UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
TRABALHO DE GRADUAÇÃO EM SEGURANÇA DE SISTEMAS DE SOFTWARE

Um framework de controle de acesso de aplicações web baseados no Acegi

Recife, Janeiro de 2008

Autor: Pablo de Araújo Borges – pab@cin.ufpe.br
Orientador: Carlos André Guimarães Ferraz – cagf@cin.ufpe.br

Pablo de Araújo Borges

Um framework de controle de acesso de aplicações baseado no Acegi

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: *Carlos André Guimarães Ferraz*

Recife

2008

Assinaturas

Este Trabalho de Graduação é resultado dos esforços do aluno Pablo de Araújo Borges, sob a orientação do professor Carlos André Guimarães Ferraz, na participação do projeto *Um framework de controle de acesso de aplicações baseados no Acegi*, conduzido pelo Centro de Informática da Universidade Federal de Pernambuco. Todos abaixo estão de acordo com o conteúdo deste documento e os resultados deste Trabalho de Graduação.

_____

Pablo de Araújo Borges

_____

Carlos André Guimarães Ferraz

À minha família e amigos.

Agradecimentos

Agradeço em primeiro lugar à minha família, principalmente minha mãe, Ana Lúcia, que sempre me incentivou do início ao fim desta jornada. Agradeço também aos meus amigos que tornaram estes anos menos estressantes.

Ao meu orientador Carlos Ferraz, por estar sempre presente tirando dúvidas e me ajudando a concluir este trabalho.

Agradeço também aos amigos da faculdade que participaram dos projetos, viradas de noite, etc. Lucindo, René, Thierry, Assis e ao Portuga (Francisco).

Ao meu chefe, Antônio Carlos, por me conceder horários flexíveis para que eu pudesse trabalhar e não me prejudicar tanto na faculdade.

Gostaria de agradecer a todos os que não foram citados aqui mas que participaram deste processo que se encerra este ano.

A todos vocês, meu muito obrigado!

Resumo

O *Acegi Security* é um framework de código aberto escrito em Java que provê autenticação, autorização, criptografia e outras funcionalidades de segurança para aplicações web baseadas em J2EE. A arquitetura padrão de autenticação e autorização de usuários do *Acegi Security* implementa o modelo baseado em papéis e depende basicamente de duas entidades: UserDetails, que representa o usuário e GrantedAuthority, que representa as permissões de um usuário. Quando um usuário logado no sistema tenta acessar um conteúdo protegido, o Acegi Security verifica se ele tem permissão para tal. Esta arquitetura é suficiente na maioria dos casos, principalmente em aplicações rodando na Internet e sendo acessada por grupos de usuários organizados hierarquicamente, por exemplo, administradores, moderadores, usuários colaboradores e usuários comuns, mas torna-se incompleta em casos onde as operações dependem de permissões em departamentos para serem executadas.

Palavras-chave: Segurança, Controle de Acesso, Programação Orientada a Aspectos, Acegi Security, Spring Framework.

Sumário

Lista de Figuras

1. Introdução

1.1 Contexto

Não restam dúvidas que um dos principais requisitos do desenvolvimento de aplicações é a segurança. Existem diversos modelos que auxiliam o desenvolvedor a modelar o sistema respeitando as regras de segurança. Um deles, o modelo baseado em papéis é considerado o mais completo quando se trata de segurança aplicações. Neste modelo os usuários não acessam as operações diretamente. Para isso ele precisa possuir um papel que tenha acesso a essa operação.

O *Acegi Security* é um framework de código aberto escrito em Java que provê autenticação, autorização, criptografia e outras funcionalidades de segurança para aplicações web baseadas em J2EE. A arquitetura padrão de autenticação e autorização de usuários do *Acegi Security* implementa o modelo baseado em papéis e depende basicamente de duas entidades: UserDetails, que representa o usuário e GrantedAuthority, que representa as permissões de um usuário. Quando um usuário logado no sistema tenta acessar um conteúdo protegido, o Acegi Security verifica se ele tem permissão para tal. Esta arquitetura é suficiente na maioria dos casos, principalmente em aplicações rodando na Internet e sendo acessada por grupos de usuários organizados hierarquicamente, por exemplo, administradores, moderadores, usuários colaboradores e usuários comuns.

Porém, em aplicações desenvolvidas para a Intranet de uma empresa ou órgão os perfis do usuário costumam ser mais complexos. Um mesmo usuário de uma empresa pode ser Diretor do Departamento de Recursos Humanos e Supervisor do Departamento de Finanças além de ter permissão de usuário no Departamento de Informática. Digamos que nessa empresa os diretores podem remover documentos de seus departamentos. Ele teria, portanto uma permissão que daria para ele o direito de remover documentos no departamento em que ele é diretor. Para isso seria necessário checar se o perfil de diretor dele está relacionado com o departamento onde ele deseja remover um arquivo. Porém, no *Acegi Security* não existe nenhuma entidade relacionada com o local onde a operação está sendo executada.

Felizmente, por se tratar de um framework muito extensível, o Acegi Security permite que novas características sejam adicionadas às funcionalidades padrão do sistema.

1.2 Objetivos

O objetivo deste trabalho é aproveitar a extensibilidade do Acegi Security e propor um framework simples e reutilizável para a criação de controles de acesso baseados no Acegi Security. A arquitetura deste framework deverá resolver o problema da dependência de departamento na autorização de uma operação. Também deve permitir operações que não dependam de nenhum departamento, ou seja, dependem apenas do perfil do usuário.

2. Controle de Acesso

2.1 Definição

Quando se trata de sistemas computacionais, o controle de acesso pode ser definido como um método ou um conjunto de métodos criados com o objetivo de restringir o uso de determinados recursos deste sistema apenas por certos usuários ou grupos de usuários. De uma forma mais completa, podemos dizer que o controle de acesso é a capacidade de permitir ou negar o uso de alguma coisa por alguém.

Além de usuários humanos, é comum conceder acesso a outros softwares ou sistemas. Em função disso, no decorrer deste capítulo, será usada a palavra sujeito para definir usuários e sistemas automatizados capazes de realizar alguma operação no sistema.

Na segurança de sistemas computacionais, o controle de acesso engloba a autenticação e identificação, autorização e auditoria. A autenticação e identificação são responsáveis por determinar quem está apto a acessar o sistema, a autorização protege o sistema permitindo ou negando o acesso de acordo com as permissões, e a auditoria é o processo responsável por manter um registro do que foi feito no sistema.

Para compreender como funciona um controle de acesso, é necessário entender também os seus conceitos básicos. Nesta seção, serão detalhados os principais desses conceitos, de forma a fornecer uma visão geral de um módulo de controle de acesso.

