MODELO DE ACESSO MULTINÍVEL EM SISTEMAS WEB UTILIZANDO PROGRAMAÇÃO ORIENTADA A ASPECTOS

RESUMO

A segurança dos dados e, principalmente, a integridade das transações, são objetos de muita preocupação para empresas que possuem sistemas on-line trabalhando direta ou indiretamente com seus ativos. O interesse pelo comércio eletrônico se difundiu em diversos tipos de mercado, devido ao baixo custo operacional e também ao seu alcance global, o que provoca uma necessidade de constantes aperfeiçoamentos em relação à segurança desses sistemas. Este artigo demonstra a construção de um sistema Web utilizando Programação Orientada a Aspectos (POA), baseado em um modelo multinível de acesso à chamada de métodos utilizando a linguagem AspectJ. Foi utilizada a IDE NetBeans como ambiente de desenvolvimento. Foi adotada uma sintaxe alternativa, conhecida como Annotations por meio de interfaces, que serviu para relacionar o método chamado ao perfil de usuário autorizado. Este modelo foi implementado em um sistema do Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco - IFPE, nomeado como Programa Monitoria. A Programação Orientada a Aspectos, utilizada neste sistema, contribuiu diretamente para a redução do espalhamento e entrelaçamento de códigos-fonte entre os módulos do sistema e os requisitos não funcionais do mesmo, diminuindo o acoplamento e consequentemente a complexidade do código-fonte. A Programação Orientada a Aspectos também promoveu a centralização da responsabilidade de segurança, que normalmente é atribuída a todos os desenvolvedores da equipe.

PALAVRAS-CHAVES: Acesso Multinível; AspectJ; Programação Orientada a Aspectos; Segurança da Informação.

MULTI-LEVEL MODEL ACCESS IN WEB SYSTEMS USING ASPECT-ORIENTED PROGRAMMING

ABSTRACT

Data security and integrity of the transactions, are important objects to the companies that have online systems working directly or indirectly with their assets. The interest in ecommerce has spread in many market types due to low operating costs and also to its global reach, causing a need for constant improvements regarding the safety of these systems. This paper demonstrates the construction of a Web system using Aspect Oriented Programming (AOP), based on a model called multilevel access methods using the AspectJ language. It was used NetBeans IDE as the development environment. It was adopted an alternative syntax, known as Annotations through interfaces, which served to connect the called method and the allowed user profile. This model was implemented in a system of the Federal Institute of Education, named as Programa Monitoria. The Aspect-Oriented Programming, used in this system, has contributed directly to the reduction of scattering and entanglement of source codes between the system modules and non-functional requirements, reducing the coupling and consequently the complexity of the source code. The Aspect-Oriented Programming also promoted the centralization of responsibility for security, which is usually assigned to all developers on the team.

KEYWORDS: AspectJ; Aspect-Oriented; Information Security; Multi-Level Access.

**Revista Brasileira de
Administração Científica,**
Aquidabã, v.3, n.2, Ago 2012.

ISSN 2179-684X

SEÇÃO: Artigos

TEMA: **Sistemas de Informação**

Anais do Simpósio Brasileiro de
Tecnologia da Informação (SBTI 2012)

DOI: 10.6008/ESS2179-684X.2012.002.0004

Maxwell Queiroz Francisco

Universidade de Pernambuco, Brasil

<http://lattes.cnpq.br/2765487386293198>
mqf@ecomp.poli.br

Felype Nery de Oliveira Vasconcelos

Instituto Federal de Educação, Ciência e
Tecnologia de Pernambuco, Brasil

<http://lattes.cnpq.br/4750001437165719>
felypenery@gmail.com

Richardson Bruno da Silva Andrade

Instituto Federal de Educação, Ciência e
Tecnologia de Pernambuco, Brasil

<http://lattes.cnpq.br/9647929519912452>
richardsonbruno@gmail.com

Aida Araújo Ferreira

Instituto Federal de Educação, Ciência e
Tecnologia de Pernambuco, Brasil

<http://lattes.cnpq.br/8515798754882166>
aidaaf@gmail.com

Recebido: 10/06/2012

Aprovado: 15/08/2012

Avaliado anonimamente em processo de pares cegas.

