Reengenharia de Sistemas Orientados a Objetos para Sistemas Orientados a Aspectos

Ricardo Argenton Ramos[1]
Anderson Pazin[*]
Rosângela Ap. D. Penteado
UFSCar - Universidade Federal de São Carlos, Departamento de Computação.
São Carlos - SP, Brasil, CEP:13565-905, 55(0xx 16) 260-8233
{ rar, rosangel }@dc.ufscar.br

*Faculdades Salesiana de Lins, Centro de Tecnologia de Informação.
Lins - SP, Brasil, CEP: 16400-505, 55(0xx14) 3522 - 4733
anderson@salesianolins.br

Abstract

The source code of non-functional concerns spread and tangled with functional concerns in Objects Oriented systems, cause problems as the agreement difficulty, reuse and to add new functionalities to the system. To bright up these problems, appears the Aspect-Oriented Programming, having as main characteristic the structures supply *that better encapsulate the concerns. This work shows the accomplishment of a process of a reengineering* process *using the Aspecting approach, where the concern of Persistence is identified, modeled and implemented in a language that supporst the Aspect-Oriented Paradigm. Two systems are used as case study, one implements the Persistence Layer design patern, and the other does not use the design patern at the persistence implementation.*

Key Words: Reengineering, Aspect Oriented Programming, Persistence.

Resumo

O código fonte de interesses não funcionais espalhados e entrelaçados com interesses funcionais em sistemas Orientados a Objetos, causa problemas como a dificuldade de entendimento, de reuso e de adicionar novas funcionalidades ao sistema. Visando amenizar problemas como esses, surge a Programação Orientada a Aspectos, que tem como principal característica o fornecimento de estruturas que melhor encapsula os interesses. Este trabalho apresenta a realização de um processo de reengenharia utilizando a abordagem Aspecting, em que o interesse de Persistência é identificado, modelado e implementado em uma linguagem que da apoio ao paradigma Orientado a Aspectos. Dois sistemas são utilizados como estudo de caso, sendo que um implementa o padrão de projeto Camada de Persistência, e o outro não utiliza o padrão na implementação da persistência.

Palavras-Chave: Reengenharia, Programação Orientada a Aspectos, Persistência.

[1] Apoio Financeiro do CNPq

1. Introdução

Sistemas legados freqüentemente possuem alto custo de manutenção, lógica desestruturada e com isso aumentam as dificuldades para a interação com novas tecnologias e novos ambientes. Porém, possuem embutidos em seu código as regras de negócio e o conhecimento de vários anos de seus desenvolvedores/mantenedores que não podem ser simplesmente descartados. Desenvolver um novo sistema utilizando novas tecnologias e conceitos, além de consumir muito esforço tem-se o receio de que todas as regras de negócio, contidas somente no código fonte possam ser perdidas.

Um sistema de software é composto por requisitos funcionais e não funcionais. Os requisitos funcionais correspondem às condições ou à capacitação que devem ser contempladas pelo software. Geralmente, trata-se de necessidades do cliente e/ou do usuário para resolver um problema ou alcançar um objetivo [5]. Os requisitos não funcionais não expressam função (transformação) a ser implementada em um sistema, mas sim condições de comportamento e restrições que devem prevalecer, como por exemplo, tratamento de exceções, persistência de dados, segurança e de desempenho [4]. Os requisitos não funcionais na maioria das vezes em sistemas Orientados a Objetos são implementados de forma entrelaçada com o código que implementa os requisitos funcionais causando problemas como a dificuldade de entendimento do código e, conseqüentemente, de reuso, de manutenção e de adição de novas funcionalidades. Uma forma de amenizar essas dificuldades é a utilização da separação dos requisitos. Os requisitos não funcionais, de agora em diante, chamados de interesses podem ser isolados e implementados com linguagens de implementação com tal capacidade.

Sistemas implementados com o paradigma Orientado a Objeto utilizam técnicas de modularização de interesses bem sucedidas, porém essa abordagem de modularização, de acordo com um único interesse inerente, é insuficiente, por não prover todas as estruturas necessárias para o desenvolvimento de sistemas complexos [17]. Embora as pesquisas em engenharia de software estejam bastante amadurecidas, a manutenção de software permanece como um problema central à área. Outras técnicas como a utilização de padrões de projeto visam minimizar o problema da modularidade, porém segundo alguns autores [12] o código que implementa o padrão de projeto ainda fica entrelaçado e espalhado pelo código funcional da aplicação, não solucionando assim o problema da dificuldade de reusar, manter o sistema.

A reengenharia de sistemas Orientados a Objetos para Orientados a Aspectos (OA), pode ser usada para minimizar as dificuldades citadas. Pois a Programação Orientada a Aspectos possui mecanismos mais eficientes para a modularização de interesses com alternativas válidas para elaborar sistemas modularizados, facilitando futuras manutenções e evoluções [10]. Tendo como motivação os problemas apresentados, Ramos [15] propõe a abordagem *Aspecting* que visa fornecer um processo para que engenheiros de software conduzam a migração de sistemas Orientados a Objetos para sistemas Orientados a Aspectos.

O objetivo deste artigo é mostrar como o interesse de persistência em banco de dados relacional existente em sistemas Orientados a Objetos é modularizado e implementado utilizando a Programação Orientada a Aspectos. Para isso, dois estudos de caso são apresentados: Um sistema que utiliza o padrão de projeto Camada de Persistência [18] para implementar o interesse de persistência e outro sistema que não utiliza padrão. A realização da reengenharia é apoiada pela abordagem *Aspecting*. Ressalta-se que essa abordagem apóia a identificação e posterior modelagem e implementação de outros interesses aqui não especificados [15].

Na Seção 2 encontram-se os assuntos relacionados a separação de interesses e a programação orientada a aspectos; na Seção 3 a abordagem Aspecting é apresentada. Na Seção 4 a aplicação da abordagem Aspecting a dois sistemas exemplo é apresentada; e as considerações finais são apresentadas na Seção 5.

