Um Processo de Reestruturação de Código Baseado em Aspectos

Ricardo Argenton Ramos[1,*] / Rosângela Penteado[1] / Paulo César Masiero[2,*]
[1] Departamento de Computação, Universidade Federal de São Carlos, Via Washington Luís, Km 235 - Caixa Postal 676, São Carlos SP, Brasil 13565-905, +55 16 33518233
[2] Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, Av. do Trabalhador São-Carlense, 400 - Caixa Postal 668, São Carlos SP, Brasil 13560-970, +55 16 33739671
e-mail: [rar, rosangel]@dc.ufscar.br, masiero@icmc.usp.br

Resumo

A maioria dos sistemas Orientado a Objetos (OO) tem em seu código fonte requisitos não funcionais entrelaçados com os requisitos funcionais. Assim, há dificuldade na manutenção e na expansão desses sistemas. o processo, proposto neste artigo, denominado Aspecting auxilia a elicitação dos requisitos não funcionais do código fonte, chamados de interesses, para sua posterior implementação utilizando uma linguagem que suporte aspectos, como por exemplo AspectJ. Diagramas de casos de uso e de classes de projeto apóiam a etapa de entendimento da funcionalidade do sistema OO. Diretrizes auxiliam na descoberta dos interesses existentes, por meio da Lista de Indícios, e a inserção de aspectos nos diagramas de classes UML, de projeto. Outras diretrizes conduzem à etapa de implementação. Três estudos de caso foram utilizados para a criação do processo Aspecting

Abstract

The great majority of Object Oriented systems (OO) have in its source code non-functional requirements entangled with functional requirements. Thus, the maintenance and expansion in those systems are difficult. The approach proposed in this paper, named Aspecting, helps to elicit the non functional requirements, called concerns, from OO system source code, for subsequently implementation of this system using the language that supports aspects, as for example AspectJ. Use cases and design class diagrams help the understanding of OO system functionalities. Guidelines are established for finding concerns, through the List of Supposed Concerns, and for modeling the concerns identified, using UML class diagram. Other guidelines govern the implementation of concerns as aspects. Three case studies had been used for the Aspecting approach creation.

1. Introdução

O bom resultado do desenvolvimento de um sistema é diretamente dependente da elicitação de requisitos. Essa etapa é essencial para que a funcionalidade do sistema seja completamente conhecida e corresponda às expectativas de seus usuários. Outro ponto a ser observado é com relação aos requisitos não funcionais. Eles, tanto quanto os requisitos funcionais, são vitais para o completo desenvolvimento de sistemas.
Os requisitos funcionais podem ser definidos como as funções ou atividades que o sistema faz (quando pronto) ou fará (quando em desenvolvimento). Devem ser definidos claramente e relatados explicitamente [5]. Podem ser elaborados a partir do relato das necessidades do cliente e/ou usuário, em que uma equipe de projeto pode especificar efetivamente um sistema,

* Financiado pelo CNPq

suas funções, desempenho, interfaces, restrições, etc., conforme as fases e subfases de um método de desenvolvimento de sistemas ou software.

Técnicas de modularização que levam a programas orientados a objetos são bem sucedidas, porém sua abordagem de modularização em sistemas de software de acordo com um único interesse inerente é insuficiente, por não prover estruturas suficientes para desenvolver sistemas complexos [13, 15]. Embora as pesquisas em engenharia de software estejam bastante amadurecidas, a manutenção de software permanece como um problema central à área. O processo de manutenção envolve não apenas a correção de erros, mas, sobretudo, a adequação do sistema para a integração de novas tecnologias e novos requisitos.

Requisitos não funcionais [4], ao contrário dos funcionais, não expressam função (transformação) a ser implementada em um sistema, expressam condições de comportamento e restrições que devem prevalecer. Como exemplo, pode-se citar tratamento de exceções, segurança, persistência de dados e desempenho, entre outros. Chung e outros [4] mencionam a dificuldade de tratar requisitos não funcionais em alguns sistemas, pois um requisito pode ser não funcional em um sistema e ser funcional em outro sistema. Para o contexto deste trabalho o termo "interesses" refere-se aos requisitos não funcionais em nível de projeto.

Dada a importância da separação de interesses, para maior reusabilidade e melhor manutenção, este trabalho propõe um processo denominado *Aspecting*, que conduz o engenheiro de software à elicitação de alguns interesses não funcionais em sistemas orientados a objetos existentes, implementados em linguagem Java, para posterior reorganização desses sistemas no paradigma orientado a aspectos utilizando a linguagem AspectJ [14].

Na Seção 2 são comentados trabalhos relacionados com a separação de interesses e programação orientada a aspectos; na Seção 3 as principais características da linguagem AspectJ são descritas; na Seção 4 o processo *Aspecting*, com suas diretrizes, etapas e passos é apresentada. Uma aplicação do processo *Aspecting* em um sistema exemplo é tratada na Seção 5 e as considerações finais são apresentadas na Seção 6.

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 [17], 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.

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 [1] implementam com aspectos o padrão de projeto Camada de Persistência

[24] e comentam as vantagens de se ter os interesses (padrões e sub-padrões que compõe a camada de persistência) separados do código funcional. Uma das vantagens é o reuso de código do padrão por causa da 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.
Elrad e outros [7] afirmam que 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. 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.
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 [13].

3. A Linguagem AspectJ

Em AspectJ os 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 [14].
A composição dos aspectos com as classes é executada em tempo de compilação, sobre *bytecode* ou código fonte Java. Uma vez executada a compilação, os entrecortes estarão permanentemente incorporados às classes, não sendo possível alterar a composição durante a execução do programa. Hilsdale e Hugunin [12] afirmam que 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 (*advices*) e pontos de corte (*pointcut*)); ii) o compilador utiliza essas informações para gerar as classes compostas (*bytecode* compatível com a especificação Java).
Os seguintes conceitos caracterizam AspectJ como uma linguagem orientada a aspectos:
Pontos de junção (*join points*): são estruturas bem definidas 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. [16] 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*).

