RAFAEL ALESSANDRO GATTO

ESTRATÉGIAS PARA REESTRUTURAÇÃO DE CÓDIGO LEGADO VISANDO À UTILIZAÇÃO DE ASPECTOS

MARINGÁ

2007

RAFAEL ALESSANDRO GATTO

ESTRATÉGIAS PARA REESTRUTURAÇÃO DE CÓDIGO LEGADO VISANDO À UTILIZAÇÃO DE ASPECTOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Prof.ª Dr.ª Elisa Hatsue Moriya Huzita

MARINGÁ

2007

Dados Internacionais de Catalogação-na-Publicação (CIP)

(Biblioteca Central - UEM, Maringá – PR., Brasil)

Gatto, Rafael Alessandro

G263e Estratégias para reestruturação de código legado visando à utilização de aspectos / Rafael Alessandro Gatto. -- Maringá : [s.n.], 2007.

111 f. : il., figs., tabs.

Orientadora : Profª. Drª. Elisa Hatsue Moriya Huzita.

Dissertação (mestrado) - Universidade Estadual de Maringá. Programa de Pós-graduação em Ciência da Computação, 2007.

1. Refatoração. 2. Programação orientada a aspectos. 3. Código legado - Estratégias de reestruturação. I. Universidade Estadual de Maringá. Programa de Pós-graduação em Ciência da Computação. II. Título.

CDD 22.ed. 005.11

RAFAEL ALESSANDRO GATTO

ESTRATÉGIAS PARA REESTRUTURAÇÃO DE CÓDIGO LEGADO VISANDO A UTILIZAÇÃO DE ASPECTOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Estadual de Maringá, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Aprovado em 06/09/2007.

BANCA EXAMINADORA

Profa. Dra. Elisa Hatsue Moriya Huzita
Universidade Estadual de Maringá – DIN/UEM

Profa. Dra. Tania Fatima Calvi Tait
Universidade Estadual de Maringá – DIN/UEM

Prof. Dr. Edmundo Sérgio Spoto
Centro Universitário Eurípides de Marília – PPGCC/UNIVEM

DEDICATÓRIA

Dedico este trabalho

Aos meus pais João e Dolores Gatto
pelo incentivo, carinho e compreensão

AGRADECIMENTOS

Inicialmente gostaria de agradecer a Capes pelo apoio financeiro, que possibilitou a minha dedicação exclusiva a realização deste trabalho. Também gostaria de agradecer à minha família, em especial aos meus pais, João Gatto e Dolores Maria Gatto, por terem sido exemplos de persistência, trabalho e honestidade, como também por todo amor e apoio moral e financeiro que me deram para alcançar meus objetivos, mesmo com a distância. Ao meu irmão e minha cunhada, Jefferson e Jeruza, meu Sobrinho João Rafael e minha Vovó Maria, que são meus grandes amigos e que de alguma forma me apoiaram no meu trabalho.

Agradeço a minha orientadora, Elisa Hatsue Moriya Huzita, por acreditar no meu esforço e trabalho e também por me transmitir sua experiência e seus conhecimentos. E ainda aos meus amigos e colegas Gécen e Coral, sem o qual a realização deste trabalho não seria possível.

Agradeço aos amigos e colegas de mestrado, com quem dividi momentos de estresses e preocupações, e o mais valioso, momentos de alegria e descontração.

Agradeço ao meu amigo Murilo e seus pais, Cristóvão e Marina, que com o tempo se tornaram parte da minha família.

Agradeço a Deus pela sorte de ter o amor e compreensão da minha família e amigos.

RESUMO

Os constantes e contínuos avanços da tecnologia tanto no que diz respeito às plataformas de hardware como de software, fazem com que muitos sistemas, apesar de ainda atenderem a seus requisitos e serem considerados estáveis, se tornem completamente obsoletos. Tal situação pode se agravar ainda mais se, eventualmente, diferentes programadores tiverem realizado a manutenção ao longo de sua existência, podendo apresentar problemas estruturais contrariando assim as boas práticas de programação. Com isso, a busca por métodos, técnicas, ferramentas e abordagens de desenvolvimento de software que auxiliem os desenvolvedores na produção de software com mais qualidade e que seja de fácil manutenção têm crescido consideravelmente. Motivado pela idéia de encontrar uma solução proveitosa para melhorar a legibilidade de códigos legados e, conseqüentemente, aumentar a sua manutenibilidade, este trabalho tem por objetivo investigar, para então propor estratégias para realizar a reestruturação de tais códigos, considerando a possibilidade de incluir aspectos. Com isso, obtém-se um novo código mais legível, melhor organizado, de fácil manutenção e apto à adição de novas funcionalidades. As estratégias, aqui propostas, combinam técnicas, já consolidadas, dos diferentes tipos de refatoração, sistematizando-as em um processo contínuo e evolutivo, a fim de obter um código com menor ocorrência de problemas estruturais. Eliminados os problemas estruturais, procede-se à identificação de possíveis interesses transversais para então modularizá-los em aspectos e desta forma tirar os proveitos oferecidos pela programação orientada a aspectos.

Palavras Chave: Refatoração. Programação Orientada a Aspectos. Estratégias de Reestruturação

ABSTRACT

The frequent and continuous advances of the technology related both with the hardware platform so as with software, bringing several problems for the systems. So, although these systems still satisfy the requirements of their users and are considered stable, they become completely obsolete. This situation can be worst if, eventually several developers had being participated of its development and programming. Probably they have structural problems in their code, generating what can be characterized as bad smells. In this way, the search for methods, techniques, tools and an approach to assist the developers in the production of systems with better quality and more maintainable had growth considerably. So, motivated by the idea to find a beneficial solution to improve the legibility of legacies codes, and consequently to facilitate their maintainability, the present dissertation has the objective to investigate and propose a set of strategies to offer an adequate support to reorganize such codes, considering the possibility of inclusion of aspects. It will make possible to obtain a new and more legible code, better organized, maintainable, and able to include new functionalities. The proposed strategies take the advantages of already consolidated techniques of different refactoring types, systemizing them in a continuous and evolutionary process, in order to obtain a code that avoid the presence of bad smells. Once the bad smells related with structural problems had been eliminated, it is initiated the search to find out the crosscutting concerns, modularize them in aspects and so take the advantages offered by aspect oriented programming.

Key words: Refactoring. Aspect Oriented Programming. Strategies for Refactoring, Bad Smells

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

OO	Orientada a Objetos
POA	Programação Orientada a Aspectos
ASOA	Arquitetura de Software Orientada a Aspectos
FOA	*Frameworks* Orientados a Aspectos
FOO	*Frameworks* orientados a objetos
FT	*Frameworks* Transversais
XP	*eXtreme Programming*
OA	Orientado a Aspectos
XML	eXtensible Markup Language
DSOA	Desenvolvimento de Software Orientado a Aspectos

SUMÁRIO

1 Introdução

No decorrer dos anos, a busca por métodos, técnicas e ferramentas que auxiliem os desenvolvedores na produção de sistemas de maior qualidade e de fácil manutenção tem tido um crescimento considerável. Entretanto, percebe-se ainda que sistemas legados frequentemente apresentam alto custo de manutenção e sérios problemas estruturais, o que aumentam as dificuldades para integração de novas funcionalidades, e também, a interação com novas tecnologias e/ou abordagens. Dessa forma, a reestruturação sistemática de código legado se apresenta como uma disciplina de grande impacto para evolução de sistemas.

No entanto, é importante também que projetos nem sempre sejam alterados como decorrência de uma reestruturação de código conduzida com disciplina à luz de abordagens como a orientação a aspectos.

Assim, a refatoração pode ser entendida como o processo de alteração da estrutura interna do código orientado a objetos (OO), sem no entanto ferir as funcionalidades já existentes. Tal alteração deve ser conduzida almejando o contínuo melhoramento de código, aumentando a modularidade e eliminando redundâncias.

A orientação a objetos é atualmente o paradigma dominante de desenvolvimento de software. Apesar de apresentar algumas limitações, a orientação a objetos trouxe muitos benefícios em relação às abordagens anteriores. Uma dessas limitações é a incapacidade de modularizar de forma adequada os interesses (do inglês, *concerns*) de um sistema de software (KICZALES, 1997). Para solucionar essas limitações, surgiram algumas extensões como a Programação Orientada a Aspectos (KICZALES, 1997), a programação orientada a assunto (OSSHER, 1999) e a programação adaptativa (LIEBERHERR, 1994). A Programação Orientada a Aspectos (POA) trata da separação de interesses que estão entrelaçados com os requisitos funcionais de um sistema.

O dinamismo proporcionado pela utilização de aspectos possibilita um verdadeiro progresso no que tange à reutilização, demonstrando um caminho promissor para alcançar maior eficiência e eficácia no desenvolvimento e manutenção de software. Entretanto 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. O alto

custo associado a essas adequações (COLEMAN, 1994) estimula o desenvolvimento de sistemas mais flexíveis a mudanças futuras. Assim, dada às vantagens oferecidas pelo Desenvolvimento de Software Orientado a Aspectos (DSOA), a aplicação da refatoração em um programa OO legado, com a subseqüente busca pelos aspectos mostrou-se uma alternativa promissora.

Com o objetivo de reduzir custos de manutenção, aumentar a produtividade no desenvolvimento, melhorar a legibilidade do código e ainda, aumentar a reutilização, esta dissertação propõe um conjunto de estratégias para apoiar a adequação de código às boas práticas de programação.

Tal conjunto de estratégias tem como principal característica constituir-se em um conjunto sistemático e evolutivo de passos que são executados pautados em catálogos de refatoração: a partir de uma reestruturação OO até uma extração de aspectos. Com isto, o código resultante estaria pronto também para migrar para a abordagem de desenvolvimento orientado a aspectos.

1.1 Motivação

Atualmente, diversas empresas estão se vendo obrigadas a migrar seus sistemas para novas linguagens de programação e/ou tecnologias, devido aos elevados custos de manutenção. Em outros casos a procura por melhorias nos sistemas existentes se dá pela dificuldade de adaptação e evolução destes. Além disso, podem ocorrer situações em que várias pessoas tenham participado do desenvolvimento e programação. Dessa maneira, faz-se necessário a busca por mecanismos adequados que apóiem a reestruturação de código.

De uma forma geral, sistemas implementados na linguagem Java, que possuem o entrelaçamento do código relativo aos interesses não funcionais com o código relativo ao interesse funcional da aplicação estão sendo migrados para a abordagem orientada a aspectos apenas levando em consideração os princípios de espalhamento e entrelaçamento de interesses. No entanto, fica claro que a evolução de um sistema se dá com a reestruturação completa de seu código e não apenas com a identificação e modularização de alguns interesses transversais. Portanto, verificou-se a necessidade de unir as vantagens de refatoração com a orientação a aspectos de forma a desenvolver um código de maior qualidade e de fácil manutenção.

1.2 Objetivos

O objetivo geral desta dissertação é investigar, para então propor estratégias que apóiem a reestruturação sistematizada de código legado com o intuito de promover a utilização de aspectos.

Como objetivo específico tem-se o estudo dos vários tipos de refatoração presentes na literatura. O estudo das abordagens que apóiam a identificação e modularização de interesses em código OO, afim de, integrar as vantagens destas abordagens para promover o desenvolvimento e a manutenção de sistemas.

Verificar os benefícios que a refatoração traz para reestruturar aplicações OO. E ainda, investigar se estes benefícios são similares para aplicações OA. Contudo, deve verificar se a refatoração é uma técnica aplicável para introduzir aspectos em um sistema.

1.3 Metodologia

Para o desenvolvimento desta dissertação foi adotada como metodologia: (i) Revisão Bibliográfica; (ii) Desenvolvimento da Dissertação; e (iii) Uso e Avaliação dos Resultados. Na Figura 1 é apresentado em detalhes a metodologia utilizada.

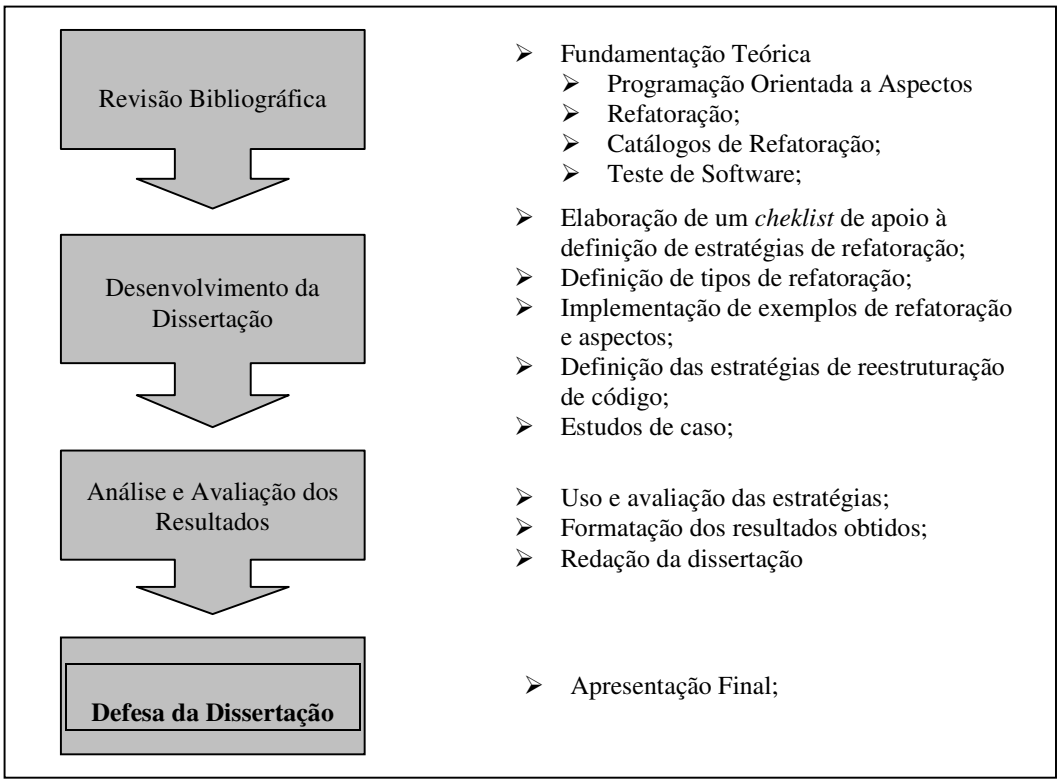

Figura 1. Metodologia de desenvolvimento do trabalho.

1.4 Organização

No Capítulo 2 é apresentada a revisão bibliográfica desta dissertação, onde é descrita a fundamentação teórica para o desenvolvimento deste trabalho. Neste capítulo são apresentados os fundamentos da programação orientada a aspectos, incluindo conceitos, princípios, objetivos e métodos. Em seguida são apresentados conceitos e definições sobre refatoração e alguns de tipos de catálogos publicados na literatura corrente. Além disso, é apresentada uma visão geral sobre teste de software.

Na primeira seção do Capítulo 3, é apresentada uma visão geral das estratégias para reestruturação de código legado. Nas seções seguintes, as estratégias são detalhadas, indicando os passos que podem ser seguidos para uma reestruturação sistemática no código. Os tipos de refatoração, utilizados no ciclo de reestruturação, são ilustrados com exemplos que demonstram o problema estrutural e como este foi solucionado. Ainda nesta seção, é definida uma divisão na refatoração para Aspectos, em dois níveis, sendo eles: (i) Extração de interesses transversais a partir de um código OO; e (ii) Análise da estruturação interna dos aspectos e a forma de tratar a generalização a partir de um código OA.

No Capítulo 4 é apresentado o uso e uma avaliação das estratégias. No primeiro estudo de casos utilizaram-se as estratégias para reestruturar uma classe de cadastro de pessoa, utilizando para isso duas iterações no ciclo. A primeira, para uma reestruturação OO do código e a segunda, para identificar e modularizar um interesse transversal em um aspecto. O segundo estudo de caso, demonstra a utilização do ciclo de reestruturação antes de adicionar uma nova funcionalidade a um código. Ao final do capítulo apresentam-se as considerações finais sobre a utilização das estratégias.

No Capítulo 6 são apresentadas as conclusões deste trabalho, enfatizando-se as suas principais contribuições e apresentando-se propostas de trabalhos futuros em continuidade ao que foi realizado.

2 Conceitos e Terminologias

Neste capítulo é apresentada uma revisão bibliográfica dos temas relevantes para este trabalho. Uma descrição dos principais fundamentos, conceitos e definições de POA e da linguagem AspectJ. Uma visão geral de refatoração e de alguns catálogos já bastante utilizados na academia é apresentada. O teste de software também é abordado sucintamente.

2.1 Programação Orientada a Aspectos

Atualmente, sabe-se que o paradigma OO não contempla todas as necessidades dos desenvolvedores. Dessa forma, considera-se que a POA não veio substituir o paradigma OO, veio para complementá-lo, permitindo que novos requisitos possam ser facilmente agregados ao sistema, e que funcionalidades inerentes a vários objetos sejam facilmente desenvolvidas modularmente. Isso ocorrerá sem mudanças no modelo estático do sistema (modelo de OO é puramente estático), o que comprova que os paradigmas são, de fato, complementares devido à natureza dinâmica do POA.

O paradigma da POA trabalha com o princípio de separação de interesses (do inglês, *separation of concerns*), onde as preocupações envolvidas no desenvolvimento de um software devem ser focadas e trabalhadas separadamente, reduzindo a probabilidade de ocorrência de erros em cada módulo e, conseqüentemente, no sistema como um todo (KULESZA, 2005). Estes conceitos favorecem o reuso e a manutenção do software. É um paradigma criado para a implementação de interesses transversais (do inglês, *crosscuting concerns*), cuja implementação atravessa os componentes responsáveis pela modularização do sistema (RESENDE, 2005).

A atividade de separação de interesses refere-se à habilidade de identificar, encapsular, e manipular somente as partes do software que são relevantes a um interesse particular. Um interesse representa a motivação preliminar para um software organizado e decomposto em partes tratáveis e compreensíveis. Muitos tipos diferentes de interesses podem ser relevantes a vários desenvolvedores em papéis distintos, ou em estágios diferentes do ciclo de vida do software. A separação dos interesses envolve a decomposição do software de acordo com uma ou mais dimensão do interesse. Conseguindo uma clara

separação do interesse garante-se a redução da complexidade do software e melhora-se a sua compreensão; promove-se o rastreamento dentro e através dos artefatos durante todo o ciclo de vida do software; limita-se o impacto da mudança, facilitando a evolução, adaptação e customização; facilita-se o reuso; e simplifica-se a integração de componentes (OSSHER, 2001).

Dessa forma pode-se dizer que na POA, os interesses transversais são encapsulados em unidades denominados aspectos, nas quais são especificados os pontos de execução nos componentes que o interesse transversal afetará e os comportamentos que devem ser adicionados nesses pontos. O sistema final é formado a partir da combinação de aspectos e componentes, em um processo denominado combinação (do inglês, *weaver*), que pode ser conduzido de forma estática (sobre o código fonte ou objeto) ou dinamica (durante a execução do programa).

Em A da Figura 2 demonstra-se a identificação de interesses transversais nos módulos de um sistema (KICZALES *et al.*, 1997). O processo 1 representa a separação dos interesses transversais, que consiste em refatorar o sistema para aspectos (MONTEIRO, 2005). Tendo-se como resultado os módulos do sistema e os aspectos representados em B da Figura 2. Em 2 tem-se a combinação de aspectos e módulos de forma que os módulos voltam a ter as mesmas funcionalidades do sistema, representada em C.

O código do aspecto pode ser alterado sempre que necessário. Essas alterações devem ocorrer em B, e após serem efetuadas, uma nova combinação deve ser executada de forma que seja gerada uma nova versão do sistema (em C) com os aspectos desejados a partir do código dos componentes e do código dos aspectos a ele fornecidos.

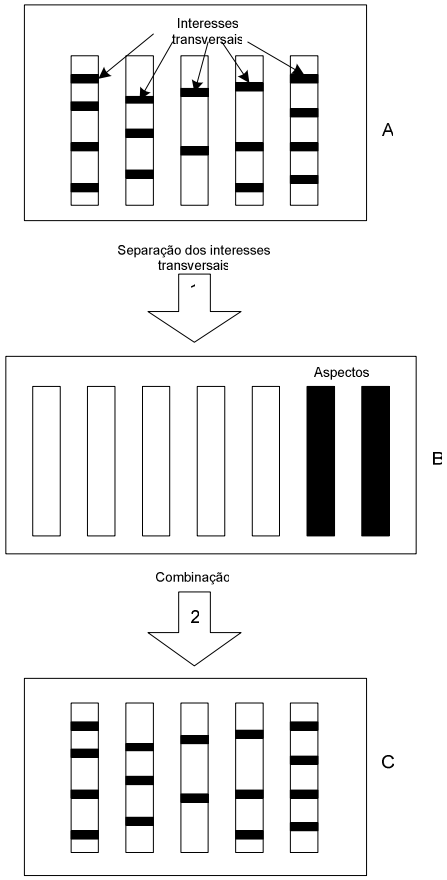

Figura 2. Processo de Identificação, Implementação e Combinação de Aspectos.

2.1.1 Linguagens de suporte a POA

As linguagens de programação e os paradigmas de desenvolvimento de software atuais apóiam várias representações modulares tais como procedimentos e objetos, e também a composição dos módulos em sistemas inteiros. No entanto, as linguagens atuais não fornecem uma abstração correta para a descrição dos aspectos, sendo de grande importância ter linguagens apropriadas para a expressão dos aspectos. Nota-se que as linguagens que apóiam aspecto fazem o código do aspecto mais conciso e mais fácil de compreender (CONSTANTINIDES, 2000). Dessa forma, as linguagens orientadas a aspecto podem ser classificadas em:

- Linguagem de propósito específico: são linguagens que possuem um comportamento restrito dentro de um sistema, como sincronismo, tratamento de exceções, não fornecendo suporte a qualquer outro tipo de componente.

- As linguagens de aspecto de propósito geral: permitem implementar qualquer tipo de aspecto em unidades separadas e capturar locais onde esses aspectos afetam os módulos básicos do programa (KICZALES *et al.*, 2001). Normalmente, essas linguagens são construídas a partir de linguagens de programação existentes, pela adição de novos construtores e operadores.

Se os aspectos forem expressos em linguagens de domínio específico, seria necessária uma linguagem de aspecto para cada tipo de aspecto e uma ferramenta automática de criação implementaria uma ou mais linguagens de aspectos (CONSTANTINIDES, 2000).

