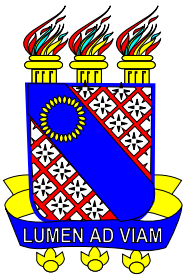


UNIVERSIDADE ESTADUAL DO CEARÁ

ANDERSON DE CASTRO LIMA

SACM - UM MODELO DE CONTROLE DE ACESSO BASEADO EM ESTADO E SENSÍVEL AO CONTEXTO

FORTALEZA - CEARÁ

2012

ANDERSON DE CASTRO LIMA

SACM - UM MODELO DE CONTROLE DE ACESSO BASEADO EM ESTADO E SENSÍVEL AO CONTEXTO

Dissertação apresentada no Curso de Mestrado Acadêmico em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Orientação: Prof. Dr. André Luiz Moura dos Santos

Orientação: Prof. Dr. Joaquim Celestino Júnior

FORTALEZA - CEARÁ

2012

Dados Internacionais de Catalogação na Publicação
Universidade Estadual do Ceará
Biblioteca Central Prof. Antônio Martins Filho

L732s	Lima, Anderson de Castro. SACM - Um Modelo de Controle de Acesso baseado em estado e Sensível ao Contexto / Anderson de Castro Lima. – Fortaleza, 2012. 82f. : il. color., enc. ; 30 cm. Orientação: Prof. Dr. André Luiz Moura dos Santos Orientação: Prof. Dr. Joaquim Celestino Júnior Dissertação (Mestrado) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Acadêmico em Ciência da Computação. Área de Concentração: Ciência da Computação. 1. Controle de Acesso. 2. Pervasivo. 3. Segurança. 4. Sensível ao Contexto. I. Universidade Estadual do Ceará, Centro de Ciências e Tecnologia.

CDD: 004.6

ANDERSON DE CASTRO LIMA

SACM - UM MODELO DE CONTROLE DE ACESSO BASEADO EM ESTADO E SENSÍVEL AO CONTEXTO

Dissertação apresentada no Curso de Mestrado Acadêmico em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial para obtenção do grau de Mestre.

Área de Concentração: Ciência da Computação

Aprovada em: 31/05/2012

BANCA EXAMINADORA

Prof. Dr. André Luiz Moura dos Santos
Universidade Estadual do Ceará – UECE

Prof. Dr. Joaquim Celestino Júnior
Universidade Estadual do Ceará – UECE

Prof. Dr. Marcial Porto Fernandez
Universidade Estadual do Ceará – UECE

Prof. Dr. Pedro Klecius Farias Cardoso
Instituto Federal do Ceará – IFCE

Profa. Dra. Rossana Maria de Castro Andrade
Universidade Federal do Ceará – UFC

AGRADECIMENTOS

Primeiramente, agradeço a Deus nosso pai pela vida e por me permitir alcançar este tão esperado sonho.

Agradeço ao meu pai José de Lima Filho e a minha mãe Antônia Neide, pela educação que me proporcionaram, mesmo com todas as dificuldades e a meus irmãos Aline e Alisson pelo apoio.

Agradeço especialmente a minha amada esposa Wládia, por sua paciência e compreensão, pois fomos privados de estarmos em vários momentos especiais para que este trabalho pudesse ser finalizado.

A meu filho Arthur pela alegria que me proporciona diariamente.

Ao professor Joaquim Celestino Júnior por ter ajudado a fortalecer minha formação no mestrado.

Ao professor André dos Santos que me deu a chance de trabalhar com ele, sem a qual não conseguiria finalizar esta etapa da minha formação.

Aos meus alunos e amigos Gideão Santana e Victor Alisson, pela ajuda em uma das partes mais importantes do trabalho que foi a implementação do SACM em JAVA.

Ao pessoal do INSERT pela ajuda nas pesquisas.

Aos membros da banca: Profa. Dra. Rossana Maria de Castro Andrade, Prof. Dr. Pedro Klecius Farias Cardoso e Prof. Dr. Marcial Porto Fernandez pelas contribuições para aperfeiçoamente deste trabalho.

A todas as pessoas que passaram pela minha vida e contribuíram para a construção de quem sou hoje.

"Não existe alguém que nunca teve um professor na vida, assim como não há ninguém que nunca tenha tido um aluno. Quanto mais se aprende, mais se quer ensinar. Quanto mais se ensina, mais se quer aprender."
Içami Tiba

RESUMO

Os mecanismos de controle de acesso são fundamentais na construção de ambientes onde a informação digital é um bem que precisa ser protegido, no entanto, a maioria das pesquisas nesta área tem sido gastas em controles de acesso para situações que não levam em consideração a mobilidade dos usuários que está presente nos dias atuais. Com o advento de novas tecnologias que proporcionaram a computação pervasiva ou ubíqua, a necessidade de se ter um controle preparado para esse novo ambiente veio à tona. Nesta dissertação, é apresentado o SACM - *Statefull Access Control Model*, projetado para ser adequado a sistemas estáticos e móveis, ou seja, sua construção se baseou em alguns modelos de controle de acesso dentre eles o RBAC, CARBAC, Chinese Wall e outros, de forma a cobrir uma vasta gama de situações que antes eram cobertas por dois modelos ou mais de maneira simultânea. Este modelo inovador, explora o conceito de Sensibilidade ao Contexto para fornecer um rico ambiente propício a sistemas móveis e ubíquos. A sensibilidade ao contexto proporciona um modelo hábil a trabalhar em ambientes onde as alterações de contexto do ambiente são constantes. Nosso modelo também poderá ser utilizado em sistemas de informação centralizados em que as regras de controle de acesso são armazenadas em um servidor central. Por fim apresentamos implementação do Modelo utilizando quatro exemplos onde podemos constatar sua expressividade.

Palavras-chave: Controle de Acesso. Pervasivo. Segurança. Sensível ao Contexto.

ABSTRACT

The access control mechanisms are fundamental in building environments where digital information is an asset that must be protected, however, most research in this area has been spent on access controls for situations that do not take into account the mobility of users who are present today. With the advent of new technologies that provided the ubiquitous or pervasive computing, the need to have a control prepared for this new environment emerged. This dissertation presents the SACM - Statefull Access Control Model, designed to be suitable for static and mobile systems, ie, its construction was based on models of access control among them RBAC, CARBAC, Chinese Wall and others to cover a wide range of conditions that were covered by two or more models simultaneously. This innovative model, explores the concept of Context-Aware to provide a rich environment for mobile and ubiquitous systems. The Context-Aware provides a model able to work in environments where changes in environment context are constant. Our model can also be used in centralized information systems in which access control rules are stored on a central server. Finally is presented the Model's implementations using four examples where we can see their expressiveness.

Keywords: Access Control. Pervasive. Security. Context-aware.

LISTA DE FIGURAS

Figura 1 Matriz de Controle de Acesso 17

Figura 2 ACL da matriz da Figura 1 18

Figura 3 CL da matriz da Figura 1 18

Figura 4 Exemplo de uma organização de objetos 20

Figura 5 Mecanismo de Acesso BEE 22

Figura 6 Framework de Autorização do Ponder2 32

Figura 7 Hierarquia dos objetos 39

Figura 8 Processo de Varredura 40

Figura 9 Diagrama UML 52

Figura 10 Quiosque de Impressão 57

Figura 11 Confirmação da compra de créditos 61

Figura 12 Tentativa de devolução do cliente com erro 61

Figura 13 Solicitação de impressão 62

Figura 14 Diminuição do valor do *token* do cliente 62

Figura 15 Muralha da China 63

Figura 16 Representação gráfica do Chinese Wall 66

Figura 17 Criando as pastas 66

Figura 18 Registro do Acesso 67

Figura 19 Acesso Negado 67

Figura 20 Distribuição de Ingressos 67

Figura 21 Interface de distribuição dos ingressos 71

Figura 22 Criação dos clientes 71

Figura 23 Solicitação de ingresso 71

Figura 24 Solicitação negada 72

Figura 25 Sensibilidade ao Contexto em um hospital 73

Figura 26 Acesso de enfermeira do tipo leitura 78

Figura 27 Negação do acesso devido a ausência do Médico 78

Figura 28 Permissão de escrita devido a presença do Médico 78

LISTA DE SIGLAS

PDA	*Personal Digital Assistant*
SACM	*Statefull Access Control Model*
MAC	*Mandatory Access Control*
DAC	*Discretionary Access Control*
ACL	*Access Control List*
CL	*Capability List*
RBAC	*Role-Based Access Control*
TRBAC	*Temporal Role-Based Access Control*
SAC	*Secure Areas of Computation*
DACM	*Dynamic Access Control Model*
BEE	*Boolean Expression Evaluation*
GTRBAC	*Generalized Temporal Role-Based Access Control*
SMC	*Self-Managed Cell*
PAF	*Ponder2 Autorization Framework*
RFID	*Radio-Frequency Identification*
API	*Application Programming Interface*

SUMÁRIO

1 INTRODUÇÃO 13
1.1 CONCEITOS BÁSICOS SOBRE CONTROLE DE ACESSO 13
1.2 MOTIVAÇÃO 13
1.3 OBJETIVOS 15
1.4 ESTRUTURA DO TRABALHO 15
2 TRABALHOS RELACIONADOS 16
2.1 MODELOS TRADICIONAIS 16
2.1.1 Modelo de Controle de Acesso Mandatário 16
2.1.2 Modelo de Controle de Acesso Discricionário 16
2.1.3 Modelo de Controle de Acesso baseado em uma Matriz 17
2.1.4 Modelo de Controle de Acesso Baseado em Papéis 19
2.1.5 Modelo de Controle de Acesso Muralha da China 20
2.1.6 Modelo de Controle de Acesso com criação dinâmica de regras 21
2.2 MODELOS DE CONTROLE DE ACESSO COM POLÍTICAS COMPLEXAS 22
3 NOVOS PARADIGMAS PARA O CONTROLE DE ACESSO 25
3.1 COMPUTAÇÃO PERVASIVA OU UBÍQUA 25
3.2 CONCEITO DE SENSIBILIDADE AO CONTEXTO 25
3.3 AQUISIÇÃO DAS INFORMAÇÕES DE CONTEXTO 27
3.4 EXEMPLOS DE APLICAÇÕES UBÍQUAS SENSÍVEIS AO CONTEXTO 28
4 O PONDER2 30
4.1 O QUE É O PONDER2 30
4.2 OS OBJETOS GERENCIÁVEIS 31
4.3 AS POLÍTICAS 31
4.3.1 Framework de Autorização do Ponder2 32
4.4 O PONDERTALK 33
4.4.1 As Politicas no Pondertalk 34
4.5 OBJETOS GERENCIÁVEIS UTILIZADOS NO PONDER2 35
4.6 UMA COMPARAÇÃO ENTRE O PONDER2 E O SACM 36

5 SACM **38**
5.1 VISÃO GERAL 38
5.2 ESTRUTURA DO SACM 39
5.3 HIERARQUIA DOS OBJETOS 39
5.4 AUTORIZAÇÕES E AVALIAÇÕES 40
5.5 ESTRUTURA DAS REGRAS 41
5.5.1 Condições **42**
5.5.2 Ações **43**
5.5.3 Identificação das Regras **43**
5.5.4 Regras *(RuleData)* **44**
5.6 EXEMPLOS DE USO DO SACM 44
5.6.1 Controle de Acesso Simples **45**
5.6.2 Controle de Acesso Temporal **45**
5.6.3 Controle de Acesso com Contador de Acesso **46**
5.6.4 Controle de acesso baseado em ações prévias (estado) **48**
5.6.4.1 Acesso baseado em ações prévias 48
5.6.4.2 Acesso baseado em regras criadas previamente 49
5.6.5 Controle de Acesso com Sensibilidade ao Contexto **50**
6 IMPLEMENTAÇÃO DO MODELO USANDO A LINGUAGEM JAVA **52**
6.1 GERENCIAMENTO DOS OBJETOS 53
6.2 EXEMPLO DE UM QUIOSQUE DE IMPRESSÃO 56
6.2.1 Exemplo com *interface* gráfica **61**
6.3 EXEMPLO BASEADO NO *CHINESE WALL* 62
6.3.1 Exemplo com *interface* gráfica **66**
6.4 EXEMPLO DE DISTRIBUIÇÃO DE INGRESSOS 67
6.4.1 Exemplo com *interface* gráfica **71**
6.5 EXEMPLO COM SENSIBILIDADE AO CONTEXTO 72
6.5.1 Exemplo com *interface* gráfica **78**
7 CONSIDERAÇÕES FINAIS **79**
REFERÊNCIAS 80

1 INTRODUÇÃO

1.1 Conceitos básicos sobre Controle de Acesso

Uma das características mais importantes dos sistemas de hoje é a proteção dos seus recursos (isto é, dados e serviços) contra a divulgação não autorizada (confidencialidade) e alterações intencionais ou acidentais não autorizadas (integridade) enquanto ao mesmo tempo, garantem a sua acessibilidade para usuários autorizados sempre que necessário (disponibilidade) (SAMARATI; VIMERCATI, 2001). Um esforço considerável está sendo dedicado para abordar vários aspectos de integridade, confidencialidade e disponibilidade.

Um dos principais serviços de segurança utilizados para alcançar a proteção de dados é o Controle de Acesso. O Controle de Acesso é o ato de garantir que um sujeito tenha acesso apenas ao que ele está autorizado e nada mais. Pesquisas significativas estão focando em alcançar sistemas de controle de acesso mais expressivos e poderosos.

Para melhor compreendermos o que seja controle de acesso, os conceitos de sujeitos e objetos devem ser conhecidos. Um sujeito é uma entidade ativa em um sistema computacional que inicia requisições por recursos; corresponde, via de regra, a um usuário ou a um processo executando em nome de um usuário. Um objeto é uma entidade passiva que armazena informações no sistema como arquivos, diretórios e segmentos de memória. O controle de acesso, é portanto, a mediação das requisições de acesso a objetos iniciadas pelos sujeitos.

O desenvolvimento de um sistema de Controle de Acesso requer, a definição de regulamentos de acordo com o qual o acesso deva ser controlado. Esse processo de desenvolvimento é realizado geralmente com uma abordagem em várias fases com base nos conceitos de política de segurança, modelo de segurança e mecanismo de segurança. A política define as regras (alto nível) segundo a qual o Controle de Acesso deve ser regulamentado. A política é então acompanhada por uma linguagem para a especificação das regras. Um modelo de controle de acesso fornece uma representação formal da política de segurança do Controle de Acesso e seu funcionamento, um mecanismo de segurança define as funções de baixo nível (software e hardware) que implementam os controles impostos pela política e formalmente indicados no modelo.

1.2 Motivação

Os modelos de controle de acesso tradicionais foram adequados para os sistemas de computação clássica, como banco de dados e sistemas de arquivos, onde necessariamente existe uma infraestrutura centralizada que armazenava as informações a serem acessadas. Porém, com o surgimento da computação Pervasiva ou Ubíqua, essas abordagens deixaram de ser adequadas.

Na computação Ubíqua ou Pervasiva os usuários tem acesso aos recursos o tempo todo e em todos os lugares, o desafio de se projetar e manter um controle de acesso se tornou ainda maior, pois os dados poderão ser acessados de forma mais fácil e transparente para o usuário. Um exemplo seria um pesquisador que possui uma pesquisa que exige o máximo de sigilo. Um controle de acesso pervasivo poderia liberar o acesso aos arquivos da pesquisa para os bolsistas que auxiliam na pesquisa somente com a presença do pesquisador chefe e em uma sala escolhida por ele, caso ele saia da sala o acesso dos bolsistas é bloqueado.

Em ambientes com computação Pervasiva, as informações do usuário e do ambiente (contexto) são coletadas em tempo real e com constantes mudanças, assim a utilização de informações de contexto são essenciais na tomada de decisão do mecanismo de controle de acesso dinâmico.

Adicionalmente, o uso de PDAs, *Tablets, Smart Cards* ou *Smartphones* com uma área de armazenamento segura podem ser usados para tarefas diárias do mundo não digital como alugar livros/filmes ou obter acesso a um clube privado ou a uma academia. Eles podem ser também conectados a sistemas existentes na Internet para prover autenticação para bibliotecas digitais e aplicações similares.

A principal diferença entre esses novos modelos de controle de acesso e os tradicionais é a necessidade de se manter o estado através das operações. Imagine uma cidade como Bonn na Alemanha, onde os governantes incentivam a utilização de bicicletas como meio de transporte para seus habitantes e para os turistas. Para não obrigar que turistas tenham que comprar bicicletas, foram instalados vários quiosques pela cidade onde é possível alugá-las. Um usuário poderá locar até duas bicicletas para utilizar por um tempo determinado, ele só poderá retirar a bicicleta caso não possua nenhuma pendência, por exemplo, esqueceu de entregar a bicicleta anteriormente alugada ou estourou sua cota. Como os quiosques estão distríbuidos por uma grande área geográfica, os custos para instalar uma rede de comunicação entre eles seria muito alto, desse modo a ideia seria que cada usuário carregasse consigo as informações necessárias para obter a autorização de acesso (SANTOS et al., 2011).

Além disso, os modelos tradicionais carecem da habilidade de mudarem dinamicamente. Isto é, um conjunto de regras tradicionais, não é capaz de expressar políticas que necessitam de regras capazes de criar novas regras ou excluir regras antigas. Tal abordagem se faz necessária em situações onde os recursos são limitados como na utilização de PDAs para controlar o acesso a um determinado objeto ou recurso.

Com as novas pesquisas em computação pervasiva os dispositivos móveis se tornaram importantíssimos no caráter computacional das aplicações atuais. Alguns aeroportos consideram a ideia de se ter quiosques de impressão que trabalham de maneira *offline* (para reduzir os custos de infraestrutura e gerenciamento de uma rede). Utilizando a tecnologia *Bluetooth* ou *WiFi* para comunicação local e tendo os quiosques distribuídos pelos aeroportos permitindo

que um usuário imprima até “n” páginas em qualquer lugar (onde “n” é o serviço adquirido pelo usuário) e que estará armazenado em uma área segura ou em um *Smart Card*, esse tipo de aplicação introduz dificuldades de controle de acesso que os modelos tradicionais são incapazes de expressar (SANTOS et al., 2011).

1.3 Objetivos

O modelo de controle de acesso proposto neste trabalho tem como objetivo atender aos requisitos dos sistemas computacionais tradicionais, assim como aos requisitos exigidos pela computação móvel, por exigir que as informações possam está descentralizadas. O Modelo de Controle de Acesso baseado em Estado SACM *(Statefull Access Control Model)*, tem como principais características: a mobilidade, a capacidade de guardar o estado *(statefull)*, a centralidade ou descentralidade e a dinamicidade.

1.4 Estrutura do Trabalho

O trabalho está organizado da seguinte forma: o Capítulo 2 apresenta conceitos e características sobre modelos de controle de acesso que serviram de base para a construção do SACM. O Capítulo 3 expõe os novos paradigmas da computação móvel discorrendo sobre suas características gerais e exemplifica um modelo que acompanha essas mudanças. No Capítulo 4 falamos sobre a linguagem de política Ponder2 por se tratar de um modelo capaz de acompanhar os novos requisitos de controle de acesso e possuir muitas similaridades com nosso modelo e fechamos com uma pequena comparação entre ambos. O capítulo 5 apresenta o SACM e exemplos utilizando as ideias propostas pelo modelo. O Capítulo 6 demonstra a implementação do SACM feito em JAVA e finalmente no Capítulo 7 é exposto os próximos passo no desenvolvimento do SACM e as considerações finais.

2 TRABALHOS RELACIONADOS

Esta Seção contém uma visão geral sobre alguns modelos de controle de acesso. Ela é centrada nos Modelos Tradicionais e Complexos. Os modelos tradicionais utilizados para descrever a aplicação de confidencialidade são baseadas na definição de regras de controle de acesso, chamados de autorizações, que são da forma <sujeito, objeto, operação>. Estas autorizações especificam quais operações podem ser executadas em objetos e por quais sujeitos. No entanto, nos sistemas atuais a definição de um modelo de controle de acesso se torna mais complicada pela necessidade de representar formalmente políticas complexas, onde as decisões de acesso podem depender da aplicação de regras diferentes que podem surgir ao longo do tempo. Como o modelo proposto é caracteristicamente discricionário, daremos maior ênfase aos trabalhos que funcionam da mesma maneira.

2.1 Modelos Tradicionais

2.1.1 Modelo de Controle de Acesso Mandatário

O MAC – *Mandatory Access Control* (Controle de Acesso Mandatário) é baseado em uma autorização prévia sendo sua identidade irrelevante, ou seja: quando um mecanismo do sistema controla o acesso a um objeto e um usuário não pode alterar esse acesso, o controle é chamado de controle de acesso mandatário (MAC), às vezes chamado de controle de acesso baseado em regras. Um exemplo de controle mandatário seria a lei que permite a um tribunal acessar os registros dos condutores de veículos sem autorização prévia dos proprietários. Este é um controle obrigatório, porque o proprietário do registro não tem controle sobre o acesso a informação pelo tribunal (SANDHU; SAMARATI, 1994).