4. Processo *Aspecting*

A elaboração do Processo denominado *Aspecting*, que é composto por um conjunto de diretrizes para conduzir a elicitação de aspectos em sistemas orientados a objetos, ocorreu de forma prospectiva, com a realização de três estudos de casos. Os sistemas utilizados, Orientados a Objetos (OO), foram escolhidos em diferentes domínios:
i) Sistema de caixa de banco, obtido via Internet [20], desenvolvido em linguagem Java, utiliza um banco de dados MS-Access, cuja conexão é feita via JDBC utilizando ODBC com o *driver* que já vem embutido no próprio MS-Access. É composto por onze classes: três contêm as regras de negócio e oito tratam da interface, que é feita utilizando o pacote `javax.swing` disponível no próprio Java.
ii) Sistema de estação flutuante, obtido de Weiss e Lai [25], emula um mecanismo de captação de informações referente à velocidade do vento em alto mar. Esse sistema é desenvolvido na linguagem Java e contêm doze classes, a interface com o usuário é realizada através de um *applet*, via navegador.
iii) Sistema GNU Grep [9], faz parte do sistema RegExp do Unix. Sua função é procurar em arquivos, por linhas que contêm correspondência a uma dada lista de padrões de expressões regulares. Quando essa correspondência é encontrada em uma linha, ela é copiada para uma saída padrão, ou é feita qualquer outra combinação de saída que o usuário designar utilizando algumas opções pré-estabelecidas.
A Figura 1 ilustra o processo que será realizado: código original OO com entrelaçamento de interesses (P); reconhecimento e implementação desses interesses em aspectos (A), e o código implementado sem o entrelaçamento e espalhamento dos interesses (C).

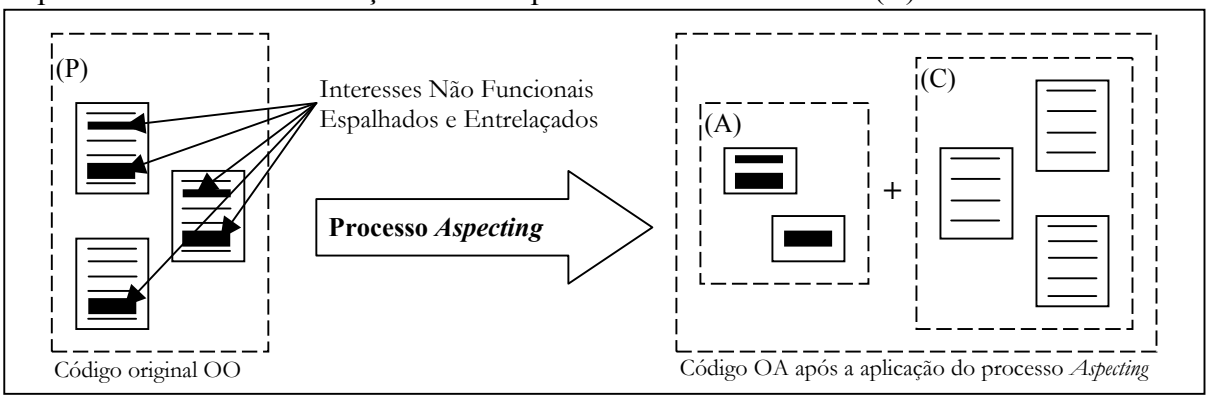

Figura 1. Ilustração do processo realizado pelo Processo *Aspecting*.

Os interesses não funcionais espalhados pelo código fonte das classes funcionais Java são compostos por fragmentos de código, que podem ser: palavras chave, atributos, métodos, classes e objetos. Observando-se essas características no código fonte dos três sistemas, elaborou-se uma lista, chamada de Lista de Indícios, para auxiliar o engenheiro de software a identificar os trechos de código fonte que podem conter indícios de um determinado interesse. Essa lista é apresentada na seção seguinte.

4.1. Lista de Indícios

A Lista de Indícios conta com seis interesses, cinco encontrados nos estudos de caso realizados e um, Rastreamento (*Tracing*), inferido de Chavez [3], portanto, ela não é completa. Os outros interesses existentes, segundo a literatura especializada, não foram

incluídos nessa lista, por não terem sido encontrados nos sistemas utilizados e pela falta de tempo para realizar simulações que comprovassem a corretitude dos indícios que caracterizam o interesse. Eles têm por objetivo auxiliar a identificar possíveis interesses existentes no código fonte, assim dois tipos de indícios foram considerados:

1 - **Para interesses que são sensíveis ao contexto**: que contêm fragmentos de código fonte específicos do interesse e estão em um determinado contexto, sem que existam palavras reservadas no código fonte que possam induzir o engenheiro de software a identificar um determinado tipo de interesse.

2 - **Para interesses não sensíveis ao contexto**: quando existem palavras reservadas no código fonte que induzem o engenheiro de software a um determinado tipo de interesse. Para esses casos foram criadas expressões regulares, que contêm essas palavras reservadas, que servem para busca manual ou automatizada, quando existir uma ferramenta para tal fim.

Kiczales [14] propõem a classificação em três níveis diferentes, que também foram considerados neste trabalho: a) de desenvolvimento: inseridos durante o desenvolvimento de aplicações, para a descoberta de erros, testes e desempenho do sistema; b) de Produção: referem-se a construções relativas à produção e afetam um número reduzido de métodos; e c) de Tempo de execução: utilizados para melhorar o desempenho em tempo de execução.

A Tabela 1 apresenta somente três interesses e os respectivos indícios para encontrá-los no código fonte Orientado a Objetos. O interesse de Rastreamento é sensível ao contexto e os outros dois não são sensíveis ao contexto. Os demais interesses que compõem a Lista de Indícios são: Persistência em Banco de Dados Relacional (não sensível ao contexto) e Tratamento de Erros (sensível ao contexto) (Nível-Produção), Persistência Memória em Temporária (*Buffering*) (não sensível ao contexto) (Nível-Tempo de Execução) [21].

Tabela 1. Composição e divisão da Lista de Indícios.