Referenciar assim:

FRANCISCO, M. Q.; VASCONCELOS, F. N.
O.; ANDRADE, R. B. S.; FERREIRA, A. A..
Modelo de acesso multinível em sistemas
web utilizando programação orientada a
aspectos. *Revista Brasileira de
Administração Científica, Aquidabã, v.3,
n.2, p.57-70, 2012.*

INTRODUÇÃO

Devido ao significativo crescimento tecnológico mundial e aos benefícios que eles proporcionam, atualmente é difícil encontrar uma empresa, e muito menos uma rede de negócios, que não possua um Sistema de Informação Computadorizado manipulando, de alguma forma os seus ativos. As informações trafegadas nos sistemas organizacionais assumem uma importância cada vez maior para estas instituições. Neste contexto, a exposição destas informações, seja por falta de políticas de segurança ou até mesmo por falta de diretrizes básicas de Segurança de Informação, podem maximizar ameaças e possíveis prejuízos para a empresa, visto que a maioria destes sistemas está conectada a redes de computadores locais e muitas vezes também na internet.

Para Marciano (2006), os conteúdos e continentes digitais estão expostos a vários tipos de ameaças, sejam elas físicas ou virtuais, que muitas vezes comprometem gravemente a segurança das informações e até a segurança das pessoas e informações relacionadas a elas. Contudo, a Tecnologia da Informação (TI) dispõe de parte da solução para este problema, não sendo capaz de resolver e garantir totalmente as falhas do universo tecnológico onde as empresas estão inseridas. O autor ainda afirma que em alguns casos a TI pode contribuir e agravar a possível falha.

Corroborando com esta afirmação, o processo de identificação e reparo de falhas pode ser lento e oneroso, e se tratando de *software*, muitas vezes são originados de falhas nas metodologias de desenvolvimento de *software* adotadas pela empresa fornecedora do mesmo. Essas falhas podem afetar a segurança e desempenho do produto utilizado pelo cliente, resultando na ineficácia de toda a estrutura de segurança da informação adotados pela empresa, pois o problema localiza-se em um nível intangível a esta, pois seria a segurança do *software*. De acordo Rangel (2003), o custo relativo para correção de um erro é cerca de duzentas vezes maior na fase de manutenção que na fase de levantamento de requisitos, isto devido a muitos erros passarem despercebidos para estágios mais avançados de desenvolvimento.

Com isto, aumenta-se a necessidade de adoção de padrões, metodologias e políticas devidamente testadas e consolidadas. As empresas que adotam esses requisitos se destacam no mercado de desenvolvimento de *software*. Segundo a Associação Brasileira das Empresas de *Software* (ABE) em 2011ⁱ, o mercado de desenvolvimento, produção e distribuição de *software* era explorado por cerca de 8.520 empresas, sendo 94% classificadas como micro e pequenas empresas.

Entre 2011 e 2012, os autores deste artigo desenvolveram um sistema para o Instituto Federal de Educação, Ciência e Tecnologia de Pernambuco, com o título de Programa Monitoriaⁱⁱ. O sistema foi implantado a partir do edital de seleção de monitoria 2012.2, para cursos superiores e técnicos do campus Recife, e se encontra atualmente em plena operação. O intuito era que este sistema fosse desenvolvido seguindo todas as etapas utilizando conceitos de Engenharia de

Software, ou seja, levantamento de requisitos, casos de uso, plano de testes, diagramas de classes, até a implantação do mesmo, na intenção de que este fosse idealizado como modelo a ser replicado em sistemas futuros da instituição.

Contudo, este artigo tem como objetivo elucidar um dos requisitos não funcionais relacionado à segurança deste sistema, o controle de acesso multinível a chamada de métodos, por meio de Programação Orientada a Aspectos (POA), utilizando uma sintaxe alternativa conhecida como *Annotations*. *Annotations* utiliza marcações predefinidas como *interfaces*, promovendo uma maior segurança e descentralização desta responsabilidade, que normalmente é atribuída a todos os desenvolvedores da equipe.

REVISÃO TEÓRICA

Segurança da Informação

Segurança da informação pode ser definida como o ato de proteger a informação e os sistemas de informação do acesso, uso, divulgação, modificação ou destruição não autorizada. (ANDRESS, 2011).

A segurança da informação em uma organização surge da necessidade da construção de produtos seguros. Segundo Howard e Leblanc (2005),

um produto seguro é um produto que protege a confidencialidade, integridade e disponibilidade das informações dos clientes e a integridade e disponibilidade dos recursos de processamento sob controle do proprietário ou administrador do sistema.

A norma ISO/IEC 27002 (ABNT, 2005) afirma que a segurança da informação é importante para os negócios, tanto do setor público quanto do privado, para proteger as infraestruturas críticas.