2.1.2 Modelo de Controle de Acesso Discricionário

O DAC – *Discretionary Access Control* (Controle de Acesso Discricionário) baseia-se na ideia de que é o proprietário da informação que deve determinar quem tem acesso a ela. Um exemplo de controle discricionário seria o seguinte: suponha que uma criança que possui um diário, a criança controla o acesso ao diário, pois ela pode permitir que alguém o leia (conceder acesso de leitura) ou não permitir que alguém o leia (negar acesso de leitura). A criança permite que a sua mãe possa lê-lo, mas ninguém mais. Este é um controle de acesso discricionário, pois o acesso ao diário baseia-se na vontade do dono de conceder ou não o acesso(SANDHU; SAMARATI, 1994).

2.1.3 Modelo de Controle de Acesso baseado em uma Matriz

O modelo Matriz de Controle de Acesso fornece um *framework* para descrever controle de acesso discricionário. Primeiramente proposto por (LAMPSON, 1974) para a proteção de recursos dentro do contexto de sistemas operacionais, o modelo foi formalizado por (HARRISON; RUZZO; ULLMAN, 1976) sendo um dos mais fundamentais entre os modelos de controle de acesso. Ele consiste em caracterizar os direitos que um sujeito possui sobre outras entidades do sistema. Na matriz de acesso existe uma linha para cada sujeito e uma coluna para cada objeto. Cada célula C_{ij} especifica o acesso autorizado que o sujeito da linha i tem sobre o objeto da coluna j. A Figura 1 ilustra uma matriz de acesso.

		Objetos			
		Arquivo 1	Arquivo 2	Programa 1	Arquivo 3
Sujeitos	Yuri	Possui Ler Escrever			Ler Escrever
	Calvin	Ler		Executar	Ler Escrever
	Susie	Escrever		Executar Ler	

Figura 1: Matriz de Controle de Acesso

Em grandes sistemas, a Matriz de controle de acesso se tornará muito grande sendo que a maioria de suas células estará vazia. Existem duas maneiras típicas de se implementar uma matriz de controle de acesso na prática:

- **Listas de Controle de Acesso** *(Access Control List* – ACL): a matriz é armazenada por coluna. Cada objeto está associado a uma lista, contendo para cada sujeito, as ações permitidas a serem executadas no objeto.
- **Listas de Capacidades** *(Capability List* – CL): a matriz é armazenada por linha. Cada sujeito está associado a uma lista, indicando para cada objeto, os acessos permitidos ao sujeito. A Figura 2 e a Figura 3 ilustram, respectivamente, a ACL e a CL da matriz de acesso da Figura 1.

Esses dois modelos são bastantes atraentes devido à simplicidade. Tipicamente os sistemas operacionais utilizavam a abordagem ACL e CL. O modelo de permissão do UNIX de 9-bits é um exemplo da utilização da ACL onde um usuário, um grupo e os demais usuários que não estão em nenhum grupo possuem os direitos de ler, escrever e executar um objeto representado através de 3 bits (111 significa que o sujeito pode ler[r], escrever[w] e executar[x] - rwx). Entretanto é devido a essa simplicidade que esses modelos não são mais adequados para fornecer as necessidades de segurança atualmente.

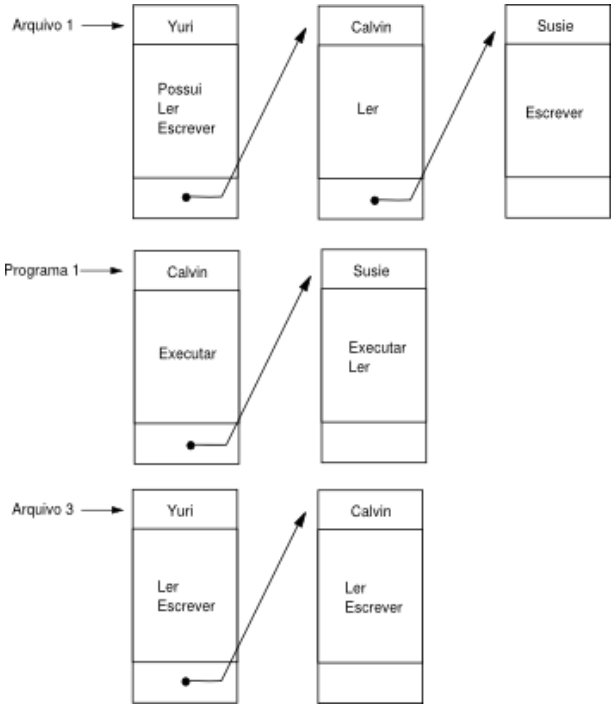


Figura 2: ACL da matriz da Figura 1

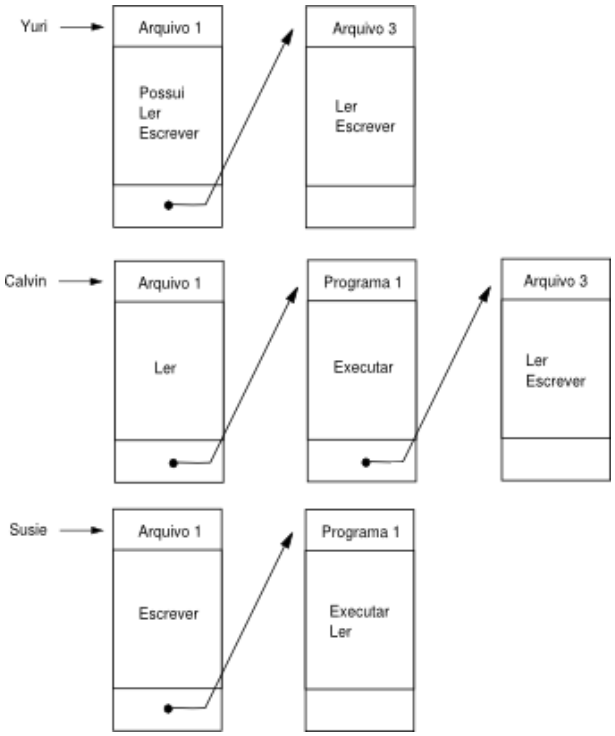


Figura 3: CL da matriz da Figura 1

Baseado no pressuposto que os sujeitos possuem os dados, esses modelos não atenderiam aos requisitos de segurança em um ambiente corporativo, onde normalmente, na empresa o importante é que o papel do usuário nela possua os dados. Um sujeito obtém acesso aos dados baseado na classificação ou o tipo do seu serviço, ou seu papel na corporação e não em quem ele é. Além disso, os modelos ACL e CL são incapazes de garantir o requisito de privilégios mínimos em um sistema (*least privileges*) de segurança, isto é, ao sujeito será fornecido apenas os direitos necessários para que ele execute suas atividades e nada mais. Esses dois problemas ocasionaram o surgimento de um outro modelo de controle de acesso chamado *Role-Based Access Control Model* (RBAC) que será exposto a seguir.

2.1.4 Modelo de Controle de Acesso Baseado em Papéis

O RBAC – *Role-Based Access Control Model* (Modelo de Controle de Acesso Baseado em Papéis) (SANDHU et al., 1996); (OSBORN; SANDHU; MUNAWER, 2000) é um modelo que vem se destacando ao logo dos anos por ser uma alternativa as políticas discricionárias e mandatárias. Ele permite que autorizações possam ser dadas a sujeitos (como nas políticas discricionárias) e permite que restrições sejam impostas (como nas políticas mandatórias). Este modelo surgiu da dificuldade de representar as políticas de segurança em ambientes coorporativos, onde o papel do sujeito é um fator determinante.

As políticas baseadas em papéis *(roles)* controlam o acesso dos sujeitos com base nas atividades e responsabilidades dos mesmos dentro da corporação. Portanto, é necessário definir os papéis do sistema através do agrupamento de regras de autorização que representam alguma atividade. Diferentemente de especificar as autorizações para cada sujeito acessar um determinado objeto, as autorizações são especificadas para os papéis que por sua vez são associados aos sujeitos. Por exemplo, uma enfermeira deve ter somente os acessos as informações pertinentes a sua função, ela não deve ter as mesmas autorizações que um médico.

Os papéis determinam o que um sujeito pode acessar no sistema, ou seja, o sujeito está autorizado a executar todos os acessos permitidos ao seu papel. Muitas vezes um sujeito pode ter mais de um papel na corporação. O trabalho proposto por (SANDHU; FERRAIOLO; KUHN, 2000) trata isso, permitindo múltiplos papéis simultaneamente. O modelo *Temporal Role-Based Access Control* TRBAC proposto por (BERTINO et al., 1998); (BERTINO; BONATTI; FERRARI, 2001) é uma extensão baseada no RBAC para tratar restrições temporais, ele permite expressar regras do tipo “o gerente pode acessar o terminal X no horário de trabalho, entre 8 horas e 12 horas”.

2.1.5 Modelo de Controle de Acesso Muralha da China

Até agora não foi mostrado nenhum modelo que, baseado em requisições anteriores, permitisse ou não uma nova requisição de acesso. Por enquanto os modelos estudados são incapazes de armazenar estados, isto é, uma decisão para conceder um acesso não afeta futuras decisões. O modelo Muralha da China - *(Chinese Wall)* de (BREWER; NASH, 1989) foi proposto com o objetivo de evitar o fluxo de dados entre áreas com conflitos de interesses. Quando um sujeito acessa um objeto de uma área, ele não poderá acessar de outra área que seja conflitante com a anterior. É fácil ver a necessidade de armazenar o estado do modelo.

Este modelo foi bastante inovador e é baseado na seguinte organização:

- **Objetos**: são itens individuais contendo informações (arquivos) referentes a uma única companhia;
- **Conjuntos de dados corporativos**: são grupos de objetos de uma mesma companhia;
- **Classes de interesses conflitantes**: são conjuntos de dados corporativos referentes a companhias competidoras.

A Figura 4 ilustra um exemplo e seus agrupamentos onde temos duas classes de conflito de interesse. Em cada classe, temos duas companhias: A e B na primeira classe, C e D na segunda. Elas competem entre si, e por isso, devem proteger os dados dos competidores.

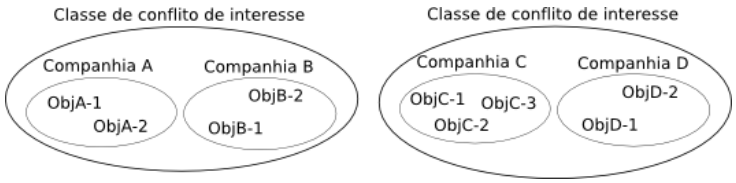


Figura 4: Exemplo de uma organização de objetos

Com essa organização, o modelo Muralha da China aplica suas restrições baseado nas seguintes propriedades:

- **Propriedade Simples**: um sujeito ‘s’ pode ler o objeto ‘o’ somente se as seguintes regras são satisfeitas:
  - O objeto ‘o’ está no mesmo conjunto de dados que alguns objetos previamente lidos pelo sujeito ‘s’ (isto é, o objeto ‘o’ pode “fazer parte do lado de dentro da muralha”) ou
  - O objeto ‘o’ pertence a uma classe de conflito de interesse dentro da qual o sujeito ‘s’ não tenha lido qualquer objeto.

- **Propriedade-***: um sujeito ‘s’ pode escrever no objeto ‘o’ somente se as seguintes regras são satisfeitas:
  - O sujeito ‘s’ pode ler o objeto ‘o’ pela propriedade simples e
  - Nenhum objeto ‘o’ pode ser lido quando estiver em um conjunto de dados diferentes daquele para o qual o acesso de escrita é requerido e contém informação que identifique sua origem (informação não sanitizada).

Enquanto a Muralha da China fornece um novo aspecto para o controle de acesso, o estado armazenado pelo modelo é muito limitado. Nós entendemos que uma capacidade mais geral para fornecer esse estado seja mais útil. No artigo (SANTOS; KEMMERER, 1999) os autores descrevem um framework para ACLs complexas com o SAC *(Secure Areas of Computation)* e no artigo (SANTOS; KEMMERER, 2000) é descrito uma forma segura de computação móvel, nesse modelo primeiramente foi introduzido a utilização de contadores para criar um modelo de controle de acesso baseado em estados anteriores. Com o uso de contadores, um nível mais alto de granularidade poderá ser armazenado em comparação com a Muralha da China. O SAC foi projetado para aplicações móveis como no uso de cartões inteligentes nos bancos, livrarias, locações de filmes e outros ambientes similares onde as transações poderão ser permitidas ou negadas baseadas em ações prévias.

2.1.6 Modelo de Controle de Acesso com criação dinâmica de regras

Os modelos tradicionais de controle de acesso geralmente exigem que um administrador seja responsável pela criação de regras de autorização. Ele decide quais acessos são permitidos a um determinado sujeito, podemos definir tais modelos como estáticos, pois eles ainda precisam da intervenção humana. Note que os ambientes em que tais modelos são utilizados não são necessariamente estáticos. Portanto, a manutenção de ambientes dinâmicos irá exigir um trabalho bastante árduo.

O modelo proposto por (WOO; LAM, 1993), o DACM *(Dynamic Access Control Model)* utiliza técnicas de orientação a objeto para descrever o modelo de controle de acesso. As regras são criadas dinamicamente a partir de informações armazenadas em tabelas persistentes. Entretanto, essas tabelas ainda são mantidas por um administrador, ou seja, para a criação de novas regras, deve-se alimentar tais tabelas que, dinamicamente, irão gerar novas regras. Esse não é o nível de dinamicidade que buscamos: a capacidade de regras criarem novas regras. Esse tipo de criação dinâmica de regras abre novos paradigmas para modelos inteligentes de controle de acesso que serão fundamentais para a computação pervasiva que será descrita no próximo capítulo.

2.2 Modelos de Controle de Acesso com políticas complexas

Os modelos tradicionais pecam pela sua expressividade, isto é, eles são incapazes de aplicar o controle de acesso para políticas mais complexas. O modelo proposto por (MILLER; BALDWIN, 1989) baseia-se no mecanismo de avaliar expressões booleanas (BEE) *Boolean Expression Evaluation*. Bastante flexível, ele permite a construção de regras mais complexas a partir de atributos associados aos sujeitos e objetos. Por exemplo, um objeto só pode ser acessado por um funcionário da empresa residente em Fortaleza. Essas características são expressas através dos atributos e avaliadas como uma expressão booleana. Esse modelo é mais flexível do que os modelos que utilizam ACL, CL, RBAC e TRBAC, pois associa características ao sujeito que poderão ser usadas como literais em expressões booleanas para determinar o acesso. Este método tem a facilidade de modelar regras complexas, porém possui limitações devido suas expressões serem baseadas apenas em um conjunto pequeno de características do perfil do sujeito.

A Figura 5 mostra como funciona o BEE em relação às CLs e ACLs.

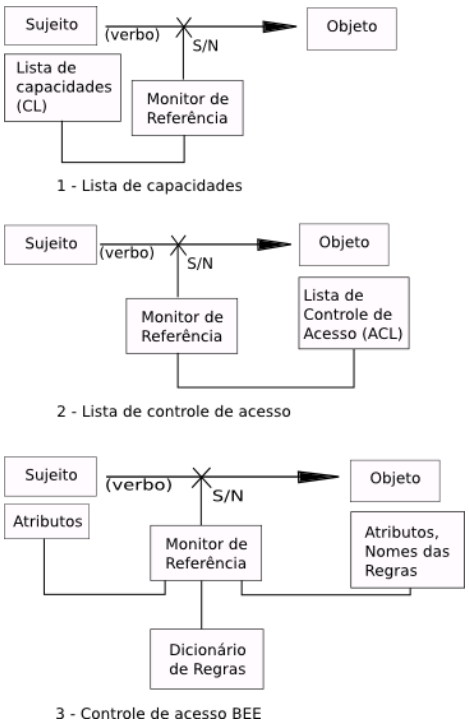


Figura 5: Mecanismo de Acesso BEE

O modelo GTRBAC *(Generalized Temporal Role-Based Access Control)* Controle de Acesso Baseado em Papéis Temporal e Generalizado foi proposto por (JOSHI, 2003), ele generaliza o modelo TRBAC (*Temporal Role-Based Access Control*) de (BERTINO; BONATTI; FERRARI, 2001). O GTRBAC estende o RBAC básico com a inclusão de uma linguagem para especificar várias restrições temporais em papéis, incluindo limitações para os tempos em que podem estar habilitados ou ativos, e também restrições para as relações usuário-papel e

papel-autorização. O estado habilitado do papel é condição por um usuário numa sessão. Por exemplo, a restrição {HorárioDiurno, enable MédicoAuditor} determina que o papel Médico-Auditor somente está habilitado (pode ser ativado) no horário diurno, por exemplo entre 06:00 e 18:00 horas.

A especificação da política de controle de acesso requer o uso de uma linguagem para especificar as regras de controle de acesso, bem como as propriedades das diferentes entidades do sistema (por exemplo, sujeitos/objetos ou propriedades) (SAMARATI; VIMERCATI, 2001). Segundo (VIMERCATI et al., 2007), as linguagens apropriadas para políticas complexas devem possuir as seguintes características:

- Uma linguagem de controle de acesso deve ser simples e expressiva. Deve ser simples para tornar fácil a tarefa de gerenciar a instalação e manutenção das especificações de segurança e deve ser expressiva para tornar possível especificar de forma flexível, maneiras diferentes de requisitos de proteção, que podem precisar ser impostos a diferentes objetos.

- Uma linguagem de controle de acesso deve suportar regras de acesso (autorizações) que serão referenciadas para acessos específicos, proporcionando uma refinada referência aos sujeitos e objetos do sistema. Mais precisamente, a linguagem deve fornecer suporte para autorizações previstas para grupos de usuários, grupos de objetos e, possivelmente, até mesmo grupos de ações.

- Os requisitos de proteção podem depender da avaliação de algumas condições (por exemplo, condições que tornam o acesso dependente da informação a ser acessada). Uma linguagem de controle de acesso deve então, permitir a especificação de restrições genéricas sobre sujeitos, objetos e sobre a informação dentro do contexto.

- Uma linguagem de controle de acesso deve apoiar a definição de diferentes tipos de regras de acesso. Tradicionalmente existem regras de acesso que especificam os acessos que não devem ser permitidos, e regras de acesso que especificam os acessos que devem ser permitidos.

- Uma linguagem de controle de acesso deve apoiar a definição de políticas administrativas que regulam a especificação de regras de acesso, ou seja, definir quem pode adicionar, excluir ou modificá-las.

Algumas linguagens foram propostas por (DAMIANOU et al., 2001); (JAJODIA; SAMARATI; SUBRAHMANIAN, 1997); (WOO; LAM, 1993); (TWIDLE et al., 2008) para especificar políticas de segurança. Essas possuem meios de criar diversas políticas para os diferentes tipos de mecanismos de controle de acesso. Assim como o BEE, tais linguagens são capazes de

expressar regras simples e regras complexas. Por possuir tal flexibilidade e por serem bastante robustas, as linguagens de especificação de políticas desempenharam um importante papel na definição do nosso modelo.

3 NOVOS PARADIGMAS PARA O CONTROLE DE ACESSO

Este capítulo descreve os conceitos relacionados a Computação Pervasiva ou Ubíqua. Neste ponto vale ressaltar que em todas as referências pesquisadas, o conceito de Computação Ubíqua e Computação Pervasiva foram os mesmos, deste modo, os dois conceitos são tratados como iguais. Tratamos também do conceito de Sensibilidade ao Contexto que é bastante útil na tomada de decisão do controle de acesso na Computação Ubíqua ou Pervasiva. Finalizamos o capítulo exemplificando uma aplicação que o utiliza o RBAC *(Role-Based Access Control)* com sensibilidade ao contexto em uma ambiente hospitalar. Os conceitos apresentados aqui, foram levados em consideração na construção do Modelo SACM.

3.1 Computação Pervasiva ou Ubíqua

A computação pervasiva ou ubíqua é uma forma de prover acesso a informação em qualquer lugar de forma transparente para o usuário. Os dispositivos pervasivos são elementos computacionais capazes de serem invisíveis, pois podem está acoplados aos objetos do cotidiano de tal maneira que se tornam indistinguíveis dos mesmos. Dessa maneira, as aplicações pervasivas podem prover serviços aos seus usuários de maneira personalizada. Nas palavras de Mark Weiser: “A mais profunda das tecnologias são aquelas que se desvanecem. Estas são envolvidas na concepção da vida do dia-a-dia de tal modo a não se distinguirem desta.” (WEISER, 1991).

