UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Kátyra Kowalski Armanini

Seguraweb: Um Framework RBAC Para Aplicações Web

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Carlos Becker Westphall
Orientador

Prof.ª Drª Carla Merkle Westphall
Co-orientadora

Florianópolis, agosto de 2002.

Seguraweb: Um Framework RBAC Para Aplicações Web

Kátyra Kowalski Armanini

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Banca Examinadora

_____

Fernando Álvaro Ostuni Gauthier, Dr.
INE, UFSC

_____

Carlos Becker Westphall, Dr. (orientador)
INE, UFSC

_____

Carla Merckle Westphall, Drª (co-orientadora)
INE, UFSC

_____

Arthur Ronald de Vallauris Buchsbaum, Dr.
INE, UFSC

_____

Joaquim Celestino Junior, Dr.
DEC, UECE

A Deus
À minha família.
A duas pessoas muito especiais.

Agradecimentos

Agradeço em primeiro lugar a Deus, aos meus pais, Eliane e Paulo e aos meus irmãos Paulinho e Karen, que apesar de todos os problemas sempre me dão apoio e força para não desistir dos meus objetivos.

Aos meus orientadores Carlos Becker Westphall e Carla Merkle Westphall, pela orientação, pelas críticas, pela paciência e por terem sido compreensivos. Aos membros da banca, pelas críticas e sugestões.

Ao pessoal da Thermus, pela grande amizade; principalmente ao André Mello Barotto, que foi quem deu a idéia inicial deste trabalho; ao Adriano de Souza pelo trabalho em conjunto; e aos meus chefes Marcos e Mirian, pela sua colaboração e compreensão.

A todos os meus amigos, principalmente a Carlinha, ao Bim e ao Juliano, por terem me agüentado nos momentos de desânimo. Agradeço também pelos conselhos importantes do Adriano.

Sumário

Lista de Abreviaturas

API : *Application Program Inteface*

ASP : *Active Server Pages*

AWT : *Abstract Window Toolkit*

CDR : *Call Detail Record*

CORBA : *Common Object Request Broker Architecture*

DAC : *Discretionary Access Control*

DCOM : *Distributed Common Object Model*

DSOM : *Distributed System Object Model*

HTTP : *Hypertext Transfer Protocol*

JAAS : *Java Authentication and Authorization Service*

JCE : *Java Cryptography Extension*

JDBC : *Java Database Connectivity*

JFC : *Java Foundation Classes*

JSSE : *Java Secure Socket Extension*

JSP : *Java Server Pages*

JVM : *Java Virtual Machine*

LDAP : *Lightweight Directory Access Protocol*

MAC : *Mandatory Access Control*

MFC : *Microsoft Foundation Classes*

NIST : *National Institute of Standards and Technology*

OCL : *Object Constraints Language*

OID : *Object Identificator*

OLE : *Object Linking and Embedding*

ORB : *Object Request Broker*

PBE : *Password-Based Encryption*

RBAC : *Role-Based Access Control*

RMI : *Remote Method Invocation*

SSL : *Secure Socket Layer*

UML : *Unified Modeling Language*

URL : *Uniform Resource Locator*

Lista de Figuras

Resumo

Este trabalho apresenta a criação de um *framework* orientado a objeto baseado no modelo de segurança RBAC que pode ser utilizado no desenvolvimento de aplicações Web ou em quaisquer outras que utilizam Java. Este *framework* é composto por várias classes (concretas e abstratas) que implementam mecanismos de autenticação, controle de acesso, auditoria e administração. O *framework* ajuda a reduzir o tempo de desenvolvimento das aplicações, retirando dos desenvolvedores das aplicações a preocupação com a implementação dos mecanismos de segurança, principalmente no que se refere ao de controle de acesso.

Abstract

This work presents the creation of a object-oriented framework based on RBAC model. This framework can be used in the development of Web applications any other applications that uses Java. It is composed by several classes (concrete and abstract) and implements authentication mechanisms, access control, auditing and administration. The framework helps to reduce the development time of new applications, because the developers do not need to concern with the implementation of security mechanisms, mainly access control.

Capítulo 1

Introdução

1.1 Motivação

Os *sites* Web aumentam o seu poder adicionando mais capacidades de consultas e atualizações em banco de dados, o que despertou o interesse das empresas em aplicações Internet e justifica sua tendência em direção ao mercado corporativo para dentro do mundo da Intranet.

Os servidores Web privados são o que há de mais barato e poderoso para garantir uma revolução na comunicação interna da empresa, copiando o modelo da Internet, mas oferecendo acesso apenas aos usuários autorizados da rede interna da instituição ou organização. De uma forma sucinta, o que caracteriza a Intranet é o uso das tecnologias da Web no ambiente privativo da empresa.

Os elementos básicos de uma Intranet são a estrutura de rede, os servidores Web e os clientes com seus *browsers* e softwares de comunicação. As redes locais são hoje parte integrante da maioria das empresas. Mesmo organizações de pequeno porte possuem sua rede, que permite compartilhar dispositivos como impressoras e arquivos, além de possibilitar a troca de mensagens através de correio eletrônico. Portanto, a experiência no uso de redes e sua infra-estrutura básica já se encontra instalada na maioria das organizações. Basta então acrescentar um servidor Intranet a esta estrutura para se ter acesso às facilidades e características da Internet na empresa.

A Intranet, no entanto não necessariamente é uma rede interna isolada do resto do mundo. O servidor Intranet pode estar ligado a Internet, permitindo então que os usuários da empresa acessem a Internet, assim como um funcionário ou cliente da empresa pode acessar a Intranet de qualquer parte do mundo utilizando a Internet.

Em vez de circular publicamente pelo mundo, como na Internet, as informações confinadas numa rede Intranet são acessíveis apenas à organização a que pertencem e às pessoas autorizadas por ela a consultá-las.

Por suas características, esse tipo de rede é uma poderosa ferramenta de gestão empresarial e, ao mesmo tempo, um meio de viabilizar o trabalho em grupo na organização.

Quando comparada com essas soluções clássicas, a Intranet ganha no custo, na facilidade de uso e na flexibilidade.

Vejamos alguns dos motivos que levam as empresas a implantarem uma Intranet , estes são enumerados a seguir:

- Disponibilização centralizada de aplicações;
- Acesso fácil via software de baixo custo;
- Necessidade mínima de treinamento para usuários novos;
- Aplicações podem consultar e atualizar bancos de dados corporativos;
- Seu alcance universal representa uma oportunidade de atuação no mercado global;
- Abre um canal ágil entre empresas e clientes. O *feedback* dos clientes pode ser usado para melhorar a qualidade de produtos e serviços;
- Seu baixo custo reduz as despesas com divulgação e comercialização de produtos e serviços;
- A Web está associada a uma imagem de modernidade e agilidade que interessa a todas as empresas;
- É um meio democrático, oferecendo chances iguais de exposição tanto para as grandes empresas quanto para as pequenas.

Com a utilização cada vez maior da Internet e das Intranets dentro das empresas, o crescimento do uso de aplicações Web se torna inevitável. Considerando que em uma empresa existem funcionários com atividades e funções diferentes, onde alguns possuem mais privilégios que outros, existe a necessidade de controlar os privilégios dos funcionários na utilização das aplicações Web da empresa. As principais preocupações com relação à segurança são a confidencialidade, as informações só são reveladas para pessoas autorizadas; integridade, a informação só pode ser modificada por funcionários autorizados das maneiras autorizadas; responsabilidade, funcionários são responsáveis por suas ações relacionadas à segurança; e acessibilidade (disponibilidade), funcionários autorizados não podem ter seu acesso negado maliciosamente.

Além disso, com o crescimento das empresas ocorreu um aumento do volume de informação e do número de funcionários, e conseqüentemente os problemas de segurança cresceram e se tornaram mais difíceis [OH00].

O controle de acesso por si só não é uma solução completa para obtenção de segurança em um sistema. Para que a segurança seja completa, é necessária a existência de outros serviços de segurança como autenticação, auditoria e administração [SAN94]. A autenticação é para garantir que um usuário é quem ele diz ser; controles de acesso, para controlar o que um usuário pode acessar em um sistema; comunicações seguras, para proteger informações em trânsito entre componentes; auditoria de segurança, para gravar e analisar o que o usuário faz no sistema; e administração de segurança, para gerenciar informações de segurança incluindo as políticas de segurança.

De acordo com um estudo do NIST (*National Institute of Standards and Technology*), essa necessidade de controles de acesso de acordo com os papéis que os usuários e ou funcionários individualmente desempenham na organização define uma política de segurança denominada de RBAC (*Role-Based Access Control*), ou seja, um modelo de segurança baseado em papéis [WES00] .

O *Role Based Access Control* (RBAC) é um termo utilizado para descrever políticas de segurança que controlam o acesso de usuários a recursos computacionais, baseado na construção de *roles* (papéis, funções). Esses papéis definem um conjunto de atividades concedidas para usuários autorizados. Pode-se imaginar um papel como se fosse um cargo ou posição dentro de uma organização, que representa a autoridade necessária para conduzir as tarefas associadas. Para muitos tipos de organização, o modelo RBAC fornece um modo mais intuitivo e eficaz de representar e gerenciar autorizações às informações que outras formas de controle de acesso [JAN98] [OH00].

Outra motivação deste trabalho é a reusabilidade de componentes, que atualmente é reconhecida como uma forma importante de aumentar a produtividade no desenvolvimento de software. Os programadores com mais experiência em sua maioria já reutilizam código e consultam códigos antigos, mas o potencial de reutilização das partes de análise e projeto de um sistema ainda é pouco explorado [LAN95] .

Dentro deste contexto existe o conceito de *frameworks*, que torna possível não somente a reutilização de código, mas também da parte de análise e projeto [LAN95] . Um *framework* nada mais é do que uma estrutura de classes inter-relacionadas, que

corresponde a uma implementação incompleta para um conjunto de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas [SIL00].

1.2 Objetivos

Observando este cenário, o objetivo deste trabalho é propor a criação de um *framework* baseado no modelo de segurança RBAC e que possa ser utilizado por aplicações Web. Este *framework* é composto por várias classes (concretas e abstratas) que implementam mecanismos de autenticação, controle de acesso, auditoria e administração. A utilidade deste *framework* está na diminuição do tempo de desenvolvimento da parte de segurança de aplicações Web ou de aplicações Java que utilizem a política de papéis.

Como o propósito deste *framework* é para ser utilizado principalmente por aplicações com interface Web, a linguagem de implementação escolhida foi Java e o item de segurança de comunicação segura omitido, já que no ambiente proposto este item é garantido pelo protocolo SSL(*Secure Socket Layer*), o qual já se encontra implementado na maioria dos servidores Web existentes no mercado. Nada impede que a especificação do *framework* seja utilizada para implementação em outra linguagem de programação e seja utilizada para qualquer tipo de aplicação que queira adotar o modelo de segurança RBAC. Neste caso, é sugerido que haja uma preocupação com relação à parte de comunicação segura.

Para alcançar este objetivo geral, os seguintes específicos foram definidos:

- Estudo do modelo de segurança RBAC;
- Estudo dos mecanismos de segurança: controle de acesso, criptografia, autenticação;
- Estudo do modelo de segurança Java e das APIs (*Application Program Interface*), JCE (*Java Cryptography Extension*), JSSE (*Java Secure Socket Extension*) e JAAS (*Java Authentication and Authorization Service*);

- Estudo da arquitetura e de técnicas de implementação de *frameworks*;
- Implementação do *framework* utilizando a linguagem Java;
- Criação de uma aplicação de segurança, para validação do *framework*.

1.3 Organização do Trabalho

Este trabalho está organizado em quatro capítulos. Este capítulo inicial descreveu a motivação e o contexto no qual este trabalho está inserido, além do objetivo geral e dos objetivos específicos.

O capítulo 2 apresenta a classificação dos modelos de controle de acesso, e alguns conceitos relacionados. Descrevem também os elementos e regras do modelo RBAC e a família de modelos RBAC.

O capítulo 3 apresenta os conceitos de *frameworks* e os aspectos que caracterizam um *framework*. Também são apresentados a classificação dos *frameworks* e seu ciclo de vida, bem como as fases e as metodologias de desenvolvimento de *frameworks*.

O capítulo 4 apresenta a proposta do trabalho, os trabalhos relacionados e as etapas de análise e projeto do desenvolvimento do *framework* proposto. Além disso, são apresentadas as partes da estrutura do *framework* desenvolvidas, bem como foram implementadas as regras do modelo RBAC dentro do *framework*.

Por fim, no último capítulo são apresentadas algumas conclusões e perspectivas da continuidade deste trabalho.

Capítulo 2

O Modelo de Segurança RBAC (*Role-Based Access Control*)

Existem dois conceitos básicos importantes para o entendimento do que seja controle de acesso: sujeito e objeto. Um sujeito é uma entidade ativa em um sistema computacional, o qual inicia as 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 [OBE01] .

Virtualmente todos os sistemas computacionais podem ser descritos em termos de sujeitos acessando objetos. O controle de acesso é, portanto, a mediação das requisições de acesso a objetos, iniciadas pelos sujeitos [OBE01] .

Uma política é simplesmente uma forma de relacionar sujeitos e objetos. Uma política de controle de acesso determina quais as operações são permitidas ao sujeito (autorização positiva ou acesso sem restrição) e quais são proibidas (autorização negativa ou acesso sem restrição) [YIA96] . De acordo com [RAM94], uma política de segurança é "um conjunto de regras, aplicadas sobre um domínio, que especifica o que é e o que não é permitido no campo da segurança".

Existe uma classificação quanto às políticas de controle de acesso, nas quais descrevem as três classes principais:

- **Controle de Acesso Discricionário** (*Discretionary Access Control* - DAC): Em uma política discricionária, os direitos de acesso a cada recurso, a cada informação, são manipulados livremente pelo responsável do recurso ou da informação (geralmente o proprietário do mesmo), segundo a sua vontade (à sua discrição). A gestão de acesso aos arquivos Unix constituem um exemplo de controles baseados em políticas discricionárias. O proprietário de um arquivo, nesse caso, pode atribuir livremente ou não os direitos (ler, escrever, executar) a ele mesmo, a um grupo de usuários e a outros usuários. As políticas discricionárias não são coerentes [WES00] , ou seja, podem ter estados inseguros. Isto porque é

o usuário quem atribui livremente os direitos de acesso. A flexibilidade das políticas de controle de acesso discricionário faz com que sejam satisfatórios para uma variedade de sistemas e aplicações. Por estas razões estas políticas têm sido amplamente utilizadas em vários tipos de implementações, especialmente em ambientes comerciais e industriais [SAN94].

- **Controle de Acesso Obrigatório** (*Mandatory Access Control* - MAC): As políticas ditas obrigatórias ou não-discricionárias resumem em seus esquemas de autorização um conjunto de regras rígidas que expressam um tipo de organização envolvendo a segurança das informações no sistema como um todo. A política obrigatória supõe que os usuários e objetos ou recursos do sistema estão todos etiquetados. As etiquetas dos objetos seguem uma classificação específica enquanto os usuários ou os sujeitos do acesso possuem níveis de habilitação. Os controles que determinam as autorizações de acesso são baseados na comparação da habilitação do usuário com a classificação do objeto. As regras definidas nesses controles que são ditas incontornáveis asseguram que o sistema verifique as propriedades de confidencialidade e de integridade [WES00] .
- **Controle de Acesso Baseado em Papéis:** Com o controle de acesso baseado em papéis, as decisões de acesso são baseadas nos papéis que os próprios usuários têm como parte de uma organização. Os usuários assumem determinados papéis (como por exemplo, médico, enfermeiro, caixa de banco, gerente etc). O processo de definição de papéis é baseado na análise completa de como uma organização funciona e pode abranger uma grande variedade de usuários em uma organização [NIBU98] .

O termo *Role Based Access Control* (RBAC) é utilizado para descrever mecanismos de segurança que controlam o acesso de usuários a recursos computacionais, baseado na construção de papéis. Esses papéis definem um conjunto de atividades concedidas para usuários autorizados. Pode-se imaginar um papel como se fosse um cargo ou posição dentro de uma organização, que representa a autoridade necessária para conduzir as tarefas associadas [JAN98] . Os usuários pertencem a um

papel de acordo com suas responsabilidades e qualificações e podem ser facilmente transferidos sem modificar a estrutura de acesso básica. Novas permissões, bem como a incorporação de novas aplicações e ações, podem ser concedidas aos papéis, e as permissões podem ser revogadas quando necessário [FER95] . Para muitos tipos de organização, o modelo RBAC fornece um modo mais intuitivo e eficaz de representar e gerenciar autorizações às informações em comparação a outras formas de controle de acesso [JAN98] .

Existe um forte sentimento nos pesquisadores e especialistas de que muitos requisitos práticos não são cobertos pelas políticas discricionárias e obrigatórias. As políticas obrigatórias se originam em ambientes rígidos, como o ambiente militar. Já as políticas discricionárias têm origem em ambientes acadêmicos. Nenhum destes dois tipos de política satisfaz as necessidades da maioria dos ambientes empresariais [SAN94]. O modelo RBAC é uma alternativa às políticas de controle de acesso discricionária e obrigatória, e está cada vez mais atraindo a atenção, principalmente para aplicações comerciais.

As políticas baseadas em papéis se beneficiam por sua independência lógica, a qual especifica as autorizações de um usuário em duas partes: uma que relaciona os usuários aos papéis e outra que relaciona estes papéis aos direitos de acesso a um objeto. Isto simplifica muito a administração de segurança. Por exemplo, supondo o caso de um funcionário de uma empresa que recebe uma promoção, isto significa que suas atividades e responsabilidades irão mudar. Para realizar a mudança de autorizações deste funcionário, basta retirar as associações com os papéis atuais e associar novos papéis apropriados para o novo cargo. Se todas as autorizações fossem entre usuários e objetos diretamente, seria necessário retirar todos os direitos de acesso existentes ao usuário e associar os novos direitos de acesso. Esta é uma tarefa trabalhosa e que consome tempo [SAN94], principalmente em grandes empresas, onde a quantidade de usuários é em torno de dezenas de centenas de milhares [SAN98a][SAN99].

Nos últimos anos, os grandes fabricantes de *software* começaram a implementar as características do modelo RBAC em bases de dados, no gerenciamento de sistemas e sistemas operacionais, mas sem nenhuma concordância geral do que realmente constitui um conjunto apropriado de características do modelo RBAC [FER99].

Com RBAC, a segurança é gerenciada em um nível muito próximo à estrutura da organização. Cada usuário está associado a um ou mais papéis, e cada papel está

associado a um ou mais privilégios que são concedidos aos usuários daquele papel. Os papéis podem possuir hierarquia. Por exemplo, alguns papéis em um banco podem ser pessoas responsáveis pelo atendimento a clientes, caixas, gerente e diretor. O papel de gerente pode incluir todos os privilégios do papel de caixa e corretor de seguros, que por sua vez inclui todos os privilégios disponíveis ao papel de atendente. A administração de segurança com RBAC consiste em determinar as operações que devem ser executadas por pessoas em tarefas específicas e associar os empregados aos papéis apropriados.

2.1 Características do Modelo RBAC

As políticas RBAC são descritas em termos de usuários, sujeitos (*subjects*), papéis, hierarquia de papéis, operações e objetos (que são os alvos de proteção). Para executar uma operação em um objeto controlado por RBAC, o usuário deve estar ativo em algum papel. Antes de um usuário poder estar ativo em um papel, tal usuário deve primeiro ser autorizado como um membro do papel pelo administrador do sistema.

Como pode ser visto na Figura 1, um usuário pode ser membro de vários papéis, e um papel pode ter múltiplos membros, formando uma associação de muitos-para-muitos. Da mesma forma um papel pode ter várias permissões, e uma mesma permissão pode estar associada a muitos papéis, também formando uma associação de muitos-para-muitos.

Figura 1 - Elementos Gerais do Modelo RBAC.

O RBAC proporciona aos administradores a capacidade de empregar restrições na autorização do papel, na ativação do papel e na execução da operação. Essas restrições possuem uma variedade de formas [FER95] .

As subseções seguintes definem as entidades RBAC e proporcionam uma definição precisa para várias restrições representativas.

2.2 Usuários, Papéis e Permissões

Em um *framework* RBAC, um *usuário* é uma pessoa, um *papel* é um conjunto de funções de trabalho e uma *operação* representa um modo específico de acesso a um conjunto de um ou mais *objetos* RBAC protegidos. Como apresentado na Figura 2, um sujeito (*subject*) representa um processo de usuário ativo com a única seta indicando um relacionamento de um para muitos.

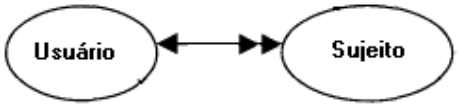


Figura 2 - Usuários e Sujeitos.

Os tipos de operações e objetos que o RBAC controla é dependente do tipo de sistema na qual ele será implementado. Por exemplo: em um sistema operacional, as operações podem incluir leitura, escrita e execução; em um sistema gerenciador de base de dados, as operações podem incluir inserção, remoção e atualização, etc. O conjunto de objetos incluídos no modelo RBAC inclui todos os objetos acessíveis pelas operações RBAC. Entretanto, nem todos os objetos do sistema e do sistema de arquivos necessitam ser incluídos em um esquema RBAC. Por exemplo, o acesso aos objetos de infra-estrutura, como por exemplo, os objetos de sincronização (ex., semáforos, *pipes*, segmentos de mensagens) e objetos temporários (ex., arquivos e diretórios temporários) não precisam necessariamente ser controlados dentro do conjunto de objetos protegidos RBAC [FER95] .

Uma operação representa uma unidade de controle que pode ser referenciado por um papel em particular, que é o responsável por regular as restrições dentro do *framework* RBAC. É importante notar a diferença entre um simples modo de acesso e uma operação. Uma operação pode ser utilizada para capturar detalhes de segurança relevantes ou restrições que não podem ser determinadas por um simples modo de acesso. Estes detalhes podem ser em termos de método ou granularidade de acesso.

Para demonstrar a importância de uma operação RBAC, considere a diferença entre o acesso de um caixa e um supervisor de contas em um banco. A empresa define

um papel de caixa como sendo capaz de realizar uma operação de depósito em poupança. Isto requer acesso de leitura e escrita a campos específicos em um arquivo de poupança. A empresa também pode definir um papel de supervisor de conta na qual é permitido realizar operações de correção. Estas operações requerem acesso de leitura e escrita aos mesmos campos de um arquivo de poupança, como o caixa. Entretanto, o supervisor de conta não pode ter permissão para iniciar depósitos ou saques, mas somente realizar correções após estas ações terem ocorrido. Da mesma forma, não é permitido ao caixa realizar qualquer correção em transações que já foram completadas. A diferença entre estes dois papéis está nas operações que são executadas.

Para demonstrar a importância da granularidade de controle, considere a necessidade de um farmacêutico de acessar o registro de um paciente para conferir interações entre medicações e para adicionar notas na seção de medicação do registro do paciente. Embora essas operações sejam necessárias, o farmacêutico pode não ter acesso de leitura ou alteração em outras partes do registro do paciente.

Como apresentado na Figura 3, as operações estão administrativamente associadas com objetos, bem como com papéis.

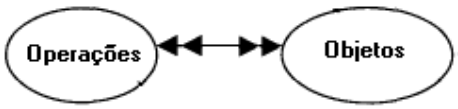


Figura 3 - Operações e Objetos.

Quando um usuário autorizado é associado a um papel, este usuário tem implicitamente o potencial de executar as operações que estão associadas com o papel. Cada operação é referenciada com um identificador único [FER95] .

2.3 Papéis e Hierarquia de Papéis

Os papéis podem ter sobreposição de responsabilidades e privilégios, ou seja, usuários pertencentes a diferentes papéis podem necessitar realizar operações comuns. Além disso, em muitas organizações existem um conjunto de operações gerais que são realizadas por todos os empregados. Como tal, se comprova ineficiente e

administrativamente incômodo especificar repetidamente essas operações gerais para cada papel que é criado. Para melhorar a eficiência e proporcionar à estrutura natural de uma empresa, RBAC inclui o conceito de hierarquias de papéis. Uma hierarquia de papel define papéis que têm atributos específicos e que podem conter outros papéis, ou seja, aquele papel pode incluir implicitamente as operações, restrições, e objetos que são associados a outro papel. As hierarquias de papéis são um modo natural de organizar papéis para refletir autoridade, responsabilidade, e competência [FER95] . Um exemplo de uma hierarquia de papel é mostrado na Figura 4. Neste exemplo, o papel de Gerente "contém" os papéis de Caixa e Atendente. Isto significa que os membros do papel de Gerente estão implicitamente associados com as operações, restrições e objetos dos papéis de Caixa e Corretor de Seguros sem a necessidade do administrador ter que listar explicitamente os atributos de Caixa e Corretor de Seguros. Os papéis mais poderosos são representados no topo do diagrama e os papéis menos poderosos estão representados embaixo, i.e., os papéis no topo do diagrama contêm o maior número de operações, restrições e objetos. Como mostrado na Figura 4, nem todos os papéis estão relacionados. Por exemplo, os papéis de Caixa e Corretor de Seguros não estão hierarquicamente relacionados, mas eles contêm o papel de Atendente em comum.

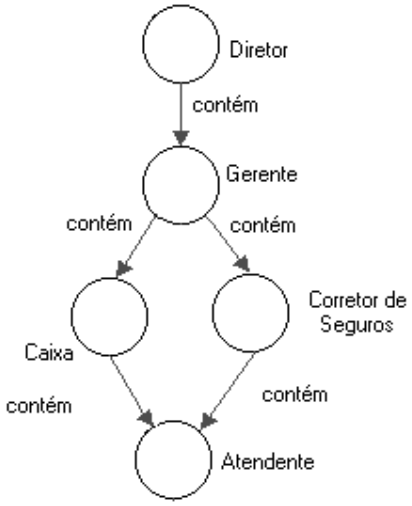


Figura 4 - Exemplo de hierarquia de papeis.

A hierarquia de papéis pode ser descrita como:

Regra 1 – Hierarquia de Papéis (*Role Hierarchy*): Se um sujeito está

autorizado a acessar um papel e este papel contém um outro papel, então ao sujeito também é permitido acessar o papel contido, como por exemplo o caso do gerente na Figura 1.

Isto garante que o papel que contém implicitamente está associado a todas as permissões associadas ao papel contido.

2.4 Autorização de Papéis

A associação de um usuário com um papel pode estar sujeito às seguintes propriedades:

- Pode não ser dado nenhum privilégio a mais ao usuário, que seja necessário para executar seu trabalho;
- O papel no qual o usuário está se tornando membro pode não ser mutuamente exclusivo com outro papel para o qual o usuário já seja membro;
- A limitação numérica que existe para a associação de um papel não pode ser excedida.

