level of separation of concerns, since the poor modularization of the crosscutting concerns can generate extra efforts to understand, reuse and improve agent-oriented projects. The principle of separation of concerns isn't new and is supported in part by some development paradigms like procedural and object-oriented programming. The aspect-oriented programming is indicated as a paradigm that complement the existing ones, which aims to provide better support in separating the crosscutting concerns into modules known as aspects. Through the use of aspects and Role-Based Access Control model, this work proposes an agent architecture that provides not only the authorization based on roles, but the advanced separation on this crosscutting concern.

Keywords: Multi-agent Systems, MAS, Software Architecture, Access Control, Authorization, RBAC, Aspect-Oriented Programming

LISTAS DE FIGURAS

Figura 3.1 – Autenticação, controle de acesso e outros elementos de segurança..... 30
Figura 3.2 – Níveis do modelo NIST RBAC. 32
Figura 3.3 – RBAC Base. 34
Figura 3.4 – RBAC Base com suporte a sessões. 35
Figura 3.5 – RBAC Hierárquico. 36
Figura 3.6 – Exemplos de hierarquias de papéis. 37
Figura 3.7 – Separação estática de responsabilidades..... 39
Figura 3.8 – Separação dinâmica de responsabilidades. 39
Figura 4.1 – Visão de um sistema como uma composição de múltiplos interesses. 43
Figura 4.2 – Analogia com um prisma para a decomposição dos interesses. 43
Figura 4.3 – Decomposição de interesses numa visão multidimensional. 44
Figura 4.4 – Emaranhamento de código. 45
Figura 4.5 – Espalhamento de código por blocos de código duplicados. 46
Figura 4.6 – Espalhamento de código por blocos de código complementares..... 46
Figura 4.7 – Mapeamento de um espaço de N dimensões utilizando uma linguagem de uma dimensão..... 47
Figura 4.8 – Interesse transversal implementado em sistemas OO e OA. 48
Figura 4.9 – Combinador de aspectos. 50
Figura 4.10 – Composição de aspectos. 51
Figura 5.1 – Modelo RBAC-MAS. 53
Figura 5.2 – Agente *Supervisor* responsável por autenticar os agentes de diferentes ambientes. 58
Figura 5.3 – Diagrama de classes dos aspectos de segurança. 60
Figura 5.4 – Ilustração da atuação do aspecto *ProactiveAuthenticationAspect*. 66
Figura 5.5 – Representação gráfica da atuação do aspecto *ReactiveAuthenticationAspect*. 67
Figura 5.6 – Representação gráfica da atuação do aspecto *ProactiveAccessControlAspect*. .. 68
Figura 5.7 – Ilustração da atuação do aspecto *ReactiveAccessControlAspect*. 69
Figura 6.1 – Analogia entre teste de sistema e teste de política de segurança. 72
Figura 6.2 – Diagrama de instalação..... 74

Figura 6.3 – Diagrama de sequência das interações dos agentes executados no ambiente interno. 74
Figura 6.4 – Diagrama de sequência das interações dos agentes executados no ambiente externo. 75

LISTAS DE QUADROS

Quadro 2.1 – Exemplo de mensagem FIPA ACL com o ato de comunicação *request*. 26
Quadro 5.1 – Código-fonte de um agente no JADE. 56
Quadro 5.2 – Exemplo de aspecto. 65
Quadro 6.1 – Resultado obtido pelo Teste 1. 79
Quadro 6.2 – Resultado obtido pelo Teste 2. 80
Quadro 6.3 – Resultado obtido pelo Teste 3. 81
Quadro 6.4 – Resultado obtido pelo Teste 4. 82
Quadro 6.5 – Resultado obtido pelo Teste 5. 83
Quadro 6.6 – Resultado obtido pelo Teste 6. 84
Quadro 6.7 – Resultado obtido pelo Teste 7. 85
Quadro 6.8 – Resultado obtido pelo Teste 8. 87

LISTAS DE TABELAS

Tabela 2.1 – Propriedades do agente. 20
Tabela 3.1 – Variações do RBAC organizadas pelo nível e suas principais funcionalidades. 33
Tabela 5.1 – PCDs primitivos utilizados..... 63
Tabela 6.1 – Política de controle de acesso RBAC. 76
Tabela 6.2 – Cenários de teste. 77
Tabela 6.3 – Descrição dos elementos utilizados para execução do Teste 1. 78
Tabela 6.4 – Descrição dos elementos utilizados para execução do Teste 2. 79
Tabela 6.5 – Descrição dos elementos utilizados para execução do Teste 3. 80
Tabela 6.6 – Descrição dos elementos utilizados para execução do Teste 4. 81
Tabela 6.7 – Descrição dos elementos utilizados para execução do Teste 5. 83
Tabela 6.8 – Descrição dos elementos utilizados para execução do Teste 6. 84
Tabela 6.9 – Descrição dos elementos utilizados para execução do Teste 7. 85
Tabela 6.10 – Descrição dos elementos utilizados para execução do Teste 8. 86

LISTAS DE ABREVIATURAS E SIGLAS

ACL	Agent Communication Language
CSI	Computer Security Institute
DAC	Discretionary Access Control
FIPA	Foundation for Intelligent Physical Agents
INCITS	International Committee for Information Technology
JADE	Java Agent Development Framework
KIF	Knowledge Interchange Format
KQML	Knowledge and Querying Manipulation Language
MAC	Mandatory Access Control
NIST	National Institute of Standards and Technology
OA	Orientação a Aspectos
OMG	Object Management Group
OO	Orientação a Objetos
PCD	Pointcut Designator
POA	Programação Orientada a Aspectos
RBAC	Role-Based Access Control
SMA	Sistemas Multiagentes
VPN	Virtual Private Network

SUMÁRIO

1. Introdução 14
1.1. Objetivos do Trabalho 16
1.2. Motivações e Resultados Esperados 16
1.3. Método e Desenvolvimento do Trabalho 17
1.4. Organização do Trabalho 18
2. Agentes 19
2.1. Autonomia 23
2.2. Interação 24
2.3. Papéis 26
3. Autenticação e Controle de Acesso 28
3.1. Modelos de Controle de Acesso 30
3.1.1. Modelo NIST RBAC 31
4. Separação de Interesses com a Programação Orientada a Aspectos 41
4.1. Separação de Interesses 41
4.2. Modularização 45
4.3. Programação Orientada a Aspectos 47
5. Controle de Acesso RBAC na Arquitetura do Agente 52
5.1. Camada de Controle de Acesso 57
5.2. Camada de Aspectos de Segurança 59
5.2.1. Componentes do AspectJ 61
5.2.2. Aspectos de autenticação 65
5.2.3. Aspectos de controle de acesso 67
6. Testes para Verificação do Controle de Acesso 71
6.1. Implementação para o Teste 73
6.2. Testes e Simulações 77
6.3. Análise dos Resultados 87
7. Conclusão 89
Referências 91
Glossário 99
Apêndice A - Códigos-fonte dos Aspectos 103

1. Introdução

Um agente é entendido como um software capaz de resolver um problema em particular de forma autônoma e ainda interagir com um ambiente para possibilitar a resolução de problemas que individualmente não conseguiria (WOOLDRIDGE, 2002).

Normalmente um agente não é encontrado sozinho num sistema. Pode se comunicar com outros agentes ou ambientes e com frequência esta comunicação é realizada através da troca de mensagens padronizadas (OMG, 2000a).

Segundo o *Object Management Group* (OMG) (2000a), geralmente os agentes são executados num ambiente computacional distribuído e um sistema de agentes executado nesta condição pode sofrer inúmeras ameaças de segurança como a interceptação, modificação ou destruição de mensagens.

A possibilidade de ocorrerem problemas de segurança tende a ser maior em ambientes abertos, como a *Internet* ou uma *Virtual Private Network* (VPN), mas podem ocorrer também em qualquer ambiente onde as entidades não são totalmente conhecidas, compreendidas e administradas por um único grupo.

Mouratidis et al. (2005) relatam que há uma necessidade crescente de armazenamento e gerenciamento de informações que são consideradas confidenciais como por exemplo, dados médicos, financeiros e outros tantos dados privados. Portanto, desenvolver sistemas seguros para o manuseio dessas informações torna-se uma necessidade e não uma simples opção, pois incidentes de segurança podem ocasionar exposição de informações confidencias e perdas financeiras (MOURATIDIS; GIORGINI; SCHUMACHER, 2003).

A preocupação com a segurança em sistemas de informação aumenta continuamente devido ao crescente emprego dos computadores nos setores empresariais, governamentais e de ensino e ao aumento da utilização da *Internet*. Esta preocupação é agravada pelos contínuos registros de incidentes de segurança (WANGHAM, 2004).

Levantamentos feitos pela pesquisa *Computer Crime & Security Survey* publicada pelo *Computer Security Institute* (CSI) em 2008 indicam que os incidentes de segurança mais frequentes ocorrem devido: (I) a atuação de vírus, (II) abuso interno, ou seja, incidente de segurança causado por funcionários insatisfeitos, (III) roubo de *laptops* e outros dispositivos portáteis e (IV) **acesso não autorizado a sistemas de informação**.

De acordo com o *National Institute of Standards and Technology* (NIST) é possível classificar as ameaças de segurança que um agente ou ambiente pode sofrer, em quatro

categorias: (I) ameaças de um agente atacando um ambiente; (II) agentes atacando outros agentes; (III) um ambiente atacando os agentes e (IV) outras entidades atacando os agentes ou ambientes (JANSEN; KARYGIANNIS, 2000).

Jansen e Karygiannis (2000) afirmam que em todas as categorias citadas com exceção da (III), o acesso não autorizado causado pela ausência ou pelo fraco controle de acesso pode ser a falha de segurança utilizada para uma investida de ataque.

Para garantir a segurança dos recursos do sistema é preciso que a autoridade dos agentes seja verificada pelo sistema de agentes. Este sistema também deve conhecer qual acesso é permitido para cada autoridade. A habilidade de identificar a autoridade dos agentes permite que o sistema de agentes realize o controle de acesso aos recursos (OMG, 2000b).

Desse modo, o controle de acesso é um requisito indispensável num sistema orientado a agentes para proporcionar a mitigação dos riscos de segurança relacionados aos incidentes causados pelo acesso não autorizado.

O modelo de controle de acesso baseado em papéis, mais conhecido como *Role-Based Access Control* (RBAC) (SANDHU; FERRAIOLO; KUHN, 2000), tem sido aplicado em Sistemas Multiagentes (SMA) integrando a abordagem de segurança baseada em papéis com a coordenação e organização dos agentes (MOLESINI; DENTI; OMICINI, 2008).

Apesar de necessária, a implementação do controle de acesso não é uma tarefa trivial, pois devido sua natureza transversal, quando implementado frequentemente causa sintomas indesejáveis no sistema como o emaranhamento e espalhamento de código. Consequentemente, em meio ao desafio de implementar o interesse de controle de acesso nos agentes, os desenvolvedores devem estar atentos também a necessidade de produzir sistemas com um nível avançado de separação de interesses, pois a modularização inadequada de interesses transversais gera esforços indesejáveis no reuso e evolução de projetos orientados a agentes (GARCIA; CHAVEZ; CHOREN, 2006).

Segundo Dijkstra (1976), a separação de interesses sempre foi tida como de importância primária no processo de desenvolvimento, pois é um instrumento básico para se reduzir a complexidade de software. A Orientação a Aspectos (OA) (KICZALES et al., 1997), é um paradigma que promete a separação avançada de interesses através de uma nova abstração, chamada de aspecto, para modularizar os interesses transversais (SANT'ANNA, 2004).

1.1. Objetivos do Trabalho

O objetivo deste trabalho consiste em propor uma arquitetura para implementação do modelo de controle de acesso RBAC nos agentes por meio da Programação Orientada a Aspectos (POA) e desenvolver um mecanismo para controle de acesso RBAC aplicável a sistemas orientados a agentes.

1.2. Motivações e Resultados Esperados

A segurança de informações protege as aplicações contra possíveis ameaças a fim de garantir a continuidade dos negócios, de minimizar prejuízos e maximizar o retorno de investimentos (WANGHAM, 2004).

A preocupação com a segurança há muito tempo está presente no desenvolvimento de sistemas orientados a agentes (WAGNER, 1997; MOURATIDIS; GIORGINI; WEISS, 2003), pois frequentemente os agentes são executados num ambiente distribuído onde interagem trocando mensagens através de redes abertas como a *internet* ou privadas como uma VPN e nestas condições, há grande possibilidade de ocorrerem incidentes de segurança (OMG, 2000a).

O controle de acesso em agentes é aplicado (CABRI; FERRARI; LEONARDI, 2004) como uma forma de minimizar os problemas relativos a falta de segurança em sistemas orientados a agentes e dentre os diferentes modelos de controle de acesso existentes, o RBAC (SANDHU, 1997) é apontado (NAVARRO et al., 2005; XIAO et al., 2007) como o mais flexível e também o mais adequado para sistemas orientados a agentes modelados segundo o conceito de papel (PARTSAKOULAKIS; VOUROS, 2002).

Apesar da existência de algumas propostas de implementação do modelo RBAC para o controle de acesso dos agentes (YAMAZAKI; HIRAISHI; MIZOGUCHI, 2004; MOLESINI; DENTI; OMICINI, 2008; QUILLINAN et al., 2008), nenhuma apresenta a preocupação com a modularização e separação deste interesse e como consequência, conforme constatado por Garcia et al. (2006), a falta de suporte à separação de interesses transversais gera esforços indesejáveis no reuso e evolução de projetos orientados a agentes.

Motivado por estas questões, este trabalho apresenta uma arquitetura para o controle de acesso RBAC nos agentes que proporciona além do controle de acesso, a separação avançada desse interesse.

Todas as funções referentes ao controle de acesso RBAC são modularizadas por um mecanismo criado para este fim. Apesar da grande importância desse mecanismo para a conclusão deste trabalho, a discussão de sua arquitetura e outros aspectos do seu modelo não fazem parte do escopo deste trabalho.

A utilização do mecanismo de controle de acesso pelos agentes é realizada por meio da programação orientada a aspectos, sendo proporcionada desta forma, a separação avançada deste interesse.

1.3. Método e Desenvolvimento do Trabalho

A fase inicial da pesquisa consistiu-se no estudo da orientação a agentes e dos principais modelos de controle de acesso existentes. Em seguida, buscou-se em artigos científicos, dissertações de mestrado e teses de doutorado os modelos e abordagens já pesquisados para a implementação do controle de acesso em agentes.

A partir dos resultados iniciais desta pesquisa, decidiu-se utilizar para o controle de acesso, o modelo RBAC e então, foi realizado um estudo dirigido à adaptação deste modelo ao contexto da orientação a agentes.

Para viabilizar a aplicação deste modelo nos agentes, optou-se por desenvolver um mecanismo de controle de acesso RBAC aplicável a sistemas orientados a agentes. Após a implementação deste mecanismo foi constatado que sua utilização pelos agentes poderia causar impactos negativos na arquitetura, devido a natureza transversal do interesse de controle de acesso, que tende a ficar espalhado e emaranhado com os demais interesses dos agentes. Por conta disto, paradigmas, técnicas e ferramentas relacionadas ao conceito da separação de interesses foram estudadas.

Após este estudo, optou-se pela utilização da orientação a aspectos para implementação do controle de acesso RBAC na arquitetura dos agentes, pois através deste paradigma é possível obter um grau maior na separação de interesses transversais.

Finalmente, foram definidos os cenários das simulações utilizadas para validar a arquitetura proposta.

1.4. Organização do Trabalho

O texto desta dissertação é uma reflexão sobre as diversas etapas que foram cumpridas durante o mestrado e está dividido em sete capítulos. O restante desta dissertação está estruturada da seguinte forma:

O Capítulo 2 – Agentes –, define o conceito de agente e descreve as propriedades de agência, principalmente as propriedades de autonomia, interação e papel. Também é abordada em detalhes a comunicação entre agente por meio de mensagens.

Conceitos fundamentais encontrados no contexto da segurança de sistemas de informação são apresentados no Capítulo 3 – Autenticação e Controle de Acesso –, onde são discutidas as abordagens de autenticação, controle de acesso e por último, o modelo de controle de acesso NIST RBAC é apresentado em detalhes.

No Capítulo 4 – Separação de Interesses com a Programação Orientada a Aspectos –, a questão da separação de interesses e modularização é apresentada, também são detalhados os principais elementos da orientação a aspectos onde é apresentado a AspectJ, uma extensão da linguagem Java para POA.

O Capítulo 5 – Controle de Acesso RBAC na Arquitetura do Agente –, apresenta a arquitetura proposta nesta dissertação. As camadas, seus principais componentes e a forma de atuação dos aspectos de autenticação e controle de acesso nos agentes são discutidos.

As simulações realizadas para o teste da arquitetura proposta, seguida da análise dos resultados, são apresentadas no Capítulo 6 – Testes para Verificação do Controle de Acesso –. Finalmente, o Capítulo 7 – Conclusão –, exprime as considerações gerais, contribuições e possíveis direcionamentos de pesquisas que este trabalho pode oferecer.

2. Agentes

Este capítulo apresenta uma visão geral do conceito de agentes e das propriedades de autonomia, interação e papel que são necessárias para o entendimento deste trabalho.

O termo agente é utilizado em diversas disciplinas da ciência da computação, como a inteligência artificial, sistemas distribuídos, sistemas de aprendizado adaptativo e sistemas especialistas. De forma que é difícil encontrar uma definição sucinta para agentes onde estejam incluídas todas as características que os pesquisadores e desenvolvedores consideram que eles apresentem (SUNDSTED, 1998).

Em geral, os pesquisadores concordam que a autonomia é uma característica fundamental dos agentes (RIBEIRO, 2001).

Segundo Franklin e Graesser (1996), apesar das diversas definições existentes a respeito dos agentes é possível formalizar uma definição onde os agentes são entendidos como programas que estão num ambiente de execução e necessitam possuir as propriedades de reatividade, autonomia, proatividade e continuidade temporária para que realmente sejam considerados agentes.

O OMG (2000a) define agentes autônomos como entidades de software autônomas que interagem com seu ambiente, percebendo-o através de sensores e agindo nele através de atuadores[1]. Esta definição é semelhante a apresentada por Russell e Norvig (2003), que definem o agente como qualquer coisa que pode perceber seu ambiente através de sensores e atuar neste ambiente através de atuadores.

Para definir o termo agente, será utilizada a definição Wooldridge (2002) que entende o agente como um sistema computacional situado em algum ambiente e que é capaz de ações autônomas neste ambiente a fim de alcançar seus objetivos.

Um agente possui uma estrutura de dados interna que se atualiza de acordo com a chegada de novas percepções e esta estrutura é utilizada nos procedimentos de tomada de decisão, os quais irão gerar ações (GIRARDI; FERREIRA, 2002).

A arquitetura interna de um único agente pode incorporar muitas propriedades. A Tabela 2.1 apresenta, segundo o OMG (2000a), algumas delas e uma breve descrição de cada uma.

---

[1] Tradução do termo em inglês *effectors*, também traduzido como efetores e efetuadores.

Tabela 2.1 – Propriedades do agente.

Propriedade	**Descrição**
Autonomia	Capacidade de atuar sem intervenção externa direta.
Interação	Capacidade de interagir com o ambiente e outros agentes.
Adaptação	Capacidade de responder em algum grau a outros agentes e ao seu ambiente. Formas mais avançadas de adaptação permitem ao agente modificar seu comportamento baseado nas suas experiências anteriores.
Mobilidade	Capacidade de se transportar de um ambiente para outro.
Proatividade	Capacidade de agir devido a sua orientação a objetivos. O agente não age simplesmente em resposta ao ambiente.
Continuidade temporária	É um processo continuamente em execução.
Caráter	Capacidade de possuir personalidade e estado emocional.
Reatividade	Capacidade de perceber mudanças no ambiente e atuar de acordo com essas mudanças.
Coordenação	Capacidade de desempenhar atividades num ambiente compartilhado com outros agentes. Estas atividades são coordenadas através de planos, fluxos de trabalho ou mecanismos de gerência de processos.
Colaboração\Cooperação	Capacidade dos agentes de se coordenarem para alcançar um objetivo comum.
Aprendizado	Capacidade do agente de aprender baseado em experiências anteriores, quando interagindo com seu ambiente.

Algumas propriedades como a interação e colaboração são vitais em agentes que compõem um SMA. Jennings et al. (1998) consideram o SMA como uma composição de entidades autônomas que interagem entre si para solucionar problemas que estão além de suas capacidades e conhecimentos individuais. Um agente que não é suficientemente capacitado para realizar uma tarefa específica pode solicitar a realização desta tarefa para outro agente mais capacitado.

Um SMA é caracterizado pela composição de um conjunto de diferentes **tipos de agentes** e objetos que são inseridos em **ambientes** (JENNINGS, 1999). As principais características dos SMAs são:

- Cada agente tem capacidade de solucionar parcialmente o problema;
- Não existe um controle global do sistema;
- Os dados são descentralizados;
- O processamento é assíncrono.

Segundo o OMG (2000a) algumas considerações sobre os SMAs são importantes:

Um agente não pode ser onipotente, ou seja, construído de forma que faça tudo sozinho, pois dessa forma ele será pesado e poderá apresentar problemas de velocidade, confiabilidade, manutenção etc. Pela atribuição de papéis aos agentes é obtido maior modularidade, flexibilidade, manutenibilidade e extensibilidade.

O agente não pode ser onisciente, o conhecimento está espalhado em vários agentes, podendo ser integrado a fim de obter se uma visão geral, quando necessário.

Aplicações que requerem computação distribuída são melhor suportadas por SMAs. Agentes podem ser projetados como pequenos componentes autônomos que trabalham em paralelo. O processamento e solução de problemas de forma concorrente trazem soluções para muitos problemas que até então eram tratados de uma maneira mais linear.

Em geral, um SMA utiliza mais de um tipo de agente. Um sistema que utiliza agentes de mais de um tipo é conhecido como sistema de agentes heterogêneos.

O tipo do agente é definido por sua classificação que pode ser realizada de diferentes maneiras, devido as propriedades e características que o diferencia dos outros (FRANKLIN; GRAESSER, 1996). Por exemplo, um agente é do tipo móvel quando possui a habilidade de transportar-se de um ambiente para outro, no entanto, um agente sem esta habilidade é classificado como estático ou estacionário. Também é possível classificar um agente de acordo com o domínio de serviços que ele oferece ao ambiente, por exemplo, agente de informação – responsável por obter informações – ou agente de segurança – responsável pela segurança do sistema – (KNAPIK; JOHNSON, 1997).

Para Ferber (1999), a capacidade cognitiva dos agentes é uma característica que também pode ser utilizada para sua classificação. Este autor apresenta três classes de agentes: cognitivos, reativos e híbridos.

O agente cognitivo reage no ambiente usando o conhecimento com base no histórico do agente e incorpora estratégias de adaptação ou técnicas de aprendizagem. Possui a capacidade de raciocínio baseada na representação interna de seu mundo. Esta representação é realizada através de componentes cognitivos conhecidos como estados mentais. Os estados mentais são responsáveis pela gestão de todas as atividades internas de um agente como a percepção e execução de ações, crenças, desejos, intenções, métodos e planos.

Já o agente reativo simplesmente reage no ambiente sem qualquer conhecimento prévio de sua história. Por último, o agente híbrido é um agente cognitivo e reativo.

Sob o ponto de vista de Nwana (1996), os agentes devem ser classificados de acordo com três dimensões: (I) mobilidade – que diferencia agentes móveis de estáticos –, (II) o paradigma lógico empregado – que classifica o agente entre cognitivo[2] ou reativo – e (III) a característica fundamental: autonomia, cooperação e aprendizado.

A combinação de duas ou mais destas características classifica o agente como híbrido, por exemplo, agente reativo colaborativo, agente estático reativo colaborativo etc.

As classificações de tipos de agente não são excludentes. De maneira que, um agente cognitivo pode ser também de informação ou um agente de segurança pode ser reativo ou cognitivo (GARCIA, 2004).

Os agentes, independentemente do tipo, estão inseridos em um ou mais ambientes. Ambiente é tudo que possa interagir com um agente, e que não faça parte da sua estrutura interna.

Segundo Russel e Norvig (2003), os ambientes de agentes podem ser classificados de acordo com as características: acessível ou inacessível; determinístico ou não determinístico; estático ou dinâmico e por último discreto ou contínuo.

O ambiente acessível é aquele em que o agente pode obter uma informação atual, completa e precisa sobre o estado do ambiente.

No ambiente determinístico não há incerteza sobre o resultado proveniente das ações do agente, ou seja, uma ação tem somente um resultado possível.

O ambiente estático é aquele que só altera seu estado mediante as ações dos agentes e nunca por qualquer outro fator externo.

Ambiente discreto é aquele em que existe um número fixo e finito de ações e percepções sobre ele.

Segundo Wooldrige (2002), os ambientes mais complexos são os inacessíveis, não determinísticos, dinâmicos e contínuos. As características do ambiente são importantes para determinar a complexidade no processo de desenvolvimento dos agentes. No entanto, não é o único fator que deve ser levado em consideração, a interação dos agentes com o ambiente também é importante.

Para a compreensão da arquitetura apresentada por este trabalho faz-se necessário além do entendimento do conceito de agente, SMA, tipos de agentes e ambiente, a discussão das propriedades de autonomia, interação e papel.

---

[2] Também denominado como deliberativo ou racional.

2.1. Autonomia

A autonomia e a interação são consideradas pela *Foundation for Intelligent Physical Agents* (FIPA) (2002) e o OMG (2000a) como propriedades requeridas para os agentes. A autonomia é caracterizada pela ausência de controle externo sobre um agente, ou seja, ele tem controle sobre suas ações e pode agir de forma independente de outros para alcançar seus objetivos.

De acordo com Garcia (2004), esta propriedade é essencial para distinguir agentes de objetos. Pois, os objetos são controlados pela parte externa, em oposição aos agentes que possuem um comportamento autônomo que não pode ser controlado externamente. Desse modo, os agentes podem decidir negar uma solicitação de serviço e ainda agir conforme desejar. Este autor apresenta a autonomia dos agentes em três dimensões:

(I) autonomia de execução;
(II) autonomia de decisão;
(III) autonomia proativa.

Para alcançar seus objetivos, os agentes têm suas próprias *threads* de controle (I), tomam decisões em relação a instanciações de objetivos (II) e realizam ações sem uma intervenção externa direta (III) (GARCIA, 2004).

A autonomia de execução é o controle das *threads* de agente. Existe diferentes estratégias para a instanciação de *threads* de agente. Por exemplo, uma única *thread* é instanciada quando o agente é criado, uma *thread* é instanciada para cada novo objetivo do agente, uma *thread* é criada para cada mensagem externa recebida etc.

A autonomia de decisão consiste em tomar decisões sobre a realização de objetivos, esta capacidade acaba deliberando num determinado momento o curso de ação que um agente toma. A decisão é implementada por ações simples ou planos, ambos associados aos objetivos. Assim que um evento é percebido por um agente estas ações ou planos são realizados e então é decidido se a realização de um objetivo deve ser iniciada ou não.

A autonomia proativa é caracterizada pela proatividade, definida como a execução de ações ou planos sem uma solicitação explícita de outros agentes, ou seja, o agente não age simplesmente em resposta ao ambiente, mas sim devido a sua orientação a objetivos.

2.2. Interação

Um agente não pode evocar ou simplesmente forçar outro agente a realizar alguma ação, porém através de atos de comunicação[3] pode interagir com ele e o influenciar a realizar esta ação (WOOLDRIDGE, 2002).

