Programação Orientada a Aspectos: Um Estudo de Caso em uma Multinacional

Alexsandro Souza Filippetto
Universidade Luterana do Brasil, Curso de Ciência da Computação,
Gravataí-RS, Brasil, 94170-240
alexsandrosf@brturbo.com.br

e

Daniel Antonio Callegari
Pontifícia Universidade Católica do Rio Grande do Sul, Faculdade de Informática,
Porto Alegre-RS, Brasil, 90619-900
danielc@inf.pucrs.br

Abstract

This paper aims to provide a basement on the use of Aspect Oriented Programming (AOP) as well as reporting a case study in a multi site company. As we know, there is a persistent concern in companies when dealing with cost reduction and quality issues on software development. Normally a well-designed system is able to achieve good results from code reuse, greater simplicity and, by effect, better results on maintainability. The need of having modularized systems has made several object-oriented approaches insufficient to solve the separation of concerns in a suitable level. In this context, Aspect Oriented Programming is a new approach on software development. It proposes simplification in the development of some features that were once spread over the source code of the application. Here we present some encouraging results of the paradigm, in which we achieved near 90% of reduction in code interlacing.

Keywords: Software Engineering, Programming Languages, Aspect Oriented Programming.

Resumo

Este artigo apresenta uma revisão dos conceitos fundamentais de POA juntamente com os resultados obtidos de sua aplicação em um projeto interno de uma multinacional. Existe uma preocupação constante nas empresas quanto à redução de custos e a qualidade no desenvolvimento de software. Normalmente, um sistema bem projetado consegue obter bons resultados quanto à reutilização de código, além de maior simplicidade no desenvolvimento e, por conseqüência, melhores resultados também em sua manutenção. A necessidade de se obter sistemas mais modularizados tornou a abordagem orientada a objetos insuficiente para resolver a separação dos interesses do sistema em um nível adequado. Uma nova abordagem no desenvolvimento de software é a Programação Orientada a Aspectos, que propõe a simplificação no desenvolvimento de algumas funcionalidades que necessitavam ficar espalhadas em diversos pontos da aplicação. Como um dos resultados obteve-se uma redução média de 90% no entrelaçamento de código, o que reduz as possibilidades de erros e torna a manutenção muito mais eficiente.

Palavras-chave: Engenharia de Software, Linguagens de Programação, Programação Orientada a Aspectos.

1. INTRODUÇÃO

A necessidade das empresas de desenvolver *software* com qualidade motivou o uso da modelagem orientada a objetos para alcançar um maior nível de reutilização e manutenibilidade do código, aumentando a produtividade do desenvolvimento e levando a uma maior facilidade para adição de novos requisitos ao sistema [11]. Com efeito, a Orientação a Objetos se tornou um dos paradigmas mais utilizados na atualidade e consolidou-se como um modelo para o desenvolvimento de *software* [4]. Sabe-se, entretanto, que algumas de suas abstrações possuem limitações para separar e compor determinadas propriedades relevantes em um sistema de *software*: muitas propriedades importantes espalham-se por vários módulos e misturam-se com outras propriedades de um sistema, dificultando a reutilização e a manutenção de seus componentes.

Como uma proposta para tratar destas questões, apresentou-se o conceito de Programação Orientada a Aspectos (POA). Apesar de ter sido criada em 1997, somente mais recentemente esta metodologia tem despontado como uma forma de solucionar alguns problemas no desenvolvimento de *software*. Ao se fazer uso do conceito da POA, pode-se decompor mais facilmente algumas funcionalidades que até então ficavam distribuídas em diversas partes do código. Conforme [7], o objetivo deste novo paradigma é oferecer novas abstrações e mecanismos de composição para contornar algumas limitações no desenvolvimento de *software* orientado a objetos. A Programação Orientada a Aspectos provê suporte à separação e à composição dos chamados "interesses ortogonais" de uma aplicação através do uso de "aspectos".

Com o objetivo de avaliar o grau de redução do entrelaçamento de código proporcionado pela aplicação de POA, este artigo apresenta os resultados da aplicação desse novo paradigma por meio de um estudo de caso conduzido em uma empresa multinacional, com sede nos EUA e operações em diversos países, incluindo o Brasil – não citaremos a empresa por motivos de confidencialidade. Inicialmente será conduzida uma breve apresentação dos principais conceitos envolvidos nesse paradigma (seção 2), seguida de uma revisão da linguagem AspectJ (seção 3). A seção 4 apresenta o detalhamento do estudo de caso realizado, e seus resultados são comentados na seção 5. Os comentários finais e a conclusão encontram-se na seção 6.

2. Programação Orientada a Aspectos

Durante o desenvolvimento de um *software*, normalmente são encontrados problemas para os quais não se consegue implementar certas decisões de projeto de forma clara com a utilização de linguagens procedurais, nem com as orientadas a objetos. Dessa forma, torna-se necessário que a implementação dessas decisões de projeto fique "espalhada" ao longo do código, gerando um produto mais complexo e mais difícil de desenvolver e manter. Essas funcionalidades são difíceis de isolar porque "atravessam" as funcionalidades básicas do sistema, ou seja, não é possível encapsulá-las em um componente ou em uma classe, pois estas são introduzidas em diversos pontos isolados em todo o sistema. Tais funcionalidades são chamadas de "aspectos" de uma aplicação [7].

