FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

Leopoldo Soares de Melo Junior

Uma Estratégia de Refatoração para AspectJ utilizando Leis de Programação e XML

Fortaleza
2007

FUNDAÇÃO EDSON QUEIROZ
UNIVERSIDADE DE FORTALEZA – UNIFOR

Leopoldo Soares de Melo Junior

Uma Estratégia de Refatoração para AspectJ utilizando Leis de Programação e XML

Dissertação apresentada ao Curso de Mestrado em Informática Aplicada da Universidade de Fortaleza como parte dos requisitos necessários para a obtenção do Título de Mestre em Informática Aplicada.

Orientador: Prof. Dr. Nabor das Chagas Mendonça

Fortaleza
2007

Leopoldo Soares de Melo Junior

Uma Estratégia de Refatoração para AspectJ utilizando Leis de Programação e XML

Data de Aprovação: _____

Banca Examinadora:

_____

Prof. Dr. Nabor das Chagas Mendonça
(Presidente da Banca)

_____

Prof. Dr. Marco Túlio de Oliveira Valente
(Pontifícia Universidade Católica de Minas Gerais - PUC MG)

_____

Prof. Dr. Fernando Antonio Mota Trinta
(Universidade de Fortaleza – UNIFOR)

MELO JUNIOR, Leopoldo Soares de

Uma Estratégia de Refatoração utilizando Leis de Programação e XML. Fortaleza. Universidade de Fortaleza (UNIFOR), Dissertação de Mestrado, 2007.

83 p.: il. 210 x 297 mm (MIA/UNIFOR, M.Sc. Informática Aplicada)

1. Refatoração
2. Programação Orientada a Aspectos
3. XML

I. MIA/UNIFOR II. TÍTULO (série)

Dedico este trabalho à minha mãe,
Alcida Frota, por todo amor e apoio.

Agradecimentos

A Deus, pela força que sempre esteve comigo neste desafio.

Meus sinceros agradecimentos ao Prof. Nabor das Chagas Mendonça, pela orientação e inestimável ajuda, fundamentais para a conclusão deste trabalho.

À minha mãe, Maria Alcida da Frota, pelo suporte incondicional neste e em vários outros projetos da minha vida.

Ao tio, José Thomé da Frota, pelo apoio técnico e por todos outros tipos possíveis de apoio.

A toda a minha família, pelo apoio e incentivo que me deram durante a realização deste trabalho.

Ao amigo, Rener Silva de Menezes, pelo enorme apoio e colaboração na realização deste trabalho.

Ao amigo, Paulo Henrique Mendes Maia, pelo apoio no arcabouço RefaX.

Aos professores Fernando Antonio Mota Trinta e Marco Túlio de Oliveira Valente, pela presença na banca examinadora.

À professora Maria Andréia Formico Rodrigues, pelo incentivo, estímulo e confiança depositada.

Aos demais professores do mestrado, pelas contribuições indiretas.

À Tânia, secretária do MIA, pela presteza e atenção sempre demonstradas.

Ao Banco do Nordeste do Brasil S.A., por ter patrocinado este projeto.

Resumo da Dissertação apresentada ao MIA/UNIFOR como parte dos requisitos necessários para a obtenção do título de Mestre em Informática Aplicada.

UMA ESTRATÉGIA DE REFATORAÇÃO PARA ASPECTJ UTILIZANDO LEIS DE PROGRAMAÇÃO E XML

Leopoldo Soares de Melo Junior

Dezembro / 2007

Orientador: Prof. Dr. Nabor das Chagas Mendonça
Programa: Informática Aplicada

Este trabalho apresenta um processo de refatoração de código orientado a aspectos que permite construir refatorações codificando apenas em uma linguagem declarativa. Esta abordagem utiliza AspectJML, uma representação em XML de AspectJ, para armazenar as estruturas sintáticas do código AspectJ; XSLT, uma linguagem declarativa de transformação para XML, para implementar as transformações definidas pelas leis de programação requeridas para cada refatoração; e uma instancia de uma adaptação do RefaX, um arcabouço de refatoração baseado em XML para a linguagem Java, para implementar a ferramenta de refatoração para AspectJ. A principal vantagem desta abordagem é a possibilidade de construir e customizar refatorações de código AspectJ se beneficiando dos recursos de alto nível de casamento de padrões e de transformação oferecidos por XSLT. Esta abordagem foi avaliada com a construção de duas refatorações para AspectJ, *extract pointcut* e *extract method calls*.

Abstract of Thesis presented to MIA/UNIFOR as a partial fulfillment of the requirements for the degree of Master of Applied Informatics.

A REFACTORING APPROACH TO ASPECTJ USING PROGRAMMING LAWS AND XML

Leopoldo Soares de Melo Junior

December / 2007

Adviser: Prof. Dr. Nabor das Chagas Mendonça

Program: Applied Informatics

This work presents a refactoring approach to aspect oriented code that allows to build refactorings coding only with a declarative language. This approach uses AspectJML, an XML-based representation of AspectJ, to store the syntactic structures of AspectJ code; XSLT, an XML declarative transformation language, to implement the transformations defined by the programming laws required by each refactoring; and an instance of an adaptation of RefaX, an XML-based refactoring framework for Java, to build the AspectJ refactoring tool. The main advantage of this approach is the possibility to build and customize AspectJ refactorings by taking advantage of the high-level pattern-matching and transformation capabilities of XSLT. This approach was evaluated with two refactorings for AspectJ, *extract pointcut* and *extract method calls*.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas e Siglas

OO	*Orientação a Objetos*
POO	*Programação Orientada a Objetos*
OA	*Orientação a Aspectos*
POA	*Programação Orientada a Aspectos*
XML	*Extensible Markup Language*
XSLT	*Extendible Stylesheet Language Transformations*
ajc	*AspectJ Compiler*
DOM	*Document Object Model*
AST	*Abstract Syntax Tree*
IDE	*Integrated Development Environment*
CASE	*Computer-Aided Software Engineering*

Capítulo 1

Introdução

Este capítulo apresenta as principais questões que motivaram a realização deste trabalho, assim como seus objetivos, contribuições e sua organização.

1.1 Motivação

Refatoração [22][64] é uma técnica amplamente utilizada para melhorar a qualidade de código fonte existente. Ela permite alterar a estrutura do programa sem alterar seu comportamento observável [22]. No entanto, mesmo em pequenos sistemas, verificar as pré-condições de aplicação e aplicar as transformações manualmente é uma tarefa exaustiva. Em grandes aplicações, efetuar estas atividades manualmente é uma tarefa praticamente inviável. Isto ratifica a importância das ferramentas automatizadas para aplicação de refatorações.

Muitas refatorações foram identificadas para programação orientada a objetos (POO) [64] e vários ambientes de desenvolvimento, como o Eclipse [18], já disponibilizam ferramentas automáticas de refatoração para esse paradigma. Entretanto, mesmo as refatorações convencionais existentes nestas ferramentas de desenvolvimento mostram-se incorretas em algumas situações [19]. Isto ressalta a importância da existência de ferramentas que permitam que usuários customizem suas próprias refatorações.

O paradigma de programação orientado a aspectos (POA) [43] introduz um novo conceito de modularização sobre a POO. Com POA, os interesses entrecortantes (ou transversais), ou seja, interesses de uma aplicação que não conseguem ser modularizados e afetam boa parte do código, podem ser encapsulados em um só módulo, o aspecto. Aspectos implementam as funcionalidades entrecortantes e são ligados às partes do programa onde devem ser aplicados através dos conjuntos de junção (*pointcuts*). Um compilador especial

mescla este código antes de sua execução. A principal linguagem de programação POA e a mais utilizada é o AspectJ [6].