Para que os aspectos possam interagir com outros componentes básicos do programa, em uma linguagem de propósito geral, é necessário que o programador possa determinar em quais pontos da execução dos componentes serão definidos comportamentos. Esses pontos são chamados de pontos de junção (do inglês *pointcut*), e são definidos de acordo com determinadas regras (que dependem da linguagem). Após a codificação dos componentes e dos aspectos com pontos de junção determinados, é necessário um processo que os una em um programa executável, e esse processo é chamado de combinação (KICZALES *et al.*, 2001). Esse processo pode ser realizado em tempo de compilação, também chamada de combinação estática, ou em tempo de execução, também chamada de combinação dinâmica, de acordo com a abordagem adotada pela linguagem.

Esclarecendo melhor a combinação, pode-se defini-la como o processo responsável por compor os elementos escritos em uma linguagem de componentes com os elementos escritos em linguagem de aspectos. É um processo que antecede a compilação, gerando um código intermediário na linguagem de componentes capaz de produzir a operação desejada, ou de permitir a sua realização durante a execução do programa.

As classes que se referem aos códigos do negócio nos sistemas não sofrem qualquer alteração para apoiar a programação orientada a aspectos. Isso é feito no momento da composição entre os componentes e os aspectos. Essa composição pode ser (WINCK, 2006):

- Estática: significa modificar o código fonte de uma classe introduzindo indicações de aspectos específicos em pontos de junção. O resultado é altamente otimizado para o

código criado cuja velocidade da execução é comparável àquela do código escrito sem uma Arquitetura de Software Orientada a Aspectos (ASOA) (CONSTANTINIDES, 2000). Dessa forma, um sistema orientado a aspectos utilizando composição estática pode trazer agilidade ao sistema, já que não há necessidade de que aspectos existam em tempo de compilação e execução. O uso de uma combinação estática previne que um nível adicional de abstração cause um impacto negativo na performance do sistema.

- Dinâmica: é indispensável que os aspectos existam tanto em tempo de compilação quanto em tempo de execução. Utilizando uma interface reflexiva, o combinador de aspectos tem a possibilidade de adicionar, adaptar e remover aspectos em tempo de execução.

Linguagens orientadas a aspectos utilizam cinco elementos para permitir a modularização de interesses transversais (ELRAD *et al.*, 2001):

- Um modelo de pontos de junção descrevendo os possíveis pontos em que o comportamento do aspecto pode ser adicionado;
- Um meio de identificar esses pontos de junção;
- Um meio de especificar o comportamento adicional nesses pontos;
- Unidades que encapsulam a especificação dos pontos de junção e as melhorias comportamentais providas pelo aspecto;
- Um mecanismo para combinar o comportamento transversal do aspecto com o comportamento do(s) componente(s) que ele afeta.

Existem inúmeras maneiras de implementar uma linguagem orientada a aspectos. Algumas linguagens utilizam os mecanismos de empacotamento de métodos (*wrapping*), outras criam novas construções para alguma linguagem existente e outras, ainda, fornecem *frameworks* que permitem a criação dos aspectos.

2.1.2 Aspect/J

AspectJ é uma extensão da linguagem Java, de propósito geral, orientada a aspectos, em que a principal unidade modular é o aspecto (KICZALES *et al.*, 2001). O objetivo dessa abordagem era ser um conjunto que permitisse a linguagem de programação, composta por

linguagem central e várias linguagens específicas de domínio, expressar de forma ideal as características transversais do comportamento do sistema.

Essa abordagem recebeu o nome de meta-programação, assim considerada devido ao seu compilador. Primeiramente, o produto final gerado pelos compiladores, destinados à orientação a aspectos, é um novo código e não um programa executável ou interpretável. Com essa primeira compilação são adicionados elementos ao código, então esse código resultante necessita ser novamente compilado para que seja gerado o produto final (WINCK, 2006).

Em uma aplicação orientada a aspectos em AspectJ, os componentes são implementados usando a sintaxe padrão de Java, e os aspectos são implementados usando uma sintaxe específica de AspectJ. Essa linguagem utiliza um processador de aspectos, denominado de combinador (*aspect weaver*), para combinar o código do aspecto com o código dos componentes e, conseqüentemente, produzir o sistema executável. Após a compilação e o processo de combinação, o código objeto gerado pode ser executado em qualquer máquina virtual Java (KICZALES *et al.*, 2001).

Essa linguagem possui mecanismos para implementação de dois tipos de entrecorte: dinâmico e estático. O entrecorte dinâmico permite que, em pontos específicos de execução do programa, seja definida uma implementação adicional ou alterado o comportamento padrão do software. O entrecorte estático, por sua vez, torna possível a adição de novas operações ou atributos em tipos existentes (esse mecanismo também é denominado declaração inter-tipos). A modularização dos interesses transversais é feita por meio de pontos de junção (*join points*) e adendos (*advices*) (KICZALES *et al.*, 2001).

AspectJ possibilita nomear um conjunto de pontos de junção e associar uma determinada implementação a eles, que pode ser executada antes, após ou apropriar-se do fluxo de execução dos eventos relacionados a esses pontos. Essa ferramenta utiliza Java como a linguagem para a implementação dos interesses individuais, e tem construções para a especificação das regras de composição, que são especificadas em termos de pontos de junção, pontos de corte e adendos, tudo encapsulado em um aspecto.

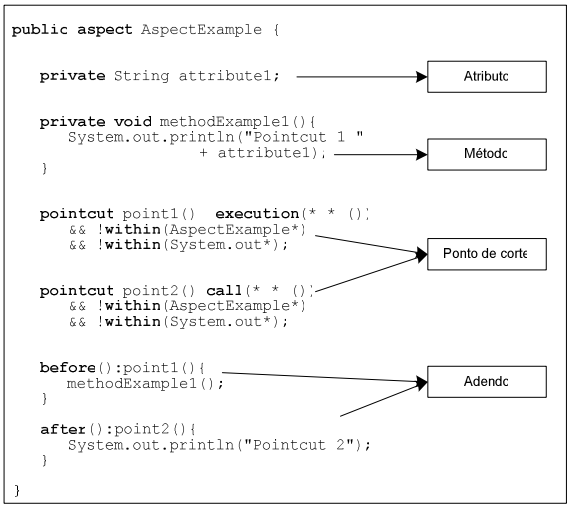

Figura 3. Estrutura de uma entidade de Aspecto em AspectJ.

Na Figura 3 é ilustrado que uma unidade do tipo Aspect pode conter membros de dados e métodos, assim como uma classe, além dos construtores próprios de AspectJ.

Os seguintes conceitos são usados em AspectJ (KICZALES *et al.*, 2001):

- Aspectos: são similares a classes, têm um tipo, podem ser estendidos, podem ser abstratos ou concretos e podem conter campos, métodos e tipos como membros. Além disso, pode conter pontos de corte e adendos como membros e acessar membros de outros tipos. Dentre as diferenças de classes observa-se que não possui construtor nem destrutor; não podem ser criados com o operador *new*;

  Cada aspecto define uma função específica que pode afetar várias partes de um sistema, como, por exemplo, a distribuição. Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos no fluxo de execução, chamados pontos de junção, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução. Normalmente, um aspecto define pontos de corte, os quais selecionam pontos de junção e valores nestes pontos de junção e adendos que definem o comportamento a ser tomado ao alcançar os pontos de junção definidos pelo ponto de corte (SOARES, 2002).

- Pontos de junção: são métodos bem definidos na execução do fluxo do programa que compõem pontos de corte. Segundo Kiczales *et al.* (2001), pontos de junção podem ser

considerados como os nós do grafo de chamada de um objeto em tempo de execução. Nesse grafo, por exemplo, os nós incluem locais onde um objeto recebe uma chamada de método e locais onde um atributo de um objeto é referenciado, enquanto as arestas representam as relações de fluxo de controle entre os nós. Os pontos de junção são classificados em diversos tipos, dependendo do local específico onde eles residem no programa.

Conjuntos de pontos de junção podem identificar um único ponto de junção ou a composição de vários deles, usando operadores lógicos como o 'e', 'ou', além de operadores unários como a negação. Os conjuntos de junção podem ser identificados na própria declaração dos adendos ou serem identificados por um nome e, posteriormente, podem ser referidos por esse mesmo nome na declaração dos adendos.

- Pontos de corte: identificam coleções de pontos de junção e pontos específicos no fluxo de execução do programa. Basicamente, essas coleções definem expressões para construir tais agrupamentos. Além disso, em alguns casos, pontos de corte podem identificar contextos de execução, o que pode ser útil na implementação dos entrecortes.
- Adendos: são construções semelhantes aos métodos, entretanto não podem ser chamados diretamente pela aplicação base e nem pelo próprio aspecto, pois sua execução é feita automaticamente após o entrecorte no ponto de junção. Uma outra diferença em relação aos métodos é que os adendos não possuem nome, não têm especificadores de acesso (*public*, *private*, etc) e tem acesso a variáveis especiais das execuções dos pontos de junção, como assinatura dos métodos entrecortados, etc. AspectJ tem três tipos diferentes de adendos:
  - **Pré-adendo** (*before*): é executada quando um ponto de junção é alcançado e antes da computação ser realizada;
  - **Pós-adendo** (*after*): é executada quando um ponto de junção é alcançado e após a computação ser realizada.
  - **Adendo substitutivo** (*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.

Na Tabela 1, a seguir, encontram-se os tipos de pontos de junção do AspectJ.

Tabela 1. Tipos de pontos de Junção no AspectJ (Adaptado de ROCHA, 2005)

Tipo	Descrição
Chamada de método ou construtor (*call*)	Um método ou um construtor de uma classe é chamado. Pontos de junção deste tipo encontram-se no objeto chamador ou possuem valor nulo (se a chamada é feita a partir de um método estático).
Execução de método ou construtor (*execution*)	Um método ou construtor é chamado. Pontos de junção deste tipo ocorrem no objeto chamado, dentro do contexto do método.
Leitura de atributo (*get*)	Um atributo de um objeto, classe ou interface é lido.
Escrita de atributo (*set*)	Um atributo de um objeto ou classe é escrito.
Execução de tratador de exceção (*handler*)	Um tratador de exceção é invocado.
Iniciação de classe (*staticinitialization*)	Iniciadores estáticos de uma classe (se existirem) são executados.
Inicialização de objeto (*initialization*)	Iniciadores dinâmicos de uma classe são executados durante a criação do objeto, abrangendo desde o retorno da chamada ao construtor da classe pai até o retorno do primeiro construtor chamado.
Pré-iniciação de objeto (*preinitialization*)	Pré-iniciadores de uma classe são executados, abrangendo desde a chamada ao primeiro construtor até a chamada ao construtor da classe pai.
Execução de adendo (*adviceexecution*)	Qualquer parte de um adendo é executada.

2.1.3 Hyper/J

É uma ferramenta baseada na linguagem orientada a objetos Java, aonde cada *hyperslice* representa um fragmento de uma hierarquia de classes. Porém, cada classe contém apenas o subconjunto de métodos e variáveis que pertencem especificamente ao interesse que está sendo modularizado. Portanto, cada *hyperslice* pode ser imaginado como um tipo de visão de classes para um interesse específico.

Em Hyper/J, sistemas são construídos a partir da composição de *hyperslices*. Esta composição envolve encontrar os pontos de junção correspondentes nos diferentes *hyperslices* que estão sendo compostos e combinar cada *hyperslice* nesses pontos de

junção. Na linguagem Hyper/J pontos de junção incluem classes, interfaces, métodos e variáveis.

Antes de especificar a composição entre *hyperslices*, deve-se analisar os relacionamentos entre membros de classes em *hyperlices* distintos. Regras de composição são feitas através de operadores simples e genéricos, tais como *mergeByName* e *override*. O primeiro operador indica que a composição em pontos de junção com os mesmos nomes (e assinaturas) em diferentes *hyperslices* devem ser feitos através de *merging*. O segundo operador indica que a implementação de um método num *hyperslice* sobrepõe a implementação de outro método em outro *hyperslice*. Além desses operadores de composição, outros operadores estão disponíveis tanto para lidar com questões como correspondência entre membros com nomes distintos, como para lidar com ordem de execução e valor de retorno de métodos combinados (OSSHER, 2001).

2.2 **Refatoração**

Refatoração é uma técnica disciplinada para reestruturação de um código existente, alterando sua estrutura interna sem modificar o seu comportamento externo, para remover duplicidades, melhorar a comunicação, simplificar ou adicionar flexibilidade. É composta de uma série de pequenas transformações preservadoras de comportamento. Cada transformação é pequena, porém uma seqüência de transformações pode produzir uma reestruturação significativa. Como cada refatoração é pequena, as chances de fracasso são reduzidas. O sistema é mantido executável depois de cada pequena refatoração, reduzindo as chances de que sejam introduzidos erros graves durante a reestruturação (FOWLER *et al.*, 2000).

Para Kerievsky (2004) o processo de refatoração envolve a remoção de código duplicado, a simplificação de lógica complexa e a clarificação de código não claro. Refatoração é um termo aplicado a sistemas orientados a objetos; para outros paradigmas de programação, esse mesmo processo é chamado de reestruturação (MAIA, 2004).

As refatorações normalmente são descritas com um nome que a descreve; um contexto no qual esta pode ser aplicada; um conjunto de passos bem definidos para sua execução; e um exemplo ou conjunto de exemplos demonstrando como a refatoração ocorre

(FOWLER *et al.*, 2000). Em alguns casos, a descrição de refatorações adiciona figuras para exemplificar a transformação.

Com esta técnica proporcionou-se um desenvolvimento crescente de práticas de desenvolvimento de software orientado a objetos. Como exemplo, pode-se citar que um dos princípios básicos de *eXtreme Programming* (XP) (BECK, 2000) é a realização de refatoração de forma contínua como parte fundamental do processo de desenvolvimento de software. A refatoração é fortemente baseado em testes de unidade, daí a expressão *Test First, Code Later*, dita pelos entusiastas do XP.

Para Fowler *et al.* (2000) existem quatro principais motivos pelos quais os desenvolvedores devem refatorar seus projetos:

- Refatorar ajuda na melhoraria de projeto de software: À medida que o projeto vai crescendo, mais pessoas se envolvem com o projeto e acabam por alterar o código. Assim, ele vai se deteriorando por algumas razões, como alterações para executar objetivos de curto prazo, alterações feitas sem conhecimento total do projeto, entre outras. Desta maneira, o código fica difícil de ser entendido, e a cada nova alteração, mais complexo fica. Através da refatoração contínua, esse problema acaba. As partes que estão no lugar errado são removidas, o tamanho do código fica reduzido, sendo eliminadas as redundâncias de código, tornando assim o projeto mais legível para os desenvolvedores.
- Refatorar torna o software mais fácil de entender: Nos projetos de software, um quesito cobrado pelos gerentes é o cumprimento dos cronogramas. Sendo assim, é importante que um código seja facilmente entendido por qualquer membro da equipe de desenvolvimento a qualquer momento. A refatoração faz exatamente isso, busca a legibilidade e fácil entendimento dos códigos escritos pelos programadores. Sempre haverá alguém que precisará alterar um código em algum momento. Então é preferível gastar um pouco mais de tempo refatorando um trecho de código, do que um desenvolvedor, no futuro, gastar uma semana para efetivar uma alteração que poderia levar apenas uma hora se tivesse entendido o código.
- Refatorar ajuda a encontrar falhas: Estruturando o programa de forma clara, o desenvolvedor tem possibilidade de encontrar falhas mais facilmente.

- Refatorar ajuda a programar mais rapidamente: Com base nos pontos anteriores, chega-se à conclusão que refatorar ajuda a programar mais rapidamente.

No intuito de detectar a utilização da refatoração, Martin Fowler e Kent Beck (2000) criaram o conceito de *bad smells*. Estes são considerados como características do código fonte que indicam má qualidade e que podem ser refatorados. Estas refatorações foram catalogadas para indicar as melhorias que poderiam ocorrer em determinados trechos de código considerados como *bad smell*.

2.2.1 Histórico

O trabalho de Opdyke (1992), trata de refatorações para programas escritos em C++, sendo que foram definidas vinte três refatorações primitivas e três exemplos de refatorações compostas, formadas por duas ou mais primitivas. Para cada refatoração primitiva, o trabalho introduz um conjunto de pré-condições que fornecem a noção de preservação do comportamento do sistema. Assim, se cada refatoração primitiva preserva o comportamento, uma refatoração composta a partir dessas primitivas também preservará.

Para Opdyke (1992) as refatorações primitivas foram classificadas em quatro categorias:

- Criar entidades: criar uma classe vazia, uma variável ou uma função.
- Remover entidades: remover classe, variável ou função não referenciada.
- Mover variável: mover variável para superclasse e mover variável para subclasse.
- Alterar entidade: alterar o nome da classe, alterar o nome da variável, alterar o nome da função, alterar tipo de variáveis e retorno de funções, alterar o modo de controle de acesso, adicionar parâmetro à função, remover parâmetro de função, reordenar parâmetros de função, adicionar corpo de função, remover corpo de função, converter variável para ponteiro, converter referência à variável para chamada de função, substituir lista de declarações por chamada de função, extrair chamada de função e alterar superclasse de uma classe.

As refatorações compostas foram:

- Abstrair acesso a uma variável.

- Converter segmento de código para função.
- Mover classe.

Para preservar o comportamento das funcionalidades do código foram identificadas sete prioridades, também conhecidas como invariantes, que uma refatoração deve obedecer, sendo elas: i) cada classe deve ter uma única superclasse; ii) cada classe deve ter um nome distinto; iii) os membros de uma classe (variáveis e métodos) devem ter nomes distintos; iv) variáveis herdadas não podem ser redefinidas; v) métodos herdados devem ter a mesma assinatura da função original; vi) atribuições de tipo seguras, que consiste em forçar que o tipo da expressão do lado esquerdo de uma atribuição seja igual, ou um subtipo da expressão contida no lado direito; vii) equivalência semântica de operações e referências, isto é, um programa deve produzir a mesma saída para uma dada entrada antes e depois de se aplicar a refatoração (OPDYKE, 1992).

Tokuda (1999) definiu mais quatro propriedades (invariantes) que uma refatoração deve obedecer para preservar o comportamento das funcionalidades, conforme seguem: (i) implementar funções puramente virtuais; (ii) manter objetos agregados; (iii) não provocar efeito colateral de instanciação; (iv) e ser independente de layout ou tamanho. Sua pesquisa também mostrou que todos os tipos de evolução de projeto, isto é, transformações de esquema, micro arquiteturas de padrões de projeto e abordagem dirigida a *hot-spot*, são automatizáveis com refatorações, e propôs, então, uma série de novas refatorações para cada tipo de evolução. Dentre suas conclusões destaca-se que a aplicação de refatorações traz muitos benefícios, entre os quais estão automatizações das mudanças no projeto, redução de testes, e criação de projetos mais simples.

Roberts (1999) desenvolveu uma das primeiras ferramentas de refatoração, o *Refactoring Browser*, para a linguagem Smalltalk. Como resultado de seu trabalho, foi proposto um *framework* para linguagens orientadas a objeto, que permite a criação de ferramentas de refatoração confiáveis e rápidas o suficiente para serem usadas na prática por desenvolvedores. Como contribuição ainda estendeu o conceito de refatoração de Opdyke (1992) acrescentando pós-condições que cada refatoração deve obedecer. Essas pós-condições descrevem o que deve ou não haver depois da aplicação da refatoração e podem ser usadas para derivar pré-condições de refatorações compostas, calcular

dependência entre refatorações e reduzir a quantidade de análises que refatorações posteriores em uma seqüência devem realizar para garantir que preservam o comportamento.

Para implementar as pré e pós-condições, foi definido o conceito de funções de análise, que descrevem o relacionamento entre componentes do programa, isto é, classes, métodos e variáveis. As funções de análise foram divididas em duas categorias: primitivas e derivadas. As primitivas são usadas tanto nas pré como nas pós-condições, enquanto as derivadas são utilizadas apenas nas pré-condições.

Baseado nesse novo conceito, ele redefiniu algumas refatorações propostas por Opdyke, e as dividiu em três grupos (ROBERTS, 1999):

- Refatorações de classe: criar nova classe, renomear e remover uma classe.
- Refatorações de métodos: criar novo método, renomear, remover e mover um método.
- Refatorações de variáveis: criar nova variável, remover e renomear variável, mover uma variável para a superclasse, subclasse ou outra classe qualquer.

Tichelaar *et al.* (2000) implementaram 15 refatorações primitivas independentes de linguagem de programação, definindo um conjunto de pré-condições que podem ser aplicadas a qualquer linguagem. Porém, para algumas refatorações, é necessário acrescentar pré-condições específicas da linguagem para que o comportamento realmente seja preservado. Como exemplo, mostraram pré-condições específicas para Java e Smalltalk, que devem ser verificadas na mesma refatoração.

Fowler *et al.* (2000) focou o processo de refatoração. Ele explica princípios e melhores práticas de refatoração, além de fornecer um guia sobre o processo de refatoração e um extenso catálogo de refatorações. Contudo, ele não define as condições que garantem que as refatorações preservam o comportamento.

Com o crescimento do uso de padrões de projetos, muitos pesquisadores passaram a estudar as oportunidades de refatorações inerentes a cada padrão. Cinnéide (2000) desenvolveu uma metodologia para desenvolvimento de transformações de padrões de projeto baseadas em refatorações. Essa metodologia tem sido aplicada para sete padrões e é implementada através de uma ferramenta para código Java.

Outra metodologia que foi responsável pelo crescimento da visibilidade de aplicar refatoração foi o XP (BECK, 1999), pois uma de suas principais idéias é que o desenvolvedor deve trabalhar em apenas um caso de uso por vez e, assim, deve projetar o software para que fique coerente com o caso de uso em questão. Se um determinado caso de uso não ficou bem projetado, deve-se aplicar refatorações até que o caso de uso possa ser implementado de uma maneira coerente. Ao contrário de tentar evitar mudanças, XP é uma metodologia que é baseada em mudanças. Um dos principais pilares de XP é a contínua aplicação de refatorações. Sem elas, XP não funcionaria.

Em sistemas OA, também existe a necessidade de refatorações que permitam a manipulação de código na presença de aspectos. Mais especificamente, refatorações que tratem de sistemas OA devem possibilitar a movimentação do código implementado em classes para aspectos; manipular código de aspectos para aspectos; e mover código de aspectos para classes. Algumas refatorações foram propostas para possibilitar a manipulação de código na presença de aspectos como o catálogo de Monteiro (2005) e Kiczales (2003).

Kerievsky (2004) traz um catálogo de refatoração para padrões como um processo de melhoria aos problemas recorrentes de projeto. Em seu trabalho sugere que usar padrões para melhorar um projeto existente é melhor do que usar padrões desde o início do projeto. A melhoria ocorre aplicando padrões às seqüências de transformações em nível de código.