A primeira propriedade é destinada a assegurar adesão ao princípio de Menor Privilégio. O princípio de Menor Privilégio requer que não seja dado a um usuário nenhum privilégio a mais do que necessário para este executar a função de seu trabalho. Assegurando o menor privilégio exige identificar as funções do trabalho do usuário, determinando o conjunto de privilégios mínimo exigido para executar aquela função, e restringindo o usuário a um domínio com esses privilégios e nada mais. Nas implementações não-RBAC, isto é freqüentemente difícil ou caro de alcançar. Por exemplo, podem ser permitidos mais privilégios a alguém associado a uma categoria de trabalho do que a pessoa precisa, por causa da inaptidão dos sistemas em definir o acesso baseado em vários atributos ou restrições. Considerando que muitas das responsabilidades se sobrepõem entre categorias de trabalho, o privilégio de máximo para cada categoria de trabalho poderia causar acesso ilegal. O RBAC pode ser configurado de forma que só essas operações que necessitam ser executadas por membros de um papel sejam concedidas ao papel, e estas operações e papéis podem estar sujeitas a políticas organizacionais ou restrições. Nos casos onde operações se sobrepõem, podem ser estabelecidas hierarquias de papéis. No passado, auditorias minuciosas eram utilizadas para justificar a concessão de maior acesso [FER95] . Por

exemplo, pode parecer suficiente permitir aos gerentes ter acesso a todos os registros de dados de todos os clientes de todas as agências do banco, se o seu acesso é monitorado suficientemente. Porém, isto pode acarretar mais auditorias e monitoramento do que seria necessário com um mecanismo de controle de acesso mais bem definido. Com RBAC, podem ser colocadas restrições no acesso do caixa de forma que, por exemplo, somente os registros associados aos clientes da agência na qual o gerente trabalha possam ser acessados.

A segunda propriedade listada anteriormente, referente a um papel mutuamente exclusivo a outro, é destinada a preservar a política de Separação Estática de Tarefas ou conflito de interesses. Isto significa que em virtude de um usuário estar autorizado como um membro de um papel, o usuário não está autorizado como membro de um segundo papel. Por exemplo, um usuário que está autorizado para ser um membro de um papel de Caixa em um banco pode não ter permissão para ser membro de um papel de Auditor do mesmo banco. Assim dizendo, os papéis de Caixa e Auditor são mutuamente exclusivos [FER95] .

A política de Separação Estática de Tarefas pode ser especificada genericamente e ser uniformemente imposta a papéis específicos. Os papeis mutuamente exclusivos para um determinado papel e a propriedade da Separação Estática de Tarefas podem ser especificadas como segue:

mutually-exclusive-authorization(r:roles) = {a lista de papéis que são mutuamente exclusivos com o papel “r”}.

Regra 2 - Separação Estática de Tarefas (*Static Separation of Duty*): Um usuário é autorizado como membro de um papel somente se este papel não é mutuamente exclusivo a quaisquer outros papéis na qual o usuário já seja membro.

A terceira propriedade, referente à limitação numérica, concedida a um usuário membro de papéis é a propriedade da Cardinalidade. Alguns papéis só podem ser ocupados por um certo número de empregados a qualquer determinado período de tempo. Por exemplo, considere o papel de Gerente. Embora outros empregados possam agir naquele papel, só um empregado pode assumir as responsabilidades de um gerente em um certo momento. Um usuário pode se tornar um membro novo de um papel

contanto que o número de membros permitidos ao papel não esteja excedido. O número de usuários permitidos para um papel e o número existente de usuários associados com um papel é especificado pelo seguinte duas funções:

membership-limit (r: roles) = o limite de membros (≥ 0) para o papel "r."

number-of-members(r:roles) = N (≥ 0) o número existente de membros no papel "r."

A cardinalidade de um papel pode ser descrita como:

Regra 3 – Cardinalidade(*Cardinality*): A capacidade de um papel não pode ser excedida por um membro adicional ao papel.

2.5 Ativação de Papéis

Cada sujeito é um mapeamento de um usuário a um ou possivelmente a vários papéis. Um usuário estabelece uma sessão durante a qual o usuário está associado com um subconjunto de papéis dos quais ele é membro. Uma autorização de papel de um usuário (que é uma conseqüência de associação de papel) é necessária, mas nem sempre é uma condição suficiente para um usuário ter permissão de executar uma operação. Outras considerações de política organizacionais ou restrições podem precisar ser levadas em conta.

A ativação de papéis fornece o contexto para o qual estas políticas organizacionais podem ser aplicadas. Como tal, o RBAC exige que um usuário seja primeiramente autorizado, bem como ser ativado em um papel, antes que possa executar uma ação.

Dependendo da política organizacional sob consideração, verificações são aplicadas em termos do papel que está sendo proposto para ativação, a operação que está sendo requisitada para execução, e/ou o objeto que está sendo acessado. Quer dizer, um papel pode ser ativado se:

- O usuário é autorizado para o papel que está sendo proposto para ativação;
- A ativação do papel proposto não é mutuamente exclusiva a alguma outra regra ativa do usuário;
- A operação proposta é autorizada para o papel que está sendo

proposto para ativação;

- A operação que é proposta é consistente dentro de uma seqüência obrigatória de operações.

As seguintes funções habilitam os sujeitos a executarem operações RBAC e definem as regras ativas de um sujeito:

exec: (s: subject, op: operation) = {VERDADEIRO se o sujeito "s" pode executar a operação "op," caso contrário é FALSO}.

active-roles(s:subject) = {a lista atual de papéis ativos para o sujeito "s"}.

A especificação que o papel ativo proposto de um sujeito deve estar em um conjunto de papéis autorizados para aquele sujeito é declarado pela seguinte propriedade:

Regra 4 – Autorização de Papéis (*Role Authorization*): Um sujeito nunca pode ter um papel ativo que não está autorizado para aquele sujeito.

Uma vez que é determinado que um papel faz parte do conjunto de papéis autorizado para o sujeito, a operação pode ser executada contanto que o papel esteja ativo. Embora um papel possa estar no conjunto de papéis, pode haver certas políticas organizacionais (como a Separação Dinâmica de Tarefas) que impedem que o papel seja ativado. Isto fornece o contexto na qual são feitas outras verificações e que são especificadas pela seguinte regra:

Regra 5 – Execução de Papéis (*Role Execution*): Um sujeito pode executar uma operação somente se o sujeito estiver agindo dentro de um papel ativo:

O modelo RBAC também proporciona aos administradores a capacidade de impor uma política específica de organização, a Separação Dinâmica de Tarefas. A Separação Estática de Tarefas proporciona à empresa a capacidade de tratar potenciais conflitos de interesses em relação ao momento em que um usuário membro de um papel é autorizado para este. Porém, em algumas organizações é permissível a um usuário ser um membro de dois papéis que não estabeleçam conflitos de interesse quando agem independentemente, mas introduzem algumas restrições políticas quando sua ação ocorre simultaneamente [FER95] .

Por exemplo, uma política estática poderia exigir que nenhum indivíduo que tem

o papel de Iniciador de Pagamento também possa ter o papel de Autorizador de Pagamento. Apesar de que tal abordagem pode ser adequada para algumas organizações, para outras pode ser considerada muito rígida, fazendo com que o custo de separação seja maior que o esperado. O objetivo por trás da Separação Dinâmica de Tarefas é permitir maior flexibilidade em operações. A Separação Dinâmica de Tarefas estabelece restrições na ativação simultânea de papéis, como por exemplo, um usuário pode estar autorizado para ambos os papéis de Iniciador e Autorizador de Pagamento, mas pode assumir dinamicamente ao mesmo tempo um único destes papéis [FER95] .

Os papéis considerados mutuamente exclusivos a um papel ativo proposto são especificados pela seguinte função:

mutually-exclusive-activation(r:roles) = {a lista de papéis ativos que são mutuamente exclusivos ao papel "r" proposto}.

A regra de Separação Dinâmica de Tarefas do modelo RBAC é definida como:

Regra 6 – Separação Dinâmica de Tarefas (*Dynamic Separation of Duty*): Um sujeito pode se tornar ativo em um novo papel somente se o papel proposto não é mutuamente exclusivo a quaisquer outros papéis no qual o sujeito é atualmente ativo.

A especificação de que um sujeito só pode executar uma operação se esta operação é autorizada para o papel ativo proposto do sujeito, é estabelecido pela seguinte propriedade.

Regra 7 – Autorização de Operação (*Operation Authorization*): Um sujeito pode executar uma operação somente se a operação é autorizada para o papel em que o sujeito está atualmente ativo.

2.6 Separação Operacional de Tarefas

O modelo RBAC pode ser utilizado por um administrador de sistema para obrigar a política de Separação Operacional de Tarefas. A Separação Operacional de Tarefas pode ser um método valioso para intimidar fraudes. Isto está baseado na idéia de que a fraude pode acontecer se existir colaboração entre várias capacidades relacionadas ao trabalho dentro de uma função empresarial crítica. Por exemplo, a função de comprar

um artigo poderia envolver as seguintes operações: autorização da ordem de compra; registro da chegada da fatura; registro da chegada do artigo; e, finalmente, autorização de pagamento[1]. Se cada uma destas operações é executada através de papéis diferentes, a probabilidade de fraude pode ser reduzida. Se for permitido a um usuário executar todas as operações, a fraude pode ocorrer.

A Separação Operacional de Tarefas estabelece que para todas as operações associadas com uma determinada função de negócio, nenhum usuário individualmente pode ter permissão para executar todas essas operações. Portanto, pode ser detectada pela organização uma falha na execução de um papel. Nos termos do modelo RBAC, a Separação Operacional de Tarefas pode ser imposta quando papéis são autorizados para usuários individuais e quando operações são associadas aos papéis [FER95] .

A Separação Operacional de Tarefas pode ser especificada pelas seguintes função e propriedade:

function-operations(f:function) = {o conjunto de todas as operações necessárias para a função de negócio "f"}.

Regra 8 – Separação Operacional de Tarefas (*Operational Separation of Duty*): Um papel pode ser associado com uma operação de uma função de negócio somente se o papel for autorizado para o sujeito e que este papel não tenha sido associado anteriormente para todas as outras operações.

2.7 Acesso de Objetos

Para assegurar a execução das políticas da empresa para os objetos RBAC, o acesso do sujeito aos objetos RBAC deve ser controlado. A seguinte função é utilizada para determinar se um sujeito pode acessar um objeto RBAC:

access (s: subject, o: object) = {VERDADEIRO se o sujeito tem acesso ao objeto, caso contrário é FALSO}.

Com as propriedades de Autorização de Papel e Execução de Papel definidas anteriormente (Regras 3 e 4), a propriedade de Autorização de Acesso a Objeto definida abaixo assegura que o acesso de um sujeito a um objeto RBAC só pode ser obtido

[1] A execução de uma operação de uma função de negócio só prossegue se a operação anterior for concluída com êxito.

através de operações autorizadas por papéis ativos autorizados[FER95] .

Regra 9 – Autorização de Acesso a Objeto (*Object Access Authorization*): Um sujeito pode acessar um objeto somente se o papel é parte do conjunto de papéis do sujeito ativos atualmente, bem como se ao papel é permitido executar a operação, e também se a operação de acesso ao objeto é autorizada.

O esquema geral do modelo RBAC, mostrando a relação entre usuários, papéis e permissões, a hierarquia de papéis e as restrições pode ser visto na Figura 5.

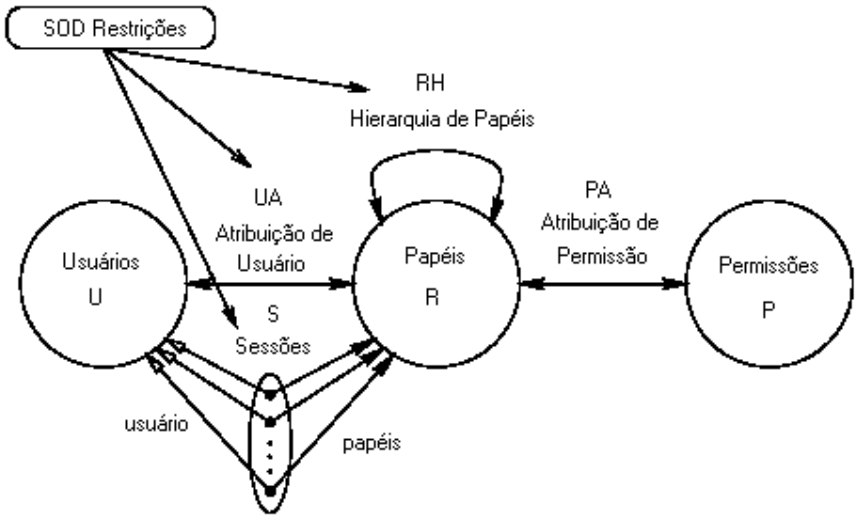


Figura 5- Modelo Geral RBAC ([OBE02]).

2.8 Família de Modelos RBAC

O RBAC é um conceito bastante amplo e aberto, que abrange um conjunto de complexidade que vai desde o muito simples até o extremamente sofisticado. Existe um consenso de que um único modelo definitivo para o RBAC estaria fora da realidade, uma vez que seria demasiadamente restritivo ou excessivamente complexo, representando apenas um dos pontos possíveis dentro do conjunto [OBE01] . Uma abordagem mais realista passa pela definição de uma **família de modelos**, que parte de um componente básico que contempla as características fundamentais do RBAC e passa por componente adicionais que acrescentam funcionalidade e requisitos ao modelo básico. A família de modelos RBAC96 [SAN96] é, possivelmente, o exemplo mais

conhecido desta linha; o modelo RBAC-NIST também compartilha desta abordagem [OBE01] .

A família RBAC-NIST define quatro modelos:

- RBAC Básico (*Flat RBAC*) ou $RBAC_0$;
- RBAC Hierárquico (*Hierarchical RBAC*) ou $RBAC_1$;
- RBAC com Restrições (*Constrained RBAC*) ou $RBAC_2$;
- RBAC Simétrico (*Symmetric RBAC*) ou $RBAC_3$.

2.8.1 RBAC Básico

O RBAC Básico inclui os aspectos essenciais do modelo RBAC. O conceito básico de RBAC é que usuários e permissões são associados a papéis e que os usuários adquirem permissões sendo membros de papéis. O modelo RBAC NIST exige que a associação de usuários-papéis e permissões-papéis seja de muitos-para-muitos. Deste modo, o mesmo usuário pode estar associado a muitos papéis e um único papel pode possuir muitos usuários. A mesma coisa acontece com relação às permissões [SAN00] .

O conceito de sessão, por sua vez, não faz parte do RBAC Básico. Uma sessão corresponde a um usuário acessando o sistema utilizando um conjunto de papéis [SAN98]. A semântica exata de uma sessão é dependente de implementação. Em alguns sistemas, todos os papéis de um usuário são ativados em uma sessão; em outros, o usuário ativa e desativa os papéis que deseja utilizar. Esta última abordagem permite que o usuário exerça o princípio do mínimo privilégio, ativando apenas os papéis necessários à execução de uma determinada tarefa. Quando todos os papéis são ativados, por outro lado, o princípio do mínimo privilégio dificilmente pode ser respeitado [OBE01] .

Uma outra exigência do RBAC Básico é o suporte à revisão usuário-papel, o que possibilita determinar quais os usuários associados a um papel e quais os papéis associados a um usuário (uma exigência semelhante é imposta para a revisão permissão-papel no RBAC Simétrico) [SAN00] .

2.8.2 RBAC Hierárquico

O RBAC Hierárquico exige que exista o suporte a hierarquia de papéis (definida na seção 2.3). Estas hierarquias são representadas matematicamente por relações de ordem-parcial [SAN96] , ou grafos acíclicos dirigidos [FER99]. O RBAC Hierárquico pode ser subdivido em:

- **RBAC Hierárquico Geral:** uma hierarquia de papéis pode constituir qualquer tipo de ordem parcial;
- **RBAC Hierárquico Limitado:** quando existe qualquer restrição em relação à estrutura da hierarquia de papéis. Geralmente isto significa que hierarquias são limitadas a estruturas simples como árvores ou árvores invertidas [OBE01] .

O RBAC Hierárquico exige que o mecanismo de revisão usuário-papel do RBAC Básico seja estendido para suportar hierarquia de papéis. Neste modelo, o mecanismo de revisão deve permitir a identificação tanto dos papéis associados diretamente a um usuário (definidos pelo administrador de segurança) como dos papéis associados indiretamente ao usuário (cuja associação se dá através de herança) [OBE01] .

2.8.3 RBAC com Restrições

O RBAC com Restrições tem como característica dar suporte ao princípio da separação de tarefas (Regras 2 e 6).

É importante salientar que, para que seja possível respeitar a separação de tarefas, é necessário atender o princípio do mínimo privilégio na definição dos papéis, ou seja, para que a separação de tarefas seja atingida através do RBAC, os papéis têm que estar associados ao mínimo de permissões necessárias ao cumprimento de suas tarefas.

2.8.4 RBAC Simétrico

O RBAC simétrico tem como requisito único o suporte a revisão de associações permissão-papel, possibilitando identificar as permissões associadas a um papel e também os papéis que possuem determinadas permissões [SAN00] .

2.9 Conclusões do Capítulo

Este capítulo apresentou uma visão geral das classes das políticas de controle de acesso, atendo-se mais nas políticas de controle de acesso baseadas em papéis, pois esta foi a classe de política de controle de acesso utilizada neste trabalho. Após, foram apresentadas as características do modelo de segurança RBAC, bem como as regras definidas neste modelo de forma mais detalhada. Por fim foi apresentada a família de modelos RBAC, mostrando algumas características de cada modelo pertencente à família de modelos do NIST.

Capítulo 3

Frameworks Orientados a Objeto

A diferença entre a demanda e a produção de *software* está aumentando continuamente e uma maneira de aumentar a produção de *software* é introduzir o conceito de reusabilidade de *software* [LAN95] .

Segundo [LAN95] o conceito de reusabilidade é não desenvolver nada que já existe, mas sim reutilizá-lo. Isto conduzirá à diminuição do tempo de desenvolvimento e do tempo-de-mercado.

O projeto de software para reusabilidade tem como objetivo produzir componentes de software genéricos e extensíveis. Os analistas precisam prever possíveis aplicações futuras e incorporar as exigências destas aplicações no projeto atual. Para executar esta tarefa, o enorme número de decisões de projeto que os analistas têm que tomar devem ser limitadas. Tradicionalmente isto tem sido feito provendo bibliotecas de funções específicas de domínio ou bibliotecas de classes reutilizáveis [LAN95] .

De acordo com [LAN95] a reusabilidade de componentes de *software* é reconhecida como um importante meio de aumentar a produtividade no desenvolvimento de *software*. Os programadores experientes sempre reutilizam código utilizando sua experiência ou buscando projetos de códigos antigos, mas a reutilização das partes de análise e projeto possui um potencial significativamente grande.

O conceito de *frameworks* torna possível a reutilização não somente de código, mas também das partes de análise e projeto [LAN95] . Para entender esta afirmação é necessário conhecer o conceito de *framework.*

De acordo com [SIL00], a abordagem de *frameworks* orientados a objeto utiliza o paradigma de orientação a objetos para produzir uma descrição de um domínio para ser reutilizada. Essa abordagem utiliza principalmente os conceitos de abstração de dados, polimorfismo e herança [JOH97] [FAY99] .

Existem várias definições para *frameworks*. O conceito que foi escolhido para este trabalho foi o de [SIL00], que define um *framework* como “uma estrutura de classes interrelacionadas, que corresponde a uma implementação incompleta para um conjunto

de aplicações de um domínio. Esta estrutura de classes deve ser adaptada para a geração de aplicações específicas."

Outra definição semelhante a de [SIL00] é a de [FAY99] : "um *framework* é uma aplicação incompleta e reutilizável, que pode ser especializada com o objetivo de produzir aplicações específicas."

Já [LAN95] define *framework* como um conjunto de classes cooperantes, abstratas ou concretas, que permitem a reusabilidade de projeto para uma classe específica de *software*.

A diferença fundamental entre um *framework* e a reutilização de classes de uma biblioteca, é que neste caso são usados artefatos de *software* isolados, cabendo ao desenvolvedor estabelecer sua interligação, e no caso do *framework*, é procedida a reutilização de um conjunto de classes inter-relacionadas – inter-relacionamento estabelecido no projeto do *framework* [SIL00].

De acordo com [SIL00] e [TAL95] dois aspectos caracterizam um *framework*:

- Os *frameworks* fornecem infra-estrutura e projeto: *frameworks* possuem infra-estrutura de projeto, que é disponibilizada ao desenvolvedor da aplicação e reduz a quantidade de código a ser desenvolvida, testada e depurada. As interconexões pré-estabelecidas definem a arquitetura da aplicação, liberando o desenvolvedor desta responsabilidade. O código escrito pelo desenvolvedor visa estender ou particularizar o comportamento do *framework*, de forma a moldá-lo a uma necessidade específica;
- Os *frameworks* "chamam", não são "chamados": um papel do *framework* é fornecer o fluxo de controle da aplicação. Assim, em tempo de execução, as instâncias das classes desenvolvidas esperam ser chamadas pelas instâncias das classes do *framework*.

Um *framework* se destina a gerar diferentes aplicações para um domínio. Precisa, portanto, conter uma descrição dos conceitos deste domínio. As classes abstratas de um *framework* são os repositórios dos conceitos gerais do domínio de aplicação. No contexto de um *framework*, um método de uma classe abstrata pode ser deixado propositalmente incompleto para que sua definição seja acabada na geração de uma aplicação. Apenas atributos a serem utilizados por todas as aplicações de um domínio são incluídos em classes abstratas [SIL00].

Os *frameworks* são estruturas de classes interrelacionadas, que permitem não apenas a utilização de classes, mas minimizam o esforço para o desenvolvimento de aplicações, por conterem o protocolo de controle da aplicação (a definição da arquitetura), liberando o desenvolvedor de *software* desta preocupação. Os *frameworks* invertem a ótica de reuso de classes, da abordagem *bottom-up*[2] para a abordagem *top-down*: o desenvolvimento inicia com o entendimento do sistema contido no projeto do *framework*, e segue no detalhamento das particularidades da aplicação específica, o que é definido pelo usuário do *framework*. Assim, a implementação de uma aplicação a partir do *framework* é feita pela adaptação de sua estrutura de classes, fazendo com que esta inclua as particularidades da aplicação [SIL00][TAL95] .

Uma outra característica é o nível de granularidade de um *framework*. Um *frameworks* pode agrupar diferentes quantidades de classes, sendo assim, mais ou menos complexos. Além disso, podem conter o projeto genérico completo para um domínio de aplicação, ou construções de alto nível que solucionam situações comuns em projetos. Deutsch [DEU89] admite *framework* de uma única classe (com esta nomenclatura) que consiste de "uma classe que fornece especificação e implementação parcial, mas que necessita de subclasses ou parâmetros para completar a implementação". *Frameworks* com mais de uma classe são denominados *frameworks* de múltiplas classes [SIL00][DEU89].

A utilização de *frameworks* vem crescendo continuamente. Alguns exemplos de *frameworks* podem ser citados, como OLE (*Object Linking and Embedding*), DSOM (*Distributed System Object Model*), Java AWT (*Abstract Window Toolkit*) que é parte de JFC (*Java Foundation Classes*), Java RMI (*Remote Method Invocation*), MFC (*Microsoft Foundation Classes*), Microsoft DCOM (*Distributed Common Object Model*) e CORBA (Visibroker, Orbix).

3.1 Classificação de *Frameworks*

De acordo com [SIL00], os *frameworks* podem ser classificados de maneira geral

[2] Desenvolvimento *bottom-up* é o que ocorre, por exemplo, quando classes de objetos de uma biblioteca são interligadas, para a construção de uma aplicação.

como dirigido a arquitetura ou dirigido a dados. No primeiro caso a aplicação deve ser gerada a partir da criação de subclasses das classes do *framework*. No segundo caso, diferentes aplicações são produzidas a partir de diferentes combinações de objetos, instâncias das classes presentes no *framework*. Os *frameworks* dirigidos à arquitetura são mais difíceis de usar, pois a geração de subclasses exige um profundo conhecimento do projeto de um *framework*, bem como em certo esforço no desenvolvimento de código. Os *frameworks* dirigidos a dados são mais fáceis de usar, porém são menos flexíveis [SIL00]. Uma outra abordagem possível seria uma combinação dos dois casos, onde o *framework* apresentaria uma base dirigida à arquitetura e uma camada dirigida a dados. Com isto, possibilita a geração de aplicações a partir da combinação de objetos, mas permite a geração de subclasses [TAL94] . Alguns autores, como Johnson [JOH97] , denominam os *frameworks dirigidos* a arquitetura como caixa-branca (*white-box frameworks*) e os *frameworks* dirigidos a dados como "caixa-preta" (*black-box frameworks*). Johnson utiliza a expressão "caixa-cinza" (*gray-box*) para a combinação dos dois casos.

De acordo com [FAY99] os *frameworks* caixa-cinza são projetados para evitar as desvantagens apresentadas pelos *frameworks* caixa-branca e os caixa-preta. Em outras palavras, um *framework* caixa-cinza bem projetado possui bastante flexibilidade e estensibilidade, e também possui a capacidade de esconder informação desnecessária dos desenvolvedores de aplicação.

De acordo com a Figura 6, adaptada de [BUT00] , o percentual de reutilização e a facilidade de uso aumentam quando são utilizados *frameworks* do tipo caixa-cinza.

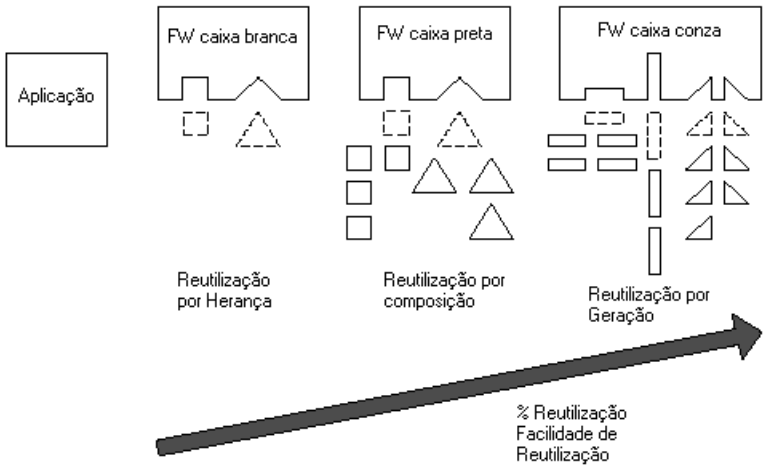


Figura 6 - Maturidade de um Framework.