O paradigma orientado a aspectos (AO) complementa o paradigma orientado a objetos (OO). Entretanto, o grau de maturidade atual das ferramentas de refatoração para POA [27][25][28] ainda é incipiente. Diferentemente da POO, não existe atualmente uma ferramenta de refatoração para POA que seja amplamente utilizada pela comunidade. Isto posto, percebe-se uma grande necessidade de construir ferramentas de refatoração de código OA customizáveis.

Paralelo ao exposto acima, trabalhos anteriores [50][32][12][2][56][52] atestaram viabilidade do uso da linguagem XML [70] para construção de ferramentas de refatoração customizáveis. Esta linguagem de marcação permite a representação da estrutura sintática de código fonte de forma explícita, e oferece uma vasta gama de ferramentas de consulta e atualização (por exemplo, Xpath [71], Xquery [74], Xupdate [76], XSLT [73]) que podem auxiliar na construção e customização de ferramentas de refatoração.

1.2 Objetivos

Este trabalho propõe uma abordagem que permita a construção de refatorações definidas pelo usuário para AspectJ sem exigir codificação em linguagens de programação imperativas.

Os resultados alcançados com esta abordagem incluem:

- Identificar os requisitos necessários para construção de um ambiente de refatoração para AspectJ que permita a construção de refatorações sem exigir codificação em linguagens de programação imperativas;

- Uma solução que atenda os requisitos identificados, de forma mais satisfatória que os trabalhos até então existentes;

- Construir uma ferramenta para dar suporte a essa abordagem;

- Implementar refatorações com a ferramenta proposta e conduzir uma avaliação preliminar da solução.

1.3 Contribuições

Para atingir os objetivos acima, realizou-se uma pesquisa sobre as estratégias que facilitam a construção de refatorações. Como resultado deste estudo, selecionou-se XML como estrutura de representação de código para aplicação das refatorações; o RefaX [56], um arcabouço para desenvolvimento de refatorações, para suportar a ferramenta de desenvolvimento de refatorações; e a derivação de refatorações como leis de programação [16], que permite decompor refatorações em transformações de código mais simples.

As tecnologias acima foram disponibilizadas na forma de uma infra-estrutura de desenvolvimento de refatorações para AspectJ que permite a customização de novas refatorações codificando-as apenas em uma linguagem de programação declarativa. Para isso, criou-se uma representação completa da árvore sintática de código AspectJ. Também foi modificado o arcabouço RefaX [56] para construir a ferramenta de refatoração para AspectJ. Além disso, utilizou-se a derivação de refatorações como leis de programação proposta por Cole e Borba [16] para facilitar a customização das refatorações. As principais contribuições deste trabalho são sumarizadas abaixo:

- Construção de uma representação sintaticamente completa de código fonte AspectJ em XML, denominada AspectJML [53][9];
- Definição de um processo de desenvolvimento de refatorações para AspectJ [55] que utiliza somente linguagens de programação declarativas para customizar novas refatorações;
- Remodelagem do arcabouço RefaX para suportar o processo de desenvolvimento de refatorações citado no item anterior;
- Construção de uma nova ferramenta de refatoração para AspectJ [54] que instancia o arcabouço RefaX remodelado para dar suporte ao processo de desenvolvimento de refatorações proposto.

1.4 Estrutura da Dissertação

Além desta Introdução, este trabalho está organizado nos capítulos descritos a seguir.

No Capítulo 2, **Refatoração de Código em POA**, é realizado um apanhado sobre o estado da arte da refatoração de código para o paradigma de programação orientado a aspectos.

No Capítulo 3, **AspectJML: Uma Linguagem de Marcação para AspectJ**, é descrita a linguagem de marcação proposta neste trabalho para representar código AspectJ em XML.

No Capítulo 4, **Um Processo de Refatoração para AspectJ utilizando Leis de Programação, AspectJML e XSLT**, a estratégia de construção de refatorações proposta neste trabalho é apresentada.

Por fim, no Capítulo 5, **Conclusão**, são apresentadas as considerações finais sobre este trabalho e enunciados tópicos para trabalhos futuros.

Capítulo 2

Refatoração de Código em POA

Este capítulo apresenta uma revisão da literatura sobre refatoração de código para a programação orientada a aspectos.

2.1 Introdução

Refatorações de código são transformações complexas que exigem a manipulação de estruturas de árvores sintáticas com fina granularidade. Elas estão bem definidas para a POO e há um número considerável de ferramentas que as automatizam. Também já existem muitos trabalhos propondo refatorações para Programação Orientada a Aspectos (POA).

Este capítulo oferece um breve resumo sobre a POA (Seção 2.2); uma visão geral sobre refatoração para POO (Seção 2.3); a classificação e os principais trabalhos existentes sobre refatoração para POA (Seção 2.4); e um relato sobre as ferramentas disponíveis para refatoração em POA (Seção 2.5).

2.2 Programação Orientada a Aspectos

Um software pode ser visto como um conjunto de módulos estruturados, cada módulo representando um interesse, i.e, funcionalidade ou requisito. No entanto, as abstrações oferecidas pelo paradigma OO (classes, objetos, métodos e atributos) são insuficientes para expressar de forma modular todos os interesses de um software. Por exemplo, tratamento de interesses como *log* de transações, trilha de auditoria ou persistência tendem a ficar espalhados em múltiplos módulos do sistema. Estes interesses são conhecidos como *interesses entrecortantes* porque eles atravessam outras funcionalidades do programa. Como conseqüência, trechos de software implementados estritamente com o paradigma OO podem reduzir a clareza, manutenibilidade e reusabilidade do sistema [24].

A POA surge como uma solução para tratar o problema de interesses entrecortantes [21]. Ela introduz uma nova abstração, chamada aspecto, e um mecanismo de integração de aspectos com componentes orientados a objetos, tais como classes, métodos e atributos. Desta forma, este novo paradigma oferece uma maneira de separar interesses entrecortantes e garantir uma melhor modularização.

A POA faz parte de um tópico de pesquisa mais amplo chamado Separação de Interesses Avançada (*Advanced Separation of Concerns* ou ASoC). Além da POA, há várias outras linhas de pesquisa sobre este tópico, tais como: Filtros de Composição (*Composition Filters)* [4][14], Programação Adaptativa (*Adaptive Programming*) [46][47], Programação Orientada a Assuntos (*Subject-Oriented Programming*) [29], Separação de Interesses Multidimensional (*Multi-Dimensional Separation of Concerns*) [61][62], entre outros. Destes, nem todos foram bem aceitos junto à comunidade de engenharia de software. Por exemplo, a Programação Orientada a Assuntos não tem mais esforços de desenvolvimento em andamento. Para ser mais preciso, nenhum dos esforços mencionados anteriormente alcançou até agora o nível de maturidade da POA.

Aspectos são compostos pelo entrelaçamento do código do aspecto com o código base, que inclui classes, interfaces e outras construções de uma linguagem OO. Este processo, denominado combinação (ou *weaving*), funciona introduzindo as instruções do código do aspecto no código base, produzindo uma versão do código combinado. No entanto, esta combinação não prejudica o processo de desenvolvimento, uma vez que ela pode ser refeita sempre que uma nova versão for requisitada. Esta combinação é especificada por uma linguagem de pontos de junção. Cada linguagem da POA possui sua linguagem de especificação de pontos de junção.