Com o objetivo de facilitar o desenvolvimento dos interesses específicos de uma aplicação, um grupo liderado por Gregor Kickzales, no Centro de Pesquisa da Xerox, em Palo Alto, lançou as origens da Programação Orientada a Aspectos [7]. O principal objetivo dessa abordagem é prover uma separação clara entre componentes e aspectos e aumentar a modularidade do sistema, facilitando então a compreensão, desenvolvimento e a manutenção do projeto [8]. Dessa forma, o objetivo desse novo paradigma é oferecer novas abstrações e mecanismos de composição para contornar algumas limitações atualmente enfrentadas no desenvolvimento de *software*.

É importante observar, contudo, que esse novo paradigma não tem o objetivo de substituir a Programação Orientada a Objetos, mas complementá-la, fornecendo um outro tipo de modularização, a qual leva para dentro de uma unidade mais simples – o aspecto – a implementação de uma responsabilidade que anteriormente necessitava ficar distribuída pelo código. Por esse motivo, os interesses tornam-se mais simples de tratar, já que ficam concentrados no código dos aspectos, e não mais espalhados pelas classes do sistema.

Do ponto de vista de implementação, diversas propostas foram apresentadas, como Eos [9], Hiper/J [3] e AspectJ [6]. Este estudo, no entanto, foi realizado com o suporte da linguagem AspectJ, a qual tem obtido grande aceitação pelo volume de pesquisas relacionadas.

3. AspectJ

AspectJ é uma extensão da linguagem Java para a implementação de aspectos, que provê a implementação em Java dos chamados "interesses ortogonais" [6], sendo seu compilador de código livre. Todo programa válido desenvolvido em Java é também válido em AspectJ, porque este último é baseado em Java. O compilador do AspecJ produz classes conforme a especificação Java *byte-code* [5], permitindo, desta maneira, que a Java Virtual Machine (JVM) execute qualquer classe gerada pelo AspectJ. Isso é importante porque permite que servidores que já executam código Java possam também executar aspectos sem nenhuma modificação em sua configuração.

Com a utilização da Programação Orientada a Aspectos no desenvolvimento de *software*, é necessário definir inicialmente quais são os interesses comuns (e utilizar a modelagem tradicional da Orientação a Objetos, por meio de classes), e quais são os interesses ortogonais (a serem desenvolvidos com a nova abstração de aspectos).

Com o AspectJ podem ser definidos pontos adicionais para a execução do programa. Para isso, a linguagem

define um conjunto básico de instruções que a ela foram agregadas para sua construção [6]:

- ***Join Points***: representam pontos bem definidos na execução de um programa onde um determinado aspecto pode ser aplicado;
- ***Point Cuts***: é um conjunto de especificação para os *Join Points*, baseando-se em um critério predefinido;
- ***Advices***: são trechos de códigos que são executados sobre os *Point Cuts*. Um *Advice* contém o código com as alterações que devem ser aplicadas "ortogonalmente" no sistema;
- ***Aspects***: são as unidades de implementação. De forma similar às classes, possuem um tipo e podem ter atributos e métodos.

Os métodos dos aspectos podem ser marcados como "before" (antes) ou "after" (após), que indicam o momento em que o aspecto "intercepta" os métodos das classes. A interceptação se dá pelo casamento da definição com a nomenclatura das classes e métodos. Uma aplicação desenvolvida em AspectJ deve ser compilada através de um processo chamado de "Weaving", o qual introduz o código necessário para relacionar os aspectos definidos com as classes comuns do modelo orientado a objetos [8]. Dessa forma, o código gerado é 100% compatível com o ambiente Java e possui a vantagem de usar aspectos para reduzir o número de linhas de código e, conseqüentemente, sua complexidade.

Após essa breve revisão dos conceitos básicos da Programação Orientada a Aspectos, apresenta-se, na próxima seção, o detalhamento do estudo de caso, iniciando pela descrição do sistema.

4. O Estudo de Caso

Para o estudo de caso realizado nesta pesquisa, buscou-se na empresa uma aplicação já existente, desenvolvida em Java, que apresentasse diversos comportamentos com as características de programação citadas anteriormente. O sistema que, então, serviu de base para este estudo tem como principal objetivo controlar e manter todos os contratos para venda de espaços de marketing em feiras, eventos e demais veículos de comunicação para empresas parceiras da instituição em questão.

Após a escolha do sistema foi realizado um *refactoring* na aplicação com a utilização da POA. Reengenharia de *software* (ou *refactoring*), segundo [2], é o exame, estudo, captura e modificação de funcionalidade de um sistema ou produto, visando reconstituí-lo de uma nova forma e com novas características, mas sem grandes alterações na funcionalidade e propósito inerentes ao sistema. Este artigo apresenta especificamente os resultados referentes à segunda etapa do projeto, na qual foi realizado um *refactoring* de suas funcionalidades.

4.1 Os Problemas Abordados

Os itens listados a seguir dizem respeito às funcionalidades identificadas no sistema para a proposta de utilização da Programação Orientada a Aspectos. Foram três os pontos contemplados por este estudo de caso: o tratamento de exceções, o controle de *Log* e o *Trace* da aplicação. As sub-seções seguintes descrevem cada um destes pontos e como eles foram anteriormente implementados no sistema em Java (etapa 1).