2.2.2 Quando Refatorar

A refatoração é utilizada para melhorar os atributos de qualidade do software, como extensibilidade, modularidade e reusabilidade, entre outros. A refatoração também pode ser usada no contexto da reengenharia, para alterar um sistema específico visando reconstruí-lo em um novo formato. Dessa forma, a refatoração provê subsídios para converter código legado em um formato mais estruturado ou modular, ou ainda, para migrar o código para uma diferente linguagem de programação, ou mesmo um diferente paradigma de linguagem (MAIA, 2004).

De acordo com Beck (1999), a refatoração deve ser utilizada quando um código apresenta uma estrutura não recomendada (*bad smells*). Alguns indícios já possuem uma ampla aceitação para promover refatoração (FOWLER *et al.*, 2004), tais como:

- **Código duplicado –** o problema mais simples de código duplicado é quando se tem a mesma expressão em dois métodos da mesma classe, então poderá ser usado o *Extrair Método* e chamar o código de ambos os lugares. Outro problema comum é quando se tem a mesma expressão em duas subclasses irmãs, nesse caso poderá ser utilizado o *Extrair Método* em ambas as classes e então *Subir Método na Hierarquia*. Caso o código duplicado esteja em duas classes não relacionadas, poderá ser usado o *Extrair Classe* em uma classe e, então, usar o novo componente na outra;
- **Método longo** – programas OO melhor desenvolvidos são aqueles que apresentam métodos curtos, fica claro que quanto maior for o método, mas difícil é entendê-lo. A extração de métodos pode ser atrapalhada por métodos com muitos parâmetros e variáveis temporárias. Neste caso, se for utilizada a refatoração *Extrair Método*, o resultado poderá não ser o esperado, pois o método extraído poderá conter tantos parâmetros e variáveis temporárias quanto o método original. Assim, para eliminar as variáveis temporárias pode ser usada a refatoração *Substituir Variável Temporária por Consulta* e *Introduzir Objeto Parâmetro* e *Preservar* o *Objeto Inteiro* para reduzir listas longas de parâmetros. Se o excesso de variáveis temporárias e os parâmetros persistir poderá ser utilizado o *Substituir Método por Objeto Método*;
- **Classe grande** – uma classe com muitas tarefas, frequentemente têm variáveis de instância demais, se isso ocorrer então um outro indício pode estar próximo, o código duplicado. Em alguns casos, uma classe não usa todas as suas variáveis de instância durante todo o tempo, nesse caso, poderá ser usada a refatoração *Extrair Classe* ou *Extrair Subclasse*;
- **Lista de parâmetros longa** – listas de parâmetros longas dificultam o entendimento, porque se tornam inconsistentes e difíceis de usar. Quando for possível obter dados em um parâmetro fazendo uma solicitação a um objeto conhecido poderá ser usada a refatoração *Substituir Parâmetro por Método*, e para obter um conjunto de dados

colhidos de um objeto e substituí-lo pelo próprio objeto poderá ser usada a refatoração *Preservar o Objetivo Inteiro*;

- **Inveja dos dados –** em sua essência os objetos são uma técnica para empacotar dados com os processos usados nesses dados. Quando um método parece ser mais interessado em uma classe diferente daquela na qual ele se encontra o problema é detectado, assim poderá ser usado o *Mover Método*. Quando apenas parte dos métodos estiverem com esse problema usa-se antes a refatoração *Extrair Método*;
- **Classe ociosa –** uma classe que não justifica sua existência deve ser eliminada. Em algumas situações esta classe pode ter sido reduzida por refatoração, e neste momento não é mais necessária, ou ainda pode ser uma classe que for adicionada para promover alguma alteração que por algum motivo foi abortada. Se existir em subclasses que não estejam trabalhando o suficiente, poderá ser usado o *Condensar Hierarquia*. Componentes quase sem uso devem ser submetidos à refatoração *Internalizar Classe*;
- **Generalidade especulativa -** pode ser identificada quando os únicos usuários de um método ou classe forem os casos de teste. Neste caso, tanto o método ou a classe como o caso de teste pode ser excluído. Entretanto, se um método ou classe for auxiliar em um caso de teste que possui uma funcionalidade legítima, deve permanecer. Assim, em ocorrências de Classes abstratas em desuso utiliza-se o *Condensar Hierarquia*, para delegação desnecessária utiliza-se o *Internalizar Classe* e para métodos batizados com nomes estranhos deve-se utilizar o *Renomear Método*;
- **Classes de dados –** são classes que possuem apenas atributos e métodos de acesso para os atributos. Dessa forma, essas classes acabam por se tornarem depósitos de dados e certamente, são manipuladas nos mínimos detalhes por outras classes. Se essas classes possuírem atributos públicos aplica-se *Encapsular Campo*. Caso os atributos sejam do tipo coleção, verificar se eles estão encapsulados apropriadamente e então aplicar o *Encapsular Coleção*, senão usa-se o *Remover Método de Gravação* sobre qualquer campo que não deva ser alterado.

  Deve-se verificar se os métodos de acesso estão sendo usados em outras classes. Há a possibilidade de utilizar o *Mover Método* para remover o comportamento para a classe de dados. Se na for possível mover o método todo, use o *Extrair Método* para criar um

método que possa ser removido. Após algum tempo, pode-se usar o *Ocultar Método* nos métodos de acesso;

- **Campo temporário –** um objeto no qual uma variável de instância recebe um valor apenas em determinadas circunstâncias. Este código fica confuso, pois se espera que um objeto precise de todas suas variáveis. Neste caso, pode ser usado o *Extrair Classe* para criar um local para as variáveis em desuso, todo o código relacionado às variáveis deve ser colocado no componente. Talvez seja possível eliminar o código condicional usando o *Introduzir Objeto Nulo* para criar um componente alternativo para quando as variáveis não forem válidas.

Refatorações também podem ser aplicadas em diferentes níveis de abstração e tipos de artefatos de software. Dessa forma, é possível aplicar refatorações a modelos de projeto, esquemas de banco de dados, requisitos, e arquitetura de software (MENS, 2004). Essa diversidade possibilita ao desenvolvedor realizar mudanças estruturais que não necessariamente se refletem no código fonte e, portanto, introduzem a necessidade de manter todos os artefatos em sincronia. As refatorações podem ser aplicadas em três níveis de abstração (PINTO, 2006):

- Nível de análise - Onde as alterações são consideradas reestruturações, pois não são descritas em termos de código fonte. Estas reestruturações são mudanças na especificação dos requisitos de software.
- Nível de projeto – Onde as refatorações se direcionam para artefatos de projeto de software.

  Padrões de projeto estão cada vez mais sendo usados por desenvolvedores. Eles fornecem uma maneira de escrever o programa em um nível de abstração mais alto. Padrões de projeto criam várias oportunidades de aplicação de refatoração (PINTO, 2006).

- Nível de código - Essa é forma mais comum de se aplicar refatoração a um sistema. Refatorações podem ser aplicadas a diversas entidades do código, e às mais variadas linguagens de programação e paradigmas de linguagem.

  Linguagens não orientadas a objeto são mais difíceis de se reestruturar, pois fluxos de controle e de dados são fortemente interligados e, por causa disso, as reestruturações são limitadas ao nível de função ou bloco de código (PINTO, 2006) apud (LAKHOTIA, 1998). Contudo, linguagens puramente orientadas a objeto apresentam características que tornam algumas refatorações extremamente trabalhosas para serem implementadas, como é o caso das refatorações que lidam com herança, polimorfismo, ligação dinâmica e interfaces. Além disso, quanto mais complexa a linguagem, maior a dificuldade de implementar refatorações.

2.2.3 Catálogos de Refatoração

A maioria das ferramentas de refatoração segue um processo comum, no qual operações de refatoração são aplicadas ao código legado dos programas de acordo com passos bem definidos (MENS, 2004). Em geral, esses passos incluem:

1. Detectar trechos do código com oportunidades de refatorações;
2. Determinar quais refatorações aplicar a cada trecho do código selecionado;
3. Garantir que as refatorações escolhidas preservem o comportamento;
4. Aplicar as refatorações escolhidas aos seus respectivos locais;
5. Verificar que o comportamento do programa foi preservado após as refatorações terem sido aplicadas.

Cada um destes passos encontra-se descritos a seguir:

- **Detecção do Local de Aplicação das Refatorações**:

Antes de refatorar deve-se identificar em que nível de abstração a refatoração se enquadra (nível de análise, nível de projeto ou nível de código). O nível de código é o mais utilizado pelos desenvolvedores e nesse caso, os passos 1 e 2 são geralmente aplicados juntos.

Existem diversas abordagens para detecção de oportunidades de refatoração.Uma abordagem que se pode citar é baseada em meta-programação declarativa, como proposto por Tourwé *et al.* (2002), usada para especificar e detectar formalmente códigos mal escritos e propor refatorações que removam esses trechos de códigos. Carneiro e Neto (CARNEIRO, 2003) relacionam métricas a oportunidades de refatoração por meio de duas abordagens, sendo uma a identificação analítica de métricas adequadas à avaliação de códigos mal escritos, e a outra que utiliza, empiricamente, mensuração de um grande conjunto de métricas para verificar o seu relacionamento com refatorações e códigos mal escritos.

- **Garantia de Preservação do Comportamento:**

O passo 3 consiste em garantir que as refatorações preservarão o comportamento do sistema. A idéia original de preservação do comportamento, introduzida por Opdyke (1992), dizia que, para um mesmo conjunto de dados de entrada, o conjunto de dados de saída deve ser o mesmo, antes e depois da aplicação da refatoração. Com isso, ele propôs o conceito de pré-condições, que são um conjunto de verificações que devem ser verdadeiras para que a refatoração possa ser aplicada. Cada refatoração tem um conjunto de pré-condições.

Outras formas de garantir a preservação do comportamento são: através de um rigoroso conjunto de casos de testes, que deve ser satisfeito após a aplicação da refatoração (PIPKA, 2002); através da redução do escopo da preservação do comportamento, por exemplo, garantir que todas as chamadas a métodos são preservadas após a refatoração (MENS, 2002); através de uma prova formal que a refatoração preserva toda a semântica do programa, o que seria mais fácil de fazer em uma linguagem de programação com uma semântica simples e formalmente definida, como Prolog (PROIETTI, 1991).

- **Aplicação das Refatorações:**

A execução das operações de alteração de código fonte, passo 4, está ligada à forma como o código fonte é representado internamente em cada ferramenta de refatoração. Geralmente, as tais ferramentas possuem um formato próprio para representar o código, seja em forma de Árvore Sintática Abstrata (*Abstract Syntax Tree* – AST) ou *eXtensible Markup Language* (XML).

- **Verificação da Preservação do Comportamento:**

A garantia que o comportamento foi preservado, se dá com a verificação de todas as refatorações realizadas. Dessa forma, aplicam-se um conjunto de pós-condições que devem ser satisfeito após a aplicação da refatoração, como as definidas por Roberts (1999) e Tichelaar *et al.* (2000).

As refatorações são roteiros criados para conduzir o programador em alterações no código que promovam a sua melhoria em algum ponto. Elas são apresentadas conforme uma nomenclatura própria, incentivando a criação de um vocabulário único, dispostas em categorias e, por fim, catalogadas. As categorias denominam-se maus cheiros (*bad smells*). Este termo foi concebido por Kent Beck (2004) para aludir a um trecho de código que necessita de melhorias. Os primeiros catálogos publicados apresentam guias para conservar os conceitos da orientação a objetos no código do sistema.

Posteriormente, outros catálogos foram publicados, ampliando a ação dos catálogos iniciais. As alterações enumeradas dirigiram-se também para a melhoria da qualidade do projeto do software. Novas refatorações foram produzidas para maximizar a flexibilidade e a reutilização do projeto durante o desenvolvimento do sistema.

Através das refatorações, padrões de projeto são adicionados ao código sempre que necessários. A modularidade pode ser aperfeiçoada com a aplicação de refatorações, removendo do código trechos com semânticas similares, cujo comportamento não condiz com as responsabilidades da classe na qual está inserida. Essa última característica é empregada mediante a utilização dos conceitos de orientação a aspectos.

Uma alteração pode ser realizada para remover das classes de negócio os trechos de código que se repetem ao longo das classes do sistema. Dessa forma, tais trechos do código são migrados para aspectos. Os métodos tornam-se enxutos, ao passo que as classes ficam mais reutilizáveis e o projeto adquire maior clareza.

2.2.4 Tipos de Refatoração

Dentre os tipos de refatoração existentes, destacam-se os de maior relevância para este trabalho:

- **Refatorando para Padrões de Projeto:**

O desenvolvimento de software reutilizável e flexível torna-se mais fácil ao recorrer a decisões de sucesso tomadas anteriormente. Gamma *et al.* (2000) publicaram o primeiro catálogo de soluções para problemas recorrentes de projeto. Estas soluções denominam-se padrões de projeto. Este catálogo apresenta padrões de projeto propostos e testados para serem reutilizados em sistemas orientados a objetos. Eles são classificados de acordo com a sua finalidade: (i) criação; (ii) estrutural ; e (iii) comportamental. Assim, os padrões de criação expressam soluções para o processo de construção de objetos. Já os padrões estruturais tratam da composição de classes ou de objetos. Por fim, os padrões comportamentais definem as interações entre as classes e os objetos, bem como a distribuição de suas responsabilidades.

Outros autores criaram padrões de projeto com características diferentes. GRAND (1998) realizou uma compilação de padrões de projeto publicados por outros autores, além de apresentar os padrões de projeto de sua criação. Esta publicação estendeu os catálogos anteriores, enumerando padrões de projeto de características até então não mencionadas. Os padrões de projeto deste catálogo foram classificados em criacionais, estruturais, comportamentais, fundamentais, particionamento e concorrência.

O catálogo de Kerievsky (2004) apresenta vinte e sete exemplos de refatorações para padrões de projeto, agrupadas em doze maus cheiros. Além de demonstrar mais de um caminho para a solução de um mesmo problema, também enumera refatorações para aperfeiçoar o projeto do sistema. Sete maus cheiros enumerados foram criados, enquanto cinco pertencem ao catálogo publicado por Fowler *et al.* (2004). Além disso, os padrões de projeto, alvo das refatorações, foram extraídos do catálogo de padrões de Gama *et al.* (2000). O catálogo de refatorações para introdução de padrões de projeto ao código fornece um direcionamento ao desenvolvedor. Ele expõe um conjunto de refatorações que podem ser realizadas para cada padrão de projeto. Assim, o sistema adquire maior flexibilidade à

medida que é construído, diminuindo a probabilidade de inclusão de falhas durante essas alterações.

Na Tabela 2 apresenta-se a titulo de ilustração alguns exemplos de refatoração para padrões de projeto.

Tabela 2. Exemplos de Refatorações para Padrões de Projeto (KERIEVSKY, 2004)

Refatoração	Problema	Ação Recomendada
Chain Constructors	Construtores com código duplicado.	Encadear os construtores para obter menos código duplicado.
Compose Method	Dificuldade de compreender rapidamente a lógica de um método.	Transformar a lógica em um pequeno número de etapas identificando o objetivo do método com o mesmo nível de detalhe.
Inline Singleton	O código necessita acessar um objeto mas não necessita de um ponto global do acesso a ele.	Mover as características do *Singleton* para uma classe que armazene e forneça o acesso ao objeto. Delete o *Singleton*.
Extract Adapter	Uma classe adapta versões múltiplas de um componente, biblioteca, API ou de outra entidade	Extrair um *Adapter* para uma versão simples de um componente, biblioteca, API ou outra entidade.
Extract Composite	Subclasses em uma hieraquia implementa o mesmo *Composite*.	Extrair uma superclass que implemente o *Composite*.
Form Template Method	Dois métodos em uma subclasse executam etapas similares na mesma ordem, contudo as etapas são diferentes.	Generalizar os métodos extraindo suas etapas em métodos com assinaturas idênticas, a seguir extrair os métodos generalizados para formar um *Template Method*.
Move Accumulation to Visitor	Um método acumula informações de classes heterogêneas.	Mover a tarefa da acumulação para um *Visitor* que possa visitar cada classe para acumular a informação.
Replace Conditional Dispatcher with Command	A lógica condicional é usada para despachar requisições e executar ações.	Criar um *Command* para cada ação. Armazenar os *Commands* em uma coleção e substituir a lógica condicional com o código para buscar e executar *Commands*.
Unify Interfaces	É necessário uma superclasse e/ou uma interface para ter a mesma interface que uma subclasse.	Encontrar todos os métodos públicos nas subclasses que faltam na superclasse/interface. Adicionar as cópias destes métodos na superclasse, alterando o comportamento para nulo.

- **Refatoração para Aspectos:**

Refatoração e POA compartilham do mesmo objetivo que é a construção de sistemas que sejam fáceis de entender e manter sem necessidade de um grande esforço de projeto (LADDAD, 2003). Refatoração Orientado a Aspectos, ou Refatoração para Aspectos,

consiste na reestruturação do código correspondente aos interesses transversais para aumentar a modularização e tentar minimizar os problemas de espalhamento e entrelaçamento de código.

Como visto na seção 2.2.2, Fowler *et al.* (2000) definem diversas refatorações utilizadas em sistemas orientados a objetos. No contexto de sistemas orientados a aspectos, também existe a necessidade de refatorações que permitam a manipulação de código na presença de aspectos. Contudo, estas refatorações devem possibilitar: (i) mover código implementado em classes para aspectos, (ii) manipular código de aspectos para aspectos e (iii) mover código de aspectos para classes.

Piveta *et.al* (2005) adaptaram os *bad smells*, identificados por Fowler e*t al.* (2000), que podem ocorrer em sistemas OO para o contexto de sistemas OA. Esta adaptação segue os seguintes critérios: (i) descrever os problemas que cada *bad smell* acarreta ao estar presente em aspectos, e (ii) propor um conjunto de refatorações (utilizando os catálogos existentes) para auxiliar na remoção destes *bad smells*.

A refatoração para aspectos pode oferecer uma melhoria substancial numa variedade de situações como políticas de tratamento de exceções, controle de concorrência, persistência e distribuição, entre outras funcionalidades já previamente caracterizadas como interesses transversais, em código existente.

Alguns catálogos de refatoração para aspectos também foram propostos para a linguagem AspectJ, a exemplo do catálogo de Monteiro (2005) e o Laddad (2003). O catálogo de refatoração para AspectJ (MONTEIRO, 2005) está dividido em 3 partes:

- Descrição das refatorações para extração dos interesses transversais do código Java para AspectJ. A refatoração deste grupo compreende o ponto de início para a maioria dos processos de refatoração de código legado OO. Como pode ser observado na Tabela 3;

Tabela 3 – Refatoração para Extração de Interesses Transversais (MONTEIRO, 2005)

Refatoração	Problema	Ação Recomendada
Change Abstract Class to Interface	Uma classe abstrata impede que suas subclasses herdem de outras classes.	Transformar uma classe abstrata em uma interface e transformar seu relacionamento com as subclasses de herança para implementação.
Extract Feature into Aspect	Código de uma característica está espalhado por muitos métodos e classes e estão entrelaçados em outras partes do código.	Extrair toda implementação dos elementos relacionados com a característica.
Extract Fragment into Advice	Parte de um método é relacionada a um interesse cujo código esteja sendo movido para um aspecto.	Criar um ponto de corte que captura o ponto de junção e o contexto requerido e mover o fragmento de código para um adendo apropriado.
Extract Inner Class to Standalone	Uma classe privada relaciona-se a um interesse que está sendo extraído de um aspecto.	Eliminar as dependências internas da classe e transformar a classe privada em classe independente.
Inline Class within Aspect	Uma classe independente é usada somente dentro de um aspecto.	Mover a classe para dentro do aspecto
Inline Interface within Aspect	Uma ou diversas interfaces são usadas apenas por um aspecto.	Mover as interfaces para dentro do aspecto.
Move Field from Class to Inter-type	Um campo relaciona-se a um outro interesse que não o interesse primário de sua classe interna.	Mover o campo da classe para o aspecto como uma declaração do inter-tipo.
Move Method from Class to Inter-type	Um método pertence a um outro interesse que não o principal de sua classe proprietária.	Mover o método para um aspecto encapsulando o interesse secundário como uma declaração inter-tipo.
Replace Implements with Declare Parents	Classes implementam interfaces relacionadas a um interesse secundário. A implementação de uma interface é usada somente quando um interesse relacionado está presente no sistema.	Substituir a implementação da classe pela declaração pai no aspecto.
Split Abstract Class into Aspect and Interface	As classes são impedidas de usar herança porque já herdam de uma classe abstrata, que define alguns membros concretos.	Mover todos os membros concretos de uma classe abstrata para um aspecto. Pode transformar a classe abstrata em uma interface.

- Lista as refatorações usadas para melhorar a estrutura interna dos aspectos criados. Na Tabela 4 é apresentada essa listagem;

Tabela 4 – Refatoração para Estruturação Interna de Aspectos (MONTEIRO, 2005)

Refatoração	Problema	Ação Recomendada
Extend Marker Interface with Signature	Uma interface interna representa um papel usado somente por um aspecto. Chamar um método específico do aspecto para um tipo de implementação não declarada pela interface.	Adicionar uma declaração abstrata de inter-tipo na assinatura especifica da interface.
Generalize Target Type with Marker Interface	Um aspecto se refere aos tipos concretos específicos, impedindo o reúso.	Substituir a referência para o tipo específico com um construtor de interface e fazer os tipos específicos implementarem o marcador de interface.
Introduce Aspect Protection	Visualizar um membro inter-tipo em todos subaspectos de um aspecto, mas não fora do encadeamento de herança do aspecto.	Declarar o membro inter-tipo como publico e colocar uma declaração de erro prevenindo o seu uso fora do encadeamento de herança do aspecto.
Replace Inter-type Field with Aspect Map	Um aspecto introduz estaticamente o estado adicional a um conjunto de classes, quando o desejável seria uma ligação mais dinâmica ou flexível.	Substituir as declarações inter-tipo por uma estrutura própria do aspecto que executa um caminho entre o estado e os objetos adicionais.
Replace Inter-type Method with Aspect Method	Um aspecto introduz métodos adicionais a uma classe ou a uma interface, quando uma composição mais dinâmica e flexível seria desejável.	Substituir o método inter-tipo por um método do aspecto que seta o objeto como um parâmetro.
Tidy Up Internal Aspect Structure	A estrutura interna de um aspecto que resulta da extração de um interesse transversal não é muito boa.	Melhorar a estrutura interna de um aspecto removendo a duplicidade e dependências sobre os casos de tipo específicos.

- Tratar a generalização dos aspectos. A terceira parte do catálogo pode ser observada na Tabela 5.