2.1.1 Identificação e Autenticação

Identificação e Autenticação é o processo de verificar se a entidade que tenta acessar é quem alega ser, ou seja, a entidade possui uma identidade ligada ao sistema. Este processo pressupõe que exista um bloqueio inicial ao recurso e que seja provida uma forma de comprovar a identidade com o uso de um autenticador para fazer a validação. É necessário, também, que a identidade seja única dentro de um domínio.

Comumente, a autenticação é feita baseando-se em pelo menos um de três fatores: algo que você sabe; algo que você tem e algo que você é. A seguir esses três fatores serão detalhados.

2.1.1.1 Autenticação baseada em algo que você sabe

É a forma mais comum de autenticação. Para autenticar o usuário precisa saber alguma informação que, a princípio, apenas ele saberia. Autenticação por senhas é o principal exemplo de autenticação baseada em algo que você sabe. Apesar de

largamente usando, este método é extremamente vulnerável. Uma vez que se descobre a senha de outra pessoa, é possível autenticar como se fosse essa pessoa. A recomendação é que sejam escolhidas senhas difíceis de serem adivinhadas, porém essas senhas também são difíceis de memorizar, e anotar em algum lugar gera uma vulnerabilidade do mesmo modo.

2.1.1.2 Autenticação baseada em algo que você possui

Também é um método bastante comum. Neste modelo o usuário precisa estar de posse de algum objeto ou arquivo para concluir a autenticação. Um ótimo exemplo são os cartões que funcionam como chaves e destravam fechaduras. A vulnerabilidade desse sistema ocorre caso o usuário perca o objeto, afinal qualquer pessoa de posse será autenticada. Normalmente este método é combinado com o anterior de forma a aumentar a segurança. É o caso dos cartões de banco, onde é necessário, além de estar de posse do cartão, saber a senha.

2.1.1.3 Autenticação baseada em algo que você é

É uma forma ainda pouco usada em comparação às outras. Consiste em apresentar alguma propriedade integrante do usuário. No caso da biometria a autenticação é feita usando características como impressão digital, padrão de voz ou padrão de íris. A vulnerabilidade deste método é que em alguns casos é possível remover a característica do usuário para que outra pessoa possa autenticar. O ideal é que este tipo de autenticação seja usado em conjunto com outro. Por exemplo, a autenticação depende da apresentação da impressão digital em conjunto com uma senha.

2.1.2 Autorização

Depois de autenticado e identificado, o usuário ganha acesso ao sistema. Mas não necessariamente a qualquer parte, apenas ao que ele está autorizado a acessar. O processo de autenticação consiste em conceder acesso aos recursos apenas por quem tem permissão para tal. A autenticação é o processo responsável por proteger recursos do sistema, permitindo que esses recursos sejam usados apenas por quem possui autoridade para usá-los.

2.1.3 Auditoria

A auditoria é o processo do sistema que mantém um registro das atividades realizadas e quem as realizou. Apesar de não ser fundamental para o funcionamento de um controle de acesso, o processo de auditoria é de grande importância, pois a partir dele é possível recuperar informações de uso do sistema e detectar possíveis falhas a partir destas informações.

2.2 Modelos de Controle de Acesso

Os modelos de controle de acesso definem regras mais detalhadas de segurança e do comportamento do sistema, atuando sempre segundo regras de uma política de segurança estabelecida. Os modelos dividem-se em três tipos básicos [SAN96]:

- Baseados em identidade ou discricionários (*discretionary*);
- Baseados em regras ou obrigatórios (*mandatory*);
- Baseados em papéis (*roles*);

2.2.1 Modelos discricionários e mandatórios

Os controles de acesso discricionários são baseados no modelo matriz de acesso proposta por Lampson [LAM71]. Neste modelo, o estado de proteção do sistema é representado através de uma matriz, onde as linhas correspondem aos sujeitos e as colunas correspondem aos objetos do sistema. Esta matriz relaciona-se através dos objetos, que podem ser definidos como sendo os recursos do sistema, dos sujeitos, que são as entidades ativas existentes no sistema e dos atributos de acesso, que são os direitos ou permissões de acesso (read, write) cabíveis ao sistema.

Neste modelo, um usuário é o dono de um objeto e pode passar uma ou mais de suas permissões para outros usuários. É o modelo usado na maioria dos sistemas operacionais comerciais para o gerenciamento de permissões de arquivos.

Nos controles de acesso obrigatórios, por sua vez, baseiam-se em uma administração centralizada de segurança, a qual dita regras incontornáveis de acesso à informação. As mesmas definem regras e estruturas válidas no âmbito de um sistema, normalmente, especificando algum tipo de política multinível (multilevel policy). As políticas multiníveis estão baseadas em algum tipo de classificação ao qual estão submetidos os acessos dos sujeitos aos objetos.

2.2.2 Modelos Baseados em Papéis (Role-Based Access Control - RBAC)

Os modelos Baseados em Papéis (RBAC) têm como objetivo intermediar o acesso dos usuários à informação com base nas atividades que são por eles desenvolvidas no sistema. A idéia central é que o usuário desempenhe diferentes papéis (roles) em um sistema. Um papel pode ser definido como um conjunto de atividades e responsabilidades associadas a um determinado cargo ou função em uma organização. Assim, no RBAC, as permissões são conferidas aos papéis e os usuários são autorizados a exercer papéis.

Em geral, o RBAC trabalha com o princípio do mínimo privilégio, um usuário ativa apenas o subconjunto de papéis que precisa para executar uma operação, esta ativação pode ou não estar sujeita a restrições.
Para os sistemas que implementam RBAC a identidade no sistema é o papel, uma vez que estes encapsulam as políticas na forma de permissões. Tem como base que os direitos de acesso sejam atribuídos a papéis e não a usuários, assim como acontece no DAC, já que os usuários obtêm estes direitos em virtude de terem papéis a si atribuídos. Por ser independente das políticas, o RBAC é facilmente ajustável a mudanças no ambiente e deve ser largamente utilizado, porque não é tão flexível quando o DAC e nem tão rígido quanto o MAC. Por mais que os modelos RBAC ainda estejam em desenvolvimento, existem vários relatos de implementações do mesmo, um deste pode ser encontrado em [GLE99].

2.3 Acegi Security

2.3.1 Visão Geral

Acegi Securiy provê serviços de segurança para aplicações baseadas em J2EE [ACE06]. É baseado no Spring Framework, que será detalhado no próximo capítulo.