De acordo com [FAY99] , os *frameworks* também podem ser classificados, de acordo com o seu escopo. Este tipo de classificação divide os *frameworks* em três tipos:

- ***Frameworks* de Infra-Estrutura de Sistema (*System Infrastructure Frameworks*):** São *frameworks* que simplificam o desenvolvimento de sistemas de infra-estrutura eficientes e portáveis, como por exemplo, sistemas operacionais. Outros exemplos são os *frameworks* de comunicação, *frameworks* para interface com o usuário e ferramentas de processamento de linguagem. Os *frameworks* de infra-estrutura de sistema são principalmente utilizados internamente dentro da organização do software e não são vendidos a clientes diretamente.
- ***Frameworks* de Integração *Middleware* (*Middleware Integration Frameworks*):** Estes *frameworks* são geralmente utilizados para integrar aplicações e componentes distribuídos. Os *frameworks* de integração *middleware* são projetados para melhorar a capacidade dos desenvolvedores de *software* de modularizar, reutilizar e estender sua infra-estrutura de *software* para trabalhar em um ambiente distribuído sem muitos problemas. Alguns exemplos destes tipos de *frameworks* incluem os

frameworks de ORBs, de *middlewares* orientados a mensagem e as bases de dados transacionais.

- ***Frameworks* de Aplicação de Empresa (*Enterprise Application Frameworks*):** Estes *frameworks* se tratam de amplos domínios de aplicação, como por exemplo, telecomunicações, indústria, engenharia financeira e são base para as atividades de negócio de uma empresa.

Em comparação com os *frameworks* de infra-estrutura de sistema e com os *frameworks* de integração *middleware*, os *frameworks* de aplicação de empresa possuem um custo de desenvolvimento mais alto. Porém estes últimos podem fornecer um significativo retorno de investimento desde que suportem o desenvolvimento de aplicações *end-user* e de produtos de forma direta. Em contrapartida, os *frameworks* de infra-estrutura de sistemas e de integração *middleware* focalizam em grande parte nos interesses internos ao desenvolvimento de *software*. Embora esses *frameworks* sejam essenciais para a criação de *softwares* de alta qualidade rapidamente, eles geralmente não geram uma renda significativa para grandes empresas [FAY99] .

3.2 Ciclo de Vida de *Frameworks*

De acordo com [SIL00], o ciclo de vida de um *framework* difere do ciclo de vida de uma aplicação convencional porque um *framework* nunca é um artefato de *software* isolado, mas sua existência está sempre relacionada à existência de outros artefatos, originadores do *framework*, originados a partir dele ou que exercem alguma influência na definição de estrutura de classes do *framework*. A Figura 7, obtida de [SIL00], ilustra as várias fontes de informação que influem na definição da estrutura de um *framework*: artefatos de *softwares* existentes, os quais são produzidos a partir de um *framework* e o conhecimento do desenvolvedor do *framework* (ou da equipe de desenvolvimento). As setas representam o fluxo de informações que levam à produção da estrutura de classes do *framework*, bem como de aplicações sob o *framework.*

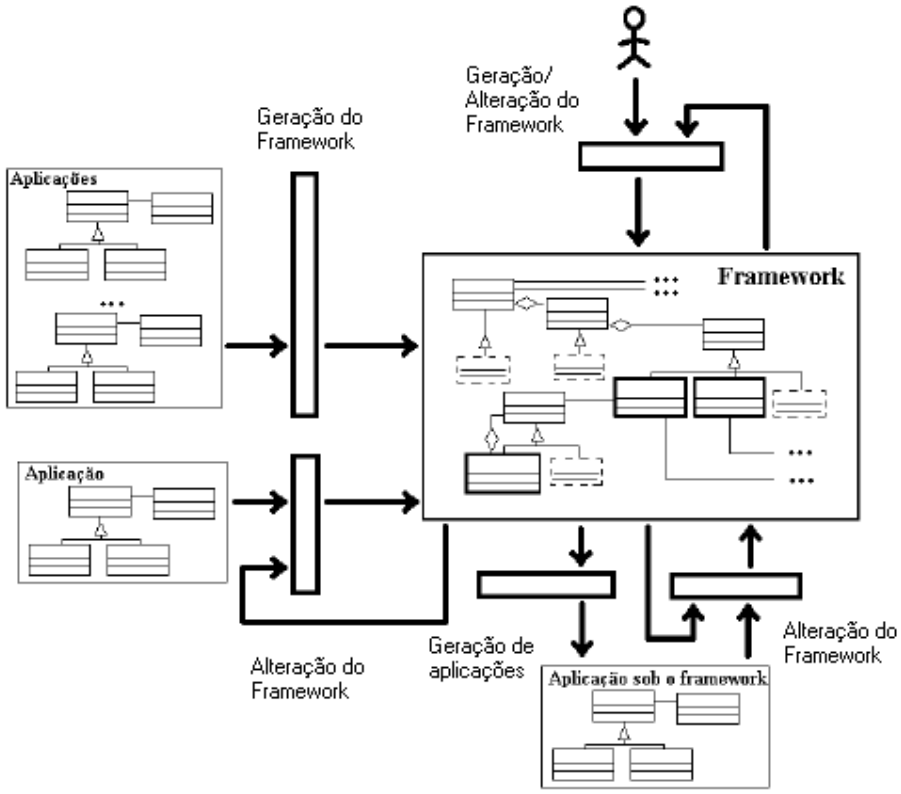


Figura 7 - Ciclo de Vida de Frameworks

Fonte:[SIL00].

Abaixo são descritos as fontes de informação que influenciam na definição da estrutura de um *framework*:

A atuação do desenvolvedor: Nenhuma abordagem de desenvolvimento de *frameworks* existente dispensa a figura do desenvolvedor. Ele é o responsável por decidir que classes comporão a estrutura do *framework*, suas responsabilidades e a flexibilidade provida aos usuários do *framework*. O desenvolvedor não atua apenas na construção do *framework*, mas também na sua manutenção [SIL00].

Aplicações do domínio tratado: Um *framework* constitui um modelo de um domínio de aplicações. Assim, pode ser desenvolvido a partir de um conjunto de aplicações do domínio, que atuam como fontes de informação deste domínio. Esta influência de aplicações do domínio pode ocorrer no processo do desenvolvimento do *framework*, em que um *framework* é constituído como uma generalização de diferentes estruturas, ou na fase de manutenção. Neste caso a alteração seria motivada pela obtenção de conhecimento do domínio tratado, não considerado (ou indisponível)

durante o desenvolvimento do *framework*.

Aplicações geradas sob o *framework*: A finalidade básica de um *framework* é ser reutilizado na produção de diferentes aplicações, minimizando o tempo e esforço requeridos para isto. A construção de um *framework* é sempre precedida por um procedimento de análise de domínio em que são observadas informações do domínio tratado. Como uma abstração de uma realidade tratada, é inevitável que o *framework* seja incapaz de conter todas as informações do domínio. De fato, um *framework* consegue ser uma descrição aproximada do domínio, construída a partir das informações até então disponíveis.

Idealmente a construção de aplicações sob *frameworks* consiste em completar ou alterar procedimentos e estruturas de dados presentes no *framework*. Sob esta ótica, uma aplicação gerada sob um *framework* não deveria incluir classes que não fossem subclasses das classes do *framework*. Porém, como um *framework* nunca é uma descrição completa de um domínio, é possível que a construção de aplicações sob um *framework* leve à obtenção de novos conhecimentos do domínio tratado, indisponíveis durante a construção do *framework*. Estas novas informações podem levar à necessidade de alterar o *framework* [SIL00].

3.3 Desenvolvimento de *Frameworks*

O desenvolvimento de um *framework* é um pouco diferente do desenvolvimento de uma aplicação padrão. A grande diferença está no fato de que o *framework* tem que cobrir todos os conceitos relevantes no domínio e a aplicação só abrange aqueles conceitos mencionados em seus requisitos [BOS97] De acordo com [SIL00], pode-se afirmar que o desenvolvimento de um *framework* é mais complexo que o desenvolvimento de aplicações específicas do mesmo domínio, devido:

- À necessidade de considerar os requisitos de um conjunto significativo de aplicações, de modo a dotar a estrutura de classes do *framework* de generalidade, em relação ao domínio tratado;
- À necessidade de ciclos de evolução para prover a estrutura de classes do *framework* de alterabilidade e estensibilidade.

Para estabelecer o contexto de problemas experimentados e identificados no desenvolvimento de *frameworks* [BOS97] define-se as seguintes atividades como parte de um simples modelo de desenvolvimento de *frameworks*:

- **Análise de Domínio (*Domain Analysis*):** Esta etapa visa descrever o domínio que será abrangido pelo *framework*. Uma forma de capturar os requisitos e identificar conceitos relacionados com o domínio é procurar aplicações desenvolvidas anteriormente, obter informações com especialistas no domínio e buscar os padrões existentes. O resultado desta atividade é um *modelo de análise de domínio*, contendo os requisitos e os conceitos do domínio e suas relações;
- **Projeto da Arquitetura (*Architectural Design*):** Esta atividade utiliza o modelo de análise de domínio como ponto de partida. O analista tem que escolher um *estilo de arquitetura*[3] satisfatório ao desenvolvimento do *framework*. A partir deste ponto é que o projeto mais alto nível do *framework* é elaborado;
- **Projeto do *Framework* (*Framework Design)*:** Durante esta fase o projeto de nível mais alto do *framework* é refinado e classes adicionais são projetadas. Os resultados desta atividade são a definição do escopo de funcionalidades dada pelo projeto do *framework*, a *interface* de reutilização do *framework*, as regras de projeto que devem ser obedecidas e que são baseadas nas decisões de arquitetura, e por fim um documento com o histórico do projeto, descrevendo os problemas encontrados e as soluções escolhidas juntamente com as argumentação de suas escolhas;
- **Implementação do *Framework (Framework Implementation)*:** É a etapa de codificação das classes abstratas e concretas que foram definidas para o *framework*;
- **Testes do *Framework (Framework Testing*):** É a fase que se determina se o *framework* fornece realmente as funcionalidades que foram planejadas, e também avaliar a usabilidade deste. Porém, não é trivial decidir se uma entidade é utilizável ou não. Johnson e Russo [JOH91] concluem que o único modo de descobrir se algo é reutilizável ou não é realmente reutilizando. No caso de *frameworks*, isto se resume a

desenvolver aplicações que utilizem o *framework*;

- **Geração de Aplicação de Teste (*Test Application Generation*):** Para avaliar a usabilidade de um *framework*, a atividade de geração de uma aplicação de teste é interessante com o desenvolvimento de aplicações de teste baseadas no *framework*. Dependendo do tipo de aplicação, cada aplicação pode testar diferentes aspectos do *framework*. A criação de aplicações de teste visa verificar se o *framework* necessita ser reprojetado ou está suficientemente bom para ser utilizado.

A documentação é uma das mais importantes atividades no desenvolvimento de *frameworks*, apesar de que sua importância nem sempre ser reconhecida. Sem uma documentação clara, completa e correta que descreve como utilizar o *framework*, um manual do usuário, e um documento de projeto descrevendo como o *framework* funciona, será quase que impossível a utilização do *framework* por engenheiros de *software* que não se envolveram no projeto deste.

3.3.1 Metodologias de Desenvolvimento de *Frameworks*

Em [SIL00] são descritas três propostas de desenvolvimento de *frameworks*: Projeto Dirigido por Exemplo (*Example-Driven Design*), Projeto Dirigido por *Hot Spot* (*Hot Spot Driven Design*) e a metodologia de projeto da empresa Taligent ([TAL95]). Estas metodologias se caracterizam por estabelecer o processo de desenvolvimento de *frameworks* em linhas gerais, sem se ater à definição de técnicas de modelagem ou detalhar o processo.

No Projeto Dirigido por Exemplo é estabelecido que o desenvolvimento de um *framework* para um domínio de aplicação é decorrente de um processo de aprendizado a respeito deste domínio, que se processa concretamente a partir do desenvolvimento de aplicações ou do estudo de aplicações desenvolvidas. Como as pessoas pensam de forma concreta e não abstrata, a abstração do domínio, que é o próprio *framework*, é

---

[3] Estilo de arquitetura (*architectural style*) caracteriza uma família de sistemas que estão relacionados por compartilharem propriedades estruturais ou semânticas[MON97]. Ou seja, definir um estilo de arquitetura é determinar se o sistema será, por exemplo, cliente-servidor, ou baseado em camadas, ou utilizará padrões como CORBA, etc. Podem existir estilos de arquitetura híbridos, ou seja, que mesclam características de outros estilos de arquitetura.

obtida através da generalização de casos concretos, as aplicações [SIL00].

Uma aplicação orientada a objetos é completamente definida. Um *framework*, ao contrário, possui partes propositalmente indefinidas, o que lhe dá a capacidade de ser flexível e se moldar a diferentes aplicações. Os *hot spots* são as partes do *framework* mantidas flexíveis. A essência da metodologia de Projeto Dirigido por *Hot-Spot* é identificar os *hot spots* na estrutura de classes de um domínio, e então o *framework* é construído [SIL00].

De acordo com [SIL00] a metodologia proposta pela empresa Taligent (empresa já extinta) difere das anteriores pelo conjunto de princípios que norteia o desenvolvimento de *frameworks*. Primeiramente, a visão de desenvolver um *framework* que englobe as características e as necessidades de um domínio é substituída pela visão de produzir um conjunto de *frameworks* estruturalmente menores e mais simples, que usados em conjunto, originam as aplicações. A justificativa para isto é que "pequenos *frameworks* são mais flexíveis e podem ser reutilizados mais freqüentemente". Assim, a ênfase passa a ser o desenvolvimento de *frameworks* pequenos e direcionados a aspectos específicos do domínio.

3.3.2 Utilizando Padrões no Desenvolvimento de *Frameworks*

Os p*atterns* constituem uma abordagem recente em termos de reutilização de projeto, no contexto de desenvolvimento de *software* orientado a objetos. A principal questão tratada é como proceder para reutilizar em um desenvolvimento de *software*, a experiência de projeto adquirida em desenvolvimentos anteriores. Por este motivo os padrões de projeto atualmente são formas populares e eficientes de implementação de *softwares* flexíveis e reutilizáveis [MEN01] .

Um único *framework* geralmente contém muitos *patterns*, sendo que estes são geralmente mais abstratos que o *framework* [LEE99].

Segundo o arquiteto Christopher Alexander [ALE77] "Cada p*attern* descreve um problema o qual ocorre repetidamente em nosso ambiente, e então descreve um conjunto de soluções para este problema, de maneira que você possa usar esta solução um milhão de vezes, sem o fazer da mesma maneira duas vezes." Alexander desenvolveu uma linguagem padrão ("*pattern language*") para permitir às pessoas projetarem suas próprias casas e comunidades (construção civil). A linguagem de

Alexander permite iniciar uma descrição em uma escala de abstração elevada e utilizar o mesmo padrão de especificação (formato) para refinar a descrição. Apesar do uso da expressão “linguagem” o padrão de descrição de Alexander se baseia em texto estruturado, e não em modelos formais [ALE77] [SIL00].

A idéia é capturar experiências comprovadamente corretas em desenvolvimento de software e ajudar a promover a prática de projeto correto. Cada *pattern* trabalha com um problema específico e recorrente no projeto ou implementação de softwares.

Segundo a GoF [BUS96] , um *pattern*, em geral, é formado de quatro elementos essenciais, a saber:

- **O nome do *pattern***, uma ou duas palavras utilizadas para descrever o problema, sua solução e conseqüências. Encontrar um bom nome é, normalmente, uma das tarefas mais difíceis e importantes da documentação de um *pattern*. Ao se designar um bom nome a um *pattern*, automaticamente aumentamos nosso vocabulário e ele passará a ser adotado pelos nossos colegas de desenvolvimento; assim, quando quisermos expressar uma solução, não há a necessidade de se explicar todo o processo de solução, mas sim apenas citar qual o *pattern* a ser utilizado;
- **O problema** descreve quando um *pattern* deve ser aplicado, explicando o problema em si, e o contexto onde é encontrado. Em alguns casos o problema pode conter uma lista de condições, as quais devem ser satisfeitas antes que a utilização do *pattern* faça sentido;
- **A solução** descreve os elementos que fazem parte do projeto, seus relacionamentos, responsabilidades e colaborações; no entanto, a solução não descreve um projeto completo ou implementação, já que o *pattern* pode ser utilizado diversas vezes sem que se repita uma mesma implementação. O que deve ser indicado é a idéia da solução do problema;
- **As conseqüências** são os resultados alcançados com a aplicação do *pattern*. Através delas podem ser verificadas as possibilidades da utilização do *pattern* para a solução no problema proposto no contexto especificado. Elas podem, inclusive, especificar a

> linguagem de programação utilizada, bem como peculiaridades ligadas diretamente à implementação. Uma vez que estas conseqüências podem afetar, também, o reuso, devem ser especificados os impactos ocasionados à flexibilidade, à extensibilidade e à portabilidade do sistema.

O ponto de vista de cada indivíduo afeta o que vem a ser ou não um *pattern*. Não existe uma definição clara em que ponto da abstração deve-se chegar para definição de um *pattern*. A GoF [BUS96] diz que "Os p*atterns* não são para projetos como listas encadeadas e tabelas h*ash*, as quais podem ser codificadas como classes e então apenas reusadas para solução de suas necessidades. Nem são complexos a ponto de chegar a projetos para domínios específicos para uma aplicação inteira ou um subsistema. Os p*atterns* são a descrição de classes e objetos que se comunicam e que são customizados para resolver um problema comum de projeto em um contexto específico".

Infelizmente os *patterns* têm que ser manualmente codificados nas aplicações, pois os ambientes de desenvolvimento de *software* atuais não possuem um suporte satisfatório ao desenvolvimento dos padrões de projeto [MEN01] .

A implementação de um padrão de projeto é dependente da linguagem, embora a maioria das linguagem orientadas a objetos possuam características comuns (herança, polimorfismo, etc) existem diferenças sutis (como por exemplo, se a gerência de memória é automática ou manual, se a linguagem possui tipos de variáveis ou não) que poderão afetar a forma como o padrão deve ser implementado [BUN01].

Dada a diferença entre as linguagens, não é possível criar uma especificação concreta de um padrão de projeto para todas as linguagens. Então existem duas formas de especificação de padrões de projeto: a primeira é a criação de uma especificação abstrata, sem preocupação com detalhes de implementação, e a segunda é a especificação concreta de um padrão de projeto para uma linguagem alvo. A primeira forma utiliza uma linguagem de especificação gráfica, como por exemplo, LePUS [EDE98], que apresenta os padrões de projeto em alto nível. A segunda forma necessita de conhecimentos específicos da linguagem na qual o padrão deverá ser implementado [BUN01].

3.3.2.1 Catálogo de Padrões de Projeto

A partir da experiência adquirida na identificação de padrões de projeto (*design patterns*) sobre *frameworks*, Gamma, Helm, Johnson e Vlissides produziram um catálogo com vinte e três padrões de projeto de uso geral [GAM95] . Cada padrão de projeto representa uma solução para um problema freqüente de projeto, que visa auxiliar na escolha de alternativas de projeto que produzam *software* reutilizável, evitando alternativas que comprometam a reusabilidade [SIL00].

Os padrões de projeto foram originados a partir da observação de que diferentes partes dos *frameworks* possuíam uma estrutura de classes semelhante. Isto demonstrou a existência de padrões de solução para problemas de projeto semelhantes, e que se repetiam à medida que se procurava produzir uma estrutura flexível (para extensão ou adaptação). O catálogo de padrões de projeto pode ser vista em [GAM95] , mas alguns exemplos são os padrões *Factory Method*, *Template Method* e *Proxy*.

Para criação de novos padrões de projeto é necessário entender bastante do assunto, os desenvolvedores geralmente não possuem muita experiência no assunto. Por este motivo é desejável verificar se um padrão de projeto foi corretamente implementado [BUN01].

3.3.2.2 Classificação dos Padrões de Projeto

Os padrões de projeto podem ser classificados em estáticos e extensíveis. Os padrões de projeto estáticos são aqueles que não permitem extensão, ou que são difíceis de estender. Já os padrões de projeto extensíveis são projetados para permitir que as funcionalidades de uma aplicação possam ser alteradas até mesmo após a compilação que utiliza o padrão de projeto [TSA99] [EID02] . Em outras palavras, é possível alterar o comportamento de uma aplicação em tempo de execução apenas adicionando novos objetos e/ou classes. Um exemplo de padrão de projeto extensível é o *Factory Method* [TSA99] .

Os padrões de projeto também podem ser classificados em: criacionais, estruturais e comportamentais. Os padrões de projeto criacionais são utilizados na criação de novas instâncias e podem fazer com que um sistema seja independente de como seus objetos são criados, compostos e representados. Este tipo de padrão de projeto pode-se utilizar

herança para variar as classes que serão instanciadas, bem como se pode delegar instanciação de uma classe a outros objetos [BUN01][GAM95] .

Os padrões de projeto estruturais são aqueles cujas propriedades são baseadas na estrutura de relacionamento entre suas classes. Este tipo de padrão geralmente utiliza a composição de classes para a criação de novas funcionalidades. Um simples exemplo é utilizar herança múltipla para combinar duas classes em apenas uma [BUN01][GAM95] .

Os padrões de projeto comportamentais são baseados na implementação de algoritmos para a execução de soluções comuns, bem como na transferência de responsabilidades entre os objetos. Os padrões comportamentais descrevem não somente os padrões de objetos e classes, mas também os padrões de comunicação entre eles [BUN01][GAM95] .

3.4 Conclusões do Capítulo

Este capítulo apresentou a importância da reusabilidade de *software* e como esta pode ser obtida com a utilização de *frameworks*. Para isto foram apresentados alguns conceitos de *frameworks*, suas características principais, os tipos de classificação e o ciclo de vida. Após foram apresentadas as principais atividades que devem ser realizadas no desenvolvimento de *frameworks*, bem como as metodologias de desenvolvimento existentes. Por último foi apresentado o conceito de padrões de projeto, a classificação destes e também o catálogo de padrões de projeto criado por [GAM95] .

Capítulo 4

Um *Framework* RBAC para Aplicações *Web*

Como já foi dito, o crescimento da utilização da Internet e de Intranet pelas empresas acarretou o crescimento do uso de aplicações Web, causando o aumento da demanda por este tipo de *software*. Além disso, dentro de uma empresa ou corporação existem várias pessoas com responsabilidades ou papéis diferentes, que por sua vez possuem privilégios diferentes. É neste contexto que a seguinte proposta de dissertação de mestrado pretende operar, ou seja, em auxiliar no controle de privilégios dos usuários no desenvolvimento de aplicações Web. Por este motivo que a seguinte proposta se trata de um *framework*, cuja idéia permite a reusabilidade não somente de código, mas também das etapas de análise e projeto que fazem parte da especificação de um sistema. Deste modo, é possível afirmar que a especificação deste *framework* poderá ser utilizada em implementações em outras linguagens de programação e que seja utilizada por qualquer tipo de aplicação, não somente em aplicações com *interface* Web. A escolha do modelo de segurança RBAC se deve ao fato de este ser o que mais se adequa ao modelo de funcionamento das empresas, pois a maioria destas baseiam-se em duas decisões de controle de acesso “nos papéis que os usuários individuais desempenham na organização”. Por este motivo, esse tipo de política vem sendo utilizado cada vez mais em aplicações comerciais e de banco de dados.

4.1 Trabalhos Relacionados

Existem vários trabalhos de implementação do modelo RBAC, tanto comerciais quanto acadêmicos. As principais implementações comerciais são os SGBDs, ou sistemas gerenciadores de bancos de dados. Também existem implementações do modelo RBAC nas áreas de gerenciamento de sistemas e de sistemas operacionais. Mas nenhuma das implementações comerciais segue à risca um conjunto apropriado de características do modelo RBAC [FER99]. No artigo de [KEL00] pode ser vista uma breve introdução de como o modelo RBAC é implementado no sistema operacional

Solaris 8 da Sun Microsystems, bem como uma implementação de RBAC utilizando o protocolo LDAP (*Lightweight Directory Access Protocol*).

Diferentemente das implementações comerciais, os trabalhos acadêmicos seguem com mais fidelidade as características do modelo RBAC. Alguns destes trabalhos serão apresentados nas seções a seguir.

4.1.1 A Proposta RBAC/Web do NIST

Existem vários trabalhos acadêmicos que possuem a mesma idéia de implementação de RBAC para aplicações Web, como a deste trabalho. Um exemplo é a proposta RBAC/Web, do NIST, que é uma implementação de RBAC para ser utilizadas pelos servidores Web [FER99]. A diferença deste trabalho do RBAC/Web, bem como da maioria das propostas para implementação de RBAC para Web, é que este utiliza o RBAC como um esquema de autorização para controlar o acesso a páginas de um servidor Web. Ou seja, as operações permitidas implementadas pelo RBAC/Web são os "métodos" definidos no protocolo HTTP (*HyperText Transfer Protocol*). Estes métodos são GET, HEAD, PUT, POST, etc. O RBAC/Web controla a habilidade de um usuário ativo em um papel de executar um método HTTP em um URL (*Uniform Resource Locator*).

O *framework* Seguraweb neste trabalho não trabalha em nível de operações HTTP, e não foi proposto para ser utilizado pelos servidores Web. A vantagem do *framework* Seguraweb está no controle de operações que podem ser realizadas através de uma página. Ou seja, como os componentes deste *framework* são implementados em Java, eles podem ser tanto utilizados em *applets* como em páginas JSP dinâmicas. Deste modo, as chamadas ao componente de autorização do *framework* podem ser incluídas dentro do código JSP ou do código do *applet*. A chamada ao componente de autorização irá obter o papel do usuário, e deste modo pode-se montar a página dinamicamente ou apresentar no *applet* apenas as operações que são permitidas para o papel deste usuário. Deste modo, o *framework* Seguraweb também pode permitir controle de acesso a páginas dentro de uma aplicação Web implementada em Java. A desvantagem do *framework* Seguraweb, é que ele não pode ser utilizado por quaisquer aplicações de interface Web, mas apenas aquelas implementadas em Java; a não ser que este seja implementado em outra linguagem, como por exemplo, Visual Basic, para ser utilizado

em páginas dinâmicas feitas em ASP (*Active Server Pages)*. Mas mesmo assim, o *framework* não pode ser utilizado em aplicações Web simples, baseadas apenas em páginas estáticas e JavaScript [BEZ99] .