2. Separação de Interesses e a Programação Orientada a Aspectos

Czarnecki e Eisenecker [6] comentam que a necessidade de manipular um requisito importante de cada vez, durante o desenvolvimento de um sistema, é chamado de princípio da separação de interesses. Linguagens de programação geralmente fornecem construtores para organizar um sistema em unidades modulares que representam os interesses funcionais da aplicação. Essas unidades são expressas como objetos, módulos e procedimentos. Mas, também, há interesses em um sistema que abrangem mais de um componente funcional, tais como: sincronização, interação de componentes, persistência e controle de segurança. Esses interesses são geralmente implementados por fragmentos de código espalhados pelos componentes funcionais. Eles são chamados de aspectos (em nível de código) e de interesses (em nível de projeto) e alguns são dependentes de um domínio específico, enquanto outros são mais gerais.

Outras técnicas que procuram auxiliar a separação de interesses é a utilização de padrões de projeto, como os propostos por Gamma e outros [8]. Porém, segundo Noda e Kishi [12], as técnicas atuais de programação não são adequadas para a utilização dos padrões de projeto por tornarem a aplicação dependente deles, diminuindo as chances de reuso da parte funcional da aplicação.

Um exemplo de padrão de projeto é o padrão Camada de Persistência que permite minimizar a incompatibilidade existente entre o paradigma orientado a objetos e a persistência de dados em um banco relacional

[18]. Ele consiste em uma camada de persistência que cuida da interface entre objetos de aplicação e tabelas de banco de dados relacional. Possui um conjunto de dez sub-padrões que podem ser utilizados para a implementação dessa camada de persistência: Camada Persistente (*Persistent Layer*), *CRUD* (Criação (**C**reate), Leitura (**R**ead), Atualização (**U**pdate) e Remoção (**D**elete)), Descrição de Código SQL (*SQL Code Description*), Gerenciador de Tabelas (*Table Manager*), Métodos de Mapeamento de Atributos (*Atributte Mapping Methods*), Conversão de Tipos (*Type Conversion*), Gerenciador de Mudanças (*Change Manager*), Gerenciador de Identificadores Únicos (*OID Manager*), Gerenciador de Conexão (*Connection Manager*) e Gerenciador de Transação (*Transaction Manager*). Cada um deles possui uma funcionalidade bem definida e enquanto alguns são obrigatórios, outros são opcionais. Por exemplo, o padrão CRUD é essencial, para a realização das operações básicas de persistência, já o padrão Conversão de Tipos é útil, mas não essencial, podendo não ser utilizado.

O padrão Camada de Persistência possui três formas de uso, segundo Cagnin [1] a terceira forma é a que permite mais facilmente reutilizar o Gerenciador de Tabelas (*Table Manager*), pois não há preocupação com a persistência de qualquer tipo de objeto no banco de dados, necessitando apenas conhecer os parâmetros dos métodos das classes. Além disso, o sistema implementado utilizando-a é o mais manutenível. Embora a classe `TableManager` seja reusável, o mesmo não ocorre com as classes da aplicação, devido à dependência do padrão, que ocorre porque essas classes invocam métodos existentes nas classes que implementam o padrão.

Com a utilização de técnicas da programação orientada a aspectos, com a intenção de melhorar a separação de interesses de sistema implementados com padrões de projeto, Camargo e outros [2] implementam com aspectos o padrão de projeto Camada de Persistência [18]. Um dos benefícios obtido com essa implementação é o reuso de código do padrão devido a inversão das dependências. O código funcional do sistema não depende do código do padrão implementado nos aspectos, como ocorre na implementação orientada a objetos. Assim, há um impacto direto na localidade de código, pois todas as dependências entre os padrões e a aplicação são localizadas no código do padrão.

A Programação Orientada a Aspectos (POA) trata os interesses que entrecortam as classes de modo análogo ao que a programação orientada a objetos faz para o encapsulamento e a herança. Ou seja, provê mecanismos de linguagem que explicitamente capturam a estrutura de entrecorte, alcançando, assim, os benefícios de melhor modularidade, tais como: código mais simples, mais fácil de manter e alterar e, conseqüentemente, aumento da reusabilidade do código [10].

A programação orientada a aspectos consiste na separação dos interesses de um sistema em unidades modulares e posterior composição (*weaving*) desses módulos em um sistema completo [6]. Os interesses podem variar de noções de alto nível, como segurança e qualidade de serviço, a noções de baixo nível, como sincronização e manipulação de *buffers* de memória.

Uma linguagem bastante difundida para o apoio a POA de propósito geral é a AspectJ [11]. Seus componentes são representados por classes Java e uma nova construção, denominada aspecto (*aspect*), é responsável por implementar os interesses que entrecortam a estrutura das classes . A composição dos aspectos com as classes é executada em tempo de compilação, sobre *bytecode* ou código fonte Java.

Os seguintes conceitos são usados em AspectJ:

Pontos de junção (*joinpoints*): são métodos bem definidos na execução do fluxo do programa que compõem pontos de corte.

Pontos de corte (*pointcuts*): identificam coleções de pontos de junção no fluxo do programa.

Sugestões (*advices*): são construções semelhantes a métodos, que definem comportamentos adicionais aos pontos de junção. São executadas quando pontos de junção são alcançados [11]. AspectJ tem três tipos diferentes de sugestões:

i) **Pré-sugestão (*before*):** é executada quando um ponto de junção é alcançado e antes da computação ser realizada.

ii) **Pós-sugestão (*after*):** é executada quando um ponto de junção é alcançado e após a computação ser realizada.

iii) **Sugestão substitutiva (*around*):** é executada quando o ponto de junção é alcançado e tem o controle explícito da computação, podendo alternar a execução com o método alcançado pelo ponto de junção.

