**UNIVERSIDADE FEDERAL DE SANTA MARIA**
**CENTRO DE TECNOLOGIA**
**CURSO DE CIÊNCIA DA COMPUTAÇÃO**

# PARADIGMA ORIENTADO A ASPECTOS, ESTUDO DE CASO: SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA

**TRABALHO DE GRADUAÇÃO**

**Viviane Aquino Rodrigues**

**Santa Maria, RS, Brasil**
**2007**

# PARADIGMA ORIENTADO A ASPECTOS, ESTUDO DE CASO: SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA


**por**

**Viviane Aquino Rodrigues**


Trabalho de Graduação apresentado ao Curso de Ciência da Computação - Bacharelado, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação**

**Orientadora: Msc. Oni Reasilvia Sichonany**

**Santa Maria, RS, Brasil**
**2007**

**Universidade Federal de Santa Maria**
**Centro de Tecnologia**
**Curso de Ciência da Computação**

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

**PARADIGMA ORIENTADO A ASPECTOS, ESTUDO DE CASO: SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA**

elaborado por
**Viviane Aquino Rodrigues**

como requisito parcial para obtenção do grau de
**Bacharel em Ciência da Computação.**

**Comissão Examinadora:**

**Oni Reasilvia Sichonany, Msc.**
(Orientadora)

**Andrea Schwertner Charão, Dra.** (UFSM)

**Gédson Mário Borges Dal Forno, Msc.** (UFSM)

**Trabalho de Graduação nº 239**
Santa Maria, 24 de agosto de 2007.

## Agradecimentos

Agradeço à minha família que sempre esteve ao meu lado me dando apoio e incentivo para encarar os obstáculos e seguir em frente.

À professora Oni Reasilvia Sichonany, minha orientadora, por me dar o apoio, contribuindo com incentivo intelectual e toda a ajuda necessária.

Aos professores do Curso de Ciência da Computação por transmitirem seus conhecimentos e experiência, auxiliando na minha formação profissional.

Aos colegas do curso com quem compartilhei meus problemas e alegrias.

Ao meu namorado, Julio, pela compreensão e por estar me incentivando em todos os momentos.

Agradeço ainda a todas as pessoas que não foram mencionadas, mas que me incentivaram com palavras de carinho e apoio.

Quando somos pacientes, coisas que normalmente consideraríamos muito dolorosas acabam não parecendo tão ruins. Ao contrário, quando não existe a tolerância paciente, até as menores contrariedades parecem insuportáveis.
(Dalai Lama)

# RESUMO

Trabalho de Graduação
Ciência da Computação
Universidade Federal de Santa Maria

**PARADIGMA ORIENTADO A ASPECTOS, ESTUDO DE CASO: SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA**

AUTORA: VIVIANE AQUINO RODRIGUES
ORIENTADORA: MSC. ONI REASILVIA SICHONANY
Data e Local da Defesa: Santa Maria, 24 de agosto de 2007.

No desenvolvimento de *software*, os objetos do mundo real são representados através de abstrações providas pelos paradigmas de programação. Porém, em certos casos, essas abstrações não são suficientes para representar determinadas características presentes no *software*. Com isso, foi criada a Orientação a Aspectos, provendo um novo nível de abstração, o aspecto, com o objetivo de representar essas características de maneira clara e concisa. A Orientação a Aspectos apresenta uma solução para problemas até então difíceis de serem resolvidos, como a dificuldade de isolar preocupações referentes a atividades de suporte da aplicação. Proporcionando assim a redução da complexidade e o aumento da produtividade no desenvolvimento e manutenção de *software*. O trabalho apresenta o Paradigma Orientado a Aspectos demonstrando suas vantagens e eficácia através do estudo de caso de um sistema de auditoria *(logging)* para aplicações Java. O sistema de auditoria é implementado utilizando-se orientação a objetos e orientação a aspectos, fazendo uso dos *frameworks* e ferramentas mais comuns existentes.

Palavras-chave: programação orientada a aspectos; programação orientada a objetos; auditoria; separação de preocupações; preocupações transversais; AspectJ; *framework*; Spring AOP.

# ABSTRACT

Trabalho de Graduação
Ciência da Computação
Universidade Federal de Santa Maria

**ASPECT ORIENTED PARADIGM, CASE STUDY: AUDIT TRAIL SYSTEM FOR JAVA APPLICATIONS**

AUTORA: VIVIANE AQUINO RODRIGUES
ORIENTADORA: MSC. ONI REASILVIA SICHONANY
Data e Local da Defesa: Santa Maria, 24 de agosto de 2007.

In software development, real world objects are represented through abstractions provided by programming paradigms. However, in some cases these abstractions are not sufficient to represent some kinds of characteristics present in software. Because of this, it was created Aspect Oriented Programming that provides a new abstraction level, called aspect; for the purpose of represent these characteristics in a clear and concise way. Aspect Orientation presents a solution for problems until then difficult to solve, as isolating concerns referring to application support activities. AOP reduces complexity and improves productivity in software development and maintenance. This work presents the Aspect Oriented Paradigm showing its advantages and efficiency through an audit trail system case study for Java applications. The system is implemented using object-oriented programming and aspect-oriented programming, with the most common frameworks and tools.

Key-words: aspect-oriented programming, object-oriented programming, audit trail, separation of concerns, crosscut concerns, framework, AspectJ, Spring AOP.

# LISTA DE ILUSTRAÇÕES


FIGURA 2.1 - Composição de um sistema orientado a aspectos........................ 13
FIGURA 2.2 - Combinação aspectual............................................................ 15
FIGURA 2.3 - Separação de Preocupações...................................................... 17
FIGURA 2.4 - Ocorrência de código emaranhado............................................ 19
FIGURA 2.5 - Aplicação com POA.................................................................. 20
FIGURA 2.6 - Classe Mensageiro.................................................................... 25
FIGURA 2.7 - Aspecto AspectoMensageiro..................................................... 26
FIGURA 3.1 - Arquitetura da aplicação........................................................... 33
FIGURA 3.2 - Diagrama de Classes da Camada *Model*..................................... 34
FIGURA 3.3 - Diagrama de Classes da Camada *Service*.................................. 35
FIGURA 3.4 - Classe “TransacaoServiceImpl” da Camada *Service*................... 36
FIGURA 3.5 - Diagrama de Classes da Camada *Action*.................................... 37
FIGURA 3.6 - Diagrama de Classes................................................................. 39
FIGURA 3.7 - Aspecto implementado com Spring AOP.................................. 42
FIGURA 3.8 - Código com POA e *Annotations*................................................ 43
FIGURA 3.9 - Arquivo de configuração........................................................... 44
FIGURA 3.10 - Página de cadastro das movimentações...................................... 46
FIGURA 3.11 - Registros das operações............................................................ 46
FIGURA 3.12 - Comparação de aplicações com e sem POA.............................. 47
FIGURA 3.13 - Implementação com POA.......................................................... 48
FIGURA 3.14 - Implementação sem POA.......................................................... 49

## SUMÁRIO


**1 INTRODUÇÃO**........................................................................................ 9
**2 REVISÃO DA LITERATURA**.............................................................. 11
**2.1 Programação Orientada a Aspectos**.................................................. 11
**2.2 Fundamentos da Orientação a Aspectos**........................................... 16
**2.3 Conceitos da Orientação a Aspectos**................................................. 21
2.3.1 Pontos de Junção.................................................................................. 22
2.3.2 Pontos de Atuação................................................................................ 22
2.3.3 Adendos................................................................................................ 23
2.3.4 Aspectos ............................................................................................... 24
**2.4 Algumas Ferramentas de Orientação a Aspectos para Java**.............. 27
2.4.1 AspectJ.................................................................................................. 27
2.4.2 Spring AOP........................................................................................... 28
2.4.3 JAC........................................................................................................ 29
**3 SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA**.................. 31
**3.1 Detalhes do Sistema de Auditoria**..................................................... 31
**3.2 Detalhes do Caso de Uso**.................................................................. 33
**3.3 Arquitetura do Sistema de Auditoria**................................................. 38
**3.4 Implementação do Sistema de Auditoria**.......................................... 40
**3.5 Configuração do Sistema de Auditoria**............................................. 43
**3.6 Funcionamento do Sistema de Auditoria com o Caso de uso**............ 45
**3.7 Comparação do Sistema de Auditoria com e sem POA**.................... 47
**4 CONCLUSÕES**...................................................................................... 52
**4.1 Contribuições**...................................................................................... 52
**4.2 Estudos Futuros**.................................................................................. 53
**4.3 Considerações Finais**.......................................................................... 53
**REFERÊNCIAS**...................................................................................... 55

# Capítulo 1
# INTRODUÇÃO

Existem certos requisitos de *software* que não são implementados com facilidade tanto com programação procedimental quanto com programação orientada a objetos. A implementação desses requisitos, difíceis de serem isolados, "corta" o código de toda a aplicação ou boa parte dela, resultando em um código complexo, difícil de manter e de desenvolver (KICZALES, 1997).

Diante da falta de um paradigma que provesse uma solução para reduzir a complexidade e aumentar a produtividade no desenvolvimento de *software*, pesquisadores do Centro de Pesquisa da Xerox em Palo Alto (Xerox Palo Alto Research Center - PARC) criaram o Paradigma Orientado a Aspectos, capaz de solucionar tais problemas (WINCK, 2006).

O Paradigma Orientado a Aspectos propõe uma nova metodologia de programação baseada na separação de preocupações (*Separation of Concerns*). As preocupações são classificadas basicamente em dois tipos: preocupações funcionais, responsáveis pelas funcionalidades da aplicação e as preocupações sistêmicas ou transversais, que fornecem suporte à aplicação e tornam-se muitas vezes indispensáveis para o funcionamento do sistema.

Por meio da Orientação a Aspectos, as preocupações podem ser isoladas em módulos, de acordo com características afins, facilitando a manutenção e aumentando o reuso de código. Podem ser consideradas preocupações sistêmicas: distribuição, tratamento de exceções, persistência, sincronização de objetos concorrentes, mecanismos de auditoria (*logging*) entre outros.

Ao se implementar uma aplicação sem o uso da Programação Orientada a Aspectos (POA), tanto as preocupações funcionais, inerentes ao negócio da aplicação, quanto as preocupações sistêmicas, são encontradas ao longo de todo o código, causando dois fenômenos, denominados de código emaranhado (*Tangled Code*) e código espalhado (*Spread Code*). A Orientação a Aspectos atua no sentido de resolver esses problemas, fornecendo um nível maior de abstração, o aspecto, para agrupar as preocupações sistêmicas. Dessa forma, as preocupações funcionais são implementadas separadamente das preocupações sistêmicas (WINCK, 2006).

As preocupações sistêmicas não precisam ser desenvolvidas concomitantemente ao desenvolvimento do *software*. Por meio da POA, é possível que até mesmo quando não se tem acesso ao código da aplicação, novas preocupações possam ser agregadas. Isso evita que o *software* seja descartado por não possuir inicialmente o comportamento proveniente das preocupações sistêmicas.

A POA permite aos programadores e analistas facilidades até então inexistentes, ajudando a romper os limites atuais do conhecimento no desenvolvimento de *software*. Com isso, a POA tem ganhado espaço na comunidade científica e tecnológica e é incorporada rapidamente em empresas (RESENDE, 2005).

O presente trabalho de graduação propõe o estudo da Programação Orientada a Aspectos através da implementação de um sistema de auditoria (*logging*) para aplicações Java.

O sistema de auditoria deverá gerar registros das ações tomadas durante a execução de uma aplicação, facilitando a identificação de possíveis erros, problemas de segurança, etc. O sistema exemplifica um dos usos do Paradigma Orientado a Aspectos.

Para fazer a análise, o projeto e a implementação do sistema serão utilizados alguns *frameworks* para orientação a aspectos existentes. O sistema de auditoria será implementado utilizando-se o *framework* Spring AOP. E como parte do trabalho, será implementada uma aplicação de gerência de contas bancárias, utilizando a linguagem de programação Java.

O capítulo 2 apresenta uma breve revisão bibliográfica sobre os principais conceitos utilizados no desenvolvimento deste trabalho. São apresentados os fundamentos e conceitos da Orientação a Aspectos. Também serão apresentadas algumas ferramentas de orientação a aspectos para Java.

O capítulo 3 mostra o uso prático da POA através do estudo de caso de um sistema de auditoria. Este capítulo apresenta também uma aplicação de gerência de contas bancárias para demonstrar o funcionamento do sistema.

Por fim, no capítulo 4 são apresentadas as conclusões deste trabalho. São descritas também, as principais contribuições do trabalho e as perspectivas para estudos futuros.

# Capítulo 2
# REVISÃO DA LITERATURA

Este capítulo buscará apresentar a Programação Orientação a Aspectos, seus fundamentos e conceitos. Também serão descritas algumas ferramentas de orientação a aspectos para Java, além do Spring AOP, que servirá de base para a implementação do sistema de auditoria.

## 2.1 Programação Orientada a Aspectos

Da necessidade de se diminuir a complexidade no desenvolvimento de *software* e aumentar a reutilização de código surgiu a Programação Orientada a Aspectos. A Orientação a Aspectos não tem como objetivo substituir os paradigmas existentes, como programação estruturada ou orientada a objetos, e sim atuar em conjunto com os mesmos, como um paradigma complementar no desenvolvimento de *software*.

Apesar de a Programação Orientada a Objetos prover um nível maior de modularização de seus componentes do que os paradigmas anteriores, ela é considerada insuficiente para isolar alguns tipos de preocupações, apresentadas na seqüência. No artigo que introduziu a programação orientada a aspectos, Kiczales (1997) diz que a POA surgiu para resolver problemas que tanto a programação orientada a objetos quanto a programação procedimental não resolviam de maneira satisfatória.

Na implementação de um sistema, os requisitos (ou preocupações) podem ser considerados como:

- Componentes: quando é facilmente encapsulado (por exemplo: objeto, método, procedimento). Um componente é geralmente uma unidade funcional do sistema.
- Aspectos: ao contrário dos componentes, não é facilmente encapsulado em um módulo. Aspectos são propriedades que afetam o desempenho ou a semântica dos componentes, por exemplo: sincronização de objetos concorrentes, manipulação de erros, etc. (KICZALES, 1997).