4.1.2 As Propostas RBAC-JACOWEB e Beznosov/Deng

Existem muitos trabalhos que implementam RBAC utilizando o Serviço de Segurança CORBA, como por exemplo, a proposta de Beznosov e Deng [BEZ99] e a proposta de RBAC-JACOWEB [OBE01] [OBE02]. A primeira proposta apresenta como os modelos de controle de acesso baseados em papéis podem ser implementados em sistemas distribuídos orientados a objeto que seguem os padrões OMG/CORBA, bem como descreve o que é necessário para o Serviço de Segurança CORBA para dar suporte aos modelos $RBAC_0$ a $RBAC_3$. A segunda proposta introduz o conceito de ativação automática de papéis pelo subsistema de segurança, que permite que o esquema de autorização seja utilizado por aplicações previamente desenvolvidas e que não possuem nenhuma ciência da segurança do sistema. Este trabalho atua no nível de *middleware*, permitindo sua utilização com qualquer aplicação baseada em CORBA (*Common Object Request Broker Architecture*). A primeira diferença entre o *framework* Seguraweb e estes trabalhos é que o primeiro não é baseado no Serviço de Segurança CORBA. A segunda diferença é que o Seguraweb é um *framework* e as propostas citadas não são. A princípio, o objetivo do *framework* Seguraweb não é utilizar CORBA, o enfoque está no desenvolvimento de um *framework*, que possui interfaces, classes abstratas e concretas que auxiliem no desenvolvimento dos sistemas de segurança das aplicações com interface Web.

4.1.3 A Proposta ORBAC

Na proposta ORBAC, que pode ser encontrada em [ZHA01] e [ZHA01a] é apresentado um modelo RBAC orientado a objetos, com o objetivo de simplificar a administração das políticas de segurança. Este trabalho também propõe uma arquitetura de gerência de segurança descentralizada, baseada em controle de acesso de múltiplos domínios e mostra como o modelo ORBAC funciona nesta arquitetura [ZHA01].

O modelo ORBAC possui algumas semelhanças de modelagem com o *framework* Seguraweb, principalmente no que diz respeito aos elementos do modelo RBAC

(usuário, papel e permissão). Este modelo também implementa as regras de separação estática e dinâmica de tarefas, bem como hierarquia de papéis. O modelo ORBAC é uma alternativa de modelagem do modelo RBAC utilizando o paradigma de orientação a objeto.

A proposta do *framework* Seguraweb além de apresentar uma alternativa semelhante ao modelo ORBAC de como implementar o modelo RBAC utilizando orientação a objetos, também apresenta preocupações em modelar outros requisitos que são necessários na implementação de aplicações que necessitem de segurança, como por exemplo, autenticação, auditoria e persistência dos dados.

4.1.4 Modelo RBAC Baseado em UML

Esta proposta apresenta uma representação genérica do modelo RBAC utilizando UML (*Unified Modeling Language*), que é uma linguagem padrão utilizada pela comunidade de desenvolvedores de aplicações [SHI00]. Este trabalho é útil aos desenvolvedores de aplicações no momento da análise de aplicações de segurança, pois apresenta as visões estática, funcional e dinâmica do modelo RBAC, tanto que foram utilizadas na definição da parte de controle de acesso do *framework* Seguraweb, principalmente a parte de casos de uso (visão funcional) e dos diagramas de colaboração (visão dinâmica); que podem ser encontrados em [SHI00].

Um outro trabalho que utiliza UML é apresentado em [AHN01], que utiliza uma linguagem declarativa, a OCL (*Object Constraints Language*), que faz parte da UML, para expressar formalmente as restrições existentes no modelo RBAC.

4.1.5 Controle de Acesso para Servidores Web de um Mesmo Domínio baseado no modelo RBAC

Este trabalho, encontrado em [SHI01], apresenta um método para fazer controle de acesso de servidores Web distribuídos para controlar a visualização de documentos dependendo do papel do usuário utilizando as informações de um servidor RBAC. Esta proposta utiliza um mecanismo de *cookies* na memória quando o usuário acessa os servidores *Web* em um mesmo domínio. Deste modo, o usuário pode acessar transparentemente todos os recursos em múltiplos servidores *Web*, sem ter que realizar

uma nova autenticação em cada servidor *Web*, dando a impressão que o usuário está acessando apenas um servidor *Web* [SHI01].

Este trabalho se aproxima mais da proposta do *framework* Seguraweb do que a proposta RBAC/Web apresentada anteriormente, pois este também controla a apresentação de informações no momento da criação das páginas dinâmicas. Mas a proposta Seguraweb ainda possui a vantagem da característica de reutilização dos *frameworks*, sendo que a implementação deste trabalho pode ser reutilizada tanto na criação de aplicações *Web* ou em aplicações Java locais. Já a proposta de [SHI01] apresenta uma aplicação de demonstração utilizando PHP versão 4 e o banco de dados MySQL. Uma característica interessante deste trabalho é a definição de *cookies* em memória, que podem ser passados para múltiplos servidores *Web* de forma transparente evitando que o usuário tenha que realizar várias sessões de autenticação. Esta característica pode ser incluída no *framework* Seguraweb, mas foge aos objetivos deste trabalho.

4.1.6 Um *Framework* Baseado na Descrição de Papéis

Este trabalho apresenta uma proposta de *framework* baseado na descrição de papéis e que pode ser utilizado na modelagem de aplicações baseadas em RBAC e de Gerência de *Workflows* [GUS96]. A diferença deste trabalho é que a definição do *framework* é baseada em papéis, ou seja, o elemento principal do *framework* é o elemento papel, enquanto que no *framework* Seguraweb os elementos principal são usuário, papel e permissão. Deste modo, a definição das classes dos elementos do *framework* de [GUS96] foi baseada em um ponto de vista diferente do *framework* Seguraweb. Um exemplo que ilustra bem esta diferença é a modelagem do relacionamento entre usuários e papéis. No *framework* de [GUS96] apenas o elemento RDO (*Role Descriptor Object*) contém a lista de usuários que podem assumir o papel, enquanto no *framework* Seguraweb, isso também é realizado na classe *Role* (seção 4.2.3.1), mas a classe *User* também contém a lista de papéis que o usuário pode assumir. O *framework* Seguraweb possui esta característica para dar suporte à revisão usuário-papel do RBAC Básico, que possibilita determinar quais os usuários estão associados a um papel e também determinar quais papéis estão associados a um usuário.

O artigo [GUS96] também apresenta como podem ser implementadas as regras do modelo RBAC (principalmente no que diz respeito à hierarquia e restrições) utilizando o *framework* proposto por eles. A vantagem do *framework* Seguraweb está no fato de que este está praticamente pronto para o uso, pois além de ter implementado a parte de controle de acesso, também dá suporte à parte de autenticação, administração das informações de segurança, auditoria e persistência. Além disso, a implementação do *framework* Seguraweb utilizou técnicas de padrões de projeto que permitem maior extensibilidade.

4.2 Etapas de Análise e Projeto do *Framework* Seguraweb

De acordo com o que foi descrito na seção 3.1, existem algumas etapas que devem ser seguidas e que fazem parte de um modelo de desenvolvimento de *frameworks* simplificado. A seguir serão descrita cada uma destas etapas no que se refere ao desenvolvimento deste trabalho.

4.2.1 Análise de Domínio

Como descrito na seção 3.1, a análise de domínio visa descrever o domínio que será abrangido pelo *framework*. Para tal é necessário obter os requisitos e identificar conceitos relacionados com o domínio e isto pode ser feito buscando aplicações desenvolvidas anteriormente, além da busca de padrões existentes. Nesta etapa foram utilizadas as aplicações de segurança desenvolvidas na empresa Thermus Com. Serv. Repr. Ltda (especializada no desenvolvimento de softwares para a área de telecomunicações), bem como o estudo do modelo de segurança RBAC, que foi descrito no capítulo 2 deste texto.

Analisando as aplicações foram definidos os seguintes requisitos para a implementação do *framework*:

- Autenticação: para garantir que um usuário é quem ele diz ser;
- Confidencialidade: garantindo proteção de informações em trânsito;

- Controle de Acesso: para controlar o que um usuário pode acessar em um sistema;
- Auditoria: para registrar e analisar o que o usuário faz no sistema;
- Administração das informações de segurança: para gerenciar informações de segurança incluindo as políticas de segurança (gerência de perfis de usuários, etc);
- Persistência dos dados (em arquivos ou base de dados): para armazenamento dos dados de autenticação (senhas,etc), de controle de acesso, de auditoria e de administração.

O modelo de segurança RBAC será utilizado principalmente na implementação dos requisitos de controle de acesso e administração das informações de segurança.

4.2.2 Projeto de Arquitetura

O objetivo deste trabalho é implementar um *framework* para aplicações com *interface* Web. A partir deste objetivo procurou-se obter informações sobre as tecnologias existentes atualmente para desenvolvimento de aplicações Web. As mais utilizadas atualmente são ASP (*Active Server Pages*), PHP (*Personal Home Page* ou um acrônimo recursivo para "PHP: *Hypertext Preprocessor*"), *Servlets*, JSP (*Java Server Pages*) e *Applets*, sendo que estas três últimas são baseadas na utilização da linguagem Java.

A tecnologia ASP possui um ambiente para programação por *scripts* baseados em VisualBasic (VBScript) que funcionam como uma extensão da linguagem HTML. Os *scripts* que são executados no servidor e permitem a criação de páginas dinâmicas. O servidor ASP é quem transforma os *scripts* em páginas HTML padrão, que podem ser acessados de qualquer *browser*. O ASP surgiu juntamente com o lançamento do servidor HTTP *Internet Information Server* 3.0. Por ser uma solução Microsoft, o servidor só pode ser executado nos sistemas operacionais Windows (95, 98, 2000 ou NT).

A tecnologia PHP tem a mesma ideologia do ASP, ou seja, também é um ambiente para programação por *scripts* que são executados no servidor para a geração

de páginas dinâmicas. A sintaxe do PHP tem um formato semelhante às linguagens C, Java e Perl, sendo que possui algumas características específicas adicionadas.

As outras tecnologias (*Servlets*, JSP e *Applets*), por serem baseadas na linguagem Java possuem a vantagem de serem independente de plataforma, ou seja, podem ser executadas tanto em plataforma Unix, Linux ou Windows. Este é o principal motivo pela escolha de Java como a linguagem de implementação do *framework*, bem como também por Java ser uma linguagem orientada a objetos. Deste modo o *framework* se torna mais abrangente, pois pode ser utilizado por aplicações baseadas em *Servlets*, JSP ou em *Applets*. Estas características fazem com que Java seja a linguagem mais escolhida para implementação de aplicações na Internet [PAP00]. Além disso, a linguagem Java possui um modelo de segurança, dispondo de algumas APIs que implementam funções de segurança, como por exemplo, JCE (*Java Cryptography Extension*), JSSE (*Java Secure Socket Extension*) e JAAS (*Java Authentication and Authorization Service*). Outra vantagem da implementação em Java é a facilidade de portar a aplicação para utilização de CORBA, já que existem vários ORBs implementados em Java, bem como já existe na versão do JDK 1.4 uma API com suporte à CORBA.

Por ser um *framework* destinado a aplicações Web, o elemento de segurança de confidencialidade não será levado em conta, já que este elemento é garantido pelo protocolo SSL (*Secure Socket Layer*), já implementado na maioria dos servidores Web utilizados atualmente.

O banco de dados escolhido para a persistência das informações de segurança foi o Oracle 8i. Esta ferramenta foi escolhida por ser uma das mais utilizadas no desenvolvimento de aplicações de grande porte, como as que são implementadas na empresa Thermus, e por ter versões para várias plataformas, principalmente Windows, Unix e Linux.

4.2.3 Projeto e Implementação do *Framework*

Os serviços de segurança dentro do contexto deste trabalho são autenticação, controle de acesso, auditoria, administração das informações de segurança e garantia de comunicação segura.

O *framework* está dividido em três camadas, como pode ser visto na Figura 8: básica, de persistência e de aplicação. A primeira camada possui classes básicas, que são utilizadas pelas outras camadas e podem ser utilizadas pelo usuário do *framework*. A camada de persistência possui classes que tem o objetivo administrar a materialização e desmaterialização dos objetos que necessitam de persistência em mecanismos de armazenamento. A camada de aplicação contém as classes que deverão ser utilizadas pelo usuário do *framework* na construção de aplicações Web. Esta camada utiliza as classes básicas e de persistência para implementar as características e regras do modelo RBAC.

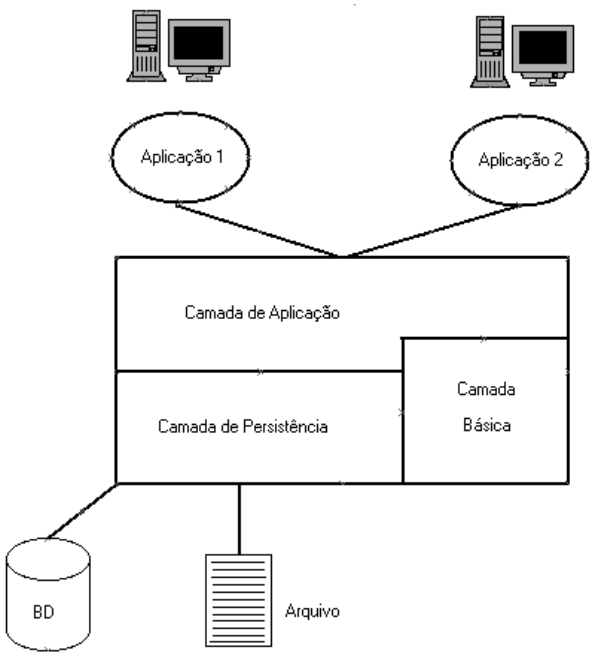


Figura 8 - Esquema Geral do Framework Seguraweb

4.2.3.1 Camada Básica

Como a proposta deste trabalho é desenvolver um *framework* baseado no modelo RBAC de segurança, existem algumas classes que espelham as entidades principais deste modelo e que são utilizadas em vários pontos do *framework*. Essas classes são: *User*, *Role* e *Permission*.

A classe *User*, que representa o elemento usuário do modelo RBAC, possui quatro atributos, como pode ser visto na Figura 9:

- _id: identificador do usuário, que é uma *string* única por usuário;
- _authenticationMethod: é o método de autenticação que será utilizado para validar o usuário;
- _fullName: nome completo do usuário;
- _roles: lista de papéis que estão relacionados com o usuário, ou seja, a lista de papéis ao qual o usuário pertence.

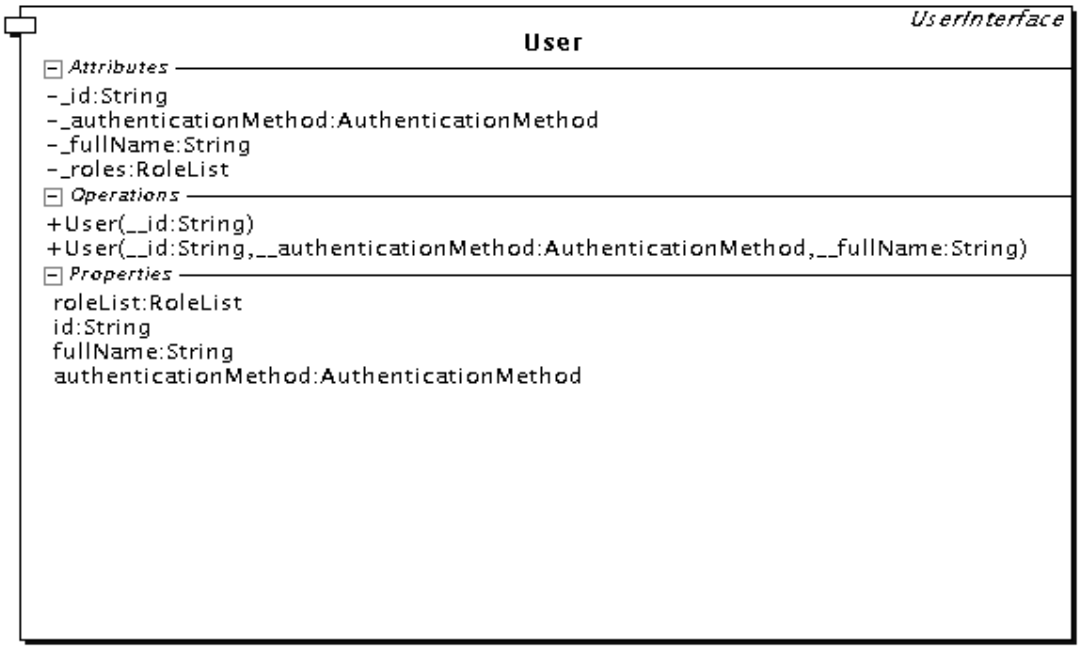


Figura 9 - Diagrama da Classe User.

A classe *Role*, que representa o elemento papel do modelo RBAC, possui cinco atributos, como pode ser visto na Figura 10:

- _OID: identificador do papel, que deve ser único por papel;
- _descr: descrição do papel;
- _hRoleList: lista de papéis do qual esse papel herda as permissões. Isto significa que o *framework* suporta hierarquia de papéis (Regra 1, capítulo 2, item 2.3);
- _permissions: lista de permissões que estão relacionadas com o papel;
- _cardinality: cardinalidade do papel, ou seja, indica o número de

usuários que podem estar relacionados com este papel ao mesmo tempo (Regra 3, capítulo 2, item 2.4).

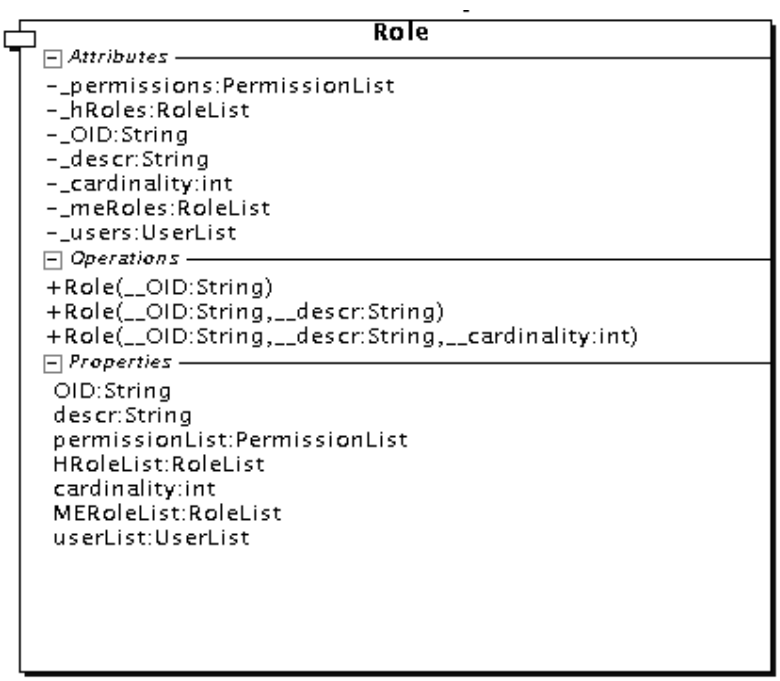


Figura 10 - Diagrama da Classe Role.

A classe *Permission*, que representa o elemento operação do modelo RBAC, possui apenas dois atributos, como pode ser visto na Figura 11:

- _OID: identificador da operação, que deve ser único por operação;
- _descr: descrição da operação.

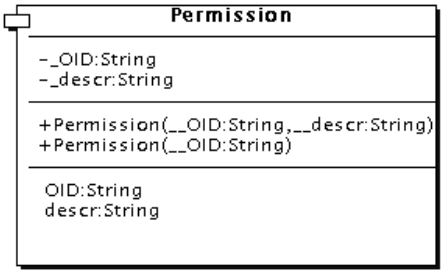


Figura 11 - Diagrama da classe Permission.

Outro componente da camada básica é a interface *AuthenticationMethod*, que funciona como um tipo abstrato, cujas classes concretas que implementam esta interface correspondem a métodos de autenticação, como *Password* (senha), *PrivateKey* (chave privada), PBE (*Password-Based Encryption*[4]) e PPK (*Password and Private Key*), que utiliza chave privada e senha conjuntamente, sendo um exemplo de criação de um novo método de autenticação utilizando outros métodos já existentes.

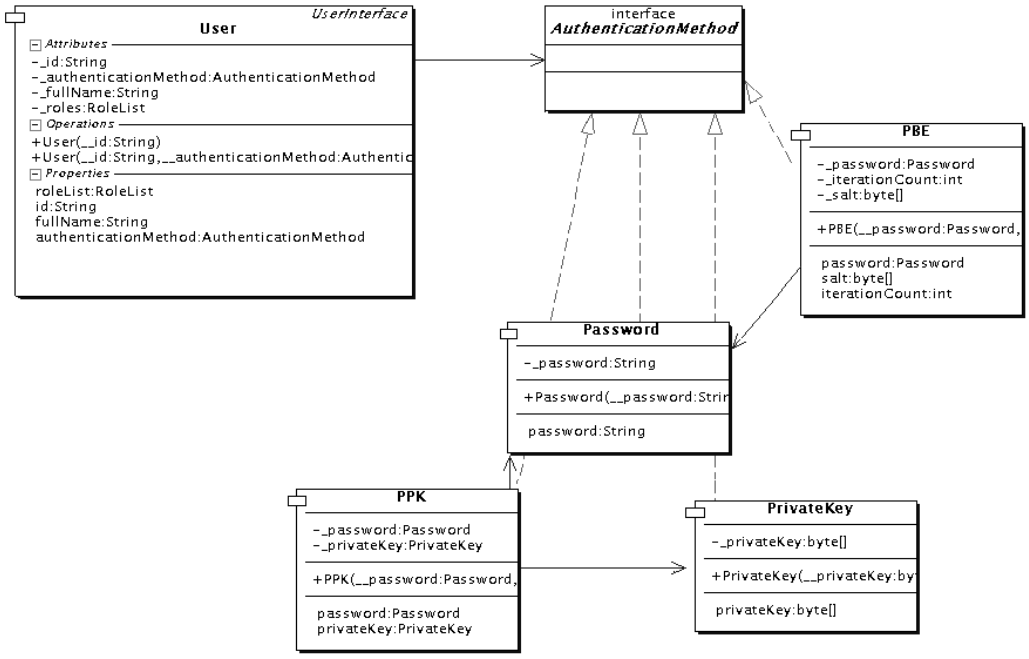


Figura 12 - Classes correspondentes aos tipos de Autenticação.

4.2.3.2 Camada de Persistência

A camada de persistência é de grande importância, pois é através de sua utilização que os elementos do modelo RBAC poderão ser armazenados e recuperados de mecanismos de armazenamento persistentes. Uma qualidade desejável desta parte do *framework* é que ela seja extensível, ou seja, que possa suportar qualquer tipo de mecanismo de armazenamento persistente, principalmente banco de dados relacionais e

[4]*Password-Based Encryption* é um mecanismo de criptografia que origina uma chave de criptografia a partir de uma senha. Com o objetivo de tornar o método mais seguro, a maior parte das implementações de PBE utilizam além da senha, um número aleatório, conhecido como *salt*, para criar a chave.

arquivos comuns. A implementação da camada de persistência deste trabalho deverá dar maior prioridade ao primeiro mecanismo, pois este oferece maior controle de segurança aos dados do que o segundo.

De acordo com [LAR00] , um *framework* para persistência é um conjunto de classes reutilizáveis – e usualmente extensíveis – que fornece serviços para objetos persistentes. Tipicamente, um *framework* para persistência tem que traduzir objetos para registros, salvá-los em um banco de dados e traduzir registros para objetos, quando for recuperar os mesmos do banco de dados.

A camada de persistência foi baseada no exemplo de *framework* de persistência de Craig Larman em [LAR00] , que utiliza alguns padrões como *Object Identifier*, *DataBase Broker*, *Cache Management*, *Template Method*, *Virtual Proxy* e *Factory Method* de [BRO96] . Ele utiliza o mecanismo de materialização sob demanda. A materialização é o ato de transformar uma representação de dados não-orientada a objeto (por exemplo, registros), existente em um armazenamento persistente, em objetos. A desmaterialização é a atividade oposta, também conhecida como passivação. A materialização sob demanda (também chamada de *Lazy*) é um mecanismo onde nem todos os objetos são materializados de uma só vez, uma instância em particular só é materializada sob demanda, quando necessária [LAR00] .

É desejável ter uma forma consistente de relacionar objetos com registros e garantir que a materialização repetida de um objeto não resulte em objetos duplicados. Neste caso, o *framework* de persistência utiliza o padrão *Object Identifier* para atribuir um identificador de objeto (OID) para cada registro de objeto. Um OID é geralmente um valor alfanumérico, que deve ser único para cada instância de objeto. Se cada objeto está associado a um OID, e cada tabela tem um OID como chave básica, então cada objeto pode ser mapeado de forma unívoca para alguma linha em alguma tabela.

O padrão *Database Broker* propõe a criação de uma classe responsável pela materialização, desmaterialização e pelo *caching* (memorização prévia) dos objetos. De acordo com [LAR00] , pode ser definida uma classe *broker* diferente para cada classe de objetos persistentes, e portanto, existirão diferentes tipos de *brokers* para diferentes tipos de armazenamento. O projeto de *Database Brokers* é baseado no padrão *Template Method*. A idéia deste padrão é definir um método gabarito (o *template method*) em uma

superclasse que define o esqueleto de um algoritmo, com suas partes variantes e invariantes. O *template method* invoca outros métodos, alguns dos quais são operações que podem ser redefinidas em uma subclasse. Assim, as subclasses podem redefinir os métodos que variam, de forma a acrescentar o seu próprio comportamento específico nos pontos de variação [LAR00] .

Um outro requisito desejável é manter os objetos materializados em um *cache* (depósito temporário de objetos) local. Isso possibilita melhorar o desempenho, pois a materialização é relativamente lenta. O padrão *Cache Management* de [BRO96] propõe tornar os *Database Brokers* responsáveis pela manutenção do seu *cache*. Se for usado um *broker* diferente para cada classe de objeto persistente, cada um desses *brokers* poderá manter seu próprio *cache.*

Em algumas vezes é desejável adiar a materialização de um objeto até que seja absolutamente necessária. Isso é conhecido como materialização sob demanda. Esse tipo de materialização pode ser implementada utilizando o padrão *Virtual Proxy*.

Um *Virtual Proxy* é uma referência inteligente para o objeto real. Ele materializa o objeto real quando referenciado pela primeira vez, portanto implementa a materialização sob demanda. É um objeto "leve" segundo [LAR00] , que substitui um objeto real que pode, ou não, estar materializado. Um *Virtual Proxy* é considerado uma *smart reference* porque é tratado por um cliente como se fosse um objeto.

Outro padrão que também foi utilizado foi o *Factory Method*, que define uma *interface* para criação de um objeto, mas deixa para as subclasses decidirem quais classes irão instanciar.