4.1.1. Tratamento de Exceções

Exceções são situações excepcionais que podem ocorrer durante a execução de um programa. Em Java, as exceções podem ser tratadas incluindo-se código adequado ao programa. Conforme [1], a manipulação e o tratamento de exceções em componentes de *software* podem consumir uma grande quantidade de recursos durante o desenvolvimento. Exatamente por isso, as exceções devem ser consideradas ao longo de todo o ciclo do desenvolvimento do *software.* Nesse sistema, as exceções são capturadas e persistidas em um banco de dados. Assim, na solução original foi criada uma classe para a persistência de tais exceções. Todos os métodos que necessitavam fazer este registro faziam referência a esta classe de forma manual e explícita.

4.1.2. Controle de Log

O controle de *log* normalmente é utilizado para armazenar um histórico com informações sobre o usuário, horário e informações pertinentes ao controle exigido. Essas informações podem ser armazenas em banco de dados, arquivos ou tabelas em geral. Para o sistema do estudo de caso, foi necessária a geração de *log* para toda transação que persistisse informações em banco de dados. Este *log* é armazenado em uma tabela do próprio banco de dados. Novamente, esta funcionalidade era ativada manual e explicitamente em cada caso.

4.1.3. Trace de Aplicação

Rastrear uma aplicação ou executar um *Trace* é uma técnica utilizada para se descobrir por quais pontos do programa a aplicação está passando ao executar determinada função. Durante a fase de desenvolvimento do *software*, a utilização dessa técnica torna-se importante principalmente em projetos legados para se descobrir todos os pontos que a aplicação está executando para determinada funcionalidade ou, até mesmo, quais valores que os métodos estão recebendo ou devolvendo, facilitando assim a depuração de erros. Com a utilização de um aspecto para a funcionalidade de *trace*, tais controles podem ser removidos sem que seja necessário alterar cada ponto isolado da aplicação, garantindo-se, assim, que nenhum código desnecessário permaneça no código final.

4.2. Arquitetura do Sistema

O sistema foi desenvolvido inteiramente em arquitetura *Web*. Para se ter uma noção da dimensão das alterações incorridas, apresenta-se seus sete módulos principais:

- **Administrador**: neste módulo estão agrupadas as funções responsáveis por manter as tabelas secundárias do sistema e documentos utilizados;
- **Programas**: este módulo mantém o cadastro de todos os programas de marketing abertos pela empresa;
- **Propostas e Contratos**: módulo responsável por disponibilizar as criações das propostas e planos de marketing até que estas propostas venham a se tornar contratos, ou até que estes programas sejam cancelados;
- **Status do Programa**: após aprovados os programas, devem ser atribuídos os responsáveis por manter este programa e algumas informações adicionais para continuação da execução destes programas;
- **Despesas**: este módulo mantém um controle de todas as despesas gastas com cada programa, como compra de materiais, suprimentos necessários ou qualquer despesa necessária para manutenção de um programa;
- **Vendedores**: módulo que mantém o cadastro das empresas parceiras que poderão vir a se tornar clientes da área de marketing;
- **Diversos**: este módulo contém uma série de relatórios financeiros, gerenciais e relatórios de uso comum da aplicação.

Por questões de espaço, este artigo apresenta apenas o desenvolvimento do módulo de manutenção de Programas (embora para a obtenção dos resultados foram utilizados dados de toda a aplicação).

Para uma melhor divisão e entendimento da aplicação, optou-se por separar as classes Java e aspectos em pacotes (*packages*) de acordo com as suas funcionalidades. Esses pacotes são descritos a seguir:

- **aspect:** pacote que armazena os aspectos implementados;
- **client:** classes responsáveis por fazer a interface com a camada JSP;
- **control:** nas classes desta camada, devem ser aplicadas as regras de negócio referentes a cada transação;
- **database:** armazena as classes responsáveis por fazer a persistência em banco de dados. Todo acesso ao BD é realizado a partir desta camada;
- **entity:** classes referentes às entidades do sistema (estas classes armazenam o conhecimento sobre estas entidades);
- **utility:** contém as classes de uso genérico para a aplicação, as quais possuem métodos compartilhados por diversas classes do sistema.

A seguir serão descritas em detalhes a modelagem (através da representação em UML para o desenvolvimento do módulo de Programas) e também os aspectos implementados (etapa 2) para atender aos requisitos da aplicação.

4.3. Detalhamento do Módulo de Manutenção de Programas

Este módulo tem como objetivo manter um cadastro de todos os programas de marketing criados pela instituição e também gerar consultas pertinentes a estes programas.

4.3.1. Casos de Uso

A Figura 1 apresenta os Casos de Uso que foram desenvolvidos para o módulo de Programas.