Campos (2007) defende que um sistema de segurança baseia-se em três princípios básicos: Confidencialidade, Integridade e Disponibilidade. Andress (2011) corrobora afirmando que esses princípios são comumente referidos como a tríade CIA (do inglês *Confidentiality*, *Integrity* e *Availability*). A tríade CIA apresenta um modelo que possibilita pensar e discutir sobre conceitos de segurança no que diz respeito aos dados.

Confidencialidade

A competitividade de uma organização pode ser determinada, em muitos casos, pelas informações que esta detém e no quão bom é o seu controle. Deverão, por isso, existir mecanismos que garantam a confidencialidade da informação, mas que não impeçam o acesso de pessoas autorizadas à mesma. Os requisitos de confidencialidade são claramente influenciados, ou até mesmo definidos, pela classificação aposta à informação (SILVA et al.,

2006). De acordo com o princípio da confidencialidade, a informação é acessível somente àqueles devidamente autorizados.

A confidencialidade também pode ser atingida com as soluções envolvendo segurança no transporte de dados, através de níveis de criptografia proporcionais à sigilosidade exigida pelos dados e controles de acesso devidamente implementados, entre outras medidas.

Integridade

De acordo com Andress (2011), integridade pode ser definida como a habilidade de evitar que os dados sejam modificados de forma não autorizada e indesejada. Para manutenção da integridade, não basta ter meios para evitar modificações não autorizadas, mas é também preciso ter meios de reverter modificações que precisam ser desfeitas.

Para manter a integridade das informações, podem-se implementar regras automáticas de validação de introdução ou talvez se faça necessária a construção de processos de validação manual. Integridade é particularmente importante quando se discute os dados que fundamentam as bases para a tomada de decisões.

Disponibilidade

Possuir a informação requisitada, porém não apresentá-la no momento da solicitação é o mesmo que não possuir informação alguma. Um sistema que esteja indisponível no momento que o usuário autorizado necessitar de alguma informação pode resultar danos tão graves quanto a perda ou remoção dessa mesma informação no sistema.

Segundo Silva et al. (2006), a disponibilidade consiste em garantir que a informação esteja sempre disponível para uso legítimo, ou seja, por aqueles usuários autorizados pelo proprietário da informação. Consiste ainda na proteção dos serviços prestados pelo sistema de forma que eles não sejam degradados ou se tornem indisponíveis sem autorização, assegurando ao usuário o acesso aos dados sempre que necessário.

Programação Orientada a Aspectos

Comumente, as linguagens tradicionais de programação articulam em sintonia com um projeto de *software*, até aonde estas requeiram apenas abstrações e mecanismos de composição conhecidos e suportados pelos tipos descritos no projeto, ou seja, sub-rotinas, procedimentos, funções, classes, APIs e *frameworks*. Entretanto, esta abordagem pode ser insuficiente dada a necessidade de um determinado projeto em propriedades não tradicionais, como tratamento de exceções, controle de concorrência, etc., onde estas estão distribuídas, na maioria das vezes, em diversos componentes do sistema afetando o desempenho da aplicação (PIVETA, 2001).

Rotinas de acesso multinível ao sistema, rastreamento de ações de usuários e controle de sessões também são tidas como trabalhosas de serem implementadas e fortemente suscetíveis a “quebras”, visto que a menor alteração em sua lógica repercute na necessidade de revisão ou modificação de parte do código das entidades que a referenciam. Além disso, tornam o código muitas vezes de difícil compreensão, isto por que as funcionalidades pertinentes às regras do negócio se misturam com outras referentes à segurança e tratamento de exceções, por exemplo.

Segundo Piveta (2001), Programação Orientada a Aspectos (POA) é uma abordagem que permite a separação dessas propriedades, tidas como ortogonais, dos componentes funcionais do sistema de forma natural e concisa, através de abstrações para representação desses aspectos, para em seguida “combiná-los” de forma transparente para o desenvolvedor, resultando numa aplicação executável.

Entretanto, a POA não trabalha de forma isolada. É um paradigma que complementa outros paradigmas como a Programação Orientada a Objetos (POO), abstraindo elementos não inerentes ao negócio, a fim de facilitar uma rotina onerosa para o desenvolvedor que seria suscetível a erros e promover diminuição do acoplamento entre os interesses e os componentes. Ou seja, boa parte do código é desenvolvida em linguagens tradicionais como Java, C, C++ e C#, e o restante é escrita em linguagem “aspectual”, em unidades (entidades) separadas como, por exemplo, AspectJ e Aspect.NET, sendo a primeira a utilizada neste artigo.