A missão de um sistema pervasivo é de prover aos seus usuários, serviços relacionados às tarefas diárias executadas nos ambientes em que se encontram, sem que os mesmos precisem realizar de maneira consciente solicitações e configurações. Para tanto é de fundamental importância para esse tipo de aplicação perceber as características dos usuários, bem como dos ambientes e seus recursos, para que possa adaptar seus serviços da melhor forma possível. Sem o uso dessas informações, os sistemas pervasivos são capazes apenas de executar um conjunto padronizado de serviços com configurações predefinidas, o que pode não ser suficiente em termos de qualidade dos serviços providos aos usuários finais. Para possibilitar tal dinamismo, foi incluído o conceito de sensibilidade ao contexto na computação pervasiva.

3.2 Conceito de Sensibilidade ao Contexto

O objetivo principal dos sistemas cientes de contexto é realizar adaptações dos serviços ou tarefas oferecidos, de acordo com as necessidades e características de seus usuários e do ambiente de execução. A característica de regras poderem criar ou remover regras que pertencente ao SACM faz com que ele se encaixe com facilidade ao conceito de sensibilidade

ao contexto, pois baseado em um acontecimento que poderá ser enviado ao SACM via sensores serão criadas ou removidas regras.

A noção de contexto em aplicações de computação pervasiva refere-se à caracterização das condições ambientais e situações do mundo real que são relevantes para a realização de tarefas apropriadas no domínio de computação para seu comportamento desejado. O contexto de uma pessoa é definido baseado em sua localização atual, os dispositivos que estão em uso, a rede em que os dispositivos estão conectados e as atividades que o usuário está envolvido.

A premissa básica nesse caso é de que sistemas pervasivos cientes de contexto modificam constantemente suas características. Devido a isso, é necessária uma natureza dinâmica que garanta a adaptabilidade do comportamento da solução de acordo com a necessidade.

A proposta dessa área é, em linhas gerais, elaborar uma maneira de coletar para os dispositivos computacionais, entradas capazes de repetir as condições atuais do usuário, do ambiente no qual o mesmo se encontra e do próprio dispositivo computacional utilizado, considerando tanto suas características de hardware, software e de comunicação. Tais entradas são os chamados de contextos.

Um dos primeiros autores que definiram a computação ubíqua foi Mark Weiser em (WEISER, 1991). Para ele, a computação ubíqua deverá estar baseada na utilização de diversos tipos de dispositivos, cada um deles com capacidades, dimensões e objetivos diferentes. Em particular, Weiser propõe três classes de equipamentos ubíquos:

- **Pads**: dispositivos leves, pequenos, portáteis, com dimensões medidas em polegares;
- **Tabs**: dispositivos intermediários, similares a livros, cadernos ou folhas de papel;
- **Boards**: dispositivos grandes e compartilhados, utilizados para realização de reuniões e palestras.

A interação direta entre os próprios dispositivos ubíquos é apontada por ele como o principal poder da computação ubíqua.

Na tese de doutorado de (BRAGA, 2010) a autora citou algumas características das aplicações ubíquas e suas particularidades como descrito a seguir:

- Dispositivos com recursos limitados: os dispositivos ubíquos, em geral, apresentam restrições de recursos, tais como capacidade de processamento, disponibilidade de memória permanente e volátil, capacidade da fonte de energia, alcance e qualidade da comunicação de dados. Essa característica se deve, principalmente, às limitadas dimensões que tais equipamentos podem apresentar, bem como ao fato de os mesmos serem projetados para serem embutidos aos elementos do cotidiano;

- Computação invisível: de acordo com a definição estabelecida por Mark Weiser, para a computação ubíqua os computadores deveriam estar tão integrados ao ambiente das pessoas, que seriam utilizados pelas mesmas sem serem notados. Dessa forma, a transparência é considerada característica básica desse tipo de sistema. Entretanto, ainda hoje, existe grande dificuldade no tratamento desse desafio, visto que o paradigma corrente aplicado a utilização de sistemas computacionais requer aprendizado específico e grande atenção por parte dos usuários;

- Segurança e Privacidade: esses dois aspectos ainda são considerados desafios para a computação em geral. Entretanto, em ambientes ubíquos as dificuldades são ainda maiores, visto que tais sistemas utilizam informações sensíveis sobre seus usuários, dispositivos e ambientes físicos, quais sejam os contextos relacionados aos mesmos. Além disso, as soluções de segurança devem ser elaboradas de tal forma a respeitar as demais características das aplicações ubíquas.

3.3 Aquisição das informações de contexto

Uma vez que os dados contextuais são elementos básicos para o funcionamento de sistemas ubíquos cientes ao contexto, é bastante importante que a aquisição dos mesmos seja realizada da melhor forma possível junto ao ambiente, usuário e dispositivos utilizados. Muitos tipos diferentes de contextos podem ser utilizados pelas diversas aplicações cientes ao contexto possíveis de serem implementadas. Além disso, para cada um desses tipos, diferentes técnicas podem estar disponíveis, variando de acordo com o consumo de recursos, precisão, disponibilidade, fontes contextuais utilizadas, dentre outros aspectos.

As informações de contexto podem ser coletadas com a utilização de sistemas baseados na localização. Vários trabalhos publicados na área da computação ubíqua ciente ao contexto, consideram especificamente um tipo de dado contextual: a localização. As aplicações definidas por esses trabalhos realizam adaptações baseadas apenas nas informações sobre a localização dos dispositivos ou de seus usuários e são chamadas de sistemas baseados em localização *(Location Based Systems - LBS)* (BRAGA, 2010).

Tais sistemas são baseados em geral, na emissão de sinais de radiofrequência ou ultrassônicos, captados por uma malha de receptores espalhados pelo ambiente de interesse. Em particular, no artigo *The active badge location system* do autor (WANT et al., 1992) é apresentado um sistema chamado *Active Badge*, no qual visitantes e trabalhadores de uma organização utilizam crachás capazes de emitir sinais infravermelhos periódicos contendo um identificador único. Esses sinais são captados por antenas receptoras posicionadas no teto e armazenados em um sistema central de localização. O sistema informa para cada usuário qual é a sua localização interna dentro do prédio da organização.

3.4 Exemplos de Aplicações Ubíquas Sensíveis ao Contexto

Existem na literatura científica vários trabalhos descrevendo diferentes aplicações ubíquas cientes de contexto. Tais sistemas utilizam dados contextuais coletados junto a seus usuários e/ou ao ambiente físico por eles compartilhado, como entradas para adaptar uma ou mais características dos serviços ofertados, tais como seus parâmetros e ordenação.

O atendimento médico representa uma das classes de aplicações que podem utilizar os recursos da computação ubíqua e ciente de contexto para melhor adaptar as ações realizadas de acordo com a necessidade dos usuários (pacientes). Os sistemas utilizados podem ajudar sugerindo as melhores práticas a serem adotadas por médicos, enfermeiras e demais profissionais da área, ou ainda identificando a ordem em que os atendimentos poderiam ocorrer. A utilização de materiais e insumos, tais como máquinas para exames, remédios e equipamentos médicos, por exemplo, também pode ser controlada por meio de um sistema desse tipo.

O Controle em Ambientes é a classe de aplicações que descreve ambientes físicos ubíquos capazes de identificar usuários, determinar as características e necessidades relacionadas aos mesmos, e adaptar de maneira correspondente os serviços ofertados. Existe na literatura sistemas desse tipo ligados a ambientes de escritórios, conferências, salas de aula, dentre outros.

O Controle de Acesso Baseado em Papéis Sensíveis ao Contexto apresentado por (KULKARNI; TRIPATHI, 2008) é um exemplo bastante interessante de como unir a controle de acesso e a sensibilidade ao contexto com o objetivo de tornar as decisões mais dinâmicas em um modelo.

Com o surgimento da era da computação pervasiva, vários usuários utilizam diversos serviços simultâneos e inúmeros tipos de recursos e informações são de propriedade pública. Nestes ambientes, o controle de acesso que concede permissões de acesso a um usuário autorizado, torna-se um fator essencial. Entre as políticas de controle de acesso, o controle de acesso baseado em papéis (RBAC), que fornece as permissões de acesso aos papéis ao invés de usuários, é um modelo de controle de acesso que é amplamente utilizado devido a sua flexibilidade e eficiência.

Em ambientes de computação pervasiva, onde o estado dos usuários e as informações de contexto são coletados em tempo real e mudam de forma dinâmica, a utilização de vários tipos de informações de contexto para a adequação dos papéis ou modificação de permissões é definitivamente necessário para um controle de acesso dinâmico. Portanto, os papéis de usuários e permissões de funções devem ser alterados dinamicamente de acordo com mudanças no contexto.

Considerem um sistema de informações de pacientes implantado em um hospital que

pode ser acessado pelas Enfermeiras e Médicos. O sistema suporta a atribuição de uma Enfermeira para uma determinada enfermaria durante um período de tempo. Durante este período a Enfermeira trabalhará com a capacidade de um papel chamado enfermeira-de-plantão nessa enfermaria. Nesta função, o sistema permite o acesso somente aos registros daqueles pacientes que estão internados na enfermaria onde a Enfermeira está presente. A associação da Enfermeira no papel enfermeira-de-plantão é revogada quando ela sai da enfermaria ou após o término de seu tempo de serviço.

O sistema permite aos Médicos criarem diferentes tipos de relatórios sobre seus pacientes. Para uma Enfermeira, o acesso aos relatórios médicos será permitido somente se algum Médico estiver presente na enfermaria ao mesmo tempo que ela. Pode acontecer de uma Enfermeira iniciar o acesso aos relatórios médicos, enquanto o Médico está presente na ala, mas se o Médico deixar a ala enquanto ela estiver acessando, o acesso a esses relatórios por parte da Enfermeira deve se encerrado. Isto assegura que uma Enfermeira não continue a acessar os relatórios na ausência de um Médico.

Para facilitar a implementação dos modelos capazes de funcionar de acordo com os requisitos exigidos pela Computação Pervasiva ou Ubíqua são utilizadas as Políticas de Linguagens para proporcionar uma maior expressividade do modelo, com base nisso o próximo capítulo apresenta o Ponder2, uma Linguagem de Política desenvolvido pelo Colégio Imperial de Londres e que suporta esses novos paradigmas.

4 O PONDER2

O Ponder2 é uma linguagem de política que foi concebida de forma a se adequar com os novos paradigmas de segurança apresentados no Capítulo 3. Ele surgiu a partir da remodelagem do Ponder (DAMIANOU et al., 2001), modelo que era voltado para ambientes tradicionais e devido o surgimento da computação pervasiva. Ele teve que se adequar a essa nova situação surgindo assim o Ponder2. Ambas as linguagens serviram de base para a concepção do SACM, desse modo, vamos fazer uma breve descrição sobre o Ponder2 por se tratar do modelo que mais se aproxima do nosso.

4.1 O que é o Ponder2

Segundo (TWIDLE et al., 2008) o Ponder2 compreende um sistema de gerenciamento de objetos de propósito geral com passagem de mensagens entre os objetos. Ele incorpora eventos e políticas implementando um mecanismo de execução dessas políticas, tem uma configuração de alto nível com uma linguagem de controle chamada PonderTalk, enquanto os objetos que nele são chamados de objetos gerenciados são desenvolvidos em JAVA.

O Ponder (DAMIANOU et al., 2001) foi um ambiente político muito bem-sucedido usado por muitos na Indústria e nas Universidades. No entanto, sua concepção sofreu com algumas das mesmas desvantagens com base em políticas para ambientes tradicionais. Seu desenho era dependente do suporte de infraestrutura centralizada, como para os diretórios LDAP. Além disso, ele não era dimensionado para dispositivos menores como é necessário em sistemas pervasivos. Embora mantendo alguns conceitos que foram responsáveis pela popularidade do Ponder, o Ponder2 foi todo redesenhado para se adaptar aos novos paradigmas e assim atingir os seguintes objetivos:

- **Simplicidade**: O projeto do sistema deve ser simples e incorporar o mínimo de elementos possíveis;
- **Extensibilidade**: Deve ser possível dinamicamente estender no sistema de politica, novas funcionalidades para interagir com novos serviços de infraestruturas e gerir novos recursos.
- **Auto-contenção**: O ambiente de política não deve confiar na infraestrutura existente, ele deve conter todos os elementos necessários para aplicar as políticas para o gerenciamento dos recursos.
- **Facilidade de uso**: O ambiente deve facilitar a utilização de políticas em novos ambientes e a prototipagem de novos sistemas de políticas para diferentes aplicações.

- **Interatividade**: Deve ser possível para os gestores e desenvolvedores interagir de forma fácil com o sistema de política e os objetos gerenciados, emitir comandos para os objetos gerenciados e criar novas políticas.
- **Escalabilidade**: O sistema de segurança deve ser executável em recursos limitados, tais como PDAs, telefones celulares e sensores, assim como nos sistemas mais tradicionais.

O Ponder2 é implementado com uma célula de autogestão (*Self-Managed Cell* - SMC). O SMC é definido como um conjunto de hardware e software que forma um domínio administrativo que é capaz de funcionar de forma autônoma e de se autogerir. Os serviços de gerenciamento interagem uns com os outros através de eventos assíncronos propagados através de um barramento de eventos baseado em conteúdo. As políticas fornecem uma adaptação livre de loops locais, assim os objetos gerenciados geram eventos, as políticas respondem e executam as atividades de gestão sobre o mesmo conjunto de objetos gerenciados sem que essas ações fiquem sendo executadas indefinidamente.

4.2 Os objetos Gerenciáveis

Tudo no Ponder2 são objetos gerenciados. Os objetos gerenciados incluem políticas de gestão e adaptadores para objetos do mundo real, tais como sensores, alarmes, interruptores etc. Os objetos básicos do Ponder2 incluem Eventos, Políticas e Domínios, cabendo ao usuário criar ou reutilizar objetos gerenciados para outros fins. Os objetos gerenciados, incluindo todos os mencionados acima devem ser carregados dinamicamente para o SMC, assim, produzindo um objeto gerenciado de fábrica.

4.3 As Políticas

As políticas definem os objetivos da gestão do sistema e os eventos que desencadeiam reações das políticas a fim de lidar com eles. Existem atualmente dois tipos de políticas básicas no Ponder2: Políticas de Obrigação e Políticas de Autorização.

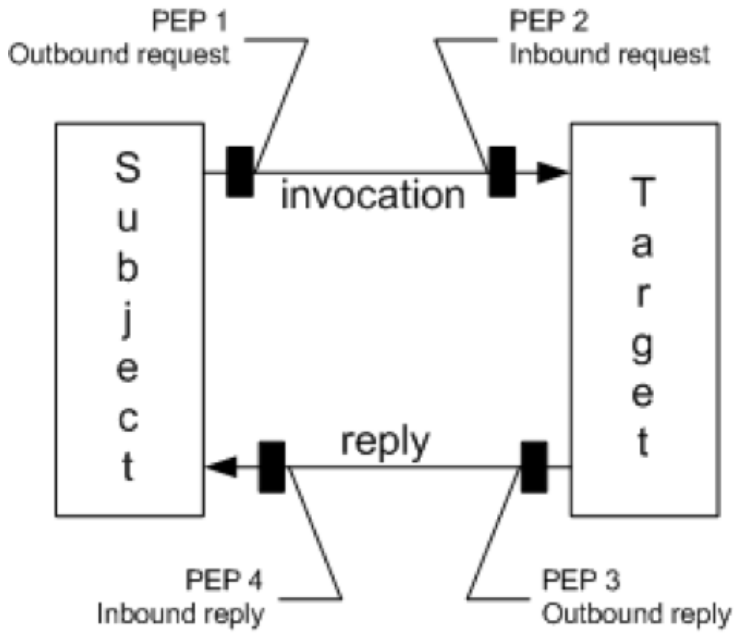
As **Políticas de Obrigação** especificam as ações que devem ser executadas pelos gestores dentro do sistema, quando certos eventos ocorrem e fornecem a capacidade de responder a novas circunstâncias. Por exemplo, as políticas de gestão de segurança especificam quais ações devem ser realizadas quando ocorrem violações de segurança e quem ou o que deve executar essas ações. Eles também podem especificar que atividades de auditoria e registro devem ser realizadas num dado momento e por alguém específico. As Políticas de Obrigação são eventos acionados e também são conhecidos como políticas Ação Condição Evento *(Event Condition Action – ECA)* ou seja, ela recebe um evento, avalia se uma ou mais condições são verdadeiras,

e se todas passarem, ela executa uma ou mais ações. Os valores mantidos dentro do evento podem ser utilizados para avaliar as condições e quando a execução das ações.

As **Políticas de Autorização** permitem ou negam passagem de mensagens entre os objetos gerenciados. Elas definem as atividades que um membro do domínio sujeito (chamador) pode executar sobre o conjunto de objetos no domínio de destino (receptor). Essas são políticas essencialmente de controle de acesso para proteger os recursos e serviços do acesso não autorizado. A política de autorização positiva define as ações que os indivíduos estão autorizados a exercer sobre os objetos. Na política de autorização negativa é especificada as ações que os indivíduos são proibidos de executar em objetos de destino. As políticas de Autorização são implementadas no host de destino por um componente de controle de acesso denominado PAF *Ponder2 Autorization Framework* que será descrito na próxima subseção.

4.3.1 Framework de Autorização do Ponder2

O Ponder2 suporta as Políticas de Autorização para controlar as interações entre os objetos gerenciados. O Framework de Autorização do Ponder2 (PAF) introduz novas ideias sobre o controle das políticas de autorização. Em particular, com o PAF as políticas de autorização podem ser especificadas de maneira uniforme e executadas para proteger tanto o sujeito como o objeto de destino para uma determinada ação (PONDER2, 2012).



Fonte: (PONDER2, 2012)

Figura 6: Framework de Autorização do Ponder2

Como mostrado na figura 6, o PAF fornece 4 pontos de aplicação de políticas (Policy Enforcement Point - PEP):

- PEP1 e PEP4 são usados para reforçar as políticas de autorização no lado do sujeito;
- PEP2 e PEP3 são usados para reforçar as políticas de autorização no lado do objeto alvo;

Através da aplicação de políticas no PEP1, torna-se possível especificar as políticas de autorização que impedem o indivíduo de realizar ações que poderiam ser prejudiciais para eles ou ao seu domínio. Por exemplo, prevenir que um navegador web envie uma solicitação para um servidor web na lista negra. Além disso, a aplicação das políticas do PEP4 poderiam impedir um sujeito de aceitar uma resposta de uma ação que poderia ameaçar sua própria integridade.

Do lado do objeto, o PEP2 pode ser usado para impor políticas de autorização de controle de acesso tradicionais. Além disso, quando as diretivas de autorização são aplicadas, com o PEP3 torna-se possível proteger a privacidade do objeto que poderia ser comprometido quando o resultado de uma ação contém informações que não devem ser reveladas (por exemplo, pela aplicação de uma política de autorização que filtra dados sensíveis a partir do resultado).

O PAF suporta políticas de autorização negativas e positivas. Quando duas ou mais políticas de sinal contrário se aplicam a mesma ação pode causar conflitos. Por esta razão, o PAF fornece uma estratégia de resolução de conflitos (com base no domínio de precedência) que lida com tais conflitos em tempo de execução.

Para que as autorizações possam ser executadas, se faz necessário o usa de regras que são escritas usando uma linguagem de programação chamada PonderTalk que será explicada a seguir.

4.4 O Pondertalk

Os desenvolvedores do Ponder2 precisavam de uma linguagem de alto nível para configurar e controlá-lo. Esta linguagem tinha que ser geral, mas também com política amigável e tinha que permitir enviar mensagens para objetos gerenciados localmente. Depois de pensar sobre a maneira que as mensagens deveriam ser enviadas para objetos gerenciados, eles perceberam que a linguagem Smalltalk se adequava a essa necessidade e foi decidido adaptá-lo aos propósitos do Ponder2. Não foi necessário pegar os conceitos de definição de classes porque os objetos são escritos em Java, assim foi utilizado apenas os aspectos de transmissão de mensagens, alguns dos tipos padrão do Smalltalk e acrescentado um pouco na sintaxe nomeando a nova linguagem de PonderTalk.

O PonderTalk essencialmente identifica um objeto gerenciado e envia mensagens com argumentos opcionais. O PonderTalk tem o conceito de variáveis que podem ser usados no lugar de nomes de caminho para identificar objetos gerenciados. Os comandos são da forma:

```
pathname message(s).
```

```
myVar := pathname.

myVar message(s).
```

4.4.1 As Politicas no Pondertalk

As políticas descritas na Seção 4.3 são importadas, criadas e usadas na mesma maneira como qualquer usuário escreve um objeto gerenciado, mas têm funções especiais dentro do sistema Ponder2 quando são ativadas.