Segundo Symeonidis e Mitka (2006), a interação é a propriedade que permite ao agente interagir com outros agentes e seu ambiente. A interatividade confere o aspecto comportamental que o agente exibe em relação a seu ambiente, este comportamento pode ser caracterizado como de reatividade ou proatividade.

A reatividade pode ser definida como a habilidade de perceber mudanças no ambiente e responder a elas quando necessário. Já a proatividade é entendida como a habilidade de agir por iniciativa própria para satisfazer algum objetivo do agente.

De acordo com Garcia (2004), a interação consiste em receber e enviar mensagens para outros agentes. Através dos sensores e atuadores os agentes detectam a chegada de mensagens de outros agentes e enviam mensagens ou geram eventos no ambiente.

Para Huhns e Singh (1997), esta forma de comunicação entre agentes pode ser vista sob uma perspectiva de três níveis: (I) sintaxe – como as mensagens são estruturadas –; (II) semântica – o conteúdo das mensagens – e (III) *pragmatics* – como a mensagem pode ser interpretada –. O significado da mensagem está na combinação da semântica com o *pragmatics* (HUHNS; STEPHENS, 1999).

Geralmente, as mensagens são estruturadas conforme alguma das linguagens de comunicação de agentes (*Agent Communication Language*, ACL) existentes e são exemplos destas linguagens a *Knowledge and Querying Manipulation Language* (KQML) (FININ et al., 1994), *Knowledge Interchange Format* (KIF) (GENESERETH, 1991) e por último a FIPA ACL (FIPA, 2002).

---

[3] Tradução do termo em inglês *communicative acts*, também conhecido como *speech acts* e *performatives*.

As ACLs citadas no parágrafo anterior estão fundamentadas na teoria conhecida como *speech acts* (SEARLE, 1969), de modo que cada mensagem representa um ato de comunicação (BELLIFEMINE; CAIRE; GREENWOOD, 2007).

Segundo Wooldridge (2002), a FIPA ACL possui sintaxe similar a KQML, pois a estrutura da mensagem é quase a mesma e os parâmetros das mensagens são muito similares. A diferença relevante entre estas duas linguagens está na coleção de atos de comunicação que elas oferecem.

Entre todos os atos de comunicação oferecidos pela FIPA ACL, são destacados por Wooldridge (2002) como sendo de grande importância o *inform* e o *request*.

O ato de comunicação *inform* é utilizado para comunicar uma informação, é usado sempre que o agente emissor da mensagem precisa informar ao receptor sobre uma proposição que considera verdadeira. Já o *request* permite ao agente emissor solicitar ao receptor que execute uma ação.

Além do ato de comunicação, uma mensagem FIPA ACL é composta também por um ou mais parâmetros.

A quantidade de parâmetros presentes numa mensagem pode variar de acordo com a comunicação em questão. O parâmetro *performative* – que denota o ato de comunicação –, é requerido enquanto os parâmetros *sender* – que define o agente emissor da mensagem –, *receiver* – que define o agente receptor da mensagem – e *content* – conteúdo da mensagem –, apesar de não serem obrigatórios, estão na maioria das mensagens.

Um exemplo de mensagem FIPA ACL é apresentado no Quadro 2.1. Neste exemplo é possível verificar o ato de comunicação do tipo *request* na linha 01; os valores dos parâmetros *sender* e *receiver*, ou melhor, o agente emissor e o receptor da mensagem nas linhas 02 e 03; e por último, o conteúdo da mensagem na linha 07 em diante.

Quadro 2.1 – Exemplo de mensagem FIPA ACL com o ato de comunicação *request*.

```
(request
  :sender (agent-identifier :name alice@mydomain.com)
  :receiver (agent-identifier :name bob@yourdomain.com)
  :ontology travel-assistant
  :language FIPA-SL
  :protocol fipa-request
  :content
  ""((action
        (agent-identifier :name bob@yourdomain.com)
        (book-hotel :arrival 15/10/2006
        :departure 05/07/2002 ... )
  ))""
)
```

Fonte: (BELLIFEMINE; CAIRE; GREENWOOD, 2007)

2.3. Papéis

Os papéis de agentes[4] têm sido utilizados durante as fases de concepção (SILVA et al., 2003), análise (ZAMBONELLI; JENNINGS; WOOLDRIDGE, 2000), projeto e implementação (KENDALL, 1999) de SMAs.

Segundo Partsakoulakis e Vouros (2004), a autonomia dos agentes faz com que eles possam decidir seu próprio comportamento e em consequência desta natureza autônoma, existe a necessidade de coordenação dos agentes ou então, o grupo pode apresentar um comportamento caótico. Uma maneira efetiva de alcançar esta coordenação é com a imposição de uma organização – composta por papéis e suas inter-relações – específica para o grupo.

Agentes podem fazer parte deste grupo se forem capazes de desempenhar um ou mais papéis que contribuam para o alcance dos objetivos coletivos (PARTSAKOULAKIS; VOUROS, 2004).

De acordo com Garcia (2004), o papel é inerente a propriedade de colaboração, pois o agente exerce diferentes papéis num SMA ao trabalhar junto com outros agentes na tentativa de alcançar os objetivos.

Um papel agrupa diferentes tipos de comportamentos em uma unidade funcional que contribui para um objetivo do grupo, ou seja, cada papel possui o conhecimento e os

[4] Tradução do termo em inglês *agent roles*.

comportamentos necessários para realizar as colaborações com outros agentes (GARCIA, 2004).

As interações dos agentes, ou seja, suas ações e comportamentos podem ser modelados segundo o conceito de papel, desta maneira, determinadas ações ou comportamentos só serão apresentados por agentes que desempenham papéis específicos (CABRI; LEONARDI; ZAMBONELLI, 2002).

Em geral, os papéis podem ser classificados em dois tipos: (I) papel intrínseco – que define o conhecimento intrínseco e as funcionalidades básicas do agente –, e (II) papel extrínseco ou colaborativo – que diz respeito ao exercício das capacidades extrínsecas do agente em seus relacionamentos colaborativos –.

A definição de um tipo de agente está relacionada ao papel intrínseco exercido por ele no sistema, enquanto a definição de um papel captura o modo como um agente interage com outros agentes em seus relacionamentos de colaboração.

Segundo Garcia (2004), a relação entre papel e agente é regida por oito princípios: (I) dependência – um papel não pode existir sem um tipo de agente associado a ele –; (II) dinamicidade – um papel pode ser adicionado ou removido durante o ciclo de vida de um agente –; (III) identidade – o papel e o agente possuem a mesma identidade –, ou seja, o tipo de agente e seus papéis são vistos e podem ser manipulados como uma entidade; (IV) herança – o papel de um tipo de agente também é o papel para qualquer subtipo de agente –; (V) multiplicidade – o agente pode exercer vários papéis ao mesmo tempo –; (VI) visibilidade – o acesso ao agente é restringido pelo papel –; (VII) abstratividade – os papéis podem ser organizados em hierarquias – e (VIII) agregação – os papéis podem ser compostos por outros papéis –.

Neste capítulo foram apresentados os conceitos de agente, SMA, tipo de agente e ambientes. As propriedades de autonomia, interação e papéis e a estrutura da mensagem FIPA ACL foram abordadas em detalhes. No próximo capítulo serão apresentados dois conceitos fundamentais da segurança de sistemas de informação: a autenticação e o controle de acesso.

3. Autenticação e Controle de Acesso

Segundo Sandhu e Samarati (1996), a autenticação, o controle de acesso e a auditoria constituem a base para a segurança de sistemas de informação. Este capítulo apresenta dois conceitos fundamentais encontrados no contexto da segurança de sistemas de informação: a autenticação e o controle de acesso. Devido ao conceito de controle de acesso e o modelo NIST RBAC serem de grande importância para este trabalho, ambos são apresentados em detalhes no decorrer deste capítulo.

A segurança da informação visa proteger as informações de ameaças, com o objetivo de preservar o valor que possuem para um indivíduo ou uma organização, de forma a assegurar a continuidade de um negócio, minimizar danos e maximizar o retorno dos investimentos e das oportunidades (CAMILO, 2001).

Através da implementação de um conjunto de controles tais como: políticas, práticas, procedimentos, estruturas organizacionais e funções de programas, três propriedades essenciais de segurança da informação podem ser preservadas: confidencialidade, integridade e a disponibilidade (ITSEC,1991).

A confidencialidade é a garantia de que a informação seja acessível somente para àqueles que têm autorização para acessá-la.

A integridade é a preservação da informação conforme definida por aqueles que têm a autorização para manipulá-las.

A disponibilidade é a garantia de que a informação esteja sempre disponível para os usuários que possuem a autorização para acessá-la.

Estas três propriedades da segurança da informação também se aplicam à segurança de sistemas de informação, onde a confidencialidade pode ser alcançada através da implementação do controle de acesso (MCLEAN, 1994).

De acordo com Lehtinen et al. (2006), o controle de acesso em sistemas de informação é composto pelo controle de acesso ao sistema e o controle de acesso aos dados.

Controle de acesso ao sistema é a primeira forma pela qual um sistema proporciona segurança, pois é o controle de acesso a si mesmo. Este processo estabelece a identidade de uma entidade – usuário, máquina, agente de software – à outra; um exemplo comum é um usuário que através de seu nome e senha tenta identificar-se num sistema para obter acesso aos seus recursos (GONG; ELLISON; DAGEFORDE, 2003).

O controle de acesso aos dados está ligado à capacidade do sistema de limitar o acesso do usuário a determinados recursos que este oferece.

Na terminologia de segurança da informação, o controle de acesso ao sistema é conhecido como identificação e autenticação enquanto o controle de acesso aos dados é simplesmente conhecido como controle de acesso ou autorização.

Autenticação e controle de acesso são propriedades de segurança que mantém um relacionamento estreito, sendo que é comum ao implementar a propriedade de controle de acesso implementar também a propriedade de autenticação.

A função do controle de acesso é garantir que apenas usuários autorizados possam manusear informações, acessar recursos ou funcionalidades dos sistemas. Para que isto seja possível é necessário identificar e autenticar o usuário que solicita o acesso (VILLAR, 2007).

A autenticação é com frequência um pré-requisito para o controle de acesso, pois através dela uma entidade não poderá se passar por outra, tendo assim, acesso a recursos não autorizados a ela.

No processo de autenticação, a entidade fornece uma prova para constatar que realmente é quem diz ser. O'Gorman (2003) agrupa os fatores da autenticação em três categorias:

- Baseados no que você sabe;
- Baseados no que você possui;
- Baseados no que você é.

O primeiro fator é baseado no conhecimento de um segredo ou algo obscuro, exemplos disto são a senha ou a cor preferida de um usuário.

O segundo fator é baseado na posse física de um objeto, um *token* ou qualquer dispositivo que de alguma maneira armazene senhas, chaves ou certificados de identidade são exemplos frequentes.

O último fator é baseado na unicidade de uma pessoa, sistemas biométricos como o reconhecimento da impressão digital, da voz e ainda a identificação pela íris ou retina, são exemplos deste tipo de autenticação.

Conforme descrito por Sandhu e Samarati (1994), o controle de acesso ou autorização, é responsável por limitar as atividades de usuários já autenticados pelo sistema. Este processo é ilustrado pela Figura 3.1 onde é possível observar que o elemento principal é o monitor de referência, que controla cada tentativa de acesso realizada por um usuário aos recursos de um sistema. Assim que o usuário tenta acessar um recurso do sistema, o monitor de referência consulta um banco de dados de autorizações para verificar se o usuário possui as devidas permissões de acesso ao recurso desejado.

As permissões do banco de dados de autorizações são mantidas por um administrador de segurança que as define de acordo com a política de segurança da organização.

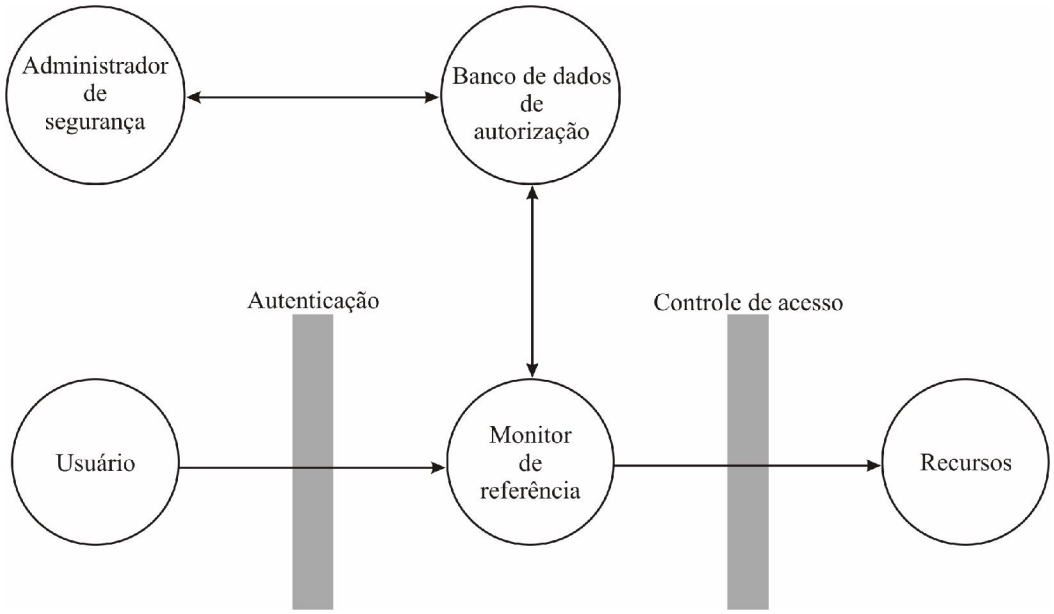

Figura 3.1 – Autenticação, controle de acesso e outros elementos de segurança.
Fonte: Adaptado de SANDHU; SAMARATI (1994)

O controle de acesso é de grande importância para este trabalho, por isto é apresentado em maiores detalhes a seguir.

3.1. Modelos de Controle de Acesso

Os mecanismos de controle de acesso são utilizados para implementar a política de autorização de um sistema e estão ligados aos modelos de controle de acesso (WANGHAM, 2004).

Sandhu e Samarati (1996) classificam os modelos de controle de acesso em três categorias: discricionário, obrigatório e baseado em papéis.

O controle de acesso discricionário[5] é baseado na ideia de que o dono da informação determina quem tem acesso a ela. Ele permite que os dados sejam livremente copiados de objeto para objeto, mesmo que o acesso aos dados originais seja negado, um sujeito sempre pode obter acesso a uma cópia.

O controle de acesso obrigatório[6] é baseado nos conceitos de confidencialidade utilizados na área militar como um meio de gerenciar informações previamente classificadas. Utiliza um controle centralizado de segurança onde usuários e objetos, ou melhor, recursos do sistema, são previamente classificados de modo que um usuário só tenha acesso a informações a qual possui habilitação (YAO, 2003).

Por último, no controle de acesso baseado em papéis, os direitos de acesso são atribuídos aos papéis ao invés de serem atribuídos a cada usuário, como no controle de acesso discricionário, os usuários obtêm estes direitos em virtude de terem papéis atribuídos a si.

A atuação do papel como intermediário que possibilita ao usuário exercitar uma permissão, é a responsável pela grande flexibilidade e granularidade das atribuições de permissão, resultado que não é alcançado na atribuição direta de permissões ao usuário, sem o uso de papéis (CONCEIÇÃO; SILVA, 2006).

3.1.1. Modelo NIST RBAC

Sandhu et al. (2000) relatam que muitos vendedores de sistemas de computação têm implementado variações do modelo RBAC em seus produtos como banco de dados, sistemas administrativos e operacionais devido a facilidade e flexibilidade deste modelo.

Em 2000, Sandhu et al. em consideração a uma requisição do NIST desenvolveram uma proposta de padrão para o modelo RBAC, com o objetivo de auxiliar fabricantes de sistemas que desejavam adotar um modelo RBAC padrão em seus produtos.

---

[5] Tradução do termo em inglês *Discretionary Access Control (DAC).*

[6] Tradução do termo em inglês *Mandatory Access Control (MAC).*

Este modelo foi publicado como NIST RBAC *model* (SANDHU; FERRAIOLO; KUHN, 2000) e depois adotado como um padrão 359-2004 RBAC pelo o *InterNational Committee for Information Technology* (INCITS) em 2004.

O NIST RBAC continua sendo aprimorado através do INCITS, que em 2009 solicitou revisões para o modelo INCITS 359-2004 RBAC.

Este modelo é inspirado no modelo RBAC96 proposto anteriormente por Sandhu (1997), porém conforme descrito por Sandhu et al. (2000), o modelo NIST RBAC é organizado em quatro níveis sequenciais, apresentados pela Figura 3.2, onde funcionalidades são acrescentadas de maneira gradual. Cada nível adiciona exatamente um requisito e os níveis são acumulativos, de forma que cada nível contém os requisitos do nível anterior da sequência.

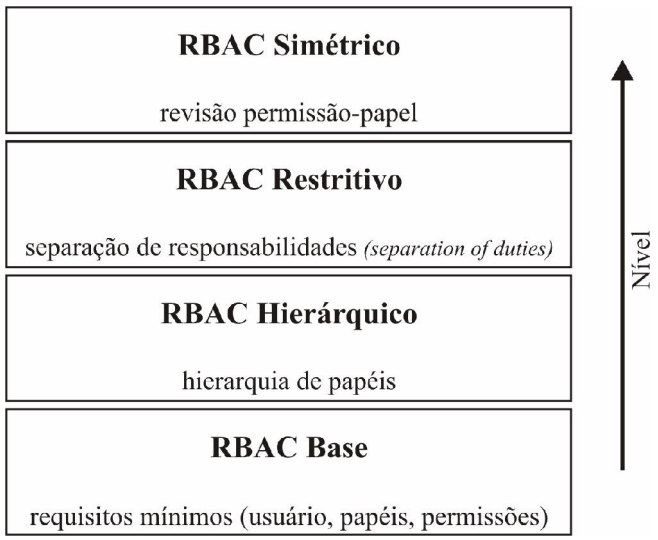

Figura 3.2 – Níveis do modelo NIST RBAC.

Alguns níveis deste modelo apresentam subníveis e em cada um deles é implementado uma funcionalidade específica do nível em questão. A Tabela 3.1 mostra os níveis, subníveis e suas principais funcionalidades.

Tabela 3.1 – Variações do RBAC organizadas pelo nível e suas principais funcionalidades.

Nível	Nome	Funcionalidades
1	RBAC Base	• usuário adquire permissões através de papéis • deve suportar uma relação muitos para muitos de usuários para papéis (UA) • deve suportar uma relação muitos para muitos de permissões para papéis (PA) • deve suportar revisão da relação usuário para papel (UA) • usuários podem utilizar as permissões de múltiplos papéis simultaneamente
2	RBAC Hierárquico	• funcionalidades do nível anterior • deve suportar hierarquia de papel (ordem parcial) • **nível 2a**, requer suporte a hierarquias arbitrárias • **nível 2b**, denota suporte a hierarquias limitadas
3	RBAC Restritivo	• funcionalidades dos níveis anteriores • deve suportar a separação de responsabilidades • **nível 3a**, requer suporte a hierarquias arbitrárias • **nível 3b**, denota suporte a hierarquias limitadas
4	RBAC Simétrico	• funcionalidades do nível anterior • deve suportar revisão da relação permissão para papel (PA) com performance comparável a revisão da relação usuário para papel (UA) • **nível 4a**, requer suporte a hierarquias arbitrárias • **nível 4b**, denota suporte a hierarquias limitadas

Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

O primeiro nível deste modelo é o RBAC Base[7], responsável pela implementação das entidades principais do modelo RBAC: usuários (U), papéis (R) e permissões (P). Sandhu et al. (2000) descrevem as estas três entidades da seguinte maneira:

Um usuário neste modelo é a representação de um ser humano, agente autônomo, processo ou máquina.

Papel é um trabalho funcional ou o título de um trabalho e confere determinada autoridade e responsabilidade ao usuário que desempenha o papel.

Permissão é a aprovação de um modo particular de acesso a um ou mais objetos do sistema. Nesse modelo, objetos representam dados ou qualquer outro recurso que o sistema possa oferecer. Elas são sempre positivas, conferindo poder ao seu portador de executar alguma ação no sistema.

[7] Tradução do termo em inglês *Flat RBAC.*

Duas relações fazem a ligação dessas três entidades: a atribuição de usuários[8] (UA) e a atribuição de permissões[9] (PA). Estas relações são do tipo muitos para muitos, ou seja, um usuário pode desempenhar vários papéis e um papel pode ser desempenhado por vários usuários. De forma similar, um papel pode ter várias permissões e uma permissão pode ser atribuída a vários papéis. A Figura 3.3 ilustra graficamente como estão relacionadas as entidades usuários, papéis e permissões neste nível. Usuários estão ligados aos papéis através da relação atribuição de usuários e as permissões estão ligadas aos papéis através da relação atribuição de permissões.

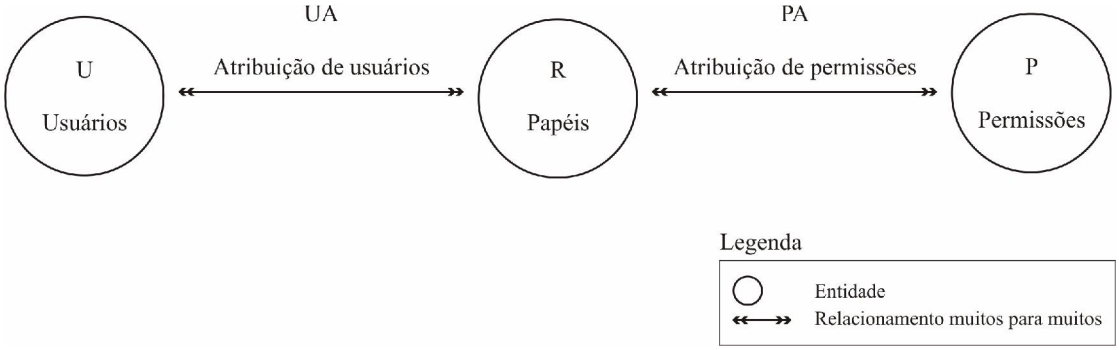

Figura 3.3 – RBAC Base.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

No primeiro nível é exigido que a revisão da **relação usuário para papel** seja implementada; através desta funcionalidade os papéis atribuídos a um usuário podem ser determinados, assim como, quais usuários possuem um determinado papel.

Por exigência deste modelo, um usuário deve exercitar, simultaneamente, as permissões de todos papéis que detém, isto impossibilita produtos que restrinjam usuários a ativação de um único papel por vez.

Sandhu (1997) formaliza os componentes deste nível por meio da definição:

- U, R, P, (usuários, papéis e permissões respectivamente);
- $UA \subseteq U \times R$, relação muitos para muitos de usuário para papel;
- $PA \subseteq P \times R$, relação muitos para muitos de permissão para papel.

[8] Tradução do termo em inglês *user assignment*.

[9] Tradução do termo em inglês *permission assignment*.

O conceito de sessões presente de forma facultativa neste modelo, ao contrário do modelo RBAC96 (SANDHU, 1997) onde é requerido, é considerado como o mapeamento de um usuário para um ou mais papéis. O usuário estabelece uma sessão ao ativar um subconjunto de papéis os quais ele detém, criando assim uma relação um para muitos. A Figura 3.4 ilustra o RBAC Base implementado com o suporte a sessões onde é possível notar que por meio da sessão o usuário é associado a um ou mais papéis.

Com a utilização da sessão é possível que um usuário, detentor de um papel que possui permissões mais amplas, possa desativar temporariamente este papel e utilizá-lo somente quando as permissões que ele oferece forem necessárias.

Assim, um usuário detentor de vários papéis, pode trabalhar apenas com um subconjunto deles adequado à execução das atividades a serem realizadas na sessão, e este princípio é conhecido como privilégio mínimo[10].

Segundo Conceição e Silva (2006), a atuação do papel como intermediário que possibilita o usuário exercitar uma permissão é a responsável pela grande flexibilidade e granularidade das atribuições de permissão, resultado que não é alcançado na atribuição direta de permissões a usuário, sem o uso de papéis.

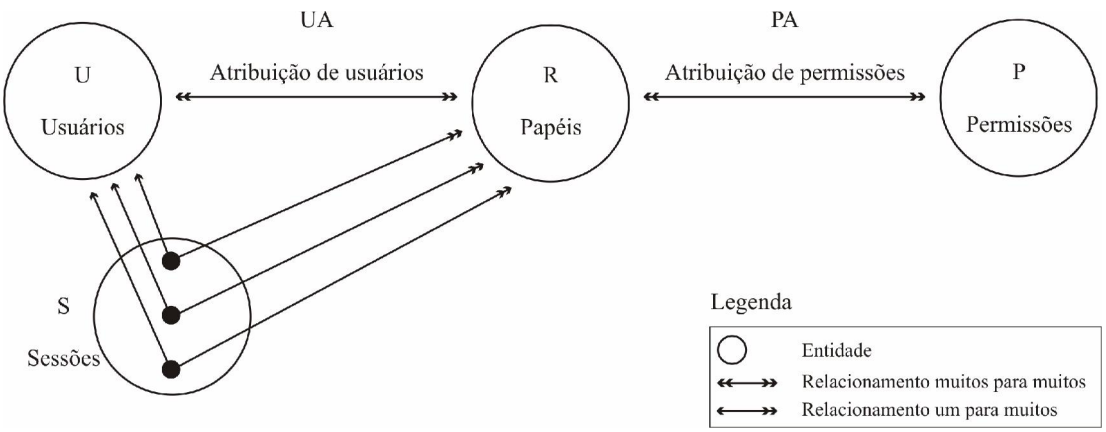

Figura 3.4 – RBAC Base com suporte a sessões.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

De acordo com Sandhu et al. (2000), o segundo nível, RBAC Hierárquico[11], difere do nível anterior pelo fato de implementar os requisitos necessários para introdução da relação de hierarquia de papel (RH). Este nível representado graficamente pela Figura 3.5, a única

[10] Tradução do termo em inglês *least privilege*.

[11] Tradução do termo em inglês *Hierarchical RBAC*.

diferença com relação ao nível anterior é a adição da relação de hierarquia de papel (RH) na entidade Papéis (R).

Sandhu (1997) formaliza os componentes deste nível através da definição:

- *U, R, P, UA, PA* (permanecem inalterados);
- $RH \subseteq R \times R$, é um ordenamento parcial em *R* chamado de hierarquia de papéis, ou de relação de domínio entre papéis.

Organizações normalmente possuem uma estrutura hierárquica de comando. A hierarquia de papéis estrutura naturalmente a autoridade e responsabilidade dentro de uma linha de uma organização (SANDHU, 1997).

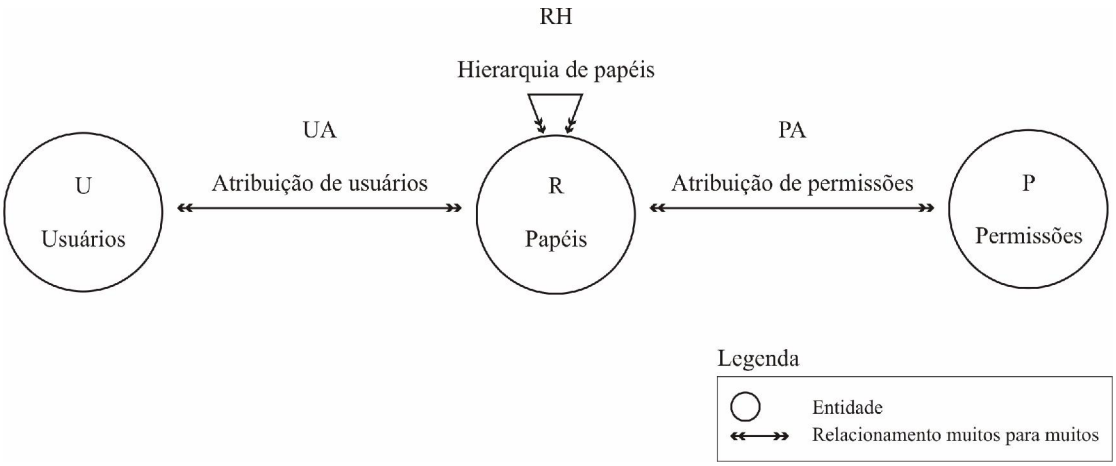

Figura 3.5 – RBAC Hierárquico.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

Por convenção, a posição mais alta na hierarquia, ou seja, papéis com o maior número de permissões são posicionados na parte superior enquanto os de menor número são posicionados na parte inferior.