A camada de persistência foi idealizada de forma que atenda a todos estes padrões apresentados anteriormente. Primeiramente criou-se a superclasse das classes *brokers*, denominada de *PersistentBroker*. Esta classe, como pode ser visto na Figura 13, possui apenas três métodos: *objectWith (OID)*, *inCache (OID)* e *materializeWith(OID)*, sendo que o primeiro é concreto e os outros dois são abstratos. O método *objectWith* é um *template method*, que retorna o objeto referente ao OID passado como parâmetro. O algoritmo deste método é simples, ele verifica se o objeto está em *cache* utilizando o método *inCache*, se estiver simplesmente retorna o objeto, e se não estiver, ele chama o método *materializeWith*, para realizar a materialização do objeto do banco de dados. O

método *inCache* verifica se o objeto já foi trazido para a memória. Caso ele já esteja na memória, o próprio objeto é retornado, senão é retornado um objeto nulo. *A* implementação dos métodos abstratos *materializeWith* e *inCache* devem ser definidas nas subclasses de *PersistentBroker*, pois dependem de como o objeto será armazenado e do tipo de objeto.

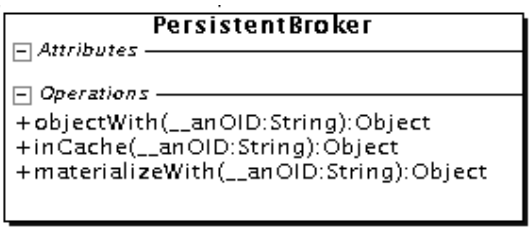


Figura 13 - Diagrama da classe PersistentBroker.

O principal mecanismo de armazenamento das informações referentes ao modelo RBAC é um banco de dados relacional. Deste modo, foi criada uma subclasse de *PersistentBroker*, denominada *RelationalPersistentBroker*, que é responsável pela materialização de objetos do banco de dados.

Esta subclasse possui também apenas três métodos: *materializeWith(OID)*, *selectFirst(OID)* e *currentRecordAsObject()*, sendo o primeiro concreto e os outros dois abstratos. O método *materializeWith* também é um *template method*, mas que implementa o método abstrato *materializeWith* de *PersistentBroker*. O algoritmo de *materializeWith* também é simples, ele faz uma chamada ao método *selectFirst* e depois uma chamada ao método *currentRecordAsObject*. O método *selectFirst* tem como objetivo executar a consulta do registro referente ao objeto no banco de dados e o método *currentRecordAsObject* tem como objetivo transformar o registro obtido na consulta em um objeto. Esses dois métodos deverão ser implementados pelas subclasses de *RelationalPersistentBroker* pois dependem do tipo de objeto que será materializado.

Como já foi dito na seção de projeto de arquitetura, o banco de dados escolhido para a implementação deste trabalho foi o Oracle 8i, mas a migração do *framework* para a utilização de outro banco de dados é simples, bastando apenas substituir o *driver* JDBC (*Java Database Connectivity*). O modelo do banco de dados é apresentado no Anexo 3.

Como os três elementos principais do modelo RBAC são *User*, Role e *Permission*, foram incluídos no *Persistence Framework* as subclasses *brokers* de *RelationalPersistentBroker* referentes a esses objetos: *RelationalBrokerUser*, *RelationalBrokerRole* e *RelationalBrokerPermission*. A principal função dessas subclasses é implementar os métodos abstratos *inCache*, *selectFirst* e *currentRecordAsObject* de acordo com o tipo de elemento (*User, Role* ou *Permission*) que representam, como pode ser visto na Figura 14.

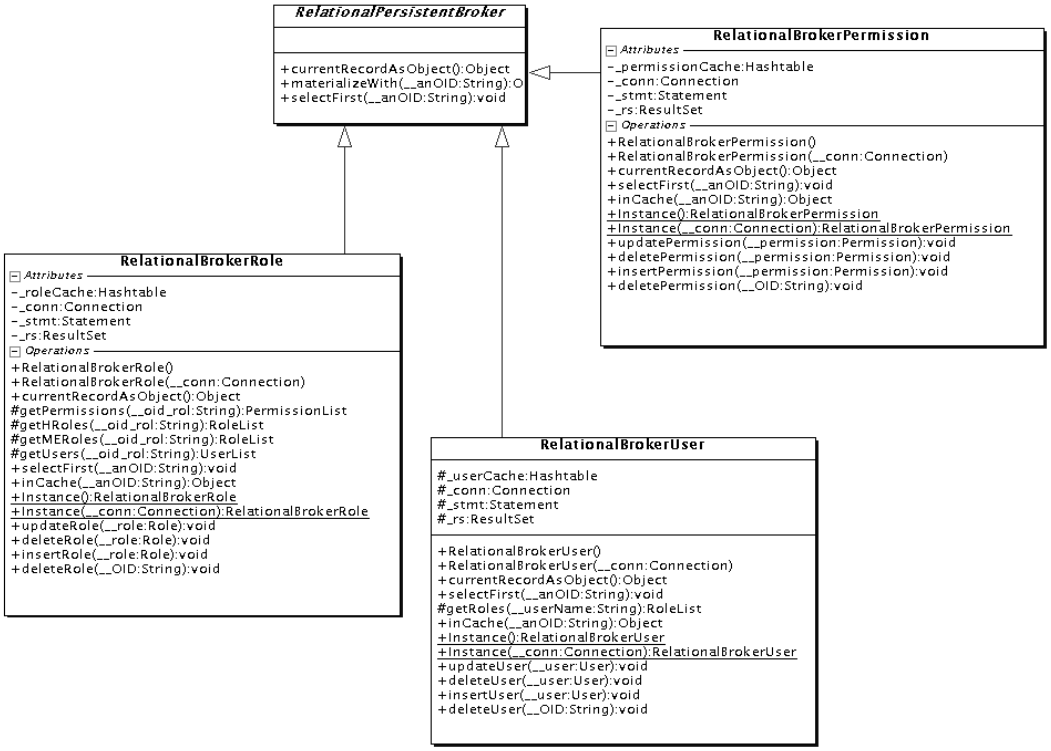


Figura 14 - Diagramas das classes brokers dos elementos do modelo RBAC.

Baseando-se nos padrões *Factory Method*, *Database Broker* e *Proxy*, foi criada a superclasse *VirtualProxy* (Figura 15). Esta classe possui atributos como o OID do objeto a qual se refere (*_OID*), o próprio objeto real (*_realSubject*), uma tabela *Hash* contendo os *brokers* que já foram utilizados (*_brokers*) e uma referência para o último *broker* que foi chamado (*_broker*).

Já que a classe *Virtual Proxy* é a superclasse dos objetos *proxy*, que irão substituir

os objetos reais, é necessário que todas as classes herdeiras dessa classe implementem o método *materializeSubject*. Além desse método existe o método abstrato *createBroker*, que deverá ser redefinido pelas subclasses de *VirtualProxy*, pois cada subclasse terá um tipo de *broker* diferente.

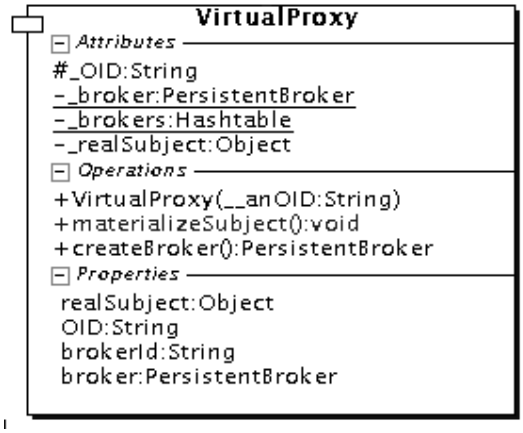


Figura 15 - Diagrama de classe de Virtual Proxy.

A partir deste ponto tornou-se necessário então a criação de um objeto *proxy* para cada classe representante de um elemento do modelo RBAC. Portanto, foram incluídas no *Persistence Framework*, as classes *UserProxy* (Figura 16), *RoleProxy* (Figura 17) e *PermissionProxy* (Figura 18). Todas as três classes herdam da superclasse *VirtualProxy*, e sobrescrevem o método *createBroker*, sendo que a classe *UserProxy* cria um *broker* de *RelationalBrokerUser*, a classe *RoleProxy* cria um *broker* de *RelationalBrokerRole* e por fim, a classe *PermissionProxy* cria um *broker* de *RelationalBrokerPermission*. Além disso, cada uma dessas classes redefine alguns métodos das classes que irão substituir, como por exemplo, a classe *UserProxy* redefine o método *getFullName* da classe *User*, sendo que dentro deste método é invocada uma chamada do método de mesmo nome do objeto real.

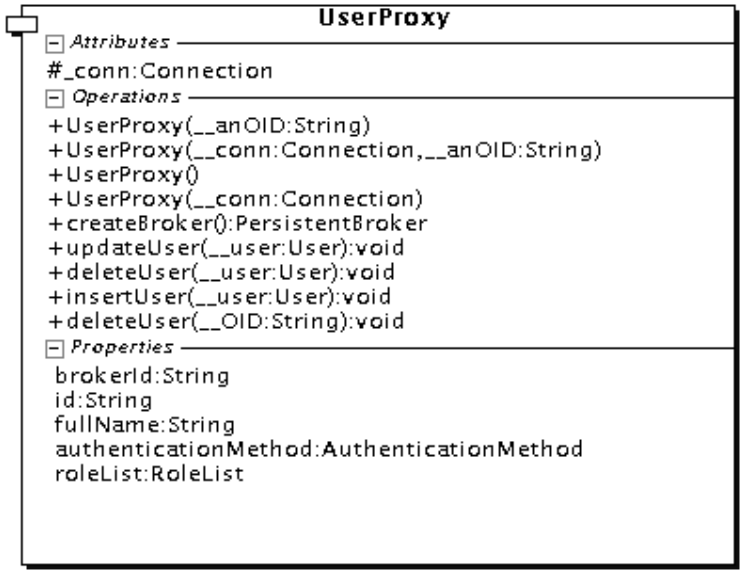


Figura 16 - Diagrama da classe UserProxy.

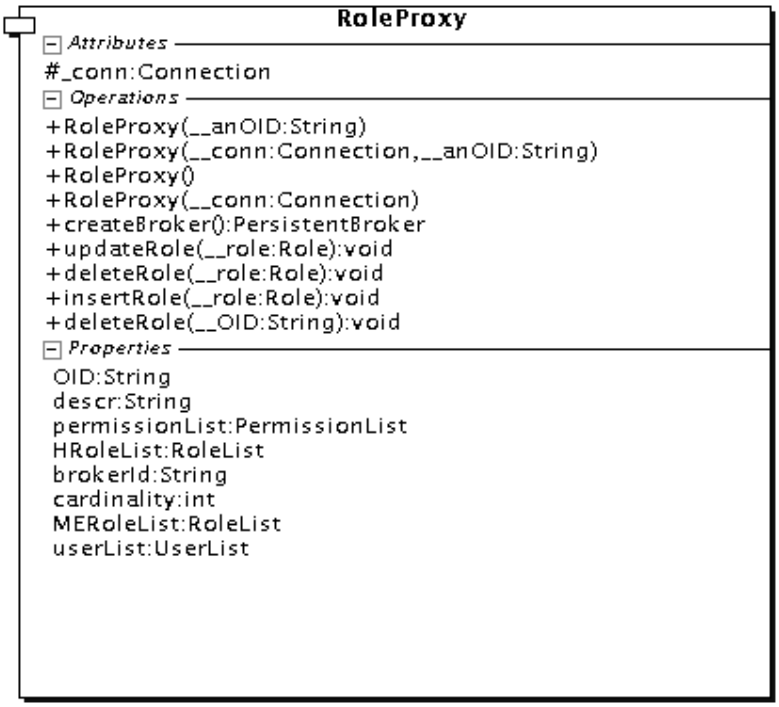


Figura 17 - Diagrama da classe RoleProxy.

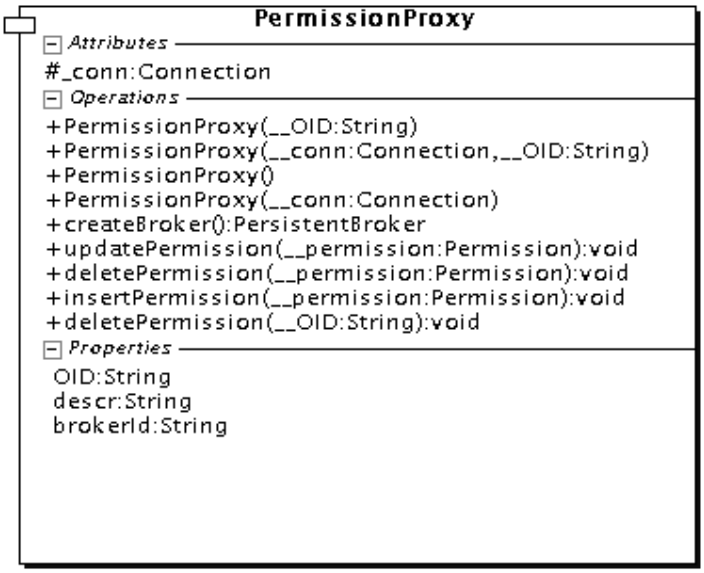


Figura 18 - Diagrama da classe PermissionProxy.

4.2.3.3 Camada de Aplicação

A camada de aplicação é aquela que contém as classes que utilizarão as camadas básica e de persistência para dar suporte à criação de aplicações de segurança seguindo o modelo de controle de acesso RBAC. Esta camada contém classes que são suporte à autenticação de usuários, ao controle de acesso, a auditoria e administração das informações de segurança. A camada de aplicação é aquela que contém a *interface* do *framework* com as aplicações que serão desenvolvidas a partir deste.

a) Autenticação

Existe uma variedade de métodos de autenticação de usuários, e estes métodos formam a base dos sistemas de controle de acesso. As três categorias de métodos para verificação da identidade de um usuário são baseadas em algo que o usuário sabe, tal qual uma senha; algo que o usuário possui, tal qual um *token* de autenticação; e alguma característica física do usuário, tal qual a impressão digital ou padrão de voz.

A autenticação pelo conhecimento é o modo mais utilizado para fornecer uma identidade a um computador, no qual destaca-se o uso de segredos, como senhas, chaves de criptografia, PIN *(Personal Identification Number*) e tudo mais que uma pessoa

pode saber. Entretanto, este tipo de autenticação tem algumas limitações: elas podem ser adivinhadas, roubadas ou esquecidas. Soluções alternativas como perguntas randômicas e senhas descartáveis geralmente são simples de utilização e bem aceita pelos usuários, baratas e fáceis de implementar. Além disso, não requerem hardware adicional como outras soluções baseadas em propriedade e características. Outra vantagem que vale destacar é que elas podem ser integradas em sistemas baseados em rede e na Web, além de diversos sistemas operacionais [FIO99] . Elas evitam vários problemas de ataques e problemas baseados em senha, mas não impedem que um usuário divulgue seu segredo para outro.

A utilização de perguntas randômicas, porém, adiciona uma dificuldade adicional ao usuário que quiser divulgar seu segredo, pois, ao contrário de contar apenas uma palavra (como no caso de senhas), terá que divulgar todas as informações constantes no questionário que serve de base para as perguntas randômicas.

As soluções de autenticação orientadas na propriedade se baseiam em um objeto físico que o usuário possui. Sua vantagem está relacionada com o princípio de que a duplicação do objeto de autenticação será mais cara que o valor do que está sendo guardado. As desvantagens deste tipo de autenticação são, que os objetos físicos podem ser perdidos ou esquecidos além do custo adicional do *hardware* [FIO99] . O problema da utilização destes dispositivos em aplicações Web em Intranets é que, assim como as senhas podem ser divulgadas para outras pessoas, os *tokens* também podem ser emprestados. Aliado ao custo do produto, isto faz com esta solução torne-se inviável para essas aplicações.

O último método de autenticação de usuários é baseado nas características físicas dos indivíduos. Os sistemas biométricos se baseiam em características fisiológicas e comportamentais de pessoas vivas. Os principais sistemas biométricos utilizados nos dias de hoje são baseados no reconhecimento de face, impressão digital, geometria da mão, íris, retina, padrão de voz, assinatura e ritmo de digitação. As vantagens desses sistemas são que eles não podem ser forjados nem tampouco esquecidos, obrigando que a pessoa a ser autenticada esteja fisicamente presente no ponto de autenticação. A desvantagem reside na falta de padrões, desconforto de usar alguns dispositivos biométricos e custo extras dos equipamentos envolvidos [FIO99] .

Para dar suporte ao elemento de autenticação, foram criadas classes concretas que já implementam a categoria de autenticação através de senhas, por ser o tipo de

autenticação atualmente mais utilizada em aplicações com interface Web. Por enquanto a autenticação utilizando perguntas aleatórias não será implementada, mas esta característica poderá ser feita pelo usuário do *framework*, pois este dará suporte para tal através de classes abstratas. O mesmo ocorre para os outros tipos de autenticação, como aqueles que utilizam objetos físicos ou sistemas biométricos.

A interface *Authentication*,(Figura 19), possui um único método, *isValidUser (id:String, auth:AuthenticationMethod)*, que serve para verificar se um determinado usuário é válido. A classe que implementa esta interface é *AuthenticationSeguraweb*, que é a classe da camada de aplicação responsável por realizar a autenticação do usuário. Esta classe implementa o método *isValidUser* da interface *Authentication* utilizando o mecanismo de senha como método de autenticação do usuário, por ser este método o mais utilizado atualmente para aplicações Web.

Como a parte de autenticação não faz parte do assunto principal deste trabalho, o modelo RBAC, não foi dispendida tanta atenção a ela, pois esta só foi criada apenas para dar suporte a parte de autenticação.

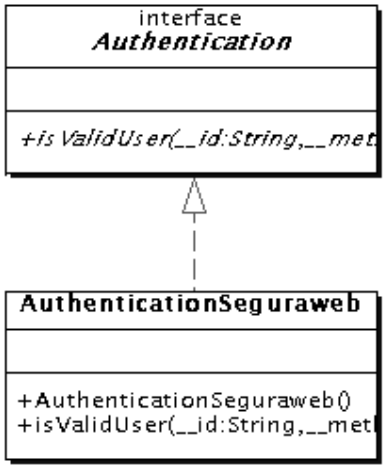


Figura 19 - Componentes de Autenticação.

b) Controle de Acesso

O objetivo desta parte do *framework* é dar suporte a verificação se os usuários possuem autorização ou não para realizar as operações, de acordo com o papel que o usuário possui no sistema. Esta parte é formada apenas por uma *interface* e uma classe concreta que implementa esta *interface*, como pode ser visto na Figura 20.

A *interface Authorization* possui apenas dois métodos que têm o mesmo objetivo, retornar se o usuário possui permissão ou não de executar uma operação. A classe *AuthorizarionSeguraweb* implementa esta *interface* utilizando as classes da camada de persistência para obter as informações do banco de dados. Por este motivo um dos atributos da classe é a conexão com o banco de dados.

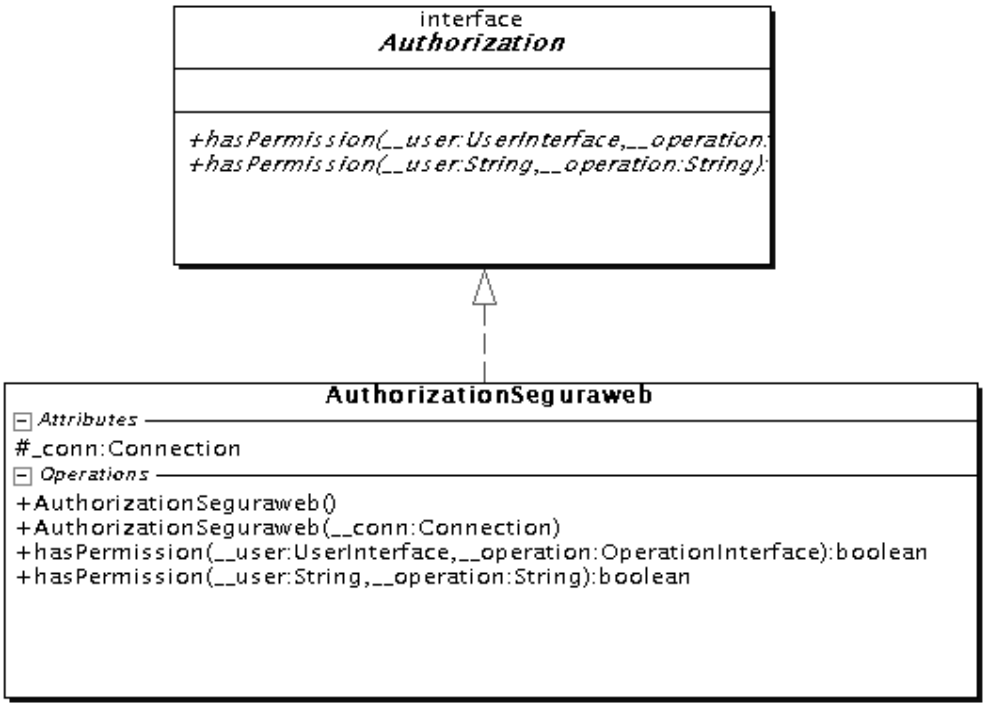


Figura 20 - Diagrama de classes de Autorização.

c) Administração

Esta parte do *framework* tem como objetivo dar suporte à gerência das informações de segurança de acordo com as regras do modelo de segurança RBAC. Para alcançar este objetivo foram criadas classes responsáveis pela inserção, remoção e atualização de usuários, papéis e permissões; bem como classes responsáveis por gerenciar a criação de relacionamentos entre usuários e papéis e entre papéis e

permissões.

A classe *DBUserManager* é a classe do *framework* que poderá ser utilizada para inserir e remover usuários, além de obter e alterar informações dos mesmos no banco de dados relacional. Esta classe pode ser vista na Figura 21.

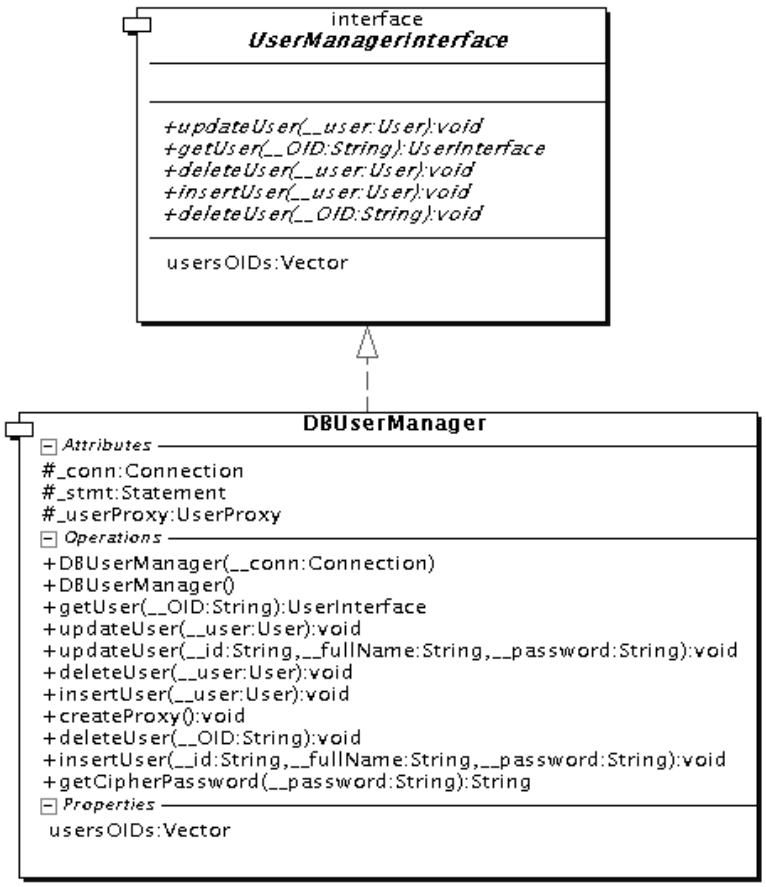


Figura 21 - Diagrama da Classe DBUserManager.

Como pode ser visto na Figura 21, a classe *DBUserManager* possui os seguintes métodos que são acessíveis ao usuário do *framework* Seguraweb:

- *getUser*: método para obter o objeto *proxy* referente ao usuário cujo identificador é passado como parâmetro;
- *updateUser*: atualiza as informações do usuário no banco de dados;

- *insertUser*: insere um novo usuário no banco de dados;
- *deleteUser*: remove o usuário no banco de dados.

A classe *DBRoleManager* dá suporte a inserção e remoção de papéis, bem como obtenção e alteração das informações dos mesmos no banco de dados relacional. Esta classe pode ser vista na Figura 22.

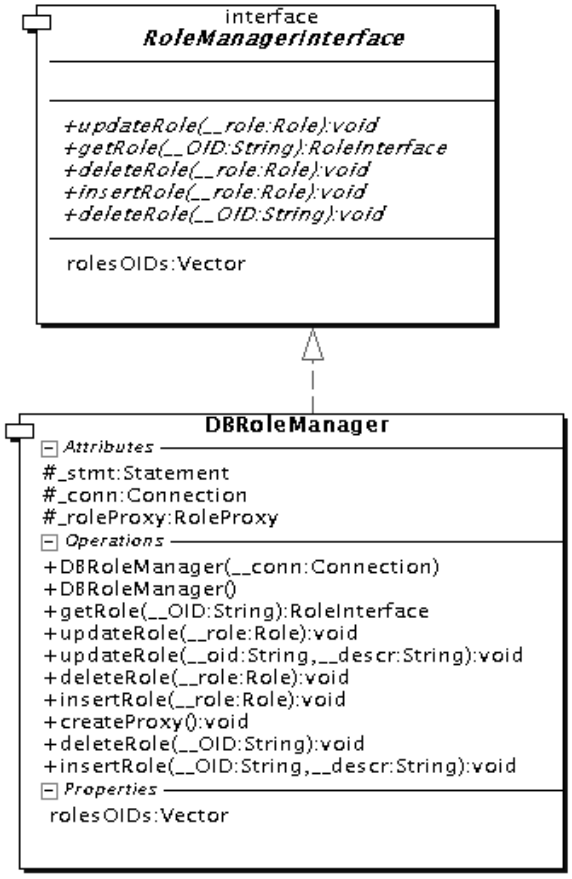


Figura 22 - Diagrama da Classe Role Manager.

Como pode ser visto na Figura 22, a classe *DBRoleManager* possui os seguintes métodos com acesso público:

- *getRole*: obtém o objeto *proxy* referente ao papel cujo identificador é passado como parâmetro;

- *updateRole*: atualiza as informações do papel no banco de dados;
- *insertRole*: insere um novo papel no banco de dados;
- *deleteRole*: remove o papel no banco de dados.

A classe *DBPermissionManager* possui métodos para inserir e remover permissões, além de obter e alterar as informações das mesmas no banco de dados relacional. Esta classe é apresentada na Figura 23.