A modularização de interesses de entrecorte é realizada usando pontos de junção (*join points*) e sugestões (*advices*).

Uma vez executada a compilação, os entrecortes estarão permanentemente incorporados às classes, não sendo possível alterar a composição durante a execução do programa. O processo de composição segue duas etapas: i) o código Java e o código AspectJ são compilados em *bytecode* Java, instrumentado com informações sobre as construções do AspectJ (como sugestões e pontos de corte); ii) o compilador utiliza essas informações para gerar as classes compostas (*bytecode* compatível com a especificação Java) [9].

3. Abordagem *Aspecting*

A abordagem é composta por três etapas distintas: I Entender a Funcionalidade do Sistema; II Tratar o Interesse e III. Comparar Sistema OA com OO, que podem ou não ser apoiadas por ferramentas CASE ou

ferramentas especificas da POA. A Figura 1 mostra as etapas (representadas por retângulos) que compreendem a abordagem *Aspecting*, utilizando a notação SADT [16].

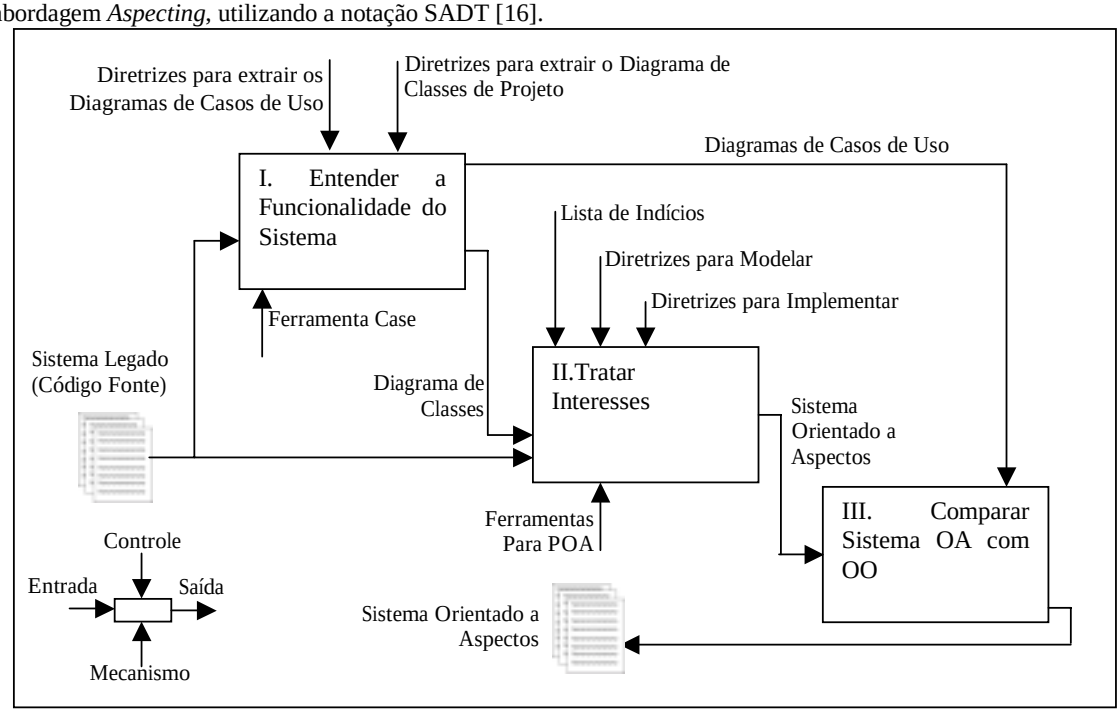

Figura 1 - Etapas da Abordagem *Aspecting*.

Cada etapa da abordagem contém diretrizes que auxiliam o engenheiro de software a conduzir a reengenharia. As etapas e os passos que compõem a abordagem são apresentados na Tabela 1.

Tabela 1 - Etapas e Passos da Abordagem *Aspecting*.

Etapas	Passos	Descrição
I. Entender a Funcionalidade do Sistema **Descrição:** Extrai as informações sobre a funcionalidade do sistema.	**I.1.** Gerar Diagramas de Casos de Uso. (caso não existam)	Executar o sistema para criar casos de uso correspondentes às funcionalidades nele existente. Utilizar as diretrizes criadas para esse fim.
	I.2. Gerar Diagrama de Classes de Projeto. (caso não exista)	Criar um diagrama de classe a partir do código fonte.Utilizar as diretrizes criadas para esse fim. **Opção**: Utilizar a Ferramenta *Omondo* [13].
II. Tratar Interesses **Descrição:** Esta etapa é evolutiva sendo os passos repetidos até que todos os interesses da Lista de Indícios sejam pesquisados ou até que o engenheiro de software decida finalizar o processo.	**II.1.** Marcar o Interesse. (caso exista)	Para um dos interesses da lista de indícios: Para cada classe implementada no código fonte, se existir tal indício Marcar o trecho do código fonte, adicionando comentários no final de cada linha, indicando o nome do interesse, seguido de um número seqüencial que deverá indicar a ordem em que esse trecho aparece na classe.
	II.2. Modelar o Interesse.	Adicionar ao diagrama de classe de projeto o aspecto. Utilizar as diretrizes específicas de cada interesse criadas para esse fim.
	II.3. Implementar o Interesse.	Implementar o aspecto que foi adicionado ao diagrama de classes no passo anterior. Utilizar as diretrizes específicas de cada interesse criadas para esse fim.Retornar ao passo 2.1.
III. Comparar Sistema Orientado a Aspectos com o Orientado a Objetos **Descrição**: Compara a funcionalidade do sistema OA que foi gerado com a do sistema OO original.	**III.1.** Comparar a Funcionalidade do Sistema Orientado a Aspectos	Utilizar os diagramas de casos de uso elaborados no passo 1.1. e suas descrições, e com o auxilio de diretrizes, criadas para esse fim, verificar se o sistema gerado atende às funcionalidades do sistema original representadas nos diagramas de casos de uso.