Através da Figura 3.6(a) é possível observar uma hierarquia de papéis. O papel de menor poder na hierarquia é o membro do projeto que está situado na parte inferior do desenho, logo acima está o analista, papel que herda as permissões do papel membro do projeto.

O papel analista pode acrescentar novas permissões além das herdadas pelo papel membro do projeto. A herança de permissões é cumulativa, ou seja, o papel analista programador herda as permissões dos papéis analista e membro do projeto, o mesmo é válido para o papel analista de teste.

Um papel pode também herdar as permissões de dois ou mais papéis. A Figura 3.6(b) exibe um exemplo de herança múltipla onde o papel supervisor do projeto herda as permissões dos papéis analista programador e analista de teste.

Dois subníveis de hierarquias são reconhecidos neste modelo: arbitrária e limitada.

Com a hierarquia arbitrária é possível a adoção de um ordenamento parcial arbitrário para servir como a hierarquia de papéis.

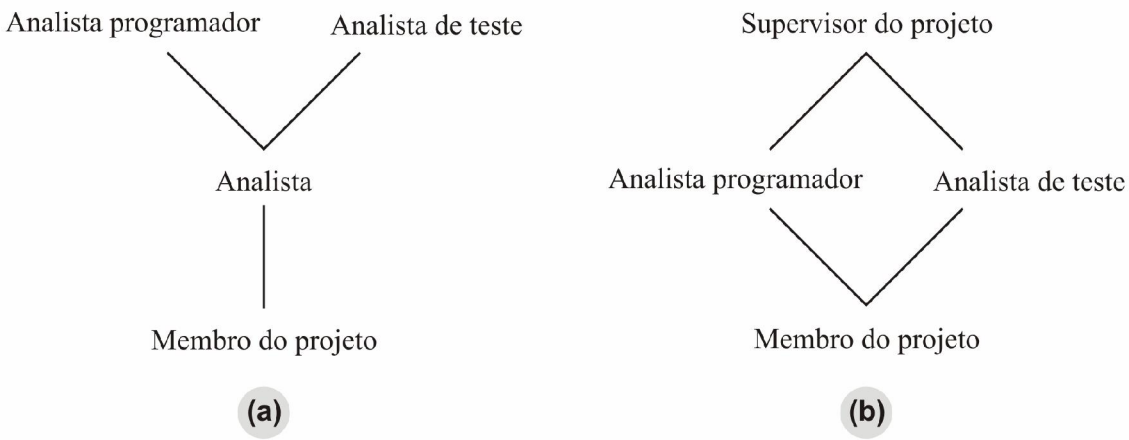

Figura 3.6 – Exemplos de hierarquias de papéis.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

Através da hierarquia limitada é possível a imposição de limitações na hierarquia de papéis, normalmente as hierarquias são limitadas a estruturas simples como árvores ou árvores invertidas.

O terceiro nível, RBAC Restritivo[12], implementa por meio de restrições[13] o conceito de separação de responsabilidades[14].

---

[12] Tradução do termo em inglês *Constrained RBAC*.

[13] Tradução do termo em inglês *constraints*.

[14] Tradução do termo em inglês *separation of duty*, também traduzido como separação de tarefas e separação de deveres.

Sandhu (1997) considera as restrições um aspecto importante do modelo RBAC, muitas vezes apontado como a principal motivação deste modelo.

Segundo Camilo (2001), a separação de responsabilidades é uma técnica conhecida muito antes da existência dos computadores, que possibilita a redução da possibilidade da ocorrência de fraudes ou danos acidentais.

A separação de responsabilidades é realizada através da divisão de responsabilidade e autoridade sobre uma ação, ou tarefa entre múltiplos usuários, diminuindo assim o risco decorrente de uma ação fraudulenta, pois a execução de uma tarefa requer o envolvimento de mais de um indivíduo.

Um exemplo prático da aplicação dessa ideia é a prática comum de algumas organizações de não permitir ao mesmo indivíduo desempenhar o papel de requisitante de pagamento e autorizador de pagamento, esta medida evita que o mesmo indivíduo possa solicitar um pagamento e logo em seguida autorizá-lo: estes dois papéis são mutuamente exclusivos.

A separação de responsabilidades pode ser implementada através da aplicação das restrições nas permissões, por exemplo, as permissões para requisitar pagamento e autorizar pagamento não poderiam ser atribuídas ao mesmo papel, evitando assim que um papel seja perigosamente poderoso.

Existe, também, a possibilidade de utilizar as restrições para aplicar a separação de responsabilidades por cardinalidade. Neste modo, a restrição para atribuição de um papel está na quantidade de usuários que podem ser atribuídos a um papel, por exemplo, o papel de diretor comercial seria limitado a no máximo um usuário.

É possível realizar a separação de responsabilidades de duas formas: estática e dinâmica.

A separação estática de responsabilidades[15] é realizada pela aplicação das restrições diretamente na atribuição do usuário aos papéis (UA), conforme ilustrado pela Figura 3.7; desta forma, um usuário nunca tem papéis mutuamente exclusivos e esta relação é considerada estática, pois não é alterada ao longo do tempo.

[15] Tradução do termo em inglês *static separation of duty*.

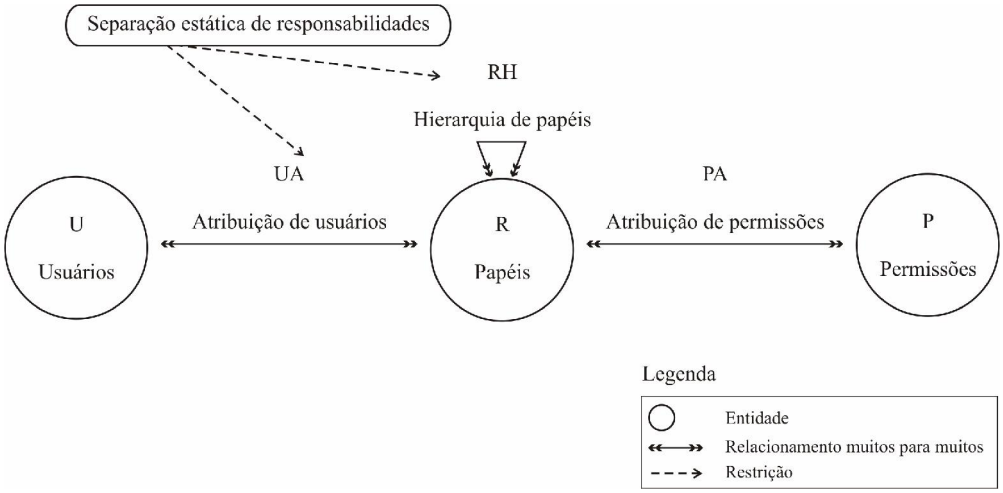

Figura 3.7 – Separação estática de responsabilidades.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

A separação dinâmica de responsabilidades[16] apresenta uma pequena diferença em relação a técnica anterior: a aplicação das restrições é realizada somente nos papéis que estão ativos numa sessão, conforme exibido pela Figura 3.8, desta forma, uma sessão nunca tem papéis mutuamente exclusivos ao mesmo tempo, porém devido ao mecanismo de sessão, usuários podem ter diferentes níveis de permissão em diferentes momentos.

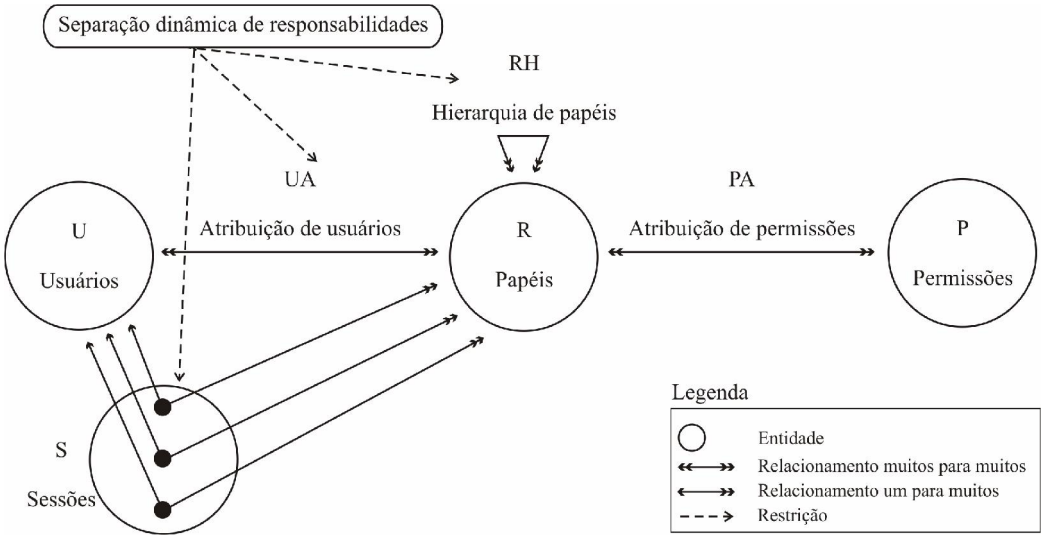

Figura 3.8 – Separação dinâmica de responsabilidades.
Fonte: Adaptado de SANDHU; FERRAIOLO; KUHN (2000)

[16] Tradução do termo em inglês *dynamic separation of duty*.

O último nível deste modelo, RBAC Simétrico[17], incorpora todas as características já adicionadas pelos níveis anteriores e acrescenta a revisão da relação de permissão para papel, similar a revisão da relação de usuário para papel apresentada no primeiro nível.

Com esta funcionalidade é possível determinar quais permissões foram atribuídas a um papel específico, assim como, a quais papéis uma determinada permissão foi atribuída.

Este capítulo teve por objetivo esclarecer os dois conceitos fundamentais encontrados no contexto da segurança de sistemas de informação: a autenticação e o controle de acesso, além de apresentar em detalhes o modelo de controle de acesso NIST RBAC. No próximo capítulo, a questão da separação de interesses será abordada e os conceitos de modularização e orientação a aspectos serão apresentados.

[17] Tradução do termo em inglês *Symmetric RBAC*.

4. Separação de Interesses com a Programação Orientada a Aspectos

Por muitos anos, teóricos da computação concordaram que a melhor maneira de criar sistemas mais gerenciáveis é identificando e separando os interesses do sistema. Esta separação é um importante instrumento para se reduzir a complexidade de sistemas de software (FIGUEIREDO, 2006). A ideia da separação de interesses[18] é antiga, alguns pesquisadores afirmam que este princípio foi formalizado por Dijkstra (1976). Este princípio é suportado parcialmente por paradigmas de desenvolvimento como a programação estruturada e a programação orientada a objetos (OO), os quais disponibilizam abstrações que, de alguma maneira, permitem a modularização dos interesses com variações de graus.

Atualmente existem algumas abordagens para a separação avançada de interesses, incluindo a programação orientada a sujeitos (HARRISON; OSSHER, 1993), filtros de composição (AKSIT et al., 1993), programação adaptativa (LIEBERHERR, 1996; LIEBERHERR; ORLEANS; OVLINGER, 2001) e separação multidimensional de interesses (TARR et al., 1999).

Segundo Lobato (2005), estas abordagens contribuem para o desenvolvimento de sistemas OO, uma vez que proporcionam a separação de interesses em outras dimensões, além de classes e objetos, ao introduzir novas abstrações de modularização e mecanismos de composição para refinar a separação de interesses transversais.

A orientação a aspectos (KICZALES et al., 1997) discutida neste capítulo, é uma das abordagens para separação avançada de interesses. É apontada como um paradigma complementar aos existentes, cujo objetivo é prover suporte à separação de interesses transversais em módulos separados, chamados de aspectos (FIGUEIREDO, 2006).

4.1. Separação de Interesses

A separação de interesses é o princípio que tenta resolver as limitações da cognição humana ao lidar com a complexidade do software (GARCIA, 2004).

---

[18] Tradução do termo em inglês *separation of concerns.*

Um interesse é uma consideração específica que deve ser atendida para que seja possível satisfazer o objetivo geral do sistema.

Um sistema é o resultado da realização de um conjunto de interesses (LADDAD, 2003).

De forma geral, os interesses podem ser classificados em duas categorias: interesses principais[19] e interesses transversais[20].

Interesse principal é o que se tenta separar dos demais, captura a essência, o relevante para o domínio da aplicação. O interesse principal provê a funcionalidade desejada sem depender de outros aspectos da computação como distribuição, persistência, segurança etc. Ele especifica o que é realmente importante para a aplicação.

Interesse transversal é qualquer forma de computação utilizada para controlar ou otimizar os interesses principais. Neste sentido, o interesse transversal provê suporte para os interesses principais desempenhando assim um papel secundário.

Uma aplicação frequentemente necessita desenvolver interesses transversais como autenticação, *logging*, persistência, transações etc. Todos estes interesses transversais citados, naturalmente se espalham pelos subsistemas ou módulos da aplicação. A Figura 4.1 ilustra esta situação, na qual é possível observar o desenvolvimento de alguns módulos do sistema que implementam tanto interesses referentes ao sistema – *system level* – como a segurança e a persistência, quanto interesses de negócio – *business concerns* – representado na figura pela lógica de negócios. A figura também exibe o sistema como uma composição de múltiplos interesses que ficam emaranhados. Devido à técnica utilizada na implementação dos módulos, a independência dos interesses não é mantida.

---

[19] Tradução do termo em inglês *core concerns*, também traduzido como interesse central e interesse básico.

[20] Tradução do termo em inglês *crosscutting concerns*, também traduzido como interesse periférico, interesse especial, interesse entrecortante e interesse ortogonal.

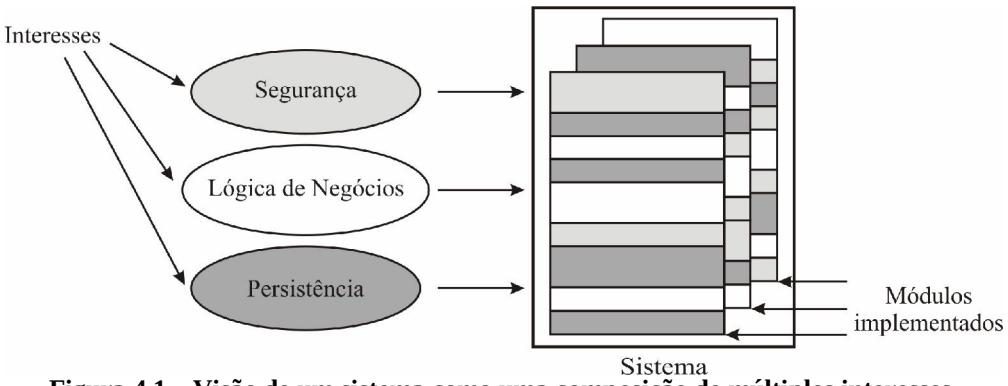

Figura 4.1 – Visão de um sistema como uma composição de múltiplos interesses.
Fonte: Adaptado de LADDAD (2003)

Esta situação não é favorável, pois para que um interesse seja compreendido e manuseado com mais facilidade, cada parte do programa dedicada a satisfazer a um determinado interesse deve estar concentrada num único local físico, separada de outros interesses (FIGUEIREDO, 2006).

Para que um desenvolvedor consiga manter o foco em cada interesse individual separadamente, reduzindo assim a complexidade do modelo e implementação de uma aplicação, é vital a identificação meticulosa dos interesses principais e transversais de um sistema. Para se obter este resultado o primeiro passo é decompor um conjunto de requisitos separando-os em interesses. Laddad (2003) apresenta duas técnicas para decompor requisitos num conjunto interesses, que são apresentadas a seguir.

A primeira técnica utiliza a analogia de um feixe de luz passando por um prisma para elucidar o processo da decomposição dos requisitos em interesses. A Figura 4.2 ilustra esta analogia; à primeira vista um requisito aparenta ser uma única unidade, entretanto, após ser aplicado um mecanismo de identificação de interesses é possível compreender que um requisito é composto por diversos interesses.

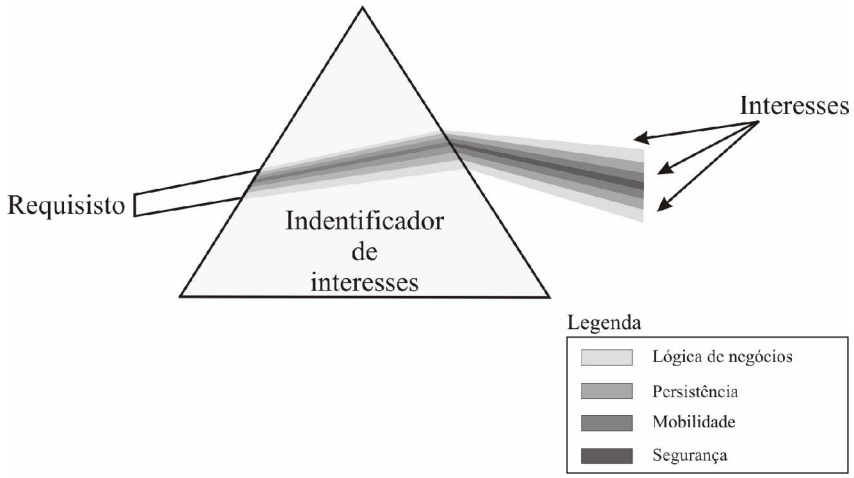

Figura 4.2 – Analogia com um prisma para a decomposição dos interesses.
Fonte: Adaptado de LADDAD (2003)

A segunda técnica projeta os interesses num espaço, com N dimensões, na qual cada interesse forma uma dimensão. A Figura 4.3 apresenta esta ideia ao exibir um espaço de interesses com três dimensões onde a lógica de negócios é o interesse principal e a persistência e o aprendizado são interesses transversais. Este tipo de visão de um sistema demonstra que os interesses apresentados neste espaço multidimensional são independentes de modo que podem evoluir sem afetar o restante.

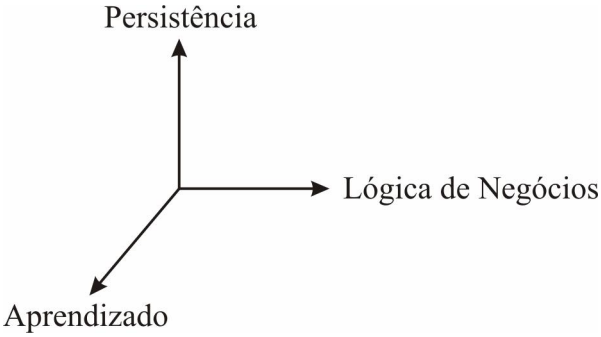

Figura 4.3 – Decomposição de interesses numa visão multidimensional.
Fonte: Adaptado de LADDAD (2003)

Segundo Hürsch e Lopes (1995), é possível distinguir dois níveis de separação de interesses: o conceitual e de implementação. No conceitual, a separação de interesses se limita em prover uma definição clara e uma identificação conceitual de todos os interesses que se distinguem de outros e assegurar que cada conceito é individual no sentido de que ele não é uma composição de conceitos. No de implementação, a separação de interesses precisa prover uma organização adequada que consiga isolar os interesses; o objetivo deste nível é separar os blocos de código que lidam com diferentes interesses e prover um acoplamento menos rígido entre eles.

Apesar de muitos paradigmas de desenvolvimento de software reconhecerem a importância da abstração conceitual e a separação de interesses, poucas linguagens de programação permitem que estas abstrações fiquem realmente separadas quando programadas. A separação de interesses é geralmente realizada no nível conceitual, e no nível de implementação pode variar seu teor de separação dependendo da linguagem de programação e técnica utilizada.

4.2. Modularização

Segundo Parnas (1972), a melhor maneira de lidar com a separação de interesses é por meio da criação de módulos os quais escondem suas implementações uns dos outros. Esta ideia é correta em sua essência, no entanto sabe-se que as linguagens disponíveis até então não disponibilizam meios suficientes para que esta modularização seja mantida. Como decorrência da má modularização, dois sintomas que prejudicam a manutenção e entendimento dos sistemas são causados com frequência: o emaranhamento de código[21] e o espalhamento de código[22].

O emaranhamento de código conforme descrito por Kiczales et al. (1997), é causado quando um módulo é implementado tendo que lidar com múltiplos interesses simultaneamente. A consequência deste emaranhamento do código é o aumento significativo da sua complexidade para o entendimento do mesmo (LOPES, 1997); a Figura 4.4 ilustra este sintoma ao apresentar os interesses de segurança, *logging*, persistência e mobilidade implementados num único lugar e de maneira intercalada.

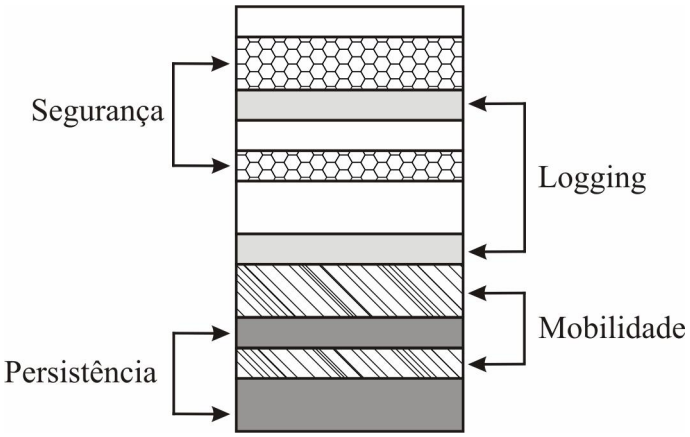

Figura 4.4 – Emaranhamento de código.
Fonte: Adaptado de LADDAD (2003)

O espalhamento de código é causado quando um procedimento é implementado em múltiplos módulos. Como os interesses transversais, por definição ficam espalhados por muitos módulos, as implementações relacionadas a eles também se dispersam por todos estes

[21] Tradução do termo em inglês *code tangling*, também traduzido como entrelaçamento de código.

[22] Tradução do termo em inglês *code scatteing*.

módulos (LADDAD, 2003). O espalhamento de código pode ser classificado em duas categorias: blocos de código duplicados e blocos de código complementares.

Blocos de código duplicados são caracterizados pela repetição de códigos que apresentam uma natureza aproximadamente idêntica, a Figura 4.5 elucida esta situação ao exibir o espalhamento de código causado pela necessidade de implementar a funcionalidade de checar autorização de acesso nos módulos: contas, ATM, cliente e atendente.

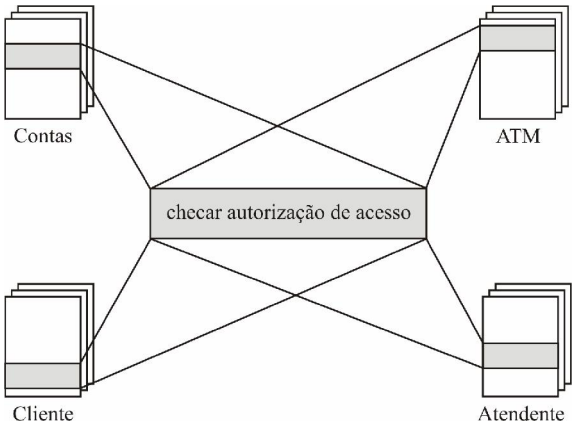

Figura 4.5 – Espalhamento de código por blocos de código duplicados.
Fonte: Adaptado de LADDAD (2003)

Blocos de código complementares são caracterizados quando alguns módulos implementam partes complementares de um interesse, a Figura 4.6 exibe o espalhamento de código causado pela necessidade de se colocar blocos de código complementares em múltiplos módulos para produzir uma funcionalidade.

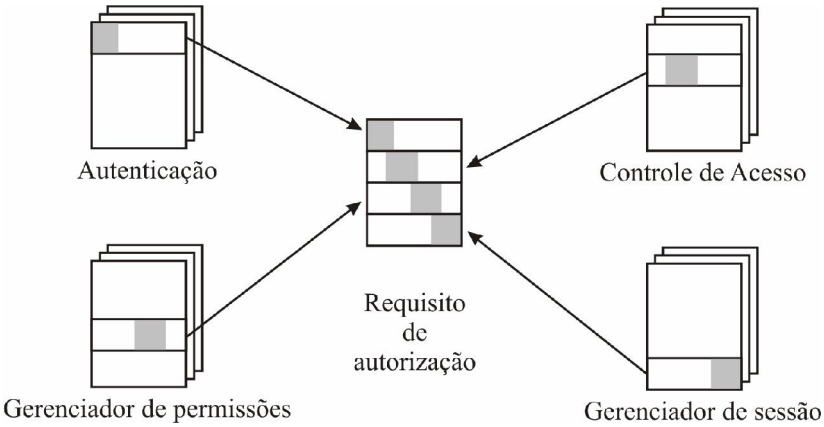

Figura 4.6 – Espalhamento de código por blocos de código complementares.
Fonte: Adaptado de LADDAD (2003)

O emaranhamento de código e o espalhamento de código juntos, impactam negativamente no projeto e desenvolvimento de software de muitas maneiras. É possível notar como consequência destes sintomas, a redução do entendimento, o aumento da dificuldade de evolução e a redução da reusabilidade dos artefatos de software (SANT'ANNA, 2004).

De acordo com Orleans (2005), as linguagens OO suportam parcialmente a separação de interesses através da utilização da herança e do polimorfismo, entretanto muitas vezes um interesse não pode ser separado apenas em classes.

Da mesma forma, Garcia (2004) afirma que a orientação a objetos possui limitações claras no tratamento de interesses que cuidam dos requisitos naturalmente envolvidos em diversas operações e componentes do sistema. Esta limitação é ilustrada pela Figura 4.7: à esquerda é mostrado um espaço tridimensional de interesses e à direita está o código unidimensional que implementa estes interesses, representado nesta figura por um fluxo contínuo de chamadas. A ortogonalidade dos interesses no espaço de interesses é perdida quando eles são mapeados para um espaço de implementação unidimensional onde seu foco é a implementação dos interesses principais, e então a implementação dos interesses transversais fica misturada com a implementação dos interesses principais.