Nível de Interesse	Interesse	Indícios
Desenvolvimento	1) Rastreamento (*tracing)*	Métodos que imprimem mensagens de texto, tais como: - System.out.print("<Mensagem>"); - System.out.println("<Mensagem>"); <Mensagem> contém informações ao desenvolvedor sobre os nomes ou valores dos métodos, ou indicando a posição física do método no código fonte.
Produção	2)Tratamento de Exceção	<i.trat.exceção>= 'try' '{' <statements> '}' 'catch' '('<statements> ')' '{'<statements>'}' 'finally' '{' <statements> '}' | 'try' '{' <statements> '}' 'catch' '('<statements> ')' '{'<statements>'}'
Tempo de Execução	3) Programação Paralela (*Thread*)	<i.Programacao.Paralela> = 'extends' 'Thread' <statements> 'run()' '{'<statements> '}' | 'extends' 'Thread' <statements> 'run()' '{' 'Thread' '.' <statements> <statements> '}' | 'extends' 'Thread' <statements> 'run()' '{' <statements> 'Thread' '.' <statements> <statements> '}' | 'extends' 'Thread' <statements> 'run()' '{' <statements> 'Thread' '.' <statements> '}' | 'implements' 'Runnable' <statements> 'run()' '{'<statements> '}' | 'implements' 'Runnable' <statements> 'run()' '{' 'Thread' '.' <statements> <statements> '}' | 'implements' 'Runnable' <statements> 'run()' '{' <statements> 'Thread' '.' <statements> <statements> '}' | 'implements' 'Runnable' <statements> 'run()' '{' <statements> 'Thread' '.' <statements> '}' | 'new' 'Thread' '(' <statements> ')'

4.2. Diretrizes para Modelar e para Implementar Aspectos

Após a identificação dos interesses no código Orientado a Objetos há necessidade de modelá-los como aspectos no diagrama de classes de projeto para depois implementá-los em linguagem de programação orientada a aspectos. As diretrizes para modelar o diagrama de classes com aspectos foram elaboradas com base em trabalhos de extensão da notação UML para a programação orientada a aspectos [19, 2], e na experiência obtida com a realização de estudos de caso.
Hanenberg e Unland [10] propõem em seu trabalho idiomas que facilitam o desenvolvedor a trabalhar com as novas características inseridas pela linguagem AspectJ [14]. Neste trabalho são fornecidas diretrizes para a implementação de aspectos a partir de experiências adquiridas em alguns trabalhos [1] e de estudos de caso realizados.

4.3. Etapas e Passos do Processo

A Figura 2 mostra as atividades que compreendem o processo *Aspecting*, utilizando a notação SADT [22]. As atividades são representadas por retângulos e sua execução pode ser apoiada por ferramentas CASE ou de apoio à programação orientada a aspectos.

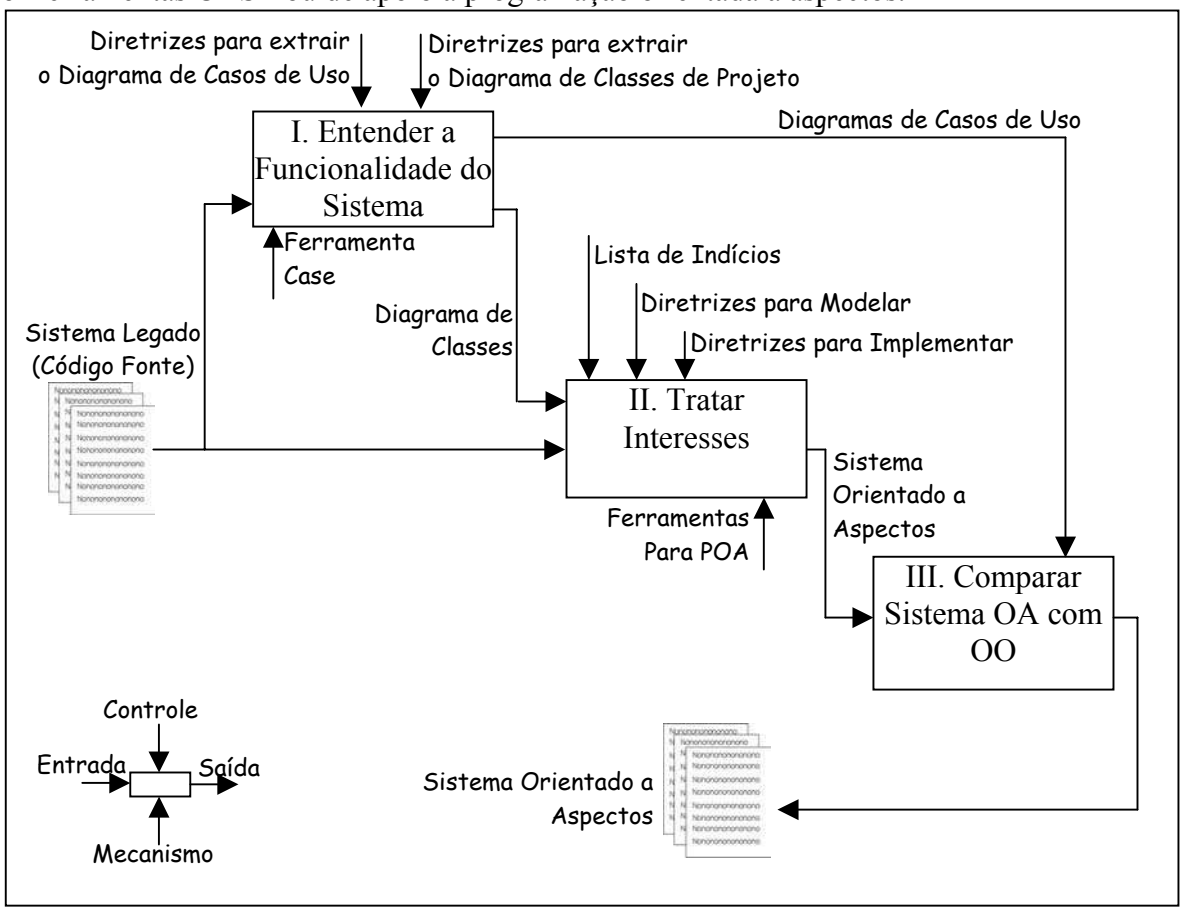

Figura 2. Etapas do processo *Aspecting*.

A Tabela 2 apresenta na primeira coluna as etapas do Processo *Aspecting*, com breve descrição do que deve ser realizado em cada uma delas; na segunda coluna, os passos correspondentes à etapa e na terceira coluna a descrição dos passos.

Tabela 2. Etapas e passos do processo *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* [18].
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 e para cada classe do sistema: 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 para esse fim.Retornar ao passo 2.1.
III. Comparar Sistema Orientado a Aspectos com o Orientado a Objetos **Descrição**: Verificar a funcionalidade do sistema OA que foi gerado.	**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.

5. Aplicação do Processo *Aspecting* a um Sistema Exemplo

Esta seção apresenta a aplicação do processo *Aspecting* para 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 do processo são realizadas seguindo os passos indicados na Tabela 2.

Etapa I - executa-se o sistema e elaboram-se, os diagramas de casos de uso (para extrair a funcionalidade) e de classes de acordo com as diretrizes apresentadas nas Figura 3 e 4.