A identificação de interesses não funcionais espalhados pelo código fonte das classes funcionais Java é realizada buscando-se fragmentos de código, que podem ser: palavras chave, atributos, métodos, classes e objetos. Observando-se essas características no código fonte em três sistemas utilizados como estudos de caso [15],

elaborou-se a Lista de Indícios, para auxiliar o engenheiro de software a identificar os trechos de código fonte que podem conter indícios de ser de um determinado interesse. Esses indícios sinalizam a possibilidade da existência de um interesse. A Lista de Indícios é composta para auxiliar a sinalização de seis diferentes interesses, mas neste artigo somente o de persistência é utilizado. Os indícios para esse interesse são expressos na forma de expressões regulares.

Na Seção seguinte é apresentada a aplicação da abordagem Aspecting para dois sistemas implementados em Java, sendo que um utiliza o padrão de projeto Camada de Persistência [18] e o outro não utiliza.

4. Aplicação da Abordagem *Aspecting* a dois Sistemas Exemplo

Os sistemas usados neste estudo de caso são:

a) O sistema de Oficina Eletrônica efetua consertos em produtos eletrônicos como televisores, vídeo-cassetes e forno de microondas e seus técnicos recebem comissão referente aos consertos efetuados, utiliza *servlets* para a comunicação das interfaces em HTML com o banco de dados, utilizando o padrão Camada de Persistência.

b) O sistema de Caixa de Banco, obtido pela Internet, implementado na linguagem Java e que não possui documentação. Sua função é gerir contas correntes e clientes do banco.

As etapas da abordagem são realizadas seguindo os passos indicados na Tabela 1.

Etapa I - a partir da execução dos sistemas foram criados casos de uso que representasse cada funcionalidade existentes neles, de acordo com diretrizes especificas não apresentadas aqui [15]. Como exemplos de casos de uso têm-se do sistema a): cadastrar produtos, cadastrar funcionários, pagar comissão entre outros; e do sistema b) têm-se: cadastrar cliente, cadastrar contas, efetuar débitos entre outros. O diagrama de classes de cada sistema foi gerado com apoio da ferramenta *Omondo* [13]. Caso essa não seja utilizada a abordagem Aspecting fornece diretrizes para que esses diagramas sejam criados. Devido a restrição de espaço, somente será apresentada com maiores detalhes a etapa II, Tratar interesses.

Etapa II - É evolutiva, composta por três passos que são repetidos até que todos os interesses da Lista de Indícios sejam pesquisados ou até que o engenheiro de software decida finalizar o processo. Porém, como comentado anteriormente, será exibido somente o processo realizado para a identificação, modelagem e implementação do interesse de Persistência em Banco de Dados Relacional (implementado com o padrão Camada de Persistência e sem o padrão).

No **Passo II.1**: A expressão regular que auxilia o reconhecimento do interesse de Persistência em banco de Dados Relacional é mostrada na Figura 2.

<i.Persistencia.BD> = 'Connection' <statements> |'Connection' <statements> <SQL>|
<SQL> <statements> 'Connection'| 'PreparedStatement' <statements>|
'PreparedStatement' <statements> <SQL> | <SQL> <statements> 'PreparedStatement'|
'ResultSet' <statements>| 'ResultSet' <statements> <SQL> | <SQL> <statements> 'ResultSet'

Figura 2 - Indícios do interesse de Persistência em Banco de Dados Relacional.

Para o engenheiro de software conseguir identificar todo o código fonte que pertence ao interesse de Persistência no sistema de Oficina Eletrônica, foi necessário que se conhecesse a granulosidade[2] da implementação do padrão de projeto Camada de Persistência. Pois atributos pertencentes ao padrão como os mostrados na Figura 3 (a) não foram reconhecidos pela expressão regular que faz parte da Lista de Indícios, a Figura 5 (b) mostra uma linha que pode ser identificada com a expressão regular.

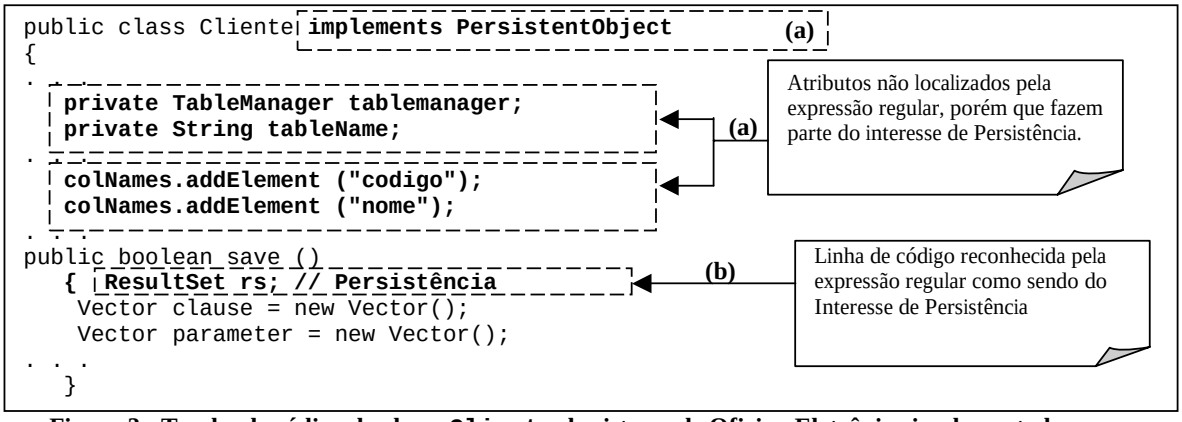

Figura 3 - Trecho de código da classe `Cliente` do sistema de Oficina Eletrônica implementado com o padrão de projeto Camada de Persistência.