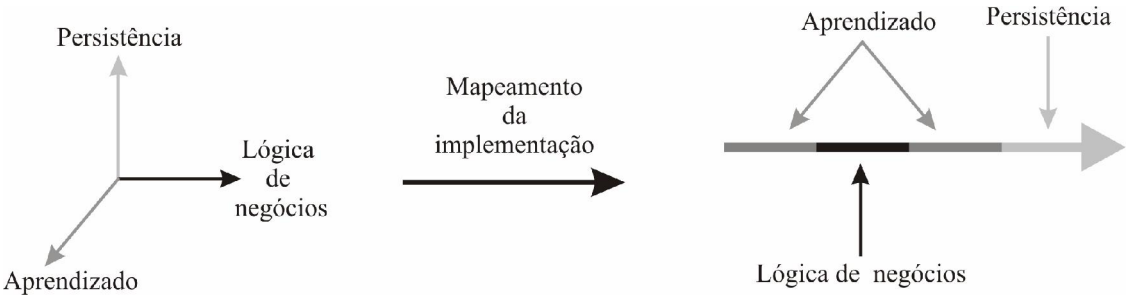

Figura 4.7 – Mapeamento de um espaço de N dimensões utilizando uma linguagem de uma dimensão. Fonte: Adaptado de LADDAD (2003)

4.3. Programação Orientada a Aspectos

Segundo Kiczales et al. (1997), existem propriedades que um sistema deve implementar e que não são modularizáveis segundo os paradigmas atualmente dominantes; estas propriedades são classificadas em componentes e aspectos. Componentes podem ser facilmente modularizados segundo o processo de decomposição funcional. Os aspectos, no entanto, são difíceis de modularizar por este processo; os aspectos são propriedades que precisam ser satisfeitas em vários componentes de um sistema, como sincronização,

persistência, *caching* etc. ou que não fazem parte das responsabilidades primárias dos componentes. Assim, o objetivo da orientação a aspectos é possibilitar ao desenvolvedor separar componentes e aspectos de forma clara, e ainda compô-los entre si da maneira apropriada de forma a constituir o sistema desejado.

A POA favorece a modularização dos interesses transversais por meio de uma abstração chamada aspecto que possibilita a sua separação e composição para produzir o sistema (SANT'ANNA, 2004). Esta abordagem contribui para a melhoria no desenvolvimento de software OO, pela separação avançada de interesses que proporciona através da utilização de aspectos. A Figura 4.8 ilustra esta forma de separação dos interesses transversais exibindo graficamente como a implementação de um interesse transversal pelos módulos de um sistema OO acaba espalhada e emaranhada, enquanto a implementação com a OA consegue modularizar este interesse transversal num único aspecto. É possível notar que na implementação OO, cada classe – representada pelo retângulo – precisa lidar diretamente com o interesse transversal em questão, enquanto na implementação OA, somente o aspecto – representado pelo losango – lida diretamente com este interesse.

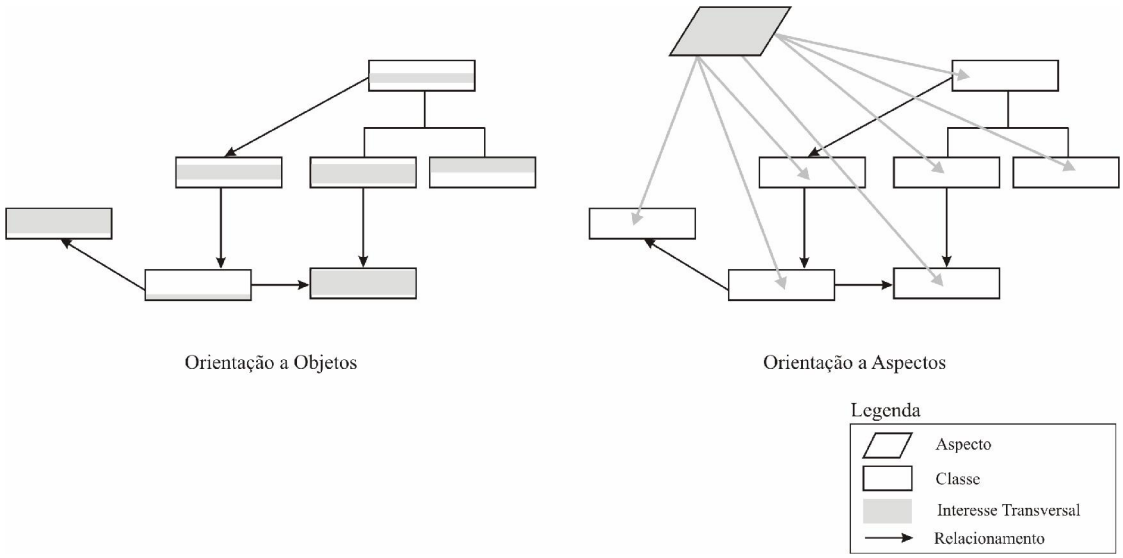

Figura 4.8 – Interesse transversal implementado em sistemas OO e OA.
Fonte: Adaptado de FIGUEIREDO (2006)

Figueiredo (2006) constata que a orientação a aspectos introduz uma terceira dimensão de decomposição quando utilizada em conjunto com a OO, ao decompor o sistema em dados e funções e, também, de acordo com os interesses transversais em abstrações denominadas aspectos.

Um sistema baseado em aspectos é composto por cinco elementos: uma linguagem de componentes, uma (ou mais) linguagem(ns) de aspectos, um (ou mais) programa(s) de componentes, um ou mais programas de aspectos e um combinador de aspectos.

A linguagem de componentes normalmente é uma linguagem de programação que se baseia teoricamente em um dos paradigmas de desenvolvimento de software[23], como Java, C++, Pascal, C, Lisp etc.

A linguagem de aspectos normalmente é uma extensão das linguagens de componentes. Alguns exemplos de linguagens de aspectos são a AspectJ (ASPECTJ, 2009), AspectC++ (ASPECTCPP, 2009), Aspect.NET (ASPECTNET, 2009) e a AspectLua (ASPECTLUA, 2009).

Dentre as linguagens de aspectos citadas, a AspectJ foi utilizada para implementação da arquitetura proposta neste trabalho.

O AspectJ é uma linguagem para programação orientada a aspectos de uso geral, criada pelo laboratório *Palo Alto Research Center*, da Xerox, e hoje é um projeto de código aberto mantido pela fundação *Eclipse* (ECLIPSE, 2009).

Ela funciona como uma extensão da linguagem Java (JAVA, 2009), sendo um superconjunto desta linguagem. Desse modo, todo programa em AspectJ, ao ser compilado, é passível de execução em qualquer máquina virtual Java (Kiczales et al., 2001).

Portanto, em uma aplicação orientada a aspectos em AspectJ, os componentes são implementados usando a sintaxe padrão de Java, e os aspectos são implementados usando a sintaxe de AspectJ.

O código do aspecto torna explícito o comportamento que é integrado ao código dos componentes, e os contextos onde a integração ocorre, os chamados pontos de junção[24].

Pontos de junção são elementos semânticos da linguagem de componentes com as quais os programas de aspectos se coordenam. Exemplos de pontos de junção comuns são: invocações de métodos – chamadas ou recebimentos –, lançamento de exceções, instanciação de objetos, entre outros.

O combinador de aspectos[25] combina os programas de componentes e de aspectos de forma a gerar o programa final (PIVETA, 2001). Ele identifica nos componentes, pontos de

---

[23] Orientado a Objetos, Procedural, Declarativa.

[24] Tradução do termo em inglês *join point.*

[25] Tradução do termo em inglês *weaver*.

junção onde os aspectos se aplicam produzindo o código final da aplicação, que implementa tanto as propriedades definidas pelos componentes como aquelas definidas pelos aspectos. Pode, ainda, atuar em tempo de compilação ou de execução, implementações de combinadores de aspectos em tempo de execução têm a possibilidade de permitir a adição ou exclusão de aspectos na aplicação em tempo de execução.

A Figura 4.9 ilustra o funcionamento do combinador de aspectos que atua em tempo de compilação. Para combinar classes e aspectos, primeiramente é realizada a compilação das classes utilizando o compilador da linguagem e em seguida elas são combinadas com os aspectos pela ação do combinador de aspectos. O produto é um programa executável, resultado da combinação das classes com os aspectos.

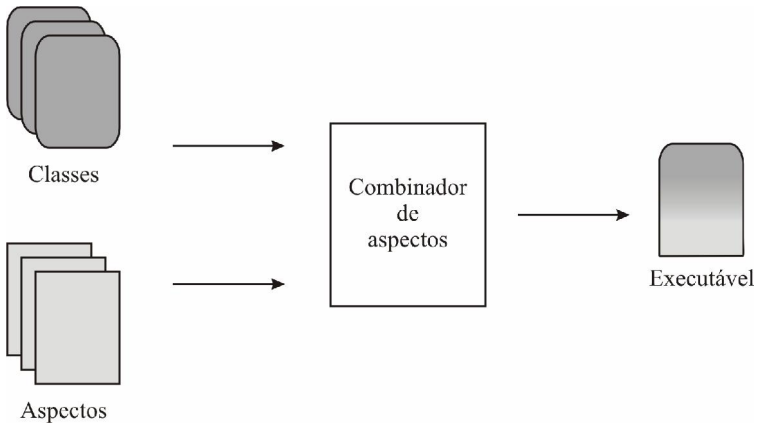

Figura 4.9 – Combinador de aspectos.
Fonte: Adaptado de LADDAD (2003)

Segundo Laddad (2003), desenvolver um sistema utilizando POA é similar a desenvolver um sistema utilizando outros paradigmas: é preciso identificar os interesses, implementá-los e por último formar um sistema final combinando-os.

Geralmente, a comunidade de pesquisa de POA define este processo em três passos: (I) decomposição dos aspectos, (II) implementação dos interesses e (III) recomposição dos aspectos.

No primeiro passo, os requisitos são decompostos com a finalidade de identificar quais são interesses principais e transversais. No segundo passo, cada interesse é implementado de maneira independente. Por último, são especificadas as regras para a recomposição dos aspectos criados e então, o combinador de aspectos utiliza estas informações para combiná-los e formar o sistema final.

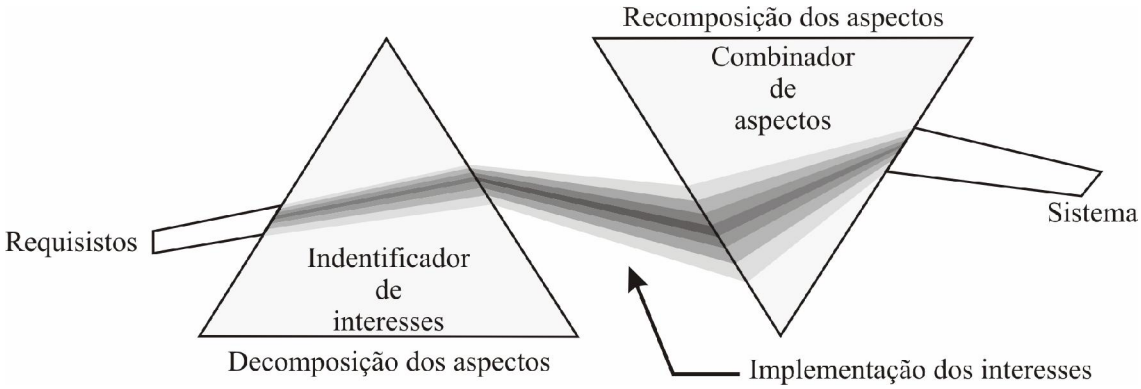

Figura 4.10 – Composição de aspectos.
Fonte: Adaptado de LADDAD (2003)

A Figura 4.10 ilustra graficamente os três passos deste processo descritos anteriormente. Ao utilizar novamente a analogia de um feixe de luz passando por um prisma, é possível observar a decomposição dos aspectos – representada pelo feixe de luz passando por um prisma que a decompõe em espectros –; estes espectros representam os interesses – que são implementados pelos aspectos –. Por último, a recomposição dos aspectos é realizada pelo combinador de aspectos – representado pelo prisma invertido que junta todos os espectros num único feixe de luz que representa o sistema.

Neste capítulo foram abordadas a questão da separação de interesses, os sintomas da má modularização e foi apresentada a programação orientada a aspectos, um paradigma complementar aos existentes, cujo objetivo é prover a separação de interesses transversais em módulos fisicamente separados chamados aspectos. A arquitetura de controle de acesso RBAC em agentes, proposta por este trabalho, será apresentada no próximo capítulo.

5. Controle de Acesso RBAC na Arquitetura do Agente

Este capítulo relata as principais decisões tomadas tendo em vista a implementação do controle de acesso RBAC na arquitetura do agente com um viés para a separação de interesses.

A aplicação do conceito de papéis em agentes é apontada como um padrão de projeto para modelagem das interações dos agentes que permite a separação de responsabilidades e o reuso de código (WOOLDRIDGE; JENNINGS; KINNY, 1999; CABRI; FERRARI; ZAMBONELLI, 2003).

Uma aplicação dos papéis em agentes é trabalhada por Viroli et al. (2007) onde é apresentado o modelo RBAC-MAS. Este modelo propõe a modelagem e organização de um SMA seguindo os conceitos do modelo RBAC. Os autores também observam que três adaptações conceituais no modelo RBAC são necessárias para que este fique adequado ao contexto dos SMAs.

A primeira adaptação é o entendimento de que neste modelo a entidade Usuário (U) é o próprio agente do sistema. A segunda é uma consequência da primeira caracterizada pela relação Atribuição de Agentes (UA), onde os Papéis (R) são atribuídos diretamente a um agente, de acordo com os papéis que este desempenha. Por último, as Permissões (P) são concedidas sobre as ações e percepções dos agentes e não acerca dos recursos que o sistema possa oferecer. O conceito de sessão e a relação Atribuição de Permissões (PA) permanecem inalterados.

Os principais componentes deste modelo podem ser formalizados através da definição:

- U, R, P, (agentes, papéis e permissões respectivamente);
- $UA \subseteq U \times R$, relação muitos para muitos de agentes para papel;
- $PA \subseteq P \times R$, relação muitos para muitos de permissão para papel.

A Figura 5.1 apresenta graficamente as entidades do modelo RBAC-MAS, à esquerda é exibida a entidade Agentes (U), a relação Atribuição de Agentes (UA) e a entidade

Sessões(S), que também está presente neste modelo de forma facultativa. Ao centro é ilustrada a entidade Papéis (R) que está ligada à entidade Permissões (P) por meio da relação Atribuição de Permissões (PA).

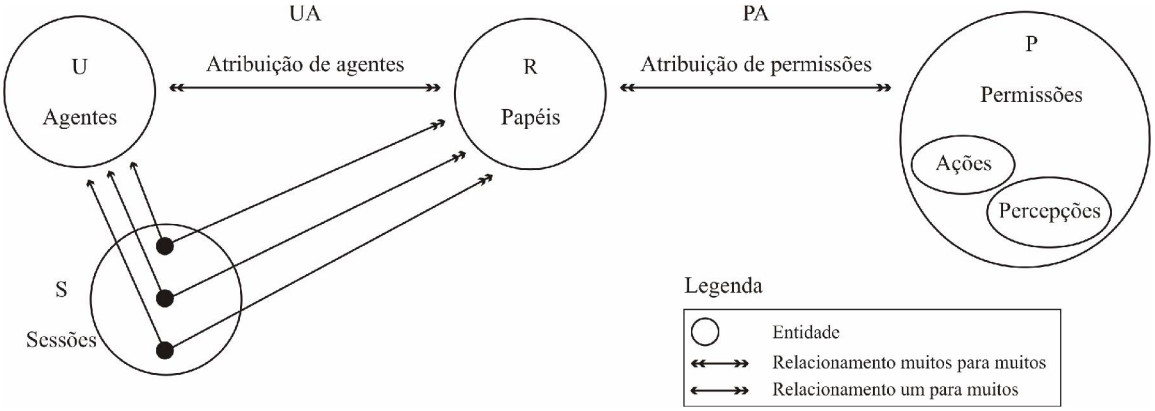

Figura 5.1 – Modelo RBAC-MAS.
Fonte: Adaptado de VIROLI; OMICINI; RICCI (2007)

Do mesmo modo que o conceito de papel é empregado na modelagem e organização dos agentes, também é aplicado no controle de acesso dos agentes (CABRI; FERRARI; LEONARDI, 2005), especialmente no controle de acesso segundo o modelo RBAC, onde o papel é a entidade central (NAVARRO et al., 2005; XIAO et al., 2007; QUILLINAN et al., 2008).

Ao observar os sistemas orientados a agentes modelados com o conceito de papel e o modelo de controle de acesso RBAC, nota-se que ambos possuem o papel como elemento central e comum. Em virtude disto, este trabalho considera que o modelo RBAC é adequado para o controle de acesso em sistemas orientados a agentes modelados segundo o conceito de papéis, pois neste contexto, conforme já elaborado por Viroli et al. (2007) o agente é entendido como o usuário que desempenha papéis.

No entanto, a implementação do interesse de controle de acesso, que é tipicamente um interesse transversal, causa paralelamente sintomas indesejáveis no sistema como o espalhamento e entrelaçamento de código quando realizada segundo as formas tradicionais de modularização (BODKIN, 2004; SHAH; HILL, 2004), de modo que a separação avançada de interesses se faz desejável.

Tendo como fundamento a abordagem em que a separação de interesses é inerente ao controle de acesso, esta arquitetura estrutura estes dois conceitos – separação de interesses e

controle de acesso – em camadas, seguindo as diretrizes estabelecidas pelo padrão arquitetural *Layer* (BUSCHMAN et al., 1996), onde cada camada representa um nível de abstração.

A estruturação da arquitetura do agente em camadas não é uma ideia nova; Kendall et al. (1999) propõem o padrão arquitetural **Agente em camadas** que é fundamentado no padrão *Layer*, para separar os diferentes interesses que um agente deve tratar como por exemplo a autonomia, a mobilidade, a interação etc.

No contexto deste trabalho, a arquitetura dos agentes está composta por três camadas:

A primeira camada trata das propriedades intrínsecas dos agentes como: a autonomia, a interação e os papéis. É nesta camada que os sensores e atuadores dos agentes estão desenvolvidos. Sua discussão não faz parte do escopo deste trabalho.

O controle de acesso RBAC é de responsabilidade da segunda camada. Sua função é apenas prover o mecanismo para a autenticação dos agentes e o manuseio de seu usuário, papéis e permissões.

A terceira camada é responsável pela separação do interesse de controle de acesso por meio dos aspectos. Ela atua como um intermediário que utiliza as funcionalidades de controle de acesso que estão na segunda camada e realiza a ligação com as propriedades essenciais dos agentes que estão na primeira camada.

Desta forma, o interesse de controle de acesso é incorporado aos agentes sem a necessidade deles implementá-lo diretamente, o que causaria os sintomas da má modularização discutidos no capítulo anterior. Os detalhes de construção da camada de controle de acesso e a de separação do interesse, chamada de camada de aspectos de segurança, são discutidos em seções distintas no decorrer deste capítulo.

Para auxiliar no desenvolvimento dos agentes foi utilizado o arcabouço[26] *Java Agent Development Framework* (JADE) (BELLIFEMINE et al., 2003). Este arcabouço OO é destinado ao desenvolvimento de SMAs em conformidade com o padrão FIPA (FIPA, 2002). Sob o ponto de vista de Caire (2003), o JADE pode ser entendido também como um *middleware*, pois além das bibliotecas de classes que oferece para criação de agentes, disponibiliza um ambiente de execução para os agentes e ferramentas gráficas para a administração e monitoramento das atividades dos agentes que são executados na plataforma.

Três propriedades foram decisivas para a escolha deste arcabouço. A primeira é o fato de um agente no JADE ser representado por um único componente Java – um exemplo está

[26] Tradução do termo em inglês *framework*.

apresentado no Quadro 5.1 –, característica que viabiliza a realização do processo de combinação dos aspectos que permite a atuação dos aspectos nos agentes. A segunda é o mecanismo conhecido como páginas amarelas[27], que permite a localização de um agente que fornece um determinado serviço. E por último, a comunicação entre agentes por mensagens FIPA ACL (FIPA, 2002), que permite aos aspectos examinarem seu conteúdo. O emprego destas propriedades na arquitetura proposta é demonstrado nas próximas seções, cujo objetivo é explicar as camadas da arquitetura e seus principais componentes.

[27] Tradução do termo em inglês *yellow pages*, também conhecido como *Directory Facilitator (DF)*.

Quadro 5.1 – Código-fonte de um agente no JADE.

```
package br.mestrado.hosp.agents;

import jade.core.Agent;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

/**
 * Exemplo de agente no JADE
 * */
public class Supervisor extends Agent {

	protected void setup() {

		// Set this agent main behaviour
		addBehaviour(new ReceiveMessagesBehaviour(this));
	}

	// Behaviour
	private class ReceiveMessagesBehaviour extends CyclicBehaviour {

		public ReceiveMessagesBehaviour(Agent a) {
			super(a);
		}

		public void action() {
			ACLMessage msg = receive();
			if (msg == null) {
				block();
				return;
			}

			try {
				// responde pelo tipo de msg
				switch (msg.getPerformative()) {

				case (ACLMessage.INFORM): {
					break;
				}
				case (ACLMessage.REQUEST): {
					break;
				}
				case (ACLMessage.QUERY_REF): {
					break;
				}
				default:
					break; //replyNotUnderstood
				}

			} catch (Exception ex) {
				ex.printStackTrace();
			}
		}
	}
}
```

5.1. Camada de Controle de Acesso

Esta camada cuja função é prover a autenticação e controle de acesso aos agentes, é constituída por dois elementos: um mecanismo de controle de acesso RBAC chamado *Heimdall* e um agente chamado *Supervisor*.

Até o momento da finalização desta pesquisa não foi encontrado nenhum mecanismo ou arcabouço para controle de acesso NIST RBAC que pudesse ser aplicado aos agentes desenvolvidos com o JADE, por conta disto o *Heimdall* foi construído.

O *Heimdall* é um mecanismo de autenticação e controle de acesso que implementa o modelo NIST RBAC, apresentado no capítulo três. Portanto, a principal função deste mecanismo é atuar como um monitor de referência de forma a realizar o controle de acesso dos agentes.

Para construção deste mecanismo foi utilizado como base, o *Java Authentication and Authorization Service* (JAAS) (2009), uma biblioteca do Java para autenticação e controle de acesso. Desta forma, as funções básicas de autenticação e controle de permissões dos usuários puderam ser aproveitadas pelo *Heimdall* podendo-se assim, concentrar os esforços para sua construção na tarefa de adaptação das funções já existentes de maneira a trabalhar conforme o modelo RBAC.

O segundo elemento integrante desta camada é o agente *Supervisor*. Sua função é realizar a autenticação dos agentes que através do envio de mensagens FIPA ACL (FIPA, 2002) solicitam autenticação e comunicar a possível falta de permissão de algum agente quando este tenta executar uma ação a qual não possui a devida permissão.

A ideia de estabelecer um único agente responsável por atender ao pedido de autenticação de todos os agentes que interagem num determinado ambiente é trabalhada por Cabri et al. (2004), que apresentam o agente *Supervisor*. Neste trabalho o agente *Supervisor* é utilizado com mesmo conceito, porém ao contrário da ideia original onde este agente assumia um papel para cada ambiente em que era capaz de realizar a autenticação, nesta arquitetura este agente assume um único papel: o de *supervisor*. Ao desempenhar este papel ele é capaz de autenticar tanto os agentes que estão no mesmo ambiente em que ele se encontra quanto os agentes que estão em outros ambientes.

O agente *Supervisor*, assim que criado, registra-se no serviço de páginas amarelas como provedor do serviço de autenticação. Desta forma, todos os agentes que desejam

participar da sociedade podem procurar por um agente capaz de realizar sua autenticação e solicitar esta operação antes de começar a interagir com os demais agentes.

Para que um agente seja autenticado deve enviar uma mensagem do tipo *SolicitarAutenticacaoAction* para o agente *Supervisor* contendo o usuário e senha que são utilizados pelo *Heimdall* no processo de autenticação. Caso a autenticação seja bem sucedida, o agente é registrado pelo *Heimdall* numa lista de agentes autenticados onde é armazenado o identificador único do agente e o usuário que ele utilizou. Desta forma, é possível controlar os agentes autenticados e os usuários utilizados. De modo que a partir do usuário utilizado por um agente é possível conhecer os papéis e permissões aos quais ele tem direito.

Uma vez autenticado, o agente está submetido – de forma direta ou indireta –, ao controle de acesso através da atuação dos aspectos de controle de acesso. As formas de atuação destes aspectos são discutidas na próxima seção.

A Figura 5.2 ilustra graficamente a disposição dos elementos envolvidos no processo de autenticação. É possível identificar o agente *Supervisor* e os demais agentes que enviam mensagens do tipo *SolicitarAutenticacaoAction* para o *Supervisor* a fim de serem autenticados. Também é demonstrado que os agentes podem estar em dois ambientes distintos: interno e externo.

O ambiente interno é definido neste trabalho como o ambiente onde o agente *Supervisor* se encontra e por isto, é também o ambiente onde o controle de acesso está implementado. Já o externo é definido como qualquer ambiente que interage com o interno embora o controle de acesso não esteja presente.

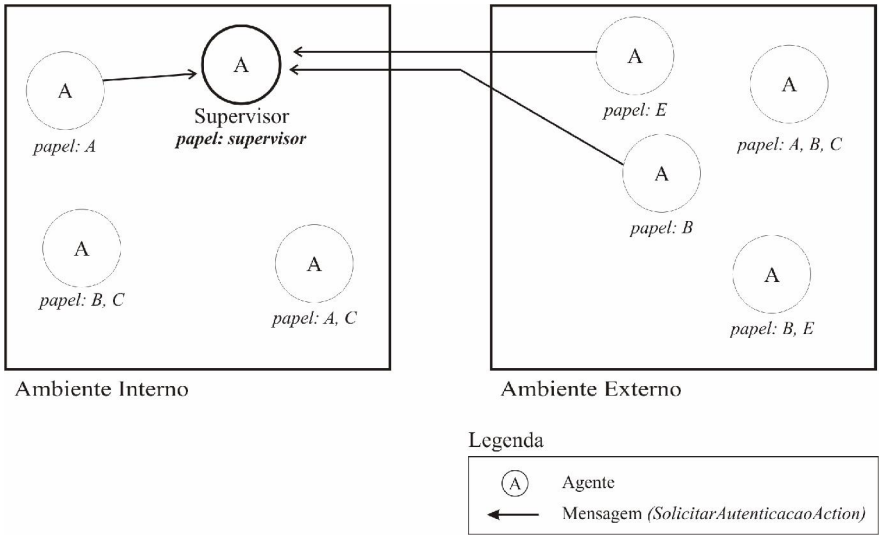

Figura 5.2 – Agente *Supervisor* responsável por autenticar os agentes de diferentes ambientes. Fonte: Adaptado de CABRI; FERRARI; LEONARDI (2004)

Embora todos os agentes devam ser autenticados, o procedimento explicado anteriormente é recomendado somente para agentes que são executados num ambiente externo, pois conforme discutido na próxima seção, a autenticação dos agentes que estão no ambiente interno pode ser realizada pela atuação do aspecto *ProactiveAuthenticationAspect*, de maneira que o fluxo de mensagens desnecessárias é evitado.

5.2. Camada de Aspectos de Segurança

A função desta camada é implementar o interesse de controle de acesso nos agentes de forma que esteja presente sem que estes tenham que lidar diretamente com ele. Para alcançar este nível de independência a POA é utilizada.

Esta camada é composta por sete aspectos de segurança:

- *SecurityAspect;*
- *AuthenticationAspect;*
- *ProactiveAuthenticationAspect;*
- *ReactiveAuthenticationAspect;*
- *AccessControlAspect;*
- *ProactiveAccessControlAspect;*
- *ReactiveAccessControlAspect.*

Os aspectos foram agrupados de acordo com sua função, autenticação ou controle de acesso e modelados de acordo com o diagrama de classes exibido na Figura 5.3. No topo do diagrama está o aspecto *SecurityAspect;* ele é o ancestral de todos os aspectos de segurança. Este aspecto possui como propriedade uma referência ao agente *Supervisor* e esta referência é herdada e utilizada pelos seus aspectos descendentes.