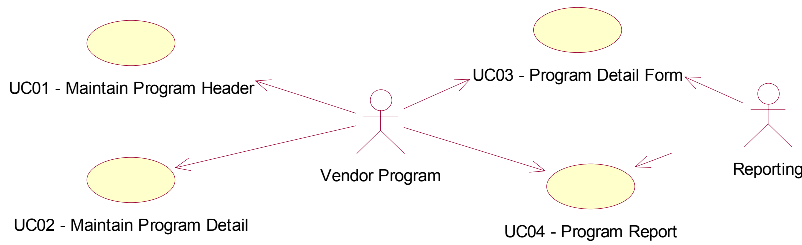

Figura 1. Casos de Uso Manter Programas

O caso de uso UC01 tem como objetivo manter o cadastro (inclusão, alteração e exclusão) dos programas. Apesar de existir apenas uma entidade em banco de dados referente ao programa, na aplicação seu cadastro foi dividido em duas partes; assim, o UC01 mantém o cadastro do "cabeçalho" dos programas, ou seja, apenas algumas informações mais importantes. O objetivo do UC02 é manter um cadastro das informações para detalhamento dos programas. O UC03 e o UC04 não serão detalhados neste trabalho por se tratarem de relatórios e não interagirem com os aspectos propostos.

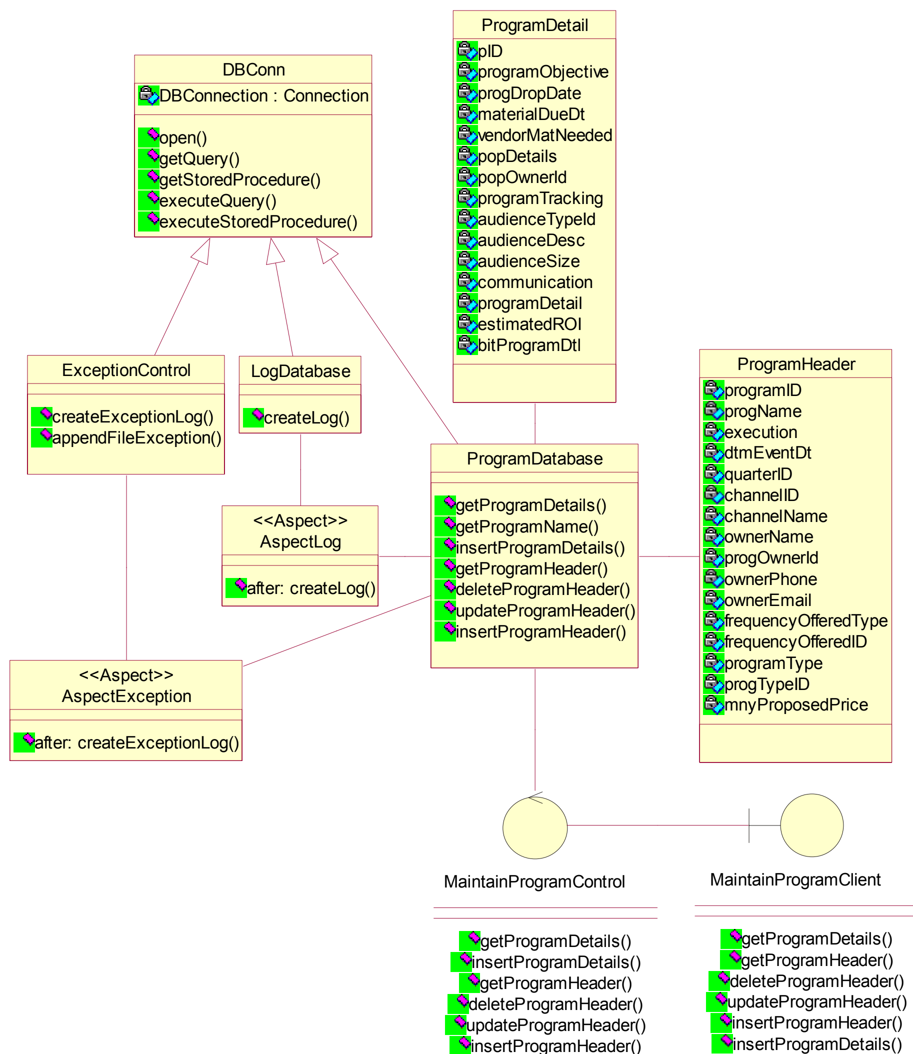

Figura 2. Diagrama de Classes Estendido do Módulo Programas

4.3.2. Diagrama de Classes

O diagrama de classes apresentado na Figura 2 representa uma abstração da realidade proposta para o desenvolvimento dos casos de uso UC01 e UC02, e não uma representação da aplicação como um todo.

Como se pode observar, na modelagem dos aspectos foi utilizado um estereótipo “<<Aspect>>” para sua representação, já que a ferramenta utilizada (Rational Rose [10]) não implementa o conceito de aspectos. Na modelagem proposta para as classes que representam os aspectos, foram acrescidos aos métodos os prefixos “after” ou “before”, indicando em que momento será executado o código referente ao aspecto. Embora não completamente apresentado neste artigo, nos diagramas de seqüência apresentados na próxima seção é possível visualizar a iteração dos aspectos com as classes.

4.3.3. Diagramas de Seqüência

Para representação dos diagramas de seqüência, as funcionalidades foram divididas em dois diagramas. O primeiro mostra a interação do usuário com a camada de interface da aplicação, enquanto que o segundo apresenta a seqüência de ações a partir da camada de negócios até a persistência em banco de dados. Neste exemplo foi utilizada como modelo a funcionalidade para criação do detalhamento de programas referente ao caso de uso UC02. A Figura 3 apresenta o diagrama de seqüência da camada de interface para editar ou incluir os dados de detalhamento de um programa.

