

LIGA DE ENSINO DO RIO GRANDE DO NORTE
FACULDADE NATALENSE PARA O DESENVOLVIMENTO DO RIO GRANDE DO
NORTE

Vladimir Fernandes Raposo

UMA ABORDAGEM PRÁTICA DE PROGRAMAÇÃO ORIENTADA A
ASPECTOS

UMA ABORDAGEM PRÁTICA DE PROGRAMAÇÃO ORIENTADA A
ASPECTOS

NATAL/RN

2007

VLADIMIR FERNANDES RAPOSO

UMA ABORDAGEM PRÁTICA DE PROGRAMAÇÃO ORIENTADA A ASPECTOS

Monografia apresentada ao curso de bacharelado em sistemas de informação da Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte como requisito obrigatório para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientadora:
Bartira Rocha, M.Sc.

NATAL/RN

2007

RB. 31466

M

FARN
BIBLIOTECA
Nº Reg. <u>0000968</u>
<u>17/9/08</u>

004.43

R128a

2007

ex. 01

Catálogo na Publicação – Biblioteca da FARN
Setor de Processos Técnicos

Raposo, Vladimir Fernandes

Uma abordagem prática de programação orientada a aspectos / Vladimir
Fernandes Raposo. – Natal: FARN, 2007.
60f.

Orientador: Bartira Rocha (M. Sc.)

Monografia (Bacharelado Sistemas de Informação) – Faculdade Natalense para
o Desenvolvimento do Rio Grande do Norte.

1. Desenvolvimento de Softwares – Monografia. 2. Programação Orientada a
Aspectos – Monografia. I. Título.

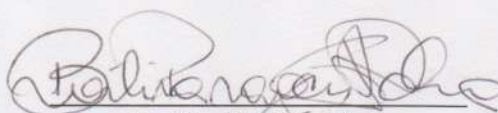
RN/FARN/BC

CDU 004.43

VLADIMIR FERNANDES RAPOSO

UMA ABORDAGEM PRÁTICA DE PROGRAMAÇÃO ORIENTADA A
ASPECTOS

Monografia apresentada ao curso de bacharelado em sistemas de informação da
Faculdade Natalense para o Desenvolvimento do Rio Grande do Norte e aprovada em
___ de ___ de ___, pela banca examinadora composta pelos seguintes membros:



Bartira Rocha, MsC
Orientador

Membro examinador

ALDOZILDO

Membro examinador

CLEYDSON

RESUMO

A complexidade crescente dos sistemas de informação tem obrigado a programadores e projetistas a evoluírem com as ferramentas e técnicas de desenvolvimento, sendo a Programação Orientada a Objetos a técnica mais difundida, porém ela possui limitações, algumas características dos sistemas de software não conseguem ser bem implementadas em classes comuns, pois estão presentes no sistema como um todo não pertencendo especificamente a nenhum domínio específico, essas características são chamadas de: características transversais. Com isso a uma quebra de sua modularidade, espalhando seu código entre todas as outras classes e dificultando a sua manutenção, a Programação Orientada a Aspectos - POA veio tentar resolver esse problema, melhorando a modularização do sistema, e encapsulando essas características em módulos análogos as classes, chamado de aspectos.

Keywords: programação, java, objetos, programação, aspect, AspectJ, aspect

Palavras-chave: programação, java, programação orientada a objetos, aspectos, AspectJ, programação orientada a aspectos.

ABSTRACT

The increasing complexity of information systems has forced the programmers and designers to evolve frequently with the tools and techniques of development, the Object Oriented Programming is the technique more widespread, but it has limitations, some characteristics of systems software can not be well implemented in common classes, as are present in the system as a whole does not belong to any of the specific areas of the system, called crosscutting concerns, thus breaking its modularity, spreading its code among all other classes and more difficult to maintain, the Aspect Oriented Programin - AOP was trying to solve this problem by improving modularization the system, and encapsulating these characteristics in similar modules of the classes, called aspects.

Keywords: programming, java, object oriented programming, aspect, AspectJ, aspect oriented programming.

LISTA DE FIGURAS

Figura 1: Etapas do desenvolvimento de software orientado a aspectos.	13
Figura 2: Interesse transversal de atualização da tela	15
Figura 3: Espalhamento do código de logging entre as classes no TomCat	16
Figura 4: Diagrama de classes UML.....	23
Figura 5: Código do logging “entrelaçado” entre os métodos	26
Figura 6: Código do Logging “espalhado” entre as classes do sistema.....	27
Figura 7: Métodos da classe Alunos sem o código de Logging	27
Figura 8: Métodos de várias classes sem o código de Logging.....	28
Figura 9: Divisão do aspecto entre o pointcut e o advice.....	28
Figura 10: Reflexão e chamada ao método da classe Log	29
Figura 11: Código do aspecto Logging que não é um advice	30
Figura 12: Advice usado para inicializar o sistema com o Login.....	31
Figura 13: Métodos da classe Alunos com o código para verificação de permissão.....	31
Figura 14: Advice com o código de verificação de permissões de acesso.....	32
Figura 15: Inclui a funcionalidade de troca de usuário ao sistema.....	32

2.4.1. Advice	19
2.4.2. Aspecto Declarativo	19
2.4.3. Advice	19
3. METODOLOGIA	20
4. DESENVOLVIMENTO	21
4.1. DESCRIÇÃO DO SISTEMA	21
4.1.1. Diagrama de Classes do Sistema	23
4.2. UTILIZAÇÃO DE ASPECTOS NO DESENVOLVIMENTO DO SISTEMA	24
4.2.1. Uso de Aspectos no Logging	24
4.2.2. Uso de Aspectos na Autenticação e Controle de Acesso	25
4.3. OPERACIONALIZAÇÃO DO LOGGING	26
4.4. AUTENTICAÇÃO E CONTROLE DE ACESSOS COM ASPECTOS	29
5. CONCLUSÃO	33
REFERÊNCIAS	34
APÊNDICE A – Código do sistema	36
APÊNDICE B – Dicionário de Dados	59

1 INTRODUÇÃO

SUMÁRIO

1 INTRODUÇÃO	7
1.1. PROBLEMA DE PESQUISA	7
1.2. OBJETIVOS	8
1.2.1. Objetivo Geral	8
1.2.2. Objetivos Específicos	8
1.3. JUSTIFICATIVA	9
2 REFERENCIAL TEÓRICO	10
2.1 PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE NA ATUALIDADE	10
2.2 O QUE SÃO ASPECTOS	11
2.3 CONCEITOS RELACIONADOS A POA	13
2.3.1 Interesses (<i>concerns</i>)	13
2.3.2 Interesses Transversais (<i>crosscutting concerns</i>)	14
2.3.3 Entrelaçamento (<i>code tangling</i>) e Espalhamento de Código (<i>code scattering</i>)	15
2.4 ASPECTJ	16
2.4.1 <i>Join Points</i>	17
2.4.2 <i>Pointcuts</i>	18
2.4.3 <i>Advices</i>	18
2.4.4 <i>Inter-type Declarations</i>	19
2.4.5 <i>Aspect</i>	19
3 METODOLOGIA	20
4 DESENVOLVIMENTO	21
4.1. DESCRIÇÃO DO SISTEMA	21
4.1.1. Diagrama de Classes do Sistema	23
4.2. UTILIZAÇÃO DE ASPECTOS NO DESENVOLVIMENTO DO SISTEMA	24
4.2.1. Uso de Aspectos no <i>Logging</i>	24
4.2.2. Uso de Aspectos na Autenticação e Controle de Acessos	25
4.3. OPERACIONALIZAÇÃO DO LOGGING	26
4.4. AUTENTICAÇÃO E CONTROLE DE ACESSOS COM ASPECTOS	29
5 CONCLUSÃO	33
REFERENCIAS	34
APÊNDICE A – Código do sistema	36
APÊNDICE B – Dicionário de Dados	59

1 INTRODUÇÃO

A atual vanguarda na tecnologia de desenvolvimento de sistemas é a POO - Programação Orientada a Objetos - largamente utilizada hoje em dia e com eficiência comprovada, porém essa tecnologia não é adequada para a modelagem e implementação de todos as especificidades dos sistemas atuais, cada vez mais exigentes e com novas tecnologias.

Muitas das novas características e exigências desses sistemas não são suficientemente bem implementadas com as ferramentas atuais, notadamente a POO, precisando de uma nova abordagem para partes específicas do mesmo. Entra nesse cenário a POA - *Programação Orientado a Aspectos*, uma evolução da POO e um novo paradigma de desenvolvimento que visa preencher as lacunas, com relação as chamadas características transversais, encontradas nos sistemas atuais.

A POA propõe uma separação entre os níveis de exigências de um sistema ou *concerns*, provendo uma melhor estruturação, modularização e diminuição da complexidade do código gerado.

Essa nova tecnologia, Integrada a POO, permite uma melhor abstração englobando características que muitas vezes não são suficientes em um único módulo da POO, por terem comportamentos que na verdade estão distribuídos por vários outros módulos do sistema, conhecidas como características transversais (*crosscutting concerns*).

Este projeto pretende analisar a Programação Orientada a Aspectos, que se apresenta como uma evolução da Programação Orientada a Objetos POO, utilizando-se da linguagem de programação Java e da Ferramenta AspectJ na implementação do código.

1.1. PROBLEMA DE PESQUISA

Com o advento e a popularização de redes mundiais (como a internet), equipamentos portáteis e a convergência de várias tecnologias, o cenário para o desenvolvimento e o uso de sistemas de informação também mudou. Essas mudanças têm que ser acompanhadas por uma nova visão no desenvolvimento

desses mesmos sistemas, para que eles se adaptem as exigências e ao ambiente em que elas são implementadas e usadas.

A Programação Orientada a Objetos, possui uma boa modularidade provendo uma boa separação entre os domínios do sistema através de classes, atributos e métodos, encapsulando bem os requisitos funcionais do sistema. Porém essa divisão não é suficiente para abstrair algumas características que se propagam por todo o sistema, tornando o código “amarrado”, isto é, difícil de ser reutilizado e com características redundantes, essas características são chamadas de *crosscutting concerns* e é o principal motivo da Programação Orientada a Aspectos existir.

Utilizando-se da orientação a aspectos nos projetos já existentes, e principalmente, nos que ainda serão construídos, é possível melhorar a modularização e até mesmo alterar algumas características que não existiam no projeto original do sistema, tornando-o software melhor de alterar, evoluir e reutilizar, no todo, ou em partes, em outros projetos.

1.2. OBJETIVOS

1.2.1. Objetivo Geral

Este trabalho tem como objetivo geral identificar e implementar as características transversais no sistema de Cadastro de Projetos de Extensão da FARN utilizando Programação Orientada a Aspectos e AspectJ.

1.2.2 Objetivos Específicos

- Apresentar e analisar a metodologia de Programação Orientado a Aspectos
- Apresentar a linguagem AspectJ que será utilizada na implementação das características transversais
- Analisar e identificar as características transversais do sistema a ser desenvolvido
- Implementar as características transversais com a ferramenta AspectJ

- Analisar as vantagens na programação do sistema usando apenas Orientação a Objetos em comparação com o mesmo sistema utilizando Orientação a Aspectos.

1.3. JUSTIFICATIVA

O aumento na complexidade dos sistemas de software gera uma demanda pelo aumento de técnicas de análise e desenvolvimento, forçando as ferramentas a evoluírem junto com as exigências do mercado. Muitas técnicas de desenvolvimento, análise e gerência, têm aparecido e desaparecido, ou simplesmente evoluído, agregando valor as suas versões antigas ou criando uma estrutura totalmente nova.

Na atualidade o paradigma de programação orientado a objetos, está sofrendo uma mudança evolutiva que tem como nova característica a previsão do uso mais coerente de características transversais, ou seja, aquelas características do sistema que não podem ser dissociadas do todo, em módulos, precisando portanto de uma abordagem específica para esse fim.

Segundo Dijkstra (1976), o princípio da separação de interesses (*concerns*) tem como objetivo dividir o domínio do sistema em partes menores com o intuito de entender melhor cada parte isoladamente, esse 'interesse' é uma parte específica do problema e deve ser tratado com uma abordagem diferente do resto do sistema, mesmo que todas essas características façam parte do mesmo sistema.