A POA visa a complementar outros modelos de programação, não substituí-los. O que diferencia aspectos dos módulos tradicionais é a natureza entrecortante dos interesses que eles modularizam. É importante perceber a natureza relativa deste conceito: um interesse pode ser entrecortante usando um modelo, mas pode ser claramente encapsulado usando outro. Por exemplo, o modelo procedural decompõe sistemas de acordo com algoritmos, e os dados são espalhados por múltiplos procedimentos. Na POO, dados e funcionalidades constituem a decomposição primária do sistema. Neste caso, os dados são claramente modularizados. Requisitos não funcionais normalmente se sobressaem como entrecortantes. Este é o motivo de aspectos que complementam aplicações OO tenderem a implementar requisitos não funcionais.

A primeira e ainda a mais conhecida linguagem orientada a aspectos é AspectJ [41][42][44], uma extensão da linguagem Java. Várias linguagens orientadas a aspectos apareceram posteriormente [40], mas a maioria foi fortemente influenciada pelo projeto de AspectJ. Atualmente, AspectJ é um padrão *de facto* para POA em termos de projeto de linguagem. Por esta razão, consideraremos nesta dissertação o AspectJ como linguagem POA padrão.

A POA melhora a compreensão e a manutenibilidade de programas complexos isolando comportamentos dispersos, que estariam espalhados pelas classes da aplicação com o uso da POO. Estes comportamentos foram originalmente chamados de interesses (*concerns*) por Dijkstra [17]. Kiczales *et al* [42] definem um aspecto como "uma implementação bem modularizada de um interesse entrecortante". Apresentamos a seguir as definições dos principais conceitos da POA.

- Ponto de junção (*Join point*): São elementos da linguagem de programação que interagem com aspectos. Há vários modelos de pontos de junção disponíveis e outros ainda em construção. Eles dependem fortemente da linguagem de programação do código base e da linguagem OA. Em várias linguagens OA disponíveis atualmente, um ponto de junção é uma região do fluxo de controle dinâmico da aplicação. Por exemplo, ele pode ser uma chamada de método, a execução de um método, o evento de atualizar um atributo ou um evento de tratamento de uma exceção.

- Conjunto de junção (*Pointcut*): É um predicado sobre pontos de junção dinâmicos. Dado um determinado ponto de junção dinâmico, um conjunto de junção pode tanto casar com este ponto de junção como com sua negação em tempo de execução. Uma outra visão de conjuntos de junção é que eles representam conjuntos de pontos de junção. Um conjunto de junção pode expor informações em tempo de execução para um adendo (ver abaixo).

- Adendo (*Advice*): Um adendo é composto por um conjunto de junção e um corpo de método. O corpo do método executa quando um conjunto de junção casa com um ponto de junção. Este corpo de método inclui um comportamento adicional ou substitui o corpo do método original do código base por algo completamente diferente.

- Declaração intertipos (*Inter-type declaration*): Também conhecidas como *static introductions*, uma declaração intertipos adiciona atributos e métodos à estrutura original de classes do código base.

- Cláusulas de declaração (*declare clauses*): Estas estruturas configuram ou estendem o comportamento do compilador da linguagem OA. AspectJ disponibiliza as cláusulas '*declare warning*' e '*declare error*' para instruir o compilador para lançar advertências e erros, respectivamente. As duas cláusulas associam conjuntos de junção com uma advertência ou um erro e sua respectiva mensagem.

2.3 Refatoração em Programação Orientada a Objetos

Refatoração é o processo de modificar um sistema de software de tal forma que seu comportamento externo não se modifique e sua estrutura interna melhore [22]. É também uma forma disciplinada de organizar o código, minimizando as chances de introduzir defeitos. Quando se aplica refatoração em um código, está-se melhorando o projeto do código depois de ele ter sido escrito. Vale ressaltar que refatoração é um termo aplicado a sistemas orientados a objeto. Este mesmo processo é descrito como reestruturação para outros paradigmas de programação [58]. Como o contexto deste trabalho é refatoração de código, a reestruturação não será abordada nesta dissertação.

Sem refatoração, a qualidade do projeto de software de um sistema com mudanças contínuas de requisitos decai. Com alterações para realizar pequenos objetivos ou mesmo alterações sem a devida compreensão do projeto do código, o código gradualmente perde sua estrutura original. Com o tempo, fica difícil entender o projeto do código a partir da sua leitura. E quanto mais difícil de ler o projeto, mais difícil será para preservá-lo. Neste contexto, refatoração se assemelha a uma arrumação de código, que ajuda a manter no código o formato original do seu projeto.

Refatorações são tipicamente realizadas através de pequenos passos, frequentemente com testes entre as etapas para garantir que nenhum erro seja introduzido. É recomendável realizar reestruturações de código como uma seqüência de pequenas refatorações somente para reduzir a probabilidade de inclusão de erros. Grandes refatorações são usualmente decompostas em várias menores. Em alguns casos, uma dada refatoração pode requerer a aplicação de outras refatorações para preparar o código para sua aplicação. Um exemplo deste caso é a refatoração *Extract Method* [22]. Esta refatoração descreve uma

transformação que aumenta a modularidade de chamadas de métodos que aparecem em vários outros métodos. Ela cria um novo método a partir de um trecho de código do método original, substituindo o trecho de código por uma chamada ao novo método criado. A Figura 1 ilustra a aplicação desta refatoração.

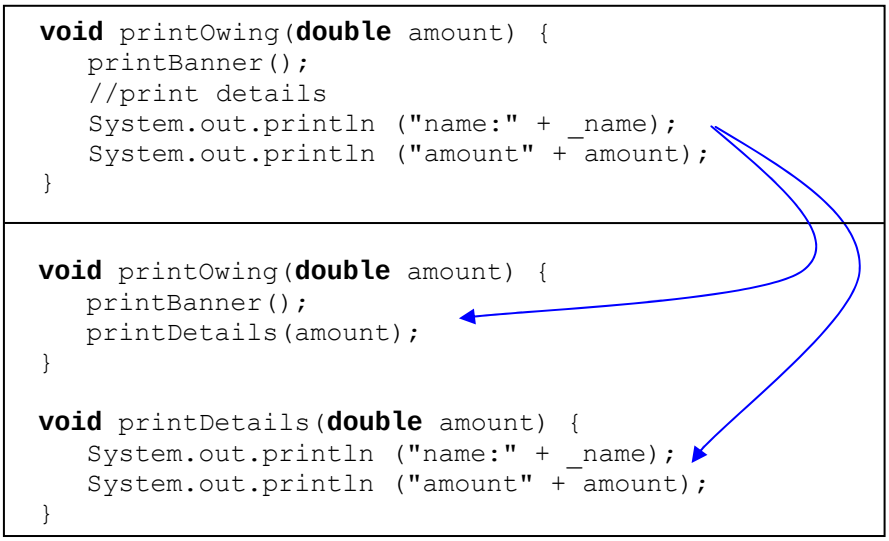

Figura 1 – Exemplo de refatoração *Extract Method*

Refatorações podem ser aplicadas manualmente ou automaticamente, através do suporte de ferramentas. No entanto, a disponibilidade de ferramentas para transformação de código é essencial para prover segurança e aumentar a produtividade. Além disso, ferramentas normalmente disponibilizam a opção de desfazer a refatoração, fundamental quando o programador encontra algum problema introduzido pela refatoração.

2.4 Refatoração em Programação Orientada a Aspectos

Esta seção apresenta alguns trabalhos existentes sobre refatoração para POA. Utilizou-se a classificação dos tipos de refatoração para POA proposta por Hannemann [26]. Também é resumida nesta seção a derivação de refatorações para AspectJ a partir de leis de programação [31] proposta por Cole e Borba [16]. O resultado deste último trabalho foi utilizado para a realização dos objetivos desta dissertação.