No nível da autenticação, o *Acegi Security* suporta uma grande quantidade de modelos de autenticação. Além desses, o *Acegi* ainda provê um conjunto próprio de funcionalidades de autenticação. O *Acegi* suporta autenticações com qualquer uma destas tecnologias:

- HTTP BASIC (padrão IEFT RFC)
- HTTP Digest (padrão IEFT RFC)
- HTTP X.509 (padrão IEFT RFC)
- LDAP
- Formulários simples
- Computer Associates Siteminder
- JA-SIG Central Authentication Service (CAS)
- Autenticação via Remote Method Invocation (RMI) e HttpInvoker (protocolo do *Spring*)
- Autenticação via "lembrar-me"
- Autenticação como anônimo
- Autenticação Run-as (para executar como outro usuário)
- Java Authentication and Authorization Service (JAAS)

- Integração com o JBoss, Jetty, Resin and Tomcat
- Definido pelo usuário

A classes mais importantes do Acegi Security são:

- *SecurityContextHolder*, provê os tipos de acesso ao *SecurityContext*.
- *SecurityContext*, mantém um objeto do tipo *Authentication* com informações específicas do usuário autenticado.
- *HttpSessionContextIntegrationFilter*, guarda o *SecurityContext* na sessão entre requisições web.
- *Authentication*, representa as informações do usuário.
- *GrantedAuthority*, reflete as permissões do usuário.
- *UserDetails*, provê informação necessária para contruir um objeto do tipo *Authentication* a partir dos *DAO*s da aplicação.
- *UserDetailsService*, usado pra criar uma instância de *UserDetails*.

3. Programação orientada a aspectos

3.1 Definição

Programação Orientada a Aspectos (*Aspect-Oriented Programming* – AOP) é um paradigma de programação que tem a finalidade de ajudar os programadores na separação de interesses (*separation of concerns*), principalmente os chamados interesses transversais ou ortogonais (*cross-cutting concerns*), como um avanço na modularização.

A Programação Orientada a Aspectos complementa a Programação Orientada a Objetos (*Object-Oriented Programming* - OOP) provendo outro meio de pensar na estrutura do programa. A chave da modularização na OOP é a classe, enquanto que na AOP a chave é o aspecto. Aspectos permitem a modularização de interesses como gerenciamento de transações que estão presentes em vários tipos e objetos. [SPR07]

3.2 Motivação

Como visto no capítulo anterior, o Acegi Framework usa conceitos de programação orientada a aspectos para atingir os objetivos de autorização. O foco deste trabalho é desenvolver um módulo de autorização que permita ao programador, ao custo de pouca configuração, permitir ou negar o acesso de um usuário a uma transação. Para atingir este objetivo será usada a API de Programação Orientada a Aspectos do Spring Framework.

Nas próximas seções deste capítulo, serão detalhados os conceitos básicos da AOP e o funcionamento do Spring Framework. No próximo capítulo será mostrado onde cada um desses conceitos foi usado na construção da arquitetura e da implementação do controle de acesso.

3.3 Conceitos Básicos

A Programação Orientada a Aspectos deve ser entendida como um complemento à Programação Orientada a Objetos, e nunca como uma substituição. O mais comum é desenvolver o sistema usando OOP sempre que for possível, e em casos onde é necessário mais desacoplamento entre o modelo da aplicação e a infra-estrutura do que a OOP é capaz, usar a AOP.

O desacoplamento entre as funcionalidades transversais é conseguido acrescentando aspectos, que são comportamentos que afetam diferentes classes, não precisando para isso estar conectados diretamente a nenhuma delas.

Na Programação Orientada a Aspectos existem quatro elementos básicos e indispensáveis [KIC01]:

- **Ponto de junção (*Join Point*)**: Responsável por definir os pontos de ligação onde será acrescentado o comportamento a ser executado. Essas ligações são pontos bem definidos do programa tais como: construtores de classes, métodos, chamadas a métodos, exceções, etc.
- **Conjunto (de pontos) de junção**: Agrupa um conjunto de pontos de junção baseados em um critério de operadores lógicos.
- **Comportamento transversal**: Responsável por acrescentar o comportamento a ser executado quando se intercepta um determinado ponto do programa definido no conjunto de pontos de junção.
- **Aspecto**: Unidade básica da programação orientada a aspectos que contém as três construções básicas já mencionadas. Tem o mesmo comportamento de uma classe e é encarregada de encapsular as funcionalidades transversais dispersas no sistema.

Estes são os conceitos chave para entender a AOP, abaixo são apresentados outros termos importantes para o desenvolvimento orientado a aspectos [SPR07], alguns deles serão usados no decorrer deste trabalho:

- **Adendo (*Advice*)**: Ação tomada por um aspecto em um ponto de junção específico. Existem diferentes tipos de adendos que serão detalhados mais adiante;
- **Objeto Alvo (*Target Object/Advised Object*)**: Objeto que recebe adendos de um ou mais aspectos.
- ***AOP Proxy***: Objeto que atua como intermediário implementando os contratos dos aspectos, como a execução de métodos com adendos.
- **Combinador de Aspectos (*Aspect Weaving*)**: liga aspectos e outras entidades para criar um objeto alvo. Esta combinação pode ser feita em tempo de compilação, carregamento ou durante a execução. Spring AOP, como será mostrado a seguir, faz a combinação em tempo de execução.

Os tipos possíveis de adendo são [SPR07]:

- **Adendo antes (*Before advice*)**: Adendo que executa antes de um ponto de junção. Não possui a capacidade de prevenir o fluxo de execução que ocorre após o ponto de junção, a não ser que jogue uma exceção.

- **Adendo após retorno (*After returning advice*)**: Adendo que é executado quando um ponto de junção completa normalmente: por exemplo, se um método retornar sem jogar uma exceção.
- **Adendo após exceção (*After throwing advice*)**: Adendo que é executado quando um método lança uma exceção.
- **Adendo após (*After advice*)**: Adendo que é executado independente da forma de saída do ponto de junção (retorno normal ou excepcional).
- **Adendo ao redor (*Around advice*)**: Adendo que cerca um ponto de junção como uma invocação de um método. É o tipo mais poderoso de adendo. Adendos ao redor podem executar comportamentos antes e após a invocação de um método. Também é capaz de escolher se deve seguir a execução normal do ponto de junção, ou se deve cortar caminho e retornar seu próprio valor de retorno ou lançar uma exceção.