A POA possibilita um nível maior de abstração no desenvolvimento de *software* e a separação bem definida de componentes e aspectos. Por estarem bem separados e em locais bem definidos, tanto os componentes quanto os aspectos podem ser mais bem reutilizados e a manutenção e legibilidade tornam-se mais fáceis. A separação entre componentes e aspectos simplifica o problema, pois cada preocupação pode ser programada de forma independente para posteriormente se realizar a combinação entre componentes e aspectos produzindo o sistema como um todo (WINCK, 2006).

O aspecto é o nível de abstração provido pela POA. Utilizando-se a POA no desenvolvimento de *software*, as preocupações funcionais são implementadas na forma de componentes e as preocupações sistêmicas na forma de aspectos. O aspecto representa as preocupações difíceis de serem isoladas, que causam problemas como o código espalhado. Basicamente, o aspecto é composto pelo código que implementa as preocupações sistêmicas e as regras que definem os locais onde o comportamento implementado deve ser inserido.

Sendo a POA um paradigma complementar no desenvolvimento de *software*, é responsável somente pela implementação das preocupações sistêmicas da aplicação. Já, a implementação das preocupações funcionais fica a cargo de outro paradigma e conseqüentemente de outra linguagem de programação. Podem ser encontradas extensões de POA para diversas linguagens de programação, como: Java, C/C++, Delphi, Perl, Lua, Common Lisp, entre outras.

Segundo Kiczales (1997), uma implementação baseada na programação orientada a aspectos é composta por:

- Linguagem de componentes: utilizada para implementar as preocupações funcionais, ou seja, as funcionalidades básicas da aplicação.
- Linguagem de aspectos: usada na implementação das preocupações sistêmicas, permite que o programador crie estruturas que descrevam o comportamento dos aspectos e definam em que situações eles ocorrem.
- Programas escritos em linguagem de componentes: onde as preocupações funcionais da aplicação são implementadas através de componentes. No contexto da programação orientada a aspectos, os componentes são abstrações providas pela linguagem, que permite a implementação da funcionalidade do sistema.
- Programas escritos em linguagem de aspectos: deve atender às preocupações sistêmicas.

- Combinador de aspectos (*aspect weaver*): combina os programas escritos em linguagem de componentes com os escritos em linguagem de aspectos, essa atividade é chamada de combinação aspectual.

A Figura 2.1 mostra como é composto um sistema orientado a aspectos.

Figura 2.1 - Composição de um sistema orientado a aspectos

A POA apresenta características da meta-programação, uma delas é a reflexão computacional, em que parte do código gerado é destinado a modificar características do próprio programa. Outra característica é que os compiladores destinados à orientação a aspectos não geram um programa compilado e executável, e sim um novo código. Na primeira compilação são acrescidos elementos ao código para dar suporte às novas abstrações. O código resultante precisa ser recompilado gerando então o produto final (WINCK, 2006).

A POA difere de outras metodologias principalmente por permitir que linguagens de componentes e de aspectos com diferentes abstrações possam ser combinadas. Um

processador de linguagem, chamado de *aspect weaver*, é o mecanismo que coordena a combinação de aspectos e componentes.

O a*spect weaver*, ou combinador aspectual, recebe programas de componentes e de aspectos como entrada e gera um programa em linguagem de componentes como saída. O combinador aspectual tem como tarefa a integração do sistema, obtida através da combinação aspectual (KICZALES, 1997).

A combinação aspectual combina os elementos escritos em linguagem de componentes com os elementos escritos em linguagem de aspectos. Esse processo é anterior à compilação e gera um código intermediário na linguagem de componentes capaz de produzir o resultado desejado, ou de permitir a sua realização durante a execução do programa. Apesar de a combinação aspectual produzir um código intermediário na linguagem de componentes, esse código acrescentado pelos aspectos é invisível ao programador.

Após a combinação aspectual, o código dos componentes permanece inalterado aos olhos do programador. Então, por não haver nada no código desses componentes que indique a presença de um comportamento adicionado pelo código dos aspectos, ocorre um problema chamado de *Obliviousness*. Entretanto esse problema encontra-se amenizado, pois existem ambientes de programação que notificam a existência do código ao programador (RESENDE, 2005).

O código referente ao negócio da aplicação, escrito em linguagem de componentes, não sofre qualquer alteração para suportar a programação orientada a aspectos. Isso é feito no momento da combinação entre os componentes e os aspectos. Esse processo também pode ser definido como recompilação aspectual, conforme demonstrado pela Figura 2.2.

Na Figura 2.2 as classes A e B são combinadas com o Aspecto através da combinação aspectual, realizada por um *weaver*, resultando em um código intermediário em linguagem de componentes.

O combinador aspectual processa as linguagens de componentes e de aspectos, combinando-as com o objetivo de produzir o sistema desejado. Entretanto, questões de segurança devem ser consideradas durante a combinação aspectual, já que o código implementado nos aspectos e injetado no código da aplicação não é visível para o programador.

A combinação aspectual pode ser:

- Estática: trazendo agilidade ao sistema orientado a aspectos, visto que não há necessidade de que os aspectos existam em tempo de compilação e execução. A

combinação estática previne que um nível adicional de abstração cause um impacto negativo no desempenho do sistema.

- Dinâmica: na combinação dinâmica é necessário que os aspectos existam tanto em tempo de compilação quanto em tempo de execução. Utilizando uma interface reflexiva, o combinador de aspectos pode adicionar, adaptar e remover aspectos em tempo de execução (WINCK, 2006).

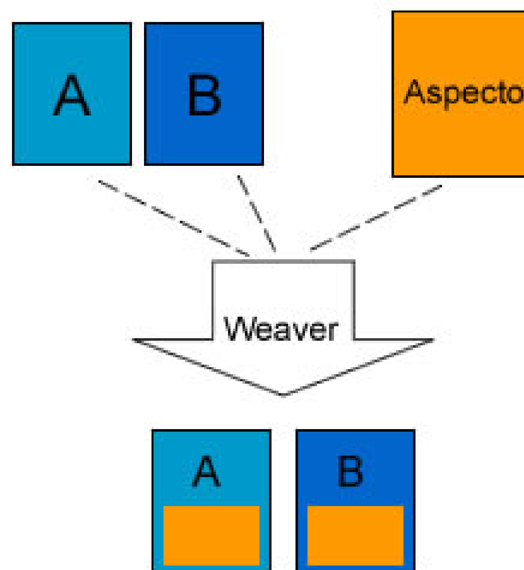



Figura 2.2 - Combinação aspectual

A Programação Orientada a Aspectos baseia-se na teoria de Separação de Preocupações, com o objetivo de permitir que os códigos referentes ao negócio da aplicação e às preocupações sistêmicas sejam implementados isoladamente, de acordo com características afins, de forma definida e centralizada. Aplicando-se a teoria de Separação de Preocupações pode-se obter um código mais legível, melhor reutilizado e de manutenção mais fácil.

O uso da POA é motivado pelas preocupações crescentes e cada vez mais complexas no desenvolvimento de *software*, visto que os paradigmas atuais, não atendem às necessidades para implementação de um sistema completo sem que certos problemas sejam gerados.

Vem sendo dada uma atenção maior à integração das aplicações e à solução de atividades secundárias, como: distribuição, segurança, persistência e auditoria, fazendo com que o desenvolvimento orientado a aspectos seja cada vez mais explorado na busca de

soluções que permitam uma maior produtividade no desenvolvimento e dinamicidade na evolução de *software*. A POA é usada desde o nível de implementação até os outros estágios do processo de desenvolvimento, incluindo especificação de requisitos, análise e projeto (RESENDE, 2005).

A POA permite que mesmo em uma situação em que não se tem acesso ao código da aplicação, seja possível agregar novos comportamentos, como auditoria, a essa aplicação. Isso é possível identificando-se os pontos onde é necessário que o novo comportamento seja executado. Essa estratégia evitaria que a aplicação fosse descartada por não possuir em sua implementação inicial esses novos comportamentos, agora necessários e possíveis de serem implementados, por meio da POA.

## 2.2 Fundamentos da Orientação a Aspectos

*Separation of Concerns*, em português Separação de Preocupações ou Separação de Interesses, é a base da Orientação a Aspectos e consiste em separar os requisitos de *software* de acordo com seus interesses, funcionalidades, preocupações, entre outros. A separação de preocupações deve ser aplicada em todas as fases do processo de desenvolvimento de *software* sendo fundamental para se obter sucesso no desenvolvimento de um sistema baseado no paradigma orientado a aspectos.

A teoria de Separação de Preocupações pode ser definida como uma teoria que investiga como separar as preocupações presentes em um sistema, permitindo que cada uma das preocupações seja tratada isoladamente, para que se obtenha redução de complexidade no desenvolvimento, evolução e integração de *software* (RESENDE, 2005).

Na Figura 2.3, cada retângulo colorido representa uma preocupação sistêmica, como distribuição, tratamento de exceções, entre outras. A figura mostra duas situações: na parte inferior, onde as preocupações sistêmicas encontram-se espalhadas dentro vários módulos e na parte superior, aplicando a teoria de Separação de Preocupações, onde pode-se perceber que cada preocupação está separada em módulos independentes, evitando-se a situação ilustrada na parte inferior.

Outra expressão encontrada na literatura é *Multidimensional Separation of Concerns*, em português, Separação Multidimensional de Preocupações, onde cada preocupação, ou

interesse, é vista como uma dimensão a ser tratada isoladamente. O ato de integrar equivale a relacionar as diferentes dimensões existentes.

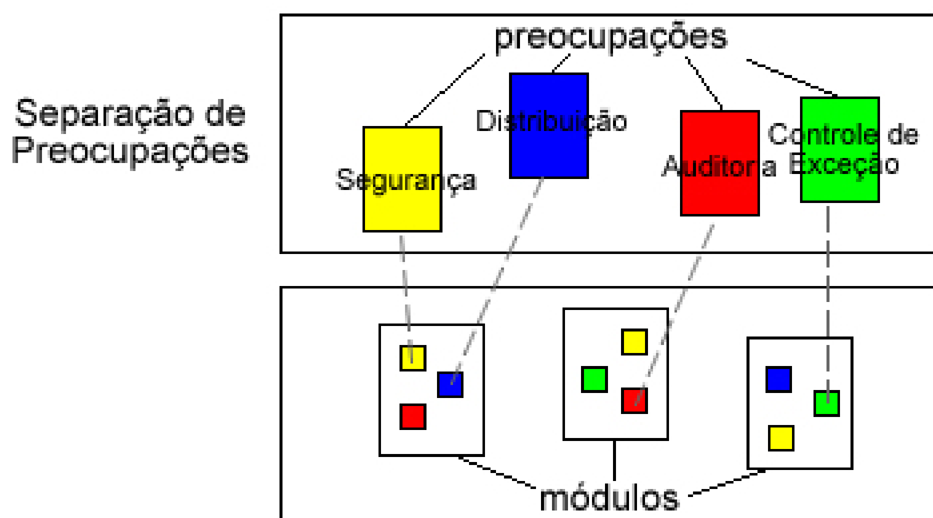



Figura 2.3 - Separação de Preocupações

A teoria de Separação de Preocupações proporciona a redução da complexidade no desenvolvimento de *software*. Estabelece que cada preocupação deva ser tratada de maneira isolada, como um problema menor, com isso diminui-se a ocorrência de erro no módulo e conseqüentemente no sistema como um todo. Embora a teoria proponha separar preocupações em módulos independentes, isolar essas preocupações nem sempre é uma tarefa fácil, pois profissionais da área podem ter diferentes percepções de arranjos do *software* (RESENDE, 2005).

Interesses ou preocupações são tanto os requisitos funcionais, ou seja, que fazem parte do domínio da aplicação, quanto os requisitos não funcionais, também chamados de preocupações sistêmicas, ortogonais ou também transversais. Essas últimas são importantes para o funcionamento do sistema, dando suporte à aplicação, porém não estão diretamente relacionadas com as funcionalidades da aplicação.

As preocupações ortogonais garantem fatores como segurança, distribuição, persistência, auditoria, entres outros. Implementadas sem a utilização da Orientação a Aspectos, o código dessas preocupações atravessa toda a aplicação, por isso chamadas de ortogonais. Pelo fato de atravessarem toda a aplicação, acabam se entrelaçando às preocupações funcionais. O espalhamento e o entrelaçamento de código dificultam a

modularização, causando problemas como o Código Espalhado (*Spread Code*) e o Código Emaranhado (*Tangled Code*).

Espalhamento de código

O código espalhado, também conhecido como *Spread Code* ou *Scattering Code* é o código necessário para cumprir uma preocupação, propagado em classes ou módulos que precisam cumprir outras preocupações. Pode ser classificado em bloco de código duplicado e bloco de código complementar.

A atividade de auditoria (*logging*) é um dos exemplos mais comuns de ocorrência de código espalhado. Se no momento em que determinadas operações forem executadas for necessário gravar um *log*[1], a chamada para o método que grava o *log* vai ser encontrada inúmeras vezes, ao longo de todos os módulos que contiverem a implementação dessas operações.

Por exemplo, considerando um sistema de venda de produtos, que grava um *log* toda vez que o usuário visualiza detalhes de um produto, ou busca por um produto. Durante a execução da operação que detalha ou procura o produto, também será feita uma chamada ao método que grava o *log*. Se por algum motivo a maneira como é feita a gravação do *log* precisar ser modificada, todos os módulos que implementam as operações que chamam o método de *log* também precisarão de alteração no código.

Outro exemplo típico de ocorrência de código espalhado é a implementação de distribuição. A distribuição chama métodos de classes pertencentes à outra aplicação, esses métodos usam chamadas remotas, precisando de métodos específicos de RMI[2] ou de CORBA[3], por exemplo. Essa implementação acarreta um número muito grande de linhas repetidas dentro do código. Caso essas chamadas remotas sejam utilizadas em 50 métodos, esses 50 métodos terão inúmeras linhas repetidas dentro do seu código. Se for necessário substituir os métodos implementados em CORBA por RMI, por exemplo, todos esses 50 métodos terão que ser alterados (RESENDE, 2005).