Curso Normal	Cursos Alternativos
1. Criar um ator usuário. **2**. Identificar os menus existentes no sistema OO como as opções desse sistema. **3**. Criar um caso de uso para cada opção do menu. **4**. Verificar se toda a funcionalidade foi representada. **5**. Encerrar caso de uso.	**2**. Não existem menus no sistema. **2.1**. Considerar as interações existentes com o usuário como opções de menu. **2.2**. Encerrar caso de uso. **4**. A funcionalidade não foi verificada completamente **4.1**. Identificar as funcionalidades presentes nos sub-menus. **4.2**. Criar um caso de uso para cada uma dessas funcionalidades. **4.3**. Encerrar caso de uso.
Criar Relacionamento entre Casos de Uso	
Curso Normal	Cursos Alternativos
1. Fixar um caso de uso, identificando se esse chama outro caso de uso, até que todos os casos de uso sejam verificados. **2**. Colocar o relacionamento <<*include*>> com origem no caso de uso que chama e com destino no caso de uso chamado. **3**. Encerrar caso de uso.	**2**. Caso de uso é chamado opcionalmente. **2.1**. Colocar o relacionamento <<*extends*>> com origem no caso de uso chamado e com destino no que chamou.

Figura 3. Diretrizes para criação dos diagramas de casos de uso e suas descrições.

Para adicionar as classes do sistema no diagrama de classes.
1. Para cada classe do sistema (`public class` ou `public abstract class`). **1.1**. Criar uma classe correspondente a essa classe, com o mesmo nome no diagrama. **1.1.1**. Considerar como atributos da classe criada no diagrama de classes cada um dos atributos de tipo primitivo que existirem na classe no código fonte Orientado a Objetos. **1.1.2.** Associar à classe criada no diagrama de classes os métodos que estão implementados associados a ela no código fonte Orientado a Objeto.
Para adicionar as interfaces do sistema no diagrama de classes.
1. Para cada interface existente no sistema (`public interface`). **1.1**. Criar uma interface correspondente à essa com o mesmo nome no diagrama de classes. **1.1.1**. Associar à interface criada no diagrama de classes os métodos, que estão implementados no código fonte Orientado a Objeto.
Para adicionar os relacionamentos de associação, de dependência e de generalização entre as classes criadas no diagrama de classes.
1. Para cada classe do sistema. **1.1.** Se existir atributo de tipo não primitivo da linguagem, para cada atributo. **1.1.1**. Criar um relacionamento direcional de associação da classe que implementa esse tipo, com a classe que contém esse atributo. **1.2**. Se existir dentro do método uma instanciação de um objeto, para cada objeto. **1.2.1**. Criar um relacionamento direcional de dependência da classe que contém esse método, para a classe que implementa o objeto. **1.3**. Se existir na assinatura da classe a palavra reservada extends que indica herança. **1.3.1**. Criar um relacionamento direcional de generalização dessa classe para a classe pai indicada. **1.4**. Se existir na assinatura da classe a palavra reservada `implements` que indica a implementação de uma interface. **1.4.1**. Criar um relacionamento direcional de generalização dessa classe para a interface indicada.

Figura 4. Diretrizes para elaborar o diagrama de classes a partir do código fonte Orientado a Objetos implementado em linguagem Java.

Observando-se o menu Caixa/Cliente do sistema de Caixa de Banco têm-se cinco opções: Procurar, Adicionar, Editar, "*Deletar*" e Limpar. Assim, foi gerada as descrições do curso normal e curso alternativo de cada caso de uso. A Tabela 3 mostra a descrição do caso de uso para a opção "Adicionar Cliente".

Tabela 3. Descrição do caso de uso para a opção "Adicionar Cliente".

Adicionar Cliente	
Curso Normal	**1**. Obter msg01 = Nome do usuário (cliente) **2**. Verificar que Cliente não existe. **3**. Obter mais inf. msg01 = Nome Completo, *Password*, CPF, RG, Endereço, Telefone, Nome do Pai. **4**. Efetuar Inclusão. **5**. Emitir msg02 = "Cliente cadastrado".
Curso Alternativo	**1**. msg01 = " " (em branco). **1.1**. Emitir msg02 = "Não foi possível adicionar o cliente", "Preencha todos os campos ou tente outro *user name*", "Exception: Java.sql.SQLException " msg. **1.2**. Abandonar caso de uso. **2**. Cliente já cadastrado. **2.1**. Emitir msg02 = "*User name* já existe. Tente outro *user name*". **2.2**. Abandonar caso de uso.

Etapa II - É evolutiva, composta por três passos: II.1. Marcar Interesse, II.2. Modelar Interesse e II.3. Implementar Interesse, 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.

No **Passo II.1**: Os indícios de tratamento de exceção, Tabela 1, foram encontrados e marcados no código fonte como mostra a Figura 5 (a), (b) e (c).

```
...public void save() {
     try { // Tratamento de Exceção 01 (a)
           String sql = "UPDATE CLIENTS SET FULL_NAME = '"+ fullName +
                        "' , USER_NAME = '" + userName +
                        ...
                        "' , ADDRESS = '" + address +
                        "' , PHONE = '"+ phone  +
                        "' WHERE ID = "+ id;
           Statement stm = this.connection.createStatement();
           System.out.println(sql);
           stm.executeUpdate(sql);
           stm.close();}
     catch (SQLException e) { // Tratamento de Exceção 01 (b)        (c)
           System.out.println("Exception:"+e.toString());//Trat de Exc01
   }
}...
```

Figura 5. Trecho do método `save()` da classe `Client` comentado como sendo parte do interesse de Tratamento de Exceção.

Passo II.2: o aspecto que modulariza o Interesse de Tratamento de Exceção é inserido no diagrama de classes seguindo as diretrizes próprias do interesse, Figura 6. A nomenclatura utilizada é a seguinte: Classe A – refere-se à classe do diagrama de classes de projeto na qual se identificou um interesse; *Aspect-class* – refere-se à modelagem do interesse ou parte do interesse em aspecto; *Aspect-methods*- referem-se aos pontos de corte do aspecto e a ordem em que esse é executado; ? - refere-se à captura da execução do método; <<*Aspect*>> - Indica um Aspecto; <<*after*>> - Indica que a execução da sugestão (*advice*) deve ocorrer após a execução ou chamada do ponto de junção. A nomenclatura completa utilizada nas diretrizes para modelar os interesses nos diagramas de classe podem ser encontrados em [21].