3.4 Spring Framework

3.4.1 Visão Geral

O *Spring Framework* é um projeto de código aberto que visa simplificar o desenvolvimento de aplicações J2EE. O *Spring* foi criado por Rod Johnson e teve seu código aberto em 2003 [SPR07]. O *Spring Framework* contém diversas funcionalidades bem organizadas em módulos, como é possível ver na Figura 1. Como não é o intuito desse trabalho investigar a fundo todos os conceitos do *Spring*, não entraremos em detalhes nos outros módulos, apenas no módulo de orientação a aspectos. Mesmo assim, é importante destacar de forma resumida a função de cada pacote do *framework*:

- **Pacote *Core*:** é a base do *framework* e provê a inversão de controle (Inversion of Control: IoC) e Injeção de Dependências.
- **Pacote Context**: construído nos alicerces do pacote *Core*, provê meios de acessar objetos. Oferece suporte à internacionalização, propagação de eventos, carregamento de recursos e criação transparente de contextos.
- **Pacote DAO**: provê uma camada de abstração ao JDBC. Também provê meios de gerenciar transações de forma programática ou declarativa.
- **Pacote ORM**: provê camadas de integração para APIs de mapeamento objeto-relacional populares, incluindo JPA, JDO, Hibernate e iBatis.
- **Pacote AOP**: provê uma implementação da programação orientada a aspectos, permitindo que se defina, por exemplo, adendos e pontos de junção

para desacoplar funcionalidades que deveriam ficar separadas da regra de negócio.

- **Pacote WEB**: provê funcionalidades básicas para desenvolvimento web, como *upload* de arquivos e a inicialização do container IoC usando *servlets*.
- **Pacote MVC**: provê separação entre código do domínio e formulários web. E ainda permite que se use todas as outras funcionalidades do Spring Framework,

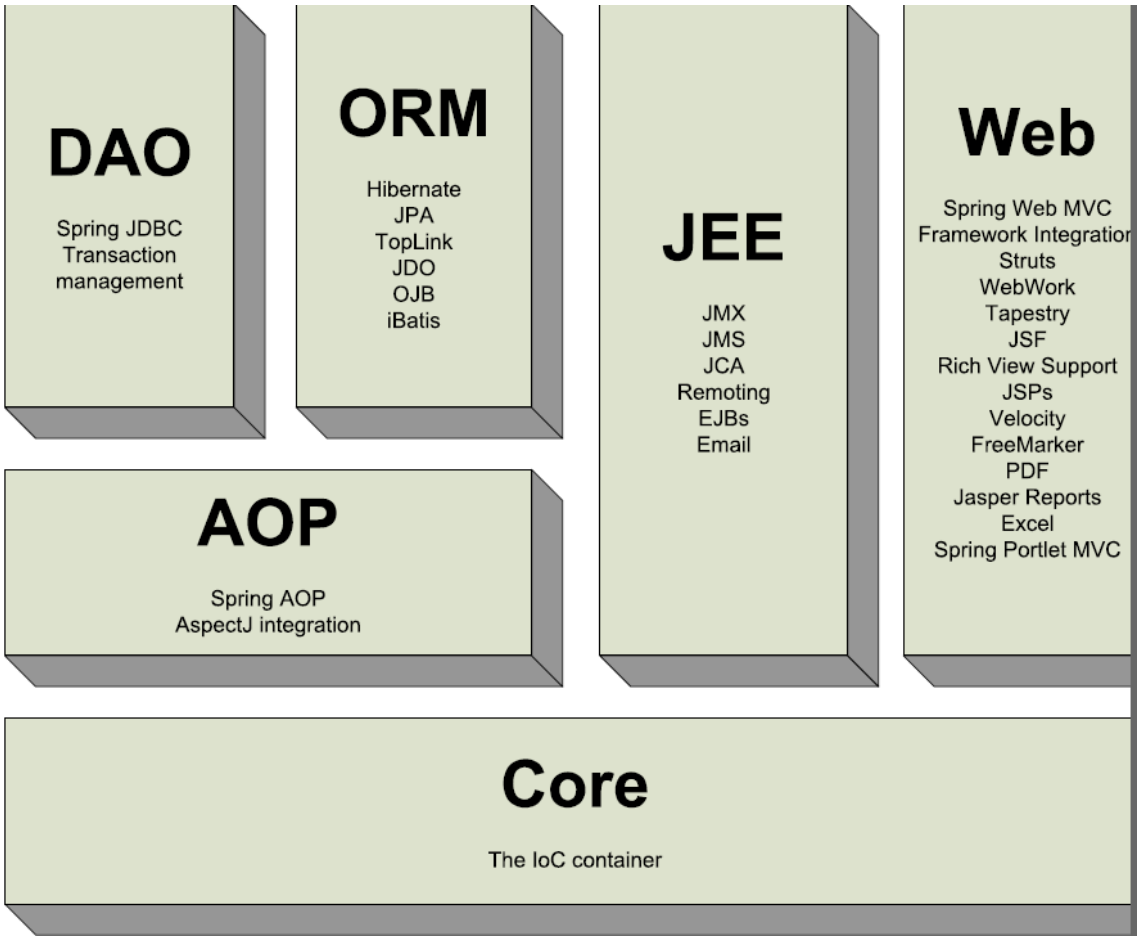

Figura 1 Módulos do Spring Framework

3.4.2 Spring AOP

Após a visão geral, entraremos em mais detalhes no módulo que, será explorado neste trabalho, o módulo AOP. Grande parte do suporte do Spring AOP é baseado na API definida pela AOP Alliance que também é um projeto código aberto cujo objetivo é promover a adoção e interoperabilidade entre diferentes implementações AOP, definindo um conjunto comum de interfaces e componentes [WAL05].

No *Spring AOP* todo adendo é uma classe. Uma instância de um adendo pode ser dividida entre vários objetos. Também é possível criar uma nova para cada objeto alvo. Isto se chama adendo por classe (*per-class*) e por instância (*per-instance*) [SPR07].

Adendo por classe é mais freqüente. É apropriado para um adendo genérico como aspectos de transação. Não depende do estado do objeto alvo. Apenas atuam sobre o método e os argumentos. Adendo por instância é apropriado quando se deseja adicionar um estado ao objeto alvo. É possível também usar ambos num mesmo objeto.

O *Spring* suporta os tipos de adendo comentados anteriormente neste capítulo. O principal deles, o adendo ao redor, é o que será usado neste trabalho, e será detalhado a seguir.

Spring implementa os padrões da interface da *AOP Alliance* para adendo ao redor usando interceptação de métodos.

Para usar interceptadores de métodos, a classe do adendo deve implementar a interface *MethodInterceptor*:

```
public interface MethodInterceptor extends Interceptor {
        Object invoke(MethodInvocation invocation) throws Throwable;
}
```

O argumento do método *invoke* que é da classe *MethodInvocation* expõe um método sendo invocado, o ponto de junção, o *proxy*, e os argumentos para o método. O método *invoke* deve retornar o resultado da invocação: o valor de retorno do ponto de junção.