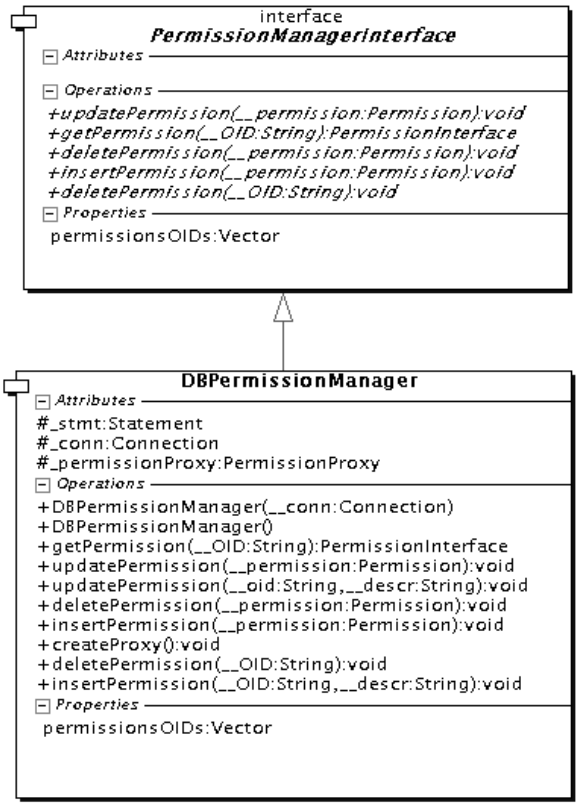


Figura 23 - Diagrama da Classe DBPermissionManager.

Como pode ser visto na Figura 23, a classe *DBPermissionManager* os métodos a seguir:

- *getPermission*: obtém o objeto *proxy* referente à permissão cujo identificador é passado como parâmetro;
- *updatePermission*: atualiza as informações da permissão no banco de

dados;

- *insertPermission*: insere uma nova permissão no banco de dados;
- *deletePermission*: remove a permissão no banco de dados.

A classe *RBACUserRoleAssociationManager* (Figura 24) é a classe que dá suporte a criação e remoção de autorizações para usuários se tornarem membros de papéis, bem como da ativação e desativação de papéis para esses usuários, sempre seguindo algumas regras do modelo de segurança RBAC. No decorrer do texto será apresentado como essas regras do modelo de segurança RBAC foram implementadas dentro do *framework* Seguraweb.

Os métodos da classe *RBACUserRoleAssociationManager* são descritos abaixo:

- *makeAssociation*: estes métodos têm como objetivo autorizar um usuário a se tornar membro do papel;
- *numberUsersAssociated*: retorna o número de usuários que estão atualmente autorizados ao papel cujo identificador é passado como parâmetro;
- *isAssociated*: retorna verdadeiro se o usuário já está autorizado a ser membro de um papel e falso caso contrário;
- *deleteAssociation*: retira a autorização do usuário ser membro do papel;
- *isME*: retorna se o papel cujo identificador é passado como parâmetro é mutuamente exclusivo a algum papel da lista de papéis ativos do usuário;
- *activateAssociation*: ativa o papel para o usuário, verificando se este é autorizado ou não;
- *disactivateAssociation*: desativa o papel para o usuário autorizado.

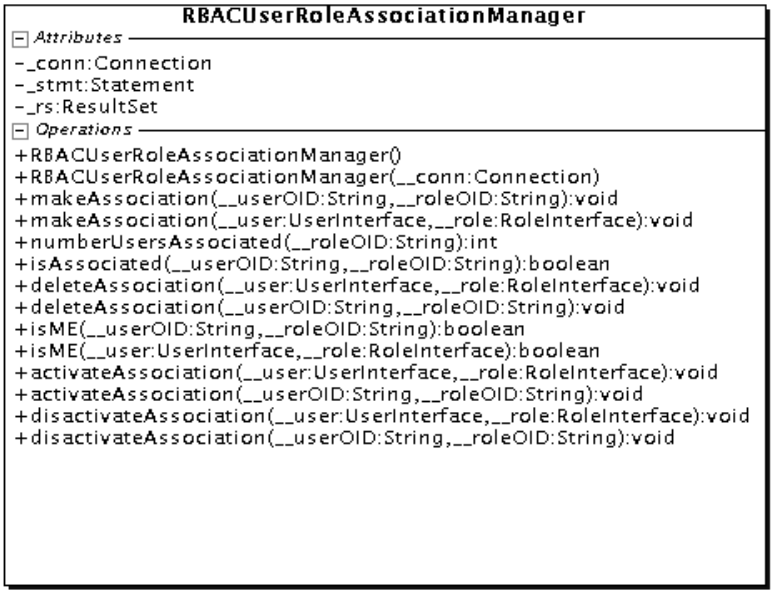


Figura 24 - Diagrama da classe RBACUserRoleAssociationManager.

A classe responsável pela associação de papéis e permissões é a classe *RBACRolePermissionAssociationManager* (Figura 25). Os métodos dessa classe são descritos a seguir:

- *makeAssociation*: realiza a associação entre o papel e a permissão;
- *isAssociated*: retorna verdadeiro caso a associação entre o papel e permissão já existe, e falso caso a associação não exista;
- *deleteAssociation*: remove a associação entre o papel e a permissão.

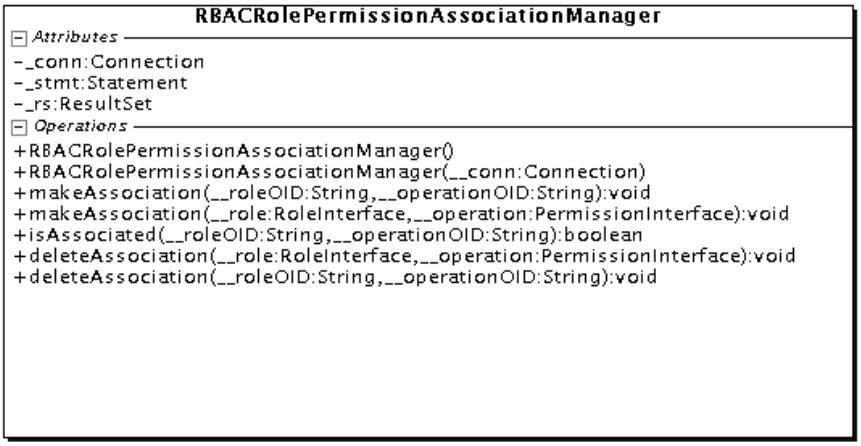


Figura 25 - Diagrama da classe RBACRolePermissionAssociationManager.

4.2.4 Implementação das Regras do Modelo RBAC

Como a proposta do *framework* Seguraweb é implementar o modelo de segurança RBAC, é necessário apresentar como esta implementação foi feita. A seguir serão apresentadas as regras do modelo e como estas foram implementadas no *framework* Seguraweb.

a) Hierarquia de Papéis

A regra da hierarquia de papéis foi implementada criando o atributo *_hRoles* do tipo *RoleList* na classe *Role* (Figura 10). Esse atributo contém a lista dos *proxies* dos papéis na qual o papel herda as permissões. Existe uma outra lista, *_permissions*, que contém as permissões que estão relacionadas com o papel. No momento da materialização do papel, as permissões relacionadas com este e com os papéis que estão na lista de hierarquia *_hRoles*, são carregadas para a memória.

b) Cardinalidade

A propriedade de cardinalidade foi implementada através da criação do atributo *_cardinality* na classe *Role* (Figura 10). Quando um papel é criado, deve ser definida uma cardinalidade para ele. Antes de realizar uma associação entre um usuário e um

papel, é verificado a quantidade de usuários que já estão relacionados com este papel e comparado com a cardinalidade dele. Se o número de usuários já relacionados ao papel for igual à cardinalidade deste, a associação entre o usuário e o papel não é realizada. Uma associação entre um usuário e um papel pode ser realizada utilizando os métodos *makeAssociation* da classe *RBACUserRoleAssociationManager* (Figura 24). O método *_numberUsersAssociated* é o responsável por obter o número de usuários já relacionados com o papel cujo identificador é passado como parâmetro.

c) Autorização de Papéis

Esta propriedade é implementada no método *activateAssociation*, que tem como objetivo ativar um papel para um usuário. Antes de realizar a ativação, é verificado se o papel está autorizado para o usuário utilizando o método *isAssociated*. Se o papel não estiver autorizado para o usuário, a sua ativação não é permitida.

d) Execução de Papéis, Autorização de Operação e Autorização de Acesso a Objeto

As regra de execução de papéis, autorização de operação e autorização de acesso a objeto foram implementadas nos métodos *hasPermission* da classe *AuthorizationSeguraweb* (Figura 20). Neste método é obtida a lista de papéis ativos do usuário. Para cada papel ativo é obtida a lista de permissões relacionadas. Neste momento é verificado se a permissão está contida na lista de permissões. Se a permissão estiver contida em alguma lista de permissões de algum papel ativo indica que o usuário pode executar a operação.

e) Separação Dinâmica de Tarefas

A regra da Separação Dinâmica de Tarefas foi implementada com a criação do atributo *_meRoles* do tipo *RoleList* na classe *Role*. Esse atributo contém a lista de papéis na qual o papel é mutuamente exclusivo. Assim, no momento em que é feita a ativação entre um usuário e um papel, é obtida a lista de papéis ativos do usuário (atributo *_roles* da classe *User*) e para cada papel desta lista é verificado se não está contido na lista de papéis mutuamente exclusivos do papel que se deseja fazer a ativação. Se existir pelo menos um papel mutuamente exclusivo, a ativação do papel para o usuário não é

realizada. Outro ponto a ser considerado é que, caso exista em um determinado momento, um usuário com dois papéis ativos e posteriormente estes papéis são definidos como mutuamente exclusivos, os dois papéis são automaticamente desativados para o usuário. Isto é realizado para evitar inconsistências, já que de acordo com a regra de Separação Dinâmica de Tarefas, um usuário não pode ter dois papéis mutuamente exclusivos ativos no mesmo momento.

Optou-se por implementar apenas a Separação Dinâmica de Tarefas por esta ser considerada menos rígida que a regra de Separação Estática de Tarefas, pois permite que o usuário esteja autorizado ao mesmo tempo a papéis mutuamente exclusivos, mas não permite que esses papéis sejam ativados ao mesmo tempo. A Separação Operacional de Tarefas também não foi implementada.

4.2.5 Aplicações de Teste

Seguindo o esquema de [BOS97] , a última etapa do desenvolvimento de um *framework* é a geração de uma aplicação de teste. Neste trabalho foram criadas duas aplicações de teste, uma para validar a parte de administração das informações de segurança e outra para validar o esquema de autorização.

a) A Aplicação RBACAdministration

Esta aplicação é um exemplo de aplicação que poderá ser utilizada pelo administrador da empresa para cadastrar usuários, papéis e permissões; autorizar e ativar papéis a usuários, definir a hierarquia de papéis e os papéis que são mutuamente exclusivos e por fim, associar papéis a permissões.

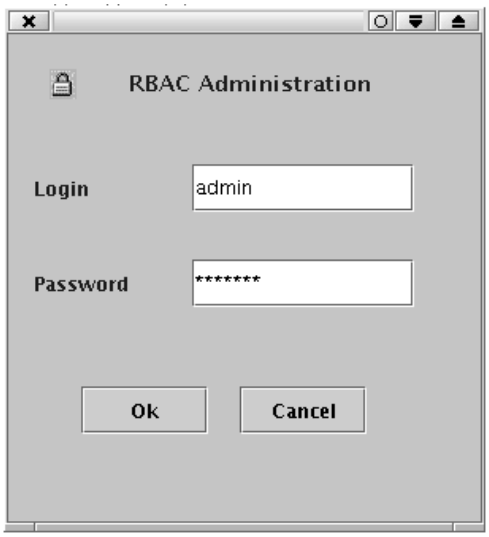


Figura 26 - Tela de login da aplicação RBAC Administration.

A tela de entrada do sistema é a tela de autenticação do usuário, como pode ser visto na Figura 26. Para utilizar esta aplicação é necessário que o usuário possua o papel de “administrador” ativo, ou que possua um papel que contenha o papel de “administrador” e que por sua vez, este papel também esteja ativo.

A aplicação *RBAC Administration* possui três painéis, como pode ser visto na Figura 27: *User*, *Role* e *Permission*. A partir do primeiro painel podem ser realizadas todas as ações relacionadas com o usuário, como cadastrar um novo usuário, consultar as informações de um determinado usuário, remover um usuário e também autorizar e ativar papéis aos usuários.

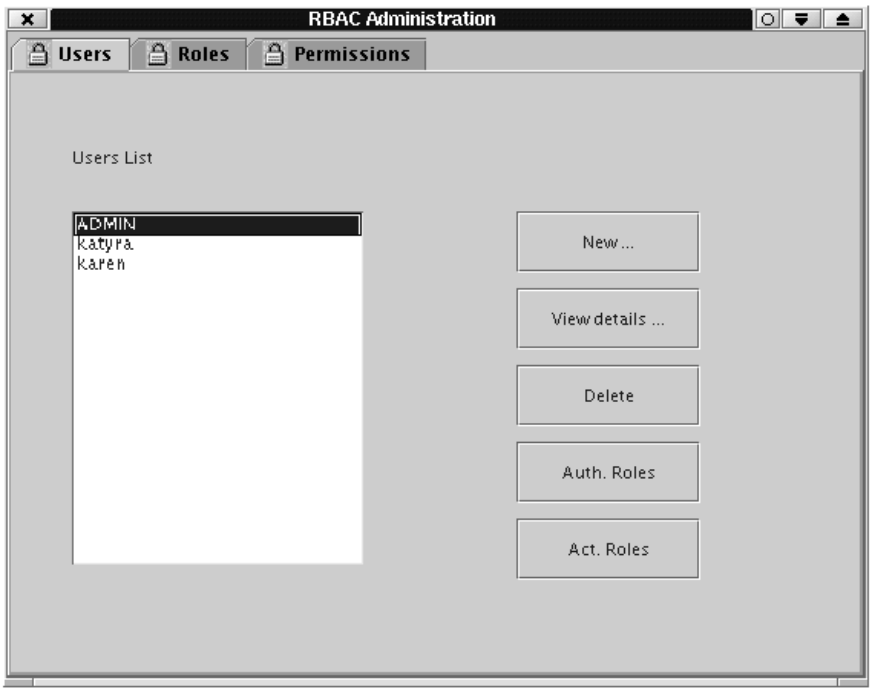


Figura 27 - Painel Users da aplicação RBAC Administration.

Para autorizar um papel a um usuário é necessário selecionar o usuário e apertar o botão "*Auth. Roles*". Neste momento será apresentada uma tela como a Figura 28 mostrando quais papéis estão autorizados para o usuário e os outros papéis que podem ser autorizados. Para autorizar um papel ao usuário basta mover o papel da lista de "*Other roles*" para a lista de "*User roles*". Uma tela semelhante a esta também é apresentada quando é apertado o botão "*Act. Roles*", só que servirá para gerenciar a ativação de papéis para o usuário selecionado.

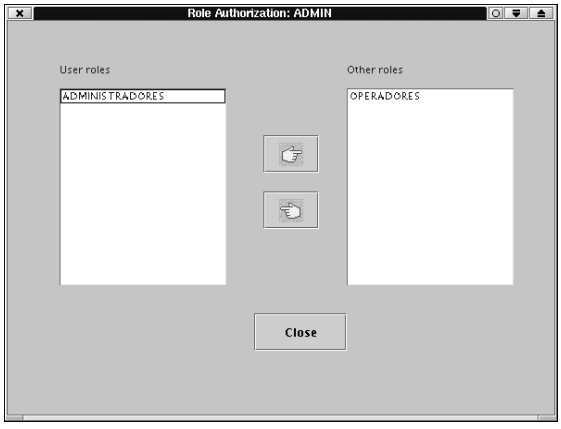


Figura 28 - Autorização de papéis em RBAC Administration.

A partir do segundo painel (Figura 29) podem ser realizadas as ações relacionadas com os papéis, como cadastrar um novo papel, consultar as informações de um determinado papel, remover um papel, associar permissões com um determinado papel, bem como definir a hierarquia de papéis e cadastrar papéis mutuamente exclusivos.

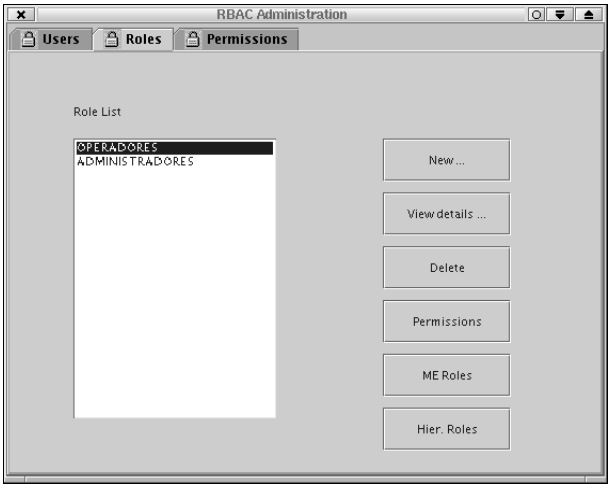


Figura 29 - Painel Role da aplicação RBAC Administration.

A hierarquia de papéis pode ser definida selecionando um papel e apertando o botão "*Hier. Roles*". Uma tela como a da Figura 30 será apresentada, mostrando quais são os papéis "pai" do papel selecionado, bem como os outros papéis que podem ser definidos como papéis "pai". Para definir um papel como "pai" é necessário mover o papel da lista de "*Other roles*" para a lista de "*Hierarchy Roles*".

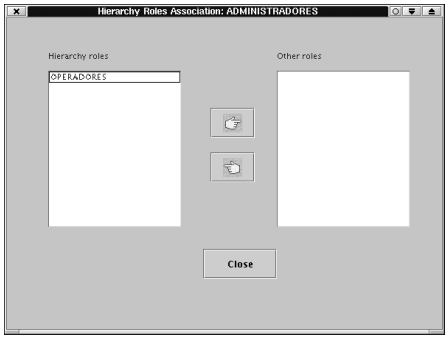


Figura 30 - Definição da hierarquia de papéis em RBAC Administration.

Para definição dos papéis mutuamente exclusivos é necessário selecionar um papel no painel *Role* e apertar o botão "*ME Roles*". Será apresentada uma janela (Figura 31) mostrando a lista de papéis que são mutuamente exclusivos ao papel selecionado, bem como a lista de papéis que podem se tornar mutuamente exclusivos. Para definir um papel como mutuamente exclusivo do papel selecionado é necessário mover o papel da lista de "*Other roles*" para a lista de "*Mutually exclusive roles*".

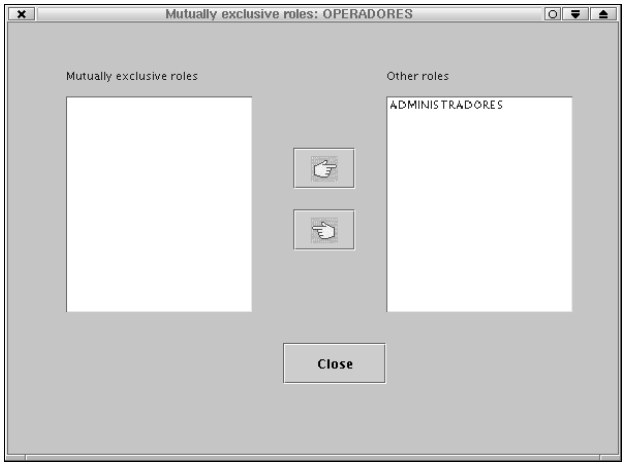


Figura 31 - Definição de papéis mutuamente exclusivos em RBAC Administration.

A partir do terceiro painel (Figura 32), podem ser realizadas as ações referentes às permissões, como cadastrar uma nova permissão, consultar as informações de uma permissão ou remover permissões.

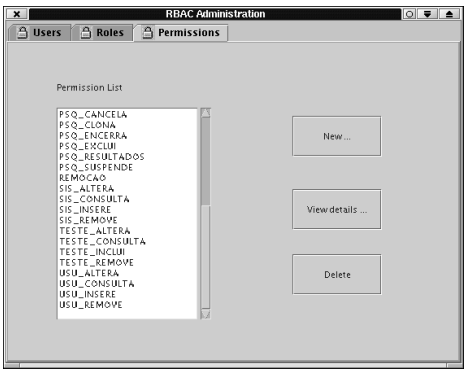


Figura 32 - Painel Permission da Aplicação RBAC Administration.

b) A Aplicação SecServer

Uma chamada telefônica é definida, segundo a ANATEL [ANA02], como uma ação realizada pelo chamador a fim de obter comunicação com o equipamento terminal desejado. Chamador é quem origina a chamada, e é também conhecido como Assinante A ou Originador. Quem recebe a chamada é conhecido como Chamado, Asssinante B ou Destino [SOU02].

Para que as prestadoras possam cobrar dos seus usuários estas chamadas é necessário registrar os dados das chamadas telefônicas, afim de permitir que posteriormente seja feito o cálculo dos custos das chamada e emissão de contas. O responsável por registrar todas as informações necessárias chama-se sistema de bilhetagem. As centrais telefônicas, além de servir como nó de comutação de chamadas, também podem executar a função de bilhetagem, mas nem todas executam esta função. As centrais com função de bilhetagem serão chamadas de centrais bilhetadoras [SOU02].

A central registra estas e outras informações da chamada num registro chamado bilhete ou CDR (Call Detail Record). A central bilhetadora armazena então este CDR juntamente com outros numa memória de massa. É este conjunto de CDR, que constituem um arquivo e que devem ser filtrados, dependendo das informações neles contidas [SOU02].

Este mesmo arquivo pode alimentar diferentes sistemas dentro de uma prestadora, mas geralmente contendo apenas informações úteis ao sistema em questão. Deste modo, é indesejável à área de faturamento receber bilhetes que não são faturáveis, provocando um processamento desnecessário. Os bilhetes não faturáveis, por exemplo por congestionamento na rede, são extremamente úteis na área de rede detectar os motivos e horários de maior congestionamento, enquanto que os faturáveis não possuem muita utilidade. Daí a importância da filtragem de bilhetes evitando o processamento de informação desnecessária [SOU02].

É neste ponto que o trabalho de mestrado de [SOU02] atua. O trabalho se trata de um mecanismo para a filtragem de CDRs. Este mecanismo é formado por um Servidor de Segurança, um Servidor de Aplicação e um Servidor de Compilação.

Resumidamente, os filtros de CDRs são construídos pelo usuário responsável pelas definições destes filtros. Após a definição dos filtros, estes serão enviados ao Servidor de Compilação, onde serão transformados em código fonte. A partir deste código fonte, são geradas *shared libraries* que estarão disponíveis para serem executadas pelo Servidor de Aplicação. Isso é feito através de uma requisição do usuário para aplicação de um determinado filtro sobre o fluxo de dados, ou seja, o conjunto de CDRs.

No trabalho de [SOU02], surgiu a necessidade de integração com um sistema de segurança, utilizando CORBA. Diante disto, bem como da necessidade de implementação de uma aplicação para validação do *framework* Seguraweb, foi criada a aplicação *SecServer*. Esta aplicação se trata de um servidor de segurança que atende a requisições de pedidos de autenticação de usuários e de verificações de controle de acesso, utilizando a tecnologia CORBA.

A definição da interface do sistema, pode ser encontradas no anexo 1.

4.3 Conclusões do Capítulo

Este capítulo apresentou alguns trabalhos relacionados com o trabalho realizado nesta dissertação, bem como foi feita uma breve comparação entre estes trabalhos e o *framework* Seguraweb. Após foram apresentadas as etapas de análise e projeto do *framework* Seguraweb, como análise de domínio, o projeto de arquitetura, o projeto do *framework* e a criação de aplicações de teste para a validação do *framework*. No momento da apresentação do projeto do *framework*, foi mostrada a estrutura deste e suas características, detalhando a implementação da persistência das informações de segurança e mostrando como cada regra do modelo de segurança RBAC foi implementada dentro do *framework*.

Capítulo 5

Conclusões

5.1 Revisão das motivações e objetivos

O objetivo principal deste trabalho foi o de projetar e implementar um *framework* para fazer o controle de acesso de aplicações com interface Web ou quaisquer outras aplicações Java utilizadas no ambiente corporativo de uma empresa. Este *framework* tem como característica principal retirar a preocupação do desenvolvedor destas aplicações de implementar a parte de controle de acesso de suas aplicações.

A política de segurança escolhida foi a política baseada em papéis, pois esta é considerada atualmente como a mais adequada para utilização em um ambiente corporativo. Para tal, foram realizados estudos do modelo de segurança RBAC (*Role-Based Access Control*) do NIST. Além do estudo do modelo RBAC, também foram necessários estudos de como projetar e implementar *frameworks* orientados a objeto.

5.2 Visão geral do trabalho

O texto da dissertação inicialmente apresentou as características do modelo de segurança RBAC, bem como as regras definidas neste modelo e algumas características de cada modelo pertencente à família de modelos RBAC do NIST.

Depois é discutida a importância da reusabilidade de *software* e como esta pode ser obtida com a utilização de *frameworks*. Foram expostos alguns conceitos de *frameworks*, suas características principais, os tipos de classificação e o ciclo de vida. Após foram recomendadas algumas atividades que devem ser realizadas no desenvolvimento de *frameworks*, bem como as metodologias de desenvolvimento existentes e a utilização de padrões de projeto.

Alguns trabalhos relacionados ao realizado nesta dissertação foram mostrados, bem como foi feita uma breve comparação entre estes e o *framework* Seguraweb. Foram

discutidas as etapas de análise e projeto do *framework* Seguraweb, como análise de domínio, o projeto de arquitetura, o projeto do *framework* (mostrando como algumas das modelo RBAC foram implementadas) e a criação de aplicações de teste para a validação do *framework.*

No decorrer do desenvolvimento foram surgindo idéias, como também foram encontrados alguns problemas não previstos inicialmente. Uma idéia foi a de importar o framework para dentro do banco de dados Oracle 8i, já que este permite a implementação de stored-procedures em Java. Infelizmente não foi possível realizar esta tarefa completamente pois o framework foi construído utilizando a JVM (Java Virtual Machine) versão 1.4.0 da Sun Microsystems, enquanto que o banco de dados Oracle utiliza a versão 1.1.8 da IBM. O principal motivo do conflito entre as versões foi a utilização da API JCE, que vem incluída na versão 1.4.0 da Sun, mas que é inexistente na versão 1.1.8 da IBM. Provavelmente esta incorporação deve ser possível na versão Oracle 9i, mas não pôde ser testada.