De maneira geral, o emprego deste modelo de programação propicia uma implementação mais ágil dos interesses sistêmicos (requisitos não funcionais) existentes no sistema, que na maioria das vezes está espalhada ao longo dos componentes que o integram, onde esses interesses (*concerns*) são centralizados em entidades de forma coerente, promovendo uma separação mais clara entre o código relativo aos negócios do sistema e o código dos interesses. (RESENDE; SILVA, 2005)

A Figura 1 ilustra a separação dos conceitos apresentados em três camadas: interesses, componentes e código emaranhado (*tangled code*). A camada superior se refere aos interesses, aonde cada nuvem representa um requisito não funcional a ser mesclado no sistema, por exemplo, uma rotina de *log*, tratamento de exceções, controle de concorrência ou uma conexão ao banco de dados. Na camada inferior são apresentados os componentes do sistema e o relacionamento entre eles, responsáveis pelos negócios do sistema e escritos em linguagem tradicional. A camada central representa o código emaranhado, que é o resultado da combinação (*aspect weaver*) entre as duas outras camadas.

De acordo com Piveta (2001), uma implementação baseada no paradigma de Programação Orientada a Aspectos é composta normalmente por uma linguagem de componente, uma ou mais linguagens de aspectos, um combinador de aspectos (*Aspect Weaver*), um programa escrito na linguagem de componentes, e um ou mais programas escritos na linguagem de aspectos, como descrito a seguir.

Figura 1: Representação dos interesses ortogonais e *tangled code*.
Fonte: Winck e Goetten Junior (2006).

AspectJ - Principais elementos

Dado que o AspectJ é uma extensão da linguagem Java, Laddad (2003) apresenta alguns elementos importantes da sintaxe para serem destacados, que têm como objetivo manipular os elementos transversais do paradigma.

Joinpoint: é a forma abstrata para representar um ponto identificável de um programa, como a chamada ou execução de métodos e construtores, acesso a atributos, inicialização de objetos, execução de inicialização estática e tratamento de exceções. No trecho de código apresentado na Figura 2, podem ser descritos alguns *joinpoints*, como execução do método *credit()* ou acesso à variável *balance*.ⁱⁱⁱ

```

1 public class Account {
2     ...
3     void credit(float amount) {
4         balance += amount;
5     }
6 }

```

Figura 2: Exemplo de *joinpoints*.

Pointcut: tem como finalidade selecionar os *joinpoints* e coletar o contexto, como por exemplo, os argumentos passados pelos mesmos. Para se capturar a execução do método *credit()* na classe *Account*, em AspectJ o *pointcut* deverá ser escrito da seguinte forma: “*execution(void Account.credit(float))*”.

Contudo, os *pointcuts* não se restringem à captura da execução de métodos. Segue no Quadro 1 uma relação com as principais designadoras e suas respectivas sintaxes.

Quadro 1: Principais designadores.

Join Point Category	Pointcut Syntax
Method execution	<i>execution(MethodSignature)</i>
Method call	<i>call(MethodSignature)</i>
Constructor execution	<i>execution(ConstructorSignature)</i>
Constructor call	<i>call(ConstructorSignature)</i>
Class initialization	<i>staticinitialization(TypeSignature)</i>
Field read access	<i>get(FieldSignature)</i>
Field write access	<i>set(FieldSignature)</i>
Object initialization	<i>initialization(ConstructorSignature)</i>

Fonte: Piveta (2001).

No AspectJ ainda são utilizados alguns *wildcards* (...*,+) e operadores (!,||,&&), que auxiliam a construção dos *pointcuts*, afim de capturar *pointcuts* que partilham características similares. A Figura 3 ilustra um exemplo de argumento utilizando *wildcards*.

Figura 3: Exemplo de argumento de *pointcut* utilizando *wildcards*.

O (*) denota qualquer número de caracteres, exceto o ponto; (..) denota qualquer número de caracteres, incluindo ponto e (+) qualquer subtipo dado tipo.

Advice: são trechos de código a serem executados quando um *joinpoint* for selecionado por um *pointcut*. Esses trechos podem ser chamados antes (*before*), depois (*after*) ou durante (*around*) à execução do *pointcut*. Cabe também destacar o momento *around*, que pode modificar e até mesmo substituir a execução de um determinado *joinpoint*. Por meio de *advices* é possível, por exemplo, enviar ao usuário uma mensagem depois que um método for executado, conforme pode ser descrito através da sintaxe na Figura 4.

```
1  before() : execution(void Account.credit(float)) {
2      System.out.println("About to perform credit operation");
3  }
```

Figura 4: Exemplo de *advice*.