Além do problema de se ter classes "inchadas" com códigos de distribuição, persistência, auditoria, dentre outros, tem-se o problema da repetição destes códigos. O código espalhado implica em retrabalho, visto a necessidade de tempo extra para alterar os

---

[1] Registro de operação

[2] *Remote Method Invocation* (RMI) permite ao programador implementar sistemas distribuídos em Java, nos quais os métodos de objetos remotos podem ser invocados de outras máquinas virtuais, possivelmente em diferentes servidores.

[3] *Common Object Request Broker Architecture* (CORBA) é uma arquitetura para estabelecer e simplificar a troca de dados entre sistemas distribuídos heterogêneos.

pontos necessários, causando uma redução de produtividade e dificultando a manutenção do código.

Entrelaçamento de código

É chamado de *Tangled Code*, ou Código emaranhado, o entrelaçamento de código. O código entrelaçado ocorre quando preocupações distintas são implementadas em um único módulo. Geralmente o desenvolvedor considera as preocupações sistêmicas como lógica do negócio durante a implementação de um módulo ou classe, o que acaba resultando na implementação de várias preocupações em um mesmo módulo (WINCK, 2006).

Quando duas preocupações com diferentes propósitos são combinadas em um módulo, diz-se que uma "corta" a outra, ou seja, que elas estão entrelaçadas, resultando no código emaranhado, difícil de ser entendido e mantido.

Por exemplo, considerando como preocupações, segurança e tratamento de exceções, se essas preocupações forem implementadas em um único módulo (numa classe, por exemplo), pode-se dizer que essas preocupações encontram-se entrelaçadas.

Uma aplicação que faz uso da atividade de auditoria, por exemplo, pode ser caracterizada pela Figura 2.4. Essa imagem mostra a ocorrência de código emaranhado. Cada coluna branca representa um módulo, já as linhas escuras indicam as preocupações transversais, neste caso códigos de auditoria. Percebe-se que essas preocupações transversais atravessam vários módulos da aplicação, comprovando a existência de código emaranhado.

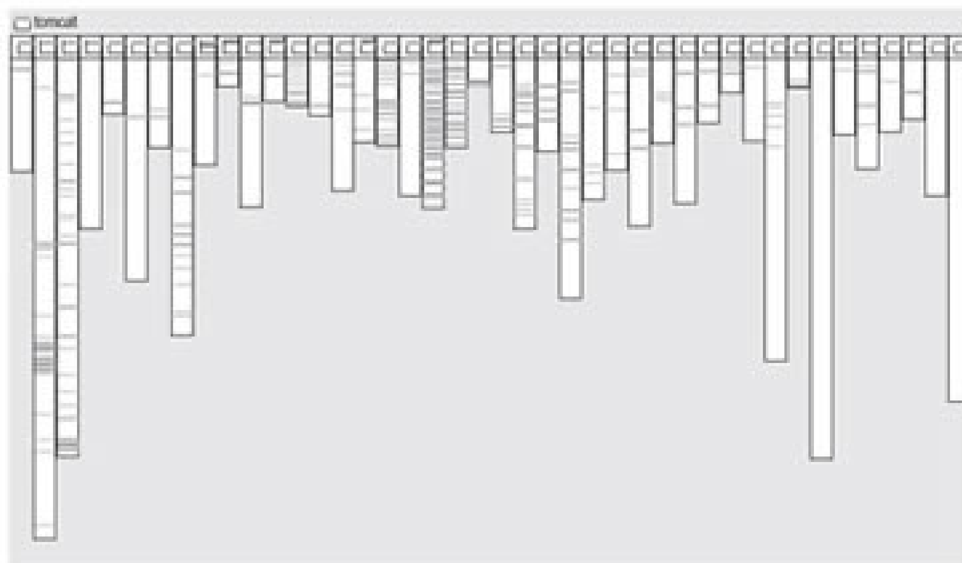


Figura 2.4 - Ocorrência de código emaranhado - Fonte: WINCK (2006, p. 35).

Implementar as preocupações sistêmicas de maneira entrelaçada e espalhada ao longo dos módulos da aplicação, implica em um código mais complexo, difícil de manter e conseqüentemente torna a evolução do sistema mais difícil.

Tanto o problema de entrelaçamento quanto o problema de espalhamento de código podem ser resolvidos com a Programação Orientada a Aspectos. Os componentes são desenvolvidos e então criados os aspectos, onde são implementadas as preocupações sistêmicas.

Aplicando-se POA para a mesma situação, representada na Figura 2.4, obtém-se a Figura 2.5. Nesta figura, assim como na Figura 2.4, as colunas brancas representam os módulos. Entretanto a preocupação sistêmica, aqui representada em vermelho, encontra-se em um local bem definido, diferentemente do que ocorre na figura anterior, demonstrando assim que o código emaranhado é evitado. Neste caso, a preocupação sistêmica é representada por um aspecto.

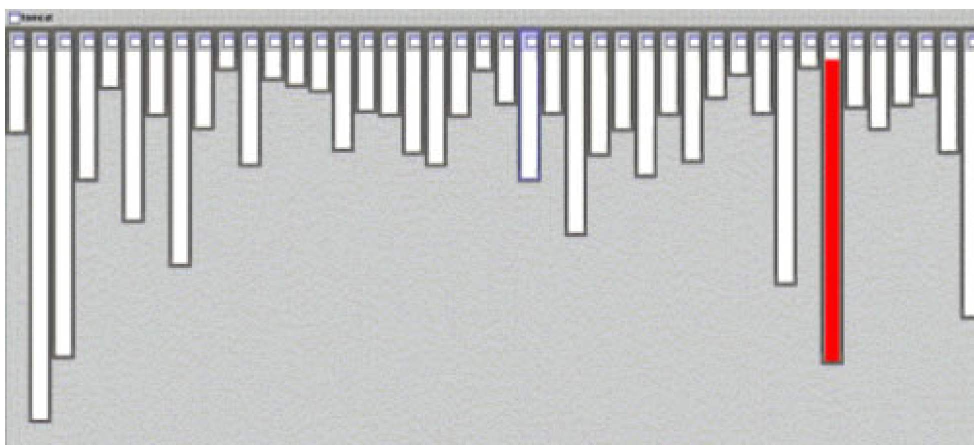


Figura 2.5 – Aplicação com POA - Fonte: WINCK (2006, p. 35).

Dentre os benefícios proporcionados pela POA, pode-se citar:

- Redução da quantidade de linhas programadas: o componente original não fica "inchado" com as linhas de código referentes a outros componentes. Todo o código redundante presente nos componentes, causado pelo entrelaçamento e espalhamento de código, não existe mais, pois esse código é implementado através de aspectos.
- Facilidade de manutenção: como o entrelaçamento e o espalhamento de código são evitados, as modificações tornam-se mais simples.

- Aumento da reusabilidade de código: como as classes ficam limpas, sem chamadas desnecessárias, torna-se fácil o reuso por não ser mais necessário desintegrar o componente de um sistema para depois reintegrá-lo a outro, ou seja, não é mais necessário entrar no código da classe e apagar as chamadas a métodos que tratam, por exemplo, de concorrência, distribuição ou persistência.
- Menor tempo e custo de desenvolvimento: obtidos em conseqüência do aumento da reusabilidade, facilidade de manutenção e da redução da quantidade de código programado.
- Maior coesão e menor acoplamento entre os objetos: a coesão é obtida, pois cada componente implementado realiza uma preocupação bem definida e a redução de acoplamento ocorre, pois a POA possibilita uma maior independência entre os componentes (RESENDE, 2005).

A POA propicia separar a atividade de desenvolver da atividade de integrar *software*. Primeiro são implementadas as funcionalidades da aplicação para depois serem integradas aos módulos de suporte ao *software*. Permite-se reutilizar componentes de outros sistemas e alterar sua estrutura interna como atributos e métodos, utilizando POA.

A diferença entre a orientação a aspectos e outros paradigmas, é que estes suportam apenas a separação de componentes enquanto que a orientação a aspectos permite a separação bem definida de componentes e aspectos. Isso é possível através de mecanismos da POA que permitem abstrair e combinar aspectos e componentes para produzir o sistema como um todo (KICZALES, 1997).

O objetivo final da POA é reduzir a complexidade no desenvolvimento de *software*, por meio do uso da Separação de Preocupações, permitindo que as diversas preocupações possam ser modularizadas independentemente de serem funcionais ou sistêmicas (RESENDE, 2005).

## 2.3 Conceitos da Orientação a Aspectos

A orientação a aspectos possui quatro conceitos fundamentais, são eles: pontos de junção, pontos de atuação, adendos e aspectos.

### 2.3.1 Pontos de Junção

Pontos de junção, ou *Join points*, são pontos bem específicos no fluxo de execução de um programa, como, por exemplo, a ocorrência de uma exceção, uma chamada a um método, dentre outros. São os locais onde ocorrem as preocupações transversais.

Os pontos de junção permitem identificar pontos de interrupção em um *software*, possibilitando inserir novos comportamentos e desviar o fluxo do programa. São considerados como uma das bases da POA.

Todo ponto de junção está associado a um contexto. Por exemplo, a chamada para um método possui um objeto chamador, objeto alvo e os argumentos do método disponível como contexto (WINCK, 2006).

Em um *software* que se pretende aplicar a orientação a aspectos, são localizados os pontos onde possa ocorrer o entrelaçamento ou o espalhamento de código e assim identificados os pontos de junção.

### 2.3.2 Pontos de Atuação

Assim como na orientação a objetos existem os atributos, na orientação a aspectos existem os pontos de atuação, também chamados de *point cuts*, que são estruturas bastante semelhantes aos atributos. Um ponto de atuação declara os pontos onde se deseja desviar a execução de um programa para inserir outro comportamento.

Os pontos de atuação são usados para definir nomes para um ou mais pontos de junção, como uma espécie de regra que especifica ações que serão atribuídas aos pontos de junção. Com os pontos de atuação os pontos de junção não precisam ser definidos individualmente.

Um ponto de atuação agrupa os pontos de junção que precisam que um comportamento em comum seja executado, ou seja, que tenham uma preocupação em comum. O ponto de atuação age como um agrupador para um determinado ponto de junção.

Se bem definidos, os pontos de atuação podem permitir que o *software* evolua, ou seja, permitir que o comportamento de um aspecto tenha efeito na execução de futuros métodos ou

classes, que ainda não foram implementados, sem que os pontos de atuação precisem ser modificados.

Outra função dos pontos de atuação é apresentar dados do contexto de execução de cada ponto de junção, que serão utilizados pela rotina disparada pela ocorrência do ponto de junção mapeado pelo ponto de atuação.

A combinação do ponto de junção e atuação evidencia a ocorrência de preocupações transversais, que causam o entrelaçamento e espalhamento de código, já que essas preocupações muito provavelmente estarão propagadas em diversas classes no sistema, e estas classes, estarão, portanto, implementando mais de uma preocupação (WINCK, 2006).

Várias linguagens orientadas a aspectos permitem a utilização de nomes, pontos de atuação, para referenciar os pontos de junção.

### 2.3.3 Adendos

Os adendos, em inglês *Advice*, fazem parte da implementação de um aspecto e são executados em pontos bem específicos do programa principal (pontos de junção). São compostos pelo ponto de atuação e pelo código que implementa o comportamento que será executado quando ocorrer o ponto de junção definido pelo ponto de atuação.

O adendo é bastante similar a um método, declara o código que deve ser executado a cada ponto de junção definido pelo ponto de atuação, ou seja, implementa o comportamento que deve ser executado toda vez que o ponto de junção for alcançado. O adendo é composto por um conjunto de instruções. Um adendo é associado a um ponto de atuação utilizando-se três palavras reservadas denominadas *before* (antes), *around* (durante) e *after* (depois) (RESENDE, 2005).

O adendo pode ser classificado basicamente em três tipos, que definem o momento em que o código do adendo será executado.

- *Before*: o comportamento do adendo é executado antes da execução do ponto de junção alcançado.
- *Around*: o comportamento do adendo é executado durante a execução do ponto de junção encontrado, e tem total controle sobre a computação do ponto de junção. Este é o tipo mais poderoso de adendo.

- *After*: após a computação do ponto de junção, o comportamento do adendo é executado. Em algumas linguagens, é possível encontrar duas variações para *after*: *after throwing* (executa após o ponto de junção ter gerado uma exceção) e *after returning* (executa após o ponto de junção ter executado normalmente), usando somente a expressão a*fter*, o comportamento é executado em ambos os casos.

O adendo é composto por três partes:

- Declaração do ponto de atuação: especifica sobre qual ponto de atuação o comportamento implementado no adendo terá efeito.
- Declaração do adendo: contém as palavras reservadas: *before*, *after* ou *around*. Assim como os métodos, também pode possuir parâmetros, porém não possui identificador próprio.
- Corpo do adendo: atua exatamente como o corpo de um método, contendo o código a ser executado. Também pode efetuar chamadas a outros métodos.

O adendo nada mais é do que o comportamento adicional, responsável por desempenhar alguma preocupação sistêmica ao longo do código base da aplicação. Como a POA pode ser tanto dinâmica quanto estática, os adendos podem ser criados dinâmica ou estaticamente. As modificações nos adendos estáticos somente têm efeito após os mesmos terem sido recompilados, já os adendos dinâmicos podem ser criados ou destruídos em tempo de execução.

### 2.3.4 Aspectos

O aspecto é o nível de abstração provido pela orientação a aspectos, que tem o objetivo de modularizar as partes do sistema referentes às preocupações sistêmicas, já que estas não são claramente modularizadas utilizando-se outros paradigmas. Pode-se considerar o aspecto análogo a uma classe.

Na POA as preocupações não inerentes ao negócio, denominadas preocupações sistêmicas, são agrupadas em aspectos, evitando o código espalhado e emaranhado. Dessa

forma, o aspecto não pode existir isoladamente para implementação das preocupações do sistema (tanto funcionais quanto sistêmicas) (WINCK, 2006).

Um programa orientado a aspectos usado com orientação a objetos tem como unidade principal o par aspecto-objeto, pois os objetos não deixam de existir, visto que a Orientação a Aspectos é apenas um paradigma complementar. Nos objetos são implementadas as preocupações funcionais e nos aspectos são tratadas as preocupações sistêmicas.