Dentro do escopo de um sistema esses 'interesses' podem ser requisitos funcionais ou não funcionais e são divididos em categorias de acordo com suas características. Utilizando-se de ferramentas já existentes para implementarem as Características Transversais, essas ferramentas podem ser de uso geral ou específico, uma vez que existem ferramentas apropriadas para resolver problemas específicos e outras que cobrem todos os requisitos.

Uma vez implementado, os aspectos são acrescentados ao código do Sistema Orientado a Objetos como se fosse parte da linguagem, através de um Combinador de aspectos (weaver), resultando em um sistema final completo, não sendo portanto a POA uma ferramenta de desenvolvimento por si só, mas um complemento da POO.

2 REFERENCIAL TEÓRICO

2.1 PARADIGMAS DE DESENVOLVIMENTO DE SOFTWARE NA ATUALIDADE

Com o crescente avanço das técnicas de desenvolvimento de sistemas e com um mercado cada vez mais exigente, novas características e desafios vão surgindo na forma de se analisar e implementar um software, segundo Gradeck;Lisiecki (2003), a engenharia de software e as linguagens de programação possuem um relacionamento mútuo.

Por volta da década de 70 uma nova linguagem a SmallTalk começa a se popularizar e inicia-se uma nova abordagem nas linguagens de programação com a POO - Programação Orientada a Objetos, substituindo com sucesso as tradicionais linguagens estruturadas, tendo grandes avanços no desenvolvimento até os dias atuais sendo atualmente o paradigma de programação dominante no desenvolvimento de sistemas de software (Laddad, 2003).

A POO possibilitou uma melhor modularização dos sistemas construídos separando essas características em classes que trabalham em conjunto formando o sistema em si. Segundo (GRADECK;LESIECKI, 2003) aplicações são coleções de módulos (classes) que trabalham juntas para fornecer determinadas características para um conjunto de requisitos bem definidos. A POO deixou os sistemas mais fáceis de serem projetados, com maior reusabilidade de seu código, modularidade, encapsulamento de interesses e menores custos de manutenção.

Porém, algumas propriedades do sistema que se pretende desenvolver não são adequadamente bem implementadas na POO, uma vez que essas propriedades podem estar presentes em várias classes ao mesmo tempo não fazendo parte da natureza de nenhuma delas em específico, desse modo há um acúmulo nas responsabilidades daquela classe, causando um entrelaçamento de sua responsabilidade inicial.

Devido a esse entrelaçamento nas responsabilidades das classes e da dificuldade de se implementar essas características de acordo com a POO, surgiu o paradigma de programação orientada a aspectos. A implementação dos aspectos fornece subsídio a POO para tratar de tais responsabilidades

extras através da decomposição funcional da classe (SOARES; BORBA, 2002).

A Programação Orientada a Aspectos – POA, foi proposta com o objetivo de facilitar a modularização dos interesses transversais, complementando a POO, ela não tem o intuito de substituir a POO, mas sim, ser uma nova técnica adicional ao desenvolvimento orientado a objetos. A programação orientada a aspectos permite aos programadores separar os interesses comuns do sistema, comportamentos que fogem a típica divisão de responsabilidade, como o logging (registro de todas as operações executadas pelo sistema).

Um aspecto faz parte de todo o processo de desenvolvimento de software, permitindo que o desenvolvedor se concentre em um assunto ou característica do sistema por vez (DIJKSTRA, 1976), sendo ele parte não apenas da lógica de um sistema mas como na análise e todos os processos referentes a engenharia de software.

O termo Desenvolvimento de Software Orientado a Aspectos, se contrapõe ao outrora mais comum Programação Orientada a Aspectos para enfatizar que o paradigma de aspectos é aplicável a todas as etapas do ciclo de desenvolvimento de software: requisitos, análise, projeto, implementação, testes etc. (Chaves, 2003).

2.2 O QUE SÃO ASPECTOS

Programação Orientada a Aspectos - Aspect-Oriented Programming ou simplesmente AOP - foi introduzida em 1996 por Gregor Kiczales, quando estava no Xerox Palo Alto Research Center.(GOETTEN JUNIOR, 2006)

A programação orientada a aspectos consiste na separação dos interesses (*concerns*) de um sistema em unidades modulares e posterior composição desses em um único sistema, algumas características de um sistema não são devidamente representados na POO por fazerem parte de todo o sistema e espalhado por vários módulos, não sendo de responsabilidade de nenhum deles em específico, um bom exemplo disso seria o logging de transações ou as diretivas de segurança de um sistema, esse espalhamento de interesses, também conhecido como interesses transversais (*crosscutting*

concerns), complementa a programação orientada a objeto por facilitar um outro tipo de modularidade que expande a implementação espalhada de uma responsabilidade dentro de uma simples unidade, esta unidade é chamada de aspecto. (LADDAD, 2003)

Na programação orientada a objetos a unidade natural de modularização é a classe, e um *crosscutting* é uma responsabilidade que está espalhada por muitas classes, este tipo de comportamento chama-se *crosscutting* porque ele atravessa a divisão de responsabilidades dos modelos de programação tradicionais, um interesse (*concerns*) pode ser visto como um requisito funcional ou não funcional de um sistema, os interesses são divididos em módulos como se fossem classes, O paradigma da AOP consiste na separação das responsabilidades transversais de um sistema em aspectos (unidades modulares) e a sua posterior composição junto às classes, assume-se que os aspectos são invisíveis para os componentes das classes da POO, é a idéia de que os componentes não precisam conhecer nem ser preparados para receber as mudanças providas pelos aspectos (LYRENE, 2006), ou seja, qualquer código válido na linguagem é passível de utilização pelos aspectos.

Depois de definir as responsabilidades entrecortantes, pertencentes ao sistema como um todo sistema, inserimos essa responsabilidade em um módulo, chamado aspecto, isso é feito usando uma linguagem de aspecto. O último passo, é unir os aspectos com o restante do código do sistema propriamente dito, essa união é feita com um montador (*weaver*), e então temos como resultado um programa completo.

O desenvolvimento de um sistema com Aspectos consiste basicamente em três etapas:

Decomposição (*aspectual decomposition*): identificar entre os diversos interesses (*concerns*) do sistema, quais os que são interesses de negócios comuns e interesses transversais.

Implementação (*concern implementation*): implementar os interesses anteriormente identificados separadamente, em classes para os *concerns* comuns e em aspectos para os *crosscutting*.

Recomposição (*aspectual recomposition*): a recomposição consiste na integração dos aspectos com os componentes já implementados, de acordo com regras de recomposição, esta fase é chamada de *weaving*.

Essas etapas foram propostas por LADDAD (2003) e podem ser melhor entendidas de acordo com a figura abaixo:

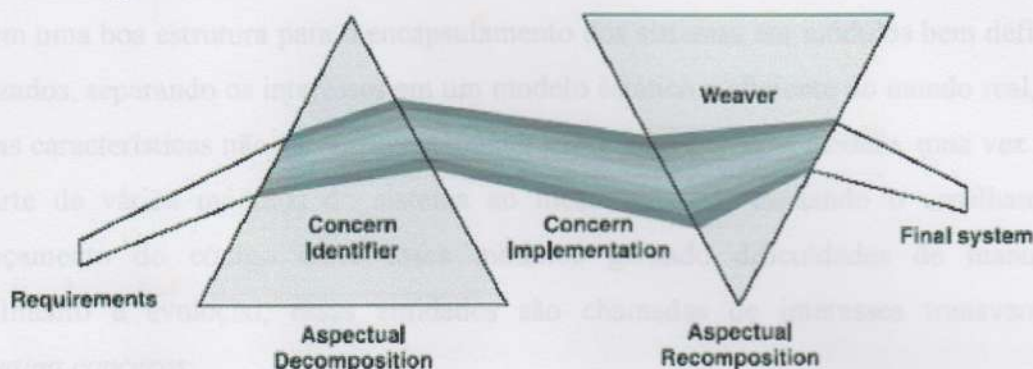


Figura 1: Etapas do desenvolvimento de software orientado a aspectos.
Fonte: LADDAD (2003)

2.3 CONCEITOS RELACIONADOS A POA

2.3.1 Interesses (*concerns*)

Introduzido em DIJKSTRA (1976), o princípio da separação em interesses (*concerns*) tem como objetivo dividir o domínio do sistema em partes menores com o intuito de entender melhor cada parte isoladamente. Cada *concern* é tratada como parte conceitual única do domínio do sistema, um *concern* pode ser um requisito funcional ou não funcional do sistema, e deve ser separado em módulos de acordo com a forma que este está sendo implementado.

Segundo LADDAD (2003) os interesses de um sistema de software podem ser classificados como:

Interesses do Negócio: capturam a funcionalidade central de um módulo, por exemplo, procedimento de quitação de uma compra.

Interesses em Nível de Sistema: capturam requisitos periféricos, no nível do sistema e que atravessam múltiplos módulos, por exemplo, segurança, logging e persistência.

2.3.2 Interesses Transversais (*crosscutting concerns*)

A necessidade de manipular um tópico importante de cada vez durante o projeto de um sistema é chamado de princípio da separação de interesses KICZALES (2003), no paradigma da POO as unidades básicas de abstração são as classes, objetos, métodos e propriedades que possuem uma boa estrutura para o encapsulamento dos sistemas em módulos bem definidos e organizados, separando os interesses em um modelo estático e eficiente do mundo real, porém algumas características não são suficientemente suportadas por esse modelo, uma vez que ele faz parte de vários módulos do sistema ao mesmo tempo, causando o espalhamento e entrelaçamento do código entre esses módulos gerando dificuldades de manutenção, entendimento e evolução, essas entidades são chamadas de interesses transversais ou *crosscutting concerns*.

Segundo Elrad (1997), alguns interesses não são claramente decompostos em apenas uma dimensão, por exemplo, objetos. Neste contexto, nasce a Programação Orientado a Aspectos - POA - outras abordagens precederam a POA na tentativa de corrigir essas falhas, tais como: linguagens específicas de domínio, programação generativa, desenvolvimento orientado a características, sendo que todas elas contribuíram de algum modo para a solidificação da POA.

Para exemplificar o conceito de interesses transversais, utilizaremos um editor de figuras apresentado em Elrad et al. (2001). Neste exemplo, há duas classes concretas de elementos da figura, Linhas e Pontos, essas classes possuem boa modularidade e o código fonte de ambos é bem relacionado (coesão) e cada classe possui uma interface bem definida, porém, sempre que os elementos da figura são movimentados na tela todos os métodos de todos os elementos da figura precisam ser atualizados. A caixa vermelha (*DisplayUpdating*) na **figura 2** circula todos os métodos de todas as classes que precisam implementar esse Interesse (*concern*), nota-se que a responsabilidade por atualizar a tela não pertence a nenhuma das classes, mas atravessa ambas e por isso é transversal a essas classes.

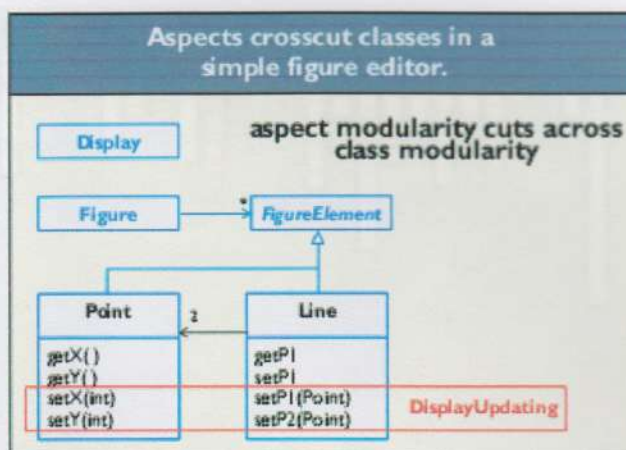


Figura 2: Interesse transversal de atualização da tela
Fonte: Elrad et al. (2001)

Usando apenas a POO a implementação desse *concern* tornaria o código espalhado entre as classes do sistema, usando a POA nesse caso, seria possível encapsular a implementação do *DisplayUpdating* em um aspecto.