2.4.1 Classificação

Hannemann descreve três grupos de refatoração para POA. A primeira classe trata de refatorações OO convencionais adaptadas para funcionamento em sistemas orientados a aspectos (*Aspect-Aware OO Refactorings*). A segunda classe trata de refatorações que

envolvem elementos da POA. E a terceira e última classe compreende as refatorações que tratam interesses entrecortantes. Estas três classes serão explanadas nas subseções a seguir.

2.4.1.1 Refatorações OO adaptadas para Sistemas Orientados a Aspetos

Hanenberg *et al*. [25] mostram que refatorações OO tradicionais podem afetar o comportamento do software quando aplicadas diretamente a sistemas orientados a aspectos. Um exemplo de refatoração citado neste trabalho é a *Rename Method* [22], que consiste em alterar o nome de um método de uma classe. A simples aplicação desta refatoração em um sistema OA pode alterar o conjunto de pontos de atuação de algum adendo, alterando o comportamento da aplicação. Os autores propõem adaptações nas refatorações OO convencionais para que elas funcionem também em sistemas orientados a aspectos.

Ainda segundo Hanenberg *et al*. [25], a razão para a existência de conflitos entre refatorações OO e as características da POA é a própria natureza da especificação dos aspectos. Conforme explanado na seção 2.2, aspectos são combinados nos locais definidos por pontos junção especificados por uma linguagem de pontos de junção. Esta linguagem, que é o principal elemento das linguagens orientadas a aspectos, especifica chamadas de métodos e construtores. Como refatorações OO eventualmente podem alterar a estrutura do código base (incluindo o nome dos métodos), a atuação dos aspectos também é modificada com a aplicação de refatorações OO.

Em [25], Hanenberg *et al.* sugerem três condições que devem ser observadas para preservar o comportamento observável de refatorações OO em sistemas OA. São elas: (i) a quantidade de pontos de junção afetados por um determinado conjunto de junção não se altera depois da refatoração; (ii) os pontos de junção afetados por um determinado conjunto de junção têm uma posição equivalente no fluxo de controle do programa quando comparado ao estado à anterior aplicação da refatoração; e (iii) as informações de contexto oferecidas por cada conjunto de junção não diminuem.

O exemplo de código apresentado na Figura 2, extraído de [25], ilustra a alteração de comportamento que a refatoração *Rename Method* causa em um sistema orientado a aspectos. Com a alteração do nome do método *foo* para *bar*, o conjunto de junção *pc1* passa a não interceptar mais este método e o conjunto de junção *pc2* passa a interceptá-lo. Estes conjuntos de junção precisam ser ajustados para não modificar o comportamento observável do sistema. Detalharemos a seguir a adequação da refatoração *Rename Method* para aplicação

em sistemas OA. Mais detalhes sobre esta e outras refatorações podem ser encontrados em [25].

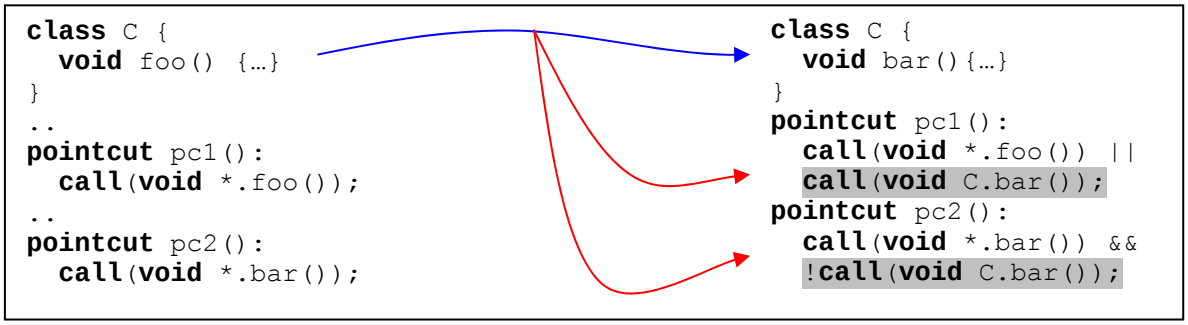

Figura 2 – Refatoração *Rename Method* afetando o comportamento observável

Os designadores de conjuntos de junção que são afetados pela refatoração *Rename Method* são os que podem conter padrões de assinatura. São eles: *call*, *execution* e *withincode*. Para adaptar a refatoração *Rename Method* para sistemas OA, precisa-se realizar os seguintes passos adicionais. Primeiramente, tem-se que identificar todos os conjuntos de junção relacionados para esta assinatura usando os designadores mencionados acima. Em seguida, altera-se o nome do método e todas as chamadas de métodos. O último passo consiste em ajustar cada conjunto de junção afetado. Se o conjunto de junção afeta apenas um ponto de junção, é aplicado um procedimento, caso contrário, é aplicado outro.

Se o conjunto de junção afetar apenas um ponto de junção, é suficiente substituir o padrão de assinatura. Já se o conjunto de junção afetar uma coleção de pontos de junção (causado, por exemplo, por utilização de coringas, ilustrado na Figura 2), os seguintes passos devem ser realizados:

- Se o conjunto de junção for anônimo, deve-se criar um nomeado;
- Copiar o designador de conjunto de junção afetado e substituir o padrão de assinatura pelo nome do novo método;
- Compor o conjunto de junção existente com o criado no passo anterior com o operador ||.

Esta abordagem mantém o conjunto original de pontos de junção afetados pelo conjunto de junção, conforme exemplificado no conjunto de junção *pc1* da Figura 2. Com isso, a primeira condição é atendida. Finalmente, tem-se que verificar se a nova assinatura do

método é referenciada por algum conjunto de junção que não o afetava antes. Nestes casos, deve-se seguir os passos indicados abaixo para remover o método do alcance deste conjunto de junção.

- Se o conjunto de junção é anônimo, deve-se criar um nomeado;
- Criar um novo conjunto de junção a partir da cópia de um existente. Substituir o padrão de assinatura com a nova assinatura de método e adicionar sua negação usando o operador `!`;
- Compor o conjunto de junção existente com o criado no passo anterior com o operador `&&`.

Esta negação exclui o ponto de junção representado pelo novo método da coleção de pontos de junção afetados pelo conjunto de junção *pc2*. Deste modo, a primeira condição é completamente atendida. Como a refatoração *Rename Method* não viola as condições 2 e 3, o procedimento acima é suficiente para adaptar esta refatoração a sistemas OA.

Hanenberg *et al.* [25] implementaram várias refatorações OO adaptadas para sistemas OA e também algumas refatorações específicas da POA. Estas refatorações foram disponibilizadas na forma de um *plug-in* do Eclipse, embora este não esteja publicamente disponível para avaliação.

2.4.1.2 Refatorações de elementos OA

Enquanto as refatorações apresentadas anteriormente focam em elementos OO, esta classe de refatorações envolve explicitamente elementos da POA. Monteiro e Fernandes [59] apresentam um catálogo de refatorações de elementos OA. O trabalho subdivide esta classe em refatorações que extraem aspectos a partir do código de sistemas OO, e refatorações oriundas de melhorias necessárias dentro das estruturas dos aspectos. Os autores ainda descrevem essas refatorações de maneira similar a descrição feita para as refatorações OO por Fowler [22].

2.4.1.3 Refatorações de interesses entrecortantes

Refatorações de interesses entrecortantes [26] transformam implementações dispersas em uma forma modularizada (um aspecto). A capacidade de modularizar interesses

entrecortantes, como persistência e trilha de auditoria, em um só módulo, é uma das características mais importantes da POA. Desta forma, as refatorações desta classe desempenham um papel de destaque, pois têm a capacidade de refatorar sistemas OO legados.