4. Framework Proposto

4.1 Motivação

Apesar de o modelo padrão de autorização do Acegi Framework cumprir os requisitos para uma implementação RBAC, ainda é complicado, por limitação do próprio Controle de Acesso Baseado em Papéis, uma situação onde um Papel está associado a um local.

Por exemplo, vamos supor que o usuário Zidane possua o papel de Supervisor que o autoriza a realizar certas transações, como a marcação de reuniões com funcionários do seu setor, o setor de venda de material esportivo para jogadores de futebol. Outro usuário, Jordan, também possui o papel de Supervisor e, portanto, também tem direito a marcar reuniões, porém em um setor diferente de Zidane, o de venda de material para jogadores de basquete. Como garantir que Jordan não possa marcar reuniões com o pessoal de Zidane? A resposta mais óbvia seria dar a Zidane o papel de Supervisor do Setor de Futebol e a Jordan o papel de Supervisor do Setor de basquete. Isso resolveria o problema, bastaria criar as transações “Marcar Reuniões com o Setor de Futebol” e “Marcar Reuniões com o Setor de Basquete”. O leitor já deve ter percebido o problema deste modelo. Ele não respeita uma das principais regras da Engenharia de Software: Não se repita (*Don’t Repeat Yourself* – DRY). A falha é rapidamente percebida se pensarmos numa situação onde exista alguém no papel de Diretor do Departamento Esportivo, e que os diversos supervisores sejam chefiados por ele. Se em algum momento este diretor resolver que apenas ele pode marcar reuniões, todos os papéis de supervisor deverão ser alterados, o que pode levar a erros como se esquecer de remover o privilégio de algum destes papéis.

Está claro, que o papel e o local onde a transação será aplicada devem ser separados. Uma opção seria criar uma entidade para representar os locais onde o usuário pode exercer seus papéis. Sendo assim, o Zidane do exemplo anterior poderia ter o papel de supervisor, e fazendo parte do Setor de Futebol, então ele poderia executar as operações permitidas aos supervisores neste setor. Esta opção parece boa, mas ainda existe um problema. Como faríamos para colocar o Zidane, também, como vendedor do setor de basquete, sem que isso o tornasse um supervisor deste departamento?

A opção encontrada para este trabalho foi criar uma nova entidade chamada Perfil representando um conjunto de Transações possíveis. O Papel fica sendo então a ligação entre um perfil e o local onde este perfil é válido. E os usuários continuam ligados aos seus papéis como definido pelo modelo RBAC.

Para facilitar o entendimento, antes de mostrar os diagramas de classes e de relacionamento, vamos mostrar quais as entidades foram usadas no exemplo. Definições, códigos e outros detalhes de cada uma dessas entidades serão mostrados no decorrer do capítulo.

- **Transações**: Marcar reunião;
- **Perfis**: Vendedor, Supervisor, Diretor;
- **Unidades Gestoras (locais):** Departamento Esportivo, Setor de Futebol, Setor de Basquete;
- **Papéis:** Vendedor do Setor de Futebol, Vendedor do Setor de Basquete, Supervisor do Setor de Futebol, Supervisor do Setor de Basquete, Diretor do Departamento Esportivo;
- **Usuários**: Zidane, Jordan.

4.2 Arquitetura Proposta

Na seção anterior, conhecemos de forma superficial as entidades que serão usadas para atingir o objetivo de separar os conceitos de Papel e Local. Nesta seção veremos com maiores detalhes a arquitetura proposta.

A Figura 2 apresenta a arquitetura proposta para o controle de acesso.

A classe abstrata *Model* implementa métodos comuns a todas as classes do modelo do controle de acesso. Do mesmo modo a classe abstrata *HierarchicalModel* herda de *Model* e complementa adicionando métodos úteis para as entidades que possuem hierarquia.

A classe *Transaction* herda de *HierarchicalModel*. As operações do sistema devem ser instância de *Transaction*. As operações possuem um caráter hierárquico, ou seja, uma operação pode possuir “operações filhas”, permitindo assim um fácil agrupamento das operações por assunto. A classe *Profile* representa os perfis, conjuntos de operações. Um perfil pode possuir uma ou mais operações, e uma operação pode fazer parte de um ou mais perfis. A classe *ManagementUnit* representa as unidades gestoras, que podem ser quaisquer locais onde uma operação possa ocorrer, uma unidade gestora pode ser uma empresa ou organização e seus departamentos, secretarias, etc. Assim como as operações, as unidades gestoras também possuem um caráter hierárquico.