2.3.3 Entrelaçamento (*code tangling*) e Espalhamento de Código (*code scattering*)

O entrelaçamento do código, ou *code tangling*, ocorre quando em um mesmo módulo (classe) há chamadas a vários interesses simultaneamente LADDAD (2003), por exemplo, em um software bancário pode existir que o procedimento de saque de valores, necessite de chamadas aos módulos de segurança, transações, validação dos saldos, etc.

Já o espalhamento do código, ou *code scattering*, ocorre quando um interesse tem de ser implementado em múltiplos módulos (Classes), ele fica espalhado por vários módulos do sistema não sendo parte intrínseca de nenhum deles, sendo que os interesses transversais são exatamente esse espalhamento de interesses do sistema por seus vários módulos LADDAD (2003).

Como exemplo de *code scattering* podemos analisar o mecanismo de *logging* do sistema *Apache Tomcat* um servidor web desenvolvido em Java, na figura 3 é possível ver o espalhamento do código (as linhas grifadas em vermelho) entre várias classes (as colunas) do sistema.

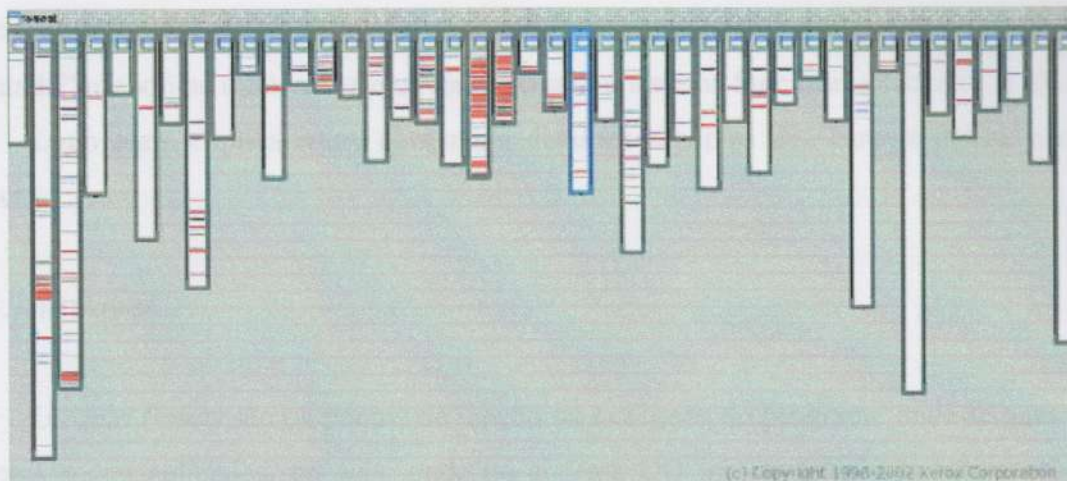


Figura 3: Espalhamento do código de logging entre as classes no TomCat
Fonte: LADDAD (2003)

Vários problemas decorrem do espalhamento e entrelaçamento de código em um sistema, como os próprios interesses transversais (*crosscutting concerns*), instruções nos métodos das classes afetadas com instruções que não pertencem ao domínio do sistema, redundância de código e dificuldade de compreensão e manutenção do sistema.

Alguns desses problemas podem ser resolvidos com uma melhor abordagem na análise e planejamento do projeto de software do sistema, porém de forma limitada e engessada, portanto a POA proposta em Kiczales et al. (1997) como uma técnica objetivando melhorar o suporte à modularização dos interesses transversais por meio de abstrações que possibilitem a separação e composição destes interesses na construção dos sistemas de software, vem sendo uma opção relevante na forma de resolver tais problemas.

2.4 ASPECTJ

O AspectJ Kiczales et al (2001) é uma ferramenta de desenvolvimento de sistemas orientados a aspectos, ela foi desenvolvida em 1997 pela *Xerox Palo Alto Research Center* e em 2002 foi incluída no Projeto Eclipse da IBM como um *plugin* acrescentando a linguagem as estruturas e ferramentas necessárias para a criação dos Aspectos do sistema.

Esta é uma ferramenta de propósito geral na criação dos aspectos, podendo também, alterar a estrutura estática de um sistema (Atributos, Construtores) através das *inter-types declarations* modificando assim a hierarquia do sistema, já a parte dinâmica (Métodos) são alterados em tempo de execução pelo compilador através dos conjuntos de *join Points*, selecionados pelos *pointcuts*, através de comportamentos (código) adicionais chamados de

*advice*s, logo depois esse sistema é compilado juntamente com o código orientado a objeto e “montado” usando-se um *weaver* e o resultado será o sistema funcional (Kiczales et al, 2001).

Logo mais explicaremos com mais detalhes algumas das características chave do AspectJ.

2.4.1 Join Points

Os *Join Points* são os pontos de junção na execução do programa onde se quer que os aspectos sejam aplicados, ou seja, onde no sistema OO irá ser feita uma interceptação do sistema para que o código do aspecto seja executado. Segundo (Gradecki; Lesiecki, 2003) os *Join Points* podem ser associados a:

- a) chamada de métodos
- b) execução de métodos
- c) chamada de construtores
- d) execução de inicialização
- e) execução de construtores
- f) execução de inicialização estática
- g) pré-inicialização de objetos
- h) inicialização de objetos
- i) referência a campos
- j) tratamento de exceções

	Característica
Target	Objeto em execução e instância do padrão tipo
Signature	Objeto de destino e instância do padrão tipo
Args	Os argumentos são instância do padrão tipo
Within	O código em execução está definido em padrão tipo

Fonte: Demetres de Souza (2007)

2.4.2 Advice

Advice é o código para ser executado em um ponto de junção que está sendo interceptado pelo conjunto de junção. Existem três maneiras de executar o *Advice*, antes, durante e depois (before, around e after).

2.4.2 Pointcuts

Eles são responsáveis por selecionar pontos de junção, ou seja, eles detectam em que ponto do programa os aspectos deverão interceptar, são aninhados através de operadores lógicos *e*, *ou* e *não* (&&, || e !) - (KISELEV, 2002) - para definir um pointcut utiliza-se construtores de AspectJ chamados designadores de pointcuts (pointcuts designators).

Podemos declarar um conjunto de junção semelhante a uma classe em Java, podendo da mesma maneira que atributos e métodos dessas classes, especificar um quantificador de acesso aos conjuntos de junção, podendo ser públicos, privados ou final, mas não podem ser sobrecarregados. Também podem ser declarados abstratos, mas somente dentro de aspectos abstratos, e ainda podem ser nomeados ou anônimos (KISELEV, 2002).

Um *pointcut designator* identifica o *pointcut* pelo nome ou por uma expressão, os termos *pointcut* e *pointcut designator* são usados frequentemente como sinônimos, pode ser declarado um *pointcut* dentro de um aspecto, classe ou interface.

Os principais tipos de designadores em AspectJ estão listados abaixo:

Quadro 1: Designadores usados no *pointcut*

Designador	Característica
Call	Invocação do método / construtor identificado por assinatura
Execution	Execução do método / construtor identificado por assinatura
Get	Acesso a atributo identificado por assinatura
Set	Atribuição do atributo identificado por assinatura
This	Objeto em execução é instância do padrão tipo
Target	Objeto de destino é instância do padrão tipo
Args	Os argumentos são instância do padrão tipo
Within	O código em execução está definido em padrão tipo

Fonte: Pesquisa de campo (2007)

2.4.3 Advice

Advice é o código para ser executado em um ponto de junção que está sendo referenciado pelo conjunto de junção. Existem três maneiras de executar os *Advice*: antes, durante e depois (before, around e after).

O *Advice* modifica a execução do programa no ponto de junção, substituindo seu código ou passando por ele, antes, antes e depois da execução normal do código OO ou, logo após a execução do mesmo.

O corpo de um *Advice* é muito semelhante ao de qualquer método, encapsulando a lógica a ser executada quando um ponto de junção é alcançado (GRADECKI; LESIECKI, 2003).

2.4.4 *Inter-type Declarations*

A estrutura estática de uma aplicação, pode ser alterada em AspectJ por meio das declarações inter-tipos que são descritas como interesses estáticos (*static crosscutting*). Estas declarações provêm uma construção chamada *Introduction*.

Introduction é um interesse estático que introduz alterações nas classes, interfaces e aspectos do sistema. Alterações estáticas em módulos não têm efeito direto no comportamento. Por exemplo, pode ser adicionado um método ou um atributo na classe (Gradecki and Lesiecki, 2003).

2.4.5 *Aspect*

O *Aspect* é a unidade central em AspectJ assim como a Classe é a base do desenvolvimento Java, possuindo características em comum entre si. Os *Aspects* encapsulam todas as características necessárias para a criação e execução do aspecto ou seja os *Join Points*, *Pointcuts*, *Advices* e *Inter-type Declarations* em uma unidade semelhante a uma classes java, o Aspecto.

Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação pontos de junção, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução (SOARES; BORBA, 2002).

3 METODOLOGIA

Quanto a Natureza

Pesquisa aplicada, com a finalidade de analisar e desenvolver os aspectos transversais de uma aplicação no cadastro de Projetos de Extensão da FARN utilizando o AspectJ.

Quanto aos Objetivos

Pesquisa Explicativa, tem como finalidade a análise e de características transversais de um sistema e seu posterior desenvolvimento, utilizando-se de ferramenta específica.

Quanto aos Procedimentos Técnicos

Pesquisa bibliográfica, que tem por finalidade levantar e analisar as contribuições de vários autores para o assunto abordado.

Pesquisa Experimental, pois será realizado o desenvolvimento e análise dos requisitos Transversais do sistema de cadastro de Projetos de Extensão da FARN.

4 DESENVOLVIMENTO

O sistema de cadastro de Projetos de Extensão da FARN, vem para solucionar o atual problema de armazenamento das informações referentes a esses projetos, que eram planejados e executados, mas cujas informações não ficavam guardadas.

Este sistema serve de exemplo, para demonstrarmos, com partes do código do sistema, como a orientação a aspectos funciona, como poderá ser implementado.

Com o sistema, essas informações além de serem armazenadas, podem ser melhor organizadas e recuperadas caso necessário, de forma mais ágil e eficiente, criando também uma padronização na forma como as informações são guardadas, e também maior agilidade no cadastro das mesmas.

4.1. DESCRIÇÃO DO SISTEMA

O sistema usa como base o cadastro das informações pertinentes a criação de um Projeto de Extensão da FARN, possui também cadastro de Alunos e Professores, que podem ser incluídos no projeto como colaboradores, e o cadastro de uma tabela Auxiliar que serve para armazenar informações extras sobre o projeto tais como: áreas de atuação, parcerias e cursos vinculados ao projeto.

O cadastro de Alunos e Professores possui informações básicas sobre essas entidades, tais como código, nome e matrícula/contrato além de informações sobre contatos e endereço completo.