2.4.2 Derivação de Refatorações para AspectJ a partir de Leis de Programação

Motivados pelas dificuldades de desenvolver e aplicar refatorações, que muitas vezes exigem mudanças abrangentes na aplicação, Cole e Borba [16] definiram um conjunto de leis de programação [31] úteis para derivar refatorações para AspectJ. Estas leis ajudam os programadores de refatorações a verificar se as transformações que eles definem de fato preservam o comportamento do código, e mostram como combinações de pequenas transformações AspectJ podem formar uma refatoração. Os autores consideram um subconjunto das linguagens Java e AspectJ para definir as transformações. Segundo eles, esta decisão simplifica a definição das transformações sem comprometer os resultados. Esta subseção relata a estrutura das leis definidas por Cole e Borba e apresenta algumas refatorações para AspectJ derivadas por eles a partir das leis definidas.

2.4.2.1 Leis de Programação

Leis de programação definem equivalência entre dois programas dado que algumas condições sejam respeitadas. Cada lei é composta por três partes: lado esquerdo, lado direito, e pré-condições. As duas primeiras partes são modelos de programas equivalentes. A terceira indica as condições que devem ser respeitadas para garantir a equivalência entre os programas.

A Figura 3 ilustra a lei de programação Usar um Conjunto de Junção Nomeado (*Use Named Pointcut*) [16]. Esta lei tem o objetivo de substituir expressões de conjunto de junção por uma referência a um conjunto de junção nomeado. Ainda na Figura 3, o modificador *ts* representa um conjunto de declarações de tipos (classes e aspectos). O termo *pcs* representa um conjunto de declarações de conjuntos de junção. Os termos *bars* e *afs* representam, respectivamente, conjuntos de declarações de adendos do tipo *before/around* e *after*. O termo *ps* representa uma lista de parâmetros. Finalmente, o termo *exp(αps)* representa a expressão de conjunto de junção que será substituída nesta lei de programação. O quadro esquerdo e o direito da Figura 3 representam os modelos de programas equivalentes, e a parte inferior da mesma figura indica que não há condições para aplicação da lei de programação em nenhum dos dois sentidos.

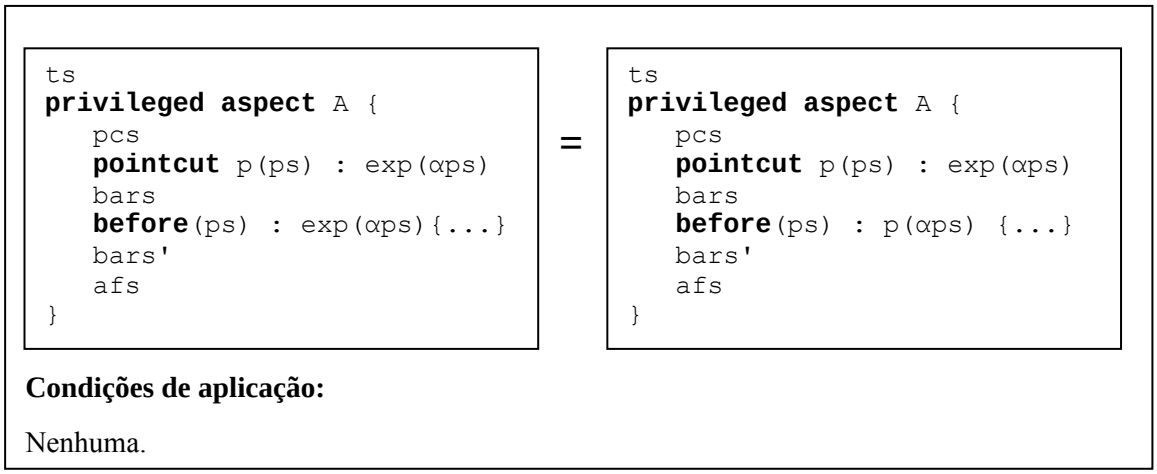

Figura 3 – Lei de programação Usar um Conjunto de Junção Nomeado *(Use Named Pointcut)*

2.4.2.2 Derivação de Refatorações a partir de Leis de Programação

Esta seção apresenta dois exemplos de refatorações, *Extract pointcut* [34] e *Extract Method Calls* [45], derivadas por Cole e Borba a partir de suas leis de programação.

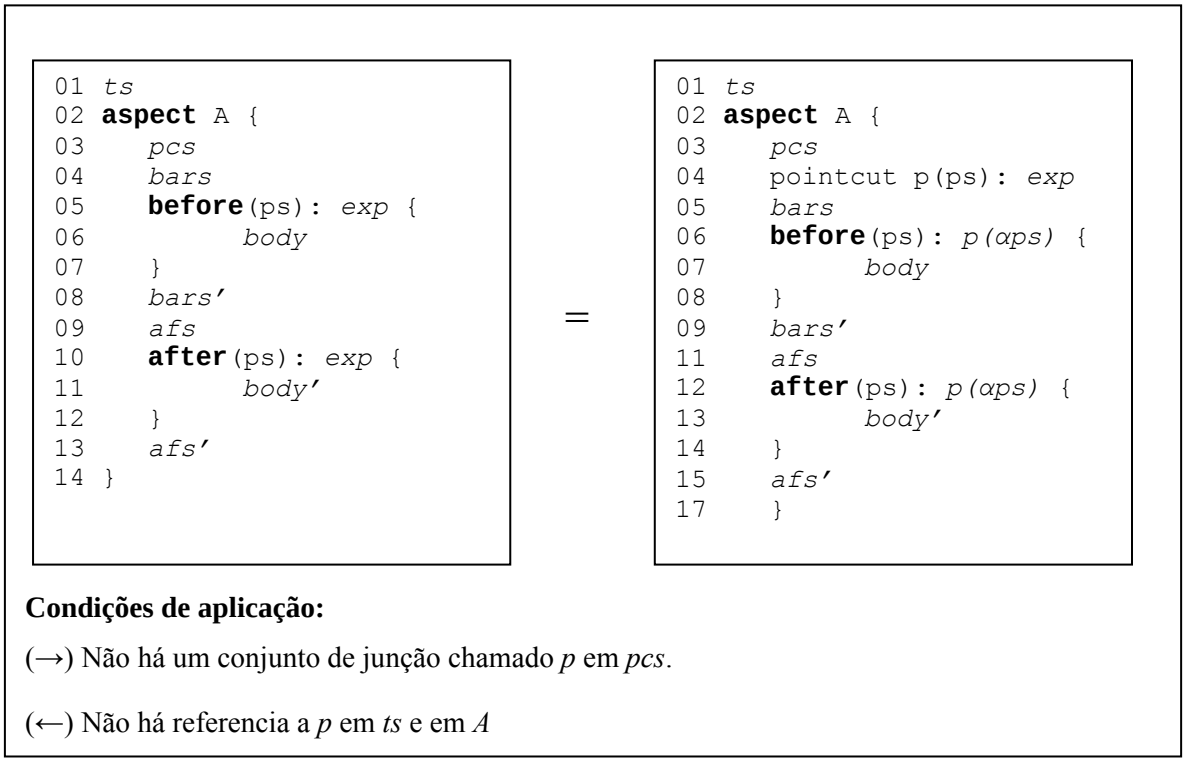

Figura 4 – Refatoração *Extract Pointcut*

A refatoração *Extract Pointcut* descreve uma refatoração de AspectJ para AspectJ que melhora a qualidade do código. Ela cria um conjunto de junção nomeado a partir de expressões usadas em adendos e fazem estes adendos usarem o novo conjunto de junção criado, promovendo o reuso de código.