Aspect: o *aspect* é a unidade central de AspectJ, da mesma forma que as classes são para a linguagem Java. Ele contém as regras expressas e os elementos, ou seja, encapsula os *pointcuts* e *advices*, que serão mesclados ao longo do sistema. Pelo fato do AspectJ ser uma extensão da linguagem Java, nele é possível utilizar normalmente métodos e atributos, da mesma forma que se define uma classe em Java. A Figura 5 apresenta um aspecto, alguns elementos básicos e seus respectivos domínios.

Figura 5: Estrutura básica de um aspecto. **Fonte:** Winck e Goetten Junior (2006).

AspectJ - Sintaxe alternativa (Annotations)

Informalmente conhecido como “@AspectJ”, este modelo suporta o desenvolvimento baseado em anotações para POA em Java. Esta modalidade permite a declaração de *joinpoints* ao longo do código de componentes através de *Annotations*, funcionando como classes de marcadores. Contudo, qualquer que seja o estilo de desenvolvimento adotado, o combinador (weaver) de AspectJ garante o mesmo funcionamento e semântica que o estilo tradicional de desenvolvimento, sendo permitido inclusive a mistura de ambas modalidades, tradicional e baseada em anotações, no mesmo arquivo.

Será apresentada abaixo a relação de equivalências entre as modalidades citadas e os conceitos de *joinpoints*, *pointcuts*, *advices* e *aspects*.

Joinpoint: diferentemente do modelo apresentado anteriormente, onde os *joinpoints* eram tratados de forma abstrata, neste agora apresentado se faz necessária uma declaração literal. Para qualquer tipo de elemento anotado (classe, método, construtor, pacote, etc.), um padrão de anotação pode ser usado para corresponder a um conjunto de anotações, como por exemplo, uma classe inteira da linguagem de componente. Esse tipo de anotação possui duas formas básicas:

- @<qualified-name>, por exemplo, @Foo ou @org.xyz.Foo.
- @(<type-pattern>), por exemplo, @(org.xyz .. *) ou @(Foo || Boo).

O trecho de código, presente na Figura 6, descreve um exemplo da primeira forma citada, aonde previamente foi criada uma interface chamada “ICadastro”, pertencente ao mesmo pacote para identificar o tipo de operação. Contudo, este tipo de marcação não se restringe a métodos, sendo possível também em classes, construtores, etc.

```

1 public class SolicitacaoController {
2     ...
3     @ICadastro
4     public String create() {
5         ...
6         System.out.println("Cadastro realizado!");
7     }
8 }
```

Figura 6: Exemplo de *joinpoint* por anotações.

Pointcut: de maneira geral, um método anotado como *pointcut* não deve possuir corpo e não deve levantar cláusulas *throws*. Contudo, é possível utilizar parâmetros ou argumentos de ligação. Um exemplo de *pointcut* com designador de execução e sem argumentos é apresentado no trecho de código presente na Figura 7.

```

1 @Pointcut("execution(@ICadastro public * *.*(..))")
2 public void logCadastroPointCut() {
3     //Corpo vazio
4 }
```

Figura 7 – Exemplo de *pointcut* por anotações.

O *pointcut* da Figura 7 resulta na captura de todos os métodos do sistema, que possuam a anotação “ICadastro”. As *Wildcards* mencionadas anteriormente também podem ser utilizadas para a construção dos *pointcuts* baseados em anotações.

Advices: os *advices* baseados neste estilo também possuem trechos de código escritos em linguagem Java tradicional e são representados pelas cláusulas *Before*, *After*, “*AfterReturning*”, “*AfterThrowing*” e *Around*, aonde os métodos marcados por estes devem ser declarados como públicos. Excepcionalmente no caso do *advice* do tipo *Around*, o retorno do método deverá ser *void*. Na Figura 8 é apresentado um exemplo de *advice*, utilizando a cláusula *Before*, que acontece antes da execução do *Pointcut* “logCadastroPointCut()”.

```

1  @Before("logCadastroPointCut()")
2  public void beforeLogCadastroPointCut(JoinPoint thisJoinPoint) {
3      System.out.println("Levantado antes do cadastro - " + thisJoinPoint);
4  }

```

Figura 8: Exemplo de *advice* por anotações.

A diferença entre os tipos “*AfterReturning*”, “*AfterThrowing*” e *After*”, é que o primeiro é chamado caso o *pointcut* que o mesmo indicou resultou em sucesso, o segundo é levantado caso ocorra alguma exceção no *pointcut* indicado e o terceiro chamado em todos os casos.