Tabela 5 – Refatoração para Tratar a Generalização (MONTEIRO, 2005)

Refatoração	Problema	Ação Recomendada
Extract Superaspect	Dois ou mais aspectos contêm código e funcionalidade similares.	Mover as características comuns para um superaspecto.
Pull Up Advice	Todos os subaspectos usam o mesmo adendo que age sobre um ponto de corte declarado no superaspecto.	Mover o adendo para o superaspecto.
Pull Up Declare Parents	Todos subaspectos usam a mesma declaração pai.	Mover a declaração pai para o superaspecto.
Pull Up Inter-type Declaration	Uma declaração inter tipo seria melhor colocada em um superaspecto.	Mover a declaração inter-tipo para o superaspecto.
Pull Up Marker Interface	Todos os subaspectos usam o construtor de interface para modelar o mesmo papel.	Mover o construtor de interfaces para o superaspecto.
Pull Up Pointcut	Todos os subaspectos declaram pontos de corte idênticos.	Mover os pontos de corte para o superaspecto.
Push Down Advice	Uma parte do adendo é usada somente por alguns subaspectos, ou cada subaspecto requer um adendo diferente.	Mover o adendo para o subaspectos que o utilizam.
Push Down Declare Parents	A declaração pai não é relevante para todos os subaspectos.	Mover a declaração pai para o subaspecto onde ele é relevante.
Push Down Inter-type Declaration	Uma declaração inter-tipo seria melhor colocada em um subaspecto.	Mover a declaração inter-tipo para o subaspecto onde ele é relevante.
Push Down Marker Interface	Um marker interface declarou dentro de um superaspecto um papel usado somente por alguns subaspectos.	Mover o marker interface para aqueles subaspectos que o usam.
Push Down Pointcut	Um ponto de corte no superaspect não é usado por alguns subaspects que o herdam.	Mover o ponto de corte para os subaspectos que o utilizam.

- **Refatorando Padrões de projeto para Aspectos**

Uma proposta apresentada por Hannemann e Kiczales (2002), diz que as técnicas atuais de programação não são adequadas para a implementação de padrões de projeto. Dessa forma, realizaram em seu trabalho uma implementação OA para os 23 padrões de projeto. Como resultado observou-se benefícios em 17 dos 23 padrões testados, caracterizados como: (i) melhor localidade de código; (ii) maior reusabilidade; (iii) facilidade de composição e de conectividade entre classes do padrão e da aplicação. Dessa forma, podem-se utilizar essas refatorações candidatas para migrar de padrões para padrões implementado com aspectos, nos casos em que essas melhorias sejam relevantes.

A titulo de ilustração apresenta-se alguns exemplos de outros catálogos de refatoração para aspectos que podem ser vistos na Tabela 6.

Tabela 6 – Exemplos de Catálogos de refatoração.

Refatoração	Solução	Fonte
Pull Up Advice	Move um adendo para uma das super-classes ou super-aspectos do aspecto atual.	(GARCIA *et al.*, 2004)
Pull Up Pointcut	Move um conjunto de junção para uma das super-classes ou super-aspectos do aspecto atual.	
Pull Up Inter-Type Declaration	Move uma declaração inter-tipos para uma das super-classes ou super-aspectos do aspecto atual.	
Collapse Aspect Hierarchy	Une uma hierarquia de aspectos.	
Extract Introduction	Extrai a característica externa da classe e adiciona a um aspecto.	(Hanenberg, 2003)
Combine Pointcut	Mescla os predicados de vários conjuntos de junção.	(KICZALES, 2003)
Extract Pointcut	Extrai uma definição de um conjunto de junção de um adendo.	

2.2.5 Considerações sobre refatoração OA

O interesse da academia em minimizar problemas clássicos, presentes no desenvolvimento de sistemas OO, impulsionou as pesquisas no setor da manutenção/reestruração de software, visando à utilização de aspectos. Esta tarefa se relaciona com diversos pontos de projeto e, conseqüentemente, com o código produzido. Dessa forma, vários autores investigaram a possibilidade de desenvolver catálogos de refatoração para aspectos.

A princípio é imprescindível estudar o trabalho de Fowler (2000) que traz a base sobre os conceitos de refatoração, fora do contexto de POA. Neste trabalho a refatoração é definida como um processo de alteração de um sistema de software de modo que o comportamento observável do código não mude, mas que a sua estrutura interna seja melhorada. Fowler (2000) explica princípios e melhores práticas de refatoração, além de fornecer um guia sobre o processo de refatoração e um extenso catálogo de refatorações.

Hanenberg *et al.* (2003) trata a relação entre refatorações orientadas a aspectos e refatorações orientadas a objetos, demonstrando os conflitos encontrados e provendo mecanismos para resolvê-los em AspectJ. Também são introduzidas novas refatorações para auxiliar na migração de software orientado a objetos para código orientado a aspecto,

bem como para reestruturar código de aspectos. Dentre estas se incluem: *Extract Advice*, *Extract Introduction* e *Separate Pointcut*.

Monteiro (2005), apresenta uma coleção de refatorações na forma de um catálogo, que propõe auxiliar na extração de aspectos de código legado OO e em seqüência reorganizar os aspectos resultantes. Ë realizada uma análise dos *bad smells* tradicionais no contexto da OA e propostos alguns novos que sejam específicos para aspectos.

Kiczales (2003) analisa diversas refatorações de Fowler *et al.* (2000), concluindo que poucas delas podem ser usadas em código com aspectos sem modificações. Também define algumas refatorações no contexto de aspectos, incluindo refatorações de extração (*Extract Advice*, *Extract Pointcut*), manipulação (*Pull Up Per Target*, *Move to Introduction*) e inserção (*Introduce Call Pointcut*, *Introduce Around Advice*).

A partir destes trabalhos surgiram também novas abordagens para a detecção de interesses transversais em sistema OO, como o *Aspecting*, apresentado no trabalho de Ramos *et al.* (2004). Está abordagem é caracterizada por três etapas distintas (i) entender a funcionalidade do sistema; (ii) tratar interesses e; (iii) comparar o sistema OA com o OO. Algumas diretrizes foram definidas para auxiliar na descoberta de interesses existentes, por meio de uma lista de indícios, e outras conduzem à etapa de implementação.

Entretanto estes trabalhos não apresentam estratégias para uma reestruturação sistemática do código legado. Nos trabalhos de Monteiro (2005), Kiczales (2003) e Hanenberg *et al.* (2003) a preocupação está voltada à identificação de interesses transversais no código legado e a forma de adequação a abordagem OA. Já no trabalho de Ramos *et al.* (2004) cria-se um conjunto de diretrizes para identificação de interesses transversais em código OO. Na abordagem *Aspecting* os aspectos de um tipo de interesse são implementados a cada iteração, dessa forma, descartam-se as interações e os relacionamentos existentes entre os interesses. Neste ponto, verifica-se a necessidade da utilização de estratégias que orientem na reestruturação do código legado antes e depois da adição de novas funcionalidades, mudança de paradigma e/ou abordagem.

2.3 Teste de Software

Independentemente da utilização de técnicas, critérios e ferramentas para desenvolvimento de software, o teste ainda se apresenta como uma atividade de fundamental importância para a eliminação de erros (MALDONADO, 1991). Por essa razão, o teste de software é um elemento crítico para a garantia da qualidade do produto e representa a última revisão de especificação, projeto e codificação (PRESSMAN, 2002). Dessa forma, os testes, se conduzidos de forma sistemática e criteriosa, contribuem para aumentar a confiança de que o software desempenha funções especificadas e evidenciar algumas características mínimas do ponto de vista da qualidade do produto (VINCENZI, 2004).

De acordo com as técnicas e critérios de teste presentes na literatura é possível fornecer uma maneira sistemática e rigorosa para selecionar um subconjunto do domínio de entrada e ainda assim ser eficiente para apresentar os erros existentes, respeitando-se as restrições de tempo e custo associado a um projeto de software (VINCENZI, 2004).

Além da utilização de técnicas e critérios de teste, para facilitar a condução do teste de software, este é, em geral, dividido em várias fases. Devido ao aspecto complementar das técnicas de teste de software, destaca-se que, dependendo da fase do teste e, conseqüentemente, dos tipos de erros que se deseja revelar, critérios de diferentes técnicas devem ser utilizados para assegurar o teste de boa qualidade. Na prática, a aplicação de um critério de teste está fortemente condicionada à sua automatização. O desenvolvimento de ferramentas de teste é de fundamental importância uma vez que o teste de software é muito propenso a erros, além de improdutivo, se aplicado manualmente. Além disso, a existência de ferramentas de teste viabiliza a realização de estudos empíricos e auxilia a condução dos testes de regressão (DOMINGUES, 2002).

Em suma os testes contribuem para aumentar a confiança de que o software funciona de acordo com o esperado, de modo que grande parte dos defeitos já foi detectada (Beizer, 1995).

2.3.1 Teste de Software OO

A atividade de teste de software é um ponto essencial para a garantia da qualidade de software e representa a ultima revisão de especificação, projeto e codificação. Dentre os

objetivos de teste destaca-se a intenção de encontrar um erro que ainda não foi descoberto (PRESSMAN, 1995).

A estratégia tradicional teste de software OO, segue as seguintes fases: teste de unidade, teste de integração e finalmente o teste do sistema.

- Unidade - Uma unidade é um componente de software que não pode ser subdividido (*IEEE Standards Board*, 1990). Para software OO, o conceito de unidades sofre algumas modificações, ao invés de testar um módulo individual, a menor unidade testável é a classe ou objeto encapsulado. Como uma classe pode conter várias operações diferentes, e uma operação particular pode existir como parte de várias classes diferentes, o significado muda repentinamente. Dessa forma, não se testa uma operação isoladamente e sim como parte de uma classe (PRESSMAN, 2002).

  Este teste desempenha papel fundamental nos métodos ágeis (BECK *et al.*, 2001) e atualmente são apoiados por diversas ferramentas como o JUnit (GAMMA and BECK, 2002), que oferece auxílio ao teste de unidade de programas Java.

- Integração - O teste de integração procura encontrar erros associados a interfaces. O objetivo é, a partir dos módulos testados no nível de unidade, construir a estrutura de programa que foi determinada pelo projeto. O teste de integração é classificado em dois tipos: incremental e não incremental (PRESSMAN, 2002).

  Na integração não-incremental todos os módulos são combinados antecipadamente e o programa completo é testado. Na abordagem incremental o software é testado em blocos e as interfaces têm maior probabilidade de serem testadas completamente, o que facilita o isolamento e a correção de erros.

- Sistema - No teste de sistema verifica-se a integração de todos os elementos que compõem o sistema e o seu funcionamento. O teste de sistema consiste de uma série de testes de diferentes tipos cuja finalidade principal é exercitar todo o software. Testes de recuperação, segurança, estresse e desempenho são alguns dos tipos de teste de sistema (PRESSMAN, 2002).

2.3.2 Teste de Software OA

Como a POA acrescenta novas construções e conceitos aos já conhecidos das linguagens de programação tradicionais novas abordagens de teste devem ser adequadas para esse contexto. Dessa forma, alguns autores já possuem trabalhos com propostas de algumas adaptações para a aplicação de abordagens de teste utilizadas no paradigma procedimental.

A quantificação é um dos principais mecanismos que devem ser considerados nas abordagens de teste. Quando aspectos adicionam comportamento por meio dos conjuntos de junção em diversos módulos do programa, a estrutura original dos módulos-base é modificada após a combinação, ou seja, novas interfaces são introduzidas dentro dos módulos (KICZALES and Mezini, 2005). Como os métodos afetados têm sua estrutura interna modificada, uma abordagem de teste adequada deve levar em conta e explicitar essas alterações. Na abordagem estrutural o modelo de fluxo de controle e de dados deve ser adaptado para explicitar os locais em que os adendos serão executados nos métodos. A partir desse modelo, critérios de teste devem ser criados para fazer com que o testador concentre-se nesses pontos, exercitando-os a partir dos casos de teste.

Em POA, as menores unidades a serem testadas devem ser os métodos e os adendos. A classe à qual o método pertence pode ser vista como o *driver* do método, pois sem ela não é possível executá-lo. Além disso, o aspecto somado a um ponto de junção que faça com que o adendo seja executado pode ser vistos como o *driver* do adendo, pois sem eles não é possível executar o adendo (a não ser que haja alguma infra-estrutura especial). Além disso, métodos comuns e intertipo declarados pertencentes a aspectos podem também necessitar de infra-estrutura especial para serem executados isoladamente.

Um aspecto engloba basicamente conjuntos de atributos, métodos, adendos e conjuntos de junção. Dessa forma, considerando um único aspecto já é possível pensar em teste de integração. Métodos de um mesmo aspecto, bem como adendos e métodos de um mesmo aspecto, podem interagir para desempenhar funções específicas, caracterizando uma integração que deve ser testada.

A atividade de teste para POA poderia ser dividida nas seguintes fases (LEMOS *et al.* 2004):

- Teste de Unidade: O teste de cada método e adendo isoladamente, também chamado de teste intra-método ou intra-adendo.

- Teste de Módulo: O teste de uma coleção de unidades dependentes – unidades que interagem por meio de chamadas ou interações com adendos. Essa fase pode ser dividida nos seguintes tipos de teste:
  - Inter-método: Consiste em testar cada método público juntamente com outros métodos da mesma classe chamados direta ou indiretamente (chamadas indiretas são aquelas que ocorrem fora do escopo do próprio método, dentro de um método chamado em qualquer profundidade);
  - Adendo-método: Consiste em testar cada adendo juntamente com outros métodos chamados por ele direta ou indiretamente;
  - Método-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta ou indiretamente (considerando que um adendo pode afetar outro adendo). Nesse tipo de teste não é considerada a integração dos métodos afetados com os outros métodos chamados por eles, nem com métodos chamados pelos adendos;
  - Inter-adendo: Consiste em testar cada adendo juntamente com outros adendos que o afetam direta ou indiretamente;
  - Inter-método-adendo: Consiste em testar cada método público juntamente com os adendos que o afetam direta e indiretamente, e com métodos chamados direta ou indiretamente por ele. Esse tipo de teste inclui os quatro primeiros tipos de teste descritos acima;
  - Intra-classe: Consiste em testar as interações entre os métodos públicos de uma classe quando chamados em diferentes seqüências, considerando ou não a interação com os aspectos;
  - Inter-classe: Consiste em testar as interações entre classes diferentes, considerando ou não a interação dos aspectos.

- Teste de Sistema: A integração de todos os módulos, inclusive com o ambiente de execução, forma um subsistema ou um sistema completo. Para essa fase geralmente é utilizado o teste funcional.

2.3.3 Ferramentas de Teste

A qualidade e produtividade da atividade de teste são dependentes do critério de teste utilizado e da existência de uma ferramenta que o suporte. Sem a existência de uma ferramenta, a aplicação de um critério torna-se uma atividade propensa a erros e limitada a programas muito simples. Ressalta-se então, que ferramentas de teste podem auxiliar na automatização dessa tarefa, permitindo a aplicação prática de critérios de teste, o teste de programas maiores, o apoio a estudos empíricos e a transferência das tecnologias de teste para a indústria.

Além disso as ferramentas de teste fornecem suporte aos testes de regressão. Os casos de teste utilizados durante a atividade de teste podem ser facilmente obtidos para revalidação do software após uma modificação. Com isso, é possível checar se a funcionalidade do software foi alterada, reduzir o custo para gerar os testes de regressão e comparar os resultados obtidos nos testes de regressão com os resultados do teste original (MALDONADO *et al.*, 1998).

O JUnit é um pequeno *framework* de teste de regressão para programas escritos em Java que fornece suporte à criação, execução e avaliação de casos de teste (GAMMA and BECK, 2002). Apesar de testar se uma saída é correta para uma dada função e elemento do domínio, não avalia a cobertura do programa segundo critérios de teste.

2.4 Considerações Finais

Neste capítulo foram relatados os principais conceitos de POA, refatoração e teste de software. Ressaltam-se os principais conceitos desse capítulo.

A orientação a aspectos surge como uma recente abordagem para modularizar sistemas complexos. Como uma extensão dos paradigmas atuais, o paradigma orientado a aspectos tem o objetivo de capturar e modularizar interesses que normalmente ficam espalhados e misturados em várias unidades de software. Dessa maneira, essa mais nova

abordagem pretende melhorar as potencialidades de reusabilidade, manutenibilidade e compreensibilidade dos artefatos, qualidades que ficavam comprometidas devido ao espalhamento e entrelaçamento dos interesses transversais nas especificações.

Refatoração de código, como atividade de manutenção, não é um processo recente. Ela vem sendo realizada há muito tempo pelos desenvolvedores, os quais não a chamavam de nenhum nome específico, e hoje está cada vez mais presente, seja como mecanismo de manutenção de software ou como um artefato para desenvolvimento e evolução de software.

Teste de software no contexto deste trabalho vem garantir a segurança de que a refatoração foi feita corretamente, ou seja, não modificou o comportamento do código, nem inseriu qualquer tipo de erro num código que anteriormente estava funcionalmente corretamente. Dessa forma, neste capítulo foi apresentada uma visão geral de testes de software e seu uso com a POA.

3 Estratégias para Reestruturação de Código Legado Visando a Utilização de Aspectos

A elaboração das estratégias para conduzir a uma reestruturação sistemática de código legado ocorreu com o estudo de alguns tipos de refatoração, dentre eles Refatoração OO, Refatoração para Padrões de Projeto e Refatoração para Aspectos, além dá análise da abordagem orientada a aspectos. A implementação de alguns exemplos forneceu suporte para o entendimento do funcionamento da POA e permitiu a observação de pontos comuns nos processos de refatoração possibilitando assim a elaboração deste trabalho.

A POA juntamente com outras práticas de programação, como a utilização de padrões de projeto, permite o desenvolvimento de sistemas flexíveis, reutilizáveis e modulares. Com o uso de um ciclo contínuo de reestruturação possibilita-se o estabelecimento destas características, de forma segura e disciplinada. Por meio deste ciclo, pode-se reestruturar um código OO, um padrão de projeto ou desacoplar os requisitos transversais de um código OO. Estas ações devem ser realizadas sem modificar as funcionalidades e a simplicidade do código.

Recentemente, refatoração transformou-se em um assunto de interesse crescente devido ao advento de metodologias ágeis, como a Programação eXtrema (XP). Dessa forma, observou-se que alguns investigadores tenderam a considerar o teste de unidade como um pré-requisito fundamental para refatoração. Assim, o procedimento de teste, seja ele de unidade, automatizado ou de regressão, constitui uma das estratégias definidas neste trabalho, para garantir que o código reestruturado sempre mantenha todas as suas funcionalidades.

Além das estratégias apresentadas neste trabalho, é definida uma divisão para refatoração de código OA. Ela é inicialmente caracterizada pela identificação de interesses transversais e, constitui-se na primeira inserção de aspectos no código fonte. Esta fase toma como base a primeira parte da estruturação sugerida no catálogo apresentado por Monteiro (2005), apresentada na seção 2.2.4. No entanto, não se exclui a utilização de outros catálogos, como o apresentado por Kiczales and Zhao (2003), que porventura venham a ser definidos em trabalhos futuros. A segunda e a terceira parte do catálogo de Monteiro (2005) são utilizadas como base para formar o segundo nível de refatoração para aspetos.

Para ilustrar a utilização do ciclo de reestruturação contínua optou-se pela apresentação de alguns exemplos independentes, com o objetivo de melhor representar os diferentes tipos de refatoração no presente trabalho. Observa-se, no entanto, que a utilização do ciclo demonstra o ápice de seu valor quando aplicada sobre um mesmo código fonte. Entretanto, pode ocorrer de não ser necessário/possível aplicar todos os tipos de refatoração listados, neste trabalho, em um mesmo código.

Este capítulo está organizado da seguinte maneira. Na Seção 3.1, apresenta-se um resumo das estratégias, assim como, uma Tabela para preenchimento e controle das reestruturações executadas a cada iteração do ciclo. Na Seção 3.2 as estratégias para reestruturação de código legado são apresentadas em maiores detalhes. Na Seção 3.3 são detalhados exemplos de refatorações baseados nos catálogos apresentados no capítulo 2. Por fim, as considerações finais deste capítulo são apresentadas na Seção 3.4.

3.1 Visão Geral das Estratégias

Esta seção apresenta uma visão geral das estratégias que podem auxiliar no processo de reestruturação de código, de um sistema que foi desenvolvido sob o paradigma OO, preparando-o para utilização de alguns padrões de projetos, bem como alguns aspectos. Dessa forma, percebe-se que é necessária a utilização da refatoração para adequação do código à utilização de novas tecnologias. A análise inicial do sistema deve seguir a seguinte orientação:

A. Obter a documentação do sistema;

B. Descrição do código analisado;

C. Identificar problemas estruturais;

D. Ciclo de reestruturação:

- **a.** Indicar o tipo de refatoração utilizado;
- **b.** Verificar a existência de conjunto de testes; Caso não haja, deverá ser criado; Executar testes;
- **c.** Aplicar refatoração nos locais indicados, utilizando catálogos disponíveis;
- **d.** Reexecução dos conjuntos de testes;

E. A reestruturação para Aspectos:

a. Nível 1 - Extração de interesses transversais a partir de um código OO;

b. Nível 2 – Análise da estruturação interna dos aspectos e a forma de tratar a generalização a partir de um código OA;

c. Em ambos os níveis, devem-se utilizar as etapas do ciclo de reestruturação detalhado no item **D**;

F. Código reestruturado.

Na Tabela 7 são apresentadas as estratégias que serão aplicadas ao código legado para promover a sua reestruturação, auxiliando na adição de novas funcionalidades, bem como, permitindo a readequação do código de acordo com novas abordagens. Para cada estratégia são propostas algumas alternativas de como deve ser direcionado o problema e em seguida as ações que serão tomadas para possibilitar a reestruturação.

Tabela 7. Estratégias aplicadas ao código legado.

Estratégias	Opções	Ações
Obter os Artefatos Disponíveis sobre o Sistema	**Documentação** Funções Interfaces Restrições	Entender as Funcionalidades do Sistema
Identificar Problemas Estruturais	***Bad Smells*** Observar problemas listados em cada catálogo utilizado	Os catálogos disponíveis oferecem sugestões de soluções para os *Bad Smells* apresentados
Iniciar Ciclo de reestruturação	OO x OO OO x Padrões Padrões x Padrões	Indicar qual tipo de refatoração
		Verificar conjunto de Testes
		Refatorar e utilizar, caso possível, um catálogo apropriado
		Reexecutar conjunto de testes
Reestruturar para Aspectos	OO x Aspectos	Extrair interesses Transversais
	Aspectos x Aspectos	Estrutura interna dos aspectos e Tratamento de Generalização

3.2 Estratégias para Reestruturação de Código Legado

Com o objetivo de trazer as vantagens de alternativas de desenvolvimento de software, como a POA, às aplicações e às estruturas OO existentes, muitas atividades de pesquisa têm

sido desenvolvidas. Uma destas atividades é a definição de um conjunto de estratégias para a reestruturação de código legado, que tem como característica principal a utilização de múltiplas refatorações no decorrer do desenvolvimento/manutenção de software, tais como Refatoração OO, Refatoração para Padrões de Projeto e a Refatoração Orientada a Aspectos.