Nas políticas de obrigação são emitidas mensagens para configurar o evento, as condições e ações. As condições e ações estão de fato em blocos PonderTalk que a política é executado quando necessário. Quando ativada, a política é anexada ao barramento de evento de domínio e envia eventos pelo mecanismo de evento interno, sempre que o evento apropriado é criado. Quando a política recebe um evento, ela executa todos os blocos de condição enviando-lhes o evento como uma mensagem. Se o resultado for verdade, então a política executa a ação de bloqueia de uma maneira similar.

```
Policy := root/factory/ecapolicy create.

Policy event: myEvent;

condition: [ :arg | bool-expression ];

action: [:arg | statements ]
```

As Políticas de Autorização autorizam o envio e recebimento de mensagens. Essas políticas têm objetos de origem e destino e a operação que deve ser permitida ou negada. Em termos de um sistema distribuído, eles governam: a mensagem que deixa o objeto de origem; a mensagem chegando ao objeto de destino; a resposta deixando o objeto de destino e a resposta chegando ao objeto de origem. No exemplo a seguir, em uma ala1 de enfermeiras é dado permissão para ler os registros de pacientes da ala1. O foco da política informa que está protegendo o objeto de destino.

```
Policy := (root/factory/authpolicy

subject: root/personnel/nurse/ward1
```

```
action: "getrecord"

target: root/patient/ward1

focus:"t" .
```

4.5 Objetos Gerenciáveis utilizados no Ponder2

A funcionalidade do sistema Ponder2 é bastante reforçada pela introdução da escrita pelos usuários dos Objetos Gerenciados. O Java foi escolhido devido ser amplamente conhecido e ter uma extensa biblioteca de sistema disponível. Uma solução ideal seria criar automaticamente um código de mapeamento que levasse as mensagens e argumentos PonderTalk, organizando os tipos de argumento e chamando um método Java dentro do objeto gerenciado. Isso significaria a criação de um código stub em tempo de execução. Anotações no Java da forma @*annotation* (argumentos) são utilizadas.

Quando o compilador se deparar com tal anotação, em nível de usuário, o código de fábrica é chamado com a estrutura da classe ou método associado com a anotação a ser disponibilizados pelo compilador Java. Dessa forma, nomes de mensagem PonderTalk são mapeados por anotações para os métodos dentro do objeto gerenciado. As anotações são colocadas acima dos métodos Java e são disponibilizados para as extensões do compilador em tempo de execução. As extensões do compilador têm acesso completo aos tipos de parâmetro do método que permite aos códigos *stub* serem gerados para executar o mapeamento entre o formato da mensagem genérica e as *strings*, inteiros e etc, exigidos pelo método em questão. Por exemplo, para criar um objeto gerenciado que aceita a mensagem de palavra-chave do Pondertalk *at:put* que armazena um nome com um valor inteiro, como: *myObjat: max put: 1000* podemos escrever o seguinte método:

```
@Ponder2op("at:put:")

public void store(String name, int value) {

...

}
```

Para ser um objeto gerenciado, o objeto deve implementar uma *interface* vazia chamada *ManagedObject*. Esta *interface* simplesmente diz ao compilador Java que esse será um objeto Gerenciado Ponder2 e deve começar a criar um código adaptador Java para realizar os mapeamentos necessários a partir do PonderTalk aos métodos Java.

Se quisermos enviar uma mensagem para um objeto gerenciado, por exemplo, para definir o nome e idade de uma pessoa, isso pode ser feito no PonderTalk como:

```
myobject name: Fred age: 24.

*/O código Java seria simplismente:*/

@Ponder2op("name:age:")

public void setInfo(String name, int age){

    this.name = name;

    this.age = age;

 }
```

Na próxima seção faremos uma breve comparação entre o Ponder e o SACM.

4.6 Uma comparação entre o Ponder2 e o SACM

Os modelos SACM e Ponder2 foram concebidos com propósitos semelhantes, ou seja, ambos foram pensados para ambientes com computação móvel, onde os equipamentos possuem recursos limitados, porém existem algumas diferenças importantes. Mesmo o Ponder2 sendo um modelo consolidado na comunidade científica existem duas características no SACM que não são apresentadas nele. São elas: a capacidade de guardar o estado e a capacidade de funcionamento totalmente descentralizado.

No artigo (LUPU et al., 2008) é mostrado a implementação de um sistema autônomo para cuidado da saúde chamado *AMUSE: autonomic management of ubiquitous e-Health systems* baseado no Ponder2. Ele consiste em uma série de equipamentos portáteis como PDAs ou *Smartphone* com o SMC em execução, que formarão a Rede de Sensores Corporal, com o objetivo de monitorar a saúde dos pacientes de um hospital e para auxiliar na tomada de decisão caso ocorra alguma anomalia com esses pacientes. O sistema é capaz também de funcionar fora do ambiente hospitalar, nas residências dos pacientes.

Para exemplificar, imagine uma enfermeira ou um médico de posse de um elemento dessa rede (SMC da enfermeira/médico), este poderá se comunicar com o elemento que monitora o paciente (SMC do paciente) que é capaz de aplicar medicamentos de forma autônoma se for necessário, baseado no estado da pessoa monitorada. Através desta comunicação o SMC da

enfermeira recebe todo o histórico de decisões do SMC do paciente e poderá atualizá-lo com novas decisões médicas para aprimorar os cuidados do paciente.

Essa implementação é bastante útil para essa situação específica, porém não ficou claro como o SMC tanto da enfermeira como do paciente se comportariam caso uma das decisões tomadas não surta o efeito desejado. Ele continuará tomando a mesma decisão anterior? Nos artigos pesquisados que retratam esse exemplo, não foi mencionada tal situação mostrando a deficiência da falta de tomada de decisão baseada em decisões prévias, pois se o sistema implementasse essa solução poderia tentar outra alternativa devido a ação inicial não ter obtido o resultado esperado.

Outro ponto interessante no exemplo exposto é que, mesmo os SMC sendo capaz de tomar decisões sem a intervenção do usuário, no caso da enfermeira ou do paciente, ele terá que atualizar as informações da base de dados do hospital para que possa saber como é o estado normal do paciente para que sirva de base para uma tomada de decisão. Seria impraticável que um equipamento como um PDA carregasse consigo as informações de todos os pacientes do hospital. Com o SACM, os nossos exemplos foram pensados para que o sistema possa trabalhar de forma totalmente descentralizada podendo até utilizar *Smart Cards* para armazenar as regras e os acessos prévios. As implementações baseadas no Ponder2, devido o SMC, precisam de uma capacidade de armazenamento maior do que o nosso modelo.

O Ponder2 possui um ponto importante que é diferente do SACM: o modo que as regras são avaliadas. No SACM utilizamos uma varredura de cima para baixo, caso em algum momento da avaliação seja retornado um valor de *token* negativo a regra é bloqueada, já no Ponder2 a tomada de decisão é baseado na posição da regra, ou seja, as regras mais específicas possuem precedência sobre as mais gerais, porém isso pode ser mudado dependo da vontade do administrador.

Apesar de o SACM ser um modelo ainda em desenvolvimento, suas ideias poderão contribuir muito para que modelos mais consolidados como o Ponder2 possam ser também aperfeiçoados.

5 SACM

O SACM é um modelo concebido com o intuito de se adequar as necessidades dos sistemas de controle de acesso centralizados assim como com as necessidades dos sistemas descentralizados, onde as informações que servirão para a decisão de autorização serão carregadas pelo próprio usuário em um dispositivo com um espaço de armazenamento seguro.

O modelo possui uma característica importante que facilita sua utilização em ambientes sensíveis ao contexto, onde regras podem criar ou remover novas regras. Como exposto na Capítulo 3, o ambientes pervasivos em sua maioria são dinâmicos e a sensibilidade ao contexto veio para auxiliar a adaptação do controle de acesso com a mudança no ambiente sem a intervenção do administrador. A grande vantagem é que, baseado nas informações recebidas através do contexto, o SACM poderá criar ou deletar regras de forma dinâmica se adequando rapidamente as mudanças do ambiente. Nesta Seção é descrito as principais características do SACM.

5.1 Visão Geral

Devido ao propósito do SACM de expressar regras que dinamicamente criam novas regras para se adequar aos novos paradigmas da computação, ele foi baseado nas características das linguagens de especificação de políticas apresentadas na Seção 2.2.

Um outro aspecto importante sobre o SACM é a maneira de organizar os dados. Ele pode ser tanto centralizado - dados são armazenados em um único lugar - ou descentralizado. A primeira abordagem é a mais comum e funciona muito bem para os ambientes computacionais tradicionais. Porém, essa abordagem exige muito esforço de um administrador para controlar todo o sistema. Além disso, ela não funciona bem para sistemas *offline*. Por exemplo, imagine uma empresa que possui estações de impressão espalhadas por aeroportos do mundo inteiro. Ter essas impressoras conectadas e prover um controle de acesso centralizado seria bastante custoso. Uma melhor solução é ter conexões *Bluetooth* ou *WiFi* permitindo a comunicação entre usuários e impressoras. Desta forma, as regras de controle de acesso seriam distribuídas armazenando-as em áreas seguras dos dispositivos móveis. No caso de não existir áreas seguras, esse ambiente computacional, como vários outros ambientes pervasivos ou com grande mobilidade, apresenta a característica de ser apropriado apenas onde o custo para violar essas áreas de armazenamento seja maior do que a própria informação ou serviço contido nessa área.

O SACM é centrado em torno do armazenamento de estado que se aplica aos sujeitos com base nos objetos. O modelo organiza suas informações em um conjunto de regras com respeito a um dado objeto e sujeito. Essas regras armazenam o estado através de *flags* e variáveis. Essa estrutura será explicada a seguir.

5.2 Estrutura do SACM

No modelo, cada sujeito ‘s’ possui um contador ‘T_o’ correspondendo a cada objeto ‘o’ que ele pode tentar acessar. Cada objeto ‘o’ possui também um conjunto de regras que operam sobre o contador ‘T_o’ de cada sujeito ‘s’. Quando um sujeito tenta acessar um objeto, as regras apropriadas são avaliadas. Isto é, apenas as regras apropriadas dentro do conjunto de regras referentes à tentativa de acesso do sujeito ao objeto são avaliadas.

Cada regra possui quatro elementos importantes: um contador *token*, duas *flags* e um conjunto de variáveis. A primeira *flag dirty* (sujeira) indica se a regra ainda é válida. A segunda, *auth* indica se o acesso foi autorizado ou não. No caso da *flag auth* não ser modificada para autorizar o acesso, este é autorizado se o *token* tiver um valor não negativo ou igual a zero. Caso contrário, ele é negado. A *flag auth* possui precedência sobre o contador de *token*. Além do uso dos *tokens* para a avaliação da requisição, as regras são capazes de usar variáveis definidas dentro das regras, ou argumentos passados na hora da requisição.

5.3 Hierarquia dos objetos

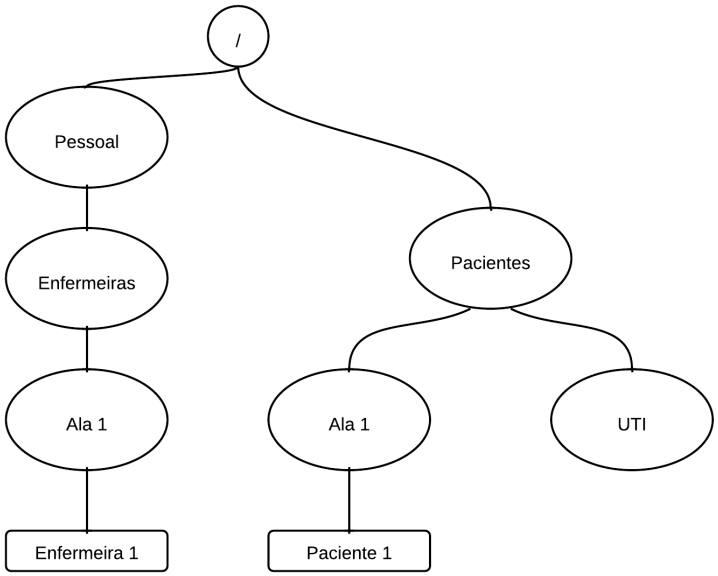


Figura 7: Hierarquia dos objetos

A organização dos objetos no SACM é bastante similar as linguagens de programação orientada a objetos. Em tais linguagens, as classes são organizadas hierarquicamente. Da mesma forma são organizados os objetos e serviços no SACM. Todo objeto recebe um nome e está hierarquicamente classificado, tendo como raiz um objeto do sistema. Cada objeto possui o seu conjunto de regras que podem expressar diferentes tipos de acesso. Por exemplo, em um sistema de aluguel de livros, podemos ter uma regra para alugar um livro e outra para dar “baixa”

no livro. Outro exemplo é uma regra para bloquear um usuário temporariamente e outra regra para desbloquear. O mesmo objeto pode ter regras para diferentes tipos de acesso. Essa organização permite expressar facilmente autorizações do tipo: “todos os clientes terão desconto de 20%”. Para tal, basta adicionar a regra para o desconto no objeto cliente. Essa generalização não impede a construção de autorizações mais restritivas. Por exemplo, podemos criar a regra “clientes do sexo feminino terão desconto de 30%” seguindo o caminho /clientes/mulheres na organização hierárquica. A Figura 7 ilustra a organização hierárquica do SACM baseado na implementação de Sensibilidade ao Contexto que será descrita no capítulo 7.

5.4 Autorizações e Avaliações

Para avaliar se uma requisição deve ser autorizada ou não, o SACM usa uma abordagem *TOP-DOWN*. Ele “varre” a hierarquia de objetos começando pela raiz e prossegue até atingir o destino, isto é, o objeto ao qual está sendo feita a requisição. Portanto, em cada nível da hierarquia as regras referentes a esta requisição são avaliadas. Caso não existam regras no caminho da raiz ao destino, o acesso é negado. Caso contrário, todas as regras do caminho são avaliadas. Se todos os *tokens* tiverem valores positivos então o acesso é autorizado. Basta um *token* com valor negativo ou igual a zero no caminho da raiz ao objeto para o acesso ser negado. Note que se a *flag auth* for definida (verdadeira ou falsa) então ela irá sobrepor-se aos *tokens* e determinar a autorização sem precisar continuar a avaliação.

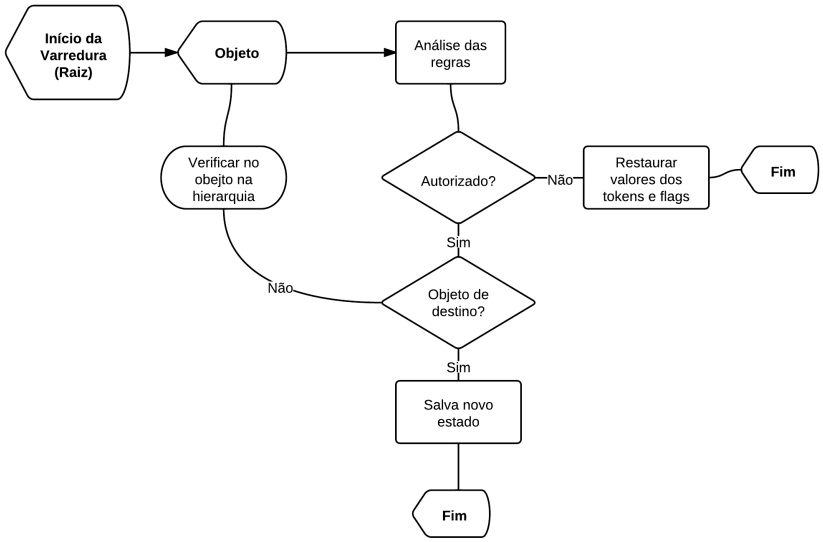


Figura 8: Processo de Varredura

É importante notar que durante o processo de “varredura” mostrado na Figura 8, os *tokens*, variáveis ou *flags* podem ser alteradas. Quando o acesso é negado, o sistema não deve mudar de estado, pois ele poderá ficar inconsistente. Portanto, as alterações não podem ser

feitas neste caso. Para evitar este problema as alterações são armazenadas e feitas apenas se a autorização for permitida.

5.5 Estrutura das regras

Os elementos mais básicos do SACM são os tipos de dados similar a uma linguagem de programação (C/C++) que têm como elementos básicos *int, float, char*, etc. Os tipos de dados são definidos a partir de três atributos: um conjunto de valores possíveis (domínio), o conjunto de comparações e o conjunto de operadores aplicáveis aos elementos do domínio.

DataTypes = (Domain, Comparisons, Operators)

Esses três conjuntos são customizáveis no sentido de que podem ser definidos pelo controlador do sistema. Ou seja, diferentemente das linguagens de programação que possuem tipos de dados básicos pré-determinados (inteiros, booleanos, etc), o SACM não o possui. Entretanto, seis tipos de dados são recomendados para prover a mínima funcionalidade do modelo: TokenData, AuthData, DirtyData, DateData, TimeData e RuleData.

- TokenData = (Integers,{<,≤,==,≠,≥,>},{+=,-=,*=,/=,=});
- AuthData = (Booleans,{==,≠},{=});
- DirtyData = (Booleans,{==,≠},{=});
- DateData = (Dates,{<,≤,==,≠,≥,>},{+=,-=,=});
- TimeData = (Times,{<,≤,==,≠,≥,>},{+=,-=,=});
- RuleData = (RuleClauses,{∃, ∄},{insert,remove}).

O *TokenData* é utilizado para representar os contadores do sistema. As comparações e operações recomendadas são as básicas da matemática. As operações funcionam como na linguagem C/C++ (e.g., x += 1 incrementa x em uma unidade) e o operador de atribuição "="complementa o conjunto de operações. O *AuthData* é utilizado para representar as decisões de autorização usando valores *booleanos*. As comparações só permitem verificar igualdade ou desigualdade (uma limitação das comparações é ter que definir o oposto, por exemplo se "="foi definido então ≠ também deve ser definido) e as operações incluem apenas o símbolo de atribuição "=". Da mesma forma é definido o *DirtyData*, porém ele serve para indicar se uma regra ainda é válida, isto é, se ela deve ser apagada ou não. *DateData* e *TimeData* servem para manipular datas e horários e funcionam da mesma forma que o *TokenData* excluindo os

operadores *= e /=. Os elementos do tipo *RuleData* serão detalhados na Seção 5.5.4 por serem um tipo especial de dado.

Esses tipos de dados usam diversas operações e comparações recomendadas. As comparações matemáticas padrão {<,≤,==,≠,≥,>} são complementadas com os avaliadores de existência {∃, ∄} para formar um conjunto de comparação booleana padrão. A única limitação sobre a forma como as comparações podem ser definidas é que, para cada comparação, o oposto lógico ou negação da condição deve também ser definido. Isto é, para definir ∃, ∄ também deve ser definido. Os operadores matemáticos {+=,-=,*=,/=,=} funcionam similarmente às construções da linguagem C/C++. Por exemplo, (x += 1) incrementa o valor de x em 1. Em conjunto com = (armazenar/setar), *insert* e *remove*, eles formam o conjunto de operadores padrão. Os operadores devem requerer um ou nenhum operando adicionalmente para o alvo, que será sempre o primeiro operando.

5.5.1 Condições

As condições são os elementos de comparação do modelo, é responsável por determinar se uma ação pode ou não ser executada. Através da avaliação de uma comparação entre dois atributos (fonte e alvo), uma condição é verdadeira ou falsa. Se for verdadeira, a ação será executada. Caso contrário, ela não é uma ação permitida pelo sistema e deve ser proibida.

```
ConditionModel = (DataType, DataSource)
Condition = (Model Comparison Target)
  Model ∈ ConditionModel
  Comparison ∈ Model.DataType.Comparisons
  Target ∈ Model.DataType.Domain∪{var_i|0 ≤ i ≤ n} ∪ {arg_i|0 ≤ i ≤ n}
```

O *ConditionModel* é o modelo de uma condição indicando o tipo e o que será comparado na condição. Ele é representa a forma geral e possui dois atributos: *DataType* que determina qual o tipo de dado (inteiro, booleano, etc) a ser comparado e o *DataSource* sendo o dado propriamente dito. A condição a ser verificada possui três atributos: um modelo *Model* instanciado pelo *ConditionModel*, uma comparação *Comparison* a ser realizada e um elemento *Target* a ser comparado. Como ilustrado acima, o *Target* é um elemento do mesmo tipo ao qual ele está sendo comparado, uma variável ou um argumento em tempo de execução. A *Comparison* deve pertencer ao conjunto de comparações possíveis para o tipo de dado do *Model (Model.DataType.Comparisons)*. Não faz sentido avaliar uma condição para elementos incomparáveis. Portanto, uma condição é avaliada comparando o elemento fonte com o elemento alvo.

Similarmente aos tipos de dados, temos também modelos de condições recomendadas:

- TokenModel = (TokenData, Tokens)
- RuleModel = (RuleData, RuleSet)
- DateModel = (DateData, SystemDate)
- TimeModel = (TimeData, SystemTime)

O *TokenModel* expressa o número de *tokens* restantes no modelo. O *RuleModel* usa o conjunto de regras onde esta condição se localiza. O *DateModel* e *TimeModel* expressam datas e horários do modelo, respectivamente. Não há necessidade de comparar as *flags Auth* e *Dirty*, portanto elas não são mencionadas.