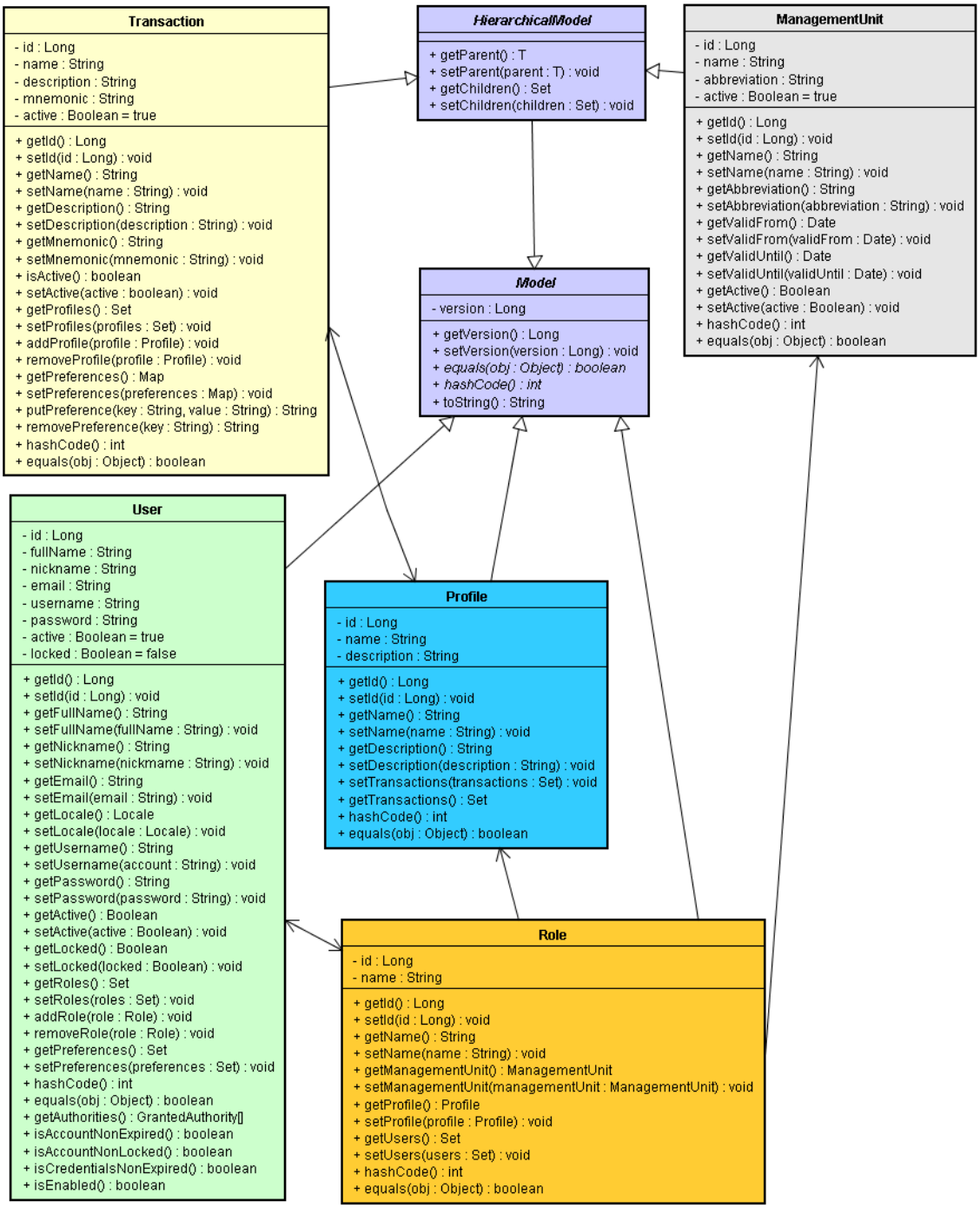

Figura 2 Diagrama de Classes do Modelo

A classe *User* representa os usuários do sistema. Por último a classe *Role* representa os papéis que agora não estão mais associadas diretamente às operações, mas sim aos perfis e às unidades gestoras ao mesmo tempo. Os usuários continuam tendo seus papéis no sistema, mas para executar uma operação é necessário que ele possua o perfil correto na unidade gestora em que ele pretende executar a operação.

A Figura 3 mostra os relacionamentos das cinco entidades prinicipais de forma simplificada.

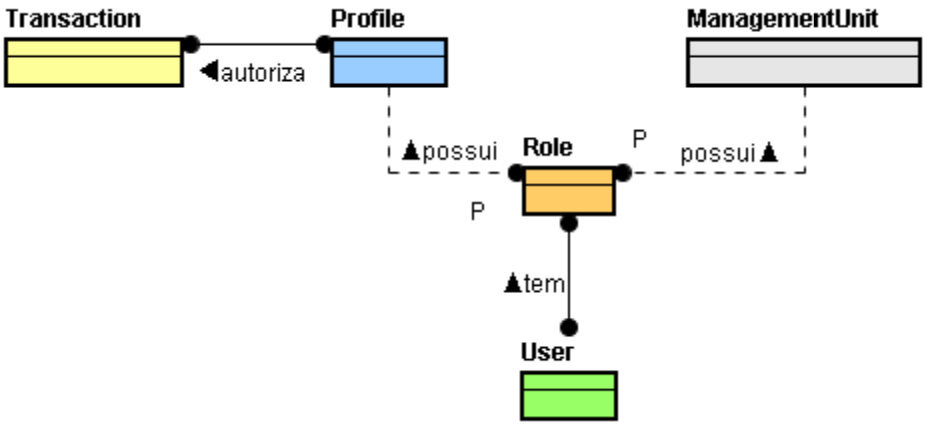

Figura 3 Diagrama Entidade-Relacionamento do Modelo

Além das classes do modelo, a criação deste framework exigiu também a inclusão de novas classes que usassem os conceitos de Programação Orientada a Aspectos e o módulo Spring AOP para autorizar os usuários a executar operações. A Figura 4 mostra as classes criadas com esse propósito.

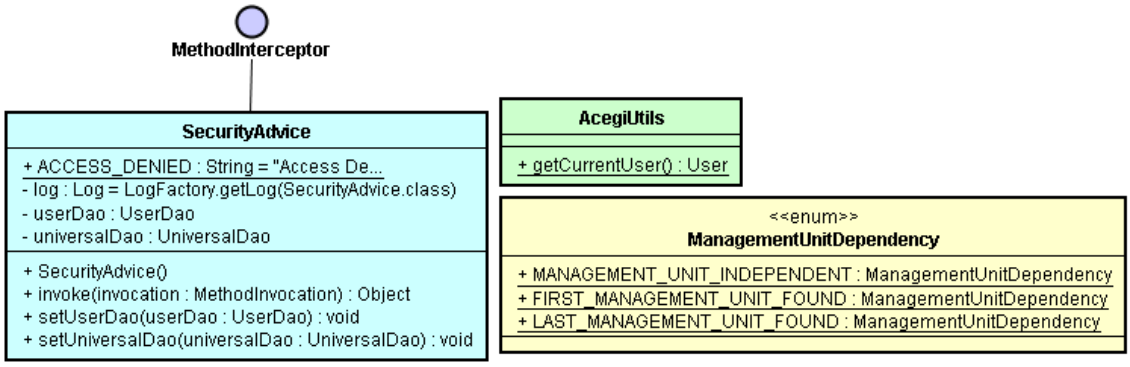

Figura 4 Diagrama de classes do pacote de segurança

A classe *SecurityAdvice*, é a espinha dorsal do framework. Ela representa um aspecto que é combinado às classes da regra de negócio do sistema. Quando um método de uma das classes da regra de negócio é invocado, o método *invoke* de *SecurityAdvice* é executado. Este método vai decidir se o usuário possui privilégios para acessar o método, que estará relacionado com uma operação do sistema.

Na verdade, o que acontece é que o *Spring Framework* substitui as classes do pacote de regras de negócio por um *AOP Proxy*, que por sua vez chama o método *invoke* de

SecurityAdvice, que decide se o método real da regra de negócio deve ou não ser invocado, dependendo do papel do usuário.