A pesquisa sobre a adição de novas tecnologias a código legado compreende uma atividade que deve ser realizada em conjunto com várias outras abordagens. Neste trabalho vê-se a necessidade de aumentar o espaço de refatoração no desenvolvimento e manutenção de um sistema para tornar o código bem estruturado.

Após sucessivas refatorações e a introdução de aspectos ao sistema OO, deve-se verificar as melhores formas de agrupamento entre esses aspectos, no intuito de promover um bom relacionamento entre eles. Com o objetivo de construir módulos coesos busca-se, ainda, utilizar boas práticas de programação orientada a aspectos, como por exemplo identificar e utilizar, de forma coerente, a combinação por justaposição entre aspectos.

A partir desse ponto algumas estratégias foram definidas para reestruturação de um código legado:

Passo 1: Obter os artefatos disponíveis sobre o sistema. Nesta fase deve-se levantar e especificar todo o sistema, suas funções, interfaces e restrições;

Passo 2: Analisar o código abstraindo claramente as especificações e requisitos do sistema.

Passo 3: Listar problemas estruturais (*bad smells*) encontrados.

Passo 4: Iniciar Ciclo de reestruturação:

Passo 4.1: Identificar os tipos de refatoração apropriados para reestruturação do código e qual se aplica nesta iteração do ciclo. Sugere-se sempre iniciar com uma refatoração OO, para adequação do código às boas práticas de programação, como polimorfismo e reúso. Caso o código seja considerado bem estruturado, pode-se iniciar por outro tipo de refatoração. Por exemplo, a refatoração para padrões ou mesmo para aspectos;

Passo 4.2: Após a identificação da refatoração a ser aplicada nessa fase de reestruturação deve-se verificar se o código apresenta um conjunto de testes. Caso não possua, desenvolver um e executá-lo.

Os testes demonstram, claramente, quais as funcionalidades da classe testada, facilitando o trabalho de mudar a implementação sem alterar o comportamento. Na Figura 4 é apresentado o processo de testes que será utilizado para os conjuntos de refatorações identificadas. Nesta figura representa-se o processo de definição e execução de testes. Destaca-se que a execução de teste ocorre em duas fases distintas do processo, uma antes e outra depois da refatoração.

A garantia da exatidão das refatorações será obtida por meio da reexecução de testes (readaptados quando necessário). Estes vão garantir que a funcionalidade do código não será alterada após a refatoração. Se houver divergência entre os resultados obtidos na comparação de casos de teste, o código refatorado deverá ser revisado. Em caso de sucesso, isto é, caso a funcionalidade originalmente definida pelo código não seja alterada, o código resultante será documentado.

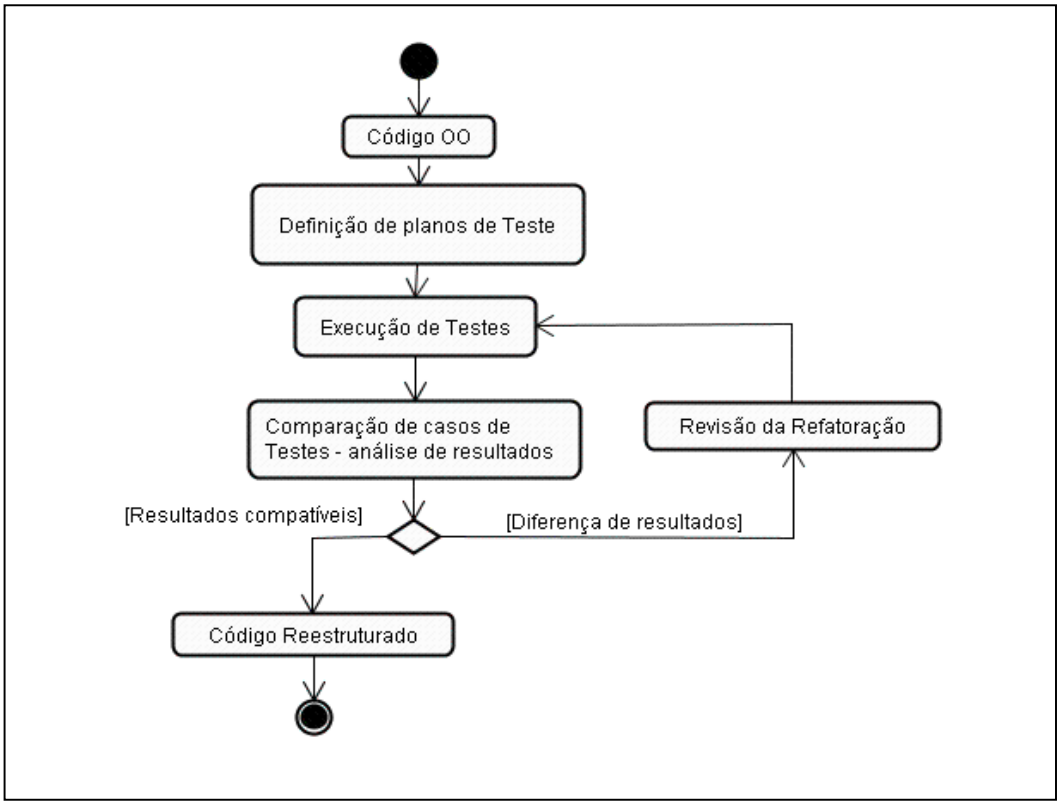

Figura 4. Processo de Testes.

Nesta dissertação, utiliza-se a linguagem Java, para elaboração de exemplos e estudos de casos. Sendo assim, optou-se pela utilização da ferramenta de desenvolvimento Eclipse, que oferece um *plugin* para o JUnit, o qual é padrão para essa IDE Java. Dessa forma, não é preciso instalar, configurar e ativar esse *plugin* para que possa ser utilizado. O Eclipse foi utilizado para auxiliar no processo de construção das classes de teste, para classes já existentes.

O resultado do teste será dado pelo Eclipse por meio de uma *view* relacionada ao *plugin* JUnit. Essa *view* possui uma barra de análise que pode ser preenchida com a cor verde ou vermelha, dependendo do resultado dos testes. Dessa forma utilizou-se neste trabalho a expressão 'barra verde' para os casos de sucesso, e 'barra vermelha' em caso de falhas.

Passo 4.3: Identificar no código os locais passíveis de refatoração, *Bad Smells*, ver seção 2.2.2, como exemplo cita-se: como código duplicado (DUCASSE, 1999); aspectos com poucas responsabilidades (MONTEIRO, 2005); detecção de fraqueza estrutural (Dudzak and Wloka, 2002);

Passo 4.4: Iniciar o Ciclo Contínuo de Reestruturação, apresentado na Figura 5. Nesta figura, está representado inicialmente o processo de testes. A primeira análise questiona quanto à existência de um conjunto de testes (neste trabalho utilizam-se os testes automatizados), caso positivo inicia-se a refatoração. Caso não possua um conjunto de testes este deve ser criado e aplicado como detalhado na Figura 4.

Após a verificação e aplicação dos testes tem-se o processo de reestruturação, iniciando pela análise do código à vista dos princípios do catálogo de refatoração que será utilizado nesta iteração do ciclo (orientada a aspectos (MONTEIRO, 2004), para inserção de padrões de projeto (KERIEVSKY, 2004), orientadas a objetos (FOWLER *et.al* 2004), entre outras).

Eventualmente, novas refatorações também podem ser customizadas de acordo com o conhecimento e experiência do programador. No entanto, devem ser apresentadas dentro do contexto do problema que se pretende a reestruturação, sugere-se então a documentação, utilizando o modelo criado por Fowler (2004) para descrever as novas refatorações adicionadas ao sistema.

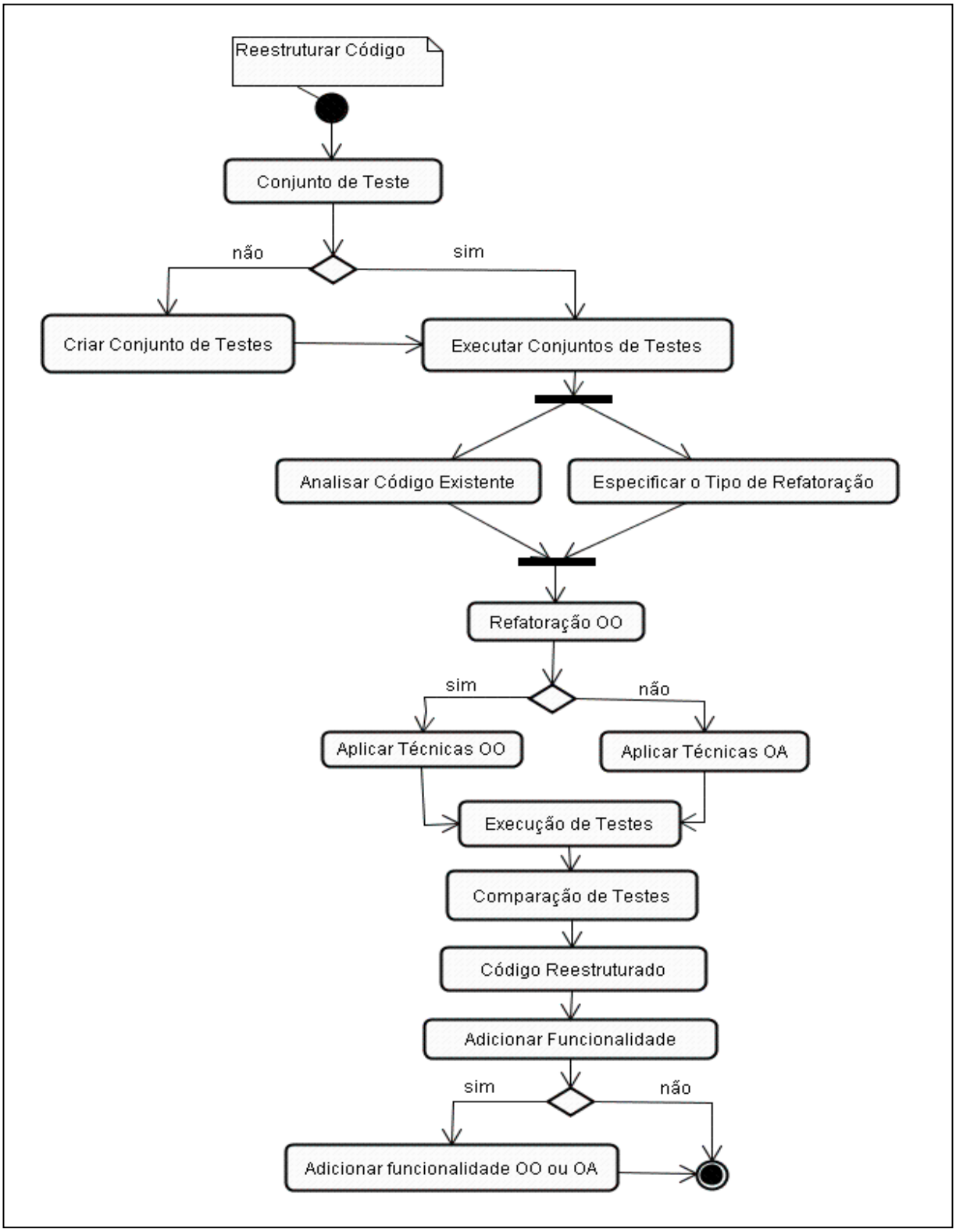

Figura 5. Ciclo Contínuo de Reestruturação.

Passo 5: Sugere-se a realização deste passo apenas na segunda iteração do ciclo de reestruturação, ou seja, após a execução de pelo menos uma refatoração OO.

Esta reestruturação foi nomeada de combinação por justaposição, considerando-a como o agrupamento adjacente em que se encontram dois, ou mais, elementos tomados indistintamente da ordem (Houaiss, 2001).

Com esta reestruturação aprimora-se o conhecimento do código, tornando-o mais legível e simplificando chamadas de métodos. Dentro desse contexto, ainda divide-se a reestruturação com aspectos em dois níveis, baseado no catálogo de Monteiro (2005):

Nível 1: Extrair interesses transversais a partir de um código OO. Este nível corresponde a uma combinação por justaposição entre um objeto e um aspecto, neste caso, deve-se verificar o código fonte à procura de interesses que estejam espalhados e entrelaçados.

Sugere-se tomar como base os três níveis de interesse propostos por Kiczales (1997): (i) Desenvolvimento - interesses inseridos durante o desenvolvimento de aplicações, para facilitar a descoberta de erros, testes e desempenho; (ii) Produção - interesses utilizados para adicionar, em uma aplicação, construções relativas à produção, ou seja, são trechos de código adicionados pelo desenvolvedor para auxiliar na descoberta de possíveis erros relacionados às classes da aplicação. Geralmente, afetam somente um pequeno número de métodos; e (iii) de Tempo de Execução - interesses utilizados para melhorar o desempenho em tempo de execução.

Nível 2: Reestruturação interna dos aspectos, ou seja , tratar código de aspecto para aspecto e manipular a generalização. Este nível corresponde, de forma geral, a combinação por justaposição entre os aspectos. Neste caso, analisaram-se alguns dos *bad smells* adaptados por Piveta *et al.*(2005), não se esgotando todos os existentes, tais como: (i) duplicidade de código – interesses espalhados ao longo das abstrações de uma aplicação podem ser encapsulados em um único aspecto ou em um conjunto pequeno de aspectos; (ii) mudanças divergentes - quando a definição de vários conjuntos de junção é praticamente igual, só mudando os modificadores ou pequenas partes do predicado; (iii) aspectos extensos - quando um aspecto trabalha com mais de um interesse, deve ser dividido em tantos aspectos quantos houver interesses; (iv) aspectos com poucas responsabilidades -

ocorre quando um aspecto tem poucas responsabilidades e sua eliminação poderia proporcionar benefícios na etapa de manutenção. (v) generalidade especulativa - algumas vezes classes e aspectos são criados para tratar requisitos futuro, dessa forma, é possível remover as funcionalidades que não estão sendo usadas.

Passo 5.1: Verificar a existência de casos de teste como no Passo 4.3.

Passo 5.2: Caso possível, utilizar um catálogo de refatoração para aspectos que trata o problema identificado como *bad smell*. Caso não seja possível, proceder como no Passo 4.4, customizando novas refatorações.

Passo 6: **Código reestruturado** - Caso o objetivo dessa reestruturação seja apenas uma mudança de abordagem diz-se que o código foi reestruturado com sucesso e fim do ciclo. A Figura 4, já apresentada acima, demonstra que, para a adição de novas funcionalidades é sempre necessário verificar se o sistema precisa de uma adequação para suportar essa modificação. Após o código reestruturado, diz que está pronto para o acréscimo de novas funcionalidades e fim do ciclo. Sugere-se sempre a criação de testes automatizados para, além de testar as novas funcionalidades, garantir possíveis reestruturações futuras.

O ciclo de reestruturação deve ser executado enquanto problemas estruturais forem encontrados no código. O exemplo da Figura 6 representa o **Passo 5:** Reestruturação com aspectos, em seus dois níveis. A transição A é resultado da extração de interesses transversais de um código OO. Considera-se como passo seguinte a realização de casos de testes para garantir as funcionalidades do sistema.

A transição B, representa uma reestruturação interna dos aspectos, onde um código OA é analisado. Uma refatoração OA deve ser realizada com base nos catálogos disponíveis e também levando em consideração a experiência do desenvolvedor no que diz respeito à solução de problemas estruturais.

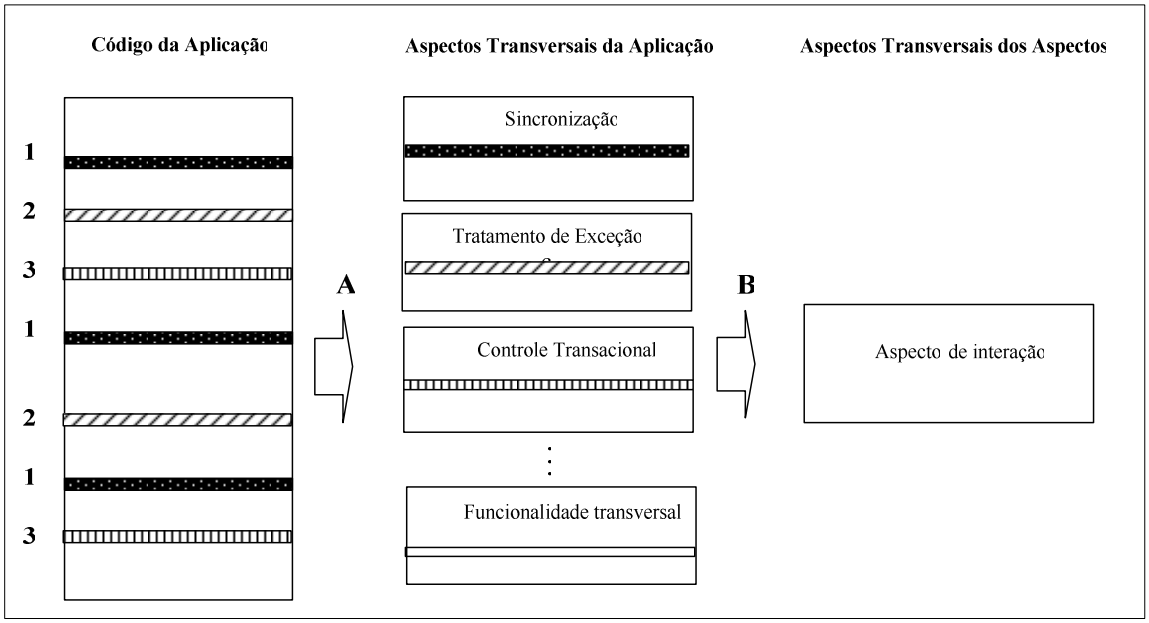

Figura 6. Iterações no Ciclo de reestruturação.

3.3 Exemplificando as Estratégias Definidas

O ciclo de reestruturação possibilita melhorar o código legado de uma aplicação sem modificar o seu comportamento inicial, possibilitando e preparando o código para o acréscimo de novas funcionalidades considerando as boas práticas de programação e, em especial, considerando os recursos de OA.

Nos exemplos descritos nesta seção, demonstra-se a utilização de estratégias para reestruturar um código legado visando à utilização da programação orientada a aspectos. Esta reestruturação sistemática tem por objetivo unir as vantagens proporcionadas pelo uso dos mais variados tipos de refatoração. Dessa forma, boas práticas de programação OO, Padrões de Projeto e refatoração podem ser empregadas em conjunto com a POA para obter projetos mais flexíveis, com códigos legíveis e reutilizáveis.

Assim, sugere-se a utilização de catálogos de refatoração já aceitos pela academia como forma de apoiar a reestruturação de código necessária para solucionar cada *bad smell* identificado nos códigos analisados.

O ciclo de reestruturação contínua, detalhado no Passo 4, prevê a utilização de alguns tipos de refatoração, sendo eles:

3.3.1 Refatoração OO x OO

Esta refatoração deve ser realizada com base no catálogo apresentado por Fowler (2004). Com esse catálogo as refatorações podem ser efetuadas para garantir o encapsulamento das informações, a definição exata das responsabilidades das classes, o emprego de polimorfismo, enfim, possibilita o aperfeiçoamento do código minimizando a ocorrência de falhas.

Como exemplo deste tipo de refatoração considera-se a seguinte iteração no ciclo.

Primeira iteração:

Passo 1: Artefato de entrada – apenas código fonte apresentado na Figura 7.

```
public class Conta {

  private double saldo;
  private int agencia, conta;

  public Conta(int agencia, int conta, double saldo) {
    this.agencia = agencia;
    this.conta = conta;
    this.saldo = saldo;
  }

  public void depositar(double valor) {
    saldo += valor;
  }

  public void sacar(double valor) throws Exception {
    if (valor > saldo)
      throw new Exception("Saldo insuficiente");
    saldo -= valor;
  }

  public void transferir(Conta destino, double valor) throws Exception {
    if (valor > saldo)
      throw new Exception();
    destino.depositar(valor);
    saldo -= valor;
  }

  // Implementação de getters e setters
}
```

Figura 7. Código fonte de entrada – Classe *Conta*.

Passo 2: Descrição do artefato de entrada - representa uma conta em um sistema bancário, contendo basicamente as funcionalidades de saque, depósito e transferência de determinado valor para uma segunda conta;

Passo 3: Problemas estruturais - A classe além de suas funcionalidades básicas (saque e deposito), tem a responsabilidade de efetuar transações entre contas, ou seja, classe com muitas responsabilidades**;**

Passo 4:

Passo 4.1: Refatoração de OO x OO

Passo 4.2: A documentação apresenta um caso de teste para esta classe, como pode ser visto na Figura 8. A classe *TesteConta*, define os testes automatizados para a classe Conta. Os testes estão definidos para o *framework* de testes JUnit.

```
import org.junit.Assert;
import org.junit.Test;

public class TesteConta {

  @Test
  public void testarDepositar() {
    Conta conta = new Conta(121, 12454, 100000.00);
    conta.depositar(145000.00);
    Assert.assertEquals(245000.00, conta.getSaldo());
  }

  @Test
  public void testarSacar() throws Exception {
    Conta conta = new Conta(121, 12454, 245000.00);
    conta.sacar(120000.00);
    Assert.assertEquals(125000.00, conta.getSaldo());
  }

  @Test(expected = Exception.class)
  public void testarSacarComSaldoInsuficiente() throws Exception {
    Conta conta = new Conta(1, 1, 25);
    conta.sacar(50);
  }

  @Test
  public void testarTransferencia() throws Exception {
    Conta conta1 = new Conta(120, 100, 500000.00);
    Conta conta2 = new Conta(2, 245, 100);
    conta1.transferir(conta2, 200000.00);
    Assert.assertEquals(300000.00, conta1.getSaldo());
    Assert.assertEquals(200100.00, conta2.getSaldo());
  }
}
```

Figura 8. Caso de teste para a Classe *Conta*.

Passo 4.3: Utiliza-se a refatoração *Extract Class*, em que a ação realizada é a extração dos métodos que estão além do escopo da classe, conforme pode ser vista na

Figura 9. Os métodos extraídos não são removidos da classe original, são delegados aos novos métodos da classe extraída.

```
/**
 * Classe extraída
 */
public class Transacao {

  public void transferir(Conta origem, Conta destino, double valor)
      throws Exception {
    origem.sacar(valor);
    destino.depositar(valor);
  }
}
```

Figura 9. Classe extraída - Classe *Transação*

Passo 4.4: Não é necessária modificação no caso de teste, sua reexecução demonstra como resultado 'barra verde', mostrando que a refatoração não alterou a funcionalidade da classe

Passo 5: Refatoração OA – não realizada nessa fase do ciclo.