Dois aspectos estendem diretamente o *SecurityAspect*: *AuthenticationAspect* e *AccessControlAspect.* Estes dois, assim como o ancestral comum, são abstratos e apenas

fornecem funcionalidades básicas para os quatro aspectos que atuam diretamente nos agentes do ambiente interno: *ProactiveAuthenticationAspect*, *ReactiveAuthenticationAspect*, *ProactiveAccessControlAspect* e *ReactiveAccessControlAspect.* Estes quatro são os principais desta arquitetura, por isto são apresentados individualmente e em detalhes no decorrer desta seção.

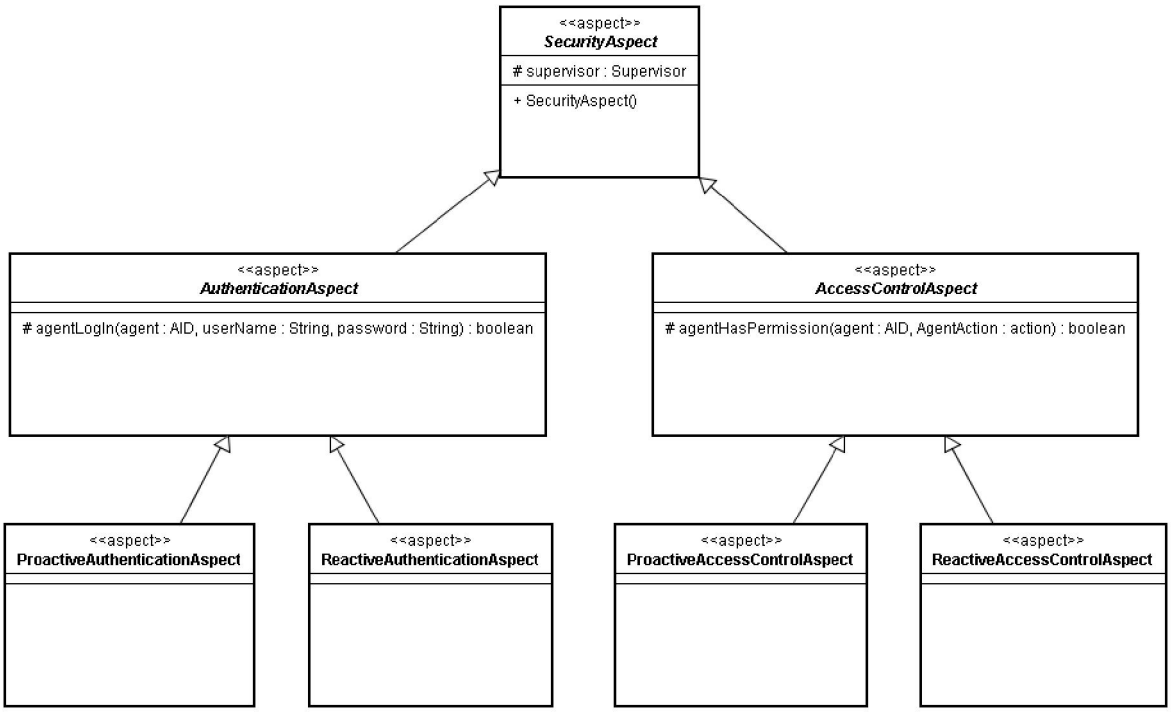

Figura 5.3 – Diagrama de classes dos aspectos de segurança.

Os aspectos de autenticação e controle de acesso atuam de duas maneiras distintas, por isto são classificados como proativos ou reativos.

Os aspectos proativos atuam nos agentes que estão no ambiente interno, de forma a evitar a troca de mensagens desnecessárias entre os agentes.

Aspectos reativos também atuam nos agentes que estão no ambiente interno, mas somente quando estes recebem mensagens de agentes que estão em ambiente externo e por isso não é possível exercer o controle de acesso proativo.

Numa visão geral, os aspectos atuam nos sensores e atuadores dos agentes, responsáveis respectivamente por receber e enviar mensagens FIPA ACL. Independentemente da forma de atuação do aspecto – proativo ou reativo – na maioria dos casos, sua atividade se

resume à interceptação e análise do conteúdo das mensagens seguida da legitimação ou não, da troca de mensagem.

Para a análise do conteúdo da mensagem os parâmetros *sender*, *receiver* e *content* da mensagem FIPA ACL são utilizados pelo aspecto, pois a partir deles é possível obter o agente emissor, receptor e o conteúdo, ou melhor, a ação que a mensagem representa. E, com estas informações, o aspecto averigua através do *Heimdall* se os agentes emissor e receptor possuem as permissões necessárias para lidarem com aquela ação. Caso possuam, a troca de mensagem é legitimada, senão esta operação é abortada.

Com esta estratégia o aspecto evita o processamento desnecessário de mensagens pelos sensores e receptores dos agentes, pois somente mensagens válidas são trocadas, além de promover a separação do interesse de controle de acesso que está presente nos agentes sem que estes lidem de forma direta com ele.

A linguagem AspectJ (ASPECTJ, 2009) foi utilizada para a programação dos aspectos de segurança e, por conta disto, alguns conceitos essenciais para a compreensão do funcionamento destes aspectos são apresentados a seguir.

5.2.1. Componentes do AspectJ

Os principais componentes do AspectJ utilizados neste trabalho foram os pontos de junção, pontos de atuação[28], atuadores[29] e aspectos.

Ponto de junção é qualquer ponto de execução identificável de um sistema, é um dos conceitos fundamentais da POA e consequentemente, do AspectJ. Como exemplos de pontos de junção em uma linguagem OO podem ser citados:

- chamada de método;
- atribuição de um valor a uma variável;
- sentença de retorno de um método;
- instanciação de um objeto;

[28] Tradução do termo em inglês *pointcut*.

[29] Tradução do termo em inglês *advice*, também traduzido como adendo e comportamento ortogonal.

- checagem de uma condição;
- uma comparação;
- o manuseio de uma exceção[30];
- laços interativos como *for, while, do*/*while.*

Apesar desta grande variedade de pontos de junção, o AspectJ, com a finalidade de evitar dependência de implementação ou ortogonalidade instável, disponibiliza somente alguns deles para a utilização, os quais são conhecidos como pontos de junção expostos. Em geral, este tipo de ponto de junção são os lugares no código fonte do sistema onde é possível alterar a execução principal do programa. Entres os pontos de junção expostos estão:

- chamada e execução de métodos;
- instanciação de objetos;
- acesso a atributos;
- manuseio de exceções.

Os aspectos de controle de acesso deste trabalho utilizam pontos de junção referente à execução dos métodos dos agentes responsáveis pelo envio e/ou recebimento de mensagens.

Todos os pontos de junção possuem um contexto associado a eles, por exemplo, uma chamada a um ponto de junção de um método tem o objeto que realizou a chamada, o objeto alvo e os argumentos do método disponíveis para o contexto.

Pontos de atuação definem, ou melhor, capturam pontos de junção em um fluxo de um programa (LADDAD, 2003); são estruturas utilizadas para sinalizar o local e o momento de atuação dos aspectos no programa (FIGUEIREDO, 2006).

Uma vez identificado um ponto de junção, o ponto de atuação é utilizado para especificar regras de combinação envolvendo pontos de junção e estas regras serão utilizadas pelos aspectos para realizar um bloco de código antes ou depois da execução de um ou mais pontos de junção (LADDAD, 2003).

Um ponto de atuação é definido através de um ou mais designador de ponto de atuação (*Pointcut Designator,* PCD). Existem diversos PCDs predefinidos na linguagem, que

[30] Tradução do termo em inglês *exception.*

são conhecidos como PCDs primitivos. A Tabela 5.1 exibe apenas os que foram utilizados neste trabalho.

Tabela 5.1 – PCDs primitivos utilizados.

Designadores	Descrição
args	Expõem ou limitam os argumentos de um ponto de junção para o ponto de atuação
call	Chamada de um método ou construtor
execution	Execução de um método ou construtor
within	Captura um ponto de junção de um determinado tipo

A palavra-chave *pointcut* é usada para definir que um ou mais PCDs constituem um ponto de atuação e de maneira semelhante à declaração de métodos no Java, é possível utilizar modificadores de acesso como o *public* e o *private.*

Um ponto de atuação pode ser classificado como: nomeado ou anônimo. O ponto de atuação nomeado é um elemento que pode ser referenciado em muitos lugares fazendo dele um elemento reutilizável. Já o anônimo é semelhante à classe anônima da OO, ou seja, é definido no lugar em que é utilizado, como uma parte de um atuador ou para definição de outro ponto de atuação.

Pelo fato do ponto de atuação anônimo ser definido somente no momento de sua utilização seu reuso fica inviabilizado e consequentemente, na prática, seu uso não é recomendado quando o código da declaração do ponto de atuação é complexo.

O atuador é uma estrutura parecida com um método da OO; através dele é definido em que momento e como o aspecto irá atuar no ponto de junção. Ele auxilia na questão do **"o que fazer"** ao definir o comportamento a ser executado pelo aspecto nos pontos de junção associados. A estrutura do atuador designa a semântica comportamental do aspecto (FIGUEIREDO, 2006).

Existem três tipos de atuadores no AspectJ: atuador do tipo anterior, posterior e de contorno.

O atuador do tipo anterior, utilizado através da palavra-chave *before*, define que o bloco de código do atuador deve ser executado antes da execução do ponto de atuação.

O atuador do tipo posterior, usado através da palavra-chave *after*, define que o bloco de código do atuador deve ser executado depois da execução do ponto de atuação. Existem

três variações deste atuador que estão relacionadas à execução do ponto de atuação, que pode ter terminado normalmente, gerado uma exceção ou terminado normalmente ou não.

O atuador do tipo de contorno, utilizado através da palavra-chave *around*, é utilizado para substituir – ou não – a execução do bloco de código do ponto de atuação pelo bloco de código do atuador, portando ele pode ser utilizado para desviar o fluxo de execução de um programa.

Os aspectos no AspectJ são construções similares a classes, que encapsulam, além dos atributos e métodos convencionais, atuadores e PCDs. Sant'Anna (2004) define o aspecto como a unidade de modularidade para interesses transversais onde cada aspecto encapsula uma funcionalidade que atravessa outras classes do programa.

É possível verificar a aplicação desta ideia pela análise dos aspectos desta arquitetura que encapsulam a funcionalidade de controle de acesso que **atravessa** as classes que representam os agentes.

As várias maneiras de escrever um aspecto e associá-lo ao código principal da aplicação estão diretamente relacionadas ao interesse ao qual será implementado através do aspecto. Um aspecto pode estar escrito em seu próprio arquivo, semelhante às classes Java que são escritas em arquivos com seu nome ou podem estar escritos no mesmo arquivo de uma classe.

O Quadro 5.2 apresenta o *SecurityAspect* como um exemplo de aspecto. Através dele é possível observar a semelhança da estrutura de um aspecto com uma classe Java e as utilizações dos elementos discutidos nesta seção como o ponto de junção, ponto de atuação e atuadores.

Quadro 5.2 – Exemplo de aspecto.

```
package br.mestrado.hosp.aspects.security;

import org.heimdall.Heimdall;
import br.mestrado.hosp.agents.Supervisor;

/**
 * Aspecto responsável pela segurança (base)
 * */
public abstract aspect SecurityAspect {

   //
   // Pega o agente supervisor da aplicação
   //
   protected Supervisor supervisor = null;

   //pointcut
   pointcut supervisorSetup() :
   execution(void br.mestrado.hosp.agents.Supervisor.setup());//PCD

   //advice
   after(): supervisorSetup(){
      this.supervisor = (Supervisor) thisJoinPoint.getTarget();
   }

   //
   // Inicializa o Heimdall
   //
   public SecurityAspect() {
      Heimdall.authenticationManager.setApplicationName("hosp");
   }

}
```

5.2.2. Aspectos de autenticação

O interesse de autenticação é modularizado por três aspectos: *AuthenticationAspect*, *ProactiveAuthenticationAspect* e *ReactiveAuthenticationAspect*.

O aspecto *AuthenticationAspect* é o aspecto ancestral da hierarquia de aspectos de autenticação; ele implementa através da utilização do *Heimdall*, a funcionalidade de autenticação de agentes que é utilizada pelos seus descendentes.

O aspecto *ProactiveAuthenticationAspect* promove a autenticação dos agentes do ambiente interno assim que estes são configurados e desta forma, não é preciso que estes agentes tenham que realizar o esforço desnecessário de enviar uma mensagem ao agente *Supervisor* solicitando sua autenticação, pois ela ocorre automaticamente. Para que esta

estratégia seja viável é preciso que os agentes sejam modelados de forma a possuírem como propriedades as informações necessárias para sua autenticação: o usuário e a senha. Desta forma este aspecto pode capturar estas informações para serem utilizadas no processo de autenticação. A Figura 5.4 ilustra a atuação proativa deste aspecto – representado pelo losango – nos agentes do ambiente interno. É possível notar também que somente o aspecto lida diretamente com o interesse de autenticação através do *Heimdall* – ilustrado pelo circulo hachurado contido no aspecto –.

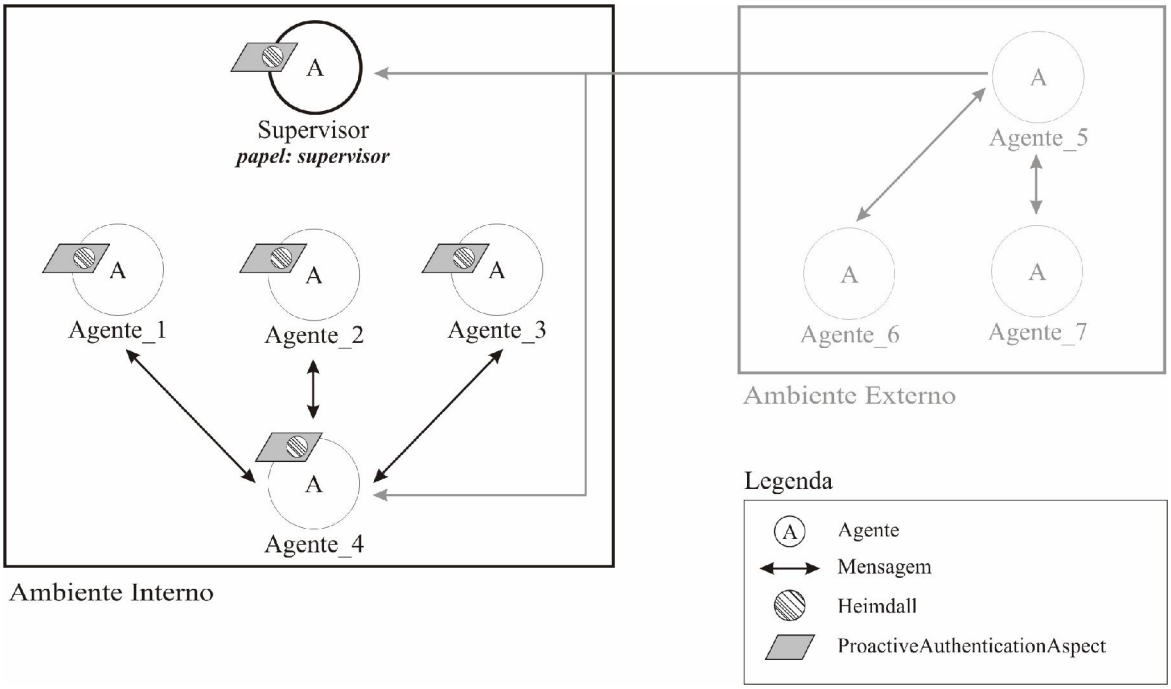

Figura 5.4 – Ilustração da atuação do aspecto *ProactiveAuthenticationAspect*.

O aspecto *ReactiveAuthenticationAspect* atua no agente *Supervisor* interceptando as mensagens que este recebe dos agentes que estão num ambiente externo. Esta forma de autenticação é entendida como reativa, pois é decorrente de uma solicitação de autenticação ao contrário da proativa onde é realizada automaticamente.

O agente *Supervisor* serve apenas como uma referência para os agentes, pois todo processo de autenticação é realizado pelo aspecto. Desta forma, a autenticação é realizada sem a necessidade do agente *Supervisor* lidar com este interesse.

A Figura 5.5 representa graficamente a atuação deste aspecto no agente *Supervisor*. Antes que qualquer mensagem seja percebida pelo sensor deste agente, é interceptada por este aspecto (Figura 5.5, a). Se a mensagem for um pedido de autenticação enviada por outro

agente, ou seja, se a mensagem é uma ação do tipo *SolicitarAutenticacaoAction*, este aspecto realiza o processo de autenticação do agente emissor (Figura 5.5, b), caso contrário o fluxo de processamento da mensagem é devolvido para o *Supervisor*. Caso a autenticação tenha sido mal sucedida, este aspecto faz com que um atuador do agente *Supervisor* emita uma mensagem do tipo *InformarAcessoNegadoAction* ao agente solicitante da autenticação (Figura 5.5, c), onde é especificado que ocorreu um erro na autenticação do agente.

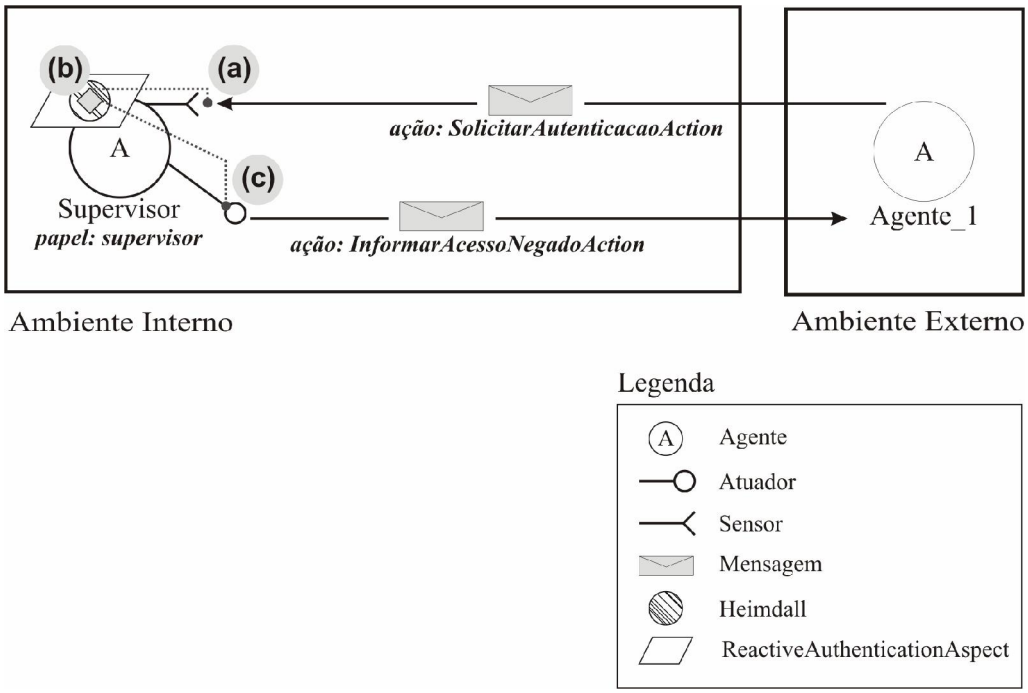

Figura 5.5 – Representação gráfica da atuação do aspecto *ReactiveAuthenticationAspect*.

5.2.3. Aspectos de controle de acesso

O interesse de controle de acesso é modularizado por três aspectos: *AccessControlAspect*, *ProactiveAccessControlAspect* e *ReactiveAccessControlAspect*.

O aspecto *AccessControlAspect* é o aspecto ancestral da hierarquia de aspectos de controle de acesso, ele implementa através da utilização do *Heimdall* a funcionalidade de controle de acesso de agentes que é utilizada pelos seus descendentes.

Numa visão geral, os aspectos *ProactiveAccessControlAspect* e *ReactiveAccessControlAspect*, promovem o controle de acesso dos agentes ao interceptar as mensagens FIPA ACL e analisar as permissões de acesso dos envolvidos na troca de mensagens.

Para que esta estratégia seja viável é preciso que para cada ação passível de execução pelos agentes, existam as permissões associadas. Pois estes aspectos analisam os parâmetros *sender*, *receiver* e *content* da mensagem para verificar se o agente emissor – contido no parâmetro *sender* – e receptor – contido no parâmetro *receiver* –, possuem permissão para trabalharem com a ação – contida no parâmetro *content* –.

O aspecto *ProactiveAccessControlAspect* intercepta e analisa as mensagens que os atuadores dos agentes situados no ambiente interno enviam para outros agentes. A Figura 5.6 representa graficamente esta ação.

Assim que um atuador do agente emissor é acionado para enviar uma mensagem, o aspecto a intercepta (Figura 5.6, a) e através dos valores dos parâmetros *sender*, *receiver* e *content* verifica se os agentes emissor e receptor possuem permissão para trabalharem com a ação desejada (Figura 5.6, b). Se ambos possuem as devidas permissões, então o processo de envio é continuado e a mensagem é entregue (Figura 5.6, c). Caso contrário, este processo é abortado e o tratamento deste erro por falta de permissão é executado (Figura 5.6, d).

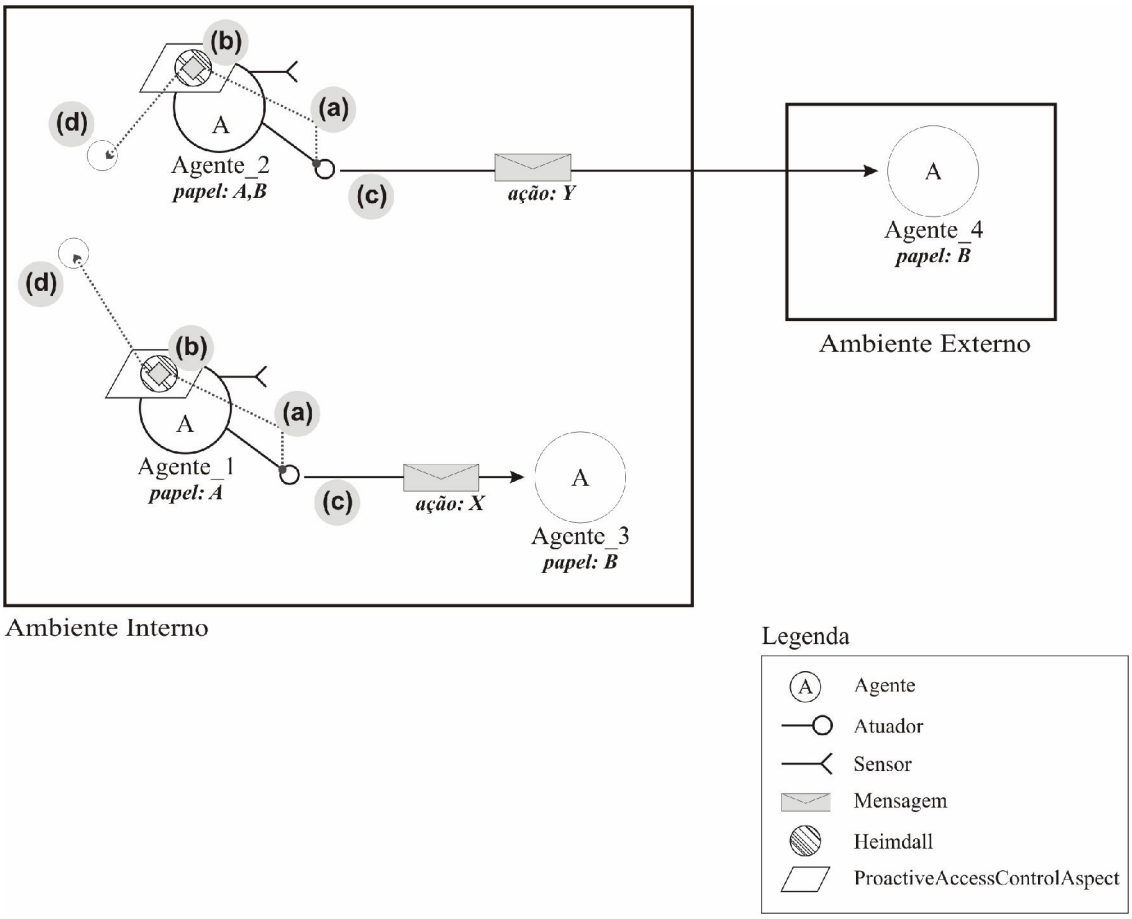

Figura 5.6 – Representação gráfica da atuação do aspecto *ProactiveAccessControlAspect*.

Devido a atuação proativa deste aspecto, não efetivado no sistema a situação de um agente solicitar uma ação a outro sendo que um dos envolvidos na troca de mensagem não possui permissão para trabalhar com esta ação. Desta forma, além da separação do interesse alcançada pela utilização do aspecto é evitado o esforço desnecessário do envio e recebimento de mensagem por parte dos agentes, pois somente ocorre a troca de mensagens válidas.

O aspecto *ReactiveAccessControlAspect* trabalha de forma semelhante ao anterior, no entanto, ao contrário do *ProactiveAccessControlAspect* ele atua nos sensores ao invés de agir sobre os atuadores do agente. Ele intercepta e analisa as mensagens que os sensores dos agentes situados no ambiente interno recebem dos agentes localizados num ambiente externo. Este processo é ilustrado pela Figura 5.7 e explicado a seguir.

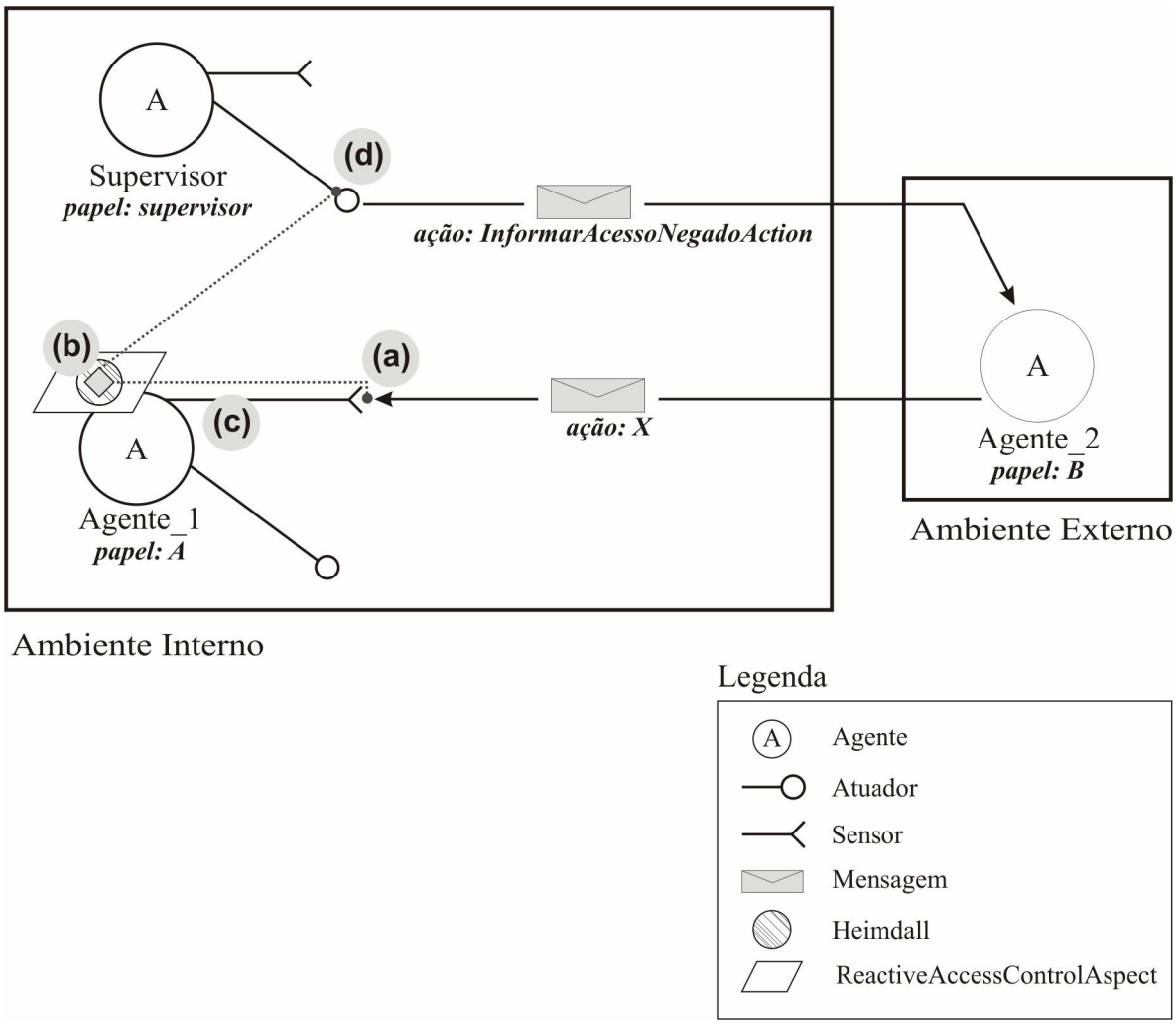

Figura 5.7 – Ilustração da atuação do aspecto *ReactiveAccessControlAspect*.