Os aspectos e as classes possuem estruturas semelhantes. O aspecto é composto por duas partes principais: uma estrutura de *point cuts* (conjunto de pontos de junção declarados) e os adendos (comportamentos a serem inseridos no fluxo de execução). Várias linguagens orientadas a aspectos também permitem que métodos e atributos possam ser implementados e definidos dentro de aspectos.

Além de permitir que comportamentos adicionais, referentes a preocupações sistêmicas, sejam inseridos na aplicação, os aspectos também permitem que o comportamento do código da aplicação seja modificado, por exemplo, alterar a hierarquia das classes. Um aspecto também pode adicionar membros e pais às classes.

Serão utilizados a classe "Mensageiro" (em Java), referente à Figura 2.6, e o aspecto "AspectoMensageiro" (em AspectJ), referente à Figura 2.7, para exemplificar em código os quatro conceitos principais da Orientação a Aspectos.

```
1  /*
2  * Classe Mensageiro
3  * Implementa dois métodos para impressão
4  * de mensagem.
5  */
6
7  // Definição do pacote
8  package com.aopbook;
9
10
11 public Class Mensageiro {
12
13     public static void entregaMsg(String mensagem){
14         System.out.println(mensagem);
15     }
16
17     public static void entregaMsg(String destino, String mensagem){
18         System.out.println(destino + ", "+mensagem);
19     }
20
21 }
```

Figura 2.6 - Classe Mensageiro

Pode-se considerar como exemplo de ponto de junção, a chamada a um método. A Figura 2.6 mostra o método "entregaMsg()" da classe "Mensageiro" como exemplo de ponto de junção. Já o tratamento desse ponto de junção é feito no ponto de atuação "entregaMensagem()", definido na linha 13 do aspecto "AspectoMensageiro". O ponto de atuação, "entregaMensagem()", captura as chamadas para todos os métodos com o nome "entregaMsg()" da classe "Mensageiro".

O adendo por sua vez (linhas 18 a 20), declara o código que deve ser executado quando o ponto de junção definido no ponto de atuação "entregaMensagem()" for alcançado. Neste exemplo, o uso de "*before*" estabelece que a mensagem "Olá" deve ser exibida antes da execução do ponto de junção, ou seja, antes da chamada ao método "entregaMsg()" da classe "Mensageiro".

A declaração de um aspecto é bastante semelhante à declaração de uma classe e serve como contêiner para o código referente a pontos de junção, pontos de atuação e adendos. O aspecto "AspectoMensageiro" é declarado na linha 10 da Figura 2.7.

Nesse exemplo, a cada chamada ao método "entregaMsg()", o fluxo de execução do programa é interrompido e é desviado para que o adendo atribuído ao ponto de atuação "entregaMensagem()", seja executado. A execução do adendo imprime a mensagem "Olá" antes que a mensagem de "entregaMsg()", seja impressa.

```
 1 /*
 2 * Aspecto para exibir mensagem Ola antes da execução
 3 * dos métodos de entregaMsg da classe Mensageiro
 4 */
 5
 6 // Definição do pacote
 7 package com.aopbook;
 8
 9 // Definição do aspecto mensageiro
10 public aspect AspectoMensageiro {
11
12     //Definição do pointcut entregaMensagem
13     pointcut entregaMensagem() : call(* Mensageiro.entregaMsg(..));
14     /*
15     * O código a seguir será executado antes do
16     * poincut entrega mensagem
17     */
18     before() : entregaMensagem() {
19         System.out.println("Olá ");
20     }
21
22 }
```

Figura 2.7 - Aspecto AspectoMensageiro

## 2.4 Algumas Ferramentas de Orientação a Aspectos para Java

Nas subseções seguintes é feita uma breve descrição de algumas ferramentas de Orientação a Aspectos para a linguagem Java.

### 2.4.1 AspectJ

O AspectJ é uma extensão direta da orientação a aspectos para a linguagem de programação Java. Possibilita uma modularização limpa de cruzamento de preocupações, como verificação e manipulação de erros, sincronização, otimização de desempenho, monitoramento e *logging*, suporte à depuração e protocolos de objetos múltiplos (ASPECTJ).

Devido ao AspectJ ser uma extensão para a linguagem Java, há uma preocupação pela compatibilidade de quatro itens:

- Compatibilidade total - todo programa Java válido é necessariamente um programa AspectJ válido;
- Compatibilidade de plataforma - todo programa AspectJ pode ser executado em uma máquina virtual Java (JVM);
- Compatibilidade de ferramentas - deve ser possível estender ferramentas (ferramentas de desenvolvimento, documentação e projeto) existentes para suportar o AspectJ de uma forma natural;
- Compatibilidade para o programador - a programação com AspectJ deve ser natural como se estivesse utilizando uma extensão da linguagem Java (WINCK, 2006).

O AspectJ é dividido em duas partes: a linguagem de especificação (define a linguagem na qual o código é escrito) e a linguagem de implementação. Com AspectJ, as preocupações funcionais são implementados em Java, e para implementação de preocupações sistêmicas são utilizadas as extensões disponibilizadas pelo próprio AspectJ, além de fornecer ferramentas para compilação, *debug* e integração com ambientes integrados de desenvolvimento.

O AspectJ implementa preocupações sistêmicas estáticas e dinâmicas. Nas preocupações dinâmicas, é onde ocorre a combinação aspectual, que pode atravessar diversos módulos de um sistema para que a preocupação em questão possa ser devidamente implementada (WINCK, 2006).

Além do estilo de programação padrão do AspectJ, visualizado na Figura 2.7, a partir da sua versão 5, foi acrescentado suporte ao estilo de programação baseado em anotação, também chamado de @AspectJ. Esse estilo faz uso das anotações (*Annotations*) introduzidas na versão 5 da linguagem Java.

As *annotations* fazem parte da linguagem Java e podem ser usadas para expressar meta-dados, podendo ser aplicadas para declarar tipos (classe, interface, etc), construtores, métodos, campos, parâmetros e variáveis. São caracterizadas pelo símbolo @, utilizadas com valor, múltiplos valores ou simplesmente como marcação.

No estilo @AspectJ, os aspectos são declarados como classes do Java, usando a anotação @Aspect para identificá-la como um aspecto. As *annotations* permitem que o código do AspectJ seja usado com qualquer compilador Java 5 e ferramentas com suporte a Java, facilitando o trabalho na programação de aspectos (ASPECTJ).

### 2.4.2 Spring AOP

O Spring AOP é um *framework* para POA que possibilita a implementação de aspectos customizados, complementando o uso da orientação a objetos com POA. O Spring AOP é executado em Java puro, não necessitando de um processo especial de compilação (SPRING AOP).

Apesar de não oferecer a solução mais completa para POA, o Spring AOP provê soluções eficientes para a maioria dos problemas englobados pela POA, exceto quando se pretende obter objetos muito granulados, para esses casos a melhor alternativa é o AspectJ.

O AspectJ é mais abrangente que o Spring AOP, porém eles não competem entre si, e sim se complementam. O Spring AOP é uma camada de abstração do AspectJ, o que permite que todos os usos de POA possam ser implementados utilizando-se o Spring AOP. No Spring AOP pode-se optar entre dois estilos de programação: configuração XML Spring ou @AspectJ.

Através de uma biblioteca oferecida pelo AspectJ para relacionar os pontos de atuação, o Spring AOP é capaz de interpretar as mesmas *Annotations* utilizadas pelo AspectJ. Para usar o Spring AOP com estilo de programação baseado em @Aspect utiliza-se um mecanismo que interpreta e combina os aspectos à aplicação (SPRING AOP).

### 2.4.3 JAC

JAC (Java *Aspect Components*) é um projeto que consiste em desenvolver uma camada de aplicação Orientada a Aspectos. Este *framework* permite construir aspectos dinâmicos e distribuídos (RESENDE, 2005).

Aplicações atuais em J2EE nem sempre produzem um resultado satisfatório ao separar preocupações sistêmicas do código do sistema. Com JAC, os complexos componentes EJBs são substituídos por POJOs (*Plain Old* Java *Objects*) e implementações de preocupações sistêmicas que estão usualmente implementadas ao longo do código são substituídas por componentes de aspectos.

A manipulação de conceitos é atingida estendendo-se as classes do *framework* e usando os métodos disponíveis pelo JAC (JAC). Há dois níveis de programação orientada a aspectos com JAC:

- Nível de programação: onde pode-se programar novos aspectos. Neste nível, os programadores podem criar novos aspectos para implementar as preocupações sistêmicas.
- Nível de configuração: onde aspectos já existentes podem ser customizados para funcionar com aplicações existentes. Este nível é suportado por uma linguagem de configuração com uma sintaxe genérica que permite que o programador chame métodos de configuração em aspectos existentes.

Na filosofia do JAC, é importante destacar que não é necessário programar aspectos para usar as características da Orientação a Aspectos. O JAC oferece um conjunto de aspectos com métodos de fácil configuração.

Como exemplo de uso da Programação Orientada a Aspectos será implementado um sistema de auditoria para aplicações Java. Como a auditoria é apenas parte de um sistema completo, será implementada uma aplicação em Java de gerência de contas bancárias que servirá como demonstração do seu funcionamento.

O sistema de auditoria será implementado utilizando Spring AOP, pois é implementado em Java puro, tornando mais simples a programação dos aspectos. O Spring AOP é compatível com ferramentas de desenvolvimento, documentação e projeto oferecidas para Java. Através do Spring AOP, o sistema de auditoria poderá ser integrado com facilidade a outras aplicações Java, por meio da configuração de um arquivo XML

# Capítulo 3
# SISTEMA DE AUDITORIA PARA APLICAÇÕES JAVA

Como exemplo de emprego da Orientação a Aspectos será utilizada a atividade de auditoria, pela facilidade em se visualizar os problemas ocorridos nessa situação, como o entrelaçamento e o espalhamento de código. Também será implementada uma aplicação de gerência de contas bancárias em Java como caso de uso para demonstrar o funcionamento do sistema de auditoria.

Este estudo de caso demonstra como a POA pode ser utilizada para implementar as preocupações transversais, neste caso a auditoria. O estudo também possibilita visualizar como a POA interfere no reuso de código, na manutenção e evolução de um sistema.

## 3.1 Detalhes do Sistema de Auditoria

Auditoria, *logging*, nada mais é do que a atividade de registrar cronologicamente as operações executadas em um *software*. Um registro de operação (*log*) é gerado toda vez que uma tarefa é executada. No registro podem ser armazenadas informações como: usuário que realizou a operação, data e horário, a operação executada, dentre outras.

Um sistema de auditoria permite que as informações armazenadas sejam posteriormente analisadas. Essas informações podem ser utilizadas para diversos fins, possibilitando, por exemplo, que possíveis erros ou problemas de segurança sejam identificados. O sistema é manipulado por um administrador da aplicação, que interpreta e monitora as informações contidas nos registros de maneira adequada.

Os registros geralmente são armazenados em arquivos de texto, mas também podem ser armazenados em outros formatos ou em banco de dados, conforme configuração do sistema. O sistema de auditoria implementado apresenta uma interface flexível, que permite escolher o modo de armazenamento dos registros, podendo ser em banco de dados ou em arquivo de texto.

A atividade de auditoria é um dos exemplos mais comuns de ocorrência de código espalhado, pois o código que implementa a auditoria é encontrado basicamente ao longo de

toda a aplicação, atravessando vários módulos. Um código como esse, é de difícil manutenção e complexa compreensão, visto que, o código da auditoria encontra-se misturado a códigos que implementam outras preocupações, obrigando o programador a compreender todas essas preocupações que se encontram entrelaçadas.

A auditoria é apenas uma dentre muitas atividades em que se pode aplicar a Programação Orientada a Aspectos. Embora existam várias atividades que apresentem os mesmos problemas que a auditoria, muitas delas são difíceis de serem demonstradas.

Aplicando-se a teoria de Separação de Preocupações, a atividade de auditoria pode ser considerada uma preocupação sistêmica e as funcionalidades da aplicação principal são vistas como preocupações funcionais. Com isso, usando a POA, a auditoria é implementada através de aspectos de maneira independente da implementação das preocupações funcionais.

O sistema de auditoria implementado registra as operações realizadas ao longo da execução da aplicação, exibindo que operação foi realizada, data e hora, usuário que realizou a operação, o estado da operação (se executou com sucesso ou falha) e o tipo de erro, caso tenha ocorrido. O registro das operações é importante, pois permitirá que o administrador do sistema identifique posteriormente possíveis problemas na aplicação, como por exemplo, a falha na execução de uma operação.

O sistema de auditoria não funcionará somente com uma aplicação específica, mas com qualquer aplicação Java por meio do uso do *framework* Spring AOP, proporcionando todos os benefícios obtidos com o uso da POA. Neste caso, com a finalidade de demonstrar o funcionamento do sistema, será utilizada uma aplicação de gerência de contas bancárias.

Além de seu uso não estar restrito a uma única aplicação, outra característica deste sistema de auditoria é a facilidade de integrá-lo às outras aplicações. A integração do sistema à uma determinada aplicação é feita por meio da configuração de um arquivo XML e do uso de *Annotations*, que serão vistos nas próximas seções. Outro fator importante apresentado no sistema é a flexibilidade no modo de armazenamento das informações obtidas dos registros.

A implementação do sistema de auditoria com POA, tem como objetivo apresentar uma solução eficaz para a atividade de *logging*, que seja facilmente integrável a outras aplicações Java e proporcione as vantagens obtidas com o uso da POA.

Para o desenvolvimento do sistema de auditoria e da aplicação de gerência de contas bancárias foi utilizado o ambiente de desenvolvimento Eclipse, na plataforma Windows. Tanto o sistema de auditoria quanto o gerenciador de contas bancárias, foram implementados na linguagem Java. O sistema de auditoria fez uso do *framework* Spring AOP, já a aplicação

de gerência de contas bancárias foi implementada utilizando os *frameworks* Spring, Struts e Hibernate e o banco de dados MYSQL.

## 3.2 Detalhes do Caso de Uso

Como caso de uso para demonstrar o funcionamento do Sistema de Auditoria, desenvolveu-se uma aplicação de gerência de contas bancárias em Java. Nesta seção serão apresentadas as características do caso de uso.