5.5.2 Ações

Uma ação é simplesmente uma operação computacional efetuada sobre um *Target* (alvo). Essa operação deve pertencer ao domínio de operações permitidas a serem executadas. Além disso, uma ação permite o uso de um operando não obrigatório. Por exemplo, podemos incrementar o valor de um *token* em 1 usando o conjunto de elementos {*token*, +=, 1} ou inserir uma nova regra com {RuleSet, insert, ID}. Abaixo ilustramos o modelo de uma ação.

$$
\begin{aligned}
&\mathrm{Actions} = \{\mathrm{Target},\ \mathrm{Operation},\ \mathrm{Operand}\}\\
&\mathrm{Target} \in \{var_i | 0 \leq i \leq n\} \cup \{\mathrm{Tokens}, \mathrm{Dirty}, \mathrm{Auth}, \mathrm{RuleSet}\}\\
&\mathrm{Operation} \in \mathrm{type}(\mathrm{Target}).\mathrm{Operations}\\
&\mathrm{Operand} \in \mathrm{type}(\mathrm{Target}).\mathrm{Domain} \cup \{var_i | 0 \leq i \leq n\} \cup \{arg_i | 0 \leq i \leq n\}
\end{aligned}
$$

O *Target* pode ser uma variável, um *token*, uma *flag* ou um conjunto de regras. O *Operand* é do mesmo tipo do *target*, uma variável ou um argumento passado durante a execução. Usamos a notação *type(Target)* para indicar que a operação é qualquer membro válido do conjunto de operações apropriadas para o tipo de dado do objeto alvo.

Tokens é o conjunto de *tokens* remanescentes do conjunto de regras. A variável var_i indica que a ith variável é o alvo. O *DirtyBit* indica se a regra está ativa ou não. O *flag Auth* representa uma decisão superior para permitir ou negar a autorização. Finalmente, o *RuleSet* refere-se ao conjunto atual de regras.

5.5.3 Identificação das Regras

Toda regra deve ser identificada unicamente pelo modelo. Para isso, existe um identificador único ID composto por três elementos ID = (Nome, Tipo de Acesso, índice):

```
Name ∈ {Hierarquia de Objetos}
AccessType ∈ {tipo de acesso no contexto do pedido}
Index ∈ Inteiros
```

O *Name* identifica o caminho completo da raiz até o destino na hierarquia dos objetos para o qual a regra se aplica. O *AccessType* indica que tipo de autorização a regra irá gerar. E por último, o *Index* que identifica unicamente a regra dentro do conjunto de regras daquele objeto utilizando um número inteiro.

5.5.4 Regras *(RuleData)*

Baseado nos elementos passados podemos definir a estrutura formal das Regras. A base de uma regra é o seu ID. Além disso, uma regra possui um inicializador (semelhante aos construtores de classes em linguagens de programação orientada a objetos) e a capacidade de executar uma nova regra. A regra sabe a sua ordem de execução, quando deverá expirar, qualquer ação condicional que deverá ser executada e as ações que acontecerão normalmente. O ID da regra é o único componente que é mandatário.

```
rule ID{
  initialize initializer
  insertafter Id
  expire condition [ and|or condition ...]
  if condition [ and|or condition ...]
    then action
    Action action
}
```

O *initializer* é nada mais que uma lista de pares ordenados do tipo (alvo, valor), onde o alvo pode ser um *token* ou uma variável e o valor representa um valor atribuído ao alvo. O comando *insertafter* executa a regra especificada pelo atributo ID. Dependendo das condições, o comando *expire* define o *flag Dirty* com o valor verdadeiro tornando a regra inválida. As ações podem ser tomadas sob condições ou não. O *if-then-else* funciona como em uma linguagem de programação. O comando *action* executa uma ação sempre que a regra é autorizada.

5.6 Exemplos de uso do SACM

Nesta Seção é descrito como utilizar o SACM para expressar os diferentes tipos de modelo de acesso. Desde os modelos mais simples como as famosas ACLs aos mais complexos como a Muralha da China, entre outros. Esses diferentes tipos de políticas podem ser utilizados para prover um controle de acesso mais elaborado, complexo e confiável.

5.6.1 Controle de Acesso Simples

Quando temos um sistema em que o acesso é garantido ou negado permanentemente, isto é, feito explicitamente por um administrador, ele é chamado de estático. Não que o ambiente em que se encontrem seja estático, mas a maneira como as regras são feitas. Nessa categoria se encontram os modelos que utilizam as ACLs, CLs, RBACs, etc. O acesso é garantido estaticamente a um sujeito ou função *(role)*. Caso o sujeito ou uma *role* do sistema possam acessar um novo arquivo, uma nova regra terá que ser adicionada. O código abaixo indica que o acesso de execução ao objeto de nome */programas/programaX* é negado. O ID da regra consiste em */programas/programaX, Execute, 0* onde Execute é o tipo de acesso e 0 é o índice da regra. Esta regra é bem simples e consiste em apenas inicializar o valor do *token* com um valor negativo, negando o acesso ou não criando regra alguma. Caso fosse garantido o acesso de execução, o valor do *token* deveria ser um número não negativo. Esse é o tipo de regra mais simples possível. Podem-se gerar regras mais flexíveis inserindo restrições temporais como no próximo exemplo.

```
rule (/programas/programaX, Execute, 0){
  initialize ( (Tokens,-1) )
}
```

5.6.2 Controle de Acesso Temporal

Acessos temporais são aqueles permitidos ou negados durante um ou mais intervalos de tempo. Esse tipo de controle aumenta bastante a flexibilidade do sistema. Por exemplo, um usuário só pode utilizar o sistema durante uma hora, ou um funcionário só tem acesso aos arquivos durante o horário de trabalho. O código abaixo expressa um objeto chamado */relatorios/relatorioX* que só pode ser alterado durante a jornada de trabalho entre oito e dezesseis horas. A inicialização da regra diz que o acesso é proibido atribuindo um valor negativo ao *token* e guarda os limites dos horários nas variáveis var_1 e var_2, respectivamente. Porém, a condição *if* verifica se o tempo atual do sistema *TIME* está entre var_1 e var_2. Caso esteja, a *flag Auth* é definida como verdadeira e então o acesso é garantindo independentemente do valor do *token* ser negativo.

```
rule (/relatorios/relatorioX, Escrita, 0){
  initialize( (Tokens,-1), (var1,8), (var2,16) )
  if(TimeT > var1) and (TimeT < var2) then(Auth=true)
  else then (Auth=False)
}
```

O acesso temporal pode ser estendido para dias, anos, etc. Pode-se usar o comando *expire* para que regras desapareçam em um determinado momento. Por exemplo, um funcionário foi contratado para atualizar um programa durante seis meses e irá embora na data x. O código abaixo expressa essa regra. Na data x o *flag Dirty* é definida como verdadeira e a regra deixa de existir. Portanto, o acesso será negado.

```
rule (/relatorios/relatorioX, Escrita, 0){
  initialize( (Tokens,1), (var1,x) )
  expire (DATE > var1 )
}
```

5.6.3 Controle de Acesso com Contador de Acesso

Esse tipo de acesso é limitado por um número de vezes. Um exemplo simples é o já mencionado sobre quiosques distribuídos por aeroportos onde um cliente compra um determinado número de impressões 'n'. Quando ele solicita as 'n' impressões, são adicionados 'n' *tokens* que a cada impressão é decrementado em uma unidade. O código abaixo ilustra esse exemplo.

```
rule(/TravellerServices/PrintKiosks, Add, 0){
  initialize ((Tokens, 0))
  Action (Token += arg1)
}
rule(/TravellerServices/PrintKiosks, Imprimir, 0){
  Action (Token -= arg1)
}
```

Inicialmente, o usuário tem 0 impressão. Quando solicitadas as impressões, a quantidade é passada como argumento e adicionado ao *token*. A cada impressão, o *token* é decrementado em uma unidade. É como se ter várias fichas simbolizando cada folha e cada folha impressa exige a utilização de uma ficha.

Esse exemplo pode ser usado juntamente com acesso temporal para expressar o uso de máquinas para saque de dinheiro. É permitido efetuar saques até um limite diário e esse valor é reduzido para o período noturno por questões de segurança. Após atingir esse limite, qualquer máquina irá negar o acesso ao dinheiro da conta. Usando a ideia anterior dos quiosques e, redefinindo as regras padrões temos as seguintes regras ilustradas abaixo.

```
rule(/ATM, Saque, 0){
  if(DateT ≠ var1) then {
    initialize( (var0, rt),(var2, t0),(var3, t1) )
```

```
    Action (Tokens = var0)
  }
  if(Saldo >= rc) then
  {
    if (TimeT > t0 and TimeT < t1) then
    {
      Action (SAQUE)
      Action (Tokens -= rc)
    }
    else if (Tokens <= rs) then
    {
      Action (SAQUE)
      Action (Tokens -= rs)
    }
    else
    {
      print (Limite maximo de saque + rs)
    }
    Action (var1 = DataT)
  }
  else print (Saldo Insuficiente ou limite diario excedido)
}
```

Legenda:

- r_t: valor máximo de saque diário;
- r_c : valor de saque solicitado pelo cliente;
- r_s : valor a ser sacado fora do horário normal;
- t_0 : horário inicial de movimentação bancária normal;
- t_1 : horário final de movimentação bancária normal;
- DataT: Data atual do sistema;
- TimeT: Horário atual do sistema.

Se a pessoa ainda não utilizou o caixa-rápido, então a condição *if-then* irá ser executada e o número de *tokens* será definido igual ao limite diário de saque e a data será atualizada para o dia atual. Quando for feito um saque, ele será reduzido do número de *tokens*. Quando esgotar o limite diário, o valor dos *tokens* será igual a zero e passará a negar o acesso.

5.6.4 Controle de acesso baseado em ações prévias (estado)

Os exemplos a seguir utilizarão algumas das ideias mais importantes no nosso modelo que são primeiramente a de liberar ou bloquear acesso baseado em ações prévias e outra importante é a de regras poderem criar ou remover outras regras, possibilitando o uso dessa função dinâmica em dispositivos pervasivos que possuem limitação de *hardware*.

5.6.4.1 Acesso baseado em ações prévias

A Muralha da China *(Chinise Wall)* é uma política que autoriza o acesso a um objeto baseado em acessos anteriores a outros objetos. O mecanismo de usar contadores *tokens* para armazenar o estado de um objeto não permite expressar esse tipo de acesso. Entretanto, a dinamicidade do SACM em que regras criam novas regras e a organização hierárquica dos objetos, permite a expressão da política Muralha da China.

O código a seguir ilustra duas classes conflitantes A e B. Se um sujeito acessar um objeto da classe A ele não poderá acessar os objetos da classe B e vice-versa. Para isso, o acesso é dado inicialmente para as duas classes definindo o valor do contador *token* com um valor positivo, porém, dada a escolha do primeiro acesso (objeto da classe A ou objeto da classe B), uma nova regra é criada para negar o acesso à classe conflitante definindo o *flag Auth* como falso.

```
rule(/Fileserver/class–A, read, 1){
  initialize( (tokens, 1) )
  if(RuleT♯(/Fileserver/class–B,read,0)) then
    (RuleSet insert
      rule (/Fileserver/class–B, read, 0){
        Action (Auth = false)
      }
    )
}
rule(/Fileserver/class–B, read, 1){
  initialize( (tokens, 1) )
  if(RuleT♯(/Fileserver/class–A,read,0)) then
    (RuleSet insert
      rule(/Fileserver/class–A,read,0){
        Action (Auth = false)
      }
    )
}
```

Ou seja, o acesso a um objeto implica na negação do acesso a outros objetos de classes conflitantes a primeira. Portanto, a dinamicidade das regras criarem novas regras é crucial neste exemplo. Os *tokens* também permitem incluir uma forte dinamicidade nos acessos, pois guardam os estados anteriores dos objetos. Isso se torna importante quando um acesso anterior ao objeto influencia o próximo acesso ao mesmo objeto. Esse é o caso dos exemplos a seguir.

5.6.4.2 Acesso baseado em regras criadas previamente

Um exemplo é o aluguel de livros em uma biblioteca. Você pode alugar até 'n' livros. Imagine várias bibliotecas espalhadas onde um sujeito pode alugar um livro em qualquer uma delas. Para expressar esse ambiente dinâmico, um sujeito pode através de um *Smart Card*, armazenar o seu estado atual com as regras abaixo.

```
rule(/biblioteca/cliente1, ALUGUEL, 0){
  if(RuleT∄(/biblioteca/cliente1,ALUGUEL,arg1)) then {
      RuleSet insert rule (/biblioteca/cliente1, ALUGUEL,arg1)
      initialize ((Tokens,0), (var0,n), (arg2,DATEDEV))
      Action (Tokens = var0)
      Action (Tokens −= arg3)
      }
  else if (Tokens < arg3) or (DateT > arg2) then
    Action (Auth=false)
  else Action (Tokens −= arg3)
}

rule(/biblioteca/cliente1, DEVOLVER, 0){
      Action (remove (/biblioteca/cliente1,ALUGUEL,arg1))
      Action (Tokens+=1)
}
```

As regras são executadas quando um sujeito tenta alugar ou devolver um livro. Cada livro é identificado por um número único passado como o argumento arg_1. Quando um livro é alugado, a regra de aluguel 0 é executada e define o valor dos *tokens* com o número de aluguéis permitido 'n' definido como arg_3. Além disso, o algoritmo verifica se já existe uma regra criada para o cliente que tem a intenção de efetuar uma locação, e caso não exista, será criada uma regra com o ID *(/biblioteca, Aluguel,* arg_1*)*. Ou seja, para cada livro alugado existirá uma regra que verifica se o livro foi entregue antes da data de retorno passada como argumento arg_2. Caso tenha passado a data, o acesso é negado. Se o valor dos *tokens* ficar igual a zero, então o sujeito não pode mais alugar um livro e deve retornar algum dos livros alugados. Quando o sujeito retornar um livro, a regra de retorno 0 é executada removendo a regra criada quando o mesmo

livro foi alugado e adicionando 1 aos *tokens* restantes. Após a execução de todas as regras sem que nenhuma delas falhe, o valor dos *tokens* restantes representará o número atual de possíveis aluguéis.

Outro exemplo é o de distribuição de ingressos gratuitos ou com desconto de um jogo de futebol para os associados, chamados sócio-torcedores de um determinado clube, neste clube existe um sistema que permite a liberação de 'n' ingressos para quem é sócio. Este exemplo é concretizado utilizando uma metodologia de criação de regras dinâmicas, mas também utilizando a natureza hierárquica dos objetos. É criado uma hierarquia de regras de tal forma que todos os eventos são filhos do objeto */Eventos/*. A regra que produzirá novas regras é inserida na raiz e então será executada em todas as ações de compra de ingressos para os eventos. O comando *if-then* verifica se existe uma regra para o evento específico 'e', como filho de Eventos, que será descrito como */Eventos/e*. Se não existir, esta é a primeira tentativa de compra de ingressos para este evento, e insere uma regra para este evento, que irá expirar após a data do evento, d_e. A regra para cada evento é, então, uma regra simples de contador de acesso visto na Seção 7.3. É importante entender que cada objeto na hierarquia tem seu próprio contador de *token*, para cada evento */Eventos/e* haverá um contador *token* próprio, uma vez que cada evento tem seu próprio número de ingressos com desconto disponíveis.

```
rule (/Events, buy, 0){
  if( RuleT ∄ (/Events/e,buy,0) ) then
        RuleSet insert rule(/Events/e,buy,0){
          initialize( (Tokens, n), (var_0, d_e) )
          expire(DateT > var_0)
          Action (Tokens −=1)
      }
}
```

5.6.5 Controle de Acesso com Sensibilidade ao Contexto

No exemplo a seguir será possível averiguar o potencial que nosso modelo possui de fazer com que regras criem ou removam novas regras em um ambiente hospitalar sensível ao contexto. A forma de aquisição do contexto pode ser baseado na solução apresentada por (WANT et al., 1992). O algoritmo abaixo utiliza sensores e a comunicação *Wireless* por RFID ou *Bluetooth* para determinar se o médico está presente na mesma ala da enfermeira e baseado nessa presença, será criada uma regra permitindo a leitura e escrita em um prontuário de um paciente, porém se o médico deixar a ala, a regra será removida e em seu lugar uma regra de leitura será imposta.

```
rule (/hospital/enfermaria1/ala1, Acesso, 0){
 initialize ((Tokens,1), (v_0, T_inicial), (v_1, T_final))
```

```
 if (TimeT >= v0) and (TimeT <= v1) and
  (RuleT∄(/hospital/enfermaria1/ala1/enfermeira1 ,READ,0)) then
    RuleSet insert rule(/hospital/enfermaria1/ala1/enfermeira1 ,
                                                        READ,0)
 else if (RuleT∃(/hospital/enfermaria1/ala1/doutor1 ,RW,0)) then
    RuleSet insert rule(/hospital/enfermaria1/ala1/enfermeira1 ,
                                                        RW,0){
    expire(RuleT∄(/hospital/enfermaria1/ala1/doutor1 ,RW,0))
    }
}
```

Legenda:

- v_0, v_1: variáveis que receberão o horário de inicio e fim respectivamente;
- RuleT: grupo de regras criadas;
- TimeT: tempo do sistema;

6 IMPLEMENTAÇÃO DO MODELO USANDO A LINGUAGEM JAVA

Neste capítulo será demonstrado quatro exemplos de implementações contendo os principais aspectos do modelo SACM. Como pré-requisito para o entendimento do modelo, é necessário um conhecimento mediano sobre a linguagem Java, suas API's e *frameworks*, pois usaremos o conjunto dessa tecnologia para a implementação dos algoritmos *PrinterKiosk*, *Chinese Wall*, Distribuição de Ingressos e o de Sensibilidade ao Contexto.

Nesta implementação é utilizado a sintaxe e outros recursos desta linguagem para aplicar o modelo SACM nas diversas situações em que o mesmo se propõe a resolver.

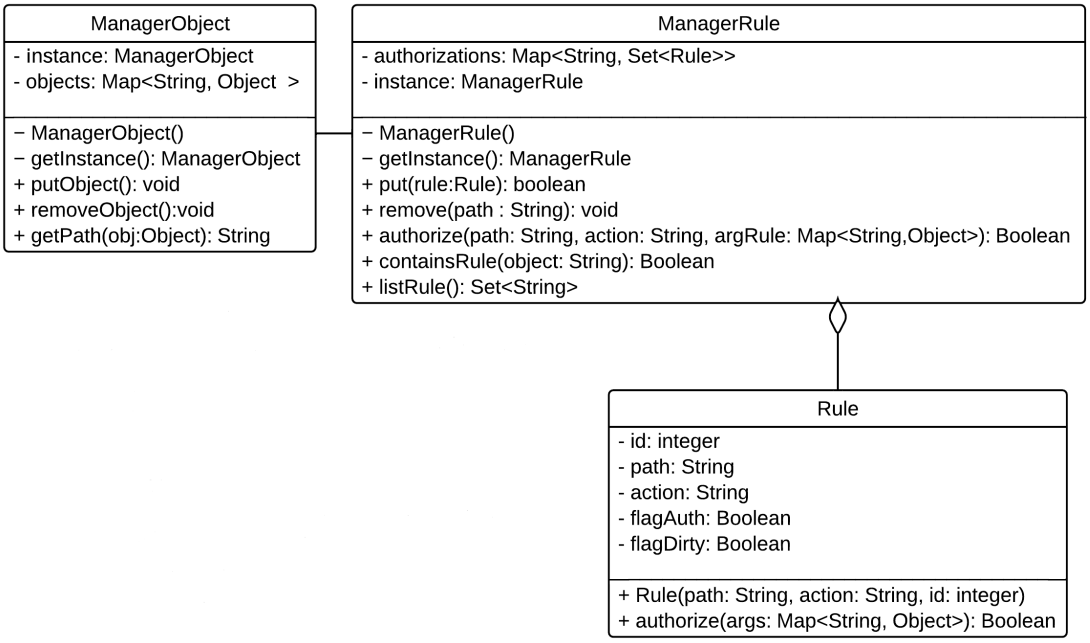


Figura 9: Diagrama UML

Esta aplicação se propõe a ser uma camada de software mediadora entre o sujeito e o objeto, de modo que um determinado usuário que desejar acessar um objeto, deverá portar consigo uma unidade de armazenamento seguro - *Smart Card*, por exemplo - onde deverão estar suas informações. Esta camada obterá as informações referentes ao controle de acesso contidas com o sujeito repassando-as para as regras de controle que irão dar sustentação as permissões sobre os objetos. Outras informações poderão ser usadas como critérios de autorização como informações sobre o estado do ambiente que pode ser capturadas através de sensores.