Como não é possível realizar o controle de acesso de forma proativa, ou seja, logo no envio de uma mensagem de um agente externo, esta mensagem é interceptada (Figura 5.7, a) e analisada (Figura 5.7, b) por este aspecto antes que seja percebida pelo sensor do agente receptor. Se tanto o agente emissor quanto o receptor possuírem a permissão necessária para a ação, a mensagem é recebida normalmente pelo receptor (Figura 5.7, c). Caso contrário, este aspecto fará com que o sensor do agente receptor não perceba a mensagem e em seguida acionará um atuador do agente *Supervisor* de modo que envie uma mensagem do tipo *InformarAcessoNegadoAction* para o emissor, onde é informada a falta de permissão (Figura 5.7, d).

Este capítulo apresentou as principais decisões que foram tomadas para a implementação do controle de acesso RBAC na arquitetura do agente. Foi abordada a aplicação do conceito de papéis em agentes e sua relação com o controle de acesso baseado em papéis, o arcabouço JADE e sua importância para esta arquitetura e, por fim, as camadas da arquitetura do agente e seus principais componentes e aspectos responsáveis pela aplicação e separação do controle de acesso RBAC nos agentes.

6. Testes para Verificação do Controle de Acesso

Qualquer falha na implementação do controle de acesso pode colocar em risco a efetividade de uma política de controle de acesso. Por isso, é vital assegurar através de testes que sua implementação está em conformidade com a política de segurança especificada (HUANG et al., 2009).

Myers (2004) define o teste de software como um processo, ou uma série de processos, definidos para garantir que o código do sistema faz aquilo que foi construído para fazer e nada, além disto. Ele também afirma que este processo deve ser executado com a intenção de encontrar erros no programa.

De forma semelhante, Pressman (2006) declara que o teste de *software* é um elemento frequentemente citado como verificação e validação. A verificação se refere ao conjunto de atividades que garantem que o *software* tenha implementado corretamente uma função específica. E a validação se refere a um conjunto de atividades diferentes que dão garantia de que o sistema construído atende aos requisitos do cliente. Todavia, o autor observa que existe uma forte divergência de opinião sobre quais tipos de teste constituem a validação, pois alguns autores consideram que todo teste é verificação e que a validação ocorre quando requisitos são revisados e aprovados.

Para a verificação da implementação proposta neste trabalho, o teste baseado em política de segurança (HWANG et al., 2008) é o indicado pois, conforme descrito pelos autores, existem duas motivações para a condução deste teste: (I) assegurar que a especificação da política de segurança está correta e (II) assegurar a conformidade entre a política especificada e sua implementação.

Segundo Martin (2006), o teste da política de segurança funciona de forma análoga ao teste de sistema. A Figura 6.1 ilustra esta analogia: as entradas para o teste são as requisições de acessos e as saídas são as respostas de acesso. A execução do teste ocorre quando uma requisição de acesso é avaliada pelo mecanismo de controle de acesso de acordo com a política de segurança implementada pelo mesmo. Desta forma, é possível verificar a conformidade entre a política de segurança e o mecanismo que a implementa através da inspeção das requisições, respostas e respostas esperadas.

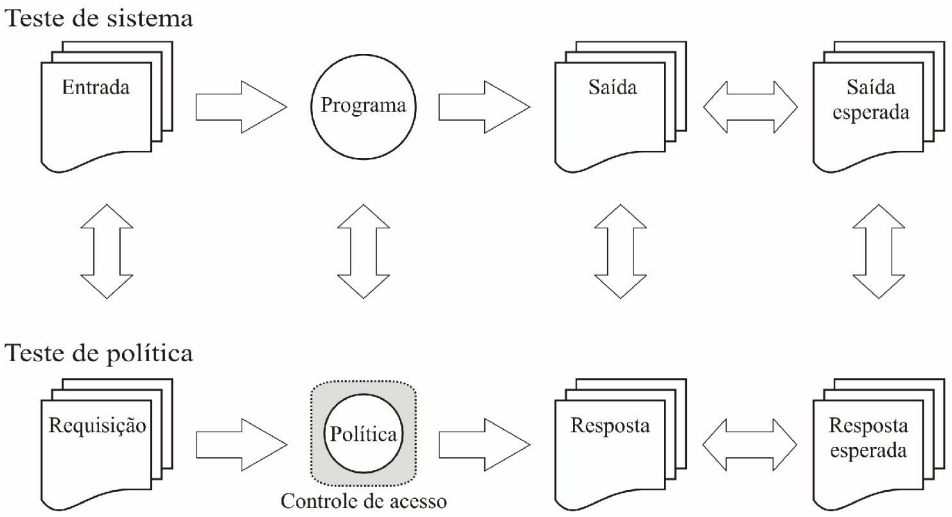

Figura 6.1 – Analogia entre teste de sistema e teste de política de segurança.
Fonte: Adaptado de Martin (2006)

Com esta abordagem é possível aumentar a certeza de que a política de segurança e sua implementação por um mecanismo estão corretas. Ao assumir que a política de segurança está especificada de forma correta, o elemento a ser testado passa a ser a sua implementação, que no caso deste trabalho é a arquitetura para controle de acesso proposta.

Hwang et al. (2008) classificam os testes baseados em política de segurança em dois tipos. No primeiro, o artefato a ser testado é a política de controle de acesso e seu principal objetivo é assegurar que ela foi especificada corretamente. No segundo tipo, o artefato a ser testado é a implementação da política, ou melhor, do controle de acesso, e o objetivo principal é assegurar a conformidade entre a política especificada e sua implementação.

Similarmente, Huang et al. (2009) definem este segundo tipo de teste como teste de controle de acesso e o classificam em dois tipos: funcional e adversário.

A responsabilidade do teste funcional de controle de acesso[31] é determinar se a implementação do controle de acesso está funcionando em conformidade com a política de segurança especificada. Já o teste adversário de controle de acesso[32] é realizado para determinar se esta implementação contém alguma vulnerabilidade – para a realização deste teste, acessos ilegais são simulados –.

Em virtude de possuírem como objetivo o teste da implementação do controle de acesso, os testes funcional e adversário citados no parágrafo anterior, tornam-se adequados

[31] Tradução do termo em inglês *functional access control test.*

[32] Tradução do termo em inglês *adversarial access control test.*

para realizar a verificação do controle de acesso RBAC implementado na arquitetura dos agentes.

6.1. Implementação para o Teste

Para validar a arquitetura de controle de acesso proposta por este trabalho, definida no capítulo anterior, elaborou-se um SMA desenvolvido com o intuito de viabilizar as simulações necessárias para efetuar os testes de controle de acesso.

Este sistema está distribuído em dois ambientes: o interno e o externo. Conforme já definido no capítulo anterior, como ambiente interno entende-se o local onde o controle de acesso está implementado e por consequência o agente *Supervisor* também está presente já, o externo, é um ambiente que interage com o interno, embora sem a presença do controle de acesso.

A distribuição dos agentes em dois ambientes distintos é necessária para a realização das simulações, pois a arquitetura proposta utiliza aspectos diferentes para tratar as requisições de cada ambiente. Aspectos proativos tratam de requisições de agentes que estão num ambiente interno enquanto os reativos daqueles que estão no externo. O diagrama de instalação[33] apresentado pela Figura 6.2 exibe estes ambientes e seus respectivos agentes criados para a realização das simulações dos cenários que são apresentados no decorrer deste capítulo.

A esquerda da Figura 6.2, está representado graficamente o ambiente interno – apelidado de *hospital01-platform* –, ele abriga o agente *Supervisor* e os demais agentes próprios deste sistema: Arquivista, Atendente, Enfermeiro, Diabetologista e Paciente. A direita está o ambiente externo – apelidado de *hospital02-platform* –, o qual é uma reprodução parcial do ambiente interno e contém apenas dois agentes: o Paciente e o Atendente.

[33] Tradução do termo em inglês *deployment diagram.*

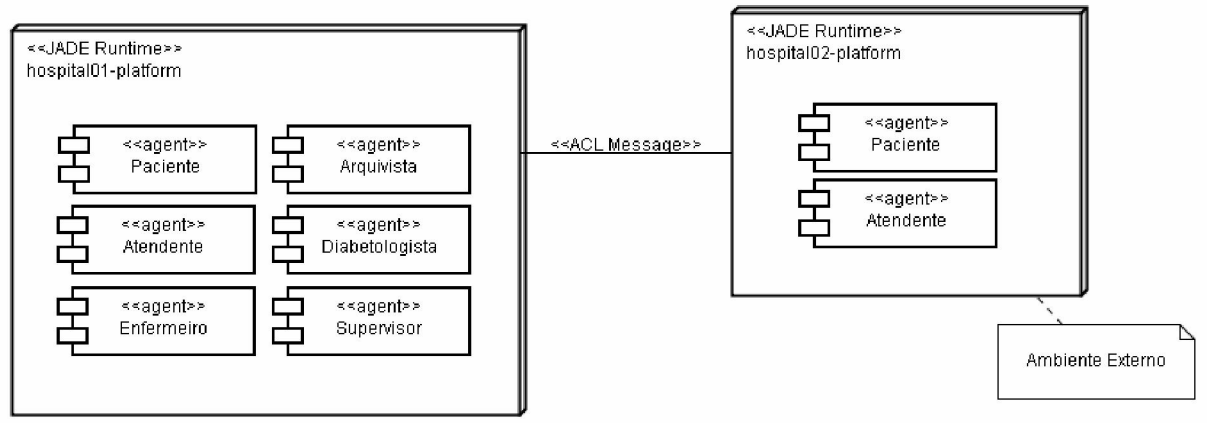

Figura 6.2 – Diagrama de instalação.

O funcionamento deste sistema, ou melhor, a interação entre os agentes que estão no ambiente interno é demonstrada pelo diagrama de sequência exibido pela Figura 6.3 e explicado a seguir.

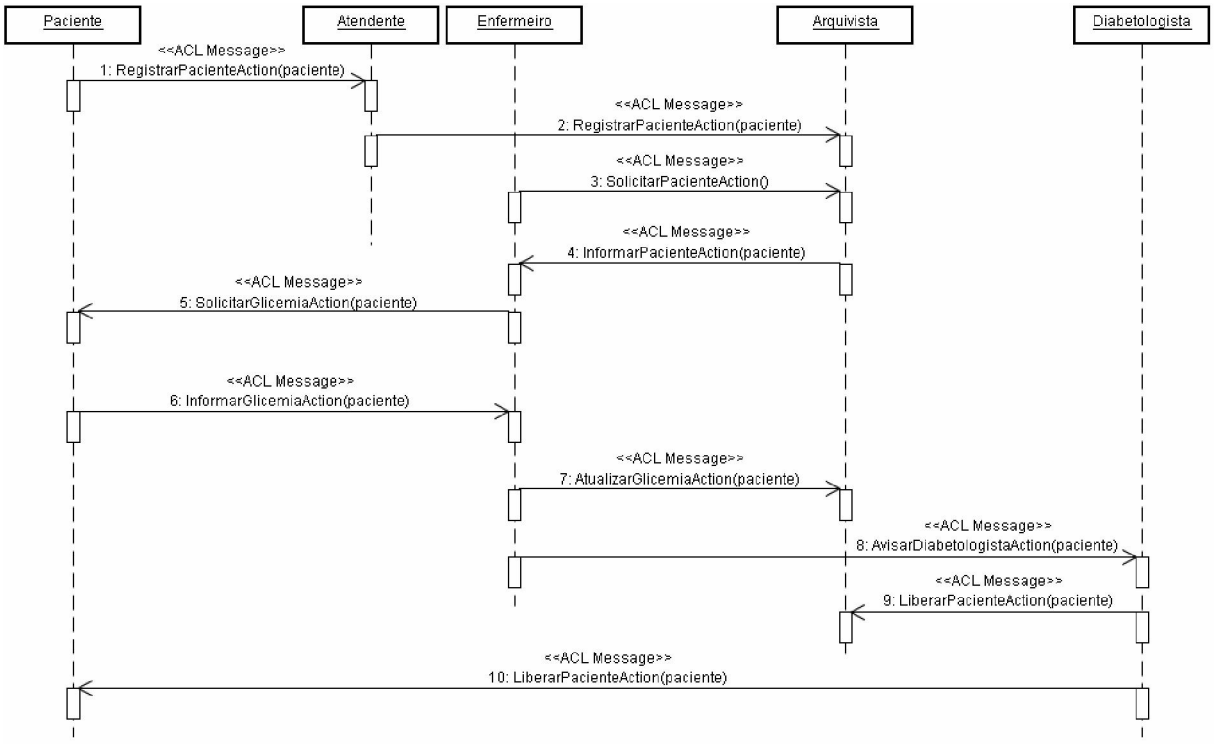

Figura 6.3 – Diagrama de sequência das interações dos agentes executados no ambiente interno.

Após o agente Paciente ser criado, ele solicita seu registro ao agente Atendente e este, por sua vez, solicita que o agente Arquivista efetive este registro persistindo os dados deste paciente numa base de dados. O agente Enfermeiro pode então solicitar ao Arquivista, um

agente Paciente registrado no sistema para que possa realizar a medição de glicemia. Após a glicemia ser medida, ou seja, ser comunicada pelo agente Paciente ao Enfermeiro o qual solicita ao Arquivista a atualização desta informação na base dados e também, de acordo com um critério de valores de glicemia, pode solicitar ao Diabetologista que analise o valor coletado. Dependendo do valor coletado o agente Diabetologista libera o Paciente, ou seja, solicita ao agente Arquivista que o remova da base de dados e avisa o fato ao Paciente fazendo com que este não faça mais parte da comunidade de agentes.

De forma similar a apresentada anteriormente, as interações entre os agentes executados no ambiente externo com aqueles do interno são demonstradas pelo diagrama de sequência exibido pela Figura 6.4. Conforme pode ser observado pela análise deste diagrama, a interações são as mesmas expostas pelo diagrama de sequência anterior, mostrado pela Figura 6.3 – com exceção das duas primeiras –.

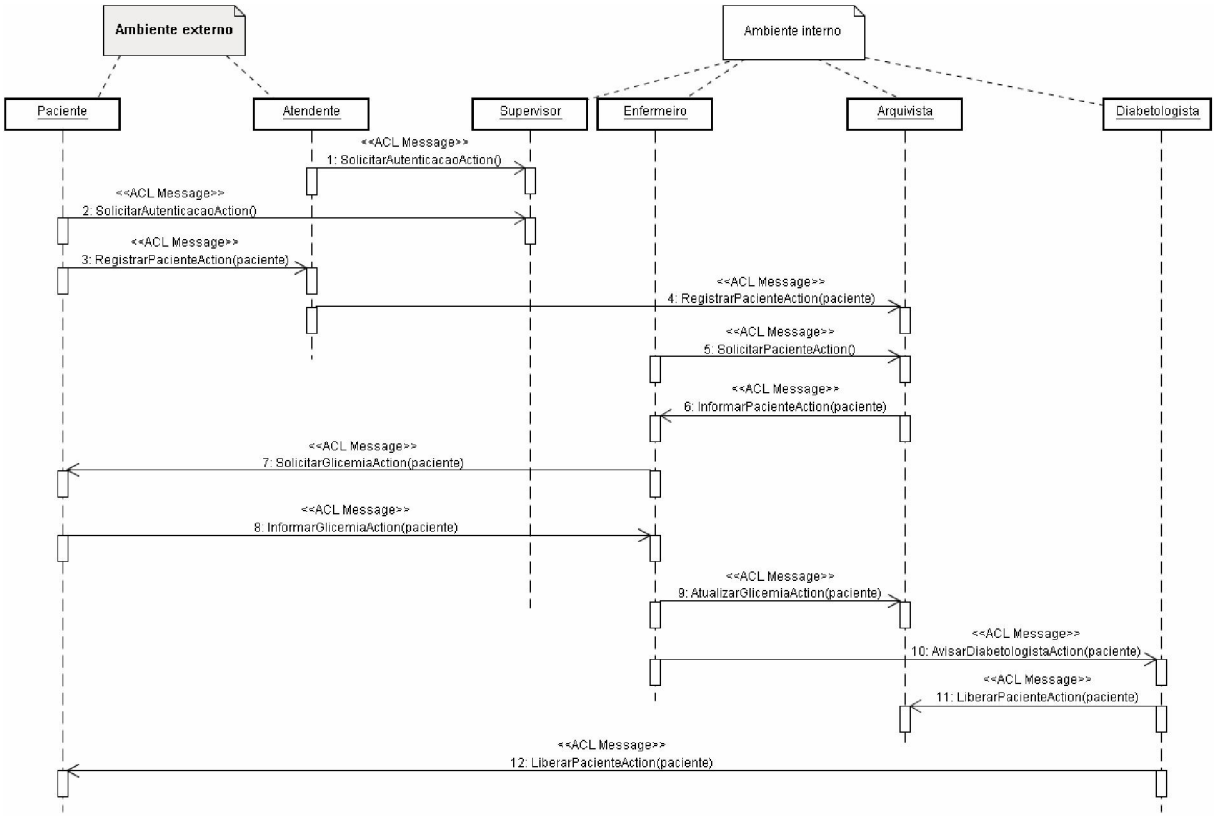

Figura 6.4 – Diagrama de sequência das interações dos agentes executados no ambiente externo.

Isto ocorre porque o ambiente externo é uma reprodução parcial do interno. Neste ambiente somente dois agentes são executados, o Paciente e o Atendente e estes dois interagem com os demais do ambiente interno para realizarem suas tarefas e por isto é preciso

que sejam autenticados. Consequentemente, a primeira ação que executam é um pedido de autorização para o agente *Supervisor* – representada no diagrama pelo envio da mensagem *SolicitarAutenticacaoAction* –. O restante das interações são as mesmas explicadas anteriormente.

Além dos agentes distribuídos em dois ambientes, para realização dos testes é necessário definir a política de controle de acesso a qual estes agentes estão submetidos. Por isso, uma política de controle de acesso é especificada de forma a definir os usuários, papéis e permissões atribuídos aos agentes do sistema em questão.

A Tabela 6.1 exibe os elementos desta política de controle de acesso RBAC. A primeira coluna contém os usuários que são vinculados aos agentes, na coluna do meio estão os papéis que cada usuário desempenha e na última coluna estão as permissões atribuídas aos papéis. Conforme discutido no capítulo quatro, no contexto dos SMAs as permissões são concedidas sobre as ações dos agentes e não acerca dos recursos que o sistema possa oferecer.

Tabela 6.1 – Política de controle de acesso RBAC.

Usuário	**Papel**	**Permissão**
arquivista	Arquivista	AtualizarGlicemia
		InformarPaciente
		LiberarPaciente
		RegistrarPaciente
		SolicitarPaciente
atendente	Atendente	RegistrarPaciente
		InfomarAcessoNegado
diabetologista	Diabetologista	AvisarDiabetologista
		LiberarPaciente
enfermeiro	Enfermeiro	AtualizarGlicemia
		AvisarDiabetologista
		InformarGlicemia
		InformarPaciente
		SolicitarGlicemia
		SolicitarPaciente
paciente	Paciente	InformarGlicemia
		LiberarPaciente
		RegistrarPaciente
		SolicitarGlicemia
		InfomarAcessoNegado
supervisor	Supervisor	InfomarAcessoNegado
		SolicitarAutenticacao

6.2. Testes e Simulações

Para a validação da implementação de controle de acesso proposta por este trabalho foram realizados os testes funcional e adversário propostos por Huang et al. (2009). Para simular as situações possíveis de requisição de acesso por um agente, quatro cenários foram definidos conforme exibido pela Tabela 6.2.

Tabela 6.2 – Cenários de teste.

Cenário	Descrição
C1	Execução de uma ação por um agente que não está autenticado
C2	Execução de uma ação por um agente cuja autenticação foi mal sucedida
C3	Execução de uma ação por um agente autenticado e que possui a permissão necessária
C4	Execução de uma ação por um agente autenticado e que não possui a permissão necessária

Os testes adversários são realizados pelas simulações dos cenários C1 e C2. O primeiro cenário – C1 – visa verificar se o controle de acesso implementado é capaz de impedir que um agente não autenticado execute uma ação. O segundo, testa a mesma situação, porém após o agente ter fracassado na tentativa de autenticação.

Já os testes funcionais são executados através das simulações dos cenários C3 e C4. O cenário C3 verifica se um agente autenticado consegue executar uma ação a qual possui a permissão necessária e por último, o cenário C4 testa se controle de acesso é capaz de impedir que um agente já autenticado execute uma ação a qual não possui permissão.

O mecanismo de controle de acesso, avalia através da atuação dos aspectos, tanto as requisições de agentes que estão num ambiente interno quanto as que chegam de agentes situados num ambiente externo. Portanto, é necessário que os quatro cenários definidos anteriormente sejam simulados a partir destes dois ambientes, pois desta maneira, além de erros no mecanismo de controle de acesso, eventuais falhas na implementação dos aspectos também podem ser detectadas.

Desse modo, um total de oito testes foram definidos. Quatro, para testar as requisições dos agentes situados no ambiente interno – um para cada cenário, identificados como Teste 1 até Teste 4 – onde é simulado a execução da ação *RegistrarPacienteAction* pelo agente Paciente. E, de maneira similar, quatro, para testar as requisições que chegam de agentes

localizados num ambiente externo – identificados como Teste 5 até Teste 8 – onde é simulado a execução da ação *RegistrarPacienteAction* pelo agente Atendente.

A seguir estão descritos os testes, as simulações executadas e os resultados obtidos.

Teste 1

O primeiro teste visa verificar se o controle de acesso implementado é capaz de impedir a execução de uma ação por um agente – localizado no ambiente interno – cuja autenticação não foi realizada. Ele é realizado através de uma simulação onde o agente Paciente, ainda não autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.3 apresenta de forma sintetizada o cenário, ambiente, agente, ação executada, elementos do modelo RBAC utilizados e o resultado esperado deste teste.

Tabela 6.3 – Descrição dos elementos utilizados para execução do Teste 1.

Teste			Teste 1
Cenário			C1
Tipo			Adversário
Requisição	**Agente**	**Ambiente**	Interno (hospital01-platform)
		Agente	Paciente (paciente01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	paciente
		Papel	Paciente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (Usuário não autenticado)

Para a realização deste teste é executada uma simulação onde o agente Paciente ainda não autenticado executa a ação *RegistrarPacienteAction*. Para isto ocorrer, o ponto de atuação do aspecto *ProactiveAuthenticationAspect*, responsável por autenticar os agentes do ambiente interno assim que são configurados, é alterado de forma a não realizar a autenticação do agente Paciente. Como consequência, o aspecto *ProactiveAccessControlAspect* aborta o envio da mensagem *RegistrarPacienteAction* do agente Paciente para o Atendente, pois o emissor não está autenticado e por isto não pode executar qualquer ação no sistema. O Quadro 6.1 apresenta a saída para o console referente à execução desta simulação, onde é possível notar que o aspecto *ProactiveAccessControlAspect* não permitiu ao agente Paciente enviar a mensagem *RegistrarPacienteAction* ao Atendente.

Quadro 6.1 – Resultado obtido pelo Teste 1.

```
...
ProactiveAccessControlAspect - (paciente01@hospital01-platform =>
atendente01@hospital01-platform):RegistrarPacienteAction
***** AGENTE NAO AUTENTICADO  *****
***** agente: [paciente01@hospital01-platform] ou
[atendente01@hospital01-platform]
***** HeimdallException: User is not authenticated
...
```

Teste 2

O segundo teste verifica se o controle de acesso implementado é capaz de impedir a execução de uma ação por um agente – localizado no ambiente interno – cuja autenticação foi mal sucedida devido a informações incorretas. Ele é realizado através de uma simulação onde o agente Paciente, não autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.4 exibe de forma resumida os elementos utilizados na simulação para realização deste teste.

Tabela 6.4 – Descrição dos elementos utilizados para execução do Teste 2.

Teste			Teste 2
Cenário			C2
Tipo			Adversário
Requisição	**Agente**	**Ambiente**	Interno (hospital01-platform)
		Agente	Paciente (paciente01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	paciente
		Papel	Paciente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (usuário não autenticado)

Nesta simulação, o agente Paciente é alterado de forma a portar uma senha inválida assim, no momento em que o aspecto *ProactiveAuthenticationAspect* realiza o procedimento para autenticação deste agente, um erro é gerado e este agente não é autenticado. Em decorrência, o aspecto *ProactiveAccessControlAspect* aborta o envio da mensagem *RegistrarPacienteAction* do agente Paciente para o Atendente, pois o emissor não está autenticado e portanto não pode executar qualquer ação no sistema. O Quadro 6.2 apresenta a saída para o console referente a execução desta simulação. É possível notar que o aspecto *ProactiveAuthenticationAspect* não realiza a autenticação do agente Paciente e em

decorrência, o aspecto *ProactiveAccessControlAspect* impede que este agente envie a mensagem ao Atendente.

Quadro 6.2 – Resultado obtido pelo Teste 2.

```
...
ProactiveAuthenticationAspect - o agente [paciente01@hospital01-
platform] nao foi autenticado com o usuario: paciente
***** AGENTE NAO AUTENTICADO  *****
***** agente: [paciente01@hospital01-platform]
...
...
ProactiveAccessControlAspect - (paciente01@hospital01-platform =>
atendente01@hospital01-platform):RegistrarPacienteAction
***** AGENTE NAO AUTENTICADO  *****
***** agente: [paciente01@hospital01-platform] ou
[atendente01@hospital01-platform]
***** HeimdallException: User is not authenticated
...
```

Teste 3

Neste teste é verificado se o controle de acesso opera em conformidade com a política de segurança, ou seja, se consegue conceder a execução de uma ação a um agente – localizado no ambiente interno – autenticado e que possui a permissão necessária. Ele é realizado através de uma simulação onde o agente Paciente, devidamente autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.5 apresenta de forma sucinta os elementos utilizados na simulação para realização deste teste.

Tabela 6.5 – Descrição dos elementos utilizados para execução do Teste 3.