O gerenciador de contas bancárias é uma aplicação WEB, que tem como objetivo registrar e controlar transações financeiras. Basicamente essa aplicação permite o acompanhamento das movimentações realizadas em contas bancárias, como entrada e saída de dinheiro.

Entre as funcionalidades da aplicação estão: cadastro de usuários, contas e de transações financeiras. A aplicação é acessada por usuários previamente cadastrados, com uso de *login* e senha.

Na Figura 3.1 é mostrada a arquitetura da aplicação de gerência de contas bancárias.

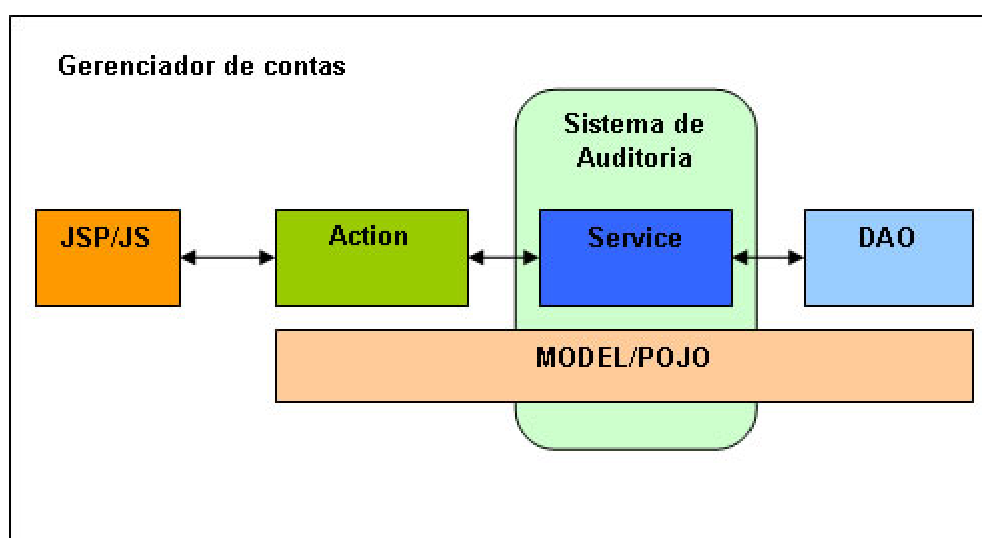



Figura 3.1 - Arquitetura da aplicação

Será descrita a arquitetura utilizada no gerenciador de contas bancárias para um melhor entendimento de como a POA atua. Foi adotada esta arquitetura por propiciar classes mais desacopladas, reduzindo a dependência entre elas. Apesar de propiciar a redução de dependência entre as classes, ainda não é obtido o isolamento necessário para que a atividade de auditoria fosse implementada utilizando-se somente programação orientada a objetos.

As camadas da aplicação, ilustradas na Figura 3.1, são:

- JSP/JS: possui todas as páginas JSPs (Java *Server Pages*) e arquivos JS (JavaScript) responsáveis pela entrada e apresentação dos dados. São as páginas acessadas, através do navegador de internet, pelo usuário da aplicação.
- *Model*: possui as entidades/POJOS (*Plain Old* Java *Objects*) utilizados para a troca de dados na maioria das camadas. A Figura 3.2 mostra o diagrama de classes desta camada. Abaixo são citadas as classes que compõem a camada *Model*.

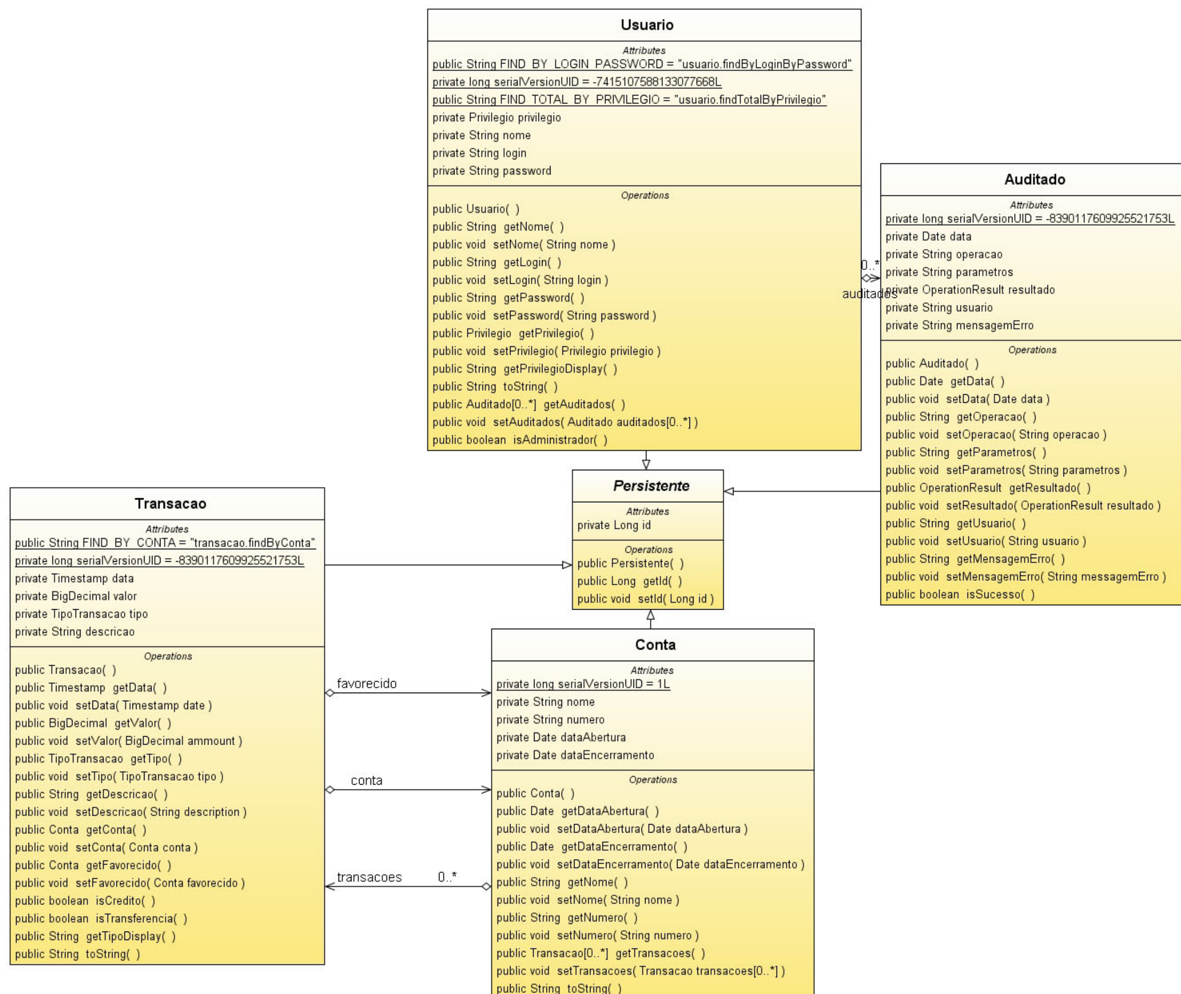



Figura 3.2 – Diagrama de Classes da Camada *Model*

- o “Persistente”: classe de onde são estendidas todas as outras classes desta camada, contém o atributo “id”, identificador. Esta classe serve para identificar quais classes devem ser persistidas.
- o “Usuario”: classe responsável por conter os atributos do usuário que fará uso da aplicação.
- o “Conta”: classe que contém os atributos referentes às contas bancárias.
- o “Transacao”: classe responsável por conter os atributos das transações (débito, crédito e transferência) possíveis de serem realizadas nas contas.
- o “Auditado”: classe responsável pelas informações de auditoria. Esta classe não seria necessária caso as informações de auditoria fossem armazenadas em arquivo de texto. Neste caso as informações de auditoria estão sendo armazenadas em banco de dados (no mesmo banco da aplicação de gerência de contas), por isso a necessidade desta classe.

- *Service*: camada de serviço, onde ficam as lógicas de negócio. As chamadas nessa camada geralmente são transacionais (no caso está sendo usado o gerenciamento de transações do Spring, que por sua vez é baseado em POA e *Annotations*). Algumas das principais classes presentes nessa camada, ilustradas no diagrama de classes da Figura 3.3, são citadas na seqüência.

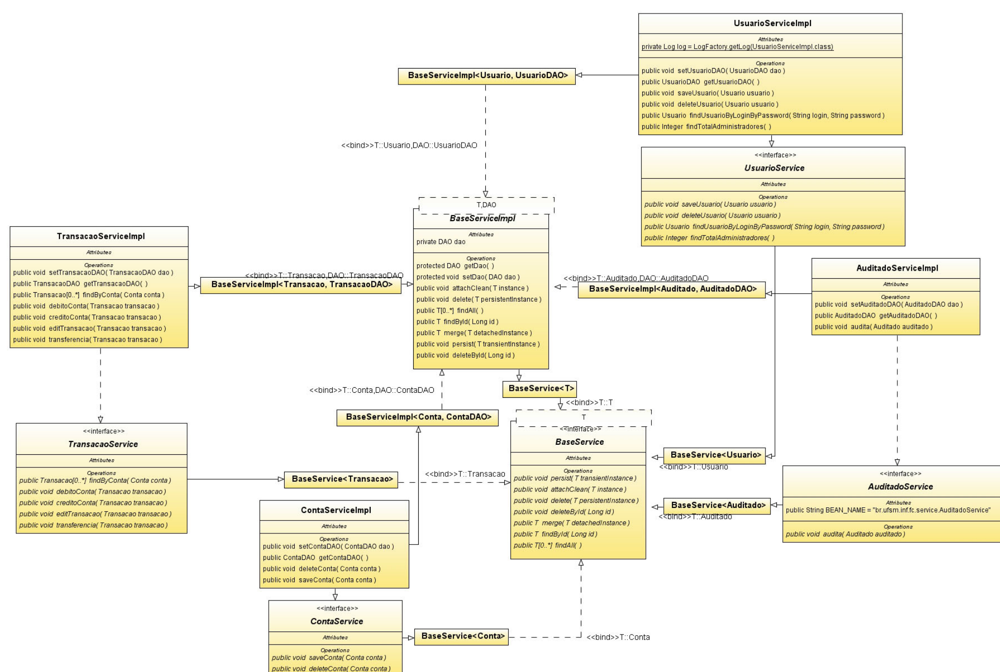



Figura 3.3 – Diagrama de Classes da Camada *Service*

É nessa camada onde estão definidos os comportamentos da aplicação, onde estão as preocupações funcionais da aplicação. O sistema de auditoria tem efeito sobre as classes dessa camada, gerando um *log* quando determinadas operações forem executadas.

- o “TransacaoServiceImpl”: uma das classes que compõe esta camada, está destacada na Figura 3.4. É onde estão implementados os métodos que realizam as transações de débito, crédito e transferência.
- o “UsuarioServiceImpl”, “ContaServiceImpl”: contém os métodos de cadastro e remoção de usuário e contas, respectivamente.

**TransacaoServiceImpl**

*Attributes*

*Operations*

```
public void  setTransacaoDAO( TransacaoDAO dao )
public TransacaoDAO  getTransacaoDAO( )
public Transacao[0..*]  findByConta( Conta conta )
public void  debitoConta( Transacao transacao )
public void  creditoConta( Transacao transacao )
public void  editTransacao( Transacao transacao )
public void  transferencia( Transacao transacao )
```

Figura 3.4 – Classe “TransacaoServiceImpl” da Camada *Service*

- *Action*: responsável por controlar o fluxo de chamadas vindas do JSP, localizar e chamar a respectiva *Service* e montar os resultados para apresentação de dados. Para cada classe da camada *Model* existe uma classe de *Action*, permitindo que seja feita a comunicação entre as chamadas do JSP e a *Service* correspondente. A Figura 3.5 representa o diagrama de classes desta camada.
- DAO: nessa camada fica a lógica de acesso a dados, onde são mantidos os códigos de persistência e de consulta a dados. É nessa camada onde estão as consultas ao banco.
- Sistema da Auditoria: o sistema de auditoria é ligado à camada *Service* via POA. Essa ligação é feita usando-se marcações nos métodos das classes da camada *Service* e pela especificação dos pontos de junção no aspecto responsável pela auditoria.