A Figura 4 ilustra a refatoração *Extract Pointcut.* Ela transforma um conjunto de junção anônimo (*exp*) usado por diversos adendos em um conjunto de junção nomeado (*p*) e altera todos os adendos que usam o mesmo conjunto de junção anônimo por uma referência ao novo conjunto de junção nomeado. Na Figura 4, *ts*, *pcs*, *bars*, *afs*, *ps* representam, respectivamente, um conjunto de declarações de tipos, um conjunto de adendos do tipo *before/around*, um conjunto de adendos do tipo *after* e uma expressão de conjunto de junção. Da mesma forma que na lei de programação apresentada na subseção anterior, a parte inferior da Figura 4 indica que há condições para a refatoração ser aplicada. Observe que, como as refatorações são derivadas de leis de programação, elas também são definidas como operações bidirecionais. No entanto, sabe-se que apenas a transformação da esquerda para a direita é uma refatoração, já que apenas esta melhora a qualidade do código.

Esta refatoração é derivada das leis *Extract Named Pointcut* e *Use Named Pointcut*. A lei *Extract Named Pointcut* cria um novo conjunto de junção nomeado da expressão *exp*. Em seguida, é necessário aplicar a lei *Use Named Pointcut* em cada adendo que usa a mesma expressão representada no novo conjunto de junção nomeado. Esta lei é responsável por mudar um adendo existente para usar um conjunto de junção nomeado que declara a mesma expressão usada pelo adendo.

2.5 Ferramentas de Refatoração em Programação Orientada a Aspectos

Hannemann *et al.* [27] apresentam uma ferramenta de suporte a refatorações para sistemas orientados a aspectos. Esta ferramenta está disponibilizada na forma de um *plug-in* do Eclipse e auxilia a aplicação de refatorações de interesses entrecortantes por meio de caixas de diálogo. Estas funcionam como *wizards* e são usadas para ajudar nas decisões de projeto de software durante o processo de refatoração. A grande vantagem desta abordagem é a facilidade oferecida ao usuário de acompanhar a operação de refatoração. No entanto, não há preocupação aparente em facilitar o desenvolvimento de novas refatorações.

A ferramenta descrita em [25] também foi implementada como um *plug-in* para o Eclipse. Esta ferramenta disponibiliza para o programador um tipo de fluxo de trabalho (*workflow*) de refatoração. O desenvolvedor participa deste fluxo de trabalho confirmando as alterações propostas pela ferramenta, por conta da complexidade das modificações. Estão disponíveis nesta ferramenta refatorações OO para sistemas OA e refatorações OA. Com esta mesma preocupação do impacto de refatorações OO em sistemas OA, a versão 3.2 do Eclipse

também implementa a adaptação nos aspectos para a refatoração Renomear Classe (*Rename Class*). Da mesma forma que a ferramenta anterior, a interação com o usuário da ferramenta é facilitado, mas não há preocupação com a facilidade de desenvolver novas refatorações.

Hannemann *et al.* [28] apresentam uma estratégia de refatoração baseada em papéis para interesses entrecortantes. Nesta abordagem, um interesse entrecortante é descrito como um conjunto de elementos ("papéis") inter-relacionados que abstraem o funcionamento dos elementos de programa que compõem o interesse. Refatorações de interesses entrecortantes são definidas em termos destes elementos, separando a descrição da refatoração da sua implementação concreta. Da mesma forma que os trabalhos supracitados, um *plug-in* do Eclipse foi construído para dar suporte a essa estratégia.

Arcoverde *et al.* [5] apresentam o AJaTS, um sistema de transformação para AspectJ. Este sistema é baseado no JaTS [15], uma estratégia de construção de refatorações baseada em linguagens de *templates* para a linguagem Java. No entanto, nesta abordagem, apenas o casamento de *templates* em nível de código fonte é utilizado. Isto obriga a análise de grande parte do código em refatorações que envolvem alterações de grande abrangência, como na remoção das chamadas de métodos da refatoração *Extract Method Calls*.

AOP-Migrator [8] é uma ferramenta que suporta migração semi-automática de um sistema escrito em Java para um sistema em AspectJ. Esta ferramenta suporta a implementação de refatorações de interesses entrecortantes. Uma coleção de refatorações desenvolvidas com AOP-Migrator foi integrada pelos autores ao ambiente de refatorações do Eclipse [18].

2.6 Conclusão

Os trabalhos descritos neste capítulo apresentam estratégias diversas para refatoração de código orientado a aspectos, mas apenas Arcoverde *et al.* [5] visa aumentar a facilidade de criar e customizar novas refatorações. A proposta deste trabalho é construir um ambiente que favoreça a customização de novas refatorações. Para isso, será apresentado no próximo capítulo uma representação de código AspectJ em XML. Esta representação será usada para facilitar a customização de refatorações pelo processo de refatoração para AspectJ proposto no Capítulo 4.

Capítulo 3

AspectJML: Uma Linguagem de Marcação para AspectJ

Este capítulo apresenta uma representação de código fonte AspectJ baseada em XML denominada AspectJML.

3.1 Introdução

Desde as primeiras linguagens de programação, programadores têm usado representações textuais para codificar programas. Esta forma de representação de código fonte é bastante concisa e similar às linguagens naturais. Contudo, o texto não é o melhor formato de manipular código fonte em um nível de abstração desejável para ferramentas de engenharia de software. Tarefas como análise estrutural e semântica, que requerem a análise sintática do código, são particularmente difíceis de realizar a partir de uma representação puramente textual.

O principal problema das representações textuais é que elas não capturam de forma explícita a estrutura sintática do código. Neste contexto, vários autores propuseram utilizar linguagens de marcação, baseadas em XML [70], para representar código fonte de forma estruturada (por exemplo, [32][50][12][2][52]). Essas representações, além de tornarem explícita (e, portanto, mais fácil de consultar, analisar e manipular) a estrutura sintática do programa, trazem ainda uma série de benefícios para os desenvolvedores de ferramentas de engenharia de software, incluindo [67]: (i) possibilidade de anotar o código com informações estruturadas (por exemplo, para representar comentários e diretivas de compilação); (ii) facilidade para formatar o código de acordo com diferentes estilos de programação ou visualização (por exemplo, tecnologias de transformação XML, como XSLT [73], poderiam ser empregadas para aumentar a legibilidade do código através da utilização adequada de leiautes, cores e fontes); (iii) possibilidade de associar fragmentos de código diretamente, facilitando a navegação e permitindo a recolocação de fragmentos de código sem a perda das

referências; e (iv) suporte para a rápida implementação de ferramentas de manipulação de código em uma variedade de plataformas e linguagens de programação (por exemplo, utilizando a abundância de padrões e tecnologias disponíveis para consulta e atualização de dados XML, como Xpath [71], Xquery [74] e Xupdate [76]).

Motivados pelos benefícios acima, e pela crescente popularidade da linguagem Java entre programadores, alguns autores propuseram representações em XML específicas para Java [50][12][51]. Destas, a representação proposta por Badros, denominada JavaML [12], tem sido a mais amplamente utilizada. JavaML representa informações semânticas e estruturais de um programa Java tipicamente presentes em árvores sintáticas (*abstract syntax trees* – ASTs). Mais recentemente, Aguiar *et al.* propuseram uma extensão de JavaML, denominada JavaML 2.0 [2], que adicionou suporte para representar referências cruzadas entre as entidades do programa e para reter as informações léxicas do código fonte (basicamente, comentários e dados de formatação).