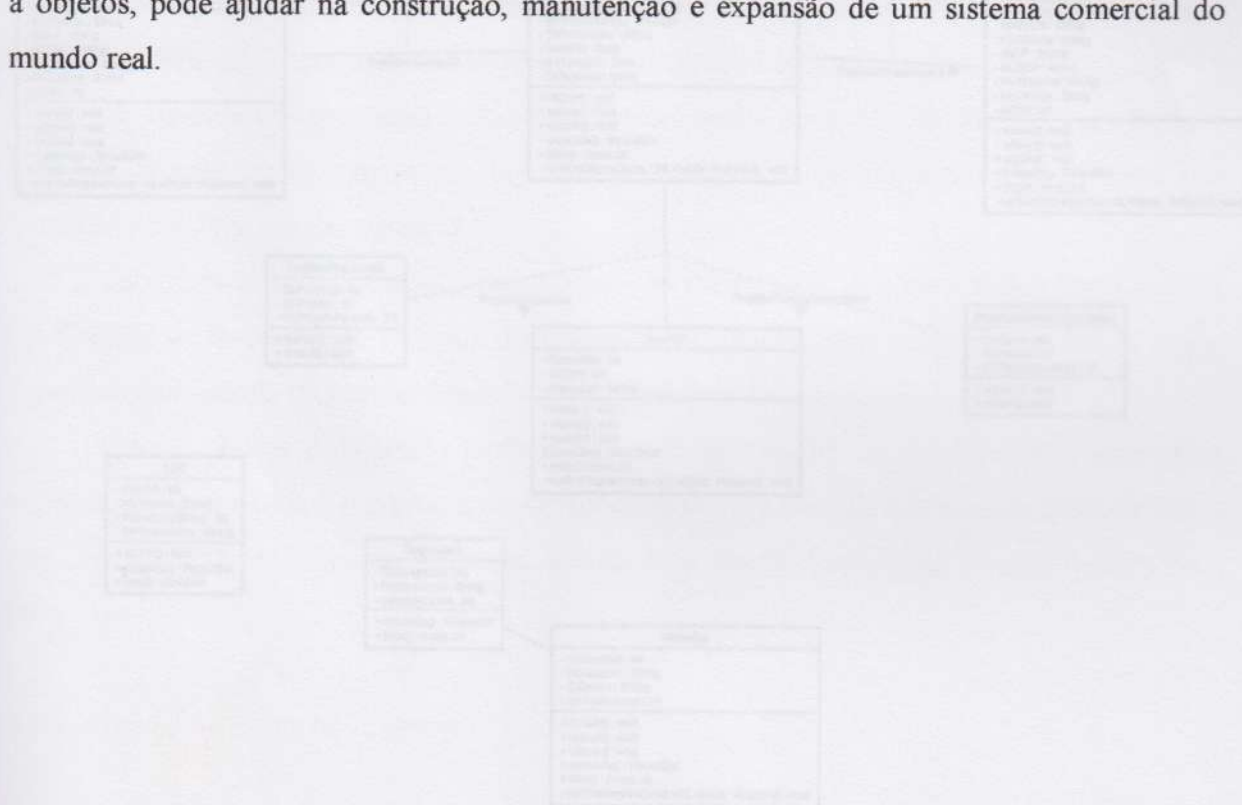
No cadastro de Projetos, além de informações básicas tais como: código, título, local de realização, entre outros, possui também um controle de todos os Professores e Alunos que estão colaborando com o projeto, além de identificar um Professor como coordenador, é possível também cadastrar todas as parcerias do Projeto, as áreas de atuação e a que cursos o Projeto está vinculado.

A tabela Auxiliar possui uma função mista, serve para cadastrar informações sobre áreas de atuação, parcerias e cursos que são vinculados ao projeto, como estas entidades servem apenas de referência, não foi detectada a necessidade de criarem-se entidades exclusivas para cada uma, sendo também essa abordagem vantajosa no sentido de atender a novas necessidades de informações de referência simples aos Projetos, sem a necessidade de se alterar a estrutura estática das tabelas.

O sistema possuirá um módulo de Segurança, onde os Usuários serão cadastrados e será atribuído a cada um, um nível de acesso ao sistema, ou seja, quais as operações que serão permitidas serem executadas pelo Usuário, servirá também de base para a autenticação no sistema, permitindo que apenas Usuários cadastrados acessem o mesmo.

O *logging* também será implementado no sistema, ele servirá como registro de todas as operações efetuadas no sistema, e armazenará as informações referentes a todas as 'classes, métodos e argumentos' que serão executados.

O sistema não possuirá uma GUI (Graphics User Interface, ou Interface Gráfica de Usuário) sendo voltado para linha de comando, uma vez que o escopo desse projeto é apenas exemplificar, usando um sistema funcional, como o uso de aspectos em um sistema orientado a objetos, pode ajudar na construção, manutenção e expansão de um sistema comercial do mundo real.



4.1.1. Diagrama de Classes do Sistema

Abaixo podemos ver o Diagrama de Classes UML do Sistema de Cadastro de Projetos de Extensão da FARN, onde podemos ver as classes usadas no sistema e seus relacionamentos, ele serve para termos uma visão geral do sistema e suas entidades.

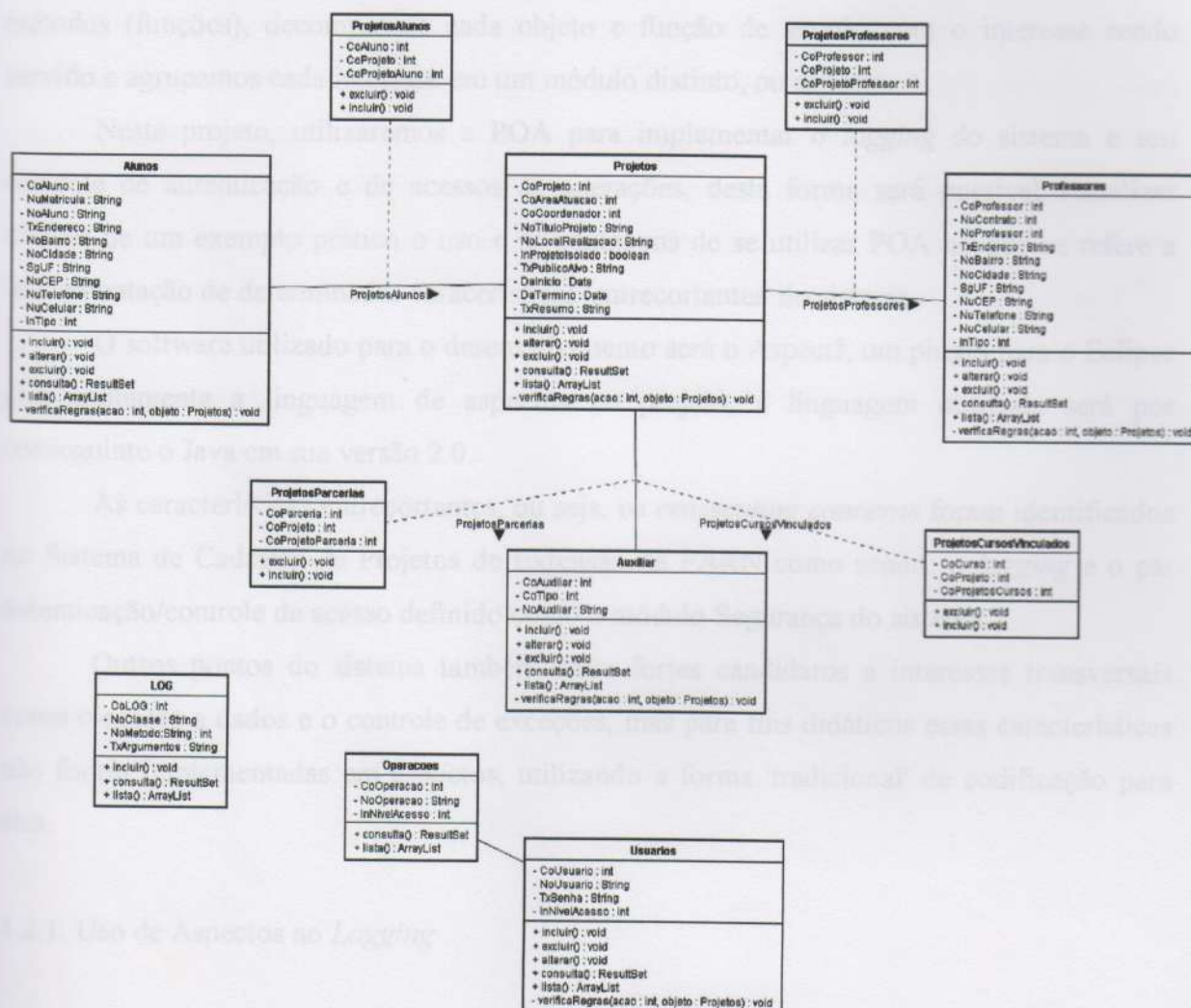


Figura 4: Diagrama de classes UML.
Fonte: O próprio autor (2007)

4.2. UTILIZAÇÃO DE ASPECTOS NO DESENVOLVIMENTO DO SISTEMA

O objetivo da POA Programação Orientada a Aspectos é encapsular interesses entrecortantes em módulos fisicamente separados do restante do código. Esses módulos são denominados aspectos. Pensando em termos abstratos, a orientação a aspectos introduz uma terceira dimensão de decomposição. Além de decompor o sistema em objetos (dados) e métodos (funções), decomparamos cada objeto e função de acordo com o interesse sendo servido e agrupamos cada interesse em um módulo distinto, ou aspecto.

Neste projeto, utilizaremos a POA para implementar o *logging* do sistema e seu controle de autenticação e de acessos as operações, desta forma será possível visualizar através de um exemplo prático o uso e as vantagens de se utilizar POA no que se refere a implementação de determinadas características 'entrecortantes' do sistema.

O software utilizado para o desenvolvimento será o AspectJ, um plugin para o Eclipse que implementa a linguagem de aspectos no projeto, a linguagem utilizada será por conseguinte o Java em sua versão 2.0.

As características entrecortantes, ou seja, os *crosscutting concerns* foram identificados no Sistema de Cadastro de Projetos de Extensão da FARN como sendo: o *logging* e o par autenticação/controle de acesso definido como o módulo Segurança do sistema.

Outros pontos do sistema também eram fortes candidatos a interesses transversais como o acesso a dados e o controle de exceções, mas para fins didáticos essas características não foram implementadas em aspectos, utilizando a forma 'tradicional' de codificação para eles.

4.2.1. Uso de Aspectos no *Logging*

O *logging* consiste em guardar registros das ações executadas por um programa para fins de análise posterior. Em geral não se sabe se o log será necessário, mas ele pode vir a ser importante para encontrar defeitos no sistema. *logging* é o exemplo canônico de POA. A forma mais simples de se mostrar as vantagens do paradigma é com um aspecto que intercepta todos os pontos de junção de chamadas de métodos e usa reflexão para mostrar informações sobre os métodos chamados.

Assim pode-se demonstrar como aspectos simples podem acrescentar comportamentos em diversos pontos do sistema simultaneamente, sem modificá-los.

4.3. OPERACIONALIZAÇÃO DO LOGGING

Como pode ser visto na **Figura 5**, o código do *logging* está 'entrelaçado' em todos os métodos da classe *Aluno*, não sendo necessariamente parte essencial da classe para funcionar.

```
//metodos CRUD
public void incluir() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.incluir(this);
    log("Aluno", "incluir", aluno);
}

public void alterar() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.alterar(this);
    log("Aluno", "alterar", aluno);
}

public void excluir() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.excluir(this);
    log("Aluno", "excluir", aluno);
}

public void consulta() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    ResultSet rs = aluno.consulta(this);

    rs.next();
    CoAluno = rs.getInt("CoAluno");
    NuMatricula = rs.getString("NuMatricula");
    NoAluno = rs.getString("NoAluno");
    ...
    log("Aluno", "consulta", aluno);
}

...

private void log(String classe, String metodo, Object argumentos) throws SQLException{
    Log log = new Log();

    log.setNoClasse(classe);
    log.setNoMetodo(metodo);
    log.setNoArgumentos(argumentos.toString());

    log.incluir();
}
```

Figura 5: Código do logging "entrelaçado" entre os métodos

Na **Figura 6** vemos que este mesmo interesse, encontra-se 'espalhado' por vários métodos de outras classes, ou seja, o *logging* é um bom candidato a se tornar um aspecto, uma vez que seu código encontra-se ligado a vários métodos e classes do sistema, que chamam o *logging*, mas não fazem parte de suas regras de negócio.

```

public class Alunos {
    ...
    public void incluir() throws SQLException{
        AlunosDAO aluno = new AlunosDAO();
        aluno.incluir(this);
        log("Alunos", "incluir", aluno);
    }
    ...
}

public class Professores {
    ...
    public void alterar() throws SQLException{
        ProfessoresDAO professor = new ProfessoresDAO();
        professor.alterar(this);
        log("Professores", "alterar", professor);
    }
    ...
}

public class Usuarios {
    ...
    public void excluir() throws SQLException{
        UsuariosDAO usuario = new UsuariosDAO();
        usuario.excluir(this);
        log("Usuarios", "excluir", usuario);
    }
    ...
}

```

Figura 6: Código do Logging “espalhado” entre as classes do sistema

Uma vez que o código de *logging* é separado em um aspecto, podemos ver que as classes e métodos que antes faziam chamadas a ele, não fazem mais parte do código.

```

//metodos CRUD
public void incluir() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.incluir(this);
}

public void alterar() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.alterar(this);
}

public void excluir() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    aluno.excluir(this);
}

public void consulta() throws SQLException{
    AlunosDAO aluno = new AlunosDAO();
    ResultSet rs = aluno.consulta(this);

    rs.next();
    CoAluno = rs.getInt("CoAluno");
    NuMatricula = rs.getString("NuMatricula");
    NoAluno = rs.getString("NoAluno");
    ...
}
...