A Figura 4 apresenta a seqüência de interação das camadas de negócio e de dados a partir da classe de fronteira (MaintainProgramClient). No diagrama foram incluídos os aspectos e como eles interagem com as demais classes do sistema.

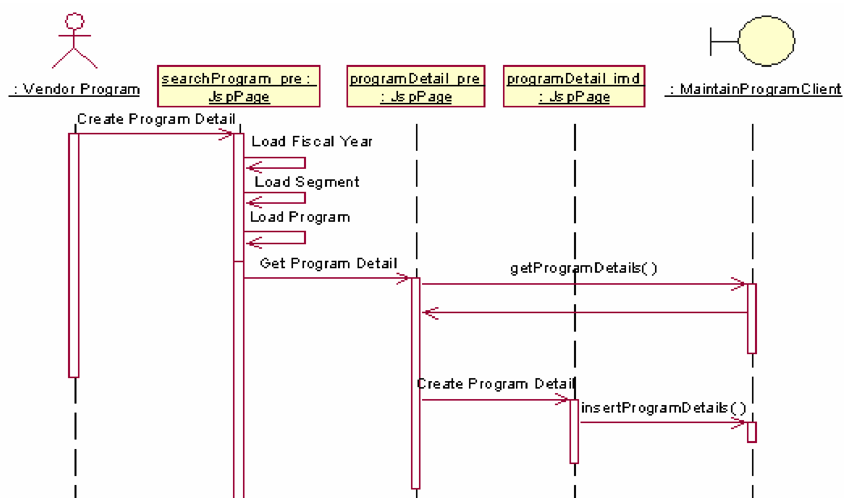

Figura 3. Diagrama de Seqüência da Camada de Interface

A geração de *logs* através de aspectos pode ser vista na chamada do método “createLog” do aspecto “AspectLog”, que está interagindo com a classe “ProgramDatabase”. Nas assinaturas dos métodos dos aspectos foi adicionado um identificador “after” para indicar o momento em que o aspecto intercepta os métodos das classes.

4.4. Detalhamento dos Aspectos Implementados

Esta seção apresenta o detalhamento dos aspectos implementados para atender alguns problemas do desenvolvimento de *software* abordados neste artigo. O objetivo é exemplificar, através do código implementado, a forma com que os aspectos interagem com as classes Java e como se pode obter melhores resultados quanto ao encapsulamento de código no desenvolvimento de *software* através da utilização de aspectos.

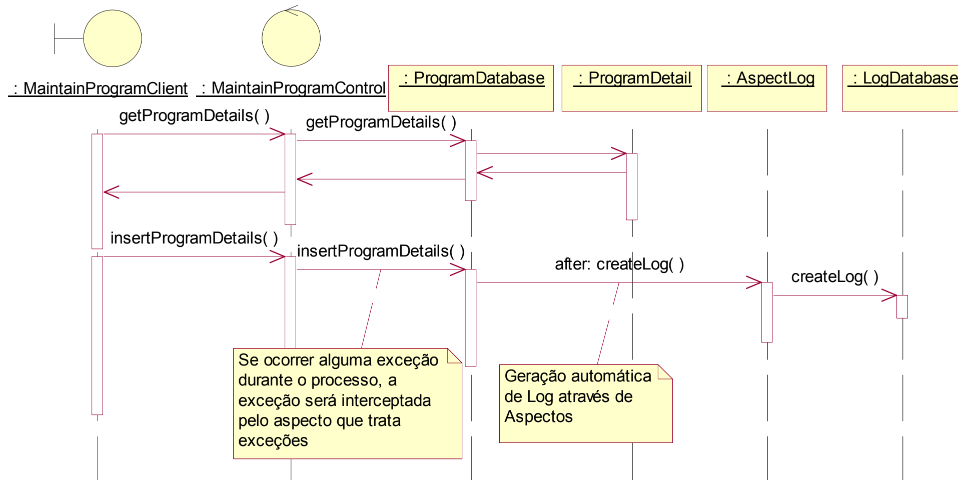

Figura 4. Diagrama de Seqüência da Camada de Regras de Negócio

4.4.1. AspectLog

Para o sistema deste estudo de caso, é necessário que seja gerado *log* apenas para transações que persistem informações no banco de dados. Desta forma, o aspecto de *log* deve interagir apenas nas classes do pacote "database". A modelagem define que, para todo método para o qual se deseja armazenar *log*, sua assinatura deve iniciar por "update", "insert" ou "delete". Esta definição se dá pelo fato de os aspectos interagirem com as classes através das assinaturas dos métodos. Para os métodos que atualizam informações em banco de dados, mas nos quais não é necessário gerar *log* para transação, o método não deve iniciar por nenhuma das três palavras descritas para que o aspecto não intercepte a ação do método. O fato de os aspectos interagirem com as classes através das assinaturas dos métodos exige (para efeito de obtenção de aspectos genéricos) uma padronização na sua nomenclatura. A Figura 5 exemplifica o desenvolvimento para o aspecto referente à funcionalidade de *log*.

```
package project.aspect;
import project.database.*;
public aspect AspectLog {
    pointcut createLog(Object obj, String pUser):
         (call (public * project.database.*.update*(.., String)) ||
          call (public * project.database.*.insert*(.., String)) ||
          call (public * project.database.*.delete*(.., String))) &&
          args(obj, pUser);
    after(Object obj, String pUser) throws Exception: createLog(obj, pUser) {
        try {
            LogDatabase log = new LogDatabase();
             log.createLog(thisJoinPoint.getSignature().getDeclaringType().toString(),
                           thisJoinPoint.getSignature().getName(), pUser);
        } catch (Exception e) { }
    }
}
```