1. Verificar no trecho de código fonte marcado como sendo do interesse de Tratamento de Exceção, qual tipo de exceção está sendo tratada. Caso não exista um aspecto para o tipo utilize o Caso I, se não utilize o Caso II.

Caso I (não existe um aspecto para o tipo de exceção tratada)

I.1.1. Adicionar ao diagrama de classes de projeto um *Aspect-class* com o estereotipo <<aspect>>, seguido do Nome do aspecto.

I.1.1.1. Criar um "*aspect-method*" e adicioná-lo ao *Aspect-class*, utilizando o modelo:
<<after>> throwing ([Tipo da Exceção] [variável]): [nome do ponto de corte]
Sendo que: [Tipo da Exceção]: indica o tipo de exceção que é tratado no bloco `try{...} catch(..){...}`.
[variável]: indica nome da variável que armazena a mensagem de erro que foi gerada.
[nome do ponto de corte]: indica o nome do ponto de corte.

I.1.2 Criar um relacionamento de associação com origem no *Aspect-class* e destino na Classe A, utilizando o estereótipo <<crosscuting>> .

I.1.2.1 Criar uma declaração do tipo: declare soft:[Tipo do Interesse] : [nome do ponto de corte] como papel do *Aspect-class*.

I.1.2.2. 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 bloco `try{...} catch(..){...}`.

Caso II (quando já existe um aspecto para o tipo de exceção tratada).

II.1.1. Adicionar ao relacionamento já criado como papel da Classe A, a captura da execução(?) ou da chamada(#) do ponto de junção, que corresponde à assinatura do método que contém o bloco `try{...} catch(..){...}`.

Figura 6. Diretrizes para modelar o interesse de Tratamento de Exceção.

A Figura 7 mostra o diagrama de classes com o aspecto de Tratamento de Exceção após seguir as diretrizes de modelagem da Figura 6. Nesse diagrama encontram-se as classes que contém as regras de negócio do sistema de caixa de banco (`Client, Account, BankBox`) e o aspecto `AspectException` que modulariza o interesse de tratamento de Exceção. O relacionamento existente entre cada classe e o aspecto é de associação, com origem no aspecto e destino na classe que contém os métodos que serão entrecortados. Foi utilizada a notação de papel da classe para indicar os pontos de junção associados ao papel do aspecto que representa o ponto de corte e o tipo da exceção que está sendo tratada. A notação utilizada foi proposta por Pawlak [19] e estendida por Camargo [2].
O estereótipo <<crosscuting>> indica que no Aspecto existem pontos de corte que entrecortam as classes associadas ao relacionamento. Por exemplo, no relacionamento do aspecto com a classe `Client` o ponto de corte `TrataSQLException()` utiliza como pontos de junção a captura da execução dos métodos `save()` e `removeAccount()`. Os relacionamentos existentes com as outras classes, que contém as regras de negócio e possuíam interesses entrelaçados, e os aspectos são tratados da mesma forma. A declaração `declare soft: [tipo da exceção]` indica o tipo de exceção que está sendo tratada no ponto de corte do aspecto. Os relacionamentos entre classes não contêm nomes. Esses casos podem ser observados na Figura 7. O exemplo completo e mais detalhes sobre a notação pode ser vista em [21].

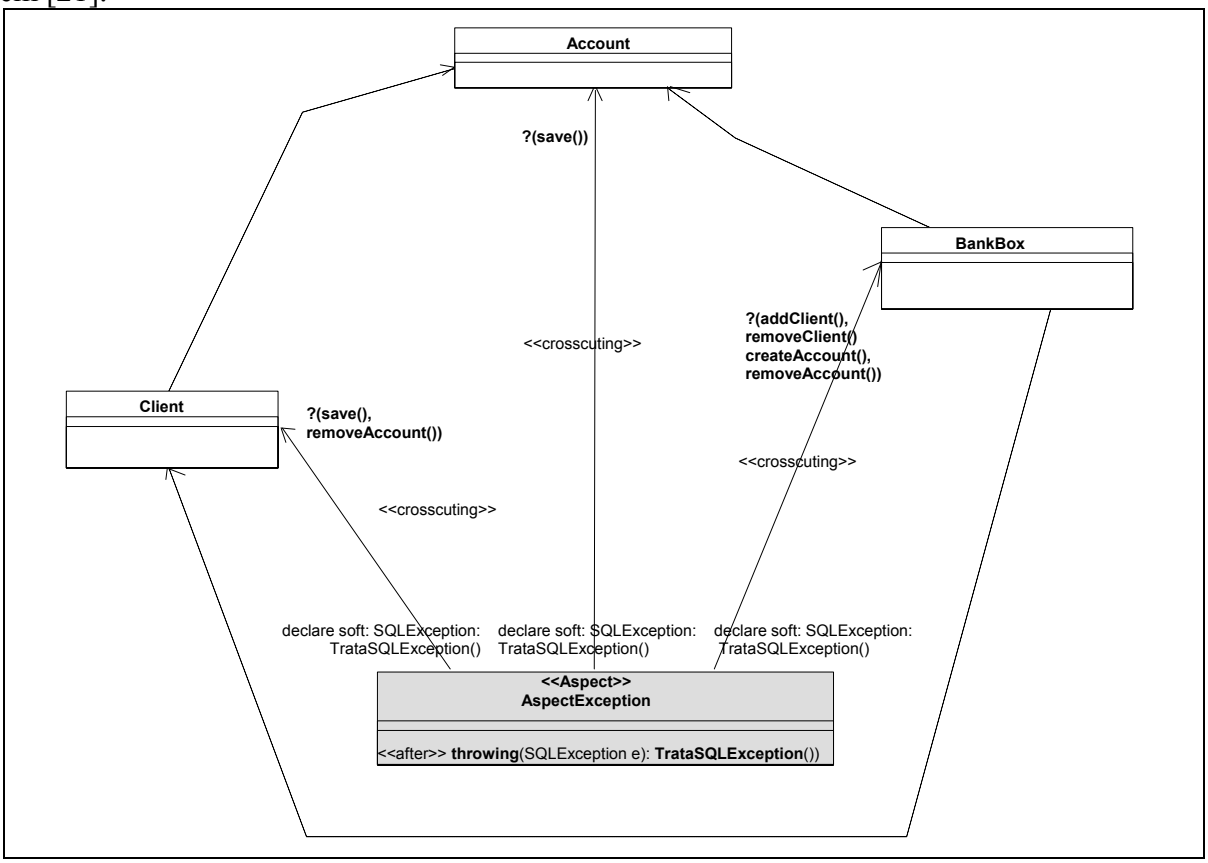

Figura 7. Diagrama de classes (parcial) do sistema de Caixa de Banco, com o aspecto modelado.