```

Figura 7: Métodos da classe Alunos sem o código de Logging

```

public class Alunos {
    ...
    public void incluir() throws SQLException{
        AlunosDAO aluno = new AlunosDAO();
        aluno.incluir(this);
    }
    ...
}

public class Professores {
    ...
    public void alterar() throws SQLException{
        ProfessoresDAO professor = new ProfessoresDAO();
        professor.alterar(this);
    }
    ...
}

public class Usuarios {
    ...
    public void excluir() throws SQLException{
        UsuariosDAO usuario = new UsuariosDAO();
        usuario.excluir(this);
    }
    ...
}

```

Figura 8: Métodos de várias classes sem o código de Logging

Porém a funcionalidade continua, uma vez que o comportamento do *logging* é implementado pelo aspecto **Logging** que possui duas partes, a primeira é onde fica o *pointcut*, o ponto de corte que contém um conjunto de chamadas ao sistema, e serve para agrupar os locais onde essas chamadas serão interceptadas pelo aspecto, conforme a **Figura 9**.

Uma vez identificados os pontos de interceptação a segunda parte do aspecto é chamada, no caso o *advice*, que contém o código que será executado nesses pontos específicos do sistema, no nosso caso, as chamadas a classe de **Log** e a inclusão das informações necessárias.

```

public aspect Logging {
    pointcut todasChamadasAlunos():
        call(* AlunosDAO.*(..));
    before(): todasChamadasAlunos(){
        Signature sig = thisJoinPointStaticPart.getSignature();
        String classe = sig.getDeclaringType().getName();
        String metodo = sig.getName();

        Log log = new Log();
        log.setNoClasse(classe);
        log.setNoMetodo(metodo);
        log.setTxArgumentos(thisJoinPoint.getArgs());

        try {
            log.incluir();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Figura 9: Divisão do aspecto entre o pointcut e o advice

Analisando o código do *advice* na **Figura 10**, podemos notar duas características importantes, primeiro, o comportamento necessário para as funções de armazenamento dos dados do *logging*, continuam a cargo da classe **Log**, não precisando necessariamente ser implementado direto no aspecto, mantendo sua modularidade, e sendo o **Logging** apenas uma ponte entre a classe de **Log** e as demais classes do sistema.

O segundo ponto diz respeito a capacidade do aspecto de receber informações vindas das operações aos quais ela está atuando, no caso, podemos saber o nome do método, da classe, e até quais os argumentos foram passados para as operações, isso se chama 'Reflexão' ou 'Introspecção', que é a capacidade de um programa de investigar fatos sobre si próprio. É dessa forma que conseguimos os dados necessários para gravar o registro de todas as operações executadas no sistema.

```

public aspect Logging {
    ...
    before(): todasChamadasAlunos(){
        Signature sig = thisJoinPointStaticPart.getSignature();
        String classe = sig.getDeclaringType().getName();
        String metodo = sig.getName();

        Log log = new Log();

        log.setNoClasse(classe);
        log.setNoMetodo(metodo);
        log.setTxArgumentos(thisJoinPoint.getArgs());

        try {
            log.incluir();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

Reflexão

chamada a classe LOG com código Orientado a Objetos

Figura 10: Reflexão e chamada ao método da classe Log

4.4. AUTENTICAÇÃO E CONTROLE DE ACESSOS COM ASPECTOS

O módulo de segurança compreende as operações de autenticação de usuários e permissão de execução de seus métodos, de acordo com o nível de acesso que o usuário 'logado' tiver.

Da mesma forma que acontece com o *logging* as operações referentes a segurança do sistema também possuem o seu código espalhado e entrelaçado entre os vários métodos e classes da aplicação. Este aspecto é dividido em três partes com funções distintas: Na primeira parte encontra-se o código de autenticação de usuários que usa um método interno – **Figura 11** - ao aspecto (porém não é um *advice*) para criar a interface e pedir nome e senha dos

usuários. A segunda parte tem como objetivo garantir que apenas usuários que possuam níveis de acesso compatíveis com as operações possam executá-los. O último *advice* serve para inserir uma nova funcionalidade ao menu principal da aplicação, a possibilidade de mudança de usuários.

```
private void login() throws SQLException{
    boolean loginOK = false;
    int tentativasLogin = 0;

    try{
        do{
            String usuarioAux = "";
            String senhaAux = "";

            BufferedReader lerTeclado = null;
            lerTeclado = new BufferedReader(new InputStreamReader(System.in));

            System.out.println("-----");
            System.out.println("SISTEMA DE CADASTRO DE PROJETOS DE EXTENSÃO");
            System.out.println("-----");

            System.out.println("Login: ");
            usuarioAux = lerTeclado.readLine();

            System.out.println("Senha: ");
            senhaAux = lerTeclado.readLine();

            loginOK = controleAcesso.login(usuarioAux, senhaAux);
            tentativasLogin++;

            if(loginOK == false)
                System.out.println("Usuario ou Senha invalido");

        }while(loginOK == false && tentativasLogin < 4);
    }catch (IOException e) {
        System.out.println("Erro durante o Login");
    }
    ...
}
```

Figura 11: Código do aspecto Logging que não é um *advice*

O primeiro procedimento a ser executado é a interceptação do método que chama o menu principal na sua inicialização, o código pode ser visto na **Figura 12**, ele serve para que a autenticação dos usuários possam ser feitas antes que o usuário possa efetivamente usar o sistema. Esse *pointcut* age exclusivamente na chamada do método construtor e usa o método *login* para gerar a interface que pedirá o nome e a senha do usuário, para posterior comparação, e se for o caso, deixa o usuário entrar no sistema.

Esse aspecto possui uma particularidade, uma propriedade que indica se ele está ou não ativo, indicando se o sistema deverá ou não usá-lo, sem a necessidade de se re-compilar o sistema sem o aspecto, aumentando a sua plugabilidade, podendo esse parâmetro ser ativado diretamente no código, ou através de algum arquivo de configuração.

```

public aspect AutenticacaoAutorizacao {

    private boolean aspectAtivo = true;
    private ControleAcesso controleAcesso = null;

    pointcut construtoraSistemas(): call(Principal.new(..));

    before(): construtoraSistemas(){
        controleAcesso = new ControleAcesso();
        if(aspectAtivo == true)
            try {
                login();
            } catch (SQLException e) {
                e.printStackTrace();
            }
    }
    ...
}

```

Figura 12: Advice usado para inicializar o sistema com o Login

Como podemos ver na Figura 13, para cada chamada dos métodos da classe Aluno, existe uma chamada ao método **testarPermissão(operacao)** que verifica se o usuário tem ou não permissão para efetuar a operação.

```

...
//metodos CRUD
public void incluir() throws SQLException{

    ControleAcesso ca = new ControleAcesso();

    if(ca.testarPermissao("incluir") == true){
        AlunosDAO aluno = new AlunosDAO();
        aluno.incluir(this);
    }else
        System.out.println("Usuario nao pode acessar a operacao: Incluir");
}

public void alterar() throws SQLException{

    ControleAcesso ca = new ControleAcesso();

    if(ca.testarPermissao("alterar") == true){
        AlunosDAO aluno = new AlunosDAO();
        aluno.alterar(this);
    }else
        System.out.println("Usuario nao pode acessar a operacao: Alterar");
}

public void excluir() throws SQLException{

    ControleAcesso ca = new ControleAcesso();

    if(ca.testarPermissao("excluir") == true){
        AlunosDAO aluno = new AlunosDAO();
        aluno.excluir(this);
    }else
        System.out.println("Usuario nao pode acessar a operacao: Excluir");
}
...

```

Figura 13: Métodos da classe Alunos com o código para verificação de permissão

São justamente essas chamadas que serão interceptadas pelos pontos de corte do aspecto **AutenticacaoAutorizacao** e que serão tratado através do *advice* apropriado. Todas as

outras classes também possuem as mesmas chamadas, e cada um deles será referenciado através de *joinpoints* próprios, porém o *advice* é o mesmo para cada um.

```
...
pointcut execucao(): execution(private void CadastroAlunos.*());
void around(): execucao() {
    String metodoNome = thisJoinPoint.getSignature().getName();
    try {
        if (controleAcesso.testarPermissao(metodoNome) == true)
            proceed();
        else
            System.out.println("Usuario nao pode acessar a operacao: " + metodoNome);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}
...
```

Figura 14: Advice com o código de verificação de permissões de acesso

Observamos na **Figura 14** que esse *advice* é de um tipo diferente do usado no aspecto do *logging*, ele é do tipo *around*, isso quer dizer que a interceptação do código dar-se-á 'durante' a execução do mesmo, podendo ou não concluir a operação de acordo com as instruções dadas, no caso, se o usuário 'logado' tiver permissão de executar a operação, o método continuará normalmente, através da palavra-chave *proceed*, senão, uma mensagem informando que o usuário não tem acesso a operação será mostrado.

O último *advice* não possui *pointcut*, ele usa o *joinpoint* diretamente na sua chamada, ele intercepta o local onde a opção do menu principal é lida e pega o seu valor, se for 9 ele chama o método *login* do próprio aspecto e pede uma nova autenticação do usuário. Ou seja, ele adiciona a opção de troca de usuário a aplicação, opção essa que não existia originalmente, esse *advice* possui também um valor de retorno, que será usado no sistema para identificar qual opção do menu o usuário escolheu, seu código pode ser visto na **figura 15**.

```
...
int around(): call(int Integer.parseInt(String)) && within(Principal){
    int result = 0;
    result = proceed();
    if(result == 9)
        try {
            login();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    return result;
}
```

Figura 15: Inclui a funcionalidade de troca de usuário ao sistema

5 CONCLUSÃO

Como ficou demonstrado nesse trabalho o uso de aspectos no desenvolvimento de software traz vários benefícios ao sistema, tais como, modularidade, facilidade de evolução e reutilização. A POA não substitui a orientação a objetos, ela é uma tecnologia complementar e não funciona sem esta. As várias partes que compõem um programa orientado a aspectos ainda são implementados dentro do modelo orientado a objetos.

O uso de aspectos diminui as responsabilidades em cada parte, como interesses entrecortantes são separados em seus próprios módulos, as partes do programa que lidam com a lógica de negócios não ficam poluídas com código que lida com interesses periféricos, os módulos em POA não se chamam diretamente, há uma redução no nível de acoplamento. Com o uso da POA a evolução do sistema é facilitada, pois novos aspectos podem ser acrescentados facilmente sem a necessidade de alterar o código existente, necessitando apenas serem “montados” novamente com o *weaver* e os aspectos que se deseja incluir. Como o código não mistura interesses, aumentam-se as possibilidades de se reutilizar módulos em sistemas diferentes. Usando-se o Sistema de Cadastro de Projetos de Extensão da FARN como exemplo, poderíamos reaproveitar os cadastros de Professores e Alunos, e incluí-los em outro sistema, mesmo que ele não possuísse as funções de *login* e segurança, uma vez que o código estaria livre de chamadas a esses métodos.

Por outro lado a POA ainda não é uma técnica amplamente difundida, e ainda não possui técnicas claras para a sua análise e decomposição, mas já conta com parcerias de peso como a IBM no seu desenvolvimento, além de um grupo de usuário que aumenta a cada dia, existe algumas desvantagens no uso de aspectos, como um módulo não chama outro diretamente, é difícil inferir o comportamento do sistema como um todo avaliando-se módulos individuais, e a quebra do encapsulamento dos objetos é visto como perigoso por alguns profissionais.