Figura 5. Código referente ao Aspecto de *Log*

Ao analisar a definição do *Pointcut* "createLog", nota-se que o aspecto será executado após a chamada de qualquer método das classes que se encontram no pacote "project.database", que iniciem por "update", "insert" ou "delete" e que contenham dois atributos: o primeiro, de qualquer tipo, e o segundo, obrigatoriamente uma String. O código do aspecto apresentado em sua execução cria uma instância da classe "LogDatabase", a qual persiste as informações de *log* necessárias no banco de dados. O código desta classe não é apresentado por não ser relevante ao conteúdo deste trabalho.

4.4.2. AspectTrace

Apesar de não fazer parte dos requisitos do sistema, a funcionalidade de *trace* é importante, pois ajuda na depuração do código durante a fase de desenvolvimento. Esta técnica para rastrear a aplicação pode ser utilizada durante desenvolvimento orientado a objetos; porém, como é necessário incluir este código nos pontos onde se quer depurar, corre-se o risco de deixar algum código desnecessário quando a aplicação é

posta em produção. Por ser uma funcionalidade que se espalha por muitos pontos da aplicação e por ficar a critério dos desenvolvedores a sua utilização no código e a posterior retirada para implantação do sistema em produção, optou-se pela criação de um aspecto para atender a este requisito, uma vez que, após a fase de desenvolvimento e testes, bastaria retirar o aspecto para a compilação do projeto, sem necessitar alterar o código implementado. A Figura 6 apresenta o código para o aspecto referente à funcionalidade de *trace*.

```
package project.aspect;
public aspect AspectTrace {
    pointcut trace(): execution (* *(..));
    before() : trace() {
        System.out.println("--Início Método: " + thisJoinPoint.getSignature());
    }
    after() : trace() {
        System.out.println("--Final Método: " + thisJoinPoint.getSignature());
    }
}
```

Figura 6. Código referente ao Aspecto de *Trace*

O código referente ao aspecto é executado para qualquer classe contida no projeto. Se for necessário restringir o escopo de aplicação do *trace*, basta adicionar no *Pointcut* parte da assinatura dos métodos aos quais se pretende aplicar o *trace*.

4.4.3. AspectException

A linguagem Java permite que as exceções ocorridas durante a execução de um sistema possam ser tratadas adequadamente sem que o sistema termine sua execução de forma inesperada. Além deste tratamento, para este caso é necessário que essas exceções sejam armazenadas para posterior avaliação e detecção de erros. A Figura 7 mostra o código implementado para o aspecto de tratamento de exceções.

```
package project.aspect;
import project.control.*;
public aspect AspectException {
pointcut logException(): (execution (public * project.database.*.*(..)) ||
 execution (public * project.database.*.*())) &&
!execution (public * project.database.DBConn.*(..));
    after() throwing (Throwable ex) : logException() {
        try {
            ExceptionControl exc = new ExceptionControl();
            exc.createException(ex.getMessage(),
                thisJoinPoint.getSignature().getDeclaringType().toString(),
                thisJoinPoint.getSignature().getName());
        } catch (Exception e) {}
    }
}
```

Figura 7. Código referente ao Aspecto para Tratamento de Exceções

A classe ExceptionControl é responsável por gravar em banco de dados as informações referentes à exceção ocorrida. Quando ocorrer algum problema na gravação destas informações em banco de dados, esses dados são gravados em um arquivo com extensão “txt”, evitando que alguma informação se perca durante a execução do sistema.

A próxima seção apresenta uma avaliação da utilização da Programação Orientada a Aspectos, através de resultados extraídos deste estudo de caso. O objetivo é abordar as questões quantitativas, como o número de classes onde se espalhava determinada funcionalidade, entre outros aspectos relevantes.

5. Resultados Obtidos

5.1. Propriedades do Sistema

Para um melhor entendimento dos resultados com a utilização da POA, é necessário apresentar alguns dados referentes à aplicação primeiramente desenvolvida através da Orientação a Objetos. A Tabela 1 apresenta alguns dados referentes ao sistema antes da aplicação do *refactoring* por meio da POA.

Tabela 1. Propriedades do Sistema

Item	*Quant.*
Total de Classes do Sistema	37
Pacotes	5
Linhas de Código	3925
Total de Métodos	256
Total de Classes em que se estende a Funcionalidade de *Log*	9
Total de Métodos em que se estende a Funcionalidade de *Log*	65
Total de Classes em que se estende o Tratamento de Exceções	10
Total de Métodos em que se estende o Tratamento de Exceções	66

5.2. Resultados após a Aplicação do Refactoring

5.2.1. Controle de Log

Os resultados após a conversão de classes para aspecto para a funcionalidade de *log* podem ser vistos na Tabela 2. A primeira coluna representa o item avaliado; a segunda apresenta os indicadores para o desenvolvimento através da OO e, por fim, a terceira coluna mostra os resultados obtidos através da utilização da POA.