Passo II.3: é implementado na linguagem AspectJ o aspecto que foi modelado no diagrama de classes, seguindo as diretrizes do interesse de Tratamento de Exceção, Figura 8 e 9. A implementação do aspecto de Tratamento de Exceção é exibida na Figura 10.

Caso I (não existe um aspecto para o tipo de exceção tratada).

1. Criar um aspecto em AspectJ, utilizando o modelo: `public aspect [nome do aspecto]`

1.1. Criar um ponto de corte de acordo com as informações inseridas no diagrama de classes de projeto, utilizando o modelo:

`pointcut [nome do ponto de corte]: execution || call([ponto de junção]);`

Sendo que: `[nome do ponto de corte]`: indicado no papel do *Aspect-class* no relacionamento de entrecorte do diagrama de classes de projeto.

`[execution || call]`: refere-se ao `(? || #)` indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de projeto.

`[ponto de junção]`: indicado no papel da Classe A no relacionamento de entrecorte do diagrama de classes de projeto.

1.2. Criar a declaração "*declare soft*", utilizando o modelo:

`declare soft: [Tipo da Exceção] : [nome do ponto de corte];`

Sendo que: `[Tipo da Exceção]`: indicado no papel do *Aspect-class*.

`[nome do ponto de corte]`: indicado no papel do *Aspect-class*.

1.3. Criar uma sugestão (*advice*), utilizando o modelo:

`after throwing([Tipo da Exceção] [variável]): [nome do ponto de corte] { [corpo do aspect-method] }`

Sendo que: `[Tipo da Exceção]`: é o mesmo referenciado no passo 1.2.

`[variável]`: refere-se a variável indicada no *aspect-method* do *Aspect-class*.

`[nome do ponto de corte]`: indicado no *aspect-method* do *Aspect-class*.

`[corpo do aspect-method]`: insere-se o código fonte para tratar a exceção, utilizando o modelo:

`try{ throw new [Tipo da Exceção] } catch([Tipo da Exceção] [variável 2]) {[Corpo do bloco catch]}`

Sendo que: `[Tipo da Exceção]`: é o mesmo referenciado no passo 1.2.

`[variável 2]`: uma variável que deverá ser criada para tratar a exceção no bloco `catch`.

`[Corpo do bloco catch]`: insere-se o conteúdo do bloco `catch` que está marcado em um trecho de código fonte da Classe A correspondente ao trecho de código que deu origem a este aspecto.

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

Figura 8. Diretrizes para auxiliar a implementação do interesse de Tratamento de Exceção (caso I).

Caso II (quando já existe um aspecto para o tipo de exceção tratada).

1. Adicionar ao ponto de corte já criado no aspecto, o pontos de junção, utilizando o modelo:

`pointcut [ponto de corte já criado]: [execution || call]([ponto de junção I]) || ... || [execution || call] ([ponto de junção n]);`

Sendo que: `[execution || call]`: `(? || #)` indicado no papel da Classe A.

`[ponto de junção n]`: ponto de junção indicado no papel da Classe [a]

2. Se o conteúdo do bloco `catch` da Classe A contido no trecho de código marcado como interesse já estiver contido no bloco catch do aspecto, então:

2.1. Utilizar comentários para marcar na Classe A todo o trecho de código que foi implementado.

3. Senão:

3.1. Para cada bloco `catch` da Classe A com o conteúdo diferente do bloco `catch` existente no Aspecto:

3.1.1. Adicionar um comando de decisão dentro do bloco `catch` do aspecto, que faça a comparação em tempo de execução do ponto de junção que foi alcançado, utilizando o modelo:

```
if(thisJoinPoint.getStaticPart().equals([ponto de junção])
{[conteúdo do bloco catch]}
```

Sendo que: `[ponto de junção]`: refere-se a cada ponto de junção que tenha o conteúdo do bloco catch diferente do já existente no aspecto.

`[conteúdo do bloco catch]`: insere-se o código fonte referente ao tratamento da exceção para o `[ponto de junção]`.

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

Figura 9. Diretrizes para auxiliar a implementação do interesse de Tratamento de Exceção (caso II).

```
public aspect AspectException {
      pointcut TrataSQLException():
     execution(public void Client.save())
       || execution(public void Client.removeAccount(..))
       || execution(public void Account.save())
       || execution(public Account BankBox.createAccount(..))
       || execution(public Account BankBox.removeAccount(..))
       || execution(public Client BankBox.addClient(..))
       || execution(public Client BankBox.removeClient(..));
   declare soft: SQLException : TrataSQLException();
   after() throwing(SQLException e): TrataSQLException(){
   try{
      throw new SQLException();
   }
   catch(SQLException e1){
     System.out.println(("Exception : " + e1.toString());
   }
  }
}
```

Assinatura do método mostrado na Figura 5 inserido como

Figura 10. Aspecto de Tratamento de Exceção implementado em AspectJ.

Na segunda iteração, da etapa II, indícios como o do interesse de Persistência em Banco de Dados Relacional, Rastreamento (*tracing*) e de Programação Paralela entre outros, devem ser pesquisados no código fonte. Devido à restrição de espaço, somente um interesse é apresentado. O processo completo utilizado para migrar esse sistema para um orientado a aspectos pode ser visto em [21].

Etapa III - A comparação entre o sistema Orientado a Objetos e o sistema Orientado a Aspectos foi realizada segundo as diretrizes apresentadas na Figura 11. A interface original do sistema OO foi mantida no OA e todas as operações que anteriormente eram realizadas continuam ocorrendo da mesma forma.

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 o sistema OA com OO.

Pôde-se inferir que a funcionalidade do sistema original foi preservada realizando-se as mesmas operações no sistema orientado a aspectos com base nos casos de uso gerados na etapa I. A Figura 12 mostra a interface com o usuário do sistema de Caixa de Banco Orientado a Aspectos, para a opção de cadastro de clientes. Um cliente não pode ser adicionado ao sistema se o campo "Nome do usuário" apresentado na interface não estiver preenchido. Caso haja essa tentativa uma exceção do tipo `Java.sql.SQLException` é

lançada, como pode ser visto na parte inferior da Figura 12 e que foi descrito no diagrama de casos de uso da versão Orientada a Objetos, Tabela 3.