5.3 Contribuições e escopo do trabalho

Considerando os objetivos iniciais, algumas contribuições desta dissertação podem ser citadas:

1. O projeto e implementação de um *framework* baseado no modelo de segurança RBAC, que também dá suporte à autenticação, auditoria, administração e persistência das informações de segurança;

2. A utilização de padrões de projeto na implementação de *framework* RBAC, o que permitiu a implementação de técnicas como a materialização sob demanda, fazendo com que apenas informações que serão utilizadas sejam carregadas para a memória;

3. A definição de um modelo de dados para o armazenamento das informações dos elementos do modelo de segurança RBAC em uma base de dados, e que também dá suporte à implementação das regras deste modelo;

4. A proposta de um modo de implementação das regras do modelo RBAC.

5.4 Perspectivas futuras

Existem algumas possibilidades de continuidade deste trabalho. A principal delas é a evolução da estrutura do *framework*, onde poderão ser incluídas novas funcionalidades, de acordo com a necessidade dos desenvolvedores que irão utilizá-lo. Um exemplo de evolução é o aprimoramento do esquema de autenticação, que poderia ser estendido para a utilização de chaves públicas, já que atualmente o *framework* só permite a utilização de senhas e chaves privadas. A parte de auditoria também poderia ser melhorada, no sentido de ser melhor analisada e projetada, com o objetivo de facilitar seu uso.

Outro trabalho que poderia ser realizado é a implementação da camada de persistência utilizando arquivos textos ou banco de dados orientados a objetos para o armazenamento das informações de segurança.

Por fim, outra sugestão é a realização de testes de importação do *framework* na versão Oracle 9i e utilização do mesmo na implementação de *stored-procedures* em Java.

Anexo 1 – Definição IDL do SecServer

```
module SecurityServer {
    interface SecServer {
  boolean isValidUser(in string username, in string password);
  string getServerIOR(in string serverId);
  boolean setServerIOR(in string serverId, out string serverIOR);
  boolean hasPermissionTo(in string username, in string permission);
    };
};
```

Anexo 2 – Documentação Javadoc do *Framework* Seguraweb

Interface UserInterface

All Known Implementing Classes:
User

---

public interface **UserInterface**

Interface referente ao elemento usuário do modelo RBAC.

Method Summary

`java.lang.String`	**`getFullName`**`()` Retorna o nome completo do usuário.
`java.lang.String`	**`getId`**`()` Retorna o identificador do usuário.

Method Detail

getId

```
public java.lang.String getId()
                       throws java.lang.Exception
```

Retorna o identificador do usuário.

---

getFullName

```
public java.lang.String getFullName()
                            throws java.lang.Exception
```

Retorna o nome completo do usuário.

Class User

```
java.lang.Object
  |
  +--User
```

All Implemented Interfaces:

UserInterface

Direct Known Subclasses:

UserSeguraweb

---

public class **User**
extends java.lang.Object
implements UserInterface

Classe que identifica o elemento Usuário no modelo de segurança RBAC.

Constructor Summary

User(java.lang.String __id) Construtor que cria um objeto User com o identificador do usuário passado como parâmetro.
User(java.lang.String __id, AuthenticationMethod __authenticationMethod, java.lang.String __fullName) Construtor que cria um objeto User com o identificador do usuário e o método de autenticação passados como parâmetro.

Method Summary

AuthenticationMethod	**getAuthenticationMethod**() Método para obter o método de autenticação do usuário.
java.lang.String	**getFullName**() Método para obter o nome completo do usuário.
java.lang.String	**getId**() Método para obter o identificador do usuário.
RoleList	**getRoleList**() Método para obter a lista de papéis associada ao usuário.
void	**setRoleList**(RoleList __roles) Altera a lista de papéis relacionada com o usuário.

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Constructor Detail

User

```
public User(java.lang.String __id)
```

Construtor que cria um objeto User com o identificador do usuário passado como parâmetro.

Parameters:

`__id:` - string de identificação do usuário.

User

```
public User(java.lang.String __id,
            AuthenticationMethod __authenticationMethod,
            java.lang.String __fullName)
```

Construtor que cria um objeto User com o identificador do usuário e o método de autenticação passados como parâmetro.

Parameters:

`__id:` - identificador do usuário.

`__method:` - método de autenticação utilizado para autenticar o usuário.

Method Detail

setRoleList

```
public void setRoleList(RoleList __roles)
```

Altera a lista de papéis relacionada com o usuário.

---

getId

```
public java.lang.String getId()
                       throws java.lang.Exception
```

Método para obter o identificador do usuário.

Specified by:

`getId` in interface `UserInterface`

---

getFullName

```
public java.lang.String getFullName()
                             throws java.lang.Exception
```

Método para obter o nome completo do usuário.

Specified by:

`getFullName` in interface `UserInterface`

---

getAuthenticationMethod

```
public AuthenticationMethod getAuthenticationMethod()
```

throws
java.lang.Exception
Método para obter o método de autenticação do usuário.

---

getRoleList

public RoleList **getRoleList**()
Método para obter a lista de papéis associada ao usuário.

Class UserList

```
java.lang.Object
  |
  +--UserList
```

---

public class **UserList**
extends java.lang.Object

Classe que representa uma lista de usuários.

Constructor Summary

UserList()

Method Summary

java.util.Enumeration	**elements**() Retorna uma enumeração contendo os usuários da lista.
UserInterface	**get**(int __index) Método que retorna o usuário armazenado na posição da lista que é passada como parâmetro.
java.util.Vector	**getAllUsers**() Retorna o vetor de usuários.
void	**put**(UserInterface __user) Método que inclui um usuário no final da lista.
void	**put**(UserInterface __user, int __index) Método que insere um usuário na posição determinada pelo índice.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

UserList

```
public UserList()
```

Method Detail

put

```
public void put(UserInterface __user)
```

Método que inclui um usuário no final da lista.

---

put

```
public void put(UserInterface __user,
                int __index)
```

Método que insere um usuário na posição determinada pelo índice.

---

get

```
public UserInterface get(int __index)
```

Método que retorna o usuário armazenado na posição da lista que é passada como parâmetro.

---

elements

```
public java.util.Enumeration elements()
```

Retorna uma enumeração contendo os usuários da lista.

---

getAllUsers

```
public java.util.Vector getAllUsers()
```

Retorna o vetor de usuários.

Interface RoleInterface

All Known Implementing Classes:

Role

---

public interface **RoleInterface**

Interface referente ao elemento Role (papel) do modelo RBAC.

Method Summary

`java.lang.String`	**`getDescr`**`()` Retorna a descrição do papel.
`java.lang.String`	**`getOID`**`()` Retorna o identificador do papel.

Method Detail

getOID

```
public java.lang.String getOID()
                        throws java.lang.Exception
```

Retorna o identificador do papel.

---

getDescr

```
public java.lang.String getDescr()
                          throws java.lang.Exception
```

Retorna a descrição do papel.

Class Role

```
java.lang.Object
  |
  +--Role
```

All Implemented Interfaces:

RoleInterface

---

public class **Role**
extends java.lang.Object
implements RoleInterface

Classe referente ao elemento Role (papel) do modelo RBAC.

Constructor Summary

Role(java.lang.String __OID) Construtor que cria um objeto Role com o identificador passado como parâmetro.
Role(java.lang.String __OID, java.lang.String __descr) Construtor que cria um objeto Role com o identificador e a descrição passados como parâmetro.
Role(java.lang.String __OID, java.lang.String __descr, int __cardinality) Construtor que cria um objeto Role com o identificador, a descrição e a cardinalidade passados como parâmetro.

Method Summary

void	**addHierarchyRole**(RoleInterface __role) Adiciona o papel que é passado como parâmetro na lista de hierarquia de papéis.
void	**addMERole**(RoleInterface __role) Adiciona o papel que é passado como parâmetro na lista de papéis mutuamente exclusivos.
void	**delHierarchyRole**(RoleInterface __role) Retira o papel que é passado como parâmetro da lista de hierarquia de papéis.
void	**delHierarchyRole**(java.lang.String __roleOID) Retira o papel cujo identificador é passado como parâmetro da lista de hierarquia de papéis.
void	**delMERole**(RoleInterface __role) Retira o papel que é passado como parâmetro da lista de papéis mutuamente exclusivos.

void	**delMERole**(java.lang.String __roleOID) Retira o papel cujo identificador é passado como parâmetro da lista de papéis mutuamente exclusivos.
int	**getCardinality**() Obtém a cardinalidade do papel.
java.lang.String	**getDescr**() Retorna a descrição do papel.
RoleList	**getHRoleList**() Retorna a lista de hierarquia do papel.
RoleList	**getMERoleList**() Obtém a lista de papéis mutuamente exclusivos ao papel.
java.lang.String	**getOID**() Retorna o identificador do papel.
PermissionList	**getPermissionList**() Retorna a lista de permissões relacionadas ao papel.
UserList	**getUserList**() Retorna a lista de usuários associados ao papel.
boolean	**isHierarchyRole**(RoleInterface __role) Retorna verdadeiro se o papel que é passado como parâmetro está na lista de hierarquia de papéis.
boolean	**isHierarchyRole**(java.lang.String __roleOID) Retorna verdadeiro se o papel cujo identificador é passado como parâmetro está na lista de hierarquia de papéis.
boolean	**isME**(RoleInterface __role) Retorna verdadeiro caso o papel que é passado como parâmetro seja um papel mutuamente exclusivo.
boolean	**isME**(java.lang.String __roleOID) Retorna verdadeiro caso o papel cujo identificador é passado como parâmetro seja um papel mutuamente exclusivo.
void	**setCardinality**(int __cardinality) Altera a cardinalidade do papel.
void	**setHRoleList**(RoleList __hRoles) Altera a lista de hierarquia do papel.
void	**setMERoleList**(RoleList __meRoles) Altera a lista de papéis mutuamente exclusivos ao papel.
void	**setPermissionList**(PermissionList __permissions) Altera a lista de permissões relacionadas ao papel.
void	**setUserList**(UserList __users) Altera a lista de usuários associados ao papel.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll,

toString, wait, wait, wait

Constructor Detail

Role

```
public Role(java.lang.String __OID)
```

Construtor que cria um objeto Role com o identificador passado como parâmetro.

Parameters:

`__OID:` - identificador do papel.

Role

```
public Role(java.lang.String __OID,
            java.lang.String __descr)
```

Construtor que cria um objeto Role com o identificador e a descrição passados como parâmetro.

Parameters:

`__OID:` - identificador do papel.

`__descr:` - descrição do papel.

Role

```
public Role(java.lang.String __OID,
            java.lang.String __descr,
            int __cardinality)
```

Construtor que cria um objeto Role com o identificador, a descrição e a cardinalidade passados como parâmetro.

Parameters:

`__OID:` - identificador do papel.

`__descr:` - descrição do papel.

`__cardinality:` - cardinalidade do papel.

Method Detail

getOID

```
public java.lang.String getOID()
```

Retorna o identificador do papel.

Specified by:

`getOID` in interface `RoleInterface`

---

getDescr

```
public java.lang.String getDescr()
```

Retorna a descrição do papel.

Specified by:

`getDescr` in interface `RoleInterface`

---

getPermissionList

`public PermissionList` **`getPermissionList`**`()`

Retorna a lista de permissões relacionadas ao papel.

---

setPermissionList

`public void` **`setPermissionList`**`(PermissionList __permissions)`

Altera a lista de permissões relacionadas ao papel.

---

getHRoleList

`public RoleList` **`getHRoleList`**`()`

Retorna a lista de hierarquia do papel.

---

setHRoleList

`public void` **`setHRoleList`**`(RoleList __hRoles)`

Altera a lista de hierarquia do papel.

---

getCardinality

`public int` **`getCardinality`**`()`

Obtém a cardinalidade do papel.

Specified by:

`getCardinality` in interface `RoleInterface`

---

setCardinality

`public void` **`setCardinality`**`(int __cardinality)`

Altera a cardinalidade do papel.

---

getMERoleList

`public RoleList` **`getMERoleList`**`()`

Obtém a lista de papéis mutuamente exclusivos ao papel.

Specified by:

`getMERoleList` in interface `RoleInterface`

---

setMERoleList

`public void` **`setMERoleList`**`(RoleList __meRoles)`

Altera a lista de papéis mutuamente exclusivos ao papel.

---

addMERole

`public void` **`addMERole`**`(RoleInterface __role)`

Adiciona o papel que é passado como parâmetro na lista de papéis mutuamente exclusivos.

Specified by:

`addMERole` in interface `RoleInterface`

---

delMERole

```
public void delMERole(RoleInterface __role)
              throws java.lang.Exception
```

Retira o papel que é passado como parâmetro da lista de papéis mutuamente exclusivos.

Specified by:

`delMERole` in interface `RoleInterface`

---

delMERole

```
public void delMERole(java.lang.String __roleOID)
              throws java.lang.Exception
```

Retira o papel cujo identificador é passado como parâmetro da lista de papéis mutuamente exclusivos.

Specified by:

`delMERole` in interface `RoleInterface`

---

addHierarchyRole

```
public void addHierarchyRole(RoleInterface __role)
```

Adiciona o papel que é passado como parâmetro na lista de hierarquia de papéis.

Specified by:

`addHierarchyRole` in interface `RoleInterface`

---

delHierarchyRole

```
public void delHierarchyRole(RoleInterface __role)
                    throws java.lang.Exception
```

Retira o papel que é passado como parâmetro da lista de hierarquia de papéis.

Specified by:

`delHierarchyRole` in interface `RoleInterface`

---

delHierarchyRole

```
public void delHierarchyRole(java.lang.String __roleOID)
                    throws java.lang.Exception
```

Retira o papel cujo identificador é passado como parâmetro da lista de hierarquia de papéis.

Specified by:

`delHierarchyRole` in interface `RoleInterface`

---

getUserList

```
public UserList getUserList()
```

Retorna a lista de usuários associados ao papel.

---

setUserList

```
public void setUserList(UserList __users)
```

Altera a lista de usuários associados ao papel.

---

isME

```
public boolean isME(java.lang.String __roleOID)
             throws java.lang.Exception
```

Retorna verdadeiro caso o papel cujo identificador é passado como parâmetro seja um papel mutuamente exclusivo. Caso contrário retorna falso.

Specified by:

`isME` in interface `RoleInterface`

---

isME

```
public boolean isME(RoleInterface __role)
             throws java.lang.Exception
```

Retorna verdadeiro caso o papel que é passado como parâmetro seja um papel mutuamente exclusivo. Caso contrário retorna falso.

Specified by:

`isME` in interface `RoleInterface`

---

isHierarchyRole

```
public boolean isHierarchyRole(java.lang.String __roleOID)
                        throws java.lang.Exception
```

Retorna verdadeiro se o papel cujo identificador é passado como parâmetro está na lista de hierarquia de papéis. Caso contrário retorna falso.

Specified by:

`isHierarchyRole` in interface `RoleInterface`

---

isHierarchyRole

```
public boolean isHierarchyRole(RoleInterface __role)
                        throws java.lang.Exception
```

Retorna verdadeiro se o papel que é passado como parâmetro está na lista de hierarquia de papéis. Caso contrário retorna falso.

Specified by:

`isHierarchyRole` in interface `RoleInterface`.

Class RoleList

```
java.lang.Object
  |
  +--RoleList
```

---

public class **RoleList**
extends java.lang.Object

Classe que representa uma lista de papéis.

Constructor Summary

RoleList() Construtor.

Method Summary

void	**delRole**(RoleInterface __role) Remove o papel que é passado como parâmetro da lista de papéis.
void	**delRole**(java.lang.String __roleOID) Remove o papel cujo OID é passado como parâmetro da lista de papéis.
java.util.Enumeration	**elements**() Retorna uma enumeração contendo os papéis da lista.
RoleInterface	**get**(int __index) Método que retorna o papel armazenado na posição da lista que é passada como parâmetro.
void	**put**(RoleInterface __role) Método que inclui um papel no final da lista.
void	**put**(RoleInterface __role, int __index) Método que insere um papel na posição determinada pelo índice.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RoleList

```
public RoleList()
```

Construtor.

Method Detail

put

```
public void put(RoleInterface __role)
```

Método que inclui um papel no final da lista.

Parameters:

`__role:` - papel a ser incluído na lista.

---

put

```
public void put(RoleInterface __role,
                int __index)
```

Método que insere um papel na posição determinada pelo índice.

Parameters:

`__role:` - papel a ser inserido.

`__index:` - posição na qual o papel deve ser inserido.

---

get

```
public RoleInterface get(int __index)
```

Método que retorna o papel armazenado na posição da lista que é passada como parâmetro.

---

elements

```
public java.util.Enumeration elements()
```

Retorna uma enumeração contendo os papéis da lista.

---

delRole

```
public void delRole(RoleInterface __role)
             throws java.lang.Exception
```

Remove o papel que é passado como parâmetro da lista de papéis.

---

delRole

```
public void delRole(java.lang.String __roleOID)
             throws java.lang.Exception
```

Remove o papel cujo OID é passado como parâmetro da lista de papéis.

Interface PermissionInterface

All Known Implementing Classes:
Permission, PermissionProxy

---

public interface **PermissionInterface**

Interface referente ao elemento Operantion do modelo RBAC.

Method Summary

`java.lang.String`	**`getDescr`**`()` Retorna a descrição da operação.
`java.lang.String`	**`getOID`**`()` Retorna o identificador da operação.

Method Detail

getOID

```
public java.lang.String getOID()
                        throws java.lang.Exception
```

Retorna o identificador da operação.

---

getDescr

```
public java.lang.String getDescr()
                          throws java.lang.Exception
```

Retorna a descrição da operação.

Class Permission

```
java.lang.Object
  |
  +--Permission
```

All Implemented Interfaces:

PermissionInterface

---

public class **Permission**
extends java.lang.Object
implements PermissionInterface

Classe referente ao elemento Permissão do modelo RBAC.

Constructor Summary

`Permission``(java.lang.String __OID)` Construtor que cria um objeto Permission com o identificador passado como parâmetro.
`Permission``(java.lang.String __OID, java.lang.String __descr)` Construtor que cria um objeto Permission com o identificador e a descrição passados como parâmetro.

Method Summary

`java.lang.String`	**`getDescr`**`()` Retorna a descrição da permissão.
`java.lang.String`	**`getOID`**`()` Retorna o identificador da permissão.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Permission

```
public Permission(java.lang.String __OID,
                  java.lang.String __descr)
```

Construtor que cria um objeto Permission com o identificador e a descrição passados como parâmetro.

Parameters:
`__OID:` - identificador da operação.
`__descr:` - descrição da operação.

Permission

`public` **`Permission`**`(java.lang.String __OID)`

Construtor que cria um objeto Permission com o identificador passado como parâmetro.

Parameters:
`__OID:` - identificador da operação.

Method Detail

getOID

`public java.lang.String` **`getOID`**`()`

Retorna o identificador da permissão.

Specified by:
`getOID` in interface `PermissionInterface`

---

getDescr

`public java.lang.String` **`getDescr`**`()`

Retorna a descrição da permissão.

Specified by:
`getDescr` in interface `PermissionInterface`

Class PermissionList

```
java.lang.Object
  |
  +--PermissionList
```

---

public class **PermissionList**
extends java.lang.Object

Classe que representa uma lista de permissões.

Constructor Summary

PermissionList() Construtor.

Method Summary

void	**addPermissionList**(PermissionList __permissions) Método que concatena todas as permissões da lista de operações que é passada como parâmetro na lista de permissões.
java.util.Enumeration	**elements**() Retorna uma enumeração contendo as permissões da lista.
PermissionInterface	**get**(int __index) Método que retorna a permissão armazenada na posição da lista que é passada como parâmetro.
java.util.Vector	**getAllPermissions**() Retorna o vetor de permissões.
void	**put**(PermissionInterface __permission) Método que inclui uma permissão no final da lista.
void	**put**(PermissionInterface __permission, int __index) Método que insere uma permissão na posição determinada pelo índice.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PermissionList

```
public PermissionList()
```

Construtor.

Method Detail

put

```
public void put(PermissionInterface __permission)
```

Método que inclui uma permissão no final da lista.

---

put

```
public void put(PermissionInterface __permission,
                int __index)
```

Método que insere uma permissão na posição determinada pelo índice.

---

get

```
public PermissionInterface get(int __index)
```

Método que retorna a permissão armazenada na posição da lista que é passada como parâmetro.

---

elements

```
public java.util.Enumeration elements()
```

Retorna uma enumeração contendo as permissões da lista.

---

getAllPermissions

```
public java.util.Vector getAllPermissions()
```

Retorna o vetor de permissões.

---

addPermissionList

```
public void addPermissionList(PermissionList __permissions)
```

Método que concatena todas as permissões da lista de operações que é passada como parâmetro na lista de permissões.

Interface Authentication

All Known Implementing Classes:

AuthenticationSeguraweb

---

public interface **Authentication**

Interface de autenticação.

Method Summary

boolean	**isValidUser**(java.lang.String __id, AuthenticationMethod __method) Método para verificação da validade de um usuário.

Method Detail

isValidUser

```
public boolean isValidUser(java.lang.String __id,
                           AuthenticationMethod __method)
```

Método para verificação da validade de um usuário.

Parameters:

`__id:` - string de identificação do usuário

`__method:` - tipo de autenticação a ser utilizada, como por exemplo senhas, chave privada etc.

Returns:

booleano identificando se o usuário é válido ou não. True: usuário é válido. False: usuário não é válido.

Class AuthenticationSeguraweb

```
java.lang.Object
  |
  +--AuthenticationSeguraweb
```

All Implemented Interfaces:

Authentication

---

public class **AuthenticationSeguraweb**
extends java.lang.Object
implements Authentication

Classe concreta que implementa a interface de autenticação.

Field Summary

protected java.sql.Connection	**_conn** Conexão com o banco de dados.

Constructor Summary

AuthenticationSeguraweb(java.sql.Connection __conn) Construtor.

Method Summary

boolean	**isValidUser**(java.lang.String __id, AuthenticationMethod __method) Retorna verdadeiro se o usuário é válido, caso contrário retorna falso.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

_conn

protected java.sql.Connection **_conn**
Conexão com o banco de dados.

Constructor Detail

AuthenticationSeguraweb

```
public AuthenticationSeguraweb(java.sql.Connection __conn)
```

Construtor.

Parameters:

`__conn:` - Objeto de conexão com o banco de dados

Method Detail

isValidUser

```
public boolean isValidUser(java.lang.String __id,
                           AuthenticationMethod __method)
```

Retorna verdadeiro se o usuário é válido, caso contrário retorna falso.

Specified by:

`isValidUser` in interface `Authentication`

Parameters:

`__id:` - Identificador único do usuário.

`__method:` - método de autenticação (senha).

Interface AuthenticationMethod

All Known Implementing Classes:
Password, PBE, PPK, PrivateKey

---

public interface **AuthenticationMethod**
Interface de métodos de autenticação.

Class Password

```
java.lang.Object
  |
  +--Password
```

All Implemented Interfaces:

AuthenticationMethod

---

public class **Password**
extends java.lang.Object
implements AuthenticationMethod

Classe que utiliza o mecanismo de senha como método de autenticação.

Constructor Summary

Password(java.lang.String __password) Construtor que cria um objeto Password com a senha que é passada como parâmetro.

Method Summary

java.lang.String	**getPassword**() Método que retorna a senha.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Password

```
public Password(java.lang.String __password)
```

Construtor que cria um objeto Password com a senha que é passada como parâmetro.

Method Detail

getPassword

```
public java.lang.String getPassword()
```

Método que retorna a senha.

Class PPK

```
java.lang.Object
  |
  +--PPK
```

All Implemented Interfaces:

AuthenticationMethod

---

public class **PPK**
extends java.lang.Object
implements AuthenticationMethod

Classe que utiliza o mecanismo de senha e chave privada em conjunto como método de autenticação.

Constructor Summary

PPK(Password __password, PrivateKey __privateKey) Construtor que cria um objeto PPK com a senha e a chave privada passados como parâmetros.	

Method Summary

Password	**getPassword**() Método para obter a senha.
PrivateKey	**getPrivateKey**() Método para obter a chave privada.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

PPK

```
public PPK(Password __password,
           PrivateKey __privateKey)
```

Construtor que cria um objeto PPK com a senha e a chave privada passados como parâmetros.

Method Detail

getPassword

`public Password getPassword()`
Método para obter a senha.

---

getPrivateKey

`public PrivateKey getPrivateKey()`
Método para obter a chave privada.

Interface Authorization

All Known Implementing Classes:

AuthorizationSeguraweb

---

public interface **Authorization**

Method Summary

boolean	**hasPermission**(java.lang.String __user, java.lang.String __operation)
boolean	**hasPermission**(UserInterface __user, PermissionInterface __operation)

Method Detail

hasPermission

```
public boolean hasPermission(UserInterface __user,
                             PermissionInterface __operation)
                      throws java.lang.Exception
```

---

hasPermission

```
public boolean hasPermission(java.lang.String __user,
                             java.lang.String __operation)
```

throws java.lang.Exception

Class AuthorizationSeguraweb

```
java.lang.Object
  |
  +--AuthorizationSeguraweb
```

All Implemented Interfaces:

Authorization

---

public class **AuthorizationSeguraweb**
extends java.lang.Object
implements Authorization

Classe concreta que implementa a interface de autorização.

Field Summary

protected java.sql.Connection	**_conn** Conexão com o banco de dados.

Constructor Summary

AuthorizationSeguraweb(java.sql.Connection __conn) Construtor.

Method Summary

boolean	**hasPermission**(java.lang.String __user, java.lang.String __permission) Retorna verdadeiro se o usuário cujo identificador é passado como parâmetro possui a permissão que é passada como parâmetro.
boolean	**hasPermission**(UserInterface __user, PermissionInterface __permission) Retorna verdadeiro se o usuário passado como parâmetro possui a permissão que é passada como parâmetro.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

_conn

```
protected java.sql.Connection _conn
```

Conexão com o banco de dados.

Constructor Detail

AuthorizationSeguraweb

```
public AuthorizationSeguraweb(java.sql.Connection __conn)
```

Construtor.

Parameters:

`__conn:` - Objeto de conexão com o banco de dados

Method Detail

hasPermission

```
public boolean hasPermission(UserInterface __user,
                             PermissionInterface __permission)
                      throws java.lang.Exception
```

Retorna verdadeiro se o usuário passado como parâmetro possui a permissão que é passada como parâmetro.

Specified by:

`hasPermission` in interface `Authorization`

Parameters:

`__user:` - usuário.

`__permission:` - permissão.

---

hasPermission

```
public boolean hasPermission(java.lang.String __user,
                             java.lang.String __permission)
                      throws java.lang.Exception
```

Retorna verdadeiro se o usuário cujo identificador é passado como parâmetro possui a permissão que é passada como parâmetro.

Specified by:

`hasPermission` in interface `Authorization`

Parameters:

`__user:` - Identificador único do usuário.

`__permission:` - Identificador único da permissão.