Teste			Teste 3
Cenário			C3
Tipo			Funcional
Requisição	**Agente**	**Ambiente**	Interno (hospital01-platform)
		Agente	Paciente (paciente01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	paciente
		Papel	Paciente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso permitido

Nenhuma alteração no sistema foi necessária para realizar esta simulação, pois ela é o fluxo normal do sistema. O Quadro 6.3 apresenta a saída para o console referente à sua execução. É possível notar que o aspecto *ProactiveAuthenticationAspect* autentica o agente Paciente e a atuação do aspecto *ProactiveAccessControlAspect* não gera nenhum erro ao avaliar as permissões dos agentes envolvidos na troca de mensagem.

Quadro 6.3 – Resultado obtido pelo Teste 3.

```
...
ProactiveAuthenticationAspect - o agente [paciente01@hospital01-
platform] foi autenticado com o usuario: paciente
...
...
ProactiveAccessControlAspect - (paciente01@hospital01-platform =>
atendente01@hospital01-platform):RegistrarPacienteAction
...
```

Teste 4

O quarto teste visa verificar se o controle de acesso implementado é capaz de impedir a execução de uma ação por um agente – localizado no ambiente interno – autenticado e que não possui a permissão necessária. É realizado através de uma simulação onde o agente Paciente, depois de autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.6 exibe de forma condensada o cenário, ambiente, agente, ação executada, elementos do modelo RBAC utilizados na simulação para realização deste teste.

Tabela 6.6 – Descrição dos elementos utilizados para execução do Teste 4.

Teste			Teste 4
Cenário			C4
Tipo			Funcional
Requisição	**Agente**	**Ambiente**	Interno (hospital01-platform)
		Agente	Paciente (paciente01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	paciente
		Papel	Paciente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (falta de permissão)

Nesta simulação, a política de controle de acesso foi alterada de forma a não permitir que o agente Paciente possa executar a ação *RegistrarPacienteAction*, ou seja, a permissão *RegistrarPaciente* do papel Paciente foi removida. Como resultado, aspecto *ProactiveAccessControlAspect* aborta o envio da mensagem *RegistrarPacienteAction* do agente Paciente para o Atendente. O Quadro 6.4 apresenta a saída para o console referente a execução desta simulação. É possível notar que apesar de autenticado pelo aspecto *ProactiveAuthenticationAspect*, o agente Paciente é impedido pelo aspecto *ProactiveAccessControlAspect* de executar a ação *RegistrarPacienteAction*.

Quadro 6.4 – Resultado obtido pelo Teste 4.

```
...
ProactiveAuthenticationAspect - o agente [paciente01@hospital01-
platform] foi autenticado com o usuario: paciente
...
...
ProactiveAccessControlAspect - (paciente01@hospital01-platform =>
atendente01@hospital01-platform):RegistrarPacienteAction
***** ACESSO NEGADO *****
***** agente: [paciente01@hospital01-platform] ou
[atendente01@hospital01-platform]
***** sem permissao para acao: (RegistrarPacienteAction)
...
```

Teste 5

O quinto teste possui como objetivo verificar se o controle de acesso implementado é capaz de impedir que uma ação executada por um agente – localizado num ambiente externo e não autenticado – seja concluída, ou seja, a mensagem que representa esta ação não seja percebida pelos sensores do agente receptor da mensagem. Neste teste é verificada a atuação do controle de acesso, ou melhor, dos aspectos de controle de acesso em relação aos agentes situados no ambiente externo. É realizado através de uma simulação onde o agente Atendente – localizado no ambiente externo –, ainda não autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.7 apresenta de forma sintetizada o cenário, ambiente, agente, ação executada, elementos do modelo RBAC utilizados e o resultado esperado deste teste.

Tabela 6.7 – Descrição dos elementos utilizados para execução do Teste 5.

Teste			Teste 5
Cenário			C1
Tipo			Adversário
Requisição	**Agente**	**Ambiente**	Externo (hospital02-platform)
		Agente	Atendente (atendenteExterno01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	atendente
		Papel	Atendente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (usuário não autenticado)

Para esta simulação, o agente Atendente foi alterado de forma a não solicitar sua autenticação ao agente *Supervisor* – localizado no ambiente interno –. Por isso, o aspecto *ReactiveAccessControlAspect* impede que a mensagem *RegistrarPacienteAction* enviada pelo Atendente seja percebida pelos sensores do agente Arquivista, pois o emissor da mensagem não está autenticado e como decorrência, ele não pode executar nenhuma ação no sistema. O Quadro 6.5 apresenta a saída para o console referente à execução desta simulação.

Quadro 6.5 – Resultado obtido pelo Teste 5.

```
...
ReactiveAccessControlAspect - (atendenteExterno1@hospital02-platform =>
arquivista01@hospital01-platform):RegistrarPacienteAction
***** AGENTE NAO AUTENTICADO  *****
***** agente: [atendenteExterno1@hospital02-platform] ou
[arquivista01@hospital01-platform]
***** HeimdallException: User is not authenticated
...
...
ProactiveAccessControlAspect - (supervisor01@hospital01-platform =>
atendenteExterno1@hospital02-platform):InformarAcessoNegadoAction
...
```

Teste 6

Este teste verifica se o controle de acesso implementado é capaz de impedir a conclusão de uma ação executada por um agente – localizado no ambiente externo, cuja autenticação foi mal sucedida devido a informações incorretas –. É realizado através de uma simulação onde o agente Atendente, não autenticado, executa a ação *RegistrarPacienteAction*.

A Tabela 6.8 exibe de forma resumida os elementos utilizados na simulação para realização deste teste.

Tabela 6.8 – Descrição dos elementos utilizados para execução do Teste 6.

Teste			Teste 6
Cenário			C2
Tipo			Adversário
Requisição	**Agente**	**Ambiente**	Externo (hospital02-platform)
		Agente	Atendente (atendenteExterno01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	atendente
		Papel	Atendente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (usuário não autenticado)

Nesta simulação, o agente Atendente foi alterado de forma a portar uma senha inválida. Assim, no momento que o aspecto *ReactiveAuthenticationAspect* realiza o procedimento para autenticação deste agente, um erro é gerado e o agente não é autenticado. Como resultado, o aspecto *ReactiveAccessControlAspect* impede que a mensagem *RegistrarPacienteAction* enviada pelo agente Atendente seja processada pelo Arquivista. O Quadro 6.6 apresenta a saída para o console referente a execução desta simulação onde é possível verificar a atuação dos aspectos *ReactiveAuthenticationAspect* e *ReactiveAccessControlAspect*.

Quadro 6.6 – Resultado obtido pelo Teste 6.

```
...
ReactiveAuthenticationAspect - o agente [atendenteExterno1@hospital02-
platform] nao foi autenticado com o usuario: atendente
***** AGENTE NAO AUTENTICADO  *****
***** agente: [atendenteExterno1@hospital02-platform]
...
ProactiveAccessControlAspect - (supervisor01@hospital01-platform =>
atendenteExterno1@hospital02-platform):InformarAcessoNegadoAction
...
...
ReactiveAccessControlAspect - (atendenteExterno1@hospital02-platform =>
arquivista01@hospital01-platform):RegistrarPacienteAction
***** AGENTE NAO AUTENTICADO  *****
***** agente: [atendenteExterno1@hospital02-platform] ou
[arquivista01@hospital01-platform]
***** HeimdallException: User is not authenticated
...
ProactiveAccessControlAspect - (supervisor01@hospital01-platform =>
atendenteExterno1@hospital02-platform):InformarAcessoNegadoAction
...
```

Teste 7

Neste teste é verificado se o controle de acesso opera em conformidade com a política de segurança, ou seja, se permite que uma ação executada por um agente – localizado no ambiente externo, autenticado e que possui a permissão necessária – seja concluída. Ele é realizado através de uma simulação onde o agente Atendente, devidamente autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.9 apresenta de forma sucinta os elementos utilizados na simulação para realização deste teste.

Tabela 6.9 – Descrição dos elementos utilizados para execução do Teste 7.

Teste			Teste 7
Cenário			C3
Tipo			Funcional
Requisição	**Agente**	**Ambiente**	Externo (hospital02-platform)
		Agente	Atendente (atendenteExterno01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	atendente
		Papel	Atendente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso permitido

Nenhuma alteração foi realizada no sistema para esta simulação, pois ela é o fluxo normal do sistema. O Quadro 6.7 apresenta a saída para o console referente à execução desta simulação. É possível notar que o aspecto *ReactiveAuthenticationAspect* autentica o agente Atendente e em seguida o aspecto *ReactiveAccessControlAspect* atua sobre o agente receptor da mensagem – agente Arquivista – e não gera nenhum erro, fato que indica o sucesso na execução da ação.

Quadro 6.7 – Resultado obtido pelo Teste 7.

```
...
ReactiveAuthenticationAspect - o agente [atendenteExterno1@hospital02-
platform] foi autenticado com o usuario: atendente
...
...
ReactiveAccessControlAspect - (atendenteExterno1@hospital02-platform =>
arquivista01@hospital01-platform):RegistrarPacienteAction
...
```

Teste 8

O último teste visa verificar se o controle de acesso implementado é capaz de impedir a conclusão de uma ação executada por um agente – localizado no ambiente externo, autenticado e que não possui a permissão necessária –. É realizado através de uma simulação onde o agente Atendente depois de autenticado, executa a ação *RegistrarPacienteAction*. A Tabela 6.10 exibe de forma condensada o cenário, ambiente, agente, ação executada, elementos do modelo RBAC utilizados na simulação para realização deste teste.

Tabela 6.10 – Descrição dos elementos utilizados para execução do Teste 8.

Teste			Teste 8
Cenário			C4
Tipo			Funcional
Requisição	**Agente**	**Ambiente**	Externo (hospital02-platform)
		Agente	Atendente (atendenteExterno01)
		Ação	RegistrarPacienteAction
	RBAC	**Usuário**	atendente
		Papel	Atendente
		Permissão	RegistrarPaciente
Resultado esperado			Acesso negado (falta de permissão)

Para realizar esta simulação, a política de controle de acesso foi alterada de forma a não permitir que o Atendente possa executar a ação *RegistrarPacienteAction,* ou seja, a permissão *RegistrarPaciente* do papel Atendente foi removida. Em decorrência, o aspecto *ReactiveAccessControlAspect* impede que a mensagem *RegistrarPacienteAction* enviada pelo agente Atendente seja processada pelo Arquivista. O Quadro 6.8 apresenta a saída para o console referente à execução dessa simulação. É possível verificar que apesar do aspecto *ReactiveAuthenticationAspect* autenticar o agente Atendente, o aspecto *ReactiveAccessControlAspect* impede que a ação seja concluída ou melhor, impede que a mensagem seja percebida pelo agente Arquivista devido a falta de permissão do emissor.

Quadro 6.8 – Resultado obtido pelo Teste 8.

```
...
ReactiveAuthenticationAspect - o agente [atendenteExterno1@hospital02-
platform] foi autenticado com o usuario: atendente
...
...
ReactiveAccessControlAspect - (atendenteExterno1@hospital02-platform
=> arquivista01@hospital01-platform):RegistrarPacienteAction
***** ACESSO NEGADO *****
***** agente: [atendenteExterno1@hospital02-platform] ou
[arquivista01@hospital01-platform]
***** sem permissao para acao: (RegistrarPacienteAction)
...
ProactiveAccessControlAspect - (supervisor01@hospital01-platform =>
atendenteExterno1@hospital02-platform):InformarAcessoNegadoAction
...
```

6.3. Análise dos Resultados

Através da análise dos resultados obtidos por cada simulação, observou-se que as respostas produzidas pelo controle de acesso implementado foram condizentes com os resultados esperados. Isto evidencia que, dentro do conjunto de cenários testados, a implementação do controle de acesso na arquitetura dos agentes está operando de forma correta.

Nota-se também, devido a utilização dos aspectos, dois benefícios agregados a implementação proposta: a separação do interesse de controle de acesso e uma provável economia do esforço computacional – realizado pelos agentes – que é destinado ao processamento de mensagens inválidas[34].

A separação do interesse de controle de acesso é explícita pelos próprios aspectos que através da atuação nos sensores e atuadores dos agentes faz com que o controle de acesso esteja presente sem que eles lidem diretamente com este interesse. Conforme pode ser inspecionado nos códigos-fonte dos aspectos – Apêndice A –.

A economia do esforço computacional que é destinado ao processamento de mensagens inválidas possui evidência de ser alcançada devido a atuação dos aspectos de controle de acesso. O *ProactiveAccessControlAspect* através de sua influência nos atuadores

[34] Entende-se por mensagem inválida qualquer mensagem onde um dos envolvidos – agente emissor ou receptor – não possua permissão para lidar com a ação em questão.

dos agentes aborta o envio de mensagens inválidas. Já o *ReactiveAccessControlAspect* faz com que uma mensagem inválida enviada por um agente situado num ambiente externo não seja percebida pelo sensor do agente receptor. Desse modo é poupado o esforço computacional do agente receptor para processar uma mensagem cuja permissão de um dos envolvidos é inexistente; fato que torna a troca de mensagem ilegítima.

7. Conclusão

A implementação do controle de acesso na arquitetura do agente é uma maneira de mitigar os riscos de segurança relativos a incidentes causados pelo acesso não autorizado.

Dentre os modelos de controle de acesso existentes, o RBAC tem sido aplicado em SMA de forma a integrar harmonicamente o controle de acesso baseado em papéis com a coordenação e organização dos agentes modelados segundo o conceito de papéis.

Para utilização deste modelo num SMA três adaptações são necessárias: a primeira é que neste contexto, a entidade Usuário deve ser o próprio agente do sistema, a segunda é a atribuição dos papéis diretamente a um agente, de acordo com os papéis que este desempenha e, por último, a concessão de permissões deve ser feita sobre as ações e percepções dos agentes e não sobre dos recursos do sistema (VIROLI et al., 2007).

Apesar dessas adaptações não serem complexas, a implementação do controle de acesso RBAC na arquitetura do agente pode gerar – devido a sua natureza transversal – sintomas indesejáveis no sistema como o emaranhamento e espalhamento de código que juntos, impactam negativamente na compreensão, reuso e evolução de projetos orientados a agentes. Estes dois sintomas decorrentes da má modularização são evitados com a utilização da POA, pois este paradigma promove a separação de interesses transversais em módulos separados, chamados aspectos.

Neste contexto, este trabalho contribui com uma proposta para controle de acesso RBAC na arquitetura do agente que promove além do controle de acesso dos agentes, a separação avançada desse interesse. Tendo como fundamento esta abordagem em que a separação de interesses é inerente ao controle de acesso, a arquitetura estrutura estes dois conceitos – separação de interesses e controle de acesso – em camadas, onde cada uma representa um nível de abstração.

Na camada de controle de acesso a autenticação e controle de acesso RBAC são modularizados por um mecanismo – intitulado *Heimdall* – criado para este fim, enquanto a utilização destas funcionalidades pelos agentes é realizada por meio de aspectos – elementos da camada de aspectos de segurança – que viabilizam a separação avançada deste interesse. É através da utilização dos aspectos que as funcionalidades de autenticação e controle de acesso são implementadas e depois integradas com os agentes.

Provavelmente a maior contribuição deste trabalho está na exposição da forma de como os aspectos podem promover o controle de acesso nos agentes. A atuação dos aspectos

diretamente nos sensores e atuadores dos agentes mostrou-se eficiente, pois possibilitou além do controle de acesso, a evidência de economia do esforço computacional que seria destinado ao processamento de mensagens inválidas.

Grande parte desta economia é resultado da atuação proativa dos aspectos *ProactiveAuthenticationAspect* e *ProactiveAccessControlAspect* que devido a forma de atuação acabam reduzindo o fluxo de mensagem entre agentes.

O trabalho apresentado nesta dissertação pode ser continuado através de um estudo quantitativo para avaliar o quanto a estratégia de utilizar aspectos nos sensores e atuadores dos agentes para o controle de acesso influencia no fluxo de mensagens entre agentes de maneira a impactar no desempenho geral da comunidade de agentes.

Como trabalhos futuros a presente dissertação sugere um estudo da utilização da POA para garantir a integridade de mensagens trocadas entre os agentes. Esta integridade pode ser implementada através de aspectos responsáveis por encriptar uma mensagem assim que ela é enviada por um atuador do agente emissor e outros que a desencriptam antes que seja percebida pelo sensor do receptor.

Outra proposta visa realizar uma comparação de desempenho entre a implementação do controle de acesso RBAC em agentes com a POA em paridade com outras implementações que não utilizam esta abordagem.

Finalmente, a última proposta de trabalho é uma análise aprofundada do desempenho geral do sistema obtido pela utilização do agente *Supervisor* em contraposição a outras estratégias para autenticação dos agentes.

Referências

AKSIT, Mehmet; WAKITA, Ken; BOSCH, Jan; BERGMANS, Lodewijk; YONEZAWA, Akinori. **Abstracting Object Interactions Using Composition Filters**. In: Proceedings of the Workshop on Object-Based Distributed Programming, ECOOP'93, 1993

ASPECTCPP. **Página oficial do projeto AspectC++**. Disponível em: <http://www.aspectc.org>. Acessado em: 21/06/2009.

ASPECTJ. **Página oficial do projeto AspectJ**. Disponível em: <http://www.eclipse.org/aspect>. Acessado em: 22/06/2009.

ASPECTLUA. **Página oficial do projeto AspectLua**. Disponível em: <http://luaforge.net/projects/aspectlua>. Acessado em: 21/06/2009.

ASPECTNET. **Página oficial do projeto Aspect.NET**. Disponível em: <http://www.facultyresourcecenter.com/curriculum/pfv.aspx?ID=6801>. Acessado em: 21/06/2009.

BELLIFEMINE, Fabio; CAIRE, Giovanni; TRUCCO, Tiziana; RIMASSA, Giovanni. **JADE Programmer's Guide**, 2003.

BELLIFEMINE, Fabio; CAIRE, Giovanni; GREENWOOD, Dominie. **Developing Multi-Agent Systems with JADE**, 2007.

BODKIN, Ron. **Enterprise Security Aspects**. In: International Conference on Aspect-Oriented Software Development (AOSD), 2004.

BUSCHMAN, Frank; MEUNIER, Regine; ROHNERT, Hans; SOMMERLAD, Peter; STAL, Michael. **Pattern-Oriented Software Architecture Volume 1: A System of Patterns**. John Wiley & Sons, 1996.

CABRI, Giacomo; LEONARDI, Letizia; ZAMBONELLI, Franco. **Modeling Role-based Interactions for Agents**. In: Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies, 2002.

CABRI, Giacomo; FERRARI, Luca; ZAMBONELLI, Franco. **Role-based Approaches for Engineering Interactions in Large-scale Multi-Agent Systems**. Software engineering for multi-agent systems II: research issues and practical applications, vol. 2940, 2003.

CABRI, Giacomo; FERRARI, Luca; LEONARDI, Letizia. **Embedding JAAS in Agent Roles to Apply Local Security Policies**. Third International Conference on Principles and Practice of Programming Java (PPPJ), 2004.

CABRI, Giacomo; FERRARI, Luca; LEONARDI, Letizia. **Applying Security Policies Through Agent Roles: a JAAS Based Approach**. Science of Computer Programming, Elsevier, Amsterdam-NL, 2005.

CAIRE, Giovanni. **JADE Tutorial: JADE Programming for Beginners**. 2003

CAMILO, José Augusto Peçanha. **FTQL: Um Modelo Para o Estudo de Organizações, do Ponto de Vista de Controle de Acesso, Baseado em Funções, Tarefas e Qualificações**. São José dos Campos, 2001. Dissertação (Mestrado) - Instituto Tecnológico de Aeronáutica. Divisão de pós-graduação.

CONCEIÇÃO, Heraldo Vieira D.; SILVA, Roberto de Oliveira. **Aplicabilidade do Modelo RBAC no Controle de Acesso para a Rede Sem Fio do Senado Federal**, Distrito Federal, 2006. Monografia de Especialização – Universidade de Brasília, Faculdade de Tecnologia. Departamento de Engenharia Elétrica.

CSI. **Computer Crime & Security Survey, The latest results from the longest-running project of its kind**, 2008. Disponível em: <http://i.cmpnet.com/v2.gocsi.com/pdf/CSIsurvey2008.pdf>. Acessado em: 17/07/2009.

DIJKSTRA, E. **A Discipline of Programming**. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

ECLIPSE. **Página oficial da fundação eclipse.** Disponível em: <http://www.eclipse.org>. Acessado em: 22/06/2009.

FERBER, Jacques. **Multi-agent Systems: An Introduction to Distributed Artificial Intelligence**, Addison-Wesley, 1999.

FIGUEIREDO, Eduardo Magno Lages. **Uma abordagem quantitativa para desenvolvimento de software orientado a aspectos**. Rio de Janeiro, 2006. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática.

FININ, Tim; FRITZSON, Richard; MCKAY, Don; MCENTIRE, Robin. **KQML as an agent communication language**. In: Proceedings of the third international conference on Information and knowledge management (CIKM '94), New York, NY, USA, 1994.

FIPA. **Página oficial da organização FIPA**. Disponível em: <http://www.fipa.org>. Acessado em: 22/06/2009.

FIPA. **FIPA ACL Message Structure Specification**, 2002. Disponível em: <http://www.fipa.org/specs/fipa00061>. Acessado em: 22/06/2009.

FRANKLIN, Stan; GRAESSER, Art. **Is it an agent, or just a program?: A taxonomy for autonomous agents**. Proceedings of Third International Workshop on Agent Theories, Architectures, and Languages (ATAL'96), 1996.

GARCIA, Alessandro Fabrício. **Objetos e agentes: uma abordagem orientada a aspectos**. Rio de Janeiro, 2004. Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

GARCIA, Alessandro; CHAVEZ, Christina; CHOREN, Ricardo. **Enhancing Agent-Oriented Models with Aspects**. In: Proceedings of the 5th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS'06), 2006.

GENESERETH, Michael. R. **Knowledge Interchange Format**. Proceedings of the Second International Conference on the Principles of Knowledge Representation and Reasoning, 1991.

GIRARDI, Rosario; FERREIRA, Steferson Lima Costa. **Arquiteturas de software baseadas em agentes**. Universidade Federal do Maranhão, Departamento de Informática, 2002.

GONG, Li; ELLISON, Gary; DAGEFORDE, Mary. **Inside Java 2 Platform Security**. Second Edition, Prentice Hall, 2003.

HARRISON, William; OSSHER, Harold. **Subject-Oriented Programming - A Critique of Pure Objects**. In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), 1993.

HUANG, Chao; SUN, Jianling; WANG, Xinyu; SI, Yuanjie. **Selective Regression Test for Access Control System Employing RBAC**. In: Proceedings of the 3rd International Conference and Workshops on Advances in Information Security and Assurance (ISA '09), 2009.

HUHNS, Michael N.; SINGH, Munindar P. **Readings in Agents**. Morgan Kaufmann, CA, EUA, 1 edição, 1997.

HUHNS, Michael N.; STEPHENS, Larry M. **Multiagent systems and societies of agents. Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. MIT Press, Cambridge, MA, USA, 1999.

HÜRSCH, Walter; LOPES, Cristina Videira. **Separation of Concerns**. College of Computer Science, Northeastern University, Boston, 1995.

HWANG, JeeHyun; MARTIN, Evan; XIE, Tao; HU, Vincent C. **Policy-Based Testing**. Department of Computer Science, North Carolina State University, Raleigh, 2008.

ITSEC. **Information Technology Security Evaluation Criteria (ITSEC)**. Luxembourg: Office for Official Publications of the European Communities, 1991.

INCITS. **InterNational Committee for Information Technology**. Página oficial do comitê. Disponível em: <http://www.incits.org>. Acessado em: 22/06/2009.

JAAS. **Java Authentication and Authorization Service**. Disponível em: <http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html>. Acessado em: 22/06/2009.

JANSEN, Wayne; KARYGIANNIS, Tom. **Mobile agent security**. NIST Special Publication 800-19, National Institute of Standards and Technology, Computer Security Division, 2000.

JAVA. **Página oficial da linguagem de programação Java**. Disponível em: <http://java.sun.com>. Acessado em: 22/06/2009.

JENNINGS, Nicholas R.; SYCARA Katia.; WOOLDRIDGE Michael. **A Roadmap of Agent Research and Development**. In: Autonomous Agents and Multi-Agent Systems Journal. Kluwer Academic Publishers, Boston, Volume 1, Issue 1, p. 7-38, 1998.

JENNINGS, Nicholas R. **Agent-Oriented Software Engineering**. In: Proceedings of the 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence, 1999.

KENDALL, E.; KRISHNA, P.; PATHAK, C.; SURESH, C. **A Framework for Agent Systems**. Implementing Application Frameworks – Object-Oriented Frameworks at Work, John Wiley & Sons, 1999.

KENDALL, Elizabeth A. **Role model designs and implementations with aspect-oriented programming**. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. Denver, Colorado, United States, 1999.

KICZALES, Gregor; IRWIN, John; LAMPING, John; LOINGTIER, Jean-Marc; LOPES, Cristina Videira; MAEDA, Chris; MENDHEKAR, Anurag. **Aspect-Oriented Programming**. In: ECOOP 97, 1997.

KICZALES, Gregor; HILSDALE, Eric; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William G. **An Overview of AspectJ**. In: 15th European Conference on Object-Oriented Programming (ECOOP 01), 2001.

KNAPIK, Michael; JOHNSON, Jay. **Developing Intelligent Agents for Distributed Systems: Exploring Architectures, Techniques, and Applications**, Osborne/McGraw-Hill, 1997.

LADDAD, Ramnivas. **AspectJ in Action, Pratical Aspect-Oriented Programming**. 1. ed., Greenwich: Manning, 2003.

LEHTINEN, Rick; RUSSEL, Deborah; GANGEMI, G.T. **Computer Security Basics**. Second Edition, O'Reilly Media, 2006.

LIEBERHERR, Karl J. **Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns**. PWS Publishing Company, 1996.

LIEBERHERR, Karl J.; ORLEANS, Doug; OVLINGER, Johan. **Aspect-oriented programming with adaptive methods**. Communications of the ACM, Volume 44, Issue 10, 2001.

LOBATO, Cidiane Aracaty. **Um Framework Orientado a Aspectos para Mobilidade de Agentes de Software**. Rio de Janeiro, 2005. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática.

LOPES, Cristina Isabel Videira. **D: A Language Framework For Distributed Programming**. Boston, 1997. Tese (Doutorado) - Northeastern University, College of Computer Science.

MARTIN, Evan. **Automated test generation for access control policies**. In: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06), 2006.

MCLEAN, John. **Security models**. In: Encyclopedia of Software Engineering (ed. John Marciniak), Wiley Press, 1994.

MOLESINI, Ambra; DENTI, Enrico; OMICINI, Andrea. **RBAC-MAS & SODA: Experimenting RBAC in AOSE**. In: 9th International Workshop Engineering Societies in the Agents World (ESAW'08), 2008.

MOURATIDIS, Haralambos; GIORGINI, Paolo; SCHUMACHER, Markus. **Security Patterns for Agent Systems**. Proceedings of the Eight European Conference on Pattern Languages of Programs (EuroPLoP), Irsee, 2003.

MOURATIDIS, Haralambos; GIORGINI, Paolo; WEISS, Michael. **Integrating Patterns and Agent-Oriented Methodologies to Provide Better Solutions for the Development of Secure Agent-Based Systems**. In: Proceedings of the Workshop on Expressiveness of Pattern Languages, 2003

MOURATIDIS, Haralambos et al. **A secure architectural description language for agent systems**, In: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems (AAMAS '05), 2005.

MYERS, Glenford J. **The art of software testing**. 6ª Edição, Hoboken, New Jersey, John Wiley & Sons, Inc., 2004.