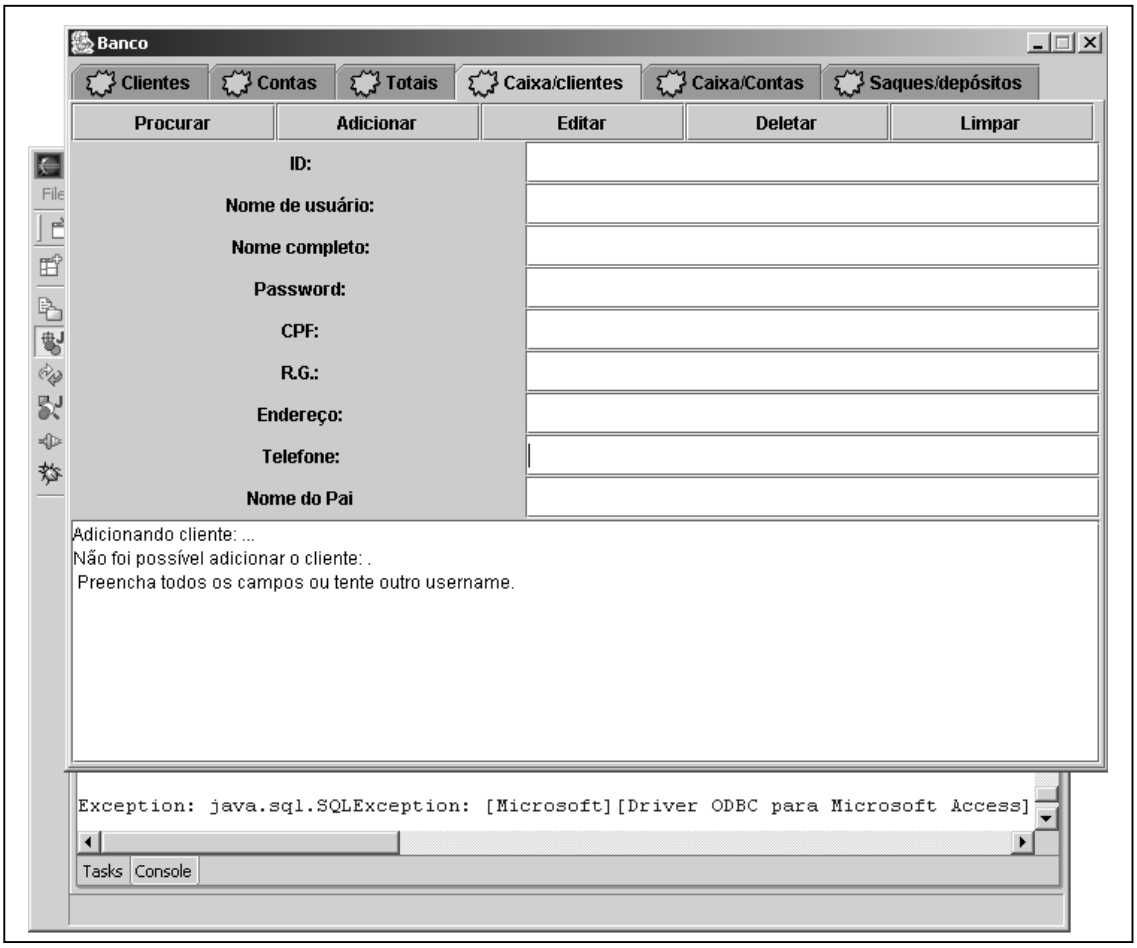

Figura 12. Interface do Sistema de Caixa de Banco, para a opção de Cadastrar Clientes.

6. Considerações Finais

O processo *Aspecting* é composto por três etapas distintas: Entender a Funcionalidade do Sistema, Tratar o Interesse e Comparar Sistema Orientado a Aspectos com o Orientado a Objetos. Cada etapa foi desenvolvida para que o engenheiro de software realize a migração de sistemas Orientados a Objetos para sistemas Orientados a Aspectos de forma segura e que o produto obtido tenha qualidade.

Em seu estágio atual a abordagem é especifica para migrar sistemas implementados na linguagem Java para sistemas na linguagem AspectJ. Caso se deseje migrar sistemas implementados em outras linguagens OO, é necessário adaptar a Lista de Indícios, para que essa reflita as características existentes na linguagem, alterando-se as expressões regulares (sensíveis ao contexto) e os métodos que as implementam (não sensíveis ao contexto). Se a linguagem alvo for a AspectJ, as diretrizes de modelagem, para elaboração do diagrama de classes e de implementação permanecem basicamente as mesmas.

Caso se deseje implementar os aspectos em uma linguagem diferente de AspectJ, que apóie a Programação Orientada a Aspectos deve-se adaptar as diretrizes de implementação.

Como a Etapa II – Tratar Interesses é evolutiva, ela possibilita que o código fique com menor grau de entrelaçamento de requisitos funcionais e não funcionais a cada iteração, facilitando a identificação de interesses por parte do engenheiro de software. Como desvantagem pode-se citar que a funcionalidade do sistema pode ser alterada, quando são descartadas as interações e os relacionamentos existentes entre os interesses em cada iteração da Etapa. Um cuidado especial deve ser tomado, pois existem casos em que um interesse pode ter precedência sobre outro.

Um conjunto de requisitos não funcionais é composto por interesses que podem ser implementados em aspectos: como os interesses de Tratamento de Exceção, Persistência em Banco de Dados Relacional, e pelos interesses que não são implementados em aspectos, como os interesses de Qualidade de Serviços [7]. Os aspectos que são tratados pelo processo *Aspecting* fazem parte de um subconjunto do conjunto dos interesses que podem ser implementados em aspectos, Figura 13.