[2] Como e o quanto espalhado é a implementação do padrão.

Para o sistema de Caixa de Banco a expressão regular que reconhece os indícios de persistência ajudou a identificar todos os trechos que pertencia ao interesse, como pode ser visto na Figura 4.

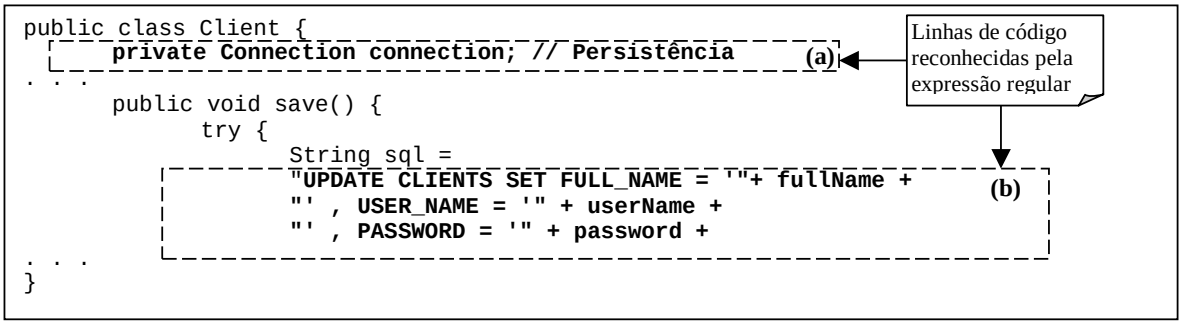

Figura 4 - Trecho de código da classe `Client` do sistema de Caixa de Banco implementado sem o padrão de projeto Camada de Persistência.

O interesse de Persistência incorpora o sub interesse de conexão com o banco de dados. Para o sistema de Oficina Eletrônica, que implementa o padrão de projeto Camada de Persistência, o sub interesse de conexão está implementado não somente nas classes que realizam a persistência, mas também nas classes *servlets* e em uma classe especifica que contém métodos para conexão e desconexão com o banco de dados. Para o sistema de Caixa de Banco, o sub interesse de conexão está somente implementado nas classes que realizam a persistência. Por esse motivo, neste artigo optou-se por não considerar a implementação do sub interesse de conexão com o banco de dados.

Passo II.2: os aspectos que modularizam o interesse de Persistência em Banco de Dados Relacional são inseridos nos diagramas de classes seguindo as diretrizes próprias do interesse exibidas na Figura 5. Essas diretrizes foram elaboradas com base em trabalhos de extensão da notação UML, para a programação orientada a aspectos, [3], [14], e na experiência obtida com a realização de estudos de caso [15].

1. Para cada classe que contenha trechos marcados como sendo do interesse de Persistência em Banco de Dados Relacional.
    1.1. Adicionar, ao diagrama de classes de projeto, um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.
    1.1.1. Adicionar ao *Aspect-class* os atributos do interesse, utilizando o modelo:
    [Tipo] [Nome da classe].[Atributo];
    Sendo que:
    [Tipo]: indica o tipo do atributo que foi retirado da Classe A.
    [Nome da classe]: indica o nome da Classe A.
        [Atributo]: indica-se o nome do atributo que foi retirado da Classe A.
        1.1.2. Adicionar ao *Aspect-class* os métodos que pertencem totalmente ao interesse, utilizando o modelo:
        [Tipo] [Nome da classe].[Método];
    Sendo que:
    [Tipo]: indica o tipo do retorno do método retirado da Classe A.
    [Nome da classe]: indica o nome da Classe A.
    [Método]: indica o nome do método que retirado da Classe A.
        1.1.3. Caso existam métodos que dependam de outros interesses:
            1.1.3.1 Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:
            (<<after>> || <<before>> || <<around>>):[Nome do Ponto de Corte]
        Sendo que:
            (<<after>> || <<before>> || <<around>>): deve-se optar pela opção dependendo da ordem em que deseja ser executar o corpo da sugestão (*advice*).
        [Nome do Ponto de Corte]: Deve-se indicar o nome do ponto de corte.
    1.2. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<Introduction>>, se existirem métodos e/ou atributos relacionados ao interesse.
        1.2.1. Criar uma declaração do tipo: {atributo_1, atributo_n, método_1(), método_n()}, como papel da Classe A.
    1.3. Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>>, se existirem métodos que dependem de outros interesses.
        1.3.1. Criar uma declaração do tipo: [nome do ponto de corte], como papel do *Aspect-class*.
        Sendo que:
        [nome do ponto de corte]: indica o nome do ponto de corte.
    1.3.2.Criar um a declaração do tipo: (**?** || **#**) [ponto de junção], como papel da Classe A.
    Sendo que:
        [nome do ponto de junção]: indica a assinatura do método que contém o trecho de código marcado com interesse.

Figura 5 - Diretrizes para modelar o interesse de Persistência em Banco de Dados Relacional.

O sub padrão Gerenciador de Tabelas, presente na implementação do sistema de Oficina Eletrônica, está implementado em uma classe, `TableManager`, não tendo o espalhamento do seu código por outras classes do

sistema. Assim, optou-se por preservar a implementação existente. Na Figura 6 a classe `TableManager` contém um relacionamento de associação com os aspectos modelados, pois esses utilizam métodos dessa classe.

O diagrama de classes parcial do sistema de Oficina Eletrônica mostrado na Figura 6 apresenta os aspectos `AspectCliente` e `AspectConserto` que adicionam às classes de aplicação, métodos e atributos referentes ao padrão de projeto Camada de Persistência. Os aspectos também entrecortam as classes, interceptando seus construtores para poder atribuir valores específicos da classe aos atributos referentes ao padrão.