Class RBACUserRoleAssociationManager

```
java.lang.Object
  |
  +--RBACUserRoleAssociationManager
```

---

public class **RBACUserRoleAssociationManager**
extends java.lang.Object

Classe responsável pela associação de usuários e papéis.

Constructor Summary

RBACUserRoleAssociationManager() Construtor
RBACUserRoleAssociationManager(java.sql.Connection __conn) Construtor.

Method Summary

void	**activeAssociation**(java.lang.String __userOID, java.lang.String __roleOID) Ativa a associação entre o usuário e o papel cujos identificadores são passados como parâmetro.
void	**activeAssociation**(UserInterface __user, RoleInterface __role) Ativa a associação entre o usuário e o papel que são passados como parâmetro.
void	**deleteAssociation**(java.lang.String __userOID, java.lang.String __roleOID) Remove a associação entre o usuário e o papel cujos identificadores são passados como parâmetro.
void	**deleteAssociation**(UserInterface __user, RoleInterface __role) Remove a associação entre o usuário e o papel que são passados como parâmetro.
void	**disactiveAssociation**(java.lang.String __userOID, java.lang.String __roleOID) Desativa a associação entre o usuário e o papel cujos identificadores são passados como parâmetro.
void	**disactiveAssociation**(UserInterface __user, RoleInterface __role) Desativa a associação entre o usuário e o papel que são passados como parâmetro.
boolean	**isActive**(java.lang.String __userOID, java.lang.String __roleOID) Retorna verdadeiro caso a associação entre o usuário e o papel cujos

	identificadores são passados como parâmetros esteja ativa.
boolean	**isAssociated**(java.lang.String __userOID, java.lang.String __roleOID) Retorna verdadeiro caso o exista a associação entre o usuário e o papel cujos identificadores são passados como parâmetros.
boolean	**isME**(java.lang.String __userId, java.lang.String __roleOID) Retorna verdadeiro caso o papel cujo identificador é passado como parâmetro é mutuamente exclusivo a algum papel associado ao usuário cujo identificador é passado como parâmetro.
boolean	**isME**(UserInterface __user, RoleInterface __role) Retorna verdadeiro caso o papel que é passado como parâmetro é mutuamente exclusivo a algum papel associado ao usuário que é passado como parâmetro.
void	**makeAssociation**(java.lang.String __userOID, java.lang.String __roleOID) Realiza a associação entre o usuário cujo identificador é passado como parâmetro com o papel cujo identificador é passado como parâmetro.
void	**makeAssociation**(UserInterface __user, RoleInterface __role) Realiza a associação entre o usuário que é passado como parâmetro com o papel que é passado como parâmetro.
int	**numberUsersAssociated**(java.lang.String __roleOID) Retorna o número de usuários associados com o papel cujo identificador é passado como parâmetro.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

RBACUserRoleAssociationManager

```
public RBACUserRoleAssociationManager()
```

Construtor

RBACUserRoleAssociationManager

```
public RBACUserRoleAssociationManager(java.sql.Connection __conn)
```

Construtor.

Parameters:

__conn: - Objeto de conexão com o banco de dados.

Method Detail

makeAssociation

```
public void makeAssociation(java.lang.String __userOID,
                            java.lang.String __roleOID)
                     throws RBACAssociationException,
                            java.sql.SQLException,
                            java.lang.Exception
```

Realiza a associação entre o usuário cujo identificador é passado como parâmetro com o papel cujo identificador é passado como parâmetro. Se a associação não puder ser realizada, é gerada uma exceção do tipo RBACAssociationException.

Parameters:

`__userOID` - identificador do usuário

`__roleOID` - identificador do papel

---

makeAssociation

```
public void makeAssociation(UserInterface __user,
                            RoleInterface __role)
                     throws RBACAssociationException,
                            java.sql.SQLException,
                            java.lang.Exception
```

Realiza a associação entre o usuário que é passado como parâmetro com o papel que é passado como parâmetro. Se a associação não puder ser realizada, é gerada uma exceção do tipo RBACAssociationException.

Parameters:

`__user` - usuário

`__role` - papel

---

activeAssociation

```
public void activeAssociation(java.lang.String __userOID,
                              java.lang.String __roleOID)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

Ativa a associação entre o usuário e o papel cujos identificadores são passados como parâmetro. Se a ativação não puder ser realizada, uma exceção do tipo RBACAssociationException é gerada.

Parameters:

`__userOID` - identificador do usuário

`__roleOID` - identificador do papel

---

activeAssociation

```
public void activeAssociation(UserInterface __user,
                              RoleInterface __role)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

Ativa a associação entre o usuário e o papel que são passados como parâmetro. Se a ativação não puder ser realizada, uma exceção do tipo RBACAssociationException é gerada.

Parameters:

__user: - usuário
__role: - papel

---

numberUsersAssociated

```
public int numberUsersAssociated(java.lang.String __roleOID)
                          throws java.sql.SQLException
```

Retorna o número de usuários associados com o papel cujo identificador é passado como parâmetro.

Parameters:

__roleOID: - identificador do papel

---

isAssociated

```
public boolean isAssociated(java.lang.String __userOID,
                            java.lang.String __roleOID)
                     throws java.lang.Exception
```

Retorna verdadeiro caso o exista a associação entre o usuário e o papel cujos identificadores são passados como parâmetros. Caso contrário é retornado falso.

Parameters:

__userOID: - identificador do usuário
__roleOID: - identificador do papel

---

isActive

```
public boolean isActive(java.lang.String __userOID,
                        java.lang.String __roleOID)
                 throws java.lang.Exception
```

Retorna verdadeiro caso a associação entre o usuário e o papel cujos identificadores são passados como parâmetros esteja ativa. Caso contrário é retornado falso.

Parameters:

__userOID: - identificador do usuário
__roleOID: - identificador do papel

---

deleteAssociation

```
public void deleteAssociation(UserInterface __user,
                              RoleInterface __role)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

Remove a associação entre o usuário e o papel que são passados como parâmetro. Retorna um exceção do tipo RBACAssociationException caso a associação não exista.

Parameters:

__user: - usuário
__role: - papel

---

deleteAssociation

```
public void deleteAssociation(java.lang.String __userOID,
                              java.lang.String __roleOID)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

Remove a associação entre o usuário e o papel cujos identificadores são passados como parâmetro. Retorna um exceção do tipo RBACAssociationException caso a associação não exista.

Parameters:

`__userOID:` - identificador do usuário

`__roleOID:` - identificador do papel

---

disactiveAssociation

```
public void disactiveAssociation(java.lang.String __userOID,
                                 java.lang.String __roleOID)
                          throws RBACAssociationException,
                                 java.sql.SQLException,
                                 java.lang.Exception
```

Desativa a associação entre o usuário e o papel cujos identificadores são passados como parâmetro. Retorna um exceção do tipo RBACAssociationException caso a associação não exista.

Parameters:

`__userOID:` - identificador do usuário

`__roleOID:` - identificador do papel

---

disactiveAssociation

```
public void disactiveAssociation(UserInterface __user,
                                 RoleInterface __role)
                          throws RBACAssociationException,
                                 java.sql.SQLException,
                                 java.lang.Exception
```

Desativa a associação entre o usuário e o papel que são passados como parâmetro. Retorna um exceção do tipo RBACAssociationException caso a associação não exista.

Parameters:

`__user:` - usuário

`__role:` - papel

---

isME

```
public boolean isME(java.lang.String __userId,
                    java.lang.String __roleOID)
             throws java.lang.Exception
```

Retorna verdadeiro caso o papel cujo identificador é passado como parâmetro é mutuamente exclusivo a algum papel associado ao usuário cujo identificador é passado como parâmetro. Caso contrário retorna falso.

Parameters:

`__userId:` - identificador do usuário

`__roleOID:` - identificador do papel

---

isME

```
public boolean isME(UserInterface __user,
                    RoleInterface __role)
             throws java.lang.Exception
```

Retorna verdadeiro caso o papel que é passado como parâmetro é mutuamente exclusivo a algum papel associado ao usuário que é passado como parâmetro. Caso contrário retorna falso.

Parameters:

`__user:` - usuário

`__role:` - papel

Class RBACRolePermissionAssociationManager

```
java.lang.Object
  |
  +--RBACRolePermissionAssociationManager
```

---

public class **RBACRolePermissionAssociationManager**
extends java.lang.Object

Classe responsável pela associação de usuários e papéis.

Constructor Summary

RBACRolePermissionAssociationManager() Construtor
RBACRolePermissionAssociationManager(java.sql.Connection __conn) Construtor.

Method Summary

void	**deleteAssociation**(RoleInterface __role, PermissionInterface __permission) Retorna verdadeiro caso o exista a associação entre o papel e a permissão que são passados como parâmetros.
void	**deleteAssociation**(java.lang.String __roleOID, java.lang.String __permissionOID) Retorna verdadeiro caso o exista a associação entre o papel e a permissão cujos identificadores são passados como parâmetros.
boolean	**isAssociated**(java.lang.String __roleOID, java.lang.String __permissionOID) Retorna verdadeiro caso o exista a associação entre o papel e a permissão cujos identificadores são passados como parâmetros.
void	**makeAssociation**(RoleInterface __role, PermissionInterface __permission) Realiza a associação entre o papel que é passado como parâmetro com a permissão que é passada como parâmetro.
void	**makeAssociation**(java.lang.String __roleOID, java.lang.String __permissionOID) Realiza a associação entre o papel cujo identificador é passado como parâmetro com a permissão cujo identificador é passado como parâmetro.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll,

toString, wait, wait, wait

Constructor Detail

RBACRolePermissionAssociationManager

```
public RBACRolePermissionAssociationManager()
```

Construtor

RBACRolePermissionAssociationManager

```
public
RBACRolePermissionAssociationManager(java.sql.Connection __conn)
```

Construtor.

Parameters:

`__conn` - Objeto de conexão com o banco de dados.

Method Detail

makeAssociation

```
public void makeAssociation(java.lang.String __roleOID,
                            java.lang.String __permissionOID)
                     throws RBACAssociationException,
                            java.sql.SQLException,
                            java.lang.Exception
```

Realiza a associação entre o papel cujo identificador é passado como parâmetro com a permissão cujo identificador é passado como parâmetro. Se a associação não puder ser realizada, é gerada uma exceção do tipo RBACAssociationException.

Parameters:

`__roleOID` - identificador do papel

`__permissionOID` - identificador da permissão

---

makeAssociation

```
public void makeAssociation(RoleInterface __role,
                            PermissionInterface __permission)
                     throws RBACAssociationException,
                            java.sql.SQLException,
                            java.lang.Exception
```

Realiza a associação entre o papel que é passado como parâmetro com a permissão que é passada como parâmetro. Se a associação não puder ser realizada, é gerada uma exceção do tipo RBACAssociationException.

Parameters:

`__role` - papel

`__permission` - permissão

---

isAssociated

```
public boolean isAssociated(java.lang.String __roleOID,
```

```
                                java.lang.String __permissionOID)
                         throws java.lang.Exception
```

Retorna verdadeiro caso o exista a associação entre o papel e a permissão cujos identificadores são passados como parâmetros. Caso contrário é retornado falso.

Parameters:

`__roleOID:` - identificador do papel

`__permissionOID:` - identificador da permissão

---

deleteAssociation

```
public void deleteAssociation(RoleInterface __role,
                              PermissionInterface __permission)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

Retorna verdadeiro caso o exista a associação entre o papel e a permissão que são passados como parâmetros. Caso contrário é retornado falso.

Parameters:

`__role:` - papel

`__permission:` - permissão

---

deleteAssociation

```
public void deleteAssociation(java.lang.String __roleOID,
                              java.lang.String __permissionOID)
                       throws RBACAssociationException,
                              java.sql.SQLException,
                              java.lang.Exception
```

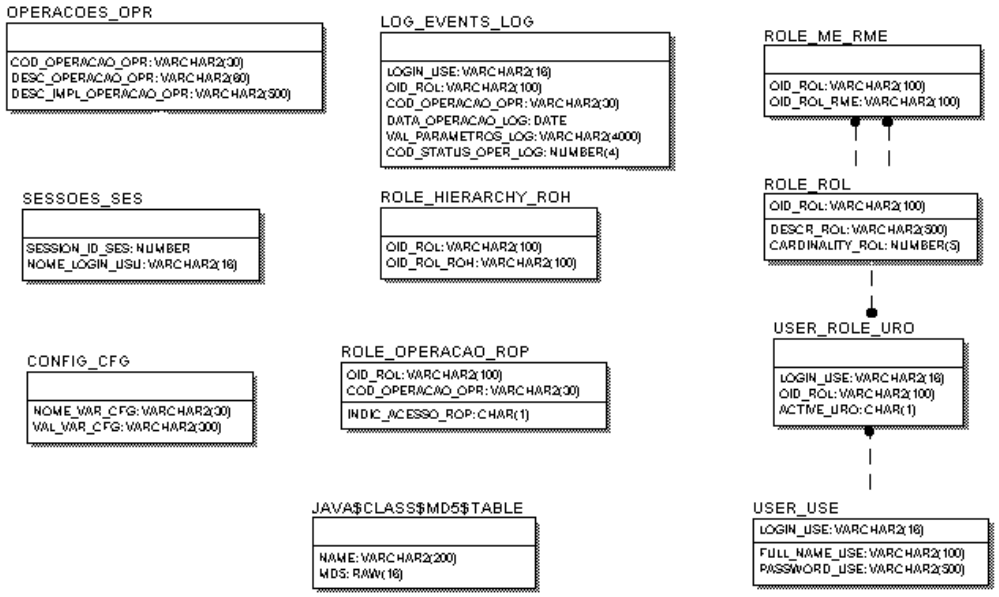
Retorna verdadeiro caso o exista a associação entre o papel e a permissão cujos identificadores são passados como parâmetros. Caso contrário é retornado falso.

Parameters:

`__roleOID:` - identificador do papel

`__permissionOID:` - identificador do permissão

Anexo 3 – Modelo do Banco de Dados



Referências Bibliográficas

[AHN01] AHN, Gail-Joon; SHIN, Michael E. **Role-Based Authorization Constraints Specification Using Object Constraint Language.** Department of Computer Science. University of North Carolina at Charlotte. Department of Information and Software Engineering. George Mason University. Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001. WET ICE 2001. Proceedings. Tenth IEEE International Workshops on 2001. Pg. 157-162.

[ALE77] ALEXANDER, Christopher. **A Pattern Language: Towns Buildings Construction.** 1977. New York: Oxford University Press.

[ANA02] ANATEL. **Glossário Termos Técnicos de Telecomunicações**. Disponível em: http://www.anatel.gov.br/AJUDA/GLOSSARIO/DEFAULT.ASP. Acesso em: 21 de março 2002.

[BAR97] BARKLEY, John F.; CINCOTTA, Anthony V.; FERRAIOLO, David F.; GRAVILLA, Serban; KUHN, D. Richard. **Role-Based Access Control for the World Wide Web.** 1997. NIST – National Institute of Standards and Technology.

[BEZ99] BEZNOSOV, Konstantin; DENG Yi. **A Framework Implementing Role-Based Access Control Using CORBA Security Service.** 1999. Center of Advanced Distributed Systems Engineering. School of Computer Science – Florida International University.

[BOS97] BOSCH, Jan; MOLIN, Peter; MATTSSON, Michael, BENGTSSON, PerOlof. **Object-Oriented Frameworks – Problems & Experiences.** 1997. Department of Computer Science and Business Administration. University of Karlskrona/Ronneby. Suécia. http://www.ipd.hk-r.se/michaelm/papers/ex-frame.ps

[BRO96] BROWN, K. WHITENACK, B. **Crossing Chasms. Pattern Languages of Program Design**. 1996. vol. 2. Reading, MA. Editora: Addison-Wesley.

[BUN01] BUNDY, Alan; BLEWITT, Alex; STARK, Ian. **Automatic Verification of Java Design Patterns.** Division of Informatics. University of Edinburg. IEEE- Automated Software Engineering, 2001. (ASE 2001). Proceedings. 16th Annual International Conference on, 2001. Pg. 324-327.

[BUS96] BUSCHMANN, Frank et al.1996.**Pattern - Oriented Software Architecture: A System of Patterns.** New York : J. Wiley.

[BUT00] BUTLER, Gregory. **Object-Oriented Application Frameworks.** Department of Computer Science. Concordia University. Montreal. http://www.cs.concordia.ca/~faculty/gregb/

[DEU89] DEUTSCH, Peter. **Frameworks and reuse in the smalltalk 80 system.** In: BIGGERSTAFF, T. **Software Reusability.** 1989. New York. ACM Press.

[EID02] EIDE, Eric; REID, Alastair, REGEHR, John; LEPREAU, Jay. **Static**

and Dynamic Structure in Design Patterns. 2002. University of Utah. School of Computing. IEEE - Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on , 2002 Pg: 208 –218.

[EDE98] EDEN, A. **LePUS – A Declarative Pattern Specification Language.** PhD Thesis, Department of Computer Science, TelAviv University, 1998. http://cs.concordia.ca/~faculty/eden/lepus/.

[FAY99] FAYAD, Mohamed; SCHMIDT, Douglas C. **Building Application Frameworks: Object-Oriented Foundations of Framework Design.** Setembro 1999. Editora John Wiley Professio. 1ª Edição. New York.

[FER95] FERRAIOLO, David F.; CUGINI, Janet A.; KUHN, D. Richard. **Role-Based Access Control (RBAC): Features and Motivations.** 1995. U. S. Department of Commerce. NIST – National Institute of Standards and Technology.

[FER99] FERRAIOLO, David F.; BARKLEY, John F., KUHN, D. Richard. **A Role-Based Access Control Model and Reference Implementation within a Corporate Intranet.** 1999. NIST – National Institute of Standards and Technology.

[FIO99] FIORESE, Maurício. **Uma Solução na Autenticação de Usuários para Ensino à Distância.** http://www.inf.ufrgs.br/pos/SemanaAcademica/Semana99/fiorese/fiorese.html.

[GAM95] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software.** 1994. Reading: Addison-Wesley.

[GUS96] GUSTAFSSON, Mats; SHAHMEHRI, Nahid. **A Role Description Framework and its Applications to Role-Based Access Control.** Maio 1996. Laboratory for Intelligent Information Systems. Department of Computer and Information Science. Linkping University. Sweden.

[JAN98] JANSEN, W. A. **A Revised Model for Role-Based Access Control.**1998. NIST 6192.

[JOH91] JOHNSON, Ralph E.; RUSSO, Vincent F. **Reusing Object-Oriented Designs**.Maio 1991.Department of Computer Science. University of Illinois. EUA. ftp://st.cs.uiuc.edu/pub/papers/frameworks/reusable-oo-design.ps

[JOH97] JOHNSON, Ralph E. **Components, Frameworks, Patterns.** Fevereiro 1997. Disponível em: ftp://st.cs.uiuc.edu/pub/papers/frameworks/framework97.ps

[KEL00] KELATH, Jayasankar. **Role Based Access Control.** Dept. of Computer Science, University of Idaho, Moscow, ID.

[LAN95] LANDIN, Nicklas; NIKLASSON, Axel. **Development Of Object Oriented Frameworks.** 1995. Ericsson Software Technology.

[LAR00] LARMAN, Craig. **Utilizando UML e Padrões: Uma Introdução à Análise e Ao Projeto Orientado a Objetos.** 2000. Porto Alegre. Editora Bookman.

[LEE99] LEE, Sangdon; CHOI, Hansuk; YANG, Youngjong; LEE, Sangduck. **Storage and Management of Object-oriented Frameworks.** 1999. Department of Computer Science. Mokpo National University. Korea. Systems, Man, and Cybernetics, 1999. IEEE SMC '99 Conference Proceedings. 1999 IEEE International Conference on , Volume: 5 , 1999 Pg. 762 -767 vol.5.

[MEN01] MENS, Tom; TOURWÉ, Tom. **A Declarative Evolution Framework for Object-Oriented Design Patterns.** 2001. Programming Technology Lab. Vrije Universiteit Brussel. Belgium. Software Maintenance, 2001. Proceedings. IEEE International Conference on , 2001 .Pg 570 –579.

[MON97] MONROE, Robert T.; KOMPANEK, Andrew; MELTON, Ralph; GARLAN, David. **Architectural Styles, Design Patterns, and Objects.** Janeiro 1997. *IEEE Software,pp.43-22*.http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/ObjPatternsArch-ieee.html

[NIBU98] **An Introduction to Role-Based Access Control.** NIST CSL Bulletin on RBAC.

[OBE01] OBELHEIRO, Rafael Rodrigues. **Modelos de Segurança Baseados em Papéis para Sistemas de Larga Escala: A Proposta RBAC-JACOWEB.** 2001. Curso de Pós-Graduação em Engenharia Elétrica. Universidade Federal de Santa Catarina.

[OBE02] OBELHEIRO, Rafael Rodrigues; FRAGA, Joni S. **Role-Based Access Control for CORBA Distributed Object Systems.** 2002. Departamento de Automação e Sistemas. Universidade Federal de Santa Catarina. IEEE - Object-Oriented Real-Time Dependable Systems, 2002. (WORDS 2002). Proceedings of the Seventh International Workshop on , 2002 Pg: 53 –60.

[OH00] OH, Sejong; PARK, Seog. **Enterprise Model as a Basis of Administration on Role-Based Access Control.** Dept. of Computer Science, Sogang University , Seoul, Korea. IEEE – Cooperative Database Systems for Advanced Applications, 2001, CODAS 2001. The Proceedings of the Third International Symposium on 2000. Pg. 150-158.

[PAP00] PAPA, M.; BREMER, O.; CHANDIA, R.; HALE, J.; SHENOI, S. **Extending Java for Package Based Access Control.** Center for Information Security. Department of Computer Science, Keplinger Hall. University of Tulsa, Oklahoma. Computer Security Applications, 2000. ACSAC'00. 16th Annual Conference 2000. Pg. 67-76.

[RAM94] RAMOS, Alexandre Moraes. **Interface de Controle de Acesso Para o Modelo de Gerenciamento OSI.** 1994. Universidade Federal de Santa Catarina. Dissertação submetida ao Curso de Pós-Graduação em Ciência da Computação.

[SAN00] SANDHU, Ravi; FERRAIOLO, David F.; JUN, Richard. **The NIST Model for Role-Based Access Control: Towards a Unified Standard.** 2000. Proceedings of the 5th ACM Workshop on Role-Based Access Control (RABC´2000). Berlin. Alemanha.

[SAN94] SANDHU, Ravi; SAMARATI, Pierangela. **Access Control: Principle and Practice.** IEEE Communications Magazine, Volume 32 Issue: 9 Sept. 1994. Pg. 40-48.

[SAN96] SANDHU, Ravi S; COYNE, Edward; FEINSTEIN, Hal; YOUMAN, Charles. **Role Based Access Control Models.** Fevreiro1996. IEEE Computer, Volume 29,Número2, pág. 38-47.

[SAN98] SANDHU, Ravi. **Role-Based Access Control.** 1998. Advances in Computers. Volume 46. Academic Press.

[SAN98a] SANDHU, Ravi; MUNAWER, Qamar. **The RBAC97 Model for Role-Based Administration of Role Hierarchies.** Laboratory for Information Security Technology (LIST) and ISE Department. George Mason University. IEEE - Computer Security Applications Conference, 1998. Proceedings. 14th Annual, 1998. Pg. 39-49.

[SAN99] SANDHU, Ravi; MUNAWER, Qamar. **The ARBAC99 Model for Administration Roles.** Laboratory for Information Security Technology (LIST) and ISE Department. George Mason University. IEEE - Computer Security Applications Conference, 1999. (ACSAC ´99) Proceedings. 15th Annual, 1999. Pg. 229-238.

[SHI00] SHIN, Michael E.; AHN, Gail-Joon. **UML-Based Representation of Role-Based Access Control.** ISE Department. George Mason University, USA. Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000. (WET ICE 2000). Proceedings. IEEE 9th International Workshops on 2000. Pg. 195-200.

[SHI01] SHIM, Won Bo; PARK, Seog. **Implementing Web Access Control System for the Multiple Web Servers in the Same Domain Using RBAC Concept.** Dept. of Computer Science, Sogang University. IEEE - Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings: Eighth International Conference on 2001. Pg. 768 –773.

[SIL00] SILVA, Ricardo Pereira e. **Suporte ao Desenvolvimento e Uso de Frameworks e Componentes.** 2000. Tese de Doutorado. Programa de Pós-Graduação em Computação. Universidade Federal do Rio Grande do Sul.

[SOU02] SOUZA, Adriano de. **Filtragem em Dados de Fluxo Contínuo: Uma Abordagem Voltada Às Prestadoras de Serviço Telefônico Fixo Comutado**. Setembro 2002. Dissertação de Mestrado. Programa de Pós-Graduação em Ciência da Computação. Universidade Federal de Santa Catarina.

[TAL94] TALIGENT. **Building object-oriented frameworks.** IBM. Taligent. 1994. White paper. http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/buildingoo.pdf

[TAL95] TALIGENT. **Leveraging object-oriented frameworks.** IBM. Taligent. 1995. White paper. http://lhcb-comp.web.cern.ch/lhcb-comp/Components/postscript/leveragingoo.pdf

[TSA99] TSAI, Wei-Tek; YONGZHONG, Tu; SHAO, Weiguang; EBNER, Ezra. **Testing Extensible Design Patterns in Object-Oriented Frameworks through Scenario Templates.** 1999. Software Engineering Laboratory. Department of Computer

Science and Engineering. University of Minnesota. IEEE - Computer Software and Applications Conference, 1999. COMPSAC '99. Proceedings. The Twenty-Third Annual International , 1999. Pg: 166 –171.

[WES00] WESTPHALL, Carla Merckle. **Um Esquema de Autorização para a Segurança em Sistemas Distribuídos de Larga Escala.** 2000. Curso de Pós-Graduação em Engenharia Elétrica. Universidade Federal de Santa Catarina.

[YIA96] YIALELIS, Nicholas; LUPU, Emil; SLOMAN, Morris. **Role-Based Security for Distributed Object Systems.** Junho 1996. IEEE Fifth Workshops on Enabling Technology: Infrastructure for Collaborative Enterprises (WET ICE ’96). Stanford, California, USA.

[ZHA01] ZHANG, Chang. N.; YANG, Cungang. **An Object-Oriented RBAC Model for Distributed System.** Department of Computer Science, University of Regina, TRLabs, Regina, Saskatchewan. Software Architecture, 2001. Proceedings. Working IEEE/IFIP Conference on , 2001 Pg. 24 –32.

[ZHA01a] ZHANG, Chang. N.; YANG, Cungang. **Specification and Enforcement of Object-Oriented RBAC Model.** Department of Computer Science, University of Regina, TRLabs, Regina, Saskatchewan. IEEE - Electrical and Computer Engineering, 2001. Canadian Conference on , Volume: 1 , 2001. Pg. 301 -305 vol.1