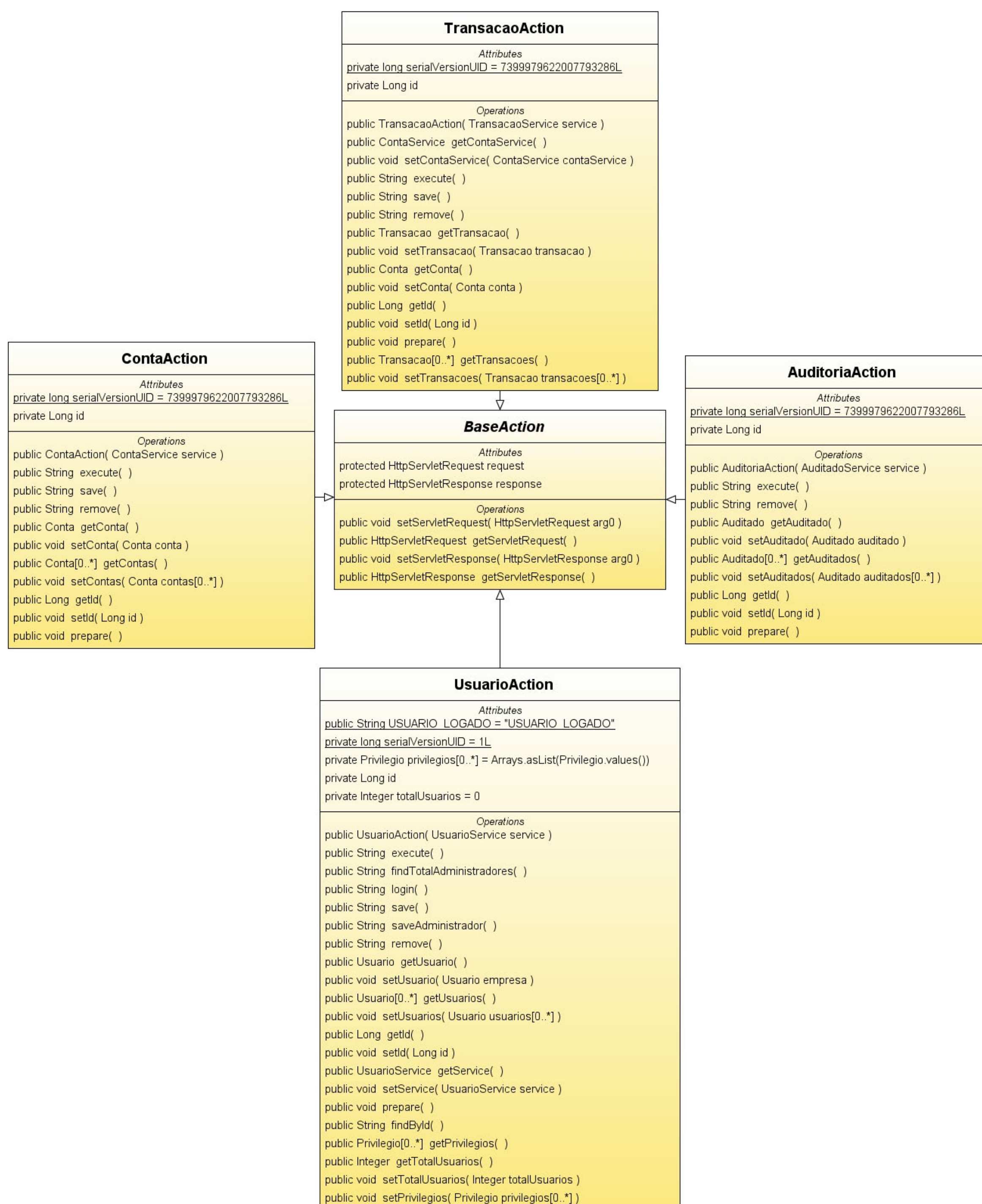



Figura 3.5 – Diagrama de Classes da Camada *Action*

Nessa aplicação é usado o *framework* Spring, que implementa a ligação entre as camadas usando inversão de controle (IoC[1]), faz o controle de transações e possibilita o

[1] Inversão de Controle (IoC) é quando o controle de chamadas a métodos não é definida pelo programador, e sim por um contêiner.

funcionamento da POA no sistema de auditoria (com Spring AOP). Além do Spring, outros dois *frameworks* também são usados, são eles: Struts e Hibernate. Para armazenar os dados da aplicação é utilizado o banco de dados MYSQL.

Caso a aplicação fosse implementada somente com orientação a objetos, o sistema de auditoria estaria implementado junto à lógica de negócio, encontrando-se ao longo das camadas *Service* e DAO, entrelaçado ao código da aplicação. Sem o uso da POA, as chamadas a métodos de auditoria estariam espalhadas na camada *Service*, sendo chamados explicitamente em classes que implementam outras funcionalidades.

Relacionando a aplicação de gerência de contas bancárias com os conceitos vistos no capítulo anterior, suas funcionalidades são consideradas como preocupações funcionais, implementadas através de componentes, neste caso, classes. Já, o sistema de auditoria é considerado uma preocupação sistêmica da aplicação, sendo implementado com POA através de aspectos.

O sistema de auditoria registra as ações realizadas na aplicação de gerência de contas bancárias. As operações sujeitas à auditora nessa aplicação são:

- Cadastro/Remoção de usuários
- Cadastro/Remoção de contas
- Cadastro de transações financeiras, como: débito, crédito, transferência.

Os métodos que implementam as operações citadas acima, que terão suas execuções monitoradas pelo sistema de auditoria, são considerados os pontos de junção da aplicação. Nesses pontos a execução da aplicação é interrompida para executar outra atividade, que neste caso será a criação de um *log*. As operações sujeitas à auditoria encontram-se implementadas nas classes da camada *Service,* cujos códigos serão vistos mais adiante.

## 3.3 Arquitetura do Sistema de Auditoria

A arquitetura do sistema de auditoria é mostrada no diagrama de classes, representado na Figura 3.6.

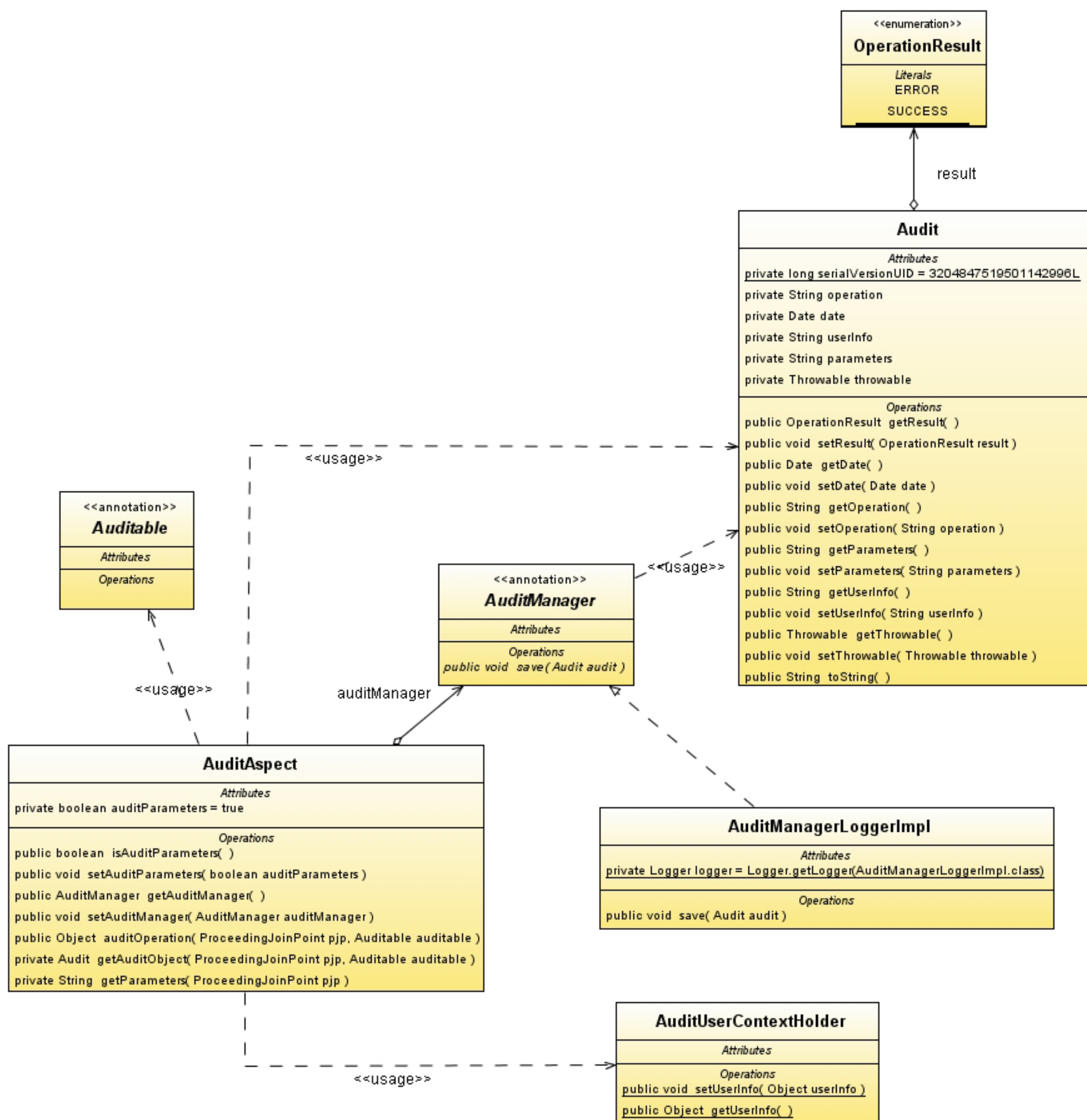



Figura 3.6 - Diagrama de classes

O diagrama é composto pelos seguintes componentes:

- “AuditAspect”: é a principal classe do sistema de auditoria, onde está implementado o aspecto. No “AuditAspect” estão definidas as regras dos pontos de junção e também está implementado o comportamento responsável por gerar o *log*.
- “Auditable”: é a *Annotation* que será usada para definir os pontos de junção dentro do “AuditAspect”. A anotação “Auditable” será utilizada posteriormente, dentro da aplicação, para marcar as operações que sofrerão auditoria, possibilitando relacionar as operações aos pontos de junção.

- “AuditManager”: é a interface que possui a assinatura do método “save”, responsável por salvar um objeto do tipo “Audit”, ou seja, as informações da auditoria. A interface “AuditManager” é utilizada dentro do “AuditAspect”. O uso dessa interface permite que o programador escolha uma implementação para o método “save”, dando maior flexibilidade para o modo de armazenamento das informações.
- “AuditManagerLoggerImpl”: é a classe que implementa o método “save” da interface “AuditManager”, essa implementação salva as informações da auditoria em arquivo de texto. O sistema de auditoria oferece essa implementação, mas outras implementações podem ser utilizadas, alguma que armazene em banco de dados, por exemplo.
- “Audit”: é a classe que define os dados a serem gravados no registro de operação. Nesse caso, os dados que estarão presentes no registro de operação, são: data, operação, parâmetros da operação, dados do usuário, resultado da operação e dados de erro.
- “OperationResult”: é utilizado para enumerar os possíveis resultados de uma operação, sucesso ou erro. Está presente na classe “Audit”, onde um dos valores é atribuído à propriedade “operationResult”.
- “AuditUserContextHolder”: é a classe usada para passar ao “AuditAspect” as informações do usuário que realiza uma operação, para que então possam ser gravadas no registro de operação. Para obter as informações, a “AuditUserContextHolder” se baseia no usuário autenticado no contexto de execução corrente (*thread* corrente). Este usuário deve ser armazenado explicitamente no contexto corrente usando AuditUserContextHolder.setUserInfo(usuario).

## 3.4 Implementação do Sistema de Auditoria

O sistema de auditoria foi implementado com o *framework* Spring AOP. O Spring AOP foi utilizado com estilo de programação @AspectJ. Aplicações que fazem uso do *framework* Spring podem ser facilmente integradas ao sistema de auditoria. A implementação realizada neste trabalho segue a estrutura mostrada no capítulo anterior, fazendo uso de:

- Linguagem de componentes: Java é a linguagem utilizada para implementar as preocupações funcionais, ou seja, as funcionalidades do gerenciador de contas bancárias.
- Linguagem de aspectos: Spring AOP é o *framework* utilizado para implementar os aspectos, permitindo o uso da POA no sistema de auditoria.
- Programas escritos em linguagem de componentes: implementação do gerenciador de contas.
- Programas escritos em linguagem de aspectos: implementação do sistema de auditoria.

Para que o sistema possa cumprir o seu papel adequadamente em uma aplicação, o mesmo precisa atender alguns requisitos. O sistema de auditoria deve ser capaz de:

- se integrar a uma aplicação Java;
- identificar as operações que precisam de auditoria;
- coletar todas as informações relevantes, que devem fazer parte do registro de operação;
- e persistir as informações em arquivo de texto ou banco de dados.

Na Figura 3.7, é mostrado o aspecto de auditoria implementado com Spring AOP. Está sendo utilizado o estilo de anotação @AspectJ. Nessa classe pode-se identificar os conceitos apresentados no capítulo anterior.

- Pontos de junção: todos os métodos públicos que tenham a marcação "auditable". *(public * *(..) && @annotation(auditable)*
- Ponto de atuação: compreende os pontos de junção.
- Adendo: inclui desde o *@Around* até o fim do "auditOperation" (da linha 33 até a linha 52). O comportamento implementado no adendo é executado durante o ponto de atuação definido dentro do *@Around*. O adendo é responsável por implementar o comportamento de auditoria na aplicação, que grava um *log* toda vez que um ponto de junção é atingido.

Na Figura 3.8 pode-se visualizar o uso de *annotations* (nas linhas 24 e 30) que serve como marcação dos métodos que devem sofrer auditoria. Essa classe não possui códigos de auditoria, evitando assim os problemas de espalhamento e entrelaçamento de código. O uso de

annotations em conjunto com a POA facilita a definição dos pontos de junção, ou seja, em quais operações um *log* deve ser gerado.

```
applicationContext.xml    AuditAspect.java
11 @Aspect
12 public class AuditAspect {
13
14     private AuditManager auditManager;
15     private boolean auditParameters = true;
16
17⊕    public boolean isAuditParameters() {
20
21⊕    public void setAuditParameters(boolean auditParameters) {
24
25⊕    public AuditManager getAuditManager() {
28
29⊕    public void setAuditManager(AuditManager auditManager) {
32
33⊖    @Around(value = "execution(public * *(..)) && @annotation(auditable)",
34            argNames = "pjp,auditable")
35     public Object auditOperation(ProceedingJoinPoint pjp,
36                                 Auditable auditable) throws Throwable {
37         Audit audit = getAuditObject(pjp, auditable);
38         Object retVal;
39         try {
40             retVal = pjp.proceed();
41         } catch (Throwable e) {
42             System.out.println("Resultado: Falha. Detail: " + e.getMessage());
43             audit.setResult(OperationResult.ERROR);
44             audit.setThrowable(e);
45             auditManager.save(audit);
46             throw e;
47         }
48         System.out.println("Resultado: Sucesso");
49         audit.setResult(OperationResult.SUCCESS);
50         auditManager.save(audit);
51         return retVal;
52     }
```

Figura 3.7 - Aspecto implementado com Spring AOP

Essa é uma classe que implementa as funcionalidades do objeto Usuário, que são operações de criar e remover usuário. Essa classe encontra-se na camada de Serviço (vista na seção 3.2). Neste caso a cada criação de um novo usuário, realizada pelo método “saveUsuario(Usuario usuario)”, será gerado um *log* com os valores passados por parâmetros e com a mensagem “Criar Usuário”, descrevendo qual operação está sendo executada.