Tabela 2. Resultados para o *Refactoring* de *Log*

Item	*OO*	*OA*	*Diferença*
Pacotes	5	6	**+1**
Total de Classes do Sistema	37	37	**=**
Total de Aspectos	-	1	**+1**
Pointcuts	-	1	**+1**
Linhas de Código	3925	3882	**-43**
Total de Métodos	256	256	**=**
Total de Classes em que se estende a Funcionalidade de *Log*	10	1	**-9**
Total de Métodos em que se estende a Funcionalidade de *Log*	65	0	**-65**

Através dos dados apresentados na Tabela 2, é possível verificar uma diminuição do número de linhas de código pela criação do aspecto do controle de *log*. Esta diminuição pode não ocorrer em todas as aplicações de aspectos, pois é necessário incluir um novo arquivo-fonte (aspecto) com um determinado código, o qual pode ser mais extenso que aquele implementado através da OO. A vantagem da utilização desse novo conceito se mostra no número de classes nas quais se estendia a funcionalidade.

5.2.2. Tratamento de Exceções

No caso do tratamento de exceções, foi possível retirar todo o código que ficava disperso nas classes e mantê-lo apenas na classe que efetua a gravação em banco de dados. A Tabela 3 apresenta estes dados.

Tabela 3. Resultados para o *Refactoring* de Exceções

Item	*OO*	*OA*	*Diferença*
Pacotes	5	6	**+1**
Total de Classes do Sistema	37	37	**=**
Total de Aspectos	-	1	**+1**
Pointcuts	-	1	**+1**
Linhas de Código	3925	3813	**-112**
Total de Métodos	256	257	**+1**
Total de Classes em que se estende o Tratamento de Exceções	11	1	**-10**
Total de Métodos em que se estende o Tratamento de Exceções	66	0	**-66**

O uso de aspectos para este requisito se mostrou bastante eficaz, permitindo que em futuras manutenções do tratamento de exceções não seja necessário alterar as suas chamadas em várias classes – apenas será preciso alterar pontos específicos no aspecto, o que, por conseguinte, torna mais simples a evolução desta funcionalidade.

5.2.3. Aplicação de Trace

A Tabela 4 apresenta os resultados considerando-se a aplicação de *trace* a todos os métodos do sistema. Assim, foram acrescidas ao total de linhas codificadas da aplicação pelo menos duas linhas por método: uma

para aplicação do *trace* no início do método e uma ao final de cada método.

Tabela 4. Resultados para a Aplicação de *Trace*

Item	*OO*	*OA*	*Diferença*
Pacotes	5	6	**+1**
Total de Classes do Sistema	37	37	**=**
Total de Aspectos	-	1	**+1**
Pointcuts	-	2	**+2**
Linhas de Código	4437	3941	**-496**
Total de Métodos	256	256	**=**
Total de Classes em que se estende o *Trace* da Aplicação	37	0	**-37**
Total de Métodos em que se estende o *Trace* da Aplicação	256	0	**-256**

Salienta-se que os resultados da Tabela 4 não são considerados na avaliação final da aplicação por se tratar de uma funcionalidade de uso específico e pontual.

5.2.4. Comparação das Soluções POO e POA

A aplicação da POA em conjunto com a POO para desenvolvimento do sistema se mostrou uma boa alternativa para complementar alguns pontos que a POO não conseguia resolver de uma forma clara e modular. A Tabela 5 apresenta os dados comparativos entre o desenvolvimento utilizando a OO e após a aplicação da OA.

Tabela 5. Resultados após o *Refactoring*

Item	*OO*	*OA*	*Diferença*
Pacotes	5	6	**+1**
Total de Classes do Sistema	37	37	**=**
Total de Aspectos	-	3	**+3**
Pointcuts	-	4	**+4**
Linhas de Código	3925	3836	**-89**
Total de Métodos	256	257	**+1**

Visualiza-se, através da Tabela 5, uma diminuição no número de linhas de código após a aplicação da POA – embora não tão representativa, pois que novos arquivos (aspectos) foram criados. Porém, as vantagens na utilização da POA não estão representadas apenas pela diminuição do número de linhas do código, mas, sim, na diminuição do entrelaçamento do código que se conseguiu obter para as funcionalidades propostas.

Reconhece-se que o ganho de tempo em futuras manutenções só poderá ser medido quando houver manutenções nestas funcionalidades. Entretanto, como foi possível encapsular estas funções, futuras manutenções serão mais pontuais. Evita-se, com isso, modificar diversos pontos da aplicação, o que diminui o percentual de testes de regressão que deverão ser aplicados para garantir a consistência da manutenção e, por conseqüência, reduzir-se-á o custo do projeto como um todo.

Analisando-se os dados referentes às classes nas quais se estendiam as funcionalidades em que foram aplicados conceitos de POA, alcançou-se uma redução de aproximadamente 90% no entrelaçamento dessas funcionalidades: houve uma redução na dispersão do código (de 10 classes para uma, na aplicação de *log*) e, no outro caso, de 11 classes para uma – caracterizando uma significativa diferença.

5.3. Pontos Positivos

A aplicação da POA para os interesses considerados ortogonais, como os apresentados neste artigo, mostrou-se satisfatória, pois os resultados mencionados indicam uma melhora no encapsulamento do código e um aumento na possibilidade de reuso do código já implementado. Como também se pôde ver, uma das grandes vantagens ocorre na redução do esforço necessário na fase de testes.