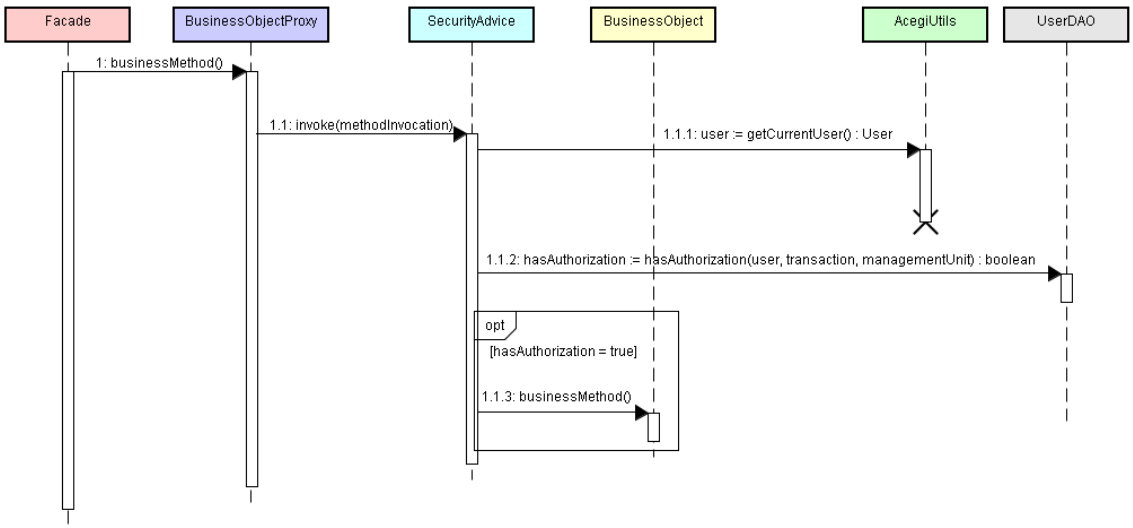

Figura 5 Diagrama de seqüência – Chamada de método da regra de negócio

A Figura 5 mostra o diagrama de seqüência de uma chamada genérica de um método da regra de negócio.

A classe *SecurityAdvice* faz uso de mais duas classes para tomar sua decisão sobre a autorização ou a negação de uma operação. *AcegiUtils* é uma classe que possui apenas um método estático *getCurrentUser* que usa a API do *Acegi Security* para descobrir quem é o usuário autenticado no sistema.

Tendo identificado o usuário, *SecurityAdvice* invoca um método de *UserDAO*, uma classe que se adéqua ao padrão *Data Access Object*. Para se adequar ao framework em questão, a classe *UserDAO* precisa implementar um método chamado *hasAuthorization*, este método deve checar se o usuário possui privilégios para executar a operação desejada numa determinada unidade gestora e retornar um *boolean* indicando se o usuário está ou não autorizado.

Para finalizar, *SecurityAdvice* invoca o método da regra de negócio caso o usuário possua os privilégios necessários, ou uma exceção caso contrário.

4.3 Implementação

Na seção anterior, foi apresentada a arquitetura do framework. Nesta seção serão apresentados trechos de código da implementação das classes mais importantes para o funcionamento do controle de acesso.

```
1      <aop:config>
2          <aop:advisor id="managerSecurity" advice-ref="securityAdvice"
3              pointcut="execution(* *..service.*Manager.*(..))" order="0" />
4      </aop:config>
5
6      <bean id="securityAdvice" class="br.ufpe.cin.pab.security.SecurityAdvice" />
```

Figura 6 Trecho da configuração do *SecurityAdvice*

A Figura 6 apresenta um trecho do código XML usado para ativar o suporte a aspectos e configurar o *SecurityAdvice*.

O atributo *pointcut* na linha 3 define quais serão os objetos alvo deste Adendo. No caso será qualquer método de todas as classes que possuam a terminação *Manager* e estejam dentro de um pacote chamado *service*.

Na linha 6, está sendo declarado o adendo *SecurityAdvice* para que seja iniciado junto com a aplicação usando o módulo de inversão de controle do *Spring Framework*.

```
1   /**
2    * A Secured <code>Annotation</code> to be used in <code>Manager</code> methods in order to
3    * limit user access.
4    *
5    * @author pab
6    * @version $Revision: 1.1 $
7    */
8   @Target(METHOD)
9   @Retention(RetentionPolicy.RUNTIME)
10  public @interface Secured {
11
12      /**
13     * The associated <code>Transaction</code> represented by its mnemonic.
14     * The presence of unrecognized mnemonic <i>will</i> result in an error.
15     */
16      String value();
17
18      /**
19     * The order of the <code>ManagementUnit</code> used to compare with user
20     * <code>Role</code>s to give or deny access.
21     */
22      ManagementUnitDependency managementUnit() default MANAGEMENT_UNIT_INDEPENDENT;
23
24  }
```

Figura 7 A *annotation Secured*

A Figura 7 mostra a implementação da classe *Secured*, uma *annotation* criada para facilitar o relacionamento dos métodos que precisam de autorização com as operações associadas. Na linha 22 está a declaração da propriedade *managementUnit*

que deve ser preenchida caso a operação dependa de alguma unidade gestora para ser realizada. As opções aceitas por esta propriedade são definidas pelo enum *ManagementUnitDependency* que possui as seguintes opções:

- **MANAGEMENT_UNIT_INDEPENDENT**: é a opção padrão, caso nenhuma outra seja indicada. Indica que a operação a ser realizada não depende de unidade gestora. Esta opção faz com que o framework funcione de modo similar ao definido pelo modelo RBAC.
- **FIRST_MANAGEMENT_UNIT_FOUND**: Indica que a operação a ser realizada depende de uma unidade gestora e deve ser usada a primeira *ManagementUnit* que aparecer na chamada do método da regra de negócio.
- **LAST_MANAGEMENT_UNIT_FOUND**: Semelhante a anterior, porém será usada a última *ManagementUnit* que aparecer na chamada do método da regra de negócio.

```
1  public Object invoke(MethodInvocation invocation) throws Throwable {
2     // Firstly, check if the method call needs authorization.
3     Secured secured = invocation.getMethod().getAnnotation(Secured.class);
4     if (secured == null) {
5        log.debug("Unsafe method call: [" + invocation.getMethod().toString() + "] proceeding with invocation.");
6        return invocation.proceed();
7     }
```

Figura 8 Trecho 1 do método *invoke* de *SecurityAdvice*

A Figura 8 mostra o primeiro trecho do código do método *invoke* na classe *SecurityAdvice*.

Na linha 1, a assinatura do método, vemos que existe um parâmetro do tipo *MethodInvocation*, que é fornecido pelo Spring AOP. De posse do *MethodInvocation*, podemos obter mais detalhes sobre o método que está sendo invocado.

Na linha 3, é verificada a existência de uma *annotation @Secured* no método. No caso de não existir essa *annotation* significa que o método é acessível para qualquer usuário e, portanto a invocação do método prossegue normalmente (linha 6).

A Figura 9 mostra o segundo trecho do código.