As *annotations* @auditable(“...”) definem a mensagem a ser gravada no *log* para cada tipo de operação, bem como quais métodos terão suas execuções monitoradas. Essas

marcações não possuem nenhuma semântica que altere o fluxo do programa. Se aplicadas corretamente, as *annotations* possibilitam simplificar a programação de aspectos, complementando o uso da POA.

Caso as *annotations* @auditable("...") não tivessem sido usadas, seria difícil afirmar que algum comportamento adicional está sendo executado, somente olhando para o código dessa classe. Esse problema é chamado de *obliviousness*, citado no capítulo anterior. As *annotations* reduzem a ocorrência desse problema.

```
23
24  @Auditable(operation="Criar Usuário")
25  public void saveUsuario(Usuario usuario) {
26      super.persist(usuario);
27
28  }
29
30  @Auditable(operation="Remover Usuário")
31  public void deleteUsuario(Usuario usuario) {
32      super.delete(usuario);
33
34  }
```

Figura 3.8 - Código com POA e Annotations

Por sua implementação estar modularizada, isolada do resto da aplicação, o sistema de auditoria também pode ser utilizado com outras aplicações que usam o *framework* Spring.

### 3.5 Configuração do Sistema de Auditoria

A habilitação do uso do sistema de auditoria em uma aplicação é feita através da configuração de um arquivo XML do Spring AOP. Na Figura 3.9 pode-se visualizar esse arquivo ("applicationContext.xml"), que será usado em qualquer aplicação Java.

É nesse arquivo onde são configurados os aspectos e todas as questões relativas à POA a serem utilizados por uma aplicação. Neste caso está sendo usado o arquivo de configuração da aplicação de gerência de contas bancárias.

Figura 3.9 - Arquivo de configuração

Nas linhas 13 e 14, é feito o mapeamento da interface “AuditManager” para a classe que a implementa. Através desse mapeamento é possível definir quais implementações serão utilizadas. Pode-se perceber que está sendo usada uma implementação própria da aplicação de gerência de contas bancárias, a “AuditManagerDataBaseImpl”.

O “AuditAspect” é o aspecto usado na aplicação e definido a partir da linha 16, onde também é feito o mapeamento da classe que o implementa. Nas linhas seguintes, são atribuídos os valores para as propriedades do aspecto. Na propriedade “AuditManager” é feita a referência ao *bean* utilizado, que foi previamente definido na linha 13. Na propriedade “AuditParameters” é atribuído o valor *true*, que define que os valores passados nos parâmetros dos métodos que sofrem auditoria também devem ser gravados no *log*.

O sistema de auditoria pode ser integrado a outras aplicações Java simplesmente realizando essas configurações no arquivo "applicationContext.xml" de acordo com cada aplicação. Por exemplo, para usar o sistema de auditoria com outra aplicação, somente o valor de "class" da linha 14 precisaria ser alterado. Usando a implementação que grava as informações de auditoria em arquivo de texto, já implementada no próprio sistema de auditoria, ficaria "class=br.ufsm.inf.auditmanager.AuditManagerLoggerImpl".

## 3.6 Funcionamento do Sistema de Auditoria com o Caso de uso

O funcionamento do sistema de auditoria será demonstrado com a aplicação de gerência de contas bancárias. Nessa aplicação foi feita uma implementação da interface "Auditmanager". A "AuditManagerDataBaseImpl", classe que implementa a interface, salvando as informações de auditoria no mesmo banco de dados da aplicação. Essa implementação foi feita com o objetivo de facilitar a emissão de relatórios e consultas (neste caso, via WEB).

A Figura 3.10 mostra a página da aplicação, onde são cadastradas as transações realizadas nas contas. No momento da execução dessas transações, também ocorre a gravação de um *log*, para cada uma delas.

A auditoria permite que se tenha um relatório cronológico das atividades realizadas na aplicação. A cada transação (débito, crédito ou transferência) realizada é gerado um *log* com informações sobre a operação. Na Figura 3.11 é possível visualizar as transações que foram realizadas através da página mostrada anteriormente.

Pelos registros podem ser visualizadas as informações de cada operação executada e saber a ordem em que elas aconteceram. Outra informação contida nesse relatório é a ocorrência de falha na operação e qual a exceção gerada. Neste caso, tentou-se realizar uma operação de débito em uma conta, sem informar o valor para a mesma. Com isso foi gerada uma exceção informando que o valor passado não pode ser nulo.

Esses registros permitem que os erros sejam mais facilmente identificados, possibilitando assim que a aplicação seja corrigida o quanto antes. Caso esses registros não fossem gerados, talvez o administrador ou o programador responsável nem tomasse conhecimento dessa falha até que algum usuário da aplicação reportasse o problema.

http://localhost:8080/fluxocaixa/pages/transacoes.jsp

**Gerenciador de Contas: Transações**

**Selecione uma conta:**

*Conta:* Conta Corrente BB

Selecionar

**Dados da transação:**

*Descrição:*

*Valor:*

*Tipo:* Débito

*Destino:* -- Selecione --

Cadastrar

Atualizar

**Conta:** Conta Corrente BB

**Transações:**

| Data | Descrição | Débito | Crédito | Saldo | Remover/Editar |
|---|---|---|---|---|---|
| 14/08/2007 | depósito | | 200,00 | 200,00 | Remover \| Editar |
| 16/08/2007 | transf. | 100,00 | | 100,00 | Remover \| Editar |
| 16/08/2007 | receb. | | 120,00 | 220,00 | Remover \| Editar |
| 16/08/2007 | pagto condomínio | 123,68 | | 96,32 | Remover \| Editar |
| 16/08/2007 | pagamento | 125,99 | | -29,67 | Remover \| Editar |

Figura 3.10 - Página de cadastro das movimentações

http://localhost:8080/fluxocaixa/pages/admin/auditoria.jsp

**Gerenciador de Contas: Relatório de Auditoria**

| Data | Operação | Parâmetros | Usuário | Status | Erro |
|---|---|---|---|---|---|
| 16/08/07 21:37:11.000 | Débito em Conta | [Transação: {Descrição=pagto condomínio, Tipo=Débito, Valor=123.68, Data=2007-08-16 21:37:11.75, Conta=Conta: {Nome=Conta Corrente BB, Número=2 , Data Abertura=2007-08-13 06:47:30.0}] | Administrador/admin | Sucesso | |
| 16/08/07 21:39:50.000 | Débito em Conta | [Transação: {Descrição=pagamento, Tipo=Débito, Valor=125.99, Data=2007-08-16 21:39:50.328, Conta=Conta: {Nome=Conta Corrente BB, Número=2 , Data Abertura=2007-08-13 06:47:30.0}] | Administrador/admin | Sucesso | |
| 16/08/07 22:15:11.000 | Débito em Conta | [Transação: {Descrição=pagamento luz, Tipo=Débito, Valor=null, Data=2007-08-16 22:15:11.312, Conta=Conta: {Nome=Conta Corrente BB, Número=2 , Data Abertura=2007-08-13 06:47:30.0}] | Administrador/admin | Falha | org.hibernate.PropertyValueException: not-null property references a null or transient value: br.ufsm.inf.fc.model.Transacao.valor |

Figura 3.11 - Registros das operações

## 3.7 Comparação do Sistema de Auditoria com e sem POA

Quando se desenvolve um *software* baseado somente em programação orientada a objetos, por exemplo, tende-se a implementar a auditoria como regra de negócio da aplicação, o que acaba gerando os problemas de entrelaçamento e espalhamento em vários locais do código. Esses problemas poderão ser visualizados mais adiante.

A Figura 3.12 ilustra a diferença entre uma implementação que não faz uso da POA e outra, baseada na POA. Na ilustração da esquerda, sem POA, é possível perceber as preocupações transversais misturadas ao código da aplicação. À direita as preocupações transversais são isoladas e implementadas isoladamente na forma de aspectos, evitando o emaranhamento do código e os problemas decorrentes dessa situação.

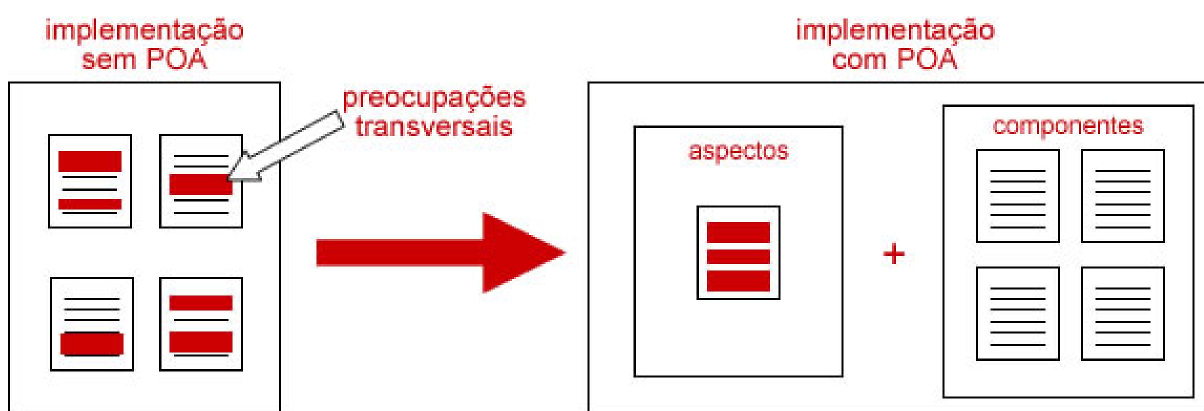
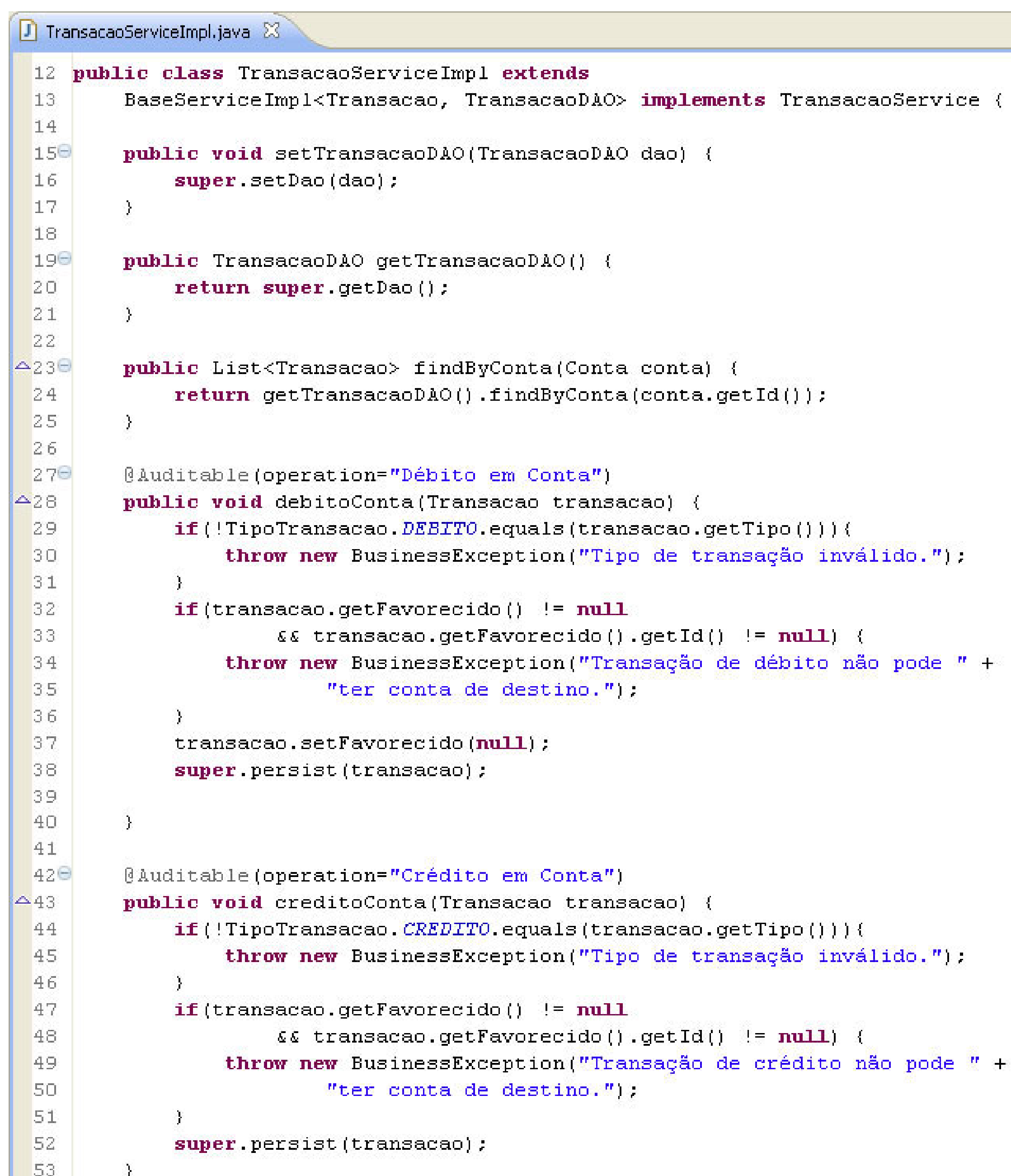



Figura 3.12 - Comparação de aplicações com e sem POA

Seguindo a arquitetura mostrada na Figura 3.1 da seção 3.2, se o sistema de auditoria fosse implementado sem o uso da POA, estaria ao longo da camada *Service*, misturando-se ao código da aplicação. Os métodos de auditoria estariam sendo chamados explicitamente em classes que implementam outros propósitos. A aplicação seria semelhante a da implementação sem POA da Figura 3.12.