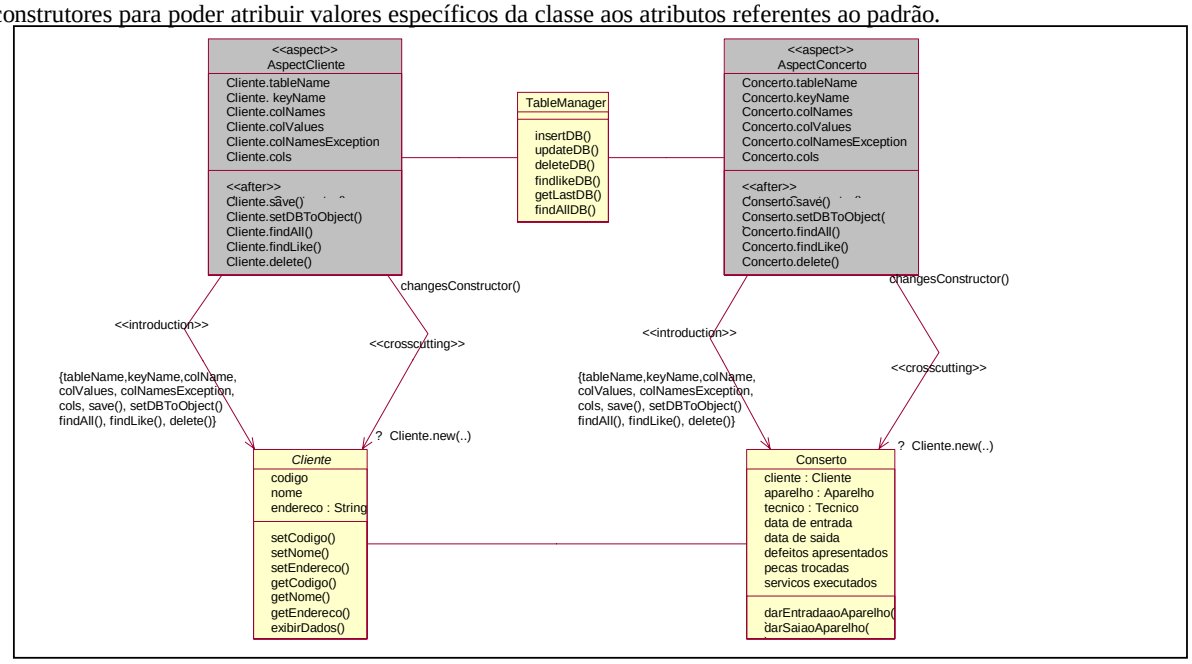

Figura 6 - Diagrama de classes parcial para o sistema de Oficina Eletrônica.

O sistema de Caixa de Banco tem em seu diagrama de classes parcial, Figura 7, dois aspectos, o `AspectClient` e `AspectAccount` que apenas introduzem métodos referentes ao interesse de persistência às classes que realizam a persistência com o banco de dados.

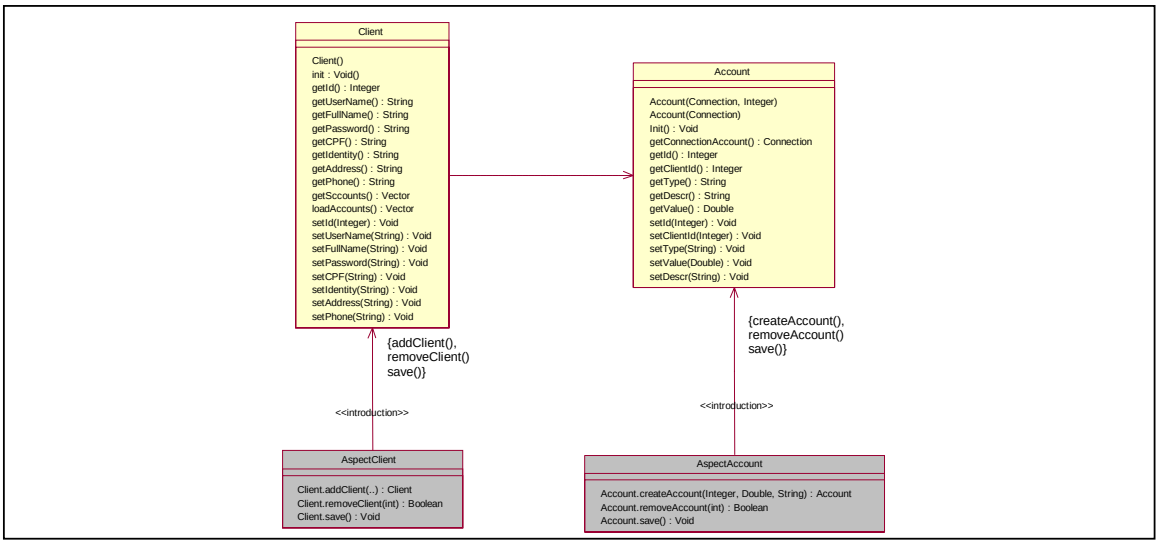

Figura 7 - Diagrama de classes parcial para o sistema de Caixa de Banco.

Os relacionamentos de associação existentes entre os aspectos e as classes rotulados com o estereótipo *<<introduction>>* indicam que no aspecto existem atributos e/ou métodos, relacionados no papel da classe, que serão introduzidos estaticamente na classe indicada, em tempo de compilação.

Já os relacionamentos de associação existentes entre os aspectos e as classes rotulados com o estereótipo *<<crosscuting>>* indicam que na classe relacionada existe um ou mais pontos de junção, relacionado no papel da classe, que compõe o ponto de corte relacionado no papel do aspecto. O sinal gráfico "?", existente no papel da classe para esses relacionamentos, Figura 6, indica a captura da execução do ponto de junção [3].