As regras irão receber um conjunto de argumentos extraídos do sujeito que passarão pelo seu algoritmo de validação. Se todas as regras de acesso para o objeto e ação desejados forem validadas corretamente, será permitido o acesso do sujeito ao objeto.

Em seguida será demostrado os passos básicos para a construção das regras de controle de acesso, de forma que estes conceitos serão reaproveitados nas seções que suscedem este

capítulo.

A explicação é iniciada com o estudo das classes utilizadas nesta implementação: *ManagerObject, ManagerRule e Rule*. Estas classes compõem o núcleo da aplicação. O diagrama UML da Figura 9 demonstra as classes e métodos utilizados na implementação que serão explicados de forma mais detalhada nas seções seguintes.

6.1 Gerenciamento dos objetos

Na classe *ManagerObject* se encontram métodos relacionados com a gerência dos objetos e associações de caminhos ao mesmos, de modo que cada objeto seja associado a um caminho que será usado na criação das regras de controle de acesso, fazendo com que seja possível identificar que regras estão ligadas a um determinado objeto, possibilitando assim, a validação de determinadas regra de um dado objeto. A classe *ManagerObject* aplica o padrão *singleton* e para se obter uma instância desta classe é invocado o método estático *'getInstance'* da própria classe evitando assim, a duplicidade de objetos instanciados.Veja o exemplo abaixo:

```
ManagerObject managerObj = ManagerObject.getInstance();
```

Será usado como exemplo o objeto 'x' para demostrar a associação de um objeto a um caminho através do método *'putObject'*:

```
Object x = new Object();
managerObj.putObject(x,"/myobject");
```

Conforme o código acima, é adicionado o objeto 'x' ao caminho "/myobject". Será necessário criar uma regra para 'x', onde a regra definirá os critérios para a autorização de um determinado comportamento do objeto. Para que isso seja possível, primeiramente terá que ser obtido uma instância da classe *ManagerRule* que será o gerenciador de todas as regras ativas. *ManagerRule* aplica também o padrão *singleton* utilizado na classe *ManagerObject*, de modo que para obter uma instância da classe *ManagerRule* é efetuado o mesmo procedimento.

```
ManagerRule managerR = ManagerRule.getInstance();
```

Com a instância de *ManagerRule*, podemos adicionar regras para o objeto 'x'. Com o método *'put'* da classe *ManagerRule* será armazenado uma nova regra para 'x'.

Antes de ser demostrado a utilização do método *'put'*, será abordada a classe *Rule* que representa a regra de autorização em si. Esta classe recebe em seu construtor três parâmetros:

uma *string* que representa um caminho que está associado a um determinado objeto, uma *string* contendo a ação (comportamento do objeto) e um número inteiro que representa o identificador da regra, de modo que este seja único para cada regra a fim de identificar diferentes regras para o mesmo objeto e ação.

A classe mencionada anteriormente contém também um método abstrato que representa o algorítimo da regra. O método é o *'authorize'* que recebe como parâmento um *'Map'* com os argumentos que podem ser usados na invocação dele próprio. Quando a regra é avaliada na instância da classe *Rule*, será invocado o método *'authorize'* da própria classe. Se o retorno da chamada deste método tem valor igual a *'true'* isto indica que a regra foi validada com êxito. Cabe a quem definir a regra implementar o método *'authorize'* que consequentemente implica na implementação do algorítimo da regra. Veja o exemplo abaixo:

```
managerR.put(new Rule(managerObj.getPath(x), "toString",0) {

    @Override
    public Boolean authorize(Map<String, Object> args) {
    return true;
    }
});
```

Acima foi adicionado uma regra para o objeto 'x'. Nos parâmetros da criação da nova regra foi passado o caminho de 'x' - através do método *'getPath'* que recebe um objeto registrado (objeto que representa o recurso no sistema) e retorna seu caminho - como também a ação *'toString'* que é um comportamento do objeto 'x' e o identificador da regra com o valor zero. Para validar um conjunto de regras que estão ligadas a um objeto é necessário invocar o método estático *'authorize'* da classe *ManagerRule*.

```
ManagerRule.getInstance().authorize(managerObj.getPath(x), "toString",
new HashMap<String, Object>());
```

O método *authorize* da classe *ManagerRule* recebe como argumentos o caminho do objeto, a ação sobre o objeto e um conjunto de parâmetros que serão passados para as regras na hora de sua avaliação. Este método irá buscar as regras que atendem ao padrão fornecido na chamado do método, ou seja, nesse exemplo ele irá procurar regras para o caminho '/myobject' que se referem a ação *'toString'*. Se a regra atender ao padrão é invocado o método *'authorize'* de cada uma delas passando como parâmetro o *'Map'* que ele próprio recebeu. Após a chamada do método *'authorize'* de cada regra que atende ao padrão, caso nenhuma delas retorne o valor *'false'*, o método *authorize* da classe *ManagerRule* retornará o valor *'true'*.

A classe *ManagerObject* possui métodos que possibilitam a gerência dos objetos, conforme listado abaixo:

- O método *putObject* tem a funcionalidade de adicionar um objeto, passando o caminho (o parâmetro *path*) e o próprio objeto;
- O método *removeObject* tem como funcionalidade remover o objeto. Para isso é necessário que seja passado à referência do objeto como parâmetro para o método;
- O método *getPath* retorna o caminho de um determinado objeto;
- O método *getObjects* retorna um *Map* contendo todos os objetos e os seus respectivos caminhos;
- O método *getObject* retorna um objeto de acordo com o caminho passado como parâmetro;
- O método *getInstance* permite que seja instanciado apenas um objeto por vez em tempo de execução.

A classe *ManagerRule* possui métodos que auxiliam na gestão das regras, conforme abaixo:

- O método *put* adiciona uma regra e caso a regra passe pelas validações impostas por este método é retornado *true* (verdadeiro);
- O método *remove* retira a regra de acordo com o caminho passado ou objeto;
- O método *authorize* tem a função de efetuar a validação da(s) regra(s) de acordo com os seguintes parâmetros: o caminho do objeto, ação exercida no objeto e o conjunto de argumentos para a autorização da ação; retornando o valor referente a permissão ou negação do acesso;
- O método *containsRule* verifica a existência de regra(s) referente a um determinado objeto;
- O método *listRules* retorna uma lista contendo todas as regras inseridas e válidas.

Na classe abstrata *Rule* temos todas as propriedades da regra como a ação, os *flags*, o id e o caminho, além de alguns métodos para a configuração dos atributos. Os atributos da classe *Rule* são: o ID da regra, o caminho (*path*) da regra, a ação (*action*), o *flag Dirty* e o *flag Auth*. O construtor tem a função de inicializar todos os atributos.

Predominantemente os métodos presentes na classe são responsáveis por fazer a configuração dos atributos de acordo com os parâmetros passados e o resgate desses, através dos métodos convencionais *set e get*.

6.2 Exemplo de um Quiosque de Impressão

O cenário da primeira implementação ocorre em um Aeroporto onde se tem várias impressoras que fazem parte de um conjunto de quiosques de impressão não possuindo conectividade entre si, a fim de minimizar custos com a parte da estruturação. Farão parte do cenário 3 elementos importantes:

1. **Impressora**: que receberá as solicitações dos usuários e efetuará a ação de imprimir a partir da verificação do *token* carregado pelo usuário, que solicitará a impressão em uma área de armazenamento que conterá a quantidade de créditos de impressão adquirida por ele. Dessa forma, a impressora liberará ou não a impressão dependendo da quantidade de crédito existente no *token*;

2. **Quiosque**: responsável por inserir ou retirar créditos no *token* do usuário, verificar o saldo e também, pela manipulação das *flags (auth, dirty)*, que neste exemplo não são utilizadas. Da mesma forma que a impressora, o quiosque de impressão se vale de regras de controle de acesso para permitir que as ações nela sejam realizadas, utilizando dados do sujeito, obtidos de igual maneira que a impressora;

3. ***Token* do usuário**: responsável por levar as informações que participarão do processo de controle de acesso ao recurso. Elas poderão estar armazenadas em um *Smart Card*, um *Tablet* ou um *Smartphone*.

Os dados referentes aos créditos do sujeito serão obtidos através de um dispositivo que os retira de uma unidade de armazenamento. Estes dados são repassados para a aplicação SACM, que extrai as informações do sujeito contidas no *Smart Card*. Valendo-se de regras de controle de acesso, a impressora negará ou a permitirá uma determinada impressão.

Quando o passageiro desejar utilizar o serviço de impressão do Aeroporto, ele deverá se dirigir a um quiosque para efetuar a adição de créditos no seu *Smart Card* mediante o pagamento com cartão de crédito da quantidade de impressões solicitadas, conforme a Figura 10 passos 1 e 2. Após a adição dos créditos, basta apenas que o passageiro, de posse de seu dispositivo móvel, se conecte a uma impressora via bluetooth através do driver apropriado (ainda em desenvolvimento) e solicite a impressão como mostrado nos passos 3 e 4. A impressora solicitará que o usuário insira seu *Smart Card* na leitora a fim de verificar se o mesmo possui créditos suficientes para suprir tal requisição (passo 5), se seu crédito for maior ou igual

a quantidade requisitada, a impressora imprimirá o documento e enviará uma mensagem para o usuário informando a quantidade de créditos remanescentes ao mesmo tempo que atualiza o valor no Smart Card.

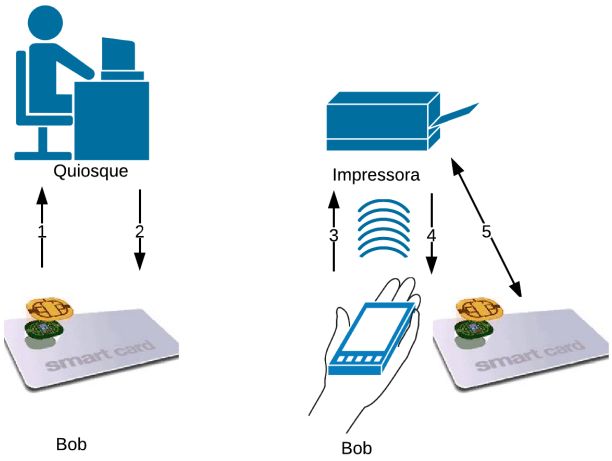


Figura 10: Quiosque de Impressão

Primeiramente será inicializado o quiosque com as propriedades necessárias para o controle das informações que virão do *Smart Card*, ou seja, será adicionado ao quiosque, uma camada de *software* que terá a finalidade de avaliar o saldo do sujeito (essa informação estará justamente com ele), adicionar ou remover uma determinada quantidade do *token* e adicionar as regras necessárias para o controle de acesso. Essa camada de *software* poderá prover o serviço necessário para a comunicação entre os dois dispositivos, ou seja, entre o *Smart Card* e o quiosque. Toda as avaliações da regra e validações serão realizadas no quiosque. Do ponto de vista de serviço ao sujeito, o quiosque irá fornecerer créditos, que serão controladas por regras que permitem adição e remoção deles.

Para a avaliação da autorização sobre uma ou mais ações em um objeto, será instanciado na inicialização do quiosque (dispositivo de autorização) um objeto do tipo *ManagerObject*, conforme o código abaixo:

```
ManagerObject managerObj = ManagerObject.getInstance();
```

Será necessario também obter uma instância da classe *ManagerRule* que possibilitará a manipulação das regras, conforme a seguir.

```
ManagerRule managerR = ManagerRule.getInstance();

PrintKiosk kiosk = new PrintKiosk();
managerObj.PutObject(kiosk, “/aeroport/kiosk”);
```

Acima, é instanciado o objeto *kiosk* que representa o próprio quiosque e o mesmo é associado a um caminho, que permitirá a criação da regra. Lembrando que tudo isso será feito no próprio quiosque.

Será criado em sequência duas regras para o *kiosk*, uma para adição e outra para remoção. A primeira, de adição, está relacionada a ação *'add'* permitindo ou negando a ação de incrementar um valor ao *token* que indica a quantidade de créditos para impressão.

```
managerR.put(new Rule (managerObj.getpath(kiosk), ‘‘add’’, 0) ){
```

Na linha de código acima, onde é criada a regra, é passado ao construtor um conjunto de parâmetros. Veja que a ação passada é a de adição. O código abaixo continua com a criação da regra.

```
public Boolean authorize(Map<String, Object> args) {
    /* Resgate dos parâmetros pelo Map */
    Integer token = (Integer) args.get("token");
    Integer cred = (Integer) args.get("cred");

    if(cred >0){
    args.put("token", token -= cred);
    return true;
    }
    return false;
  }
});
```

A segunda regra a ser criada no quiosque será para a autorização da ação da remoção de uma quantidade de créditos no *token*, com objetivo de ressarcir financeiramente o sujeito com os créditos que sobraram e que não serão utilizados, conforme código abaixo:

```
managerR.put(new Rule(managerObj.getPath(kiosk),"remove",0) {

    @Override
    public Boolean authorize(Map<String, Object> args) {

      /* Resgate dos parâmetros pelo Map */

      Integer token = (Integer) args.get("token");
```

```
        Integer cred = (Integer) args.get("cred");

        /* Condição de avaliação para remoção dos créditos */
        if(cred <= token){
        args.put("token", token -= cred);
        return true;
        }
        return false;
    }
});
```

Será passado os valores que serão usados na validação. O valor do parâmetro *'token'* será extraido do *Smart Card* e o parâmetro 'cred', que informa o valor de créditos a serem adicionados ou removidos, será indicado na hora da solicitação do crédito pelo passageiro. Depois de finalizado o processo de adição ou remoção, o novo valor do parametro *'token'* será armazenado no *Smart Card* do usuário que requisitou a ação.

Os códigos abaixo efetuarão uma verificação para visualizar se a ação de adição de crédito no *token* foi executada. Nesta situação, o retorno da função *authorize* é *true*, porque a verificação do valor no *token* é superior a zero. Antes será instaciado um *Map* para armazenar os parâmetros utilizados na verificação da autorização.

```
Map<String, Object> args = new HashMap<String, Object>());

argsRule.put("cred", 2);

System.out.print(managerR.authorize(managerObj.getPath(kiosk),
"add",argsRule));
```

Caso o passageiro deseje recuperar os créditos inseridos no seu *token*, será executada uma avalição em cima da ação de remoção de créditos (*remove*), com o objetivo de retirar o valor inserido no *token*.

```
System.out.print(managerR.authorize(managerObj.getPath(kiosk),
"remove",argsRule));
```

Neste ponto será descrito a camada de software implementada para a impressora. Vale lembrar que ainda está em desenvolvimento um driver que permitirá a comunicação entre a aplicação SACM e o *Smart Card*, como também para o dispositivo móvel. Primeiramente, será criado um objeto *printer* com o objetivo de receber a solitação de impressão.

```
Printer printer = new Printer();
```

A seguir, é feito uma associação como já demonstrado anteriormente entre o objeto e o endereço. Todo esse processo será feito na própria impressora através da aplicação (SACM) que está instalada.

```
managerObj.putObject(printer, "/aeroport/pinter");
```

Também será criado uma regra de autorização para o objeto *printer*. A regra associa-se ao caminho do objeto *printer* e a ação *print*:

```
managerR.put(new Rule(managerObj.getPath(printer),"print",0) {

  @Override

  public Boolean authorize(Map<String, Object> args) {

    Integer token = (Integer) args.get("token");
    Integer imp = (Integer) args.get("imp");

    if(token == null && imp == null)
    return false;

    if(token >= imp && token > 0){
    args.put("token", token -= imp);
    }else{
    return false;
    }
    return true;
  }
});
```

A chave *'token'* do *MAP args*, na regra acima, retornará o valor dos créditos que o usuário tem para impressão e a chave 'imp' terá o número de impressões que o mesmo deseja obter. Até esse passo, foi inicialiazado a regra na impressora para que ela possa prover o serviço de acordo com o controle feito pelo SACM.

Nesse momento será feito a representação da interação entre o sujeito e a impressora. Com o método *put* é possível manipular tanto a adição de créditos no *token* do usuário como

a quantidade de impressões solicitadas por ele. Essas propriedades referentes ao crédito serão passadas do *Smart Card* para a impressora.

```
argsRule.put("token", 1);
argsRule.put("imp", 1);
System.out.print(ManagerR.authorize(ManagerObj.getPath(printer),
‘‘print’’, argsRule));
```

É verificado que nesse caso após a execução do método *authorize*, a saída será *‘true’*, pois pode-se percebe que o algorítimo da regra avalia se o valor do *token* é maior ou igual ao número de impressões, se positivo, é decrementado do *token* esse valor, retornando *true*, demonstrando que a transação foi executada.

6.2.1 Exemplo com *interface* gráfica

Observe que é executada a ação de adição inserindo o valor de crédito no total de 20 conforme a Figura 11. Depois de clicar em confirmar a operação será realizada com sucesso:

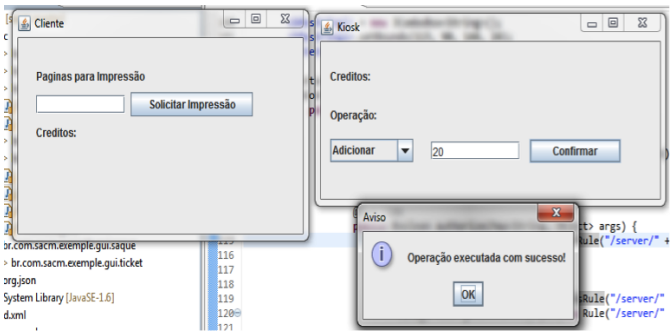


Figura 11: Confirmação da compra de créditos

Agora veja que será executada a ação de remover um total de 40 de créditos sendo que existem somente 20 armazenados. Com isso, é retornado um código de erro para o usuário:

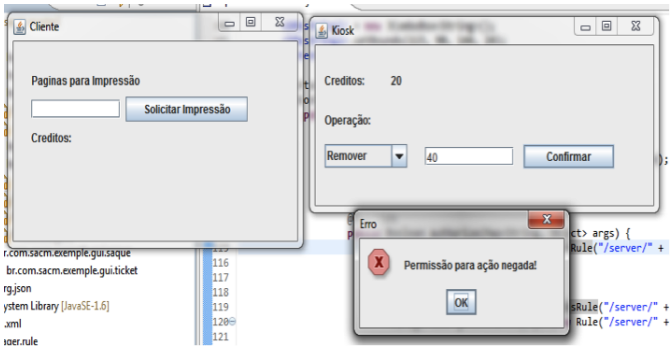


Figura 12: Tentativa de devolução do cliente com erro

Na Figura 13 é simulado o pedido de impressão de 10 folhas, como o cliente havia solicitado 20, ele terá a impressão autorizada e na Figura 14 é mostrado o seu *token* decrementado.

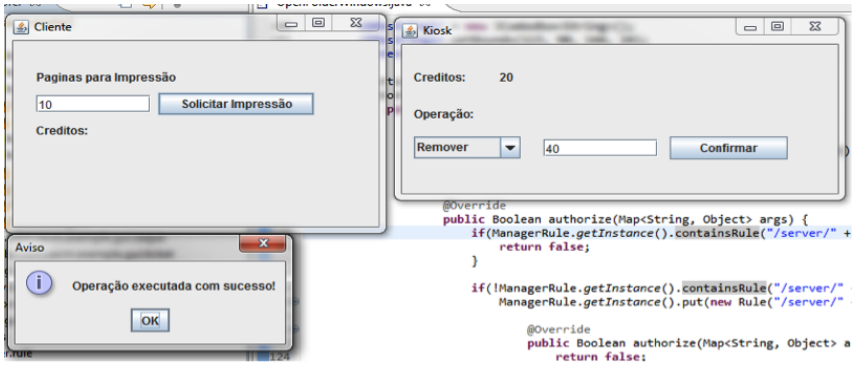


Figura 13: Solicitação de impressão

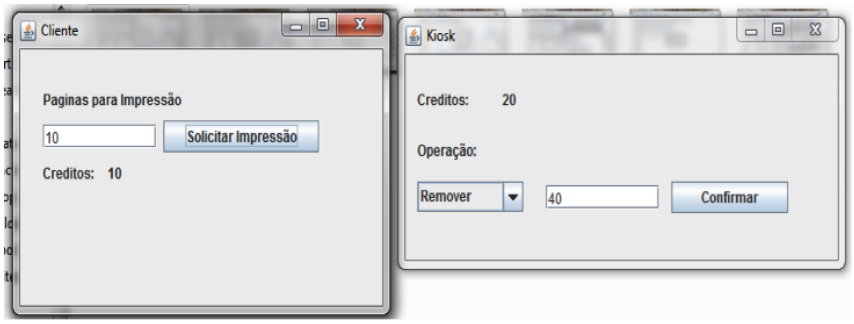


Figura 14: Diminuição do valor do *token* do cliente

6.3 Exemplo baseado no *Chinese Wall*

Nesta seção será exposto o exemplo de um ambiente corporativo onde existe dois setores com acesso restrito, levando em consideração que os acessos de ambos os setores entrará em conflitos de interesses, será feito uma avalição a partir do momento em que um usuário tenha acesso a um setor A, e caso este queira acessar o outro setor B, o acesso será bloqueado.