Este capítulo apresenta uma evolução de JavaML 2.0, chamada AspectJML (*AspectJ Markup Language*) [53], que incorpora as características da linguagem JavaML 2.0 e inclui suporte à estrutura sintática de AspectJ [6]. Esta melhoria foi motivada pela inexistência de representações sintaticamente completas em XML para linguagens baseadas no paradigma da POA [43], aqui representadas por AspectJ. Assim, a principal contribuição de AspectJML é permitir que os benefícios inerentes às representações de código fonte em XML também possam ser estendidos ao contexto da POA. Em particular, o AspectJML pode ser utilizado para construir refatorações utilizando as ferramentas de manipulação XML existentes.

O restante do capítulo está organizado da seguinte forma. A Seção 3.2 ilustra os principais elementos que compõem as representações XML existentes para a linguagem Java, em particular, JavaML e JavaML 2.0, que constituem a base para o projeto de AspectJML. A Seção 3.3 discute algumas representações existentes de código AspectJ em XML. A Seção 3.4 apresenta a linguagem AspectJML em detalhes. A Seção 3.5 descreve a implementação de ferramentas de suporte para AspectJML. Por fim, a Seção 0 apresenta as conclusões do capítulo.

3.2 Representações em XML para a Linguagem Java

Apresentaremos a seguir as principais representações em XML para a Linguagem Java, discutindo as características de cada uma.

3.2.1 JavaML

A linguagem de marcação JavaML [12] provê uma representação completa em XML para código fonte Java. Esta representação, diferentemente da representação clássica em texto plano, reflete a estrutura de programas Java diretamente na estrutura hierárquica do documento XML. Uma das principais contribuições de Badros ao projetar JavaML foi determinar um ponto intermediário entre a representação clássica do código fonte, em texto plano, e sua AST. Isto garantiu que JavaML não fosse demasiadamente descritiva (representação direta da AST) e nem exigisse nenhum tipo de análise sintática dos nós do documento XML resultante. A Figura 5 mostra um pequeno exemplo de código Java e a Figura 6 ilustra a sua representação correspondente em JavaML. O JavaML é mais que a estrutura do código fonte. Esta representação também deixa explicita as referências cruzadas dos elementos do código. Por exemplo, na Figura 6, no uso do parâmetro *g* na linha 13, o atributo *idref* do elemento *var-ref* referencia o elemento parâmetro, através do atributo *id*.

```
01 import java.applet.*;
02 import java.awt.*;
03 public class MyApplet extends Applet {
04      public void paint(Graphics g) {
05           g.drawString("3^(-2) = " + Math.pow(3,-2), 25, 50);
06      }
07 }
```

Figura 5 - Exemplo de código Java.

3.2.2 JavaML 2.0

A linguagem JavaML 2.0 [2], por sua vez, adiciona a JavaML informações léxicas e semânticas que originalmente foram deixadas de fora por Badros. Sobre o aspecto léxico, JavaML 2.0 preserva informações sobre formatação e comentários. Em relação às informações semânticas, esta versão implementa a referência cruzada de todos os símbolos do programa e inclui informações sobre as dependências de tipos externos. Devido a essas informações adicionais, a representação em JavaML 2.0 de um código Java tende a ser muito mais verbosa do que a representação correspondente em JavaML do mesmo código. Esta propriedade pode ser notada na Figura 7, que ilustra a representação em JavaML 2.0 do

código fonte mostrado na Figura 5. O JavaML 2.0 enriquece o JavaML com informações léxicas e semânticas, como pode-se observar no nó *<type-dependences>*, nó que contém informações sobre as dependências da unidade de compilação (linha 22 da Figura 7).

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <!DOCTYPE java-source-program SYSTEM "java-ml.dtd">
03 <java-source-program>
04 <java-class-file name="FirstApplet.java">
05 <import module="java.applet.*"/> <import module="java.awt.*"/>
06 <class name="FirstApplet" visibility="public">
07   <superclass name="Applet"/>
08   <method name="paint" visibility="public" id="FirstApplet_mth-15">
09     <type name="void" primitive="true"/>
10     <formal-arguments>
11       <formal-argument name="g" id="FirstApplet_frm-13"><type name="Graphics"/></formal-argument>
12     </formal-arguments> <block>
13       <send message="drawString"> <target><var-ref name="g" idref="FirstApplet_frm-13"/></target>
14           <arguments><binary-expr op="+"><literal-string value="3^(-2) = "/>
15               <send message="pow">
16                   <target><var-ref name="Math"/></target>
17                   <arguments><literal-number kind="integer" value="3"/><unary-expr op="-">
18                               <literal-number kind="integer" value="2"/></unary-expr></arguments>
19               </send>
20               </binary-expr><literal-number kind="integer" value="25"/><literal-number kind="integer" value="50"/>
21           </arguments>
22       </send> </block> </method> </class> </java-class-file>
23 </java-source-program>
```

Figura 6 - Representação do exemplo da Figura 5 em JavaML.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <java-source-program xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="javaml-2.xsd">
03 <java-class-file name="FirstApplet.java"> <import module="java.applet.*"/> <import module="java.awt.*"/>
04 <class name="FirstApplet" id="LFirstApplet;" idkind="type">
05   <modifiers> <modifier name="public"/> </modifiers>
06   <superclass name="Applet" idref="Ljava/applet/Applet;" idkind="type"/>
07     <method name="paint" id="LFirstApplet;paint(Ljava/awt/Graphics;)V" idkind="method">
08       <modifiers> <modifier name="public"/> </modifiers> <type name="void" primitive="true"/>
09       <formal-arguments>
10         <formal-argument name="g" id="LFirstApplet;arg15" idkind="formal"><type name="Graphics" idref="Ljava/awt/Graphics;"
11           idkind="type"/></formal-argument>
12       </formal-arguments>
13       <block><send message="drawString" idref="Ljava/awt/Graphics;drawString(Ljava/lang/String;II)V" idkind="method">
14           <target><formal-ref name="g" idref="LFirstApplet;arg15" idkind="formal"/></target>
15           <arguments><binary-expr op="+"><literal-string value="3^(-2) = "/><send message="pow" idref="Ljava/lang/Math;pow(DD)D"
16             idkind="method">
17               <target><type name="Math" idref="Ljava/lang/Math;" idkind="type"/></target>
18               <arguments><literal-number kind="integer" value="3"/><unary-expr op="-"><literal-number kind="integer" value="2"/>
19               </unary-expr></arguments> </send>
20           </binary-expr><literal-number kind="integer" value="25"/><literal-number kind="integer" value="50"/></arguments>
21       </send> </block> </method> </class> </java-class-file>
22  <type-dependences>
23       <type-dependence filename="FirstApplet.java" signature="LFirstApplet;">
24           <type-ref signature="Ljava/applet/Applet;"/><type-ref signature="Ljava/awt/Graphics;"/>
25           <type-ref signature="Ljava/lang/Math;"/><type-ref signature="Ljava/lang/Double;"/>
26           <type-ref signature="Ljava/lang/StringBuffer;"/>
27       </type-dependence>
28       <type-dependence filename="rt.jar/java/applet(Applet.class)" signature="Ljava/applet/Applet;"/>
29       <type-dependence filename="rt.jar/java/awt(Graphics.class)" signature="Ljava/awt/Graphics;"/>
30       <type-dependence filename="rt.jar/java/lang(Math.class)" signature="Ljava/lang/Math;"/>
31       <type-dependence filename="rt.jar/java/lang(Double.class)" signature="Ljava/lang/Double;"/>
32       <type-dependence filename="rt.jar/java/lang(StringBuffer.class)" signature="Ljava/lang/StringBuffer;"/>
33 </type-dependences> </java-source-program>
```

Figura 7 - Representação do exemplo da Figura 5 em JavaML 2.0.