Os relacionamentos existentes não rotulados indicam a existência de uma associação entre classes e/ou entre classes e aspectos.

Passo II.3: Os aspectos que foram modelados nos dois diagrama de classes são implementados na linguagem AspectJ, seguindo as diretrizes do interesse de Persistência em Banco de Dados Relacional, Figura 8. Trechos dessas implementações para os sistemas de Oficina Eletrônica e Caixa de Banco são exibidos, respectivamente, nas Figuras 9 e 10.

Para cada *Aspect-class* criado no diagrama de classes de projeto para o interesse de Persistência em Banco de Dados Relacional.

1. Criar um aspecto em AspectJ, utilizando o modelo:
`public aspect [nome do aspecto]`
Sendo que:
`[nome do aspecto]`: é o nome indicado no *Aspect-class*.

1.1. Inserir ao aspecto os atributos, se existirem, utilizando o modelo:
`[tipo] [nome da classe].[nome do atributo];`
Sendo que:
`[tipo]`: refere-se ao tipo do atributo indicado no *Aspect-classes* do diagrama de classes de projeto.
`[nome da classe]`: refere-se ao nome da Classe A.
`[nome do atributo]`: refere-se ao nome do atributo indicado no *Aspect-classes* do diagrama de classes de projeto.

1.1.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.2. Inserir ao aspecto os métodos que pertencem ao interesse, se existirem, utilizando o modelo:
`[tipo] [nome da classe].[nome do método] {[corpo do método]}`
Sendo que:
`[tipo]`: refere-se ao tipo do método indicado no *Aspect-classes* do diagrama de classes de projeto.
`[nome do método]`: refere-se ao nome do método indicado no *Aspect-classes* do diagrama de classes de projeto.
`[corpo do método]`: Insere-se o corpo do método da Classe A referente a esse método que esta sendo implementado no aspecto.

1.2.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado.

1.3. Criar um ponto de corte, se existir, de acordo com as informações inseridas no diagrama de classes de projeto, utilizando o modelo:
`pointcut [nome do ponto de corte] ([objeto]): [execution || call]([ponto de junção]) && [target || this]([objeto]) && args([argumentos]);`
Sendo que `[nome do ponto de corte]`: refere-se ao nome do ponto de corte, indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de projeto.
`([objeto])`: refere-se ao objeto que se deseja capturar.
`[execution || call]`: refere-se ao (? || #) indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de projeto.
`[target || this]`: refere-se ao indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de projeto.
`([argumentos])`: refere-se aos argumentos indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de projeto.

1.4. Criar uma sugestão (*advice*), utilizando o modelo:
`[after || before || around] ([argumentos]): [nome do ponto de corte] ([argumentos]) {[corpo da sugestão]}`
Sendo que: `[after || before || around]`: refere-se ao indicado no *aspect-method* do *Aspect-class* do diagrama de classes de projeto.
`[argumentos]`: são os mesmos referenciados no passo 1.3.
`[nome do ponto de corte]`: é o mesmo referenciado no passo 1.3.
`[corpo da sugestão]`: insere-se o trecho de código fonte referente ao interesse marcado no método da Classe A.

1.4.1. Utilizar comentários para marcar na classe todo o trecho de código que foi implementado

Figura 8 - Diretrizes para implementar o interesse de Persistência em Banco de Dados Relacional.

```
public aspect AspectCliente
{    public String Cliente.tableName;
     public String Cliente.keyName;
     . . .
pointcut changesConstructor(Cliente c): target (c) && execution (Cliente.new(int, String,
String, int, String, String, String, String, String, String));
      after (Cliente c): changesConstructor(c)
       {   c.setTableName("Cliente");
           c.setKeyName("codigo");
           c.setColNames("codigo");
           . . .
           id = new Integer(c.getCodigo());
           c.setColValues(id);
           c.setColValues(c.getNome());
           . . .
      public boolean Cliente.save() { . . .}
      public boolean Cliente.setDBToObject() { . . .}
      public ResultSet Cliente.findall() { . . .}
      public ResultSet Cliente.findlike() { . . .}
      public boolean Cliente.delete () { . . .}
}
```

Figura 9 - Trechos da implementação do aspecto `AspectCliente`, do sistema de Oficina Eletrônica.

```
public aspect AspectClient{
public void Client.save(){
       try {
              String sql =
              "UPDATE CLIENTS SET FULL_NAME = '"+ fullName +
              "' , USER_NAME = '" + userName +
              "' , PASSWORD = '" + password + }
              . . .
public void Client.addClient(){}

public void Client.removeClient(){}
. . .
}
```

Figura 10 - Trechos da implementação do aspecto `AspectClient`, do sistema de Caixa de Banco.

Etapa III - A comparação entre as versões dos sistemas Orientado a Objetos e dos sistemas Orientado a Aspectos foram realizada segundo as diretrizes apresentadas na Figura 11. As interfaces originais do sistema OO foram mantidas e todas as operações que anteriormente foram preservadas.

1. Se todos os casos de uso forem atendidos com sucesso, então:
    1.1. Retirar do código fonte do sistema Orientado a Aspectos, todos os trechos de código que foram marcados após a implementação dos aspectos.
2. Senão:
    2.1. Enquanto a funcionalidade do caso de uso não for atendida:
        2.1.1. Para cada interesse implementado no sistema orientado a aspecto:
            2.1.1.1.Desmarcar no código fonte os trechos desse interesse e isolar o(s) aspecto(s) implementado(s) para esse interesse.
            2.1.1.2. Verificar se a funcionalidade para o caso de uso, que não foi atendida, agora é atendida.
            2.1.1.3. Se for atendida, então:
                2.1.1.3.1. Procura-se outra forma de implementar o interesse em aspectos, ou esse interesse não é implementado.
                2.1.1.3.2. Retornar ao passo 1.1.