Com o crescimento dos estudos referentes a este novo paradigma, novas linguagens que suportam a POA estão surgindo. Foi disponibilizada também a implementação de aspectos para a arquitetura .Net da Microsoft [9]. Assim, cresce a possibilidade de utilização desse paradigma e, por conseqüência, aumenta a maturidade quanto aos conceitos e utilização de aspectos em conjunto com a OO.

5.4. Pontos Negativos

Um dos aspectos negativos da aplicação desse novo paradigma, identificado durante o desenvolvimento do sistema, diz respeito aos projetos legados. Muitos projetos não seguem um padrão bem definido para seu

desenvolvimento, gerando uma série de classes e métodos que não adotam uma mesma linha de padronização. Para esse tipo de projeto, a aplicação da POA pode se tornar um tanto complexa, já que a criação dos aspectos ocorre sobre a nomenclatura das classes, pacotes e métodos. Se esses artefatos não seguiram um mesmo padrão de nomenclatura, a criação de aspectos que atendam toda a aplicação se torna mais difícil.

Um outro ponto importante que deve ser avaliado ocorre durante a fase de projeto. Para essa fase não foram encontradas ferramentas de modelagem que contemplem a Orientação a Aspectos, tendo sido necessária a criação de estereótipos para diferenciar as classes de aspectos, por exemplo.

Os aspectos qualitativos são difíceis de se medir em apenas um projeto. Seria necessária uma avaliação durante um maior período de tempo e essa análise deveria ser realizada em um maior número de projetos para que fosse possível a obtenção de indicadores mais realistas sobre o desenvolvimento deste novo paradigma.

6. Conclusão

Este artigo apresentou uma avaliação dos resultados do uso do paradigma da Programação Orientada a Aspectos em conjunto com a Orientação a Objetos em um estudo de caso. A partir dos resultados observados, pode-se verificar que esse paradigma demonstrou-se eficaz na diminuição do entrelaçamento do código, uma vez que os objetivos iniciais referentes ao encapsulamento das funcionalidades propostas foram alcançados.

A utilização da POA, em conjunto com a OO, apresentou-se como uma valiosa alternativa para a resolução de pontos que a OO não resolvia de uma forma modular. Com a aplicação de *refactoring* através da POA, espera-se que as manutenções sobre essas funcionalidades ocorram em pontos mais específicos do sistema. Seria interessante fazer um estudo sobre essas futuras manutenções para se avaliar questões como o tempo de desenvolvimento e a qualidade resultante após a o uso da POA, ficando a sugestão para trabalhos futuros.

É importante salientar que não se deve tomar como regra a utilização de aspectos em todos os projetos. Em certos casos, isso poderá gerar apenas mais uma camada no desenvolvimento de *software*, o que não é o objetivo desse paradigma. Desta forma, é importante realizar uma avaliação das funções da aplicação para averiguar a necessidade ou não de se utilizar esse paradigma.

Referências

[1] ALVARO, A.; LUCRÉDIO, D.; ALMEIDA, E, S.; PRADO, A, F.; TREVELIN, L, C. Um Framework de Componentes para Aspectos não Funcionais. In: 3o Workshop de Desenv. Baseado em Componentes, 2003, São Carlos - Brazil.

[2] FOWLER, Martin; BECK, Kent; BRANT, John; OPDYKE, William; ROBERTS, Don. Refactoring: Improving the Design of Existing Code. 2. ed. Massachusetts, EUA: Ed. Addison-Wesley, 1999. 362p.

[3] HUGUNIN, Jim. The Next Steps For Aspect-Oriented Programming Languages. Vanderbilt Workshop. White Paper. Palo Alto, EUA. 2001. 5 f.

[4] JACOBSON, Ivar; CHRISTERSON, Magnus; JONSSON, Patrik; ÖVERGAARD, Gunnar. Object-Oriented Software Engineering. 9. ed. Greenwich, EUA: Ed. ACM Press, 1998. 528p.

[5] Java Technology. Disponível em: <http://www.java.sun.com>. Acesso em: 07 novembro 2004.

[6] KICZALES, Gregor; HILSDALE, Erik; HUGUNIN, Jim; KERSTEN, Mik; PALM, Jeffrey; GRISWOLD, William. An Overview of AspectJ. Lecture Notes in Computer Science. Vancouver, Canadá. 2001. 27 f.

[7] KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA, Chris; LOPES, Cristina Videira; LOINGTIER, Jean-Marc; IRWIN, John. Aspect-Oriented Programming. Proceedings of European Conference on Object-Oriented Programming. Palo Alto, EUA. 1997. 25 f.

[8] KISELEV, Ivan. Aspect-Oriented Programming with AspectJ. 1 ed. Indianápolis, EUA: Ed Sams, 2002. 274 p.

[9] Aspect Orienting .NET Component. Disponível em: <http://www.theserverside.net>. Acesso em: 02 junho 2005.

[10] Rational Rose. Disponível em: <http://www-306.ibm.com/software/rational/>. Acesso em: 05 junho 2005.

[11] SOARES, Sérgio; BORBA, Paulo. Desenvolvimento de Software Orientado a Aspectos Utilizando RUP e AspectJ. In: Tutorial do XVIII Simpósio Brasileiro de Engenharia de Software, 2004, Brasília-DF, 2004.