Passo 6 : Código reestruturado – Dessa forma pode-se ver na Figura 10 a Classe Conta refatorada;

```
public class Conta {
  private double saldo;
  private int agencia, conta;

  Transacao transacao;

  public Conta(int agencia, int conta, double saldo) {
    this.agencia = agencia;
    this.conta = conta;
    this.saldo = saldo;

    transacao = new Transacao();
  }

  public void depositar(double valor) {
    saldo += valor;
  }

  public void sacar(double valor) throws Exception {
    if (valor > saldo)
      throw new Exception("Saldo insuficiente");
    saldo -= valor;
  }

  /**
   * Método refatorado
   */
  public void transferir(Conta destino, double valor) throws Exception {
    transacao.transferir(this, destino, valor);
  }

  // Implementação de getters e setters
}
```

Figura 10. Classe *Conta* Refatorada.

Na Tabela 8 é apresentado um resumo das estratégias que foram aplicadas ao código da Classe Conta. Para cada estratégia é descrita a situação em que o código se encontra, bem como a forma que deve ser direcionado para solução do problema, por meio de ações que serão tomadas para possibilitar à reestruturação.

Tabela 8. Resumo das estratégias aplicadas a Classe *Conta*.

Estratégias	Opções	Ações Executadas
Obter os Artefatos Disponíveis sobre o Sistema	**Documentação** Apenas código fonte	Análise dó código, execução passo a passo, analisar caso de teste
Identificar Problemas Estruturais	***Bad Smells*** Classe com muitas responsabilidades	Apenas um problema estrutural encontrado. Identificar refatoração apropriada
Iniciar Ciclo de reestruturação – Primeira iteração	✓ OO x OO OO x Padrões Padrões x Padrões	OO x OO Existe conjunto de testes. Executá-lo *Extract Class* Reexecução e comparação do caso de teste – ***barra verde***

3.3.2 Refatoração OO x Padrões

Neste trabalho consideraram-se as refatoração para padrões de projeto propostas por Kerievsky (2004). Neste tipo de refatoração tem-se por objetivo reconhecer formas de reutilizar código OO, catalogados, que satisfaçam o problema em questão de maneira elegante, porém simples.

Como exemplo para este tipo de refatoração, considera-se a seguinte iteração do ciclo:

Segunda Iteração:

Passo 1: Artefato de entrada – apenas código fonte apresentado nas Figuras 11 e 12.

```
import java.util.List;

public abstract class NoXML {

  public static class Elemento extends NoXML {
    public String nome;
    public List<NoXML> atributos;
    public List<NoXML> filhos;
    public Elemento(String nome, List<NoXML> atributos, List<NoXML> filhos) {
      this.nome = nome;
      this.atributos = atributos;
      this.filhos = filhos;
    }
  }

  public static class Atributo extends NoXML {
    public String nome;
    public String valor;
    public Atributo(String nome, String valor) {
      this.nome = nome;
      this.valor = valor;
    }
  }


  public static class Texto extends NoXML {

    public String texto;

    public Texto(String texto) {

      this.texto = texto;

    }

  }
}
```

Figura 11. Código fonte de entrada - Classe *NoXML*.

```
public class ExtraiTextoXML {
  private NoXML noXML;
  private StringBuilder str;

  public ExtraiTextoXML(NoXML noXML) {
    this.noXML = noXML;
    this.str = new StringBuilder();
  }

  public String extairTexto() {
    str.setLength(0);
    analisarRaiz(noXML);
    return str.toString();
  }

  private void analisarRaiz(NoXML no) {
    if (no instanceof Elemento)
      analisarElemento((Elemento)no);
    else if (no instanceof Atributo)
      analisarAtributo((Atributo)no);
    else if (no instanceof Texto)
      analisarTexto((Texto)no);
  }

  private void analisarElemento(Elemento ele) {
    str.append("<").append(ele.nome);
    for (NoXML no : ele.atributos) {
      str.append(" ");
```

Figura 12. Código fonte de entrada - *ExtraiTextoXML*.

Passo 2: Descrição do artefato de entrada – Esta classe extrai um texto XML de uma estrutura em árvore;

Passo 3: Problemas estruturais – Um método acumula informações de classes heterogêneas. O método extrairTexto percorre todos os nós da árvore acumulando informações para a construção do texto XML**;**

Passo 4:

Passo 4.1: Refatoração OO x Padrões

Passo 4.2: A documentação apresenta um caso de teste para esta classe, como pode ser visto na Figura 13;

```
import java.util.ArrayList;

import org.junit.Assert;
import org.junit.Test;

public class ExtraiTextoXMLTest {

  @Test
  public void testarExtrairTexto() {
    ArrayList<NoXML> atributos = new ArrayList<NoXML>();
    atributos.add(new Atributo("atr1", "valor1"));
    atributos.add(new Atributo("atr2", "valor2"));
    ArrayList<NoXML> filhos = new ArrayList<NoXML>();
    filhos.add(new Texto("Texto"));
```

Figura 13. Caso de teste - *ExtraiTextoXMLTest*.

Passo 4.3: Utiliza-se a refatoração *Move Accumulation to Visitor* (KERIEVSKY, 2004), apresentada na Figura 14, onde a tarefa de acumulação é movida para um *Visitor* que pode visitar cada nó acumulando informações.

```
public interface NoXMLVisitor {

  public void visitarElemento(Elemento elemento);
  public void visitarAtributo(Atributo atributo);
```

Figura 14. *Visitor* com tarefa de acumulação

Passo 4.4: Não é necessária modificação no caso de teste, sua reexecução demonstra como resultado 'barra verde', mostrando que a refatoração não alterou a funcionalidade da classe.

Passo 5: Refatoração OA – não realizada nessa fase do ciclo.

Passo 6 : Código reestruturado – Dessa forma pode-se ver nas Figuras 15 e 16 as Classes *NoXML* e *ExtraiTextoXML* ;

```
import java.util.List;

public abstract class NoXML {

  public abstract void aceitarVisitor(NoXMLVisitor visitor);

  public static class Elemento extends NoXML {
    public String nome;
    public List<NoXML> atributos;
    public List<NoXML> filhos;
    public Elemento(String nome, List<NoXML> atributos, List<NoXML> filhos) {
      this.nome = nome;
      this.atributos = atributos;
      this.filhos = filhos;
    }
    @Override
    public void aceitarVisitor(NoXMLVisitor visitor) {
      visitor.visitarElemento(this);
    }
  }

  public static class Atributo extends NoXML {
    public String nome;
    public String valor;
    public Atributo(String nome, String valor) {
      this.nome = nome;
      this.valor = valor;
    }
    @Override
    public void aceitarVisitor(NoXMLVisitor visitor) {
      visitor.visitarAtributo(this);
    }
  }

  public static class Texto extends NoXML {
    public String texto;
    public Texto(String texto) {
      this.texto = texto;
    }
    @Override
    public void aceitarVisitor(NoXMLVisitor visitor) {
      visitor.visitarTexto(this);
    }
  }
}
```

Figura 15. Classe *NoXML* refatorada.

```
public class ExtraiTextoXML implements NoXMLVisitor {
  private NoXML noXML;
  private StringBuilder str;

  public ExtraiTextoXML(NoXML noXML) {
    this.noXML = noXML;
    this.str = new StringBuilder();
  }

  public String extairTexto() {
    str.setLength(0);
    noXML.aceitarVisitor(this);
    return str.toString();
  }

  public void visitarElemento(Elemento elemento) {
    str.append("<").append(elemento.nome);
    for (NoXML no : elemento.atributos) {
      str.append(" ");
      no.aceitarVisitor(this);
    }
    str.append(">");
    for (NoXML no : elemento.filhos)
      no.aceitarVisitor(this);
    str.append("</").append(elemento.nome).append(">");
  }

  public void visitarAtributo(Atributo atributo) {
    str.append(atributo.nome).append("=\"").append(atributo.valor).append("\"");
  }

  public void visitarTexto(Texto texto) {
    str.append(texto.texto);
  }
}
```

Figura 16. Classe *ExtraiTextoXML* refatorada.

Na Tabela 9 é apresentado um resumo das estratégias que foram aplicadas ao código das Classes *NoXML* e *ExtrairTextoXML* . Para cada estratégia é descrita a situação em que o código se encontra, bem como a forma que deve ser direcionado para solução do problema, por meio de ações que serão tomadas para possibilitar à reestruturação.

Tabela 9. Resumo das estratégias aplicadas as Classes *NoXML* e *ExtrairTextoXML*.

Estratégias	Opções	Ações Executadas
Obter os Artefatos Disponíveis sobre o Sistema	**Documentação** Apenas código fonte	Análise dó código, execução passo a passo, analisar caso de teste
Identificar Problemas Estruturais (*Bad Smellls*)	***Bad Smells*** Um método acumula informações de classes heterogêneas	Apenas um problema estrutural encontrado. Identificar refatoração apropriada
Iniciar Ciclo de reestruturação – Segunda iteração	OO x OO ✓ OO x Padrões Padrões x Padrões	OO x Padrões Existe conjunto de testes. Executá-lo *Move Accumulation to Visitor* Reexecução e comparação do caso de teste – ***barra verde***

Outro exemplo de Refatoração para padrões.

Terceira Iteração:

Passo 1: Artefato de entrada – apenas código fonte, apresentado na Figura 17.

```
import java.util.Stack;

public class AnalisadorRPN {

	private String[] fatores;

	public AnalisadorRPN(String expressao) {
		fatores = expressao.split(" ");
	}

	public Integer calcular() {
		Stack<Integer> pilha = new Stack<Integer>();

		for (String fator : fatores) {
			if (fator.equals("+") || fator.equals("*")) {
				pilha.push(fator.equals("+") ? pilha.pop() +
pilha.pop() : pilha.pop() * pilha.pop());
			}
			else if (fator.equals("-") || fator.equals("/")) {
				Integer temp = pilha.pop();
				pilha.push(fator.equals("-") ? pilha.pop() - temp :
pilha.pop() / temp);
			}
			else {
				pilha.push(new Integer(fator));
			}
		}

		return pilha.peek();
	}
}
```

Figura 17 – Código fonte – Classe *AnalisadorRPN*.

Passo 2: **Descrição do artefato de entrada** – Esta classe é um analisador (*Parser*) de expressões do tipo RPN (*Reverse Polish notation*);

Passo 3: Problemas estruturais – Vários métodos de uma classe combinam elementos de uma linguagem implícita**;**

Passo 4:

Passo 4.1: Refatoração OO x Padrões

Passo 4.2: A documentação apresenta um caso de teste para esta classe, como pode ser visto na Figura 18;

```
import org.junit.Assert;
import org.junit.Test;

public class AnalisadorRPNTest {

	@Test
	public void calcularTest() {
		Assert.assertEquals(10, new AnalisadorRPN("2 8 +").calcular());
		Assert.assertEquals(-3, new AnalisadorRPN("5 8 -").calcular());
		Assert.assertEquals(24, new AnalisadorRPN("4 6 *").calcular());
		Assert.assertEquals(5, new AnalisadorRPN("25 5 /").calcular());
		Assert.assertEquals(50, new AnalisadorRPN("5 6 8 9 10 2 / - - +
*").calcular());
	}
}
```

Figura 18 – Caso de teste – Classe *AnalisadorRPNTest*.

Passo 4.3: Utiliza-se a refatoração *Replace Implicit Language with Interpreter* (KERIEVSKY, 2004), apresentada Figura 18, para definir classes para os elementos da linguagem implícita de tal forma que instâncias podem ser combinadas para formar funções interpretáveis.

Passo 4.4: Não é necessária modificação no caso de teste, sua reexecução demonstra como resultado 'barra verde', mostrando que a refatoração não alterou a funcionalidade da classe.

Passo 5: Refatoração OA – não realizada nessa fase do ciclo.

Passo 6 : Código reestruturado – . Dessa forma pode-se observar nas Figuras 19 e 20 a definição de classe para cada linguagem e a classe refatorada, respectivamente;

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Stack;
interface Expressao {
	public void interpretar(Stack<Integer> pilha);
}

class Inteiro implements Expressao {
	private Integer numero;

	public Inteiro(String fator) {
		numero = new Integer(fator);
	}

	public void interpretar(Stack<Integer> pilha) {
		pilha.push(numero);
	}
}

class Soma implements Expressao {
	public void interpretar(Stack<Integer> pilha) {
		pilha.push(pilha.pop() + pilha.pop());
	}
}

class Subtracao implements Expressao {
	public void interpretar(Stack<Integer> pilha) {
		Integer temp = pilha.pop();
		pilha.push(pilha.pop() - temp);
	}
}

class Multiplicacao implements Expressao {
	public void interpretar(Stack<Integer> pilha) {
		pilha.push(pilha.pop() * pilha.pop());
	}
}


class Divisao implements Expressao {
	public void interpretar(Stack<Integer> pilha) {
		Integer temp = pilha.pop();
		pilha.push(pilha.pop() / temp);
	}
}
```

Figura 19 – Definição de classe para cada linguagem.

```
public class AnalisadorRPN {

	private static Map<String, Expressao> mapa = new HashMap<String,
Expressao>();
	static {
		mapa.put("+", new Soma());
		mapa.put("-", new Subtracao());
		mapa.put("*", new Multiplicacao());
		mapa.put("/", new Divisao());
	}

	private ArrayList<Expressao> expressoes = new ArrayList<Expressao>();

	public AnalisadorRPN(String expressao) {
		for (String fator : expressao.split(" ")) {
			expressoes.add(mapa.containsKey(fator) ? mapa.get(fator) :
new Inteiro(fator));
		}
	}

	public Integer calcular() {
		Stack<Integer> pilha = new Stack<Integer>();

		for (Expressao expressao : expressoes)
			expressao.interpretar(pilha);

		return pilha.peek();
	}
}
```

Figura 20 – Classe refatorada – Classe *AnalisadorRPN*.

Na Tabela 10 é apresentado um resumo das estratégias que foram aplicadas ao código da Classe *AnalisadorRPN* . Para cada estratégia é descrita a situação em que o código se encontra, bem como a forma que deve ser direcionado para solução do problema, por meio de ações que serão tomadas para possibilitar à reestruturação.

Tabela 10. Resumo das estratégias aplicadas a Classe *AnalisadorRPN*.

Estratégias	Opções	Ações Executadas
Obter os Artefatos Disponíveis sobre o Sistema	**Documentação** Apenas código fonte	Análise dó código, execução passo a passo, analisar caso de teste
Identificar Problemas Estruturais (*Bad Smellls*)	***Bad Smells*** Vários métodos de uma classe combinam elementos de uma linguagem implícita	Apenas um problema estrutural encontrado. Identificar refatoração apropriada
Iniciar Ciclo de reestruturação – Terceira iteração	OO x OO ✓ OO x Padrões Padrões x Padrões	OO x Padrões Existe conjunto de testes. Executá-lo *Replace Implicit Language with Interpreter* Reexecução e comparação do caso de teste – ***barra verde***

3.3.3 Refatoração para aspectos

A refatoração para aspectos pode ocorrer de duas formas: (i) OO x Aspectos; e (ii) Aspectos x Aspectos. Neste trabalho utilizou-se como base de refatoração para aspectos o catálogo apresentado por Monteiro (2005), que está dividido em três partes. A relação com está dissertação fica da seguinte forma: A primeira parte do catálogo, extração de interesses transversais, diz respeito à primeira forma de refatoração citada acima. A outra forma de refatoração utiliza, a segunda e a terceira parte do referido catálogo, a saber, melhoria da estrutura interna dos aspectos e generalização dos aspectos.

Considerando o conceito de combinação por justaposição, definido na seção 3.2, pode-se entender que este agrupamento entre um objeto e um aspecto é realizado pela combinação dinâmica, ou melhor, quando da ocorrência de um evento em um objeto (ponto de junção) este deve ser capturado pelo ponto de corte, dessa forma o aspecto executará o respectivo adendo. Neste caso, considera-se a primeira forma de refatoração OO x Aspectos, onde são detectados interesses que entrecortam várias funcionalidades do código e então, são transcritos na forma de aspectos.

Para esclarecer, demonstra-se por meio da Figura 21, à combinação por justaposição entre um objeto e um aspecto. Neste exemplo, *PersonDao* representa uma classe de persistência, um componente que não é um aspecto; como aspecto tem-se *AspectLogPersistence*, cujo intuito é realizar *log* somente das classes de persistência. Sua função é capturar a **execução** (1) dos métodos de persistência e realizar o *log* da classe *PersonDao*. Assim, o objeto que possui uma instância da classe *PersonDao* interage com o aspecto *AspectLogPersistence*.

A forma de refatoração de Aspectos x Aspectos, deve ser entendida como a ocorrência de uma combinação por justaposição entre aspectos, que ocorre quando um aspecto captura um evento de outro aspecto. Neste caso, o evento pode ser: (i) a própria execução de um adendo; ou (ii) uma chamada de método ou construtor realizada no corpo de um adendo. A Figura 21, ilustra esta forma de refatoração da seguinte maneira: o aspecto *AspectTime* foi introduzido na aplicação com o intuito de calcular o tempo gasto com *log*. Assim ele captura a **chamada** (2) de método de *log* que é realizada em *AspectLogPersistence*, ocorrendo interação entre os mesmos. O aspecto *AspectTrace* foi

introduzido no exemplo para demonstrar mais uma forma de interação. Neste caso, ele captura a **execução dos adendos** (*Advice*) (3)(4) dos aspectos *AspectLogPersistence* e *AspectTime*, mostrando que qualquer aspecto introduzido na aplicação pode ser capturado pelo aspecto *AspectTrace*s.

Outra solução para calcular o tempo de *log* seria realizando este cálculo no próprio aspecto de *log* (combinação por justaposição), para isso considera-se que a *Application Programming Interface* (API) de *log* não é de terceiros. Dessa forma, o cálculo não incluiria o tempo gasto com a chamada do método.

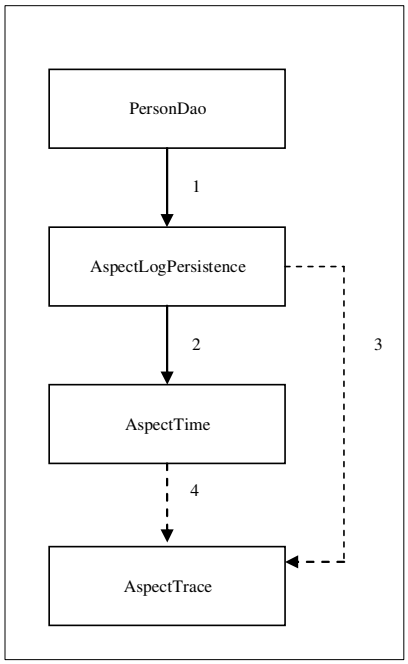

Figura 21 – Combinação por justaposição entre aspectos.

- **Aplicação da Interação entre Aspectos**

Nas Figuras 22 e 23 pode-se observar o código fonte da classe e dos Aspectos que foram utilizados para ilustrar a combinação por justaposição, que representa a forma adequada de utilizar aspectos. Pode-se observar com detalhes: como *AspectLogPersistence* define um ponto de corte *persistenceLog*; como o *AspectTime* define *timeLog*; e como *AspectTrace* define *traceAspect*.

```
package dao;

import entity.Person;

public class PersonDao {
   public void insert(Person person) {
       //Acessa o banco de dados e realiza //a inserção
   }
   public void update(Person person) {
       //Acessa o banco de dados e realiza //a alteração
   }
   public void delete(Integer personId) {
       //Acessa o banco de dados e realiza //a exclusão
   }
   public Person retrieve(Integer personId) {
       //Acessa o banco de dados e //recupera os dados
   }
   public Collection<Person> list() {
       //Acessa o banco de dados e //recupera os dados
   }
```

```
package infra;
import org.aspectj.lang.Signature;
public aspect AspectLogPersistence {

   pointcut persistenceLog():
      execution(* dao.*.*(..));

   before() : persistenceLog() {
      Signature sig =            thisJoinPoint.getStaticPart()
          .getSignature();
      LogUtil.print(sig.getDeclaringType(),
                 sig.getName());
   }
}
```

Figura 22 – Classe *PersonDao* e Aspecto *AspectLogPersistence*.