Neste exemplo, considere que na empresa X existe um setor de Contabilidade onde existem diversos dados sigilosos. O funcionário, usuário do sistema, acessou o setor de Contabilidade e com isso obteve o acesso a recursos, que de acordo com o paradigma do conflito de interesses, não poderá acessar outro setor da empresa, como por exemplo o Financeiro. Através de regras que regem os conflitos de interesses, será eliminado essa possibilidade do funcionário ter o acesso ao outro setor de administração que possa entrar em conflito.

Será instanciado dois objetos que representam as pastas que podem ser acessadas, o primeiro com a referência folderOfClient1 e o segundo com a referência folderOfClient2 onde

ambos serão adicionados utilizando o método *putObject*, associando-os ao caminho informado. Ambas as pastas estarão em um dispositivo com o interpretador SACM. Segue o código abaixo:

```
Folder folderOfClient1 = new Folder();
Folder folderOfClient2 = new Folder();

managerObj.putObject(folderOfClient1, "/server1/folder1");
managerObj.putObject(folderOfClient2, "/server2/folder2");
```

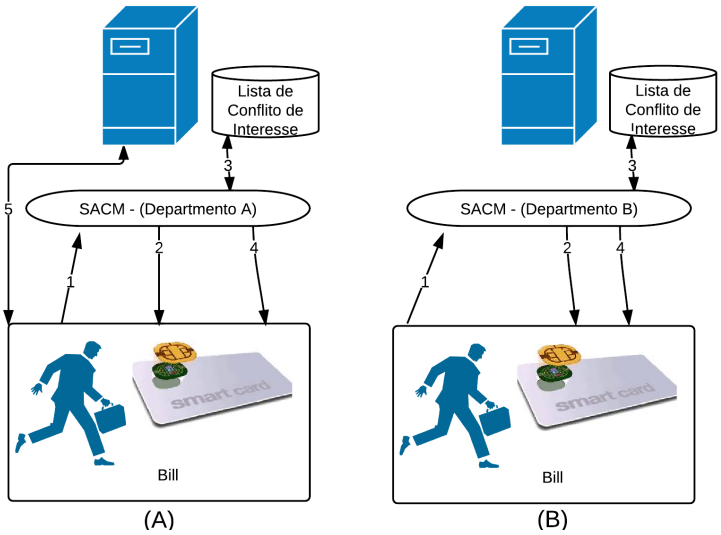


Figura 15: Muralha da China

A partir do momento em que o funcionário inserir o seu *Smart Card* no dispositivo de autorização, conforme a Figura 15A-1, que avalia as regras de acesso (interpretador SACM), com intuito de obter o acesso a uma informação que está contida no servidor do setor, automaticamente será invocado no próprio dispositivo o método que ficará responsável por obter essas informações do *Smart Card*, Figura 15A-2, com o objetivo de repassá-las ao método que valida as regras responsáveis pela autorização do acesso conforme Figura 15A-3. Esse método é o *authorize*, da classe *ManagerRule*. Consequentemente, se a autorização for concedida (Figura 15A-4), neste exemplo, os registros de acesso referentes ao usuário naquele setor de contabilidade serão armazenada de forma criptografada no *Smart Card* do sujeito através do método responsável por fazer a persistência do dado no *Smart Card* (Figura 15A-5).

O processo será feito da mesma forma para o outro setor, onde no exemplo será o da administração de Recursos Financeiros. O funcionário carregará os acessos prévios no seu *Smart Card* e a partir do momento em que ele o inserir no dispositivo de autorização, (Figura 15B-1), será invocado uma regra que avaliará se existe um acesso anterior que entra em conflito de interesses com acesso em questão (Figura 15B-2 e 3). Caso exista, o interpretador SACM

negará o acesso (Figura 15B-4), caso contrário, será permitido e o acesso referente a este setor e ocorrerá o registrado no *Smart Card*.

No *Map* de argumentos a chave *'classConflict'* guardará os acessos e somente os acessos das pastas visualizadas pelo usuário. O valor desse argumento será avaliado pelo interpretador SACM de forma que mesmo o obtenha do *Smart Card*. Primeiramente, será verificado se existe dentro da variável *'ClassConflict'*, contida no *Smart Card*, algum registro de um acesso prévio, caso não exista, o registro do acesso será inserido na variável e o acesso concedido. A função do *Smart Card* em todo o processo é somente de transferir o argumento *'classConflict'* de um setor a outro. Na linha abaixo o parâmetro *'ClassConflict'* é definido com um *Set* vazio que irá armazenar todos os acessos.

```
argsRule.put("classConflict",new HashSet<>(String));
```

Neste ponto, é criado uma regra passando a ação *'open'* e a localização do objeto definido anteriormente juntamente com o seu ID:

```
managerR.put(new Rule("/server1/folder1","open",1) {
```

A partir daqui é criado a regra pelo método *authorize* para fazer a validação do acesso com base em acessos anteriores:

```
@Override
public Boolean authorize(Map<String, Object> args) {
```

Através do método *get* é resgatado a informação do *Hash* declarado anteriormente:

```
Set<String> classConflict = (Set<String>) args.get("classConflict");
```

A verificação do conflito é feito na regra. Caso esteja contido no *Hash* o acesso a pasta2, é retornado o valor *false*, com isso é introduzido o conceito de conflito de interesses, pois aqui é verificado se o usuário já acessou a pasta2.

```
  if(classConflict.contains("/server2/folder2")){

  return false;
  }
```

Caso não exista registro de acesso que entrem em conflito com a tentativa atual, o acesso será permitido e o caminho da pasta acessada registrado:

```
    if(!classConflict.contains(getPath())){
      classConflict.add(getPath());
    }
    return true;
  }
});
```

Na regra a seguir, é feito o mesmo processo em relação ao conflito de interesse da pasta2 para a pasta1.

```
managerR.put(new Rule("/server2/folder2","open",1) {

@Override
public Boolean authorize(Map<String, Object> args) {
```

Seguindo a mesma lógica, será recuperado através do método *get* o *Hash* referente a classe de conflito:

```
Set<String> classConflict = (Set<String>) args.get("classConflict");
```

Neste ponto será verificado se há conflito de interesse com o acesso solicitado:

```
if(classConflict.contains("/server/folder1")){
  return false;
}
```

Se o caminho da pasta acessada não estiver registrado no *Hash* que contém os acessos anteriores, ele será adicionado:

```
    if(!classConflict.contains(getPath()) ){
      classConflict.add(getPath());
    }
    return true;
  }
 });
```

Nesse momento será implementado a validação da ação para o acesso de ambas as pastas, de acordo com os critérios definidos anteriormente. Veja:

```
System.out.println(ManagerRule.getInstance().authorize(ManagerObject.
getInstance().getPath(folderOfClient1), "open",argsRule));
System.out.println(ManagerRule.getInstance().authorize(ManagerObject.
getInstance().getPath(folderOfClient2), "open",argsRule));
```

Aqui é visualizado o conjunto de regra(s) contida(s) no objeto da classe *ManagerRule*:

```
for (String string : ManagerRule.getInstance().listRules()) {
    System.out.println(string);
}
```

6.3.1 Exemplo com *interface* gráfica

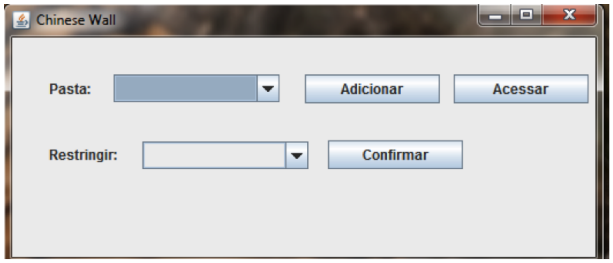


Figura 16: Representação gráfica do Chinese Wall

A Figura 16 demonstra os campos para criação das pastas e adição na classe de conflito de interesses. O campo para restrição significa que se uma pessoa X quiser ter acesso a uma pasta B, não conseguirá acessar outra pasta C, devido B e C estarem em um grupo de conflito de interesse. Essa informação da ação prévia será levada com a pessoa X de modo a impedir que as pastas B e C tenham que se comunicar para tomar a decisão de acesso.

Vamos criar duas pastas, como mostrado na Figura 17. Uma chamada *Josh* e outra *Victoria*, depois vamos colocá-las dentro da classe de conflito de interesse como mostra a Figura 18:

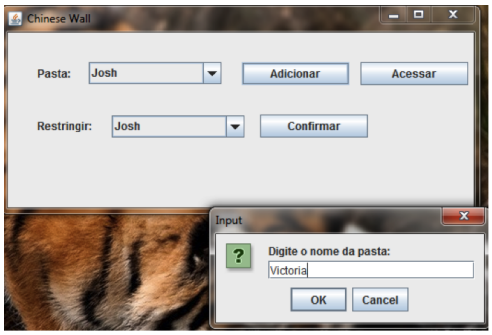


Figura 17: Criando as pastas

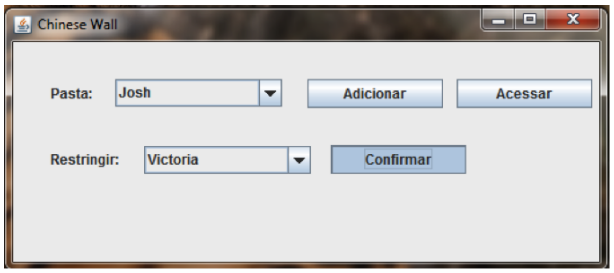


Figura 18: Registro do Acesso

Agora se tentarmos acessar a pasta Victoria não será possível, pois quando acessamos a pasta Josh automaticamente foi disparado um gatilho fazendo com que o acesso a pasta Victoria fosse negada:

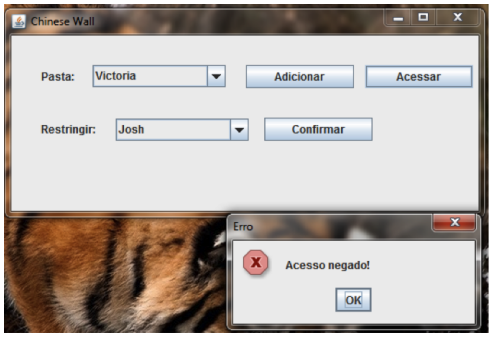


Figura 19: Acesso Negado

6.4 Exemplo de Distribuição de ingressos

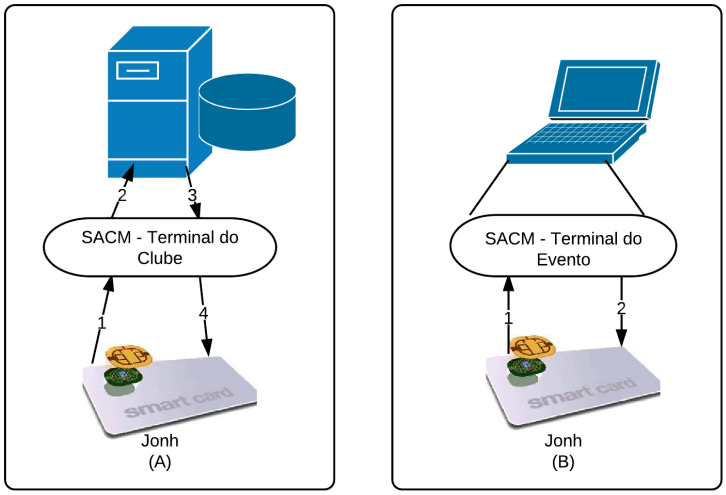


Figura 20: Distribuição de Ingressos

Neste exemplo é descrito um cenário onde necessita-se realizar o controle da distribuição de ingressos de um Clube de futebol entre seus associados, de modo que cada associado

tenha direito a uma cota ingressos para cada jogo do clube. Para isso, será criado regras de controle de acesso para realizar esta distribuição.

Como descrito no parágrafo anterior, cada associado tem o direito a uma cota x de ingressos e se ele não solicitar sua cota ou não utilizar todos os ingressos disponíveis, eles serão disponibilizados para venda a torcedores não associados.

O associado que desejar obter sua cota de ingressos deverá se dirigir até a sede do clube onde com no seu cartão de sócio, será registrado a sua cota mediante sua adimplência com o clube conforme figura 20A. Vale salientar que o sócio terá garantido a sua cota de ingresso para o evento até 24 horas antes do seu inicio, após esse tempo, os ingressos serão disponibilizados para os torcedores não sócios.

No terminal do clube, onde o sócio solicitará sua cota com seu *Smart Card*, será registrado seu saldo de ingressos. A aplicação SACM instalada neste terminal instânciará, antes da criação das regras de controle de acesso para a solicitação, dois objetos: um para representar o evento e outro o próprio associado.

```
Client client = new Client("Jonh");
Event event  = new Event();
```

O objeto *'event'* e o *'client'* serão associado a um caminho:

```
managerObj.putObject(event, "/event");
managerObj.putObject(client, "/event/client/" + client.getName());
```

Em seguida é criada uma regra para o objeto que representa o evento que permitirá a cada associado uma cota de ingressos:

```
managerR.put(new Rule("/event","buy",0) {
  @Override
  public Boolean authorize(Map<String, Object> args) {
```

Dentro desta regra criada acima, será passado o parâmetro *"client"* para o método *get* da referência *args* que contém o objeto que representa o torcedor que deseja obter os ingressos, conforme demostrado no código abaixo:

```
Client client = (Client) args.get("client");
```

Neste momento é verificado se uma regra existe para o objeto que representa o torcedor passado, para evitar que o mesmo receba cotas duplicadas. Isso é feito através do método

constainsRule que recebe como argumento o caminho de um objeto ao qual se deseja saber se há regras vinculadas:

```
if(!ManagerRule.getInstance().containsRule("/event/client/" +client
.getName())){
```

Caso não exista regra para o objeto que representa o torcedor, será adicionado no *Map* - que será salvo no *Smart Card* - o *token* com o nome do cliente juntamente com o seu valor que é o número de *tickets* que cada sócio-torcedor receberá por padrão, como mostrado na Figura 20A nos passos 3 e 4. Vale lembrar que não existem parâmetros duplicados. Para cada chave existe somente um valor associado:

```
args.put("token" + client.getName(), 5);
```

Neste ponto será adicionado pelo terminal do clube uma regra dentro do *Smart Card* do torcedor, que será executada a posteriori quando este desejar validar a ação da requisição de ingressos no terminal do evento.

```
ManagerRule.getInstance().put(new Rule("/event/client/" + client.getName(),
"buy", 0) {
```

Na sequência desta regra criada para o associado, é resgatado o nome do torcedor para que possa ser obtido a quantidade de ingressos remanescentes de sua cota, assim sendo possível o débito da quantidade de ingressos solicitadas por ele no momento em que se encontrar no terminal de distribuição de ingressos do evento.

```
Client client = (Client) args.get("client");
Integer requestTicket = (Integer) args.get("requestTicket");
Integer allTicket = (Integer) args.get("allTicket");
Integer token = (Integer) args.get("token" + client.getName());
```

A regra será inserida dentro do *Smart Card* através do dispositivo localizado no terminal do clube, que será avaliada por outro dispositivo no terminal do evento. A regra será executada para que seja avaliada os parâmetros que estavam com o sujeito no *Smart Card*. Durante o processo de avalição da regra por este terminal, será feito uma verificação se a quantidade requisitada (*requestTicket*) é menor ou igual ao *token* criado para o cliente que armazena sua cota, e se a quantidade total de ingressos (*allTicket*) é maior ou igual a quantidade requisitada. Caso esta condição seja verdadeira, a requisição pedida é atendida e o valor solicitado de ingressos é decrementado do *token* pessoal e da quantidade total de ingressos do evento, conforme o código abaixo e os passos 1 e 2 da Figura 20B.

```
      if (requestTicket <= token && allTicket >= requestTicket){
       args.put("token" + client.getName(),token - requestTicket);
       args.put("allTicket",allTicket - requestTicket);
       return true;
      }
      return false;
    }
  });
 }
 return true;
}
```

No códigos abaixo, é demonstrado os valores dos parâmetros sendo carregados que serão utilizados na avaliação da regra da ação de solicitação de ingressos.

```
argsRule.put("allTicket", 100);
argsRule.put("requestTicket", 2);
argsRule.put("client", client);
```

Aqui é retornada a autorização de uma regra específica do cliente no momento da sua solicitação de ingressos no terminal do evento:

```
System.out.println(ManagerRule.getInstance().authorize(ManagerObject
.getInstance().getPath(event), "buy",argsRule));
```

A linha de código abaixo mostrará o conjunto de regras existentes:

```
for (String rule : ManagerRule.getInstance().listRules()) {
    System.out.println(rule);
}
```

E aqui é listado o conjunto de clientes existentes:

```
for (Object value : argsRule.keySet()) {
    System.out.println(value);
}
```

6.4.1 Exemplo com *interface* gráfica

Na Figura 21 é demonstrada a *interface* do administrador, onde podemos colocar a quantidade total de ingressos do evento, a quantidade máxima de ingressos por sócio e a data limite do evento. Na Figura 22 são criados os sócios que automaticamente recebem a quantidade configurada de ingressos.

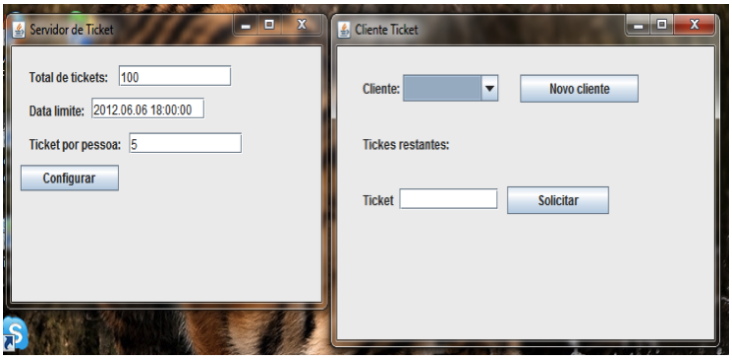


Figura 21: Interface de distribuição dos ingressos

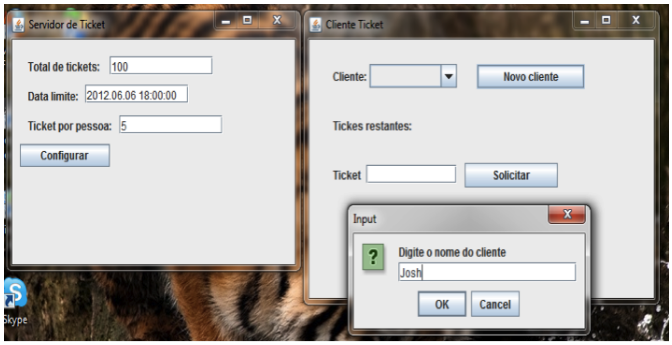


Figura 22: Criação dos clientes

Na Figura 23 é simulado uma tentativa de pedido de ingresso, como o pedido estava dentro da cota do sócio, ele foi atendido e em seguido foi decrementado do total dele. Já na Figura 24 simulamos um pedido maior do que a cota do sócio fazendo com que o sistema envie um erro.

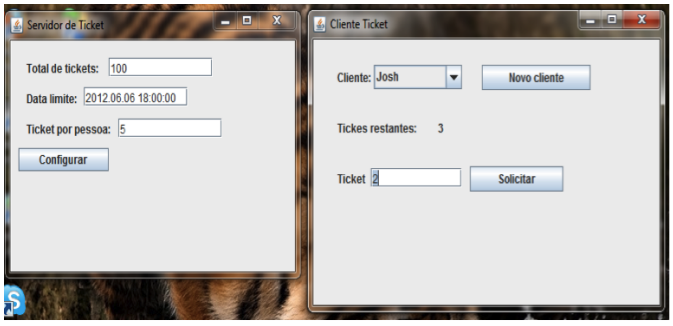


Figura 23: Solicitação de ingresso

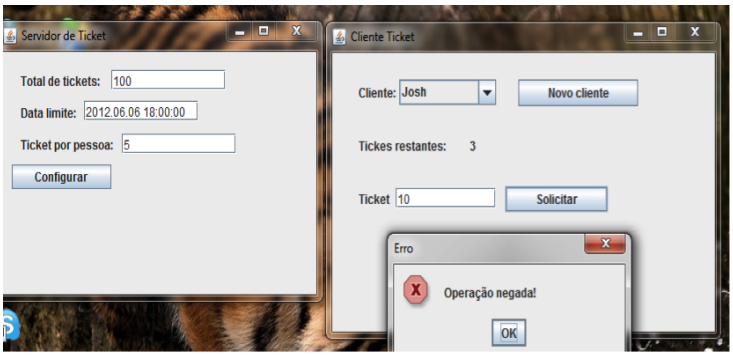


Figura 24: Solicitação negada

6.5 Exemplo com Sensibilidade ao Contexto

Neste último exemplo será exposto um cenário onde em um Hospital é necessário gerenciar o acesso das informações dos pacientes por: Médicos e Enfermeiras. Para que uma Enfermeira ou Médico possa obter uma informação sobre um paciente é necessário que este tenha um dispositivo móvel para visualizar a informação desejada. O dispositivo móvel que está em posse do funcionário requisitará as informações do paciente a uma Base de Dados ligada a rede de computadores do Hospital.