Figura 11 – Diretrizes para Comparar Sistema OA com OO.

5. Considerações Finais

Este artigo apresentou o processo de reengenharia realizado com a aplicação da abordagem *Aspecting* em dois sistemas Orientados a Objetos. Esses sistemas contêm o interesse de Persistência, sendo que o sistema de Oficina Eletrônica utiliza o padrão Camada de Persistência e o sistema de Caixa de Banco não utiliza. Esse interesse foi identificado nos sistemas, modelado e implementado na linguagem AspectJ. A principal contribuição deste trabalho é mostrar que o interesse de persistência pode ser implementado em aspectos utilizando a abordagem *Aspecting*, mesmo se a persistência estiver implementada seguindo o padrão de projeto Camada de Persistência.

Os sistemas resultantes deste estudo de caso possuem vantagens da boa localização de código e diminuição das redundâncias provenientes da utilização do paradigma Orientado a Aspectos, conseqüentemente o reuso, a manutenção desses sistemas é mais fácil. Porém há um aumento de módulos do sistema com a adição da estrutura aspecto (*aspect*).

Os aspectos implementados para o interesse de Persistência podem ser refinados a fim de se tornarem abstratos o bastante para obter maiores vantagens quanto ao reuso. A elaboração de um *Framework* Orientado a Aspectos que está sendo realizada atualmente pode ser talvez apontado como uma solução mais eficaz para o reuso da implementação do interesse de Persistência.

A expressão regular que faz parte da abordagem *Aspecting* não foi eficaz para encontrar todos os indícios do interesse de Persistência quando foi aplicada na versão implementada com o padrão. Porém, na versão onde o interesse de Persistência não utiliza um padrão a lista auxiliou a encontrar todos os indícios do interesse. Esse fato ocorreu, pois a expressão regular foi elaborada a partir de um sistema que não continha o padrão Camada de Persistência. Os novos indícios encontrados na implementação do Padrão Camada de Persistência foram adicionados na expressão regular, para que em novos estudos de caso essa expressão regular seja mais eficiente.

No processo realizado, deseja-se reusar o código fonte existente, separando o interesse que estava espalhado e entrelaçado. Porém se o processo for de engenharia avante o interesse de Persistência pode ser considerado isoladamente. Desse modo, as diretrizes mostradas neste trabalho para modelagem do interesse e implementação em aspectos, podem ser seguidas, sem que a marcação do interesse no código fonte original tenha de ocorrer.

6. Referências

[1] Cagnin, M.I. Avaliação das Vantagens quanto à Facilidade de Manutenção e Expansão de Sistemas Legados Sujeitos à Engenharia Reversa e Segmentação. Dissertação de Mestrado, DC-UFSCar, 1999.

[2] Camargo, V.V.; Ramos, R.A.; Penteado, R.A.D.; Masiero, P.C. Projeto Baseado em Aspectos do Padrão Camada de Persistência. In: Simpósio Brasileiro de Engenharia de Software (SBES), Manaus, 2003.

[3] Camargo, V.V. Masiero, P.C. UML-AOF – Um Perfil UML para o Projeto de Frameworks Orientados a Aspectos. Relatório Técnico, ICMC-USP, 2004. Disponível para download em: *http://www.icmc.usp.br/~valter/publicacoes*.

[4] Chung, L.; Nixon, B.; Yu, E.; Mylopoulos, J. *Non-functional requirements in software engineering*. In: *Boston: Kluwer Academic*, pág. 439, 1999.

[5] Cysneiros, L.M.; Leite, J.C.S.P. Definindo Requisitos Não Funcionais. In: Simpósio Brasileiro de Engenharia de Software (SBES´97), pág. 49-54, Outubro 1997.

[6] Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[7] Elrad, T. Aksit, M.; Kiczales, G.; Lieberherr, K.; Ossher, H.. *Discussing Aspects of AOP*. In: Anais do ACM, pág. 33 – 38, 2001.

[8] Gama, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[9] Hilsdale, E.; Hugunin, J. *Advice Weaving in AspectJ*. Submetido à *3rd International Conference on Aspect-Oriented Software Development* – AOSD. abril 2004.

[10] Kiczales, G.; Lamping, J.; Mendhekar, A. *RG: A Case-Study for Aspect-Oriented Programming*. Artigo Técnico, *SPL97*. Xerox Palo Alto Research Center, 1997.

[11] Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. *Griswold, W.G. Getting Started with AspectJ*. In: Anais do ACM, pág. 59-65, Outubro 2001.

[12] Noda, N. and Kishi, T. "*Implementing Design Patterns Using Advanced Separation of Concerns*", in *Proceedings of OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, FL, October 2001.

[13] Omondo – Ferramenta para Modelagem – *Plug-in* disponível em http://www.omondo.com.Último acesso em 04/2004.

[14] Pawlak, R., Duchien, L., Florin G., Legong-Aubry, F., Seinturier, L, Martelli, L. *A UML Notation for Aspect-Oriented Software Design*. In: *Workshop of Aspect Oriented Modeling with UML of Proceedings of Aspect Oriented Software Development Conference (AOSD) 2002*, Enschede, Abril, 2002.

[15] Ramos, R., A. *Aspecting*: Abordagem para Migração de Sistemas OO para Sistemas AO. Dissertação de Mestrado. Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, São Carlos - SP, maio de 2004.

[16] Ross, D., T. *Structure Analysis (SA): A language for communicating Ideas*. In: *IEEE Transaction Software Engineering*, 1997.

[17] Tarr, P.; Ossher, H.; Harrision, W.; Sutton, S. M. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In: *International Conference on Object-Oriented Programming (ICSE)*, 1999.

[18] Yoder, J. W.; Johnson, R. E.; Wilson, Q. D. Connecting Business Objects to Relational Databases. Conference on the Pattern Languages of Programs, 1998.