```
package infra;
public aspect AspectTime {
   private long accumulatedTime = 0;

   pointcut timeLog():
      call(* LogUtil.print(..));

   void around() : timeLog() {
      long timeBefore  =
       System.currentTimeMillis();
      proceed();
      long timeAfter  =
       System.currentTimeMillis();
      long timeSpend = timeAfter -
                timeBefore;
      System.out.println("Tempo gasto: "
       + timeSpend);
      System.out.println("Tempo acumulado: "
       + (accumulatedTime += timeSpend));
   }
}
```

```
package infra;

public aspect AspectTrace {

  pointcut traceAspect(): adviceexecution()
        && within(!AspectTrace)
        && within(Aspect*);

   before():traceAspect(){
      System.out.print("TRACE");
   }
}
```

Figura 23 – Aspectos *AspectTime* e *AspectTrace*.

Nestes casos utilizam-se expressões regulares e alguns tipos de pontos de junção do AspectJ (*call*, *execution*, *adviceexecution*) para capturar os eventos e assim ocorrer interação entre os mesmos. O *adviceexecution* é específico para capturar a execução de adendos (*advice*). Dessa forma, quando for encontrada a definição desse tipo de ponto de junção, será caracterizada uma combinação por justaposição entre aspectos. A combinação por justaposição que ocorre entre os aspectos *AspectLogPersistence* e *AspectTime* é mais sutil, pois na realidade o *AspectTime* captura a chamada de um método, porém esta chamada de método é realizada pelo *AspectLogPersistence*, caracterizando, assim este tipo de combinação. Assim, se ao invés da definição do ponto de corte *timeLog* com a expressão *call(* LogUtil.print(..))* ocorresse *execution(* LogUtil.print(..))*, não seria combinação por justaposição entre aspectos e sim combinação por justaposição entre objeto e aspecto, pois não haveria entrecorte da chamada de método mas sim sua execução.

Dentre os benefícios para criação de um aspecto que calcule o tempo de *log* pode-se observar: (i) o auxílio na abstração, neste caso cada componente do sistema deve fornecer um 'serviço' na aplicação, facilitando sua manutenção e estendendo conceitos de OO para POA, onde cada componente orientado a aspecto deve ter um propósito bem definido; (ii) facilidade de reutilização, por exemplo, pode-se criar um aspecto que calcule o tempo de *log* do *log4j*. Assim qualquer sistema que utilize o *log4j* como API de *log* pode reutilizar esse aspecto para 'monitorar' o tempo de *log* gasto nesta aplicação. Isto pode ser realizado com chamadas de *log* em classes (sem interação) e/ou aspectos (com interação), resultando em uma alta coesão, onde cada aspecto possuiria a sua devida responsabilidade.

3.4 Considerações Finais

Reestruturação de código, como atividade de manutenção, não é um processo recente. Ela vem sendo realizada há muito tempo pelos desenvolvedores, os quais não a chamavam de nenhum nome específico, e hoje está cada vez mais presente, seja como mecanismo de manutenção ou como um artefato para desenvolvimento e evolução de software.

Como não há um roteiro explícito que garanta a migração de um sistema para outras abordagens, ou ainda, que garanta a agregação de novas funcionalidades sem comprometer a estrutura original do sistema, neste capítulo foi apresentado um conjunto de estratégias para auxiliar na reestruturação sistemática de código legado.

Além das estratégias apresentadas, a refatoração de código OA foi dividida em dois níveis. Essa divisão permite que haja uma evolução sistemática para introdução de aspectos no código a ser reestruturado. O primeiro nível trabalha apenas com a identificação e modularização de interesses transversais em aspectos, enquanto, o segundo nível, fica responsável pela reestruturação interna dos interesses já modularizado em aspectos.

4 Uso e Avaliação das Estratégias

No capítulo anterior foram apresentadas, em detalhes, as estratégias para reestruturação de código legado, ilustrado por três exemplos independentes. Nestes exemplos, por questões didáticas, o ciclo de reestruturação foi apresentado em apenas uma iteração, demonstrando apenas um tipo de refatoração por vez. Neste capítulo, é executada mais de uma passagem pelo ciclo de reestruturação para uma melhor visualização/utilização das estratégias definidas.

Para avaliar as estratégias apresentadas no capítulo anterior, serão utilizados dois estudos de caso. O primeiro deles representa parte de um sistema de cadastro desenvolvido na linguagem Java, sem a utilização de padrões de projeto e aspectos. O segundo exemplo, apenas ilustrativo, representa a inserção de *log* para uma nova funcionalidade. Estes exemplos, foram implementados por diferentes pessoas e não possuem documentação além do código fonte.

Este capítulo está organizado da seguinte forma: as seções 4.1 e 4.2 apresentam os dois estudos de caso que utilizam as estratégias descritas no capítulo anterior. A seção 4.3 fornece uma avaliação das estratégias, e por fim as considerações finais são apresentadas.

4.1 Estudo de Caso 1

Este estudo de caso utiliza um fragmento de código que apresenta apenas uma classe para cadastro de pessoa. A primeira análise do código busca entender todas as funcionalidades da classe, para que possa ser identificado à presença de problemas estruturais. Neste caso, foi possível encontrar a presença de um método muito longo que pode ser considerado um *bad smell* dentre os listados no catalogado de Fowler (2000). Com o objetivo de identificar interesses transversais no trecho de código, duas iterações no ciclo de reestruturação já podem ser previamente identificadas. A primeira, como se identificou à presença de um método muito longo, corresponde a uma extração do método de conexão com o banco, utilizando a refatoração OO *Extract Method* (FOWLER, 2000). A segunda iteração no ciclo, representa uma reestruturação do código com a utilização de aspectos em primeiro nível. Como definido neste trabalho tem-se uma combinação por justaposição entre um objeto e um aspecto. O tratamento de exceção é identificado como, interesse transversal, e

será modularizado na forma de um aspecto, por meio da refatoração *Extract Feature into Aspect* (MONTEIRO, 2005).

As estratégias realizadas para essa reestruturação deste estudo de caso são apresentadas a seguir:

Primeira iteração:

Passo 1: Artefato de entrada – código fonte, apresentado na Figura 24;

Passo 2: Descrição do artefato de entrada – classe utilizada para realizar o cadastro de pessoa;

Passo 3: Problemas estruturais – Método muito longo

```
public class CadastraPessoa {

  private Logger logger = Logger.getLogger("app");

  public boolean cadastra(Pessoa pessoa){
    logger.log(Level.INFO, "Entrando no método cadastra(Pessoa pessoa)");
    try {
      Class.forName("com.mysql.jdbc.Driver");

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

    try {
       Connection connection = DriverManager
              .getConnection(
                     "jdbc:mysql://localhost/test?user=root&password=root");
      PreparedStatement ps = connection
        .prepareStatement(
              "INSERT INTO pessoa(NOME, CPF, ENDERECO) VALUES(?, ?, ?)");
      ps.setString(1, pessoa.getNome());
      ps.setString(2, pessoa.getCpf());
      ps.setString(3, pessoa.getEndereco());
      return ps.executeUpdate() > 0;

    } catch (SQLException e) {
      loogger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
      throw new AppException();
    }
    return false;
  }
}
```

Figura 24 – Código Fonte – Classe *CadastraPessoa*.

Passo 4:

Passo 4.1: Refatoração OO x OO

Passo 4.2: A documentação não apresenta um caso de teste para esta classe. Criação de uma classe de teste, utilizando a ferramenta JUnit, que pode ser vista na Figura 25 ;

```
public class CadastraPessoaTest {

  @Test
  public void testCadastra() {
    Pessoa p = new Pessoa("Fulano Test", "00463888825", "Av. Test 2032" );
    CadastraPessoa cadastraPessoa = new CadastraPessoa();
     assertTrue(cadastraPessoa.cadastra(p));
  }

}
```

Figura 25 – Caso de teste criado – Classe *CadastraPessoaTest*.

Passo 4.3: Utiliza-se a refatoração *Extract Method* (FOWLER, 2000), para extrair um método de conexão com o banco que pode ser visto na Figura 26, no trecho em destaque.

```
public class CadastraPessoa {

  private Logger logger = Logger.getLogger("app");

  public Connection getConnection(){
    Connection connection = null;
        String stringConnect =
        "jdbc:mysql://localhost/test?user=root&password=root";
    try {
      Class.forName("com.mysql.jdbc.Driver");

      connection = DriverManager
        .getConnection(stringConnect);

    } catch (ClassNotFoundException e) {
      logger.log(Level.SEVERE, "Classe não encontrada", e);
      throw new AppException();
    }  catch (SQLException e) {
      loogger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
      throw new AppException();
    }

    return connection;

  }

  public boolean cadastra(Pessoa pessoa){
    logger.log(Level.INFO, "Entrando no método cadastra(Pessoa pessoa)");
    try {
      PreparedStatement ps = getConnection()
             .prepareStatement(
             "INSERT INTO pessoa(NOME, CPF, ENDERECO) VALUES(?, ?, ?)");
      ps.setString(1, pessoa.getNome());
      ps.setString(2, pessoa.getCpf());
      ps.setString(3, pessoa.getEndereco());
      return ps.executeUpdate() > 0;

    } catch (SQLException e) {
      logger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
    }

    return false;

  }
}
```

Figura 26 – Classe refatorada – Classe *CadastraPessoa*.

Passo 4.4: O caso de teste deve ser reajustado, adicionado um caso de teste para o novo método extraído, como pode ser visto na Figura 27. A execução do caso de teste ocorreu com sucesso, mostrando como resultado: ‘barra verde’.

```
public class CadastraPessoaTest {
  @Test
  public void testConnection() {
    CadastraPessoa cadastraPessoa = new CadastraPessoa();
    assertNotNull(cadastraPessoa.getConnection());
  }

  @Test
  public void testCadastra() {
    Pessoa p = new Pessoa("Fulano Test", "00463888825", "Av. Test 2032" );
    CadastraPessoa cadastraPessoa = new CadastraPessoa();
     assertTrue(cadastraPessoa.cadastra(p));
  }
}
```

Figura 27 – Reajuste da Classe de teste – Classe *CadastraPessoaTest*.

Passo 5: Refatoração OA – não aplicada nesta iteração.

Passo 6: Código reestruturado

Na Tabela 11 é apresentado um resumo das estratégias que foram aplicadas ao código da Classe *CadastraPessoa*, apresentada neste estudo de caso em sua primeira iteração. Para cada estratégia é descrita a situação em que o código se encontra, bem como o direcionamento dado para a solução do problema, por meio de ações que serão tomadas para possibilitar à reestruturação.

Tabela 11. Resumo das estratégias aplicadas as Classes *CadastraPessoa*.

Estratégias	Opções	Ações
Obter os Artefatos Disponíveis sobre o Sistema	**Artefato disponível** Apenas código fonte	Análise dó código, execução passo a passo
Identificar Problemas Estruturais (*Bad Smellls*)	***Bad Smells*** Método longo demais ou de difícil entendimento	Apenas um problema estrutural encontrado. Identificar refatoração apropriada
Iniciar Ciclo de reestruturação – Segunda iteração	✓ OO x OO OO x Padrões Padrões x Padrões	OO x OO Não existe caso de Teste. Criá-lo e executá-lo *Extract Method* Execução e comparação do caso de teste – ***barra verde***

Segunda iteração:

Passo 1: Artefato de entrada – código fonte – Artefato resultante da primeira iteração do ciclo, já observado na Figura 26. Neste caso, tem-se o caso de teste como documentação;

Passo 2: Descrição do artefato de entrada – classe utilizada para realizar o cadastro de pessoa;

Passo 3: Problemas estruturais – identificação de interesse transversal, caracterizado como não funcional, que pode ser modularizado na forma de um aspecto. Tratamento de exceção *SQLException*.

Passo 4: Já aplicado na primeira iteração do ciclo.

Passo 5: Refatoração OA

Passo 5.1: Refatoração OO x Aspectos – nível 1 – combinação por justaposição entre objeto e aspecto.

Passo 5.2: Utiliza-se o caso de teste criado na iteração anterior, já apresentado na Figura 27. Como sua execução já demonstrou o funcionamento da classe refatorada, na primeira iteração do ciclo, não é necessário aplicar o teste nesta primeira fase da reestruturação do código;

Passo 5.3: Utiliza-se a refatoração *Extract Feature into Aspect* (MONTEIRO, 2005), para extrair um interesse que está espalhado e entrelaçado por vários métodos. Neste caso, sabe-se que o tratamento de exceção ocorrerá em vários outros métodos, por isso esta extração se justifica. Dessa forma, deve-se extrair toda a implementação dos elementos que estão relacionados com o tratamento de exceção, como pode ser observado na Figura 28.

```
public aspect TrataExcecao {
  private Logger logger = Logger.getLogger("app");
  declare soft: SQLException : execution ( * * CadastraPessoa.*(..) );

  after()throwing(SoftException e):call (  * * CadastraPessoa.*(..) ){
    logger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
    throw new AppException();
  }
}
```

Figura 28 – Aspecto *TrataExcecao*.

Passo 5.4: Não é necessário modificar o caso de teste. Após a reexecução do caso de teste tem-se a garantia de que as funcionalidades da classe não foram modificadas. Resultado do teste: 'barra verde'.

Passo 6: Código reestruturado – Na Figura 29 é mostrado no quadro A, a classe refatorada na primeira iteração, destacando o tratamento de exceção realizado nos dois métodos da classe. O quadro B mostra o resultado da segunda iteração com o tratamento de exceção extraído para um aspecto (Figura 28).

```
public class CadastraPessoa {
  private Logger logger = Logger.getLogger("app");
  public Connection getConnection(){
    Connection connection = null;
       String stringConnect = "jdbc:mysql://localhost/test?user=root&password=root";
    try {
      Class.forName("com.mysql.jdbc.Driver");
      connection = DriverManager
        .getConnection(stringConnect);
    } catch (ClassNotFoundException e) {
      logger.log(Level.SEVERE, "Classe não encontrada", e);
      throw new AppException();
    }  catch (SQLException e) {
      loogger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
      throw new AppException();
    }
    return connection;
  }
  public boolean cadastra(Pessoa pessoa){
    logger.log(Level.INFO, "Entrando no método cadastra(Pessoa pessoa)");
    try {
      PreparedStatement ps = getConnection()
               .prepareStatement(
               "INSERT INTO pessoa(NOME, CPF, ENDERECO) VALUES(?, ?, ?)");
      ps.setString(1, pessoa.getNome());
      ps.setString(2, pessoa.getCpf());
      ps.setString(3, pessoa.getEndereco());
      return ps.executeUpdate() > 0;
    } catch (SQLException e) {
      logger.log(Level.SEVERE,
       "Erro em uma operação realizada com o banco de dados.", e );
    }

    return false;

  }

}
```

A

```
public class CadastraPessoa {
  private Logger logger = Logger.getLogger("app");
  public Connection getConnection(){
    Connection connection = null;
    String stringConnect = "jdbc:mysql:///test?user=root&password=root";
    try {
      Class.forName("com.mysql.jdbc.Driver");
      connection = DriverManager.getConnection(stringConnect);
    } catch (ClassNotFoundException e) {
      logger.log(Level.SEVERE, "Classe não encontrada", e);
      throw new AppException();
    }
    return connection;
  }
  public boolean cadastra(Pessoa pessoa){
    PreparedStatement ps = getConnection()
               .prepareStatement(
               "INSERT INTO pessoa(NOME, CPF, ENDERECO) VALUES(?, ?, ?)");
    ps.setString(1, pessoa.getNome());
    ps.setString(2, pessoa.getCpf());
    ps.setString(3, pessoa.getEndereco());
    return ps.executeUpdate() > 0;
  }
}
```

B

Figura 29 – Classe refatorada – Classe *CadatraPessoa*.

Na Tabela 12 é apresentado um resumo das estratégias que foram aplicadas ao código da Classe *CadastraPessoa* em sua segunda iteração. Para cada estratégia é descrita a situação em que o código se encontra, bem como a forma que deve ser direcionado para solução do problema, por meio de ações que serão tomadas para possibilitar à reestruturação.

Tabela 12. Resumo das estratégias aplicadas a Classe *CadastraPessoa* - segunda iteração.

Estratégias	Opções	Ações
Obter os Artefatos Disponíveis sobre o Sistema	**Artefato disponível** Apenas código fonte	Análise dó código, execução passo a passo,
Identificar Problemas Estruturais (*Bad Smellls*)	***Bad Smells*** Método longo demais ou de difícil entendimento	Apenas um problema estrutural encontrado. Identificar refatoração apropriada
Reestruturar para Aspectos	✓ OO x Aspectos Aspectos x Aspectos	OO x Aspectos - Nível 1 Caso de teste da iteração anterior *Extract Feature into Aspect* Execução e comparação do caso de teste – ***barra verde***

4.2 Estudo de Caso 2

O segundo estudo de caso tem por objetivo demonstrar a estratégia de combinação por justaposição entre aspectos, definida na seção 3.3.3. Para isto, foi criado um exemplo ilustrativo, onde o *log* deverá ser introduzido a uma nova funcionalidade da aplicação exemplo, por meio da utilização de aspectos.

Em uma análise inicial deste estudo de caso, deve-se considerar que o código já utiliza aspectos. Para adição desta nova funcionalidade o código deve utilizar a refatoração de Aspectos para Aspectos para evitar a ocorrência de futuros problemas estruturais. Contudo, a primeira análise do código deve avaliar se este apresenta problemas em sua implementação OO. Considerando que o código está limpo, ou melhor, sem problemas estruturais de caráter OO, pode-se então, analisar diretamente a estratégia de refatoração para aspectos.

A partir deste ponto, deve-se avaliar como a nova funcionalidade afetará o código. A reestruturação deve ser caracterizada como de nível 1 ou de nível 2. Neste caso,

identificamos a refatoração de nível 2, pois a introdução dessa nova funcionalidade será aplicada alterando a estrutura interna de um aspecto existente.

Seguem abaixo as estratégias utilizadas neste estudo de caso.

Primeira iteração: para este código não é necessária a refatoração OO, pois não apresentam a ocorrência de problemas estruturais, como os catalogados por Fowler (2000).

Passo 1: Artefato de entrada – código fonte. O código a ser analisado está divido em duas figuras, sendo elas: a Figura 30 que representa as classes *Produto* e *ProdutoDAO* e a Figura 31 que apresenta o aspecto *AspectTimingDAO*;

```
package org.dao;
public class Produto {
	private int id;
	private String nome;
	private double preco;
	private int quantidade;

	public Produto(int id, String nome, double preco, int quantidade) {
		super();
		this.id = id;
		this.nome = nome;
		this.preco = preco;
		this.quantidade = quantidade;
	}
	public int getId() {
		return id;
	}
	public void setId(int id) {
		this.id = id;
	}
	public String getNome() {
		return nome;
	}
	public void setNome(String nome) {
		this.nome = nome;
	}
	public double getPreco() {
		return preco;
	}
	public void setPreco(double preco) {
		this.preco = preco;
	}
	public int getQuantidade() {
		return quantidade;
	}
	public void setQuantidade(int quantidade) {
		this.quantidade = quantidade;
	}
```

```
package org.dao;
import java.util.ArrayList;
import java.util.List;

public class ProdutoDAO {

	public boolean insert(Produto p){ //implementação de uma inserção
		return true;
	}
	public boolean update(Produto p){ //implementação de alteração
		return true;
	}

	public boolean delete(int id){ //implementação da exclusão
		return true;
	}
	public List<Produto> list(){
		return new ArrayList<Produto>();
	}
}
```

Figura 30– Código Fonte – classes *Produto* e *ProdutoDao*.

```
package org.asp;
import java.util.Date;
public aspect AspectTimingDAO {
	pointcut methodsOfDao():
		execution(* *org.dao.ProdutoDAO.* (..));

	before(): methodsOfDao(){
		System.out.println("Iniciando acesso a dados..... " +
				new Date().toString()  + " " +
thisJoinPoint.getSignature().toShortString());
	}

	after(): methodsOfDao(){
		System.out.println("Finalizando acesso a dados... " +
				new Date().toString()  + " " +
thisJoinPoint.getSignature().toShortString());
	}
}
```

Figura 31 – Código fonte – Aspecto *AspectTimingDAO*.

Passo 2: Descrição do artefato de entrada – representa uma classe de persistência da classe *Produto*;

Passo 3: Problemas estruturais – evitar dois aspectos com código e funcionalidades similares. Para evitar este problema deve-se refatorar o código preparando-o para a adição de novas funcionalidades sem incorrer em problemas estruturais.

Passo 4: não será realizada refatoração OO, nenhum problema estrutural encontrado.

Passo 5: Refatoração OA

Passo 5.1: Refatoração Aspectos x Aspectos – Nível 2

Passo 5.2: A documentação não apresenta um caso de teste para esta classe. Criação de uma classe de teste utilizando a ferramenta JUnit, apresentado na Figura 32.

```
package org.test;

import static org.junit.Assert.*;

import org.dao.Produto;
import org.dao.ProdutoDAO;
import org.junit.Before;
import org.junit.Test;

public class ProdutoDAOTest {
	@Before
	public void setUp() throws Exception {
		//carrega os registros necessários ou apaga os registros
necessários
	}

	@Test
	public void testInsert() {
		Produto produto = new Produto(1, "Produto 1", 10.3, 50);
		ProdutoDAO produtoDAO = new ProdutoDAO();
		assertTrue(produtoDAO.insert(produto));
	}

	@Test
	public void testUpdate() {
		Produto produto = new Produto(1, "Produto 1", 10.3, 50);
		ProdutoDAO produtoDAO = new ProdutoDAO();
		assertTrue(produtoDAO.insert(produto));
	}

	@Test
	public void testDelete() {
		ProdutoDAO produtoDAO = new ProdutoDAO();
		assertTrue(produtoDAO.delete( 5 ));
	}

	@Test
	public void testList() {
		ProdutoDAO produtoDAO = new ProdutoDAO();
		assertNotNull( produtoDAO.list() );
	}

}
```

Figura 32 – Case de teste criado – Classe *ProdutoDAOTest*.

Passo 5.3: Utiliza-se a refatoração *Extract Superaspect* (MONTEIRO, 2005), para extrair um superaspecto (*AspectTiming*) a partir do aspecto *AspectTimingDAO*, que pode ser visto na Figura 33. Ápos essa extração o *AspectTimingDAO* é reescrito como um *extends* do *AspectTiming*.

Passo 5.4: Não é necessário modificar o caso de teste. Após a reexecução do caso de teste tem-se a garantia de que as funcionalidades da classe não foram modificadas. Resultado do teste: ‘barra verde’.

```
package org.asp;

public abstract aspect AspectTiming {
	abstract pointcut methods();

	before(): methods(){
		System.out.println(
				getMsgInitMethod() + " " +
thisJoinPoint.getSignature().toShortString());
	}

	after(): methods(){
		System.out.println(
				getMsgEndMethod()  + " " +
thisJoinPoint.getSignature().toShortString());
	}

	public abstract String getMsgInitMethod();
	public abstract String getMsgEndMethod();
}
```

```
package org.asp;

import java.util.Date;

public aspect AspectTimingDAO extends AspectTiming{

	pointcut methods():
		execution(* *org.dao.ProdutoDAO.* (..));

	public String getMsgInitMethod(){
		return "Iniciando acesso a dados..... " + new Date().toString();
	}
	public String getMsgEndMethod(){
		return "Finalizando acesso a dados... " + new Date().toString();
	}


}
```

Figura 33 – Superaspecto extraído *AspectTiming* e *AspectTimingDAO* modificado.

Passo 6: Código reestruturado – código reestruturado e pronto para adição da nova funcionalidade. O carrinho de compras está representado na parte A da Figura 34. A parte B da mesma figura representa o novo aspecto como um *extends* do Aspecto *AspectTiming* com os métodos subscritos.

```
package org.servico;

import java.util.List;

import org.dao.Produto;

public class CarrinhoComprasServico {

	public boolean finalizaCompra(List<Produto> produtos){// implementação
da finalização de uma compra
		return true;
	}
}
```

A

```
package org.asp;

import java.util.Date;

public aspect AspectTimingService extends AspectTiming{

	pointcut methods():
		execution(* *org.servico.CarrinhoComprasServico.* (..));

	public String getMsgInitMethod(){
		return "Iniciando método de negócio..... " + new
Date().toString();
	}
	public String getMsgEndMethod(){
		return "Finalizando método de negócio... " + new
Date().toString();
	}

}
```

B

Figura 34 – Classe *carrinhoComprasServico* e aspecto *AspectTimingService*.

Seguindo os princípios das boas práticas de programação, um novo caso de teste deve ser criado para a nova classe adicionada, assim tem-se uma forma de documentação, além de que será útil em futuras iterações pelo ciclo de reestruturação de código, como pode ser visto na Figura 35.