NAVARRO, G.; ORTEGA-RUIZ, J. A.; AMETLLER, J.; ROBLES, S. **Distributed Authorization Framework for Mobile Agents**. Lecture Notes in Computer Science, Mobility Aware Technologies and Applications, Vol. 3744, 2005.

NWANA, Hyacinth S. **Software agents: An overview, Knowledge Engineering Review**. Cambridge University Press, Vol. 11, No. 3, 1996.

O'GORMAN, Lawrence. **Comparing Passwords, Tokens, and Biometrics for User Authentication**. Proceedings of the IEEE, Vol. 91, No. 12, 2003.

OMG, Object Management Group. **Agent Technology Green Paper**, 2000. Disponível em: <http://www.objs.com/agent/agents_Green_Paper_v100.doc>. Acessado em: 15/09/2009.

OMG, Object Management Group. **Mobile Agent Facility Specification**, 2000. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/00-01-02.pdf>. Acessado em: 15/09/2009.

ORLEANS, Doug. **Programming language support for separation of concerns**. 2005. Tese (doutorado) - Northeastern University, Faculty of the Graduate School. College of Computer Science.

PARNAS, David Longe. **On the Criteria to Be Used in Decomposing Systems into Modules**. Carnegie-Mellon University, Department of Computer Science. Communications of the ACM, Vol. 15, No. 12, 1972.

PARTSAKOULAKIS, Ioannis; VOUROS, George. **Importance and Properties of Roles in MAS Organization: A review of methodologies and systems**. In: First International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS 02), 2002.

PARTSAKOULAKIS, Ioannis; VOUROS, George. **ROLES IN MAS: Managing the Complexity of Tasks and Environments**. An Application Science for Multi-Agent Systems: Multiagent Systems, Artificial Societies, And Simulated Organizations, Volume 10, Spring US, 2004.

PIVETA, Eduardo. **Um Modelo de Suporte a Programação Orientada a Aspectos**. Santa Catarina, 2001. Dissertação (Mestrado) - Universidade Federal de Santa Catarina.

PRESSMAN, Roger S. **Engenharia de software**. Tradução de Rosângela Delloso Penteado, revisão técnica de Fernão Stella R. Germano, José Carlos Maldonado, Paulo César Masiero. 6ª Edição, São Paulo, Editora McGraw Hill, 2006.

QUILLINAN, Thomas B.; WARNIER, Martijn; OEY, Michel; TIMMER, Reinier; BRAZIER, Frances. **Enforcing Security in the AgentScape Middleware**. In: Proceedings of the 1st International Workshop on Middleware Security (MidSec), 2008.

RIBEIRO, Paula Clark. **Modelagem e Implementação OO de Sistemas Multi-Agentes**. Rio de Janeiro, 2001. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática.

RUSSELL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. Prentice Hall, 2003.

SANDHU, Ravi; SAMARATI, Pierangela. **Access control: Principles and practice**. IEEE Communications Magazine, p. 40, 1994.

SANDHU, Ravi; SAMARATI, Pierangela. **Authentication, Access Control, and Audit**. ACM Computing Surveys, Vol. 28, No. 1, 1996.

SANDHU, Ravi S. **Role-Based Access Control**. George Manson University, Laboratory for Information Security Technology, ISSE Department, 1997.

SANDHU, Ravi; FERRAIOLO, David; KUHN, Richard. **The NIST Model for Role-Based Access Control: Towards A Unified Standard**. Information Technology Laboratory, National Institute of Standards and Technology (NIST), 2000.

SANT'ANNA, Cláudio Nogueira. **Manutenibilidade e reusabilidade de software orientado a aspectos: um framework de avaliação**. Rio de Janeiro, 2004. Dissertação (Mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

SEARLE, John R. **Speech Acts: An essay in the philosophy of language**. Cambridge, MA, Cambridge University Press, 1969.

SHAH, Viren; HILL, Frank. **An Aspect-Oriented Security Framework: Lessons Learned**. In: Proceedings of AOSDSEC'04 (AOSD Technology for Application-Level Security). Workshop of the Aspect Oriented Software Development Conference, Lancaster, UK, 2004.

SILVA, Viviane; GARCIA, Alessandro; BRANDÃO, Anarosa; CHAVEZ, Christina, LUCENA, Carlos; ALENCAR, Paulo. **Taming Agents and Objects in Software Engineering**. In: Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'2003), 2003.

SUNDSTED, Todd. **An introduction to agents**. Java World. 1998. Disponível em: <http://www.javaworld.com/jw-06-1998/jw-06-howto.html>. Acessado em: 15/09/2009

SYMEONIDIS, Andreas; MITKA, Pericles. **Agent Intelligence Through Data Mining.** An Application Science for Multi-Agent Systems: Multiagent Systems, Artificial Societies, And Simulated Organizations, Volume 14, Spring US, 2006.

TARR, Peri; OSSHER, Harold; HARRISON, William; SUTTON, Stanley M. Jr. **N Degrees of Separation: Multi-Dimensional Separation of Concerns**. In: Proceedings of the 21st International Conference on Software Engineering, 1999.

VILLAR, Melissa Vieira Fernandes. **Modelo de autenticação e autorização baseado em certificados de atributos para controle de acesso de aplicações em ambiente distribuído utilizando redes de Petri coloridas**. Fortaleza, 2007. Dissertação (Mestrado) – Universidade Federal do Ceará, Programa de Pós-graduação em Engenharia de Teleinformática.

VIROLI, Mirko; OMICINI, Andrea; RICCI, Alessandro. **Infrastructure for RBAC-MAS: An Approach Based on Agent Coordination Contexts**. Applied Artificial Intelligence, Vol. 21, 2007.

WAGNER, Gerd. **Multi-Level Security in Multiagent Systems**. Cooperative Information Agents, LNAI 1202, 1997.

WANGHAM, Michelle Silva. **Esquema de segurança para agentes móveis em sistemas abertos**. Florianópolis, 2004. Tese (Doutorado) - Universidade Federal de Santa Catarina, Programa de Pós-graduação em Engenharia elétrica.

WOOLDRIDGE, Michael; JENNINGS, Nicholas R.; KINNY, David. **A Methodology for Agent-Oriented Analysis and Design**. In: Proceedings of the third annual conference on Autonomous Agents, 1999.

WOOLDRIDGE, Michael. **An introduction to multiagent systems**. John Wiley & Sons, 2002.

YAMAZAKI, Wataru; HIRAISHI, Hironori; MIZOGUCHI, Fumio. **Designing an Agent-Based RBAC System for Dynamic Security Policy**. In: Proceedings of the 13th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '04), 2004.

YAO, Walt Teh-Ming. **Trust Management for Widely Distributed Systems**. 2003. Teste (Doutorado) - University of Cambridge, Jesus College.

XIAO, Liang et al. **An Adaptive Security Model for Multi-agent Systems and Application to a Clinical Trials Environment**. In: 31st Annual International Computer Software and Applications Conference (COMPSAC), 2007.

ZAMBONELLI, Franco; JENNINGS, Nicholas R.; WOOLDRIDGE, Michael. **Organisational Abstractions for the Analysis and Design of Multi-Agent Systems**. Workshop on Agent-Oriented Software Engineering, 2000.

Glossário

Agente: sistema computacional situado em algum ambiente e que é capaz de ações autônomas neste ambiente a fim de alcanças seus objetivos.

Ambiente: tudo que possa interagir com um agente, e que não faça parte da sua estrutura interna.

Ambiente Externo: ambiente cujo controle de acesso não está implementado.

Ambiente Interno: ambiente cujo controle de acesso está implementado.

AspectJ: linguagem de programação pertencente ao paradigma de orientação à aspectos.

Aspecto: abstração utilizada na orientação a aspectos para modularização de interesses transversais.

Atuador: estrutura que define o momento de atuação de um aspecto num determinado ponto de junção.

Autenticação: processo onde uma entidade fornece uma prova onde é constatado se ela realmente é quem diz ser.

Autorização: ato de limitar as atividades de usuários já autenticados pelo sistema.

Biblioteca: conjunto de funções específicas para a utilização por programas que não querem envolver-se em detalhes de implementação destas funções, mas apenas querem utilizar seus serviços.

Camada: abstração que representa uma divisão lógica de componentes ou aspectos.

Cenário: meio natural para escrever especificações parciais, os cenários capturam uma sequência de interações que representam uma transição no sistema, ou uma função no sistema.

Combinador de Aspectos: elemento responsável pelo cruzamento de um ou mais aspectos com os componentes descritos na linguagem de componentes de forma a gerar o programa final.

Controle de Acesso: ato de limitar as atividades de usuários já autenticados pelo sistema.

Decomposição Funcional: divisão e modularização de um processo ou sistema, de acordo com suas funcionalidades, em partes mais compreensíveis e menores.

Emaranhamento de Código: sintoma causando quando um módulo é implementado tendo que lidar com múltiplos interesses.

Espalhamento de Código: sintoma causado quando um procedimento é implementado em múltiplos módulos.

Interesse: consideração específica que deve ser atendida para que seja possível satisfazer o objetivo geral do sistema.

Interesse Principal: especifica o que é realmente importante para a aplicação, captura a essência, o relevante para o domínio da aplicação.

Interesse Transversal: qualquer forma computacional utilizada para controlar ou otimizar os interesses principais.

Middleware: programa que faz mediação entre outros programas.

Modularização: processo de criar módulos os quais escondem suas implementações uns dos outros.

Monitor de Referência: elemento que controla as tentativas de acesso realizadas por um usuário aos recursos de um sistema.

Orientação a Aspectos: paradigma de programação de sistemas que permite a separação de interesses transversais por meio de abstrações chamadas aspecto.

Orientação a Objetos: paradigma de análise, projeto e programação de sistemas baseado na composição e interação entre diversos componentes de *software* chamados de objetos.

Orientação a Agentes: paradigma que tem como elemento central o agente.

Páginas Amarelas: mecanismo que permite a localização de um agente que oferece um determinado serviço.

Papel: trabalho funcional ou título de um trabalho que confere determinada autoridade e responsabilidade ao usuário que o desempenha.

Permissão: aprovação de um modo particular de acesso a um ou mais objetos do sistema.

Plataforma: local de instalação e execução de um agente.

Política de Controle de Acesso: conjunto de papéis e permissões atribuídos a um usuário.

Ponto de Atuação: estrutura utilizada para sinalizar o local e momento de atuação do aspecto.

Ponto de Junção: pontos definidos na execução de um programa, por exemplo: chamadas a métodos, acessos a membros e instanciação de objetos.

Requisito: objetivos ou restrições estabelecidas por clientes e usuários que definem as características, atributos, habilidades ou qualidade que um sistema (ou qualquer um de seus módulos e sub-rotinas), deve prover para ser útil a seus usuários.

Role-Based Access Control: modelo de controle de acesso onde os direitos de acesso são atribuídos aos papéis ao invés de serem atribuídos a cada usuário.

Separação de Interesses: princípio que guia a divisão em partes do sistema onde a situação ideal seria que a parte do programa dedicada a satisfazer a um determinado interesse estivesse concentrada em uma única localidade física, separada de outros interesses, para que o interesse possa ser estudado e compreendido com facilidade.

Separação de Responsabilidade: divisão de tarefas e de permissões associadas entre diferentes papéis de forma a prevenir que um único usuário acumule muita autoridade.

Sistema Multiagente: composição de um conjunto de diferentes tipos de agentes e objetos que são inseridos em um ou mais ambientes.

Teste Adversário de Controle de Acesso: simulações de acessos ilegais realizadas com a finalidade de determinar se o controle de acesso implementado contém alguma vulnerabilidade.

Teste de Controle de Acesso: conjunto de atividades realizadas para assegurar a conformidade entre a política de segurança e sua implementação.

Teste Funcional de Controle de Acesso: simulações de acessos realizadas com a finalidade de determinar se a implementação do controle de acesso está funcionando em conformidade com a política de segurança especificada.

Teste de Verificação: conjunto de atividades realizadas para garantir que o sistema testado implementa corretamente uma função específica.

Thread: divisão de um processo em duas ou mais tarefas que podem ser executadas em paralelo.

Usuário: representação de um ser humano, agente autônomo, processo ou máquina.

Vulnerabilidade: brecha em um sistema computacional, também conhecida como *bug*.

Virtual Private Network: rede de comunicações privada normalmente utilizada por uma empresa ou um conjunto de empresas ou instituições. Utiliza como infraestrutura uma rede de comunicação pública, como por exemplo, a *Internet*.

Apêndice A -
Códigos-fonte dos Aspectos

SecurityAspect

```
package br.mestrado.hosp.aspects.security;

import org.heimdall.Heimdall;

import br.mestrado.hosp.agents.Supervisor;

/**
 * Aspecto responsável pela segurança (base)
 * */
public abstract aspect SecurityAspect {

	//
	// Pega o agente supervisor da aplicação
	//
	protected Supervisor supervisor = null;

	//pointcut
	pointcut supervisorSetup() :
		execution(void
	br.mestrado.hosp.agents.Supervisor.setup());//PCD

	//advice
	after(): supervisorSetup(){
		this.supervisor = (Supervisor) thisJoinPoint.getTarget();
	}

	//
	// Inicializa o Heimdall
	//
	public SecurityAspect() {
		Heimdall.authenticationManager.setApplicationName("hosp");
	}

}
```

AuthenticationAspect

```
package br.mestrado.hosp.aspects.security;

import jade.core.AID;
import javax.security.auth.login.LoginException;
import org.heimdall.Heimdall;


/**
 * Aspecto responsável pela autenticacao (base)
 * @author Willian
 */
public abstract aspect AuthenticationAspect extends SecurityAspect {


      /**
       * Login do agente
       *
       * @param agente O agente a ser autenticado
       * @param userName usuario
       * @param password senha
       */
      protected boolean agentLogIn(AID agent, String userName,
      String password) {

            try {
                  if (Heimdall.authenticationManager.logIn(agent,
                   userName, password)) {

                        //System.out.println(Utils.getTime() + " - agente
                        [" + agent.toString() +
                        "] foi autenticado com o usuario: " + userName);

                        System.out.println(this.getClass().getSimpleName()+
                        " - o agente [" + agent.getName()+ "] foi
                        autenticado com o usuario: "
                        + userName);

                        return true;
                  }
                  else {
                        //System.err.println(Utils.getTime() + " - o agente
                        [" + agent.getName() + "] nao foi autenticado com o
                        usuario: " + userName);
                        System.err.println(this.getClass().getSimpleName()+
                        " - o agente [" + agent.getName() + "] nao foi
                        autenticado com o usuario: " + userName);

                        return false;
                  }

            }
            catch (LoginException e) {

                  System.err.println("***** AGENTE NAO AUTENTICADO
                  *****");
```

```
                System.err.println("***** agente: [" + agent.getName());
                System.err.println("***** " + e.getMessage());
                return false;
            }
        }
    }
```

ProactiveAuthenticationAspect

```
package br.mestrado.hosp.aspects.security;

import br.mestrado.hosp.agents.BaseAgent;

/**
 * Aspecto responsável pela autenticacao (agentes no ambiente)
 * @author Willian
 */
public aspect ProactiveAuthenticationAspect extends AuthenticationAspect {

	//
	// Authentication (Proactive), agente deste ambiente
	//

	pointcut proActiveAuthentication() :
		within(br.mestrado.hosp.agents.*) &&
		 !within(br.mestrado.hosp.agents.BaseAgent) &&

		 //para fazer o teste T1
		 //!within(br.mestrado.hosp.agents.Paciente) &&
		 //para fazer o teste T1

			execution(protected void setup());

	after(): proActiveAuthentication(){

		//pega o agente
		BaseAgent agente = (BaseAgent) thisJoinPoint.getTarget();

		//pega o usuario e senha do agente;
		String userName = agente.getUser();
		String password = agente.getPassword();

		if(agentLogIn(agente.getAID(), userName, password))
		{
			System.out.println("*** Proactive Authentication: " +
			agente.getAID().toString() + "user: " + userName);
		}
		else
		{
			System.err.println("***** AGENTE NAO AUTENTICADO
			*****");
			System.err.println("***** agente: [" + agente.getName() +
			"]");
		}

	}

}
```

ReactiveAuthenticationAspect

```
package br.mestrado.hosp.aspects.security;

import jade.content.Concept;
import jade.content.ContentElement;
import jade.content.onto.basic.Action;
import jade.lang.acl.ACLMessage;
import br.mestrado.hosp.agents.ontology.InformarAcessoNegadoAction;
import br.mestrado.hosp.agents.ontology.SolicitarAutenticacaoAction;

/**
 * Aspecto responsável pela autenticacao (agentes fora do ambiente)
 * @author Willian
 */
public aspect ReactiveAuthenticationAspect extends AuthenticationAspect {

	//
	// Authentication (Reactive), agente de outro ambiente
	//
	pointcut reActiveAuthentication() :
		call(void
br.mestrado.hosp.agents.Supervisor.ReceiveMessagesBehaviour.processAction(A
CLMessage, ContentElement));

	//advice
	void around() : reActiveAuthentication(){

		Object[] params = thisJoinPoint.getArgs();
		ACLMessage msg = (ACLMessage) params[0];
		ContentElement content = (ContentElement) params[1];

		// pega a acao
		Concept action = ((Action) content).getAction();

		if (action instanceof SolicitarAutenticacaoAction) {

			SolicitarAutenticacaoAction solicitarAutenticacaoAction =
			(SolicitarAutenticacaoAction) action;

			//AID agent = msg.getSender();

			String[] values =
			solicitarAutenticacaoAction.getEntidade().toString()
			.split(",#@,");

			if (agentLogIn(msg.getSender(), values[0], values[1])) {
				System.out.println("*** Reactive Authentication: "+
				msg.getSender().toString() + " user: " +values[0]);
			}
			else {

				System.err.println("***** AGENTE NAO AUTENTICADO
				*****");
				System.err.println("***** agente: [" + m
				sg.getSender().getName() + "]");
```

```
                        InformarAcessoNegadoAction informarAcessoNegado =
                        new InformarAcessoNegadoAction();

      informarAcessoNegado.setAgentAction(solicitarAutenticacaoAction);

      informarAcessoNegado.setAccessDeniedMessage("USUARIO ou SENHA
INVALIDA");

                        supervisor.replyAcessoNegado(msg.getSender(),
                        informarAcessoNegado);

                  }

            } else
                  proceed();
      }
}
```

AccessControlAspect

```
package br.mestrado.hosp.aspects.security;

import jade.content.AgentAction;
import jade.core.AID;

import javax.security.auth.login.LoginException;

import org.heimdall.Heimdall;
import org.heimdall.security.authorization.rbac.Permission;

import br.mestrado.hosp.agents.ontology.AtualizarGlicemiaAction;
import br.mestrado.hosp.agents.ontology.AvisarDiabetologistaAction;
import br.mestrado.hosp.agents.ontology.InformarAcessoNegadoAction;
import br.mestrado.hosp.agents.ontology.InformarGlicemiaAction;
import br.mestrado.hosp.agents.ontology.InformarPacienteAction;
import br.mestrado.hosp.agents.ontology.LiberarPacienteAction;
import br.mestrado.hosp.agents.ontology.RegistrarPacienteAction;
import br.mestrado.hosp.agents.ontology.SolicitarAutenticacaoAction;
import br.mestrado.hosp.agents.ontology.SolicitarGlicemiaAction;
import br.mestrado.hosp.agents.ontology.SolicitarPacienteAction;


/**
 * Aspecto responsável pelo controle de acesso (base)
 * */
public abstract aspect AccessControlAspect extends SecurityAspect {

	//
	// Controle de acesso
	//
	/**
	 *
	 * Verifica se um agente possui permissao para uma ação.
	 *
	 * @param agent Um agente já autenticado
	 * @param action A ação que o agente está tentando
	executar/solicitando execução
	 */
	protected boolean agentHasPermission(AID agent, AgentAction action)
	throws LoginException {

		//
		// Mapeamento (action <-> permission)
		//

		org.heimdall.security.authorization.rbac.Permission p = null;


		if (action instanceof AtualizarGlicemiaAction) {
			p = new
Permission(SecurityConstants.Permissions.AtualizarGlicemia.toString(),
null);
		}
		else if (action instanceof AvisarDiabetologistaAction) {
```

```
                  p = new
Permission(SecurityConstants.Permissions.AvisarDiabetologista.toString(),
null);
            }
            else if (action instanceof InformarAcessoNegadoAction) {
                  p = new
Permission(SecurityConstants.Permissions.InformarAcessoNegado.toString(),
null);
            }
            else if (action instanceof InformarGlicemiaAction) {
                  p = new
Permission(SecurityConstants.Permissions.InformarGlicemia.toString(),
null);
            }
            else if (action instanceof InformarPacienteAction) {
                  p = new
Permission(SecurityConstants.Permissions.InformarPaciente.toString(),
null);
            }
            else if (action instanceof LiberarPacienteAction) {
                  p = new
Permission(SecurityConstants.Permissions.LiberarPaciente.toString(), null);
            }
            else if (action instanceof RegistrarPacienteAction) {
                  p = new
Permission(SecurityConstants.Permissions.RegistrarPaciente.toString(),
null);
            }
            else if (action instanceof SolicitarAutenticacaoAction) {
                  p = new
Permission(SecurityConstants.Permissions.SolicitarAutenticacao.toString(),
null);
            }
            else if (action instanceof SolicitarGlicemiaAction) {
                  p = new
Permission(SecurityConstants.Permissions.SolicitarGlicemia.toString(),
null);
            }
            else if (action instanceof SolicitarPacienteAction) {
                  p = new
Permission(SecurityConstants.Permissions.SolicitarPaciente.toString(),
null);
            }
            else {
                  return false;
            }

            //
            //Verifica se o agente (usuario do agente) possui a permissao
            //
            if (Heimdall.rbacManager.hasPermission(agent, p)) {
                  return true;
            } else {
                  return false;
            }

      }

}
```

ProactiveAccessControlAspect

```
package br.mestrado.hosp.aspects.security;

import jade.content.AgentAction;
import jade.core.AID;

import javax.security.auth.login.LoginException;

import org.heimdall.Heimdall;

import br.mestrado.hosp.agents.BaseAgent;

/**
 * Aspecto responsável pelo controle de acesso proativo (no envio da
mensagem)
 * @author Willian
 */
public aspect ProactiveAccessControlAspect extends AccessControlAspect {

      pointcut agentSendMessage(int performative, AID receiver, AgentAction
action) :
            args( performative, receiver, action) &&
            execution(void br.mestrado.hosp.agents.*.sendMessage(int,
            AID , AgentAction )) ;

      //advice
      void around(int performative, AID receiver, AgentAction action):
      agentSendMessage(performative, receiver, action){

            BaseAgent sender = (BaseAgent) thisJoinPoint.getTarget();

            //System.out.println("PAC[" + performative + "]:\t(" +
            sender.getName() + " => " + receiver.getName() + "):\t" +
            action.getClass().getSimpleName());
            System.out.println("ProactiveAccessControlAspect - (" +
            sender.getName() + " => " + receiver.getName() + "):" +
            action.getClass().getSimpleName());

            //
            //Analisa se o agente tem permissao
            //

            try {



                  if(  (!(sender.getAID().getAddressesArray()[0]).
                        equals(receiver.getAddressesArray()[0]))
                              &&
      !Heimdall.authenticationManager.isAuthenticated(receiver))
      //agente externo, nao autenticado que precisa ser avisado
                  {
                        proceed(performative, receiver, action);
                        //se tem permissao envia a mensagem
                  }
                  else if (agentHasPermission(sender.getAID(), action) &&
```

```
                        agentHasPermission(receiver, action)) {
                        proceed(performative, receiver, action);
                        //se tem permissao envia a mensagem
                  } else {

                        //
                        // Exibe a mensagem que o agente sender ou o
                        receiver nao esta autorizado a executar a acao
                        //
                        System.err.println("***** ACESSO NEGADO *****");
                        System.err.println("***** agente: [" +
                        sender.getName() + "] ou [" + receiver.getName() +
                        "]");
                        System.err.println("***** sem permissao para acao:
                        ("+action.getClass().getSimpleName()+")");

                  }
            } catch (LoginException e) {

                  //
                  // Exibe a mensagem que o agente sender nao esta
                  autenticado
                  //
                  System.err.println("***** AGENTE NAO AUTENTICADO
                  *****");
                  System.err.println("***** agente: [" + sender.getName() +
                  "] ou [" + receiver.getName() + "]");
                  System.err.println("***** HeimdallException: "
                  +e.getMessage());
            }

      }

}
```

ReactiveAccessControlAspect

```
package br.mestrado.hosp.aspects.security;

import jade.content.AgentAction;
import jade.content.ContentElement;
import jade.content.onto.basic.Action;
import jade.core.AID;
import jade.lang.acl.ACLMessage;

import javax.security.auth.login.LoginException;

import br.mestrado.hosp.agents.ontology.InformarAcessoNegadoAction;

/**
 * Aspecto responsável pelo controle de acesso reativo (no recebimento da
mensagem)
 * @author Willian
 */
public aspect ReactiveAccessControlAspect extends AccessControlAspect {

      //recebimento de mensagens pelo agente
      pointcut agentReceiveMessage(ACLMessage message, ContentElement
content) :
            args( message, content) &&
            execution(void
br.mestrado.hosp.agents.*.ReceiveMessagesBehaviour.processAction(ACLMessage
, ContentElement));



      //advice
      void around(ACLMessage message, ContentElement content) :
agentReceiveMessage(message,  content){

            AID sender = message.getSender();
            AID receiver = ((AID) message.getAllReceiver().next());

            if (!(sender.getAddressesArray()[0]).
                  equals(receiver.getAddressesArray()[0])) {

                  AgentAction action = (AgentAction)
                  (((Action)content).getAction());

                  //System.err.println("RAC[" + message.getPerformative() +
                  "]:\t(" + sender.getName() + " => "
                  + receiver.getName() + "):\t"
                        //    + action.getClass().getSimpleName());

                  System.out.println("ReactiveAccessControlAspect - (" +
                  sender.getName() + " => " + receiver.getName() + "):" +
                  action.getClass().getSimpleName());


                  //
                  //Analisa se o agente tem permissao e avisa ele caso não
                  tenha
```

```
                //
                InformarAcessoNegadoAction informarAcessoNegado =
                new InformarAcessoNegadoAction();
                informarAcessoNegado.setAgentAction((AgentAction)
                action);

                try {

                      if (agentHasPermission(sender, (AgentAction)
                      action))
                      //e o receiver tb
                      {
                            proceed(message, content);
                            //se tem permissao recebe a mensagem
                      } else {


                            System.err.println("***** ACESSO NEGADO
                             *****");
                            System.err.println("***** agente: [" +
                sender.getName() + "] ou [" + receiver.getName() + "]");
                            System.err.println("***** sem permissao para
                      acao: ("+action.getClass().getSimpleName()+")");


                            //aborta o envio da mensagem e envia uma msg
                            para ele dizendo que nao tem direito.

      informarAcessoNegado.setAccessDeniedMessage("ACESSO NEGADO");

                            supervisor.replyAcessoNegado(sender,
                            informarAcessoNegado);
                      }
                } catch (LoginException e) {

                      System.err.println("***** AGENTE NAO AUTENTICADO
                      *****");
                      System.err.println("***** agente: [" +
                sender.getName() + "] ou [" + receiver.getName() + "]");
                      System.err.println("***** HeimdallException: "
                      +e.getMessage());


                      informarAcessoNegado.setAccessDeniedMessage("AGENTE
                      NAO ESTA AUTENTICADO");
                      supervisor.replyAcessoNegado(sender,
                      informarAcessoNegado);
                }

          } else
                proceed(message, content); //local

      }
}
```