Os trechos de códigos exibidos nas Figuras 3.13 (com POA) e 3.14 (sem POA), esta utilizando somente programação orientada a objetos, permitem visualizar as diferenças entre a implementação que faz uso da POA e a implementação clássica. A classe “TransaçãoServiceImpl” é responsável por implementar os métodos de Transação (crédito,

débito e transferência) em ambos os casos. Porém pode-se perceber claramente a diferença entre os dois códigos, o primeiro, que faz uso da POA, está mais limpo que o segundo, e conseqüentemente mais legível.



```
TransacaoServiceImpl.java
12 public class TransacaoServiceImpl extends
13     BaseServiceImpl<Transacao, TransacaoDAO> implements TransacaoService {
14
15     public void setTransacaoDAO(TransacaoDAO dao) {
16         super.setDao(dao);
17     }
18
19     public TransacaoDAO getTransacaoDAO() {
20         return super.getDao();
21     }
22
23     public List<Transacao> findByConta(Conta conta) {
24         return getTransacaoDAO().findByConta(conta.getId());
25     }
26
27     @Auditable(operation="Débito em Conta")
28     public void debitoConta(Transacao transacao) {
29         if(!TipoTransacao.DEBITO.equals(transacao.getTipo())){
30             throw new BusinessException("Tipo de transação inválido.");
31         }
32         if(transacao.getFavorecido() != null
33                 && transacao.getFavorecido().getId() != null) {
34             throw new BusinessException("Transação de débito não pode " +
35                     "ter conta de destino.");
36         }
37         transacao.setFavorecido(null);
38         super.persist(transacao);
39
40     }
41
42     @Auditable(operation="Crédito em Conta")
43     public void creditoConta(Transacao transacao) {
44         if(!TipoTransacao.CREDITO.equals(transacao.getTipo())){
45             throw new BusinessException("Tipo de transação inválido.");
46         }
47         if(transacao.getFavorecido() != null
48                 && transacao.getFavorecido().getId() != null) {
49             throw new BusinessException("Transação de crédito não pode " +
50                     "ter conta de destino.");
51         }
52         super.persist(transacao);
53     }
```

Figura 3.13 - Implementação com POA

Na Figura 3.14 pode-se visualizar como seria implementada a auditoria sem o uso da POA. Percebe-se que os códigos que implementam auditoria se encontram no interior da classe “TransacaoServiceImpl”, que implementa serviços de transação.

TransacaoServiceImpl.java | TransacaoServiceImpl.java

```
16
17 public class TransacaoServiceImpl extends
18     BaseServiceImpl<Transacao, TransacaoDAO> implements TransacaoService {
19
20     private AuditManager auditManager;
21
22     public AuditManager getAuditManager() {
23       return auditManager;
24     }
25
26     public void setAuditManager(AuditManager auditManager) {
27       this.auditManager = auditManager;
28     }
29
30     public void setTransacaoDAO(TransacaoDAO dao) {
31         super.setDao(dao);
32     }
33
34     public TransacaoDAO getTransacaoDAO() {
35         return super.getDao();
36     }
37
38     public List<Transacao> findByConta(Conta conta) {
39         return getTransacaoDAO().findByConta(conta.getId());
40     }
41
42     public void debitoContaSemAOP(Transacao transacao) {
43         Audit audit = new Audit();
44         audit.setDate(new Date());
45         audit.setOperation("Débito em Conta");
46         if(AuditUserContextHolder.getUserInfo() != null) {
47             audit.setUserInfo(AuditUserContextHolder.getUserInfo().toString())
48         }
49         if(transacao != null) {
50             audit.setParameters(transacao.toString());
51         }
52         try {
53             if(!TipoTransacao.DEBITO.equals(transacao.getTipo())){
54                 throw new BusinessException("Tipo de transação inválido.");
55             }
56             if(transacao.getFavorecido() != null
57                     && transacao.getFavorecido().getId() != null) {
58                 throw new BusinessException("Transação de débito não pode " +
59                         "ter conta de destino.");
60             }
61             transacao.setFavorecido(null);
62             super.persist(transacao);
63             audit.setResult(OperationResult.SUCCESS);
64         } catch (RuntimeException e) {
65             audit.setResult(OperationResult.ERROR);
66             audit.setThrowable(e);
67             throw e;
68       } finally {
69           auditManager.save(audit);
70         }
71     }
```

Figura 3.14 - Implementação sem POA

Todo o código em destaque na Figura 3.14, é desnecessário na implementação com POA. Com POA esse código é centralizado, sendo implementado no aspecto “AuditAspect”, como pôde ser visto na Figura 3.7.

Para realizar a auditoria apenas no método “debitoContaSemAOP” foram adicionadas aproximadamente vinte linhas de código. A cada novo método que faça uso da auditoria deveria ser acrescentado todo o código selecionado dentro do método “debitoContaSemAOP”. Implementando todos os métodos de Transação sem usar a POA, seriam acrescentadas aproximadamente sessenta e cinco linhas de código, em uma única classe.

Aplicando a auditoria para o método “debitoContaSemAOP”, várias linhas de código foram acrescentadas, somente para adicionar a auditoria a uma única operação. Em uma aplicação completa, que necessita de auditoria em muitos métodos de várias classes, pode-se afirmar que uma quantidade significativa de código é acrescentada, gerando códigos replicados.

Além do código extra gerado, essa implementação acarretou a dependência entre várias classes, como as classes “Audit” e “AudituserContextHolder”. Essa dependência pode ser confirmada quando for necessária alguma modificação em “Audit” ou “AudituserContextHolder”, pois implicaria em modificar todas as classes que fazem uso delas.

Toda essa situação caracteriza nitidamente a ocorrência de código espalhado e entrelaçado. O entrelaçamento do código ocorre devido ao código de auditoria estar presente em uma classe que implementa outra preocupação, no caso, as funcionalidades do objeto Transação.

O código espalhado também ocorre, pois esses códigos de auditoria estão presentes em várias classes da camada de serviços. Caso fosse preciso substituir ou modificar a maneira como a auditoria é feita, essas modificações deveriam ser realizadas em todas as classes que possuem as chamadas para os métodos de auditoria.

Além da replicação de código, também é visível o acoplamento entre as classes, visto que as classes ficam dependentes umas das outras. Um código como esse dificulta a manutenção, tornando-se menos legível e aumentando o retrabalho no caso modificações.

Implementando-se a atividade de auditoria com a POA obtém-se um código mais limpo, como mostra a Figura 3.13. Sem códigos de auditoria, reduz-se a quantidade de linhas programadas e os problemas com o entrelaçamento e o espalhamento de código são evitados.

Um código mais limpo facilita a manutenção, já que uma possível modificação na implementação da auditoria não implicaria em modificações nas classes que fazem uso da mesma. Com a manutenção facilitada, vários outros fatores se tornam mais simples, o que acarreta um menor tempo e conseqüentemente menor custo de desenvolvimento.

O reuso do código é aumentado, tanto para o código que implementa a auditoria quanto para o código da aplicação que faz uso da mesma. Por estar em um módulo separado da aplicação, a auditoria pode ser integrada a outras aplicações mais facilmente, através da configuração de arquivos XML e do uso das anotações nas classes que farão uso da mesma.

Também não é mais necessário, na aplicação, entrar no código de cada classe e apagar os códigos que seriam de auditoria, caso fosse necessário desintegrar o sistema de uma aplicação.

Outro benefício da implementação de um sistema com POA, é a possibilidade de novas funcionalidades serem criadas na aplicação sem que seja necessário realizar modificações no aspecto, para que o mesmo tenha efeito sobre o novo código. Os comportamentos implementados no aspecto, passam a funcionar sobre a nova funcionalidade apenas com a inserção de uma *annotation* no método que implementa a mesma.

Se as *annotations* @auditable na classe "TransacaoServiceImpl" não fossem usadas para marcar quais métodos terão suas execuções gravadas no *log*, certamente a adição desse comportamento não seria notado apenas analisando-se a classe em questão. Seria necessário que o programador soubesse previamente que esses métodos estão sujeitos a atividade de auditoria, implementada através de aspectos.

Além de as *annotations* permitirem uma melhor visualização do que pode estar acontecendo em uma classe, essas marcações dão uma maior flexibilidade de evolução ao *software*. Essa maior flexibilidade é possibilitada devido às *annotations* complementarem a definição dos pontos de junção, não sendo necessário alterá-los nos aspectos cada vez que uma nova funcionalidade for adicionada na aplicação.

Caso novas funcionalidades precisem ser incorporadas à aplicação, basta adicionar as *annotations* junto aos métodos que as implementam, permitindo assim que os comportamentos implementados pelos aspectos tenham efeito sobre essas novas funcionalidades. O uso de *annotations* possibilita também definir de maneira mais específica quais métodos necessitam de auditoria.

# Capítulo 4
# CONCLUSÕES

O aumento da complexidade e a produtividade limitada no desenvolvimento de *software* têm obrigado desenvolvedores a buscar soluções para tais limitações. A POA tem ganhado espaço e mostrado ser muito eficiente na solução de problemas que até então eram difíceis de ser tratados.

Através do estudo de caso pôde-se observar como a POA age e perceber as vantagens propiciadas por esse paradigma. Com a POA são obtidos códigos mais reutilizáveis, fáceis de manter, eliminando problemas como replicação de código e entrelaçamento. Isso implica em códigos mais coesos e menos acoplados, além de permitir que novas funcionalidades sejam facilmente agregadas às aplicações. Com a facilidade de manutenção é possível reduzir o tempo gasto com essa atividade e assim aumentar a produtividade no desenvolvimento de *software*.

O sistema de auditoria implementado com POA apresentou uma solução para a atividade de *logging* não invasiva, de fácil manutenção, permitindo que o *software* evolua sem necessitar de muitas modificações. Outro mérito do sistema, que deve ser salientado é a facilidade de integração a outras aplicações Java. Todas essas características do sistema de auditoria foram possíveis de ser obtidas devido ao uso da POA.

Nas próximas seções são apresentadas as principais contribuições deste trabalho, possíveis extensões e algumas considerações finais a respeito do mesmo.

## 4.1 Contribuições

No capítulo inicial foram apresentados os principais fundamentos e conceitos da Orientação a Aspectos. Também foi realizado um levantamento de alguns *frameworks* orientados a aspectos para a linguagem Java, que viessem a contribuir no desenvolvimento desta proposta.

Este trabalho apresenta como principal contribuição a implementação de um sistema de auditoria para aplicações Java. O sistema de auditoria implementado, baseado na POA, tem como principais vantagens a capacidade de utilização com outras aplicações Java e a

facilidade de integração às aplicações. Além dessas características outro fator importante é a flexibilidade no modo de armazenamento das informações, se em banco de dados ou arquivos de texto. A integração do sistema de auditoria pode ser realizada facilmente através da configuração de arquivos XML e do uso de a*nnotations*, o que permite que o sistema seja amplamente utilizado com outras aplicações.

O sistema de auditoria tem como função a geração de um histórico de execução da aplicação, importante para diversos fins. Através da implementação desse sistema, foi possível visualizar como a POA atua numa aplicação, o que permitiu comprovar suas vantagens e eficácia como solução para os problemas decorrentes da falta de modularização.

## 4.2 Estudos Futuros

Este trabalho apresenta possíveis extensões para o sistema de auditoria. A seguir são descritas as extensões e pesquisas que podem ser realizadas, visualizadas a partir do desenvolvimento do sistema de auditoria com POA.

O sistema de auditoria poderia ser implementado com outros *frameworks* de POA, para comparação e análise, possibilitando identificar a implementação mais eficiente para esse tipo de atividade.

Outra extensão possível seria aprimorar o sistema de auditoria, permitindo que novas informações sejam agregadas ao registro de operação e possam ser configuradas de acordo com a necessidade de cada aplicação. Isso proporcionaria uma abrangência maior do uso do sistema de auditoria.

Ao sistema de auditoria, também poderiam ser acrescentados, mecanismos de geração de relatórios das informações armazenadas, como por exemplo, arquivos em pdf, ou envio dos relatórios por e-mail.

## 4.3 Considerações Finais

Obter uma aplicação que pode ser facilmente mantida, sem todos os problemas já citados, torna-se viável com o uso da POA. A POA propicia um aumento significativo de produtividade e está sendo amplamente difundida no desenvolvimento de *software*.

Apesar de o uso da POA dificultar a visualização do que acontece a uma classe que sofre interferência de aspectos, isso pode ser amenizado se em conjunto com a POA forem usadas as *annotations*. As *annotations* além de reduzirem o problema, complementam o uso da POA, ajudando na evolução do *software*.

Uma característica notável do uso da POA é a sua natureza dinâmica diferente dos outros paradigmas. Isso permite que comportamentos sejam inseridos, removidos ou adaptados provendo a evolução dos sistemas.

# REFERÊNCIAS BIBLIOGRÁFICAS


ASPECTJ - The AspectJ Programming Guide. Disponível em <http:/www.eclipse.org/aspectj>. Acesso em: 23 jul. 2007.

GRADECKI, J. D.; LSIECKI, N. **Mastering AspectJ**: Aspect-Oriented Programming in Java. Wiley Publishing, Inc. 2003. 456 p.

JAC - A Framework for Aspect-Oriented Programming in Java. Disponível em <http://jac.objectweb.org/>. Acesso em: 23 abr. 2007.

KICZALES, G.; LAMPING, J.; MENDHEKAR, A.; MAEDA, C.; LOPES, C.; LOINGTIER, J.; IRWIN, J. **Aspect-Oriented Programming**. European Conference on Object-Oriented Programming (ECOOP97). Finlândia, 1997. 25 p. Disponível em <http://www.parc.com/research/projects/aspectj/downloads/ECOOP1997-AOP.pdf>. Acesso em: 16 ago. 2007.

KISELEV, I; **Aspect Oriented Programming With AspectJ**. SAMS, 2002. 288 p.

PIVETA, E. **Um modelo de suporte a programação orientada a aspectos**. Dissertação (Mestrado) - Universidade Federal de Santa Catarina, Florianópolis, 2001.

RESENDE, A. M. P. de; SILVA, C. C. da. **Programação Orientada a Aspectos em Java**. Rio de Janeiro: Brasport, 2005. 177 p.

SPRING AOP - Aspect Oriented Programming with Spring. Disponível em <http://www.springframework.org/docs/reference/aop.html>. Acesso em 10 ago. 2007.

SUN MICROSYSTEMS - Java Language Specification. Disponível em <http://java.sun.com/javase/index.jsp>. Acesso em: 25 abr. 2007.

WINCK, D. V.; GOETTEN JR., V. **AspectJ**: Programação Orientada a Aspectos com Java. São Paulo: Novatec, 2006. 229 p.