```
package org.test;

import static org.junit.Assert.*;

import java.util.ArrayList;

import org.junit.Test;
import org.servico.CarrinhoComprasServico;

import org.dao.*;

public class CarrinhoComprasServicoTest {

	@Test
	public void testFinalizaCompra() {
		CarrinhoComprasServico servico = new CarrinhoComprasServico();
		 assertTrue(servico.finalizaCompra(new ArrayList<Produto>()));
	}

}
```

Figura 35 – Teste da classe C*arrinhoComprasServico*.

Na Tabela 13 é apresentado um resumo das estratégias que foram aplicadas a este estudo de caso. Para cada estratégia é descrita a situação em que o código se encontra, bem como o direcionamento que deve ser dado para a solução do problema, por meio de ações que serão tomadas para possibilitar a reestruturação.

Tabela 13. Resumo das estratégias aplicadas a Classe *AspectTimingDAO*.

Estratégias	Opções	Ações
Obter os Artefatos Disponíveis sobre o Sistema	**Artefato disponível** Apenas código fonte	Análise dó código, execução passo a passo,
Identificar Problemas Estruturais (*Bad Smellls*)	***Bad Smells*** Dois aspectos com código e funcionalidades similares	Problema estrutural encontrado, identificar refatoração apropriada
Reestruturar para Aspectos	OO x Aspectos ✓ Aspectos x Aspectos	OO x Aspectos - Nível 1 Caso de teste da iteração anterior *Extract Superaspect* Execução e comparação do caso de teste – ***barra verde***

4.3 Avaliação das Estratégias

Esta seção tem o objetivo de avaliar a aplicação das estratégias para reestruturação de código legado, visando à utilização de aspectos, por meio de dois estudos de casos. Em ambos os casos, a primeira estratégia a ser analisada deve ser com relação ao problema que se pretende resolver: (i) apenas refatoração, (ii) adição de uma nova funcionalidade; ou (iii) mudança de paradigma/abordagem.

O primeiro estudo de caso identifica como resultado da primeira estratégia aplicada apenas a refatoração de código. A utilização do ciclo de reestruturação se dá em duas iterações, demonstrando assim, a evolução do código possibilitada pela utilização destas estratégias. Em se tratando de problemas estruturais com código OO leva-se em consideração o catálogo publicado por Fowler (2000), que já documenta os principais problemas de estruturação de código e sugere as refatorações que permitirão uma solução adequada para o problema. A aplicação de refatorações contínuas garante o encapsulamento das informações, a definição exata das responsabilidades das classes, o emprego do polimorfismo, entre outras. Dessa forma, com a segunda iteração do ciclo foi possível identificar um interesse transversal no código da aplicação e modularizá-lo na forma de um aspecto.

O que se alcança com a utilização destas estratégias é a ampliação no uso conjunto dos vários tipos de catálogos de refatoração disponíveis na literatura. Esta reestruturação sistemática prove um código cada vez mais próximo do desejável, isto é, com um nível reduzido de problemas estruturais. Adicionalmente, o acréscimo de novas funcionalidades, as mudanças de paradigma, e ou, a introdução de novas abordagens se tornam bem mais fáceis de serem aplicadas.

O segundo estudo de caso demonstra a utilização do ciclo de reestruturação para a adição de *log* a uma nova funcionalidade do código.

Este caso, identifica como resultado da primeira estratégia aplicada uma reestruturação para promover a adição de uma nova funcionalidade, assim, por meio da análise do código verifica-se que o mesmo se trata de um código OA e que a adição de *log* a uma nova funcionalidade deverá ser incluída por meio da utilização dos princípios dessa abordagem. A partir de então, identifica-se a utilização da refatoração para aspectos em seu

segundo nível. Caso uma nova funcionalidade seja identificada como um interesse diferente dos já modularizados na aplicação, seria pertinente o uso da refatoração para aspectos de primeiro nível. Entretanto, observa-se que o interesse a ser modularizado já está definido em um aspecto da aplicação.

A utilização das estratégias para promover a reestruturação de código legado, demonstra sua força pela sua sistematização, ou melhor, permite potencializar as características individuais de cada catálogo de refatoração utilizado neste trabalho, como também outros que porventura sejam publicados, agrupando-os em um conjunto ordenado e evolutivo.

4.4 **Considerações finais**

Neste capítulo foram apresentados dois fragmentos de sistemas para permitir a avaliação da utilização das estratégias de reestruturação de código legado, tendo como objetivo a adequação do código para utilização de aspectos. Cada exemplo buscou demonstrar a utilização do ciclo de reestruturação de código em situações distintas para que a avaliação pudesse averiguar a abrangência das estratégias propostas.

5 Conclusão

Nesta dissertação foram apresentadas estratégias para reestruturação de código legado visando à utilização de aspectos. Para isso, foi preciso explorar os conceitos, princípios e objetivos mais importantes sobre programação orientada a aspectos e refatoração, no sentido de obter maior aproveitamento de suas vantagens no desenvolvimento e manutenção de software. A partir do estudo da abordagem orientada a aspectos, constatou-se que ela se propõe a resolver, ou amenizar, os problemas causados pelo espalhamento e entrelaçamento de interesses no código de programas orientado a objetos, por meio da implementação desses na forma de aspectos. Entretanto, a simples implementação de interesses na forma de aspectos não é suficiente para tornar um código legível, de fácil manutenção, propício à adição de novas funcionalidades e ao reuso. Diante destes problemas, apresentou-se a refatoração como um processo auxiliar para a reestruturação de sistemas que visam à utilização de aspectos. Contudo, foi necessário um estudo sobre os catálogos de refatoração já consolidados e disponibilizados na literatura corrente, buscando neles subsídios para o aprimoramento de estratégias que possibilitassem a reestruturação de código de forma sistemática.

O primeiro catálogo de refatorações publicado compõe-se de roteiros para o aperfeiçoamento contínuo do código, seguindo os conceitos da orientação a objetos (FOWLER, 2000). Após a publicação do primeiro catálogo para problemas recorrentes de projeto, a saber, os padrões de projeto (GAMMA *et al.*, 2000) surgiu a necessidade de modificar o projeto do sistema durante o seu desenvolvimento, motivando a criação de um catálogo de refatorações específicas para o contexto de padrões de projeto. Dessa forma, Kerievsky (2004) propôs um catálogo de refatorações para a inclusão de determinado padrão de projeto ao código. Este catálogo pode ser considerado um complemento ao de Fowler (2004), pois expandiu a ação das refatorações existentes. Com o surgimento da orientação a aspectos, o interesse da comunidade em unir suas vantagens a outros mecanismos se tornou crescente. Contudo, as refatorações existentes estavam voltadas para códigos OO. Assim, novos catálogos de refatoração foram elaborados para permitir a inclusão de aspectos nas estruturas de códigos OO. Dentre os vários catálogos OA publicados, o de Monteiro (2005) foi o mais estudado para realização deste trabalho.

Abordagens, como o *Aspecting* (RAMOS, 2004) que objetivam a implementação de sistemas utilizando uma linguagem que suporte aspectos, foram analisadas. Esta abordagem se caracteriza por identificar, por meio de uma lista de indícios, requisitos não funcionais no código legado, para posterior tratamento e implementação com a utilização de aspectos. Entretanto, o ponto fraco desta abordagem é que não são todos os interesses que apresentam uma lista de indícios de como encontrá-los no código fonte OO. Além disso, como os aspectos de um tipo de interesse são implementados a cada iteração, descartam-se as combinações por justaposição entre aspectos e os relacionamentos existentes entre os interesses.

A partir do estudo sobre refatoração, programação orientada a aspectos e as abordagens que promovem à inclusão de aspectos em código OO, foram apresentadas estratégias para a reestruturação de código legado. Estas estratégias representam uma combinação das tecnologias de refatoração e reestruturação de código, apoiada pelos catálogos já existentes e publicados pela academia. A partir de então, foi criado um conjunto de etapas e atividades para facilitar a introdução de aspectos em uma aplicação de forma a apoiar o uso de boas práticas de programação. Todos os passos, definidos como estratégias, foram apresentados em detalhes, juntamente, com a aplicação de exemplos de códigos independentes que fortaleceram o entendimento de cada tipo de refatoração identificada neste trabalho.

A definição de combinação por justaposição, definidas neste trabalho, sugere uma divisão na refatoração para aspectos. Onde a refatoração de primeiro nível se preocupa na extração de aspectos a partir de um código legado, enquanto a refatoração de segundo nível se ocupa da reestruturação interna dos aspectos já existentes na aplicação. Dessa forma demonstra-se a preocupação em minimizar algumas práticas não recomendadas de programação que já foram apontadas em OO, tais como a repetição de código e a dupla personalidade em aspectos.

Para realizar um experimento utilizando as estratégias definas neste trabalho, foram apresentados dois estudos de caso. O primeiro, utilizava uma classe para cadastro de pessoa. Neste exemplo permitiu-se a utilização do ciclo de reestruturação de código em duas iterações. A primeira delas demonstrou à necessidade de uma refatoração OO, pois em

primeira análise foi encontrado um problema estrutural catalogado por FOWLER (2000), caracterizado pela ocorrência de métodos muito longos. Para este problema foi realizada uma extração de método. A segunda iteração do ciclo detecta o interesse de tratamento de exceção que pode ser modularizado na forma de um aspecto. Assim, foi possível demonstrar que as estratégias são úteis para uma reestruturação sistemática de código legado.

O segundo estudo de caso, demonstrou a utilização das estratégias para a adição de novas funcionalidades. Estas estratégias levam o desenvolvedor a uma análise mais criteriosa do código legado, apoiados por catálogos de refatoração já consolidados na academia. Consequentemente, tem-se um código resultante melhor formado e mais adequado para a introdução de novas funcionalidades.

As estratégias definidas para a reestruturação de código dependem substancialmente da existência de testes. A identificação do teste como uma estratégia tem por objetivo assegurar que a reestruturação foi realizada corretamente, ou seja, não modificou o comportamento do código, nem inseriu qualquer tipo de erro que não existia anteriormente.

5.1 Contribuições

Atualmente percebe-se uma grande preocupação com a qualidade do código legado, e a programação orientada a aspectos vem como uma forma de melhorar a manutenibilidade e reusabilidade, pois a implementação dos interesses transversais fica encapsulada em módulos fisicamente separados do restante do código. Dessa forma, surgiu o interesse em migrar sistemas para essa nova abordagem. Neste ponto, é que a refatoração vem como uma forma de apoiar na simplificação do código e facilitar a sua manutenção.

Uma das grandes contribuições deste trabalho é a disponibilização de um conjunto de estratégias que definem um roteiro sistemático de ação a serem realizadas, e de recursos (catálogos) a serem utilizados com o objetivo de apoiar a migração de um sistema desenvolvido utilizando a abordagem OO para outras abordagens, e ainda garantir a agregação de novas funcionalidades, sem que comprometa a estrutura original do sistema.

5.2 Limitações e Trabalhos Futuros e em Andamento

Embora evidencie melhorias ao sistema, a utilização dos diferentes tipos de refatorações e da programação orientada a aspectos, em conjunto, é restrita e não soluciona todos os problemas. As refatorações existentes conduzem, explicitamente, às boas práticas de programação, mas nem sempre pode garantir que ao final de uma refatoração será conveniente a inclusão de aspectos. Além disso, a preparação e a experiência da equipe de desenvolvedores influenciarão no entendimento do código resultante. Deve-se avaliar ainda, se o projeto ganhará flexibilidade e manutenibilidade proporcional ao esforço despendido.

Ao modularizar interesses para aspectos, deve-se ponderar a relação custo/benefício obtida pela refatoração. No caso em que um código a ser reestruturado com aspectos possua centenas de classes, que por sua vez, possuam dezenas de aspectos diferentes, é provável que essa grande quantidade de aspectos torne o sistema mais difícil de programar. Isto ocorre porque, o código de um método pode estar sendo executado em outro lugar, assim, a clareza em relação ao que ele faz fica reduzida.

Este trabalho de dissertação descreveu um conjunto de estratégias para apoiar a reestruturação de código legado com o intuito de melhorar a legibilidade e facilitar a manutenção. Alguns trabalhos futuros puderam ser vislumbrados, tais como:

- Utilizar um código com regras de negócio mais complexo, ou até mesmo uma aplicação inteira. Esta aplicação deve ser de complexidade tal que possibilite explorar as diferentes estratégias definidas.
- Estabelecer métricas que permitam avaliar ganhos quanto à produtividade e facilidade de manutenção.
- Explorar a relação entre dois aspectos que possam, eventualmente, gerar um terceiro aspecto e verificar o seu impacto no software como um todo.

6 Referencias

BECK, K., (1999). Extreme Programming Explained: Embrace Change. Addison-Wesley.

BECK, K. et al.(2001). Manifesto for Agile Software Development. Disponível em: <http://agilemanifesto.org/>.

CARNEIRO, G. F., Neto, M. G. M., (2003). Relacionando Refactorings a Métricas de Código Fonte – Um Primeiro Passo para Detecção Automática de Oportunidades de Refactoring, in Proc. of 17º. Simpósio Brasileiro de Engenharia de Software (SBES 2003), Manaus, Brasil.

CINNÉIDE, M. O., (2000). Automated Apllications of Design Patterns: A Refactoring. Tese (Doutorado em Ciências da Computação), Universidade de Dublin, 2000.

COLEMAN, D. Object-Oriented Development - The Fusion Method. Prentice Hall, 1994.

CONSTANTINIDES, C.A., Bader, A., ELRAD, T.H. and Fayad, M.F. (2000). Designing an Aspect-Oriented Framework in an Object-Oriented Environment. ACM Computing Surveys, v.32, Nº 41.

DOMINGUES A. L. dos S. (2002). Avaliação de Critérios e Ferramentas de Teste para Programas OO. Dissertação de Mestrado, ICMC-USP, São Carlos - SP, Brasil, 2002.

DUCASSE, S., Rieger, M. and Demeyer, S. (1999). A language independent approach for detecting duplicated code, in Proc. Int'l Conf. Software Maintenance, pp. 109–118, IEEE Computer Society.

ELRAD, T.; Aksit, M.; KICZALES, G.; LIEBERHERR, K. and OSSHER, H. (2001). Discussing Aspects of AOP. Communications of the ACM, 44(10):33–38.

FOWLER, M., BECK, K., BRANT, J., OPDYKE, W., and ROBERTS, D. (2000). Refactoring: improvingthe design of existing code. Object Technology Series. Addison-Wesley.

GAMMA, E. et al. (2000). "Padrões de Projeto – Soluções Reutilizáveis de Software Orientado a Objetos", Bookman, 1º Edição.

GAMMA, E.; BECK, K. (2002). JUnit, Testing Resources for Extreme Programming. Disponível em: <http://www.junit.org/>.

GARCIA, V. C.; PIVETA, E. K.; LUCRÉDIO, D.; ALVARO, A.; ALMEIDA, E. S., PRADO, A. F., and ZANCANELLA, L. C. (2004). Manipulating crosscutting concerns. 4th Latin American Conference on Patterns Languages of Programming (SugarLoafPlop 2004). Porto das Dunas-CE, Brazil

GRAND, M. (1998). Patterns in Java: A Catalog of Reusable Design Patterns Illustred with UML. Willey, 1 ed.

HANNEMANN, J. and KICZALES, G. (2002). Design Pattern Implementation in Java and AspectJ, Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 161-173, November.

HANENBERG, S., Oberschulte, C., and Unland, R. (2003). Refactoring of aspect-oriented software. In Net.Object Days.

IEEE Standards Board (1990). IEEE Standard Glossary of Software Engineering Terminology. New York.

IWAMOTO, M. and ZHAO, J. (2003). Refactoring aspect-oriented programs. In The 4th AOSD Modeling With UML Workshop.

KERIEVSKY, J. (2004). Refactoring to Patterns, Addison-Wesley, 1 ed.

KICZALES, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J. and Irwin, J. (1997). Aspect-Oriented Programming. In M. Aksit and S. Mat- suoka, editors, European Conference on Object-Oriented Programming, volume 1241 of LNCS, pages 220-242.

KICZALES, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J. Griswold, W.G. (2001). Getting Started with AspectJ. In: Anais do ACM, pág. 59-65, Outubro.

KICZALES, G., Mezini, M. (2005). Aspect oriented programming and modular reasoning. In Proceedings of the 27th International Conference on Software Engineering (ICSE'2005), pages 49–58. ACM Press.

KULESZA, Uirá; SANT'ANNA, Cláudio; LUCENA, J. P. CARLOS. (2005).Técnicas de projeto orientado a aspectos. Uberlândia. Disponível em: http://www.sbbdsbes2005.ufu.br/mini_cursos.aspx. Acessado em 01/05/2006.

LADDAD, R. (2003). Aspect Oriented Refactoring Series. Acessado em 02/05/2007 Disponível em: < http://www.theserverside.com/tt/articles/article.tss?l=AspectOrientedRefactoringPart1 >.

LAKHOTIA, A., DEPREZ, J. C., (1998). Restructuring Programs by Tucking Statements into Functions, Information and Software Tecnology, special issue on Program Slicing, vol. 40, pp. 670-689.

LEMOS, O. A. L., MALDONADO, J. C., MASIERO, P. C. (2004). Data flow integration testing criteria for aspect-oriented programs. In Anais do 1o Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'2004), Brasília/DF - Brasil.

LIEBERHERR, K. J. e outros. (1994). Adaptive Object-Oriented Programming Using Graph-Based Customization. In: Anais do ACM, vol. 37, pág. 94-101.

MAIA, Paulo H. M. (2004). REFAX: Um arcabouço para desenvolvimento de ferramentas de refatoração baseado em XML. Dissertação – Programa de Pós-Graduação em Ciência da Computação, UFC, Fortaleza, Ceará.

MALDONADO, J. C. (1991). Critérios potenciais usos: Uma contribuição ao teste estrutural de software. Tese de Doutoramento, DCA/FEE/UNICAMP, Campinas, SP.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. R. S.; DELAMARO, M. E. (1998). Aspectos teóricos e empíricos de teste de cobertura de software. Relatório Técnico 31, Instituto de Ciências Matemáticas e de Computação – ICMC-USP.

MASIERO, P. C. ; LEMOS, Otávio A L ; FERRARI, F. C., MALDONADO, J. C. (2006). Teste de Software Orientado a Objetos e a Aspectos: Teoria e Prática. In: Karin Breitman; Ricardo Anido. (Org.). Atualização em Informática. 1 ed. Rio de Janeiro: Editora PUC-Rio, v. 1, p. 13-72.

MENS, T., DEMEYER, S., JANSSENS, D., (2002). Formalising Behavior Preserving Program Transformations, Graph Transformation.

MENS, T. and TOURWÉ, T., (2004). A Survey of Software Refactoring. IEEE Transactions on Software Engineering, Vol. 30, No. 2.

MONTEIRO, M. P.; FERNANDES, J.M. (2005). The Search for Aspect-Oriented Refactorings Must Go On, Position paper for the workshop on Linking Aspect Technology and Evolution (LATE 2005) at AOSD 2006, 14th March.

MONTEIRO, M. P.; Fernandes, J. M. (2005). Towards a Catalog of Aspect-Oriented Refactorings. Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, ACM press, pp. 111-122.

OPDYKE, W. F., (1992). Refactoring: A Program Restructuring Aid in Designing Object Oriented Aplications Framework. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign.

OSSHER, H.; TARR, P. (1999). Multi-dimentional separation of concerns in Hyperspace. In: Aspect Oriented Programming Workshop at ECOOP'99, Finlândia: Springer-Verlag, 1999.

OSSHER, H.; TARR, P. (2001). Using multidimensional separation of concerns to (re)shape evolving software. Commun. ACM, v. 44, n. 10, p. 43–50. Disponível em: http://www.st.informatik.tu-darmstadt.de/pages/seminars/aose/sac2000-HyperJ.pdf. Acessado em 02/12/2006.

PINTO, Adrio S. (2006). Utilizando a Refatoração para Reorganizar Estruturas de Código. CONSIPRO – I Congresso Simulado de Técnicas de Programação. Disponível em http://www.inf.unisinos.br/~barbosa/pipca/consipro1/a15.pdf. Acessado em 23/03/2007.

PIPKA, J. U., (2002). Refactoring in a 'Test First' - World, in Proc. of 3th International Conference on Extreme Programming and Flexible Processes in Software Engineering, pp. 71-76.

PIVETA, E. K.; HECHT, M.; PIMENTA, M. S.; PRICE, R. T. (2005). Bad Smells em Sistemas Orientados a Aspectos. In: XIX Simpósio Brasileiro de Engenharia de Software. Uberlândia. Anais do XIX SBES. 2005. v. 1, p. 184-199.

PRESSMAN, R. S. (1995). Engenharia de Software. 3. ed. Rio de Janeiro: McGraw-Hill.

PRESSMAN, R. S. (2002). Engenharia de Software. 5. ed. Rio de Janeiro: McGraw-Hill.

PROIETTI, M., PETTOROSSI, A., (1991). Semantics Preserving Transformation Ruler for Prolog, in Proc. of Symposium Partial Evaluation and Semantics-Based Program Evaluation, vol. 26, no. 9, pp. 274-284.

RAMOS, R. A., PENTEADO, R., MASIERO, P. C. (2004). Um Processo de Reestruturação de Código Baseado em Aspectos. XVIII Simpósio Brasileiro de Engenharia de Software. V.1.p. 225-240. Brasília, DF, Brasil.

RESENDE, M. P. A.; SILVA, C. C. (2005). Programação orientada a aspectos em Java. Rio de Janeiro, Ed. Brasoft.

ROCHA, D. A. (2005). Uma ferramenta baseada em aspectos para apoio ao teste funcional de programas Java. Dissertação de Mestrado - ICMC/USP, São Carlos.

ROBERTS, D. B., (1999). Practical Analysis for Refactoring. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1999.

SOARES S. ; BORBA, P. (2002) . AspectJ - Programação orientada a aspectos em Java. In: VI Simpósio Brasileiro de Linguagens de Programação. Rio de Janeiro. VI Simpósio Brasileiro de Linguagens de Programação. p. 39-55.

TICHELAAR, S. et al., (2000). A Meta-model for Language-Independent Refactoring, in Proc. Of the International Symposium on Principles of Software Evolution (ISPSE 2000). Kanazawa, Japan.

TOKUDA, L., BATORY, D., (1999). Evolving object-oriented designs with refactorings. in,Preceedings of 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, FL , Estados Unidos, p. 174-181, 12 – 15.

TOURWÉ, T., BRICHAU, J., MENS, T., (2002). Using Declarative Metaprogramming to Detect Possible Refactorings, in Proc. of 17th Automated Software Engineering (ASE 2002), 23–27 September, Edinburgh, SCOTLAND.

VINCENZI, A. M. R. (2004). Orientação a objetos: Definição e análise de recursos de teste e validação. Instituto de Ciências Matemáticas e de Computação – ICMC/USP, São Carlos, SP.

WINCK, V. Diogo; JUNIOR, G. Vicente (2006). AspectJ – Programação Orientada a Aspectos com Java. Editora Novatec, São Paulo.