Aspect: a Figura 9 ilustra a estrutura básica de um aspecto construído utilizando o modelo baseado em anotações. Pode-se observar que sua forma é similar a uma classe comum em Java, contudo possui os marcadores @{Aspect, Pointcut, Before}, que o torna característico para a sintaxe *@AspectJ* e a IDE *NetBeans*.

```

package aspects;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.*;

@Aspect
public class CadastroAspect {

    @Pointcut("execution(@ICadastro public * *.*(..))")
    public void logCadastroPointCut() {
    }

    @Before("logCadastroPointCut()")
    public void beforeLogCadastroPointCut(JoinPoint thisJoinPoint) {
        System.out.println("Acontece antes do cadastro - " + thisJoinPoint);
    }
}

```

Figura 9: Estrutura de um aspecto baseado em *Annotations*.

De acordo com Laddad (2003), AspectJ oferece suporte integrado para edição, compilação, execução e depuração em diversos IDEs populares, como Eclipse, Netbeans e JBuilder. Entretanto, a Xerox Corporation, detentora do projeto AspectJ, transferiu definitivamente os direitos para a Eclipse.org, que é responsável por dar suporte e manter o código sob a filosofia *open source*.

Contudo, esta informação está desatualizada, pois o módulo de extensão, também conhecido como *plugin*, *AspectJ Development Environment* (AJDE) para NetBeans não é

atualizada desde a versão 3.5 da IDE^{iv}, limitando o desenvolvimento apenas a nível de experiência, visto que esta versão do NetBeans está obsoleta. Para a resolução desse problema, pode-se adotar uma solução alternativa baseada no conceito de “*Ant Support*”.

METODOLOGIA

O objeto de estudo desta pesquisa foi realizado em um dos requisitos não funcionais relativo à segurança do sistema Programa Monitoria do IFPE, campus Recife. Consistiu em modelar uma política de desenvolvimento que tratasse o acesso aos métodos dos controladores do sistema de forma multinível, onde só fosse autorizado o acesso aos usuários com permissões suficientes, de acordo com seu perfil associado no sistema e operação solicitada.

Todo o projeto foi desenvolvido seguindo a metodologia do Processo Unificado Racional, mais conhecido como RUP, que contempla as etapas de plano de projeto, levantamento de requisitos, diagramação de casos de uso, implementação, testes, implantação e a manutenção do projeto.

Ambiente de Desenvolvimento

Para alcançar o objetivo, que era a elaboração do modelo proposto, foi necessário realizar a preparação física do ambiente de desenvolvimento e testes, utilizando os recursos disponíveis da instituição ou adequando-se a alternativas possíveis para realização do mesmo. Como estava definida a utilização da IDE NetBeans no plano de projeto do sistema, devido ao fato de sua licença ser gratuita, para fins não-comerciais, e dos inúmeros recursos disponíveis, utilizou-se a versão 7.0.1. Foi necessário configurá-la manualmente, para que o mesmo trabalhasse com a versão mais atual do AspectJ, visto que o *plugin* AJDE está obsoleto e não oferece suporte desde a versão 3.5 do NetBeans. Contudo, foi elaborado um tutorial (disponibilizado como projeto no *GoogleCode*^v), no qual é demonstrado o passo-a-passo desta configuração e onde obter as ferramentas necessárias, como o compilador AspectJ e suas bibliotecas, bem como um pequeno projeto *Web* previamente configurado.

Modelo Proposto

Por se tratar de um requisito não funcional, vislumbrou-se a utilização do paradigma de Programação Orientado a Aspectos devido à sua praticidade na implementação de requisitos sistêmicos, que possivelmente estão espalhados na maioria dos módulos.

O sistema possui quatro atores, sendo que a segurança multinível é realizada em apenas três (Administrador, Coordenador e Professor) e os casos de uso que o ator Aluno participa não possuem segurança, pois estes não realizam cadastro de perfil no sistema. Cada perfil contém um conjunto de responsabilidades e operações liberadas. Contudo, os perfis compartilham algumas

funcionalidades, a hierarquia de acesso permite que um perfil acima contenha todas as permissões de um perfil abaixo, onde o Administrador está no topo e o Professor na base. A Figura 10 apresenta as responsabilidades básicas de cada ator do sistema.

Quadro 2: Relação ator vs atribuições.