A aplicação SACM estará junto ao terminal onde se encontra a Base de Dados com as informações referentes aos pacientes. As regras de controle de acesso utilizarão informações relacionadas ao sujeito que quer obter acesso ao banco de dados da aplicação e o contexto em que tal sujeito está inserido no Hospital para permitir ou negar a autorização de acesso a base de dados.

Sensores que monitoram o ambiente estarão espalhados pelo Hospital e enviarão dados em tempo real para aplicação SACM. Esses sensores avaliarão as informações dos funcionários contidos em chips com tecnologia RFID nos crachás.

Em uma enfermaria, através de uma rede sem fio, a Enfermeira poderá por meio de um dispositivo móvel, solicitar informações de um paciente e se a mesma desejar editar estas informações, o SACM, baseando-se na presença ou ausência de um Médico no mesmo local, autorizará ou negará a solicitação feita pela Enfermeira.

A partir do momento em que a Enfermeira entra na enfermaria dentro do seu horário de trabalho, conforme a Figura 25A, os sensores do ambiente captam e avisam ao Interpretador SACM que criará uma regra de leitura para a Enfermeira acessar as informações dos pacientes que se encontram na mesma enfermaria. No instante em que um Médico entrar no mesmo local (Figura 25B), os sensores enviarão essa informação para o Interpretador que automaticamente criará a regra de leitura e escrita para o acesso do Médico as informações dos pacientes. Caso a Enfermeira queira efetuar alguma alteração no prontuário de um determinado paciente, essa

ação só será permitida mediante a existência da regra do Médico. Caso não exista essa regra, a ação será negada, caso contrário, a ação será permitida. Se no momento em que a Enfermeira estiver efetuando as alterações no prontuário do paciente, o Médico deixar a enfermaria, ela perderá o direito de continuar efetuando as modificações, voltando a ficar apenas com o direito de leitura.

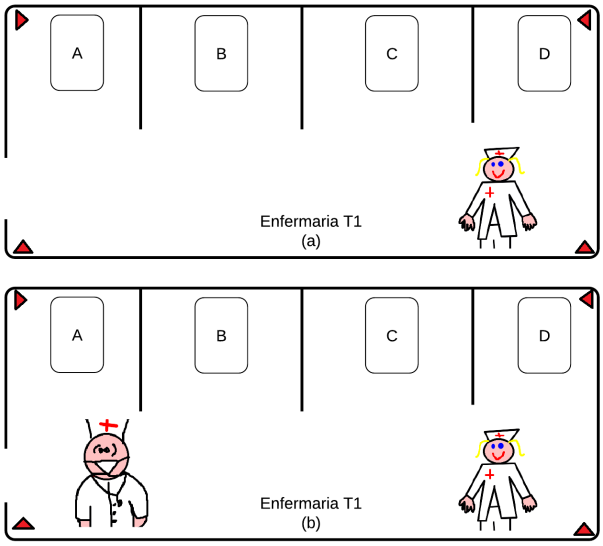


Figura 25: Sensibilidade ao Contexto em um hospital

No momento da inicialização do sistema será instânciado na aplicação SACM três objetos que representarão o contexto e os sujeitos presentes neste exemplo. O objeto *'room'* representará a enfermaria onde o Médico ou a Enfermeira poderão estar localizados, representando o contexto do ambiente. O objeto *'nurse'* e *'doctor'* serão análogos a Enfermeira e ao Médico que são os sujeitos ativos no contexto. Segue abaixo o código que representa a criação dos objetos.

```
Room room = new Room();
Nurse nurse  = new Nurse();
Doctor doctor = new Doctor();
```

Para possibilitar a criação de regras a estes objetos, eles serão associados aos seus respectivos caminhos.

```
managerObj.putObject(room, "/hospital/ward/room");
managerObj.putObject(nurse, "/hospital/ward/room/nurse");
managerObj.putObject(doctor, "/hospital/ward/room/doctor");
managerObj.putObject(patient, "/hospital/ward/room/patient");
```

Agora será criada uma regra destinada ao objeto *'room'*, a ação *'access'* e o ID que é zero. Esta regra está associada com o controle do acesso ao objeto que representa o ambiente deste exemplo.

```
managerR.put(new Rule(ManagerObject.getInstance().getPath(room),"access",0){
@Override
public Boolean authorize(Map<String, Object> args) {
```

Neste ponto, serão criadas duas variáveis de referência do tipo *Date*. Uma para indicar o início e a outra para o final do intervalo de trabalho dos funcionários na enfermaria.

```
Date timeStart;
Date timeStop;
```

De forma sequencial as variáveis declaradas anteriormente são definidas e a hora local é associada a variável *'timeNow'*, para que assim possa ser feita a comparação entre a hora em que a regra é validada e o intervalo onde o acesso é permitido.

```
try {
  timeStart = new SimpleDateFormat("HH:mm").parse("06:00");
  timeStop = new SimpleDateFormat("HH:mm").parse("23:00");
  Date timeNow = Calendar.getInstance().getTime();
```

No código abaixo será feita a verificação se a hora de acesso ao objeto está dento do intervalo definido:

```
      if(timeNow.getHours() >= timeStart.getHours() &&
      timeNow.getHours() <= timeStop.getHours()){
       return true;
      }
} catch (ParseException e) {
      e.printStackTrace(); }
      return false;
}
});
```

A segunda regra que será criada em sequência para o objeto *'room'* verificará se existe regras de acesso para o Médico indicando sua presença. Se positivo, criará regras que permitem a edição e visualização de informações sobre um paciente solicitado, caso contrário, criará uma regra que permitirá apenas a visualização de informações por parte das Enfermeiras.

```
managerR.put(new Rule(ManagerObject.getInstance().getPath(room),"access",1) {
    @Override
    public Boolean authorize(Map<String, Object> args) {
```

Será feita a verificação para saber se existe uma regra para Enfermeira (referência *nurse*).

```
if(!ManagerRule.getInstance().containsRule("/hospital/ward/room/nurse",
"read")) {
```

Se a regra não existe, será criada uma uma regra que possibilitará a Enfermeira visualizar a informações do paciente.

```
  ManagerRule.getInstance().put(new Rule("/hospital/ward/room/nurse",
  "read",0) {
    @Override
    public Boolean authorize(Map<String, Object> args) {
     if(!ManagerRule.getInstance().authorize("/hospital/ward/room", "access",
     0, args)){
      setFlagDirty(true);
      return false;
     }
     return true;
    }
  });
}
```

Neste ponto é verificada a existência da regra que indica a presença do Médico:

```
if(ManagerRule.getInstance().containsRule("/hospital/ward/room/doctor",
"write")) {
```

Caso a regra verificada exista, poderá ser criada uma nova regra que permitirá a edição dos dados do paciente por parte da Enfermaria, enquanto o Médico estiver presente no mesmo local.

```
if(!ManagerRule.getInstance().containsRule("/hospital/ward/room/nurse",
"write")){
```

```
ManagerRule.getInstance().put(new Rule("/hospital/ward/room/nurse",
"write",0){
  @Override
  public Boolean authorize(Map<String, Object> args) {
```

Note que, caso o Médico saia da ala, sua regra será excluída, fazendo com que a regra de escrita da Enfermeira, caso exista, seja invalidada através da *flag Dirty*.

```
                    if(!ManagerRule.getInstance().authorize(
                        "/hospital/ward/room","access",0,args)
                        || !ManagerRule.getInstance().containsRule(
                        "/hospital/ward/room/doctor","write")){
                        setFlagDirty(true);
                            return false;
                        }
                        return true;
                    }
                });
            }
        }
        return true;
    }
});
```

Neste ponto será criada uma regra para o paciente que validará as permissões de acesso sobre o objeto que desejar executar a ação de leitura, podendo ser ele um Médico ou Enfermeira sobre o objeto paciente.

```
managerR.put(new Rule(ManagerObject.getInstance()
.getPath(patient), "getInformation",0) {
  @Override
  public Boolean authorize(Map<String, Object> args) {
    String path = (String) args.get("path");
    if(ManagerRule.getInstance().containsRule(path, "read")){
    return ManagerRule.getInstance().authorize(path, "read", args);
    }
    return false;
}
});
```

A regra seguinte funcionará de forma semelhante a anterior mudando somente a ação executada sobre o paciente, sendo essa de escrita.

```
managerR.put(new Rule(ManagerObject.getInstance().getPath(patient),
"setInformation",0) {
@Override
public Boolean authorize(Map<String, Object> args) {,
    String path = (String) args.get("path");
    if(ManagerRule.getInstance().containsRule(path, "write")){
    return ManagerRule.getInstance().authorize(path,"write",args);
    }
    return false;
  }
});
```

Abaixo é demonstrada a execução da ação de leitura e escrita por parte da Enfermeira:

```
argsRule.put(‘‘path’’, managerObj.getPath(nurce));
ManagerRule.getInstance().authorize("/hospital/ward/room/patient",
"getInformation",argsRule);
```

Nesta segunda parte, a validação da regra que modifica as informações do paciente retornará um resultado negativo, pois a regra referente ao Médico não foi criada.

```
ManagerRule.getInstance().authorize("/hospital/ward/room/patient",
"setInformation",argsRule);
```

O código abaixo exemplifica uma regra que será criada no momento em que a presença do Médico é detectada:

```
ManagerRule.getInstance().put(new Rule("/hospital/ward/room/doctor","write",0) {
    @Override
    public Boolean authorize(Map<String, Object> args) {
      if(!ManagerRule.getInstance().authorize(getPath(), getAction(), 0, args)){
       setFlagDirty(true);
       return false;
}
      return true;
    }
});
```

Após a adição da regra referente ao Médico, poderá ser criada uma regra para Enfermaria que poderá permitir a edição de informações dos pacientes enquanto a regra de acesso para o Médico estiver ativa.

6.5.1 Exemplo com *interface* gráfica

De acordo com o conceito da sensibilidade ao contexto, caso o Médico se encontre ausente, a Enfermeira só terá acesso para leitura. Se ele estiver presente, será criada uma regra que foi engatilhada por um sensor que identificou o Médico e a partir desse momento a Enfermeira terá permissão de escrita, enquanto ele estiver presente. A Figura 26 mostra a Enfermeira solicitando acesso de leitura e recebendo a autorização.

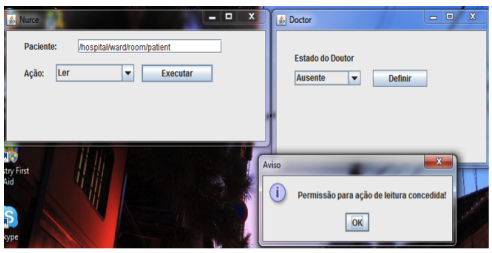


Figura 26: Acesso de enfermeira do tipo leitura

Na Figura 27, é mostrada a Enfermeira pedindo a autorização para escrita, porém, devido não haver nenhum Médico presente na mesma ala, o acesso é negado. Na Figura 28 mostra que devido o Médico se encontrar presente na ala, a Enfermeira poderá receber a autorização de escrita.

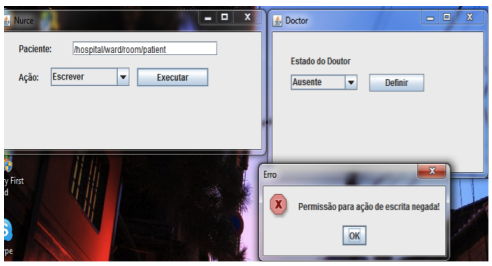


Figura 27: Negação do acesso devido a ausência do Médico

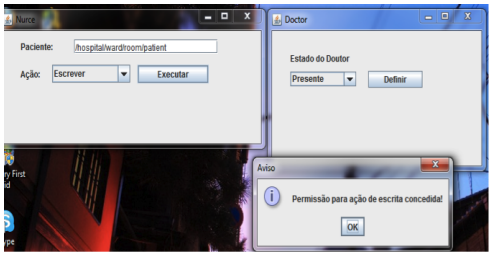


Figura 28: Permissão de escrita devido a presença do Médico

7 CONSIDERAÇÕES FINAIS

Neste trabalho propomos novas características em um modelo de controle de acesso que ajudarão com os novos paradigmas de controle da atualidade. Esse modelo acomoda políticas que necessitam de informações sobre operações anteriores *(statefull)*, a fim de tomar decisões de autorização. Mostramos a importância de manter o estado através de contadores e das modificações dinâmicas de regras em novos ambientes móveis sendo propostos pela computação ubíqua ou pervasiva. Através da característica de regras mais gerais poderem criar ou remover regras mais específicas, podemos concluir que o modelo se adequa totalmente a ambientes sensíveis ao contexto, pois uma simples modificação no ambiente poderá iniciar a inclusão ou exclusão de uma determinada regra. Essas novas capacidades permitirão a políticas que anteriormente não podiam ser expressadas, serem expressas em um modelo generalizado de controle de acesso, ao invés de soluções personalizadas.

Nesta dissertação foi proposto o *Stateful Access Control Model - SACM* (Modelo de Controle de Acesso baseado no estado anterior), como o modelo que inclui essas novas características. Além disso, ele é capaz de utilizar uma característica que é pouco utilizada na computação móvel, que é a capacidade para distribuir a informação de controlo de acesso em dispositivos protegidos de forma a não depender de estruturas centralizadas. Através dos exemplos das políticas e do conjunto de regras demonstrados, apresentamos a versatilidade do nosso modelo para representar uma variedade de novos paradigmas existentes, bem como, para as políticas tradicionais.

A implementação foi feita em JAVA com o objetivo de verificar a viabilidade das ideias do modelo e o quanto de espaço a implementação ocupará para mensurar o impacto sobre equipamentos que tem limitações de memória e processamento. O próximos passos do projeto serão a implementação de um interpretador para viabilizar as configuração de regras de forma mais amigável, evitando que o administrador do sistema tenha que possuir um conhecimento avançado em programação e testar essa implementação em cartões inteligentes para analisar o desempenho sobre ele, verificando como dispositivos com recursos limitados podem avaliar regras complexas.

Além de o SACM ser adequado para ambientes móveis e está preparado para ser centralizado ou descentralizado, queremos que este sistema seja capaz de suportar os requisitos de segurança existentes na computação em nuvem, por se tratar de um assunto pouco explorado.

REFERÊNCIAS

BERTINO, Elisa et al. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 23, p. 231–285, September 1998. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/293910.293151>.

BERTINO, Elisa; BONATTI, Piero Andrea; FERRARI, Elena. Trbac: A temporal role-based access control model. *ACM Trans. Inf. Syst. Secur.*, ACM, New York, NY, USA, v. 4, p. 191–233, August 2001. ISSN 1094-9224. Disponível em: <http://doi.acm.org/10.1145/501978.501979>.

BRAGA, Thais Regina de Moura. *Tratamento de Conflitos Coletivos em Sistemas Ubíquos Cientes de Contexto*. Tese (Doutorado em Ciência da Computação) — Universidade Federal de Minas Gerais, Belo Horizonte, 2010.

BREWER, Dr. David F.C.; NASH, Dr. Micheal J. The chinese wall security policy. *Security and Privacy, IEEE Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 206, 1989. ISSN 1540-7993.

DAMIANOU, Nicodemos et al. The ponder policy specification language. In: *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*. London, UK: Springer-Verlag, 2001. (POLICY '01), p. 18–38. ISBN 3-540-41610-2. Disponível em: <http://portal.acm.org/citation.cfm?id=646962.712108>.

HARRISON, Michael A.; RUZZO, Walter L.; ULLMAN, Jeffrey D. Protection in operating systems. *Commun. ACM*, ACM, New York, NY, USA, v. 19, p. 461–471, August 1976. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/360303.360333>.

JAJODIA, Sushil; SAMARATI, Pierangela; SUBRAHMANIAN, V. S. A logical language for expressing authorizations. *Security and Privacy, IEEE Symposium on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 0031, 1997. ISSN 1540-7993.

JOSHI, James B. D. *A generalized temporal role based access control model for developing secure systems*. Tese (Doutorado), West Lafayette, IN, USA, 2003. AAI3113822.

KULKARNI, Devdatta; TRIPATHI, Anand. Context-aware role-based access control in pervasive computing systems. In: *Proceedings of the 13th ACM symposium on Access control models and technologies*. New York, NY, USA: ACM, 2008. (SACMAT '08), p. 113–122. ISBN 978-1-60558-129-3. Disponível em: <http://doi.acm.org/10.1145/1377836.1377854>.

LAMPSON, Butler W. Protection. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 8, p. 18–24, January 1974. ISSN 0163-5980. Disponível em: <http://doi.acm.org/10.1145/775265.775268>.

LUPU, Emil et al. Amuse: autonomic management of ubiquitous e-health systems. *Concurrency and Computation: Practice and Experience*, v. 20, n. 3, p. 277–295, 2008.

MILLER, Donald V.; BALDWIN, Robert W. Access control by boolean expression evaluation. In: *Computer Security Applications Conference, 1989., Fifth Annual*. [S.l.: s.n.], 1989. p. 131–139.

OSBORN, Sylvia; SANDHU, Ravi; MUNAWER, Qamar. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur.*, ACM, New York, NY, USA, v. 3, p. 85–106, May 2000. ISSN 1094-9224. Disponível em: <http://doi.acm.org/10.1145/354876.354878>.

PONDER2: Site. 2012. Disponível em: <http://www.ponder2.net>. Acesso em: 20 de abril de 2012.

SAMARATI, Pierangela; VIMERCATI, Sabrina De Capitani di. Access control: Policies, models, and mechanisms. In: *Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design: Tutorial Lectures*. London, UK, UK: Springer-Verlag, 2001. (FOSAD '00), p. 137–196. ISBN 3-540-42896-8. Disponível em: <http://dl.acm.org/citation.cfm?id=646206.683112>.

SANDHU, R.S. et al. Role-based access control models. *Computer*, v. 29, n. 2, p. 38 –47, fev. 1996. ISSN 0018-9162.

SANDHU, Ravi; FERRAIOLO, David; KUHN, Richard. The nist model for role-based access control: Towards a unified standard. 2000. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=?doi=10.1.1.43.1355>.

SANDHU, R.; SAMARATI, P. Access control: Principles and practice. *IEEE Communications*, v. 32, n. 9, p. 40–48, September 1994.

SANTOS, André L. M. dos; KEMMERER, Richard A. Safe areas of computation for secure computing with insecure applications. In: *ACSAC*. [S.l.: s.n.], 1999. p. 35–44.

SANTOS, André L. M. dos; KEMMERER, Richard A. Implementing security policies using the safe areas of computation approach. In: *ACSAC*. [S.l.: s.n.], 2000. p. 90–99.

SANTOS, Andre L. M. dos et al. Sacm: Stateful access control model. *Local Computer Networks, Annual IEEE Conference on*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 159–162, 2011.

TWIDLE, Kevin P. et al. Ponder2 - a policy environment for autonomous pervasive systems. In: *9th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2008), 2-4 June 2008, Palisades, New York, USA*. [S.l.]: IEEE Computer Society, 2008. p. 245–246. ISBN 978-0-7695-3133-5.

VIMERCATI, Sabrina De Capitani Di et al. Access control policies and languages. *Int. J. Comput. Sci. Eng.*, Inderscience Publishers, Inderscience Publishers, Geneva, SWITZERLAND, v. 3, n. 2, p. 94–102, nov. 2007. ISSN 1742-7185. Disponível em: <http://dx.doi.org/10.1504/IJCSE.2007.015739>.

WANT, Roy et al. The active badge location system. *ACM Trans. Inf. Syst.*, ACM, New York, NY, USA, v. 10, n. 1, p. 91–102, jan. 1992. ISSN 1046-8188. Disponível em: <http://doi.acm.org/10.1145/128756.128759>.

WEISER, Mark. The computer for the 21st century. *Scientific American*, 02/1991 1991. Disponível em: <http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html>.

WOO, Thomas Y. C.; LAM, Simon S. Authorizations in distributed systems: A new approach. *Journal of Computer Security*, v. 2, p. 107–136, 1993.