3.2.3 XJava XML

O XJava XML é uma representação em XML produzida por um *parser* Java chamado BeaultyJ [13]. Diferentemente das representações mostradas anteriormente, XJava XML não representa a AST (arvore sintática) do código. Conforme ilustrado na Figura 8, o corpo dos métodos e construtores é armazenado em uma *tag* XML em texto plano (linha 12 da Figura 8). Mesmo assim, sua estrutura de marcadores e atributos é bastante simples, sendo bem parecida com a de JavaML, como pode ser observado na Figura 8.

```
01 <?xml version="1.0" encoding="UTF-8"?>
02 <xjava version="1.1">
03   <class name="MyApplet" public="yes" unqualifiedName="MyApplet">
04     <import kind="package">java.applet.*</import>
05     <import kind="package">java.awt.*</import>
06     <extends class="java.applet.Applet"/>
07     <method name="MyApplet.paint" public="yes" unqualifiedName="paint">
08       <type dimension="0" fullName="void" name="void" unqualifiedName="void"/>
09       <parameter name="g">
10         <type dimension="0" fullName="java.awt.Graphics" name="java.awt.Graphics" unqualifiedName="Graphics"/>
11       </parameter>
12       <code>g.drawString("3^(-2) = " + Math.pow(3,-2), 25, 50);</code>
13     </method>
14   </class>
15 </xjava>
```

Figura 8 - Representação em XJava da classe MyApplet exibida na Figura 5.

Assim como JavaML, esse padrão também representa cada elemento do código através de um marcador, com o nome dos elementos estando contido no atributo *name*. Os marcadores que representam pacotes, importações, classes, atributos, construtores e métodos possuem os mesmos nomes dos marcadores correspondentes em JavaML. Outra similaridade é que o tipo de retorno de métodos, bem como o tipo de dados de atributos e parâmetros, também é representado pelo marcador *type* e seu valor armazenado no atributo *name*.

XJava XML, diferentemente das duas representações JavaML, armazena todos os arquivos Java transformados em um só arquivo XML, organizando-os internamente por pacotes. Esta estrutura pode não comprometer projetos com poucas unidades de compilação, mas, para sistemas com muitos arquivos, essa representação pode não ser adequada, uma vez que pode gerar um documento XML extremamente grande e trazer problemas de processamento para as ferramentas de consulta e manipulação.

Um ponto positivo dessa representação é armazenar em XML os comentários feitos no código pelo programador. Além disso, esse padrão traz como vantagem a facilidade de especificar consultas que não dependam de informações contidas no código fonte, devido

ao seu esquema XML simplificado. No entanto, consultas que envolvam o corpo de métodos e construtores não serão atendidas. Como o corpo de métodos e construtores fica armazenado de forma textual no marcador *code*, não se pode realizar consultas sem utilizar pesquisa em texto, que vai de encontro aos benefícios oferecidos pela linguagem XML. Outra desvantagem de XJava XML é que ela também não armazena informações estruturais do código, como linhas, colunas, espaços em branco, e formatação em geral, prejudicando a conversão desse formato para a representação textual.

3.3 Representações em XML para Linguagens Orientadas a Aspectos

Algumas representações em XML para AOP são JBoss AOP [37], AspectWerkz [10] e AML [48]. No entanto, nenhum destes trabalhos visa especificamente utilizar o código fonte representado em XML para realizar atividades de análise estrutural e semântica nem implementar refatorações.

O JBoss AOP é um arcabouço AOP de propósito geral que visa suportar aplicações Java/J2EE comerciais. Além da capacidade de definir comportamentos entrecortantes, o produto oferece também a implementação de um conjunto de aspectos de sistema como *cache*, segurança e suporte a transações. O exemplo de código da Figura 9, extraído do Guia de Aplicação JBoss [38], especifica que o ponto de junção da chamada do método *Foo.fooMethod()* é interceptado pelo adendo *trace()*.

```
01 <aop>
02   <aspect class="FooAspect" scope="PER_VM"/>
03
04   <bind pointcut="execution(public String Foo->fooMethod(..))">
05     <advice name="trace" aspect="FooAspect"/>
06   </bind>
07 </aop>
```

Figura 9 – Exemplo de representação XML do JBOSS AOP.

O AspectWerkz é um arcabouço voltado para POA em Java. AspectWerkz permite definir aspectos através de documentos XML ou de *tags* JavaDoc. Ele foi incorporado ao AspectJ a partir da versão 5 deste último. A Figura 10 apresenta um exemplo da representação XML de um aspecto com conjuntos de junção (linhas 15 e 16) e adendos (linhas 13 e 14).

Por fim, o AML é um modelo independente de linguagem para construir linguagens orientadas a aspectos baseadas em XML. O AML provê *tags* para mapeamento de linguagens sem especificar detalhes relacionados à plataforma, linguagem base ou especificidade de implementação. Em linhas gerais, o AML abstrai detalhes da POA e suporta a implementação de extensões da linguagem para solucionar problemas específicos. A Figura 11 apresenta um exemplo de aspecto representado em AML com conjuntos de junção (linhas 03 a 11) e um adendo (linha 13).

```
01 <aspectwerkz>
02   <system id="sample">
03     ...
04     <deployment-scope name="mutliLineScope">
05       within(com.DynamicAOP)
06     </deployment-scope>
07     <package name="examples">
08       <aspect class="caching.CachingAspect" deployment-model="perInstance">
09         <param name="timeout" value="10"/>
10         <introduce class="IntroductionTestAspect$MarkerInterface" bind-to="within(test.mixin.perinstance.ToBeIntroduced)"/>
11         <pointcut name="callee" expression="execution(int examples.caching.Pi.getPiDecimal(int))"/>
12         <pointcut name="caller" expression="call(int examples.caching.Pi.getPiDecimal(int)) && within(examples.caching.*)"/>
13         <advice name="invocationCounter" type="before" bind-to="caller"/>
14         <advice name="cache" type="around" bind-to="callee"/>
15       </aspect>
26     </package>
17   </system></aspectwerkz>
```

Figura 10 – Exemplo de representação XML do AspectWerkz.

```
01 <aml>
02   <aspect id="History">
03     <pointcut id="update">
04       <or>
05         <pointcut type="method-call"  pattern="* *.move(..)"/>
06         <pointcut type="method-call"  pattern="* Rec*.setTopLeft(..)"/>
07         <pointcut type="method-call" pattern="* Rec*.setBottomRight(..)"/>
08         <pointcut type="method-call" pattern="* Circle.setOrigin(..)"/>
09         <pointcut type="method-call" pattern="* Circle.setRadius(..)"/>
10       </or>
11     </pointcut>
12     <interceptor class = "HistoryInterceptor">
13       <advice type = "after" pointcut-refid = "update" interceptor-method = "public void beforeUpdate()"/>
14     </interceptor>
15   </aspect>
16 </aml>
```

Figura 11 – Exemplo de representação XML do AML.

As propostas de representação XML acima visam aumentar o nível de abstração necessário para programar em linguagens OA. No entanto, elas não contêm a estrutura sintática completa do código, propriedades oferecidas pelo JavaML e JavaML 2.0. A ausência desta estrutura exige a análise léxica do conteúdo das *tags* para realização de análises sintáticas e semânticas. A necessidade desta análise léxica pode ser observada, por exemplo, na Figura 11, na *tag pattern* (linha 06). Para identificar os pontos de atuação deste conjunto de junção, é preciso analisar lexicamente o conteúdo desta *tag*.

3.4 A Linguagem de Marcação AspectJML

A linguagem AspectJML estende a linguagem JavaML 2.0 com suporte às estruturas sintáticas particulares