A POA não tenta resolver problemas não solucionados. Há outras soluções para os problemas de espalhamento e entrelaçamento que não envolvem a criação de novas tecnologias, como: padrões de projeto, e o uso de interfaces abstratas, mesmo assim ainda é necessário que o restante do código chame outros métodos. Ou seja, o problema é diminuído, mas não resolvido, sendo portanto a Programação Orientada a Objetos uma tecnologia de futuro.

REFERENCIAS

- CHAVES, Rafael Alves. **Aspectos e Midlware**. 2002. 50f. Trabalho Individual submetido à Universidade Federal de Santa Catarina como parte dos requisitos para obtenção do grau de Mestre em Ciências da Computação, Florianópolis, 2002.
- DIJKSTRA, Edsger W. **A Discipline of Programming**. Prentice Hall, 1976.
- ELRAD, T.; KICZALES, G.; AKSIT, M.; LIEBERHER, k.; OSSHER, H. Discussing Aspects of AOP. **Communications of the ACM**, v. 44, n. 10, p. 33-38, 2001.
- ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming. **Communications of the ACM**, v. 44, n. 10, p. 29-32, 2001.
- GRADECK, Joe; LESIECKI, Nicolas. **Mastering AspectJ: aspect-oriented programming in Java**. Indianapolis, Indiana. Wiley, 2003, p. 453.
- KISELEV, I. **Aspect – Oriented Programming with AspectJ**. Ed. Sams Publishing, 2002.
- LADDAD, Ramnivas. **AspectJ in Action: Practical Aspect-Oriented Programming**. Ed. Mannig, Canada, [2003].
- PIVETA E. K. et al. Termos em português para Desenvolvimento de Software Orientado a Aspectos. In: WORKSHOP BRASILEIRO DE DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS (WASP'04), 2004
- SILVA, Lyrene Fernandes da. **Uma Estratégia Orientada a Aspectos para Modelagem de Requisitos**. 2006. Tese (Doutorado em Informática) - PUC-Rio, Rio de Janeiro, 2006
- SOARES, S.; BORBA, P. AspectJ - Programação orientada a aspectos em Java, In: SIMPÓSIO BRASILEIRO DE LINGUAGENS DE PROGRAMAÇÃO, 6, 2002, Rio de Janeiro, 2002.
- SOARES, S.; BORBA, P. Progressive implementation with aspect-oriented programming. In Verlag, S., editor, The 12th Workshop for PhD Students in Object-Oriented Systems, Malaga, Spain. To appear.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W.G. An overview of AspectJ, In Knudsen, J. L., editor, proceedings of the European Conference on Object-Oriented Programming (ECOOP), Berlin, Springer-Verlag, 2001.

KICZALES, G. et al. **Aspect-Oriented Programming**. Proceedings European Conference on Object-Oriented Programming, 1997.

WINCK, Diogo Vinícios; GOETTEN JUNIOR, Vicente. **AspectJ: Programação Orientada a Aspectos Com Java**. São Paulo: novatec, 2006

APÊNDICE A – Código do sistema

```

package apresentacao;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.SQLException;
import java.util.ArrayList;

import negocio.Alunos;

public class CadastroAlunos {

    private BufferedReader lerTeclado = null;

    public CadastroAlunos() throws IOException {
        lerTeclado = new BufferedReader(new
InputStreamReader(System.in));
    }

    public void menu() throws IOException {

        String valorLido;
        int opcao = -1;

        do{

            System.out.println("-----
-----");
            System.out.println("SISTEMA DE CADASTRO DE PROJETOS DE
EXTENSÃO");
            System.out.println("-----");

            System.out.println("Escolha a operação desejada");
            System.out.println("-----");
            System.out.println("1 - Incluir Aluno");
            System.out.println("2 - Editar Aluno");
            System.out.println("3 - Excluir Aluno");
            System.out.println("4 - Consultar Aluno");
            System.out.println("5 - Listar Alunos");
            System.out.println("0 - Voltar para o Menu Principal");
            valorLido = lerTeclado.readLine();
            opcao = Integer.parseInt(valorLido);

            switch (opcao) {
                case 1: incluir(); break;
                case 2: alterar(); break;
                case 3: excluir(); break;
                case 4: consulta(); break;
                case 5: lista(); break;
                case 0: ; break;
            }

        }while(opcao != 0);

    }

    private void carregadados(Alunos aluno){

```

```

String valorLido;

try{
    System.out.println("Matricula: ");
    valorLido = lerTeclado.readLine();
    aluno.setNuMatricula(valorLido);

    System.out.println("Nome do Aluno: ");
    valorLido = lerTeclado.readLine();
    aluno.setNoAluno(valorLido);

    System.out.println("Endereco: ");
    valorLido = lerTeclado.readLine();
    aluno.setTxEndereco(valorLido);

    System.out.println("Bairro: ");
    valorLido = lerTeclado.readLine();
    aluno.setNoBairro(valorLido);

    System.out.println("Cidade: ");
    valorLido = lerTeclado.readLine();
    aluno.setNoCidade(valorLido);

    System.out.println("UF: ");
    valorLido = lerTeclado.readLine();
    aluno.setSgUF(valorLido);

    System.out.println("CEP: ");
    valorLido = lerTeclado.readLine();
    aluno.setNuCEP(valorLido);

    System.out.println("Telefone: ");
    valorLido = lerTeclado.readLine();
    aluno.setNuTelefone(valorLido);

    System.out.println("Celular: ");
    valorLido = lerTeclado.readLine();
    aluno.setNuCelular(valorLido);

    System.out.println("Tipo (0;1): ");
    valorLido = lerTeclado.readLine();
    aluno.setInTipo(Byte.parseByte(valorLido));
}
catch(IOException e){
    e.printStackTrace();
}

}

private void acessoDados(Alunos aluno){

    System.out.println("Código....:" + aluno.getCoAluno());
    System.out.println("Matricula.:" + aluno.getNuMatricula());
    System.out.println("Nome.....:" + aluno.getNoAluno());
    System.out.println("Tipo.....:" + aluno.getInTipo() + "\n");

}

private void incluir() throws IOException{

```

```
try{
    Alunos a = new Alunos();

    System.out.println("INCLUSÃO DE ALUNO\n");
    carregadados(a);

    a.incluir();
    System.out.println("Aluno Incluido com Sucesso");

    System.out.println("\nPressione ENTER para continuar\n");
    String valorLido = lerTeclado.readLine();

}
catch(SQLException e){
    e.printStackTrace();
}

private void alterar() throws IOException{

    try{
        Alunos a = new Alunos();

        System.out.println("ALTERAÇÃO DE ALUNO\n");

        System.out.println("Código do Aluno: ");
        String valorLido = lerTeclado.readLine();
        a.setCoAluno(Integer.parseInt(valorLido));

        carregadados(a);

        a.alterar();
        System.out.println("Aluno Alterado com Sucesso");

        System.out.println("\nPressione ENTER para continuar\n");
        valorLido = lerTeclado.readLine();

    }
    catch(SQLException e){
        e.printStackTrace();
    }
}

private void excluir(){
    String valorLido;

    try{

        System.out.println("EXCLUSÃO DE ALUNO");

        Alunos a = new Alunos();

        System.out.println("Código do Aluno: ");
        valorLido = lerTeclado.readLine();
        a.setCoAluno(Integer.parseInt(valorLido));

        System.out.println("Confirma Exclusão do Aluno?(S/N)");
```

```

        valorLido = lerTeclado.readLine();

        if(valorLido.equals("s")){
            a.excluir();
            System.out.println("Aluno Excluido com Sucesso");
        }else{
            System.out.println("Operação cancelada pelo
usuário");
        }

        System.out.println("\nPressione ENTER para continuar\n");
        valorLido = lerTeclado.readLine();

    }
    catch(IOException e){
        e.printStackTrace();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

private void consulta() throws IOException {
    try{
        Alunos a = new Alunos();

        System.out.println("CONSULTA ALUNO\n");

        System.out.println("Código do Aluno: ");
        String valorLido = lerTeclado.readLine();
        a.setCoAluno(Integer.parseInt(valorLido));

        a.consulta();

        acessoDados(a);

        System.out.println("\nPressione ENTER para continuar\n");
        valorLido = lerTeclado.readLine();
    }
    catch(SQLException e){
        e.printStackTrace();
    }
}

private void lista() throws IOException{
    try{
        Alunos a = new Alunos();

        System.out.println("CoAluno: ");
        String valorLido = lerTeclado.readLine();
        a.setCoAluno(Integer.parseInt(valorLido));

        ArrayList<Alunos> listagem = new ArrayList<Alunos>();

        listagem = a.lista();
    }
}

```

```

        for(Alunos aluno : listagem){
            acessoDados(aluno);
        }

        System.out.println("\nPressione ENTER para continuar\n");
        valorLido = lerTeclado.readLine();

    }
    catch (SQLException e) {
        e.printStackTrace();
    }
}

}

package apresentacao;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Principal {

    private BufferedReader lerTeclado = null;

    public Principal(){
        lerTeclado = new BufferedReader(new
        InputStreamReader(System.in));
    }

    public static void main(String[] args) throws IOException {
        Principal executor = new Principal();
        executor.menuPrincipal();
    }

    private void menuPrincipal() throws IOException {

        String valorLido;
        int opcao = -1;

        do{
            System.out.println("-----");
            System.out.println("SISTEMA DE CADASTRO DE PROJETOS DE
EXTENSÃO");
            System.out.println("-----");
            System.out.println("Escolha a operação desejada");
            System.out.println("-----");
            System.out.println("1 - Cadastro de Alunos");
            System.out.println("2 - Cadastro de Professores");
            System.out.println("3 - Cadastro de Projetos");
            System.out.println("4 - Cadastro de Usuários");
            System.out.println("5 - Tabela Auxiliar");
            System.out.println("0 - Sair");
            valorLido = lerTeclado.readLine();
            opcao = Integer.parseInt(valorLido);

            switch (opcao) {
                case 1:

```

```

        CadastroAlunos aluno = new CadastroAlunos();
        aluno.menu();
        break;
    }
}while(opcao != 0);
}
}

package aspectos;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.SQLException;

import negocio.ControleAcesso;
import apresentacao.*;

public aspect AutenticacaoAutorizacao {

    private boolean aspectAtivo = true;
    private ControleAcesso controleAcesso = null;

    pointcut construtoraSistemas(): call(Principal.new(..));

    before(): construtoraSistemas(){
        controleAcesso = new ControleAcesso();
        if(aspectAtivo == true)
            try {
                login();
            } catch (SQLException e) {
                e.printStackTrace();
            }
    }

    private void login() throws SQLException{

        boolean loginOK = false;
        int tentativasLogin = 0;

        try{
            do{
                String usuarioAux = "";
                String senhaAux = "";

                BufferedReader lerTeclado = null;
                lerTeclado = new BufferedReader(new
InputStreamReader(System.in));

                System.out.println("-----");
                System.out.println("SISTEMA DE CADASTRO DE PROJETOS
DE EXTENSÃO");
                System.out.println("-----");
            }while(!loginOK && tentativasLogin < 4);
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

```

package testes;

import java.sql.*;
import org.aspectj.lang.*;
import org.aspectj.lang.annotation.*;
import org.springframework.test.annotation.*;
import org.springframework.test.context.*;

import javax.swing.*;

public class Testes {

    System.out.println("Login: ");
    usuarioAux = lerTeclado.readLine();

    System.out.println("Senha: ");
    senhaAux = lerTeclado.readLine();

    loginOK = controleAcesso.login(usuarioAux,
    senhaAux);

    tentativasLogin++;

    if(loginOK == false)
        System.out.println("Usuario ou Senha
    invalido");

    while(loginOK == false && tentativasLogin < 4);
    }
    catch (IOException e) {
        System.out.println("Erro durante o Login");
    }

    if(loginOK == false){
        System.out.println("Numero maximo de tentativas
    excedido");
        System.exit(1);
    }else{
        System.out.println("para trocar de Usuario, digite 9 no
    Menu");
    }
}