Na linha 2, o usuário autenticado no sistema é identificado. Na linha 3 obtêm-se a operação. A linha 8 verifica se é uma operação dependente de unidade gestora. Da linha 9 a 28 é feita a procura pela unidade gestora. As linhas 29 a 34 fazem uma verificação se a operação requer uma unidade gestora, mas a assinatura do método não possui nenhuma.

```
1  try {
2     User user = AcegiUtils.getCurrentUser();
3     Transaction transaction = (Transaction) universalDao.findByProperty(Transaction.class,
4           TransactionProperties.MNEMONIC, secured.value());
5     ManagementUnit managementUnit = null;
6
7     //Search for the ManagementUnit if necessary
8     if (secured.managementUnit() != MANAGEMENT_UNIT_INDEPENDENT) {
9        if (secured.managementUnit() == FIRST_MANAGEMENT_UNIT_FOUND) {
10          Object[] arguments = invocation.getArguments();
11          for (Object o : arguments) {
12             if (o instanceof ManagementUnit) {
13                managementUnit = (ManagementUnit) o;
14                break;
15             }
16          }
17
18       }
19       else if (secured.managementUnit() == LAST_MANAGEMENT_UNIT_FOUND) {
20          Object[] arguments = invocation.getArguments();
21          int total = arguments.length;
22          for (int i = total; i >= 0; i--) {
23             if (arguments[i] instanceof ManagementUnit) {
24                managementUnit = (ManagementUnit) arguments[i];
25                break;
26             }
27          }
28       }
29       if (managementUnit == null) {
30          throw new IllegalArgumentException("Parameter [" + secured.managementUnit() + "] passed to ["
31                + Secured.class.getName() + "] at [" + invocation.toString() + "], but no ["
32                + ManagementUnit.class.getName() + "] argument found. Use [" + MANAGEMENT_UNIT_INDEPENDENT
33                + "] parameter or declare the [" + ManagementUnit.class.getName() + "] argument");
34       }
35
36       if (userDao.hasAuthorization(user, transaction, managementUnit)) {
37          log.debug("[" + transaction.getMnemonic() + "] allowed to [" + user.getUsername() + "] at ["
38                + managementUnit.getAbbreviation() + "]");
39          return invocation.proceed();
40       }
41       log.debug("[" + transaction.getMnemonic() + "] denied to [" + user.getUsername() + "] at ["
42             + managementUnit.getAbbreviation() + "]");
43       throw new AccessDeniedException("[" + transaction.getMnemonic() + "] denied to [" + user.getUsername()
44             + "] at [" + managementUnit.getAbbreviation() + "]");
45
46    }
```

Figura 9 Trecho 2 do método *invoke* de *SecurityAdvice*

As linhas 36-38 verificam se o usuário tem permissão para executar a operação. A linha 39 libera a invocação do método.

A linha 43 lança uma exceção alertando que o usuário não têm permissão para executar a operação em questão.

A Figura 10 mostra o fim do código do método *invoke*. A linha 1 checa se o usuário possui permissão para executar a transação. Não é necessário verificar a unidade gestora, pois este trecho só será executado se a operação for independente de unidade gestora (vide linha 8 da Figura 3).

```
1  if (userDao.hasAuthorization(user, transaction)) {
2     log.debug("[" + transaction.getMnemonic() + "] allowed to [" + user.getUsername() + "]");
3     return invocation.proceed();
4  }
5  log.debug("[" + transaction.getMnemonic() + "] denied to [" + user.getUsername() + "]");
6  throw new AccessDeniedException("[" + transaction.getMnemonic() + "] denied to [" + user.getUsername() + "]");
```

Figura 10 Trecho 3 do método *invoke* de *SecurityAdvice*

5. Conclusão

Neste trabalho foi feita uma pesquisa sobre modelos de controle de acesso, e sobre a programação orientada a aspectos. De posse desse conhecimento, foi desenvolvido um framework de controle de acesso usando as técnicas de programação orientada a aspectos e baseando-se no *Acegi Security*.

O objetivo deste trabalho que era o desenvolvimento de um controle de acesso com suporte a verificação de departamentos foi atingido. O *framework* englobou algumas técnicas de AOP como adendos e pontos de junção.

A fácil extensibilidade do framework também foi alcançada, sendo necessárias poucas configurações e alterações para que se adapte a uma nova aplicação. A autorização dos métodos pode ser feita via *annotations*, com apenas uma linha de código é possível indicar qual operação aquele método realiza e em que unidade gestora.

O framework foi usado com sucesso, numa aplicação comercial da empresa Áxon Tecnologia. Lá ele foi extendido para se acomodar às necessidades do projeto, mas a base é a mesma apresentada neste trabalho.

5.1 Trabalhos Futuros

Como trabalhos futuros posso sugerir:

- Aumentar ainda mais a extensibilidade do projeto de forma a permitir que o desenvolvedor possa configurar tudo via xml, caso prefira.
- Acrescentar outros modelos de controle de acesso ao *framework*.
- Criar uma aplicação usando o framework e extendê-lo de acordo com a necessidade.

5.2 Considerações Finais

Com base no que foi estudado neste trabalho, pode-se concluir que a programação orientada a aspectos é uma forte aliada da segurança. O controle de acesso desenvolvido neste trabalho, dificilmente seria possível com essa flexibilidade e extensibilidade não fosse pelo uso da AOP.

O *Acegi Security* se mostrou um framework de segurança maduro e facilmente extensível. Ao lado do Spring Framework é possível fazer praticamente qualquer tipo de controle de acesso baseando-se nele.

6. Referências

[ACE06] ALEX, B. Acegi - Reference Documentation Version 1.0.0, 2006. 92p.

[GLE99] GLENN, F. *RBAC in UNIX Administration*, Proceedings of 4th ACM Workshop on Role-Based Access Control, Fairfax, VA, Oct. 1999, p. 95-101.

[KIC01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). Getting started with AspectJ. Commun. ACM, 44(10):59–65.

[LAM71] LAMPSON, B.W. *Protection*. Proc. 5th Princeton Conf. on Information Sciences and Systems, Princeton, p. 437, 1971.

[SAN96] SANDHU, R.; COYNE, E.J.; FEINSTEIN, H.L.; YOUMAN, C.E. *Role-Based Access Control Models*. IEEE Computer, Vol. 29, Nº. 2, Feb. 1996, p. 38-47.

[SPR07] JOHNSON, R; HOELLER, J; ARENDSEN, A; et al. The Spring Framework - Reference Documentation Version 2.5.1, 2007. 583p.

[WAL05] WALLS, C.; BREIDENBACH, R. Spring in Action. Greenwich: Manning, 2005.