ATOR	DESCRIÇÃO
Administrador	Possui poderes administrativos, tais como criação, alteração de usuários, coordenações, departamento, listagens, etc.
Coordenador	O ator Coordenador é o responsável por basicamente efetuar as análises dos pedidos, tanto dos componentes curriculares quanto das inscrições dos alunos.
Professor	Este ator é responsável por efetuar o cadastro do plano de monitoria e componentes curriculares, bem como emitir listagem dos alunos selecionados e fichas de inscrição.
Aluno	O ator Aluno pode inscrever-se para monitoria de uma determinada disciplina, verificar o resultado através da listagem e reemitir ficha de inscrição.

RESULTADOS

Inicialmente foi definida a criação de um aspecto genérico para cada tipo de usuário, aonde era capturado da sessão o perfil do usuário e elencados todos os controladores disponíveis para ele. Entretanto, esta ideia não resultou em sucesso.

Partindo desse embargo, surgiu a necessidade de utilizar a sintaxe alternativa *Annotations*, que consistia em “anotar” sobre cada operação do sistema um nível de usuário permitido. Para garantir uma maior padronização foram criadas interfaces para cada operação básica do sistema, como consulta, cadastro, alteração e exclusão. Também foram criadas três interfaces para representar os atores Administrador, Coordenador e Professor, denominados respectivamente no sistema como *Level A*, *Level B* e *Level C*.

O trecho de código apresentado na Figura 10 apresenta como é descrita uma interface em Java para utilização com *Annotations*, lembrando que o corpo deverá ser vazio.

```

1  @Target(ElementType.METHOD)
2  @Retention(RetentionPolicy.CLASS)
3  public @interface ILevelA {
4      //Corpo vazio
5  }
```

Figura 10: Exemplo de interface para anotações.

Com as interfaces bem definidas, as regras referentes à segurança multinível puderam ser implementadas seguindo a estrutura de acordo com o trecho de código da Figura 11.

```

1  @Aspect
2  public class AccessLevelAspect {
3      ...
4      @Around("execution(@ILevelA public * *.(..))")
5      public Object proceedLevelA(ProceedingJoinPoint thisJoinPoint)
6          throws Throwable {
7          ...
8      }
9  }
```

Figura 11: Exemplo do modelo de anotação para aspectos.

O *Advice* presente no aspecto implementado no trecho de código da Figura 11 resulta na “interceptação” dos métodos que foram marcados como “ILevelA”, sendo executados o trecho de código contido neste *Advice*. No sistema foi utilizada essa cláusula para verificar na sessão o tipo de usuário logado que chama o método, para prosseguir ou não com a execução do método.

Em seguida foram marcados os principais métodos dos controladores que realizavam as regras de negócio, relacionando a funcionalidade com o nível de acesso permitido, baseando-se no documento de casos de uso do sistema. A marcação foi realizada da maneira descrita no trecho de código da Figura 12, que resultava no controle da chamada do método relativo à criação de edital do sistema, sendo permitido apenas aos usuários “Level A”.

```

1 public class TbEditalController implements Serializable {
2     ...
3     @ILevelA
4     public String create() {
5         ...
6     }

```

Figura 12: Exemplo de marcação dos métodos.

Para realizar os testes do modelo implementado, foi simulada uma situação clássica de erro, que comumente acontece em ambientes de desenvolvimento aonde as responsabilidades dos membros não são bem definidas ou há membros que participam de mais de uma equipe, propiciando certa insegurança durante a construção do sistema. A situação proposta foi montada de forma que se oferecia um formulário não pertencente a um determinado perfil de usuário, atribuindo responsabilidades não inerentes ao mesmo e tornando o sistema inseguro e menos coeso.

Para isto, foi criado um perfil do tipo Professor e disponibilizado a este um formulário de criação de Departamento, visto que este tipo de operação só é permitida ao Administrador do sistema, conforme pode ser observado nas Figuras 13 e 14.

Figura 13: Tentativa de cadastro sem permissão.

Figura 14: Resultado de uma ação não permitida.

O resultado desta tentativa gerou um registro de *log* na base de dados do sistema, como uma ação suspeita, invalidando automaticamente a sessão do usuário, conforme pode ser visto na Figura 15.

1..3/3

Log N°	Usuário	Tipo Ação	Método/Invoke	Resultado	Hora/Data	Endereço IP
1	Maria José de Almeida	LOGIN	execution(public java.lang.String controller.LoginController.ValidaUsuario())	SUCESSO	13:45:51 08/03/2012	127.0.0.1
2	Maria José de Almeida	ACESSO SUSPEITO	execution(public java.lang.String controller.TbDepartamentoController.create())	FALHA	13:46:01 08/03/2012	127.0.0.1
3	Administrador	LOGIN	execution(public java.lang.String controller.LoginController.ValidaUsuario())	SUCESSO	13:46:39 08/03/2012	127.0.0.1

Sair

Figura 15 : Tabela de log do sistema.