@pointcut execucao(): execution(private void CadastroAlunos.*());

void around(): execucao(){
    String metodoNome = thisJoinPoint.getSignature().getName();
    try {
        if(controleAcesso.testarPermissao(metodoNome) == true)
            proceed();
        else
            System.out.println("Usuario nao pode acessar a
    operacao: " + metodoNome);
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

int around(): call(int Integer.parseInt(String)) &&
within(Principal){
    int result = 0;
    result = proceed();
    if(result == 9)
        try {
            login();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    return result;
}
}

```

```

package aspectos;

import java.sql.SQLException;

import org.aspectj.lang.Signature;

import dados.*;
import negocio.Log;

public aspect Logging {

    pointcut todasChamadasAlunos():
        call(* AlunosDAO.*(..));

    before(): todasChamadasAlunos() {

        Signature sig = thisJoinPointStaticPart.getSignature();

        String classe = sig.getDeclaringType().getName();
        String metodo = sig.getName();

        Log log = new Log();
        log.setNoClasse(classe);
        log.setNoMetodo(metodo);
        log.setTxArgumentos(thisJoinPoint.getArgs());

        try {
            log.incluir();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

package dados;

import java.sql.*;

/**
 *
 * @author vladimirf
 */
public class AcessoDados {

    public static Connection con;

    public static Connection getConnection() {

        if (con == null) {
            // Caso a conexao ainda não tenha sido aberta
            try {
                Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                con =
                DriverManager.getConnection("jdbc:odbc:Educacional","postgres","3161885");
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}

```

```

        return con;
    }
}

package dados;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import negocio.Alunos;

public class AlunosDAO {

    public void incluir(Alunos aluno) throws SQLException{

        String SQL = "INSERT INTO tbalunos(" +
            "NuMatricula, NoAluno, TxEndereco, NoBairro,"
            + "NoCidade, SgUF, NuCEP, NuTelefone,
            NuCelular, InTipo) " +
            "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(1, aluno.getNuMatricula());
        stm.setString(2, aluno.getNoAluno());
        stm.setString(3, aluno.getTxEndereco());
        stm.setString(4, aluno.getNoBairro());
        stm.setString(5, aluno.getNoCidade());
        stm.setString(6, aluno.getSgUF());
        stm.setString(7, aluno.getNuCEP());
        stm.setString(8, aluno.getNuTelefone());
        stm.setString(9, aluno.getNuCelular());
        stm.setByte(10, aluno.getInTipo());

        stm.execute();
        stm.close();
    }

    public void alterar(Alunos aluno) throws SQLException{

        String SQL = "UPDATE tbalunos " +
            "SET NuMatricula = ?, NoAluno = ?, TxEndereco
            = ?, NoBairro = ?, " +
            "NoCidade = ?, SgUF = ?, NuCEP = ?,
            NuTelefone = ?, " +
            "NuCelular = ?, InTipo = ? " +
            "WHERE CoAluno = ?";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(1, aluno.getNuMatricula());
        stm.setString(2, aluno.getNoAluno());
        stm.setString(3, aluno.getTxEndereco());
        stm.setString(4, aluno.getNoBairro());

```

```

package test;
import java.sql.*;
import java.util.*;

public class Alunos {

    public void cadastrar(Alunos aluno) throws SQLException{
        String SQL = "INSERT INTO tbAlunos (" +
            "NoCidade, SgUF, NuCEP, NuTelefone, NuCelular, InTipo, CoAluno" +
            ") VALUES (" +
            ? + ", ? + ", ? + ", ? + ", ? + ", ? + ", ?";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(5, aluno.getNoCidade());
        stm.setString(6, aluno.getSgUF());
        stm.setString(7, aluno.getNuCEP());
        stm.setString(8, aluno.getNuTelefone());
        stm.setString(9, aluno.getNuCelular());
        stm.setByte(10, aluno.getInTipo());
        stm.setInt(11, aluno.getCoAluno());

        stm.execute();
        stm.close();
    }

    public void excluir(Alunos aluno) throws SQLException{
        String SQL = "DELETE FROM tbAlunos " +
            "WHERE CoAluno = ?";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setInt(1, aluno.getCoAluno());

        stm.execute();
        stm.close();
    }

    public ResultSet consulta(Alunos aluno) throws SQLException{
        String SQL = "SELECT * FROM tbAlunos " +
            "WHERE CoAluno = ?";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setInt(1, aluno.getCoAluno());

        ResultSet rs = stm.executeQuery();

        return rs;
    }

    public ResultSet lista(Alunos aluno) throws SQLException{
        String SQL = "SELECT * FROM tbAlunos " +
            "WHERE CoAluno >= ?";

        PreparedStatement stm =
            AcessoDados.getConnection().prepareStatement(SQL);

        stm.setInt(1, aluno.getCoAluno());

        ResultSet rs = stm.executeQuery();

        return rs;
    }
}

```

```
package dados;

import java.sql.PreparedStatement;
import java.sql.SQLException;
import negocio.Log;

public class LogDAO {

    public void incluir(Log log) throws SQLException{

        String SQL = "INSERT INTO tbLOG " +
            "(NoClasse, NoMetodo, TxArgumentos) " +
            "VALUES (?, ?, ?)";

        PreparedStatement stm
        AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(1, log.getNoClasse());
        stm.setString(2, log.getNoMetodo());
        stm.setString(3, log.getTxArgumentos());

        stm.execute();
        stm.close();

    }
}
```

```
package dados;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import negocio.Operacoes;

public class OperacoesDAO {

    public ResultSet consulta(Operacoes operacao) throws SQLException{

        String SQL = "SELECT * FROM tbOperacoes " +
            "WHERE NoOperacao = ?";

        PreparedStatement stm
        AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(1, operacao.getNoOperacao());

        ResultSet rs = stm.executeQuery();

        return rs;
    }
}
```

```

package dados;

import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import negocio.Usuarios;

public class UsuariosDAO {

    public ResultSet consulta(Usuarios usuario) throws SQLException{

        String SQL = "SELECT * FROM tbUsuarios " +
                     "WHERE NoUsuario = ?";

        PreparedStatement stm
        AcessoDados.getConnection().prepareStatement(SQL);

        stm.setString(1, usuario.getNoUsuario());

        ResultSet rs = stm.executeQuery();

        return rs;
    }
}

```

```

package negocio;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import dados.AlunosDAO;

public class Alunos {

    private int CoAluno;
    private String NuMatricula;
    private String NoAluno;
    private String TxEndereco;
    private String NoBairro;
    private String NoCidade;
    private String SqUF;
    private String NuCEP;
    private String NuTelefone;
    private String NuCelular;
    private byte InTipo;

    //metodos getters/setters
    public int getCoAluno() {
        return CoAluno;
    }

    public void setCoAluno(int coAluno) {
        CoAluno = coAluno;
    }

    public byte getInTipo() {

```

```

        return InTipo;
    }

    public void setInTipo(byte inTipo) {
        InTipo = inTipo;
    }

    public String getNoAluno() {
        return NoAluno;
    }

    public void setNoAluno(String noAluno) {
        NoAluno = noAluno;
    }

    public String getNoBairro() {
        return NoBairro;
    }

    public void setNoBairro(String noBairro) {
        NoBairro = noBairro;
    }

    public String getNoCidade() {
        return NoCidade;
    }

    public void setNoCidade(String noCidade) {
        NoCidade = noCidade;
    }

    public String getNuCelular() {
        return NuCelular;
    }

    public void setNuCelular(String nuCelular) {
        NuCelular = nuCelular;
    }

    public String getNuCEP() {
        return NuCEP;
    }

    public void setNuCEP(String nuCEP) {
        NuCEP = nuCEP;
    }

    public String getNuMatricula() {
        return NuMatricula;
    }

    public void setNuMatricula(String nuMatricula) {
        NuMatricula = nuMatricula;
    }

    public String getNuTelefone() {
        return NuTelefone;
    }

    public void setNuTelefone(String nuTelefone) {
        NuTelefone = nuTelefone;
    }

```

```

    }

    public String getSgUF() {
        return SgUF;
    }

    public void setSgUF(String sgUF) {
        SgUF = sgUF;
    }

    public String getTxEndereco() {
        return TxEndereco;
    }

    public void setTxEndereco(String txEndereco) {
        TxEndereco = txEndereco;
    }

    //metodos CRUD
    public void incluir() throws SQLException{
        AlunosDAO aluno = new AlunosDAO();
        aluno.incluir(this);
    }

    public void alterar() throws SQLException{
        AlunosDAO aluno = new AlunosDAO();
        aluno.alterar(this);
    }

    public void excluir() throws SQLException{
        AlunosDAO aluno = new AlunosDAO();
        aluno.excluir(this);
    }

    public void consulta() throws SQLException{

        AlunosDAO aluno = new AlunosDAO();
        ResultSet rs = aluno.consulta(this);

        rs.next();
        CoAluno = rs.getInt("CoAluno");
        NuMatricula = rs.getString("NuMatricula");
        NoAluno = rs.getString("NoAluno");
        TxEndereco = rs.getString("TxEndereco");
        NoBairro = rs.getString("NoBairro");
        NoCidade = rs.getString("NoCidade");
        SgUF = rs.getString("SgUF");
        NuCEP = rs.getString("NuCEP");
        NuTelefone = rs.getString("NuTelefone");
        NuCelular = rs.getString("NuCelular");
        InTipo = rs.getBytes("InTipo");
    }

    public ArrayList lista() throws SQLException{

        AlunosDAO aluno = new AlunosDAO();
        ResultSet rs = aluno.lista(this);

        ArrayList<Alunos> listagem = new ArrayList<Alunos>();
    }

```

```

package negocio;

while(rs.next()){

    Alunos a = new Alunos();

    a.setCoAluno(rs.getInt("CoAluno"));
    a.setNuMatricula(rs.getString("NuMatricula"));
    a.setNoAluno(rs.getString("NoAluno"));
    a.setTxEndereco(rs.getString("TxEndereco"));
    a.setNoBairro(rs.getString("NoBairro"));
    a.setNoCidade(rs.getString("NoCidade"));
    a.setSgUF(rs.getString("SgUF"));
    a.setNuCEP(rs.getString("NuCEP"));
    a.setNuTelefone(rs.getString("NuTelefone"));
    a.setNuCelular(rs.getString("NuCelular"));
    a.setInTipo(rs.getBytes("InTipo"));

    listagem.add(a);
}

return listagem;
}

}

package negocio;

public class Auxiliar {

    private int CoAuxiliar;
    private int CoTipo;
    private String NoAuxiliar;

    public int getCoAuxiliar() {
        return CoAuxiliar;
    }

    public int getCoTipo() {
        return CoTipo;
    }

    public String getNoAuxiliar() {
        return NoAuxiliar;
    }

    public void setCoAuxiliar(int coAuxiliar) {
        CoAuxiliar = coAuxiliar;
    }

    public void setCoTipo(int coTipo) {
        CoTipo = coTipo;
    }

    public void setNoAuxiliar(String noAuxiliar) {
        NoAuxiliar = noAuxiliar;
    }

}

```

```

package negocio;

import java.lang.reflect.Method;
import java.sql.SQLException;

public class ControleAcesso {

    private String usuario = "";
    private int nivelAcesso = 0;

    public int getNivelAcesso() {
        return nivelAcesso;
    }

    public String getUsuario() {
        return usuario;
    }

    public boolean login(String usuario, String senha) throws
SQLException{

        Usuarios user = new Usuarios();

        user.setNoUsuario(usuario);
        user.consulta();

        String userAux = user.getNoUsuario();
        String senhAux = user.getTxSenha();

        if (userAux.equals(usuario) == true && senhAux.equals(senha) ==
true){
            usuario = userAux;
            nivelAcesso = (user.getInNivelAcesso());
            return true;
        }
        return false;
    }

    public boolean testarPermissao(String operacao) throws SQLException{

        Operacoes op = new Operacoes();

        op.setNoOperacao(operacao);
        op.consulta();