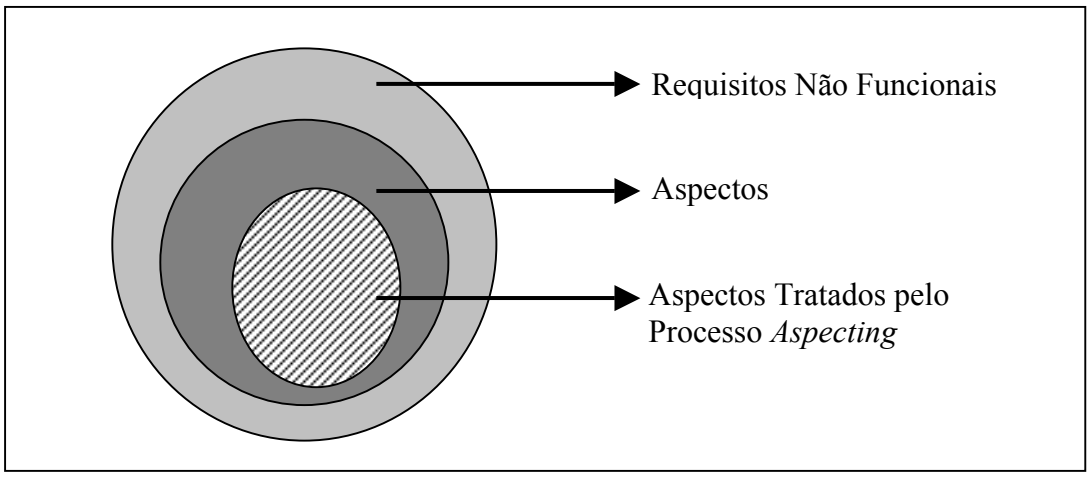

Figura 13. Ilustração do subconjunto de aspectos tratados pelo processo *Aspecting* dentro do conjunto de requisitos funcionais.

O ponto fraco da *Aspecting* é que não são para todos os interesses que existem indícios de como encontrá-los no código fonte Orientado a Objetos (OO). Assim, pode ainda ocorrer o entrelaçamento e espalhamento de algum interesse no código fonte OO e esse continuar entrelaçado no sistema Orientado a Aspectos. Periodicamente, pode-se reavaliar a Lista de Indícios e completá-la para incluir outros interesses.

Algumas ferramentas para encontrar os indícios dos interesses no código fonte como a Ferramenta *Aspect Mining Tool* AMT [11], podem ser utilizadas juntamente com o processo para agilizar o passo de marcar o interesse. Outros recursos, como o próprio serviço de busca (*Search*) do ambiente de desenvolvimento, podem ser utilizados para auxiliar a busca de indícios. Um apoio computacional pode ser construído para auxiliar a identificação de indícios de interesses no código fonte Orientado a Objetos, utilizando as expressões regulares criadas no processo *Aspecting*.

No processo proposto deseja-se reusar o código fonte existente, separando os interesses que estão espalhados e entrelaçados. Porém, se o processo de desenvolvimento de sistemas Orientados a Aspectos for desejado, a elaboração de casos de uso já possibilita a implementação com aspectos [23]. Desse modo, as diretrizes elaboradas para a modelagem do diagrama de classes de projeto com aspectos e para a implementação de aspectos podem ser seguidas, eliminando os passos referentes à marcação dos interesses no código fonte original, na Etapa I.

Referências Bibliográficas

1. 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 - 2003, pág. 114-129, Manaus - AM, Brasil, 2003.
2. Camargo, V.V. Um Perfil UML para Projeto de Sistemas Orientados a Aspectos. In: Relatório Técnico - Universidade de São Paulo – ICMC/USP, São Carlos - SP, Brasil, Abril de 2004.
3. Chavez, C.F.G.; Lucena, C.J.P. Design Support for Aspect-Oriented Software Development. In: Doctoral Symposium and Pôster Section of OOSPLA 2001. Tampa Bay, Florida, USA, Outubro, 2001.
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´1997, Fortaleza - CE, Brasil, pág. 49-54, Outubro 1997.
6. Czarnecki, K.; Eisenecker, U. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.
7. Elrad, T.; Filman R. Bader A. Aspect-Oriented Programming. In: Communications of ACM, pág. 29-32, 2001.
8. Gamma, E.; Helm, R.; Johnson, R.E.; Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
9. GNU - Grupo de desenvolvedores para a Linguagem Java – Códigos fonte disponíveis em: http://www.cacas.org/java/gnu/regexp.html. - Último acesso em 05/2003.
10. Hanenberg, S.; Unland, R. AspectJ Idioms for Aspect-Oriented Software Construction. In: 2nd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), Boston, MA, Março 17, 2003.
11. Hannemann, J.; Kiczales, G. Overcoming the Prevalent Decomposition in Legacy Code. In: Workshop on Advanced Separation of Concerns, International Conference on Software Engineering. Toronto, Canadá, Maio 2001.
12. Hilsdale, E.; Hugunin, J. Advice Weaving in AspectJ. Submetido à 3rd International Conference on Aspect-Oriented Software Development – AOSD. abril 2004.
13. Kiczales, G.; Lamping, J.; Mendhekar, A. RG: A Case-Study for Aspect-Oriented Programming. In: SPL97. Xerox Palo Alto Research Center, Technical Report, 1997.
14. 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.
15. Kulesza, U.; Silva, D. M. Reengenharia do Projeto do Servidor Web JAWS Utilizando Programação Orientada a Aspectos. In: XIV Simpósio Brasileiro de Engenharia de Software - SBES, 2000, Sessões técnicas, João Pessoa - PB, Brasil.
16. Laddad, R. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Company, Connecticut – USA, 2003. 512 p.
17. Noda, N.; Kishi, T. Implementing Design Patterns Using Advanced Separation of Concerns. In: OOPSLA 2001, Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay, Florida, Outubro 2001.
18. Omondo – Ferramenta para Modelagem – Plug-in disponível em http://www.omondo.com. - Último acesso em 04/2004.
19. 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.
20. Portal Java - Downloads de código fonte de sistemas implementados em Java. Disponível em: http://www.portaljava.com.br. - Último acesso em 04/2004.
21. Ramos, R., A. Aspecting: Abordagem para Migração de Sistemas OO para Sistemas AO. Dissertação de Mestrado. In: Dissertação de Mestrado – Programa de Pós Graduação em Ciência da Computação, Universidade Federal de São Carlos, São Carlos - SP, 2004.
22. Ross, D., T. Structure Analysis (SA): A language for communicating Ideas. In: IEEE Transaction Software Engineering, 1977.
23. Souza, G.; Silva, I.; Castro, J. Adapting the FRN Framework to Aspect-Oriented Requirements Engineering. In: Simpósio Brasileiro de Engenharia de Software (SBES), pág. 177-192, Manaus, Brasil, outubro 2003.
24. Yoder, J. W.; Johnson, R. E.; Wilson, Q. D. Connecting Business Objects to Relational Databases. In: Conference on the Pattern Languages of Programs 5 (PLOP). Monticello-IL, EUA, 1998.
25. Weiss, D., Lai, C. T. R. Software Product-Line Engineering: a family-based software development process. Ed. Addison Wesley, 1999. 426 p.