CONCLUSÕES

A Programação Orientada a Aspectos agrega valores inovadores no desenvolvimento do sistema. Dentre seus benefícios pode-se destacar a modularização de requisitos não funcionais do programa, uma vez que a partir do momento em que ela possui a capacidade de agrupar vários interesses em um aspecto, dividindo o problema em partes menores, ela vem a colaborar significativamente no que diz respeito à modularização, trazendo benefícios para o trabalho em equipe, visto que não é mais necessário que os especialistas na parte de segurança e tratamento de exceções, dentre outros, conheçam todo o código referente ao negócio do sistema.

Isto contribui diretamente para a redução do espalhamento e o entrelaçamento de código-fonte por vários componentes, reduzindo a complexidade de tal código-fonte, tornando-o mais simples, de fácil entendimento e, conseqüentemente, facilitando também sua manutenção e reusabilidade, visto que os interesses ortogonais do sistema estão centralizados em aspectos bem definidos. Contudo, como a linguagem AspectJ possibilita a construções de regras bem abrangentes, estas devem ser escritas com bastante cautela, pois podem atingir áreas indesejadas do sistema, principalmente ao se utilizar as *Wildcards*.

O sistema desenvolvido, nomeado como Programa Monitoria, foi implementado com este paradigma, proporcionando todos os benefícios relacionados ao mesmo. O modelo de acesso

multinível, comum nos sistemas *Web*, possui muito código entrelaçado e de difícil manutenção com as mudanças de requisitos inerentes ao desenvolvimento de *software*.

Entretanto, depois de ter sido bem definida a estrutura dos interesses relativos à segurança multinível, as regras foram implementadas com facilidade utilizando *Annotations* por meio de interfaces, que serviram para relacionar o método chamado ao perfil de usuário permitido. O sistema está integrado ao da instituição de ensino e em produção, sendo aprovado tanto pelo corpo docente como pelo discente, encorajando a prática nas fábricas de *software* e nas salas de aula, formando um profissional mais qualificado e preparado para o mercado de trabalho.

Como trabalhos futuros, espera-se estudar técnicas de ataque em aplicações *Web* tais como sequestro de sessões, *SQL-injection*, *Forceful Browsing* e outros métodos intrusivos, com o intuito de viabilizar soluções para essas vulnerabilidades por meio de Programação Orientada a Aspectos.

REFERÊNCIAS

ABNT. Associação Brasileira de Normas Técnicas. Tecnologia da informação: código de prática para a gestão da segurança da informação. In: **NBR ISO/IEC 27002**. Rio de Janeiro, 2005.

ANDRESS, J.. **The basics of information security**: understanding the fundamentals of infosec in theory and practice. USA: Elsevier, 2011.

CAMPOS, L. N. A.. **Sistema de segurança da informação**. Florianópolis: Visual Books, 2007.

LADDAD, R.. **AspectJ in action**: practical aspect-oriented programming. Greenwich: Manning, 2003.

MARCIANO, J. L. P.. **Segurança da informação**: uma abordagem social. Tese (Doutorado em Ciência da Informação) – Universidade de Brasília, Brasília, 2006.

PIVETA, E. K.. **Um modelo de suporte à programação orientada a aspectos**. Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis, 2001.

RANGEL, G. S.. **Uma Ferramenta de Prototipação de Software para o Ambiente PROSOFT**. Dissertação (Mestrado em Informática) – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2003.

RESENDE, A. M. P.; SILVA, C. C.. **Programação orientada a aspectos em Java**. São Paulo, Brasport, 2005.

SILVA, P. T.; CARVALHO, H.; TORRES, C. B.. **Segurança dos sistemas de informação**: gestão estratégica da segurança empresarial. Centro Atlântico: Lisboa, 2003.

WINCK, D. V.; GOETTEN JUNIOR, V.. **AspectJ**: programação orientada a aspectos com Java. São Paulo: Novatec, 2006.

i http://www.abes.org.br/UserFiles/Image/PDFs/Mercado_BR2011.pdf

ii <http://monitoria.recife.ifpe.edu.br>

iii <http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html>

iv <http://aspectj-netbeans.sourceforge.net>

v <http://code.google.com/p/projeto-netbeans-poa>