        String operacaoAux = op.getNoOperacao();

        if(operacaoAux.equals(operacao) == true)
            return nivelAcesso >= op.getInNivelAcesso();
        return true;
    }

    public boolean testarPermissao(Method operacao) throws SQLException{
        return testarPermissao(operacao.getName());
    }

}

```

```

package negocio;

import java.sql.SQLException;
import dados.LogDAO;

public class Log {

    private int CoLOG;
    private String NoClasse;
    private String NoMetodo;
    private String TxArgumentos;

    public int getCoLOG() {
        return CoLOG;
    }

    public void setCoLOG(int coLOG) {
        CoLOG = coLOG;
    }

    public String getNoClasse() {
        return NoClasse;
    }

    public void setNoClasse(String noClasse) {
        NoClasse = noClasse;
    }

    public String getNoMetodo() {
        return NoMetodo;
    }

    public void setNoMetodo(String noMetodo) {
        NoMetodo = noMetodo;
    }

    public String getTxArgumentos() {
        return TxArgumentos;
    }

    public void setTxArgumentos(String txArgumentos) {
        TxArgumentos = txArgumentos;
    }

    public void setTxArgumentos(Object argumentos[]) {
        String arg = "";
        for(int i = 0; argumentos != null && i < argumentos.length;
i++){
            arg = arg + argumentos[i].toString();
        }
        setTxArgumentos(arg);
    }

    //metodos CRUD
    public void incluir() throws SQLException{
        LogDAO log = new LogDAO();
        log.incluir(this);
    }

```

```

}

package negocio;

import java.sql.ResultSet;
import java.sql.SQLException;

import dados.OperacoesDAO;

public class Operacoes {

    private int CoOperacao;
    private String NoOperacao;
    private int InNivelAcesso;

    public int getCoOperacao() {
        return CoOperacao;
    }
    public void setCoOperacao(int coOperacao) {
        CoOperacao = coOperacao;
    }
    public int getInNivelAcesso() {
        return InNivelAcesso;
    }
    public void setInNivelAcesso(int inNivelAcesso) {
        InNivelAcesso = inNivelAcesso;
    }
    public String getNoOperacao() {
        return NoOperacao;
    }
    public void setNoOperacao(String noOperacao) {
        NoOperacao = noOperacao;
    }

    public void consulta() throws SQLException{

        OperacoesDAO operacao = new OperacoesDAO();
        ResultSet rs = operacao.consulta(this);

        rs.next();
        CoOperacao = rs.getInt("CoOperacao");
        NoOperacao = rs.getString("NoOperacao");
        InNivelAcesso = rs.getInt("InNivelAcesso");
    }
}

```

```
package negocio;
```

```
public class Professores {
```

```
    private int CoProfessor;  
    private String NuContrato;  
    private String NoProfessor;  
    private String TxEndereco;  
    private String NoBairro;  
    private String NoCidade;  
    private String SgUF;  
    private String NuCEP;  
    private String NuTelefone;  
    private String NuCelular;  
    private byte InTipo;
```

```
    public int getCoProfessor() {  
        return CoProfessor;  
    }
```

```
    public byte getInTipo() {  
        return InTipo;  
    }
```

```
    public String getNoBairro() {  
        return NoBairro;  
    }
```

```
    public String getNoCidade() {  
        return NoCidade;  
    }
```

```
    public String getNoProfessor() {  
        return NoProfessor;  
    }
```

```
    public String getNuCelular() {  
        return NuCelular;  
    }
```

```
    public String getNuCEP() {  
        return NuCEP;  
    }
```

```
    public String getNuContrato() {  
        return NuContrato;  
    }
```

```
    public String getNuTelefone() {  
        return NuTelefone;  
    }
```

```
    public String getSgUF() {  
        return SgUF;  
    }
```

```
    public String getTxEndereco() {  
        return TxEndereco;  
    }
```

```

public void setCoProfessor(int coProfessor) {
    CoProfessor = coProfessor;
}

public void setInTipo(byte inTipo) {
    InTipo = inTipo;
}

public void setNoBairro(String noBairro) {
    NoBairro = noBairro;
}

public void setNoCidade(String noCidade) {
    NoCidade = noCidade;
}

public void setNoProfessor(String noProfessor) {
    NoProfessor = noProfessor;
}

public void setNuCelular(String nuCelular) {
    NuCelular = nuCelular;
}

public void setNuCEP(String nuCEP) {
    NuCEP = nuCEP;
}

public void setNuContrato(String nuContrato) {
    NuContrato = nuContrato;
}

public void setNuTelefone(String nuTelefone) {
    NuTelefone = nuTelefone;
}

public void setSgUF(String sgUF) {
    SgUF = sgUF;
}

public void setTxEndereco(String txEndereco) {
    TxEndereco = txEndereco;
}

}

package negocio;

import java.util.Date;

public class Projetos {

    private int CoProjeto;
    private int CoAreaAtuacao;
    private int CoCoordenador;
    private String NoTituloProjeto;
    private String NoLocalRealizacao;
    private byte InProjetoIsolado;
    private String TxPublicoAlvo;
    private Date DaInicio;

```

```

private Date DaTermino;
private String TxResumo;

public int getCoAreaAtuacao() {
    return CoAreaAtuacao;
}

public int getCoCoordenador() {
    return CoCoordenador;
}

public int getCoProjeto() {
    return CoProjeto;
}

public Date getDaInicio() {
    return DaInicio;
}

public Date getDaTermino() {
    return DaTermino;
}

public byte getInProjetoIsolado() {
    return InProjetoIsolado;
}

public String getNoLocalRealizacao() {
    return NoLocalRealizacao;
}

public String getNoTituloProjeto() {
    return NoTituloProjeto;
}

public String getTxPublicoAlvo() {
    return TxPublicoAlvo;
}

public String getTxResumo() {
    return TxResumo;
}

public void setCoAreaAtuacao(int coAreaAtuacao) {
    CoAreaAtuacao = coAreaAtuacao;
}

public void setCoCoordenador(int coCoordenador) {
    CoCoordenador = coCoordenador;
}

public void setCoProjeto(int coProjeto) {
    CoProjeto = coProjeto;
}

public void setDaInicio(Date daInicio) {
    DaInicio = daInicio;
}

public void setDaTermino(Date daTermino) {
    DaTermino = daTermino;
}

```

```

    }

    public void setInProjetoIsolado(byte inProjetoIsolado) {
        InProjetoIsolado = inProjetoIsolado;
    }

    public void setNoLocalRealizacao(String noLocalRealizacao) {
        NoLocalRealizacao = noLocalRealizacao;
    }

    public void setNoTituloProjeto(String noTituloProjeto) {
        NoTituloProjeto = noTituloProjeto;
    }

    public void setTxPublicoAlvo(String txPublicoAlvo) {
        TxPublicoAlvo = txPublicoAlvo;
    }

    public void setTxResumo(String txResumo) {
        TxResumo = txResumo;
    }

    public void inserir(Projetos p){

    }

}

```

```
package negocio;
```

```
import java.sql.ResultSet;
import java.sql.SQLException;
```

```
import dados.UsuariosDAO;
```

```
public class Usuarios {
```

```
    private int CoUsuario;
    private String NoUsuario;
    private String TxSenha;
    private int InNivelAcesso;
```

```
    public int getCoUsuario() {
        return CoUsuario;
    }
```

```
    public void setCoUsuario(int coUsuario) {
        CoUsuario = coUsuario;
    }
```

```
    public int getInNivelAcesso() {
        return InNivelAcesso;
    }
```

```
    public void setInNivelAcesso(int inNivelAcesso) {
        InNivelAcesso = inNivelAcesso;
    }
```

```
    public String getNoUsuario() {
        return NoUsuario;
    }
```

```
    public void setNoUsuario(String noUsuario) {
```

```

        NoUsuario = noUsuario;
    }
    public String getTxSenha() {
        return TxSenha;
    }
    public void setTxSenha(String txSenha) {
        TxSenha = txSenha;
    }

    public void consulta() throws SQLException{

        UsuariosDAO usuario = new UsuariosDAO();
        ResultSet rs = usuario.consulta(this);

        rs.next();
        CoUsuario = rs.getInt("CoUsuario");
        NoUsuario = rs.getString("NoUsuario");
        TxSenha = rs.getString("TxSenha");
        InNivelAcesso = rs.getInt("InNivelAcesso");
    }
}

```

APÊNDICE B - Dicionário de Dados

Módulo Acadêmico:

tbAlunos				
CH	Campo	Tipo	Tam	Observação
PK	CoAluno	inteiro	-	*autonumeração
	NuMatricula	texto	10	
	NoAluno	texto	50	
	TxEndereco	texto	60	
	NoBairro	texto	50	
	NoCidade	texto	50	
	SgUF	texto	2	
	NuCEP	texto	8	
	NuTelefone	texto	10	
	NuCelular	texto	10	
	InTipo	byte	-	0-Simples;1-Colaborador

tbProfessores				
CH	Campo	Tipo	Tam	Observação
PK	CoProfessor	inteiro	-	*autonumeração
	NuContrato	texto	10	
	NoProfessor	texto	50	
	TxEndereco	texto	60	
	NoBairro	texto	50	
	NoCidade	texto	50	
	SgUF	texto	2	
	NuCEP	texto	8	
	NuTelefone	texto	10	
	NuCelular	texto	10	
	InTipo	byte	-	0-Simples;1-Coordenador;2-Colaborador

tbProjetos				
CH	Campo	Tipo	Tam	Observação
PK	CoProjeto	inteiro	-	*autonumeração
FK	CoAreaAtuacao	inteiro	-	relacionado com a tabela tbAuxiliar
FK	CoCoordenador	inteiro	-	relacionado com a tabela tbProfessores
	NoTituloProjeto	texto	60	
	NoLocalRealizacao	texto	60	
	InProjetoIsolado	byte	-	0-sim;1-não
	TxPublicoAlvo	texto	250	
	DaInicio	data	-	
	DaTermino	data	-	
	TxResumo	memo	-	

tbProjetosProfessores				
CH	Campo	Tipo	Tam	Observação
PK	CoProjetoProfessor	inteiro	-	*autonumeração
FK	CoProjeto	inteiro	-	relacionado com a tabela tbProjeto
FK	CoProfessor	inteiro	-	relacionado com a tabela tbProfessor

tbProjetosAlunos				
CH	Campo	Tipo	Tam	Observação
PK	CoProjetoAluno	inteiro	-	*autonumeração
FK	CoProjeto	inteiro	-	relacionado com a tabela tbProjeto
FK	CoAluno	inteiro	-	relacionado com a tabela tbAlunos

tbProjetosParcerias				
CH	Campo	Tipo	Tam	Observação
PK	CoProjetoParceria	inteiro	-	*autonumeração
FK	CoProjeto	inteiro	-	relacionado com a tabela tbProjeto
FK	CoParceria	inteiro	-	relacionado com a tabela tbAuxiliar

tbProjetosCursosVinculados				
CH	Campo	Tipo	Tam	Observação
PK	CoProjetoCursos	inteiro	-	*autonumeração
FK	CoProjeto	inteiro	-	relacionado com a tabela tbProjeto
FK	CoCursos	inteiro	-	relacionado com a tabela tbAuxiliar

tbAuxiliar				
CH	Campo	Tipo	Tam	Observação
PK	CoAuxiliar	inteiro	-	*autonumeração
	CoTipo	byte	-	0-areas_atuacao;1-parcerias;2-cursos
	NoAuxiliar	texto	60	

Módulo de Segurança:

tbUsuarios				
CH	Campo	Tipo	Tam	Observação
PK	CoUsuario	inteiro	-	*autonumeração
	NoUsuario	texto	50	
	TxSenha	texto	6	
	InNivelAcesso	tinyint	-	

tbOperacoes				
CH	Campo	Tipo	Tam	Observação
PK	CoOperacao	inteiro	-	*autonumeração
	NoOperacao	texto	60	
	InNivelAcesso	tinyint	-	*valor mínimo para acessar a operação

Logging:

tbLOG				
CH	Campo	Tipo	Tam	Observação
PK	CoLOG	inteiro	-	*autonumeração
	DaLOG	data	-	
	NoOperacao	texto	60	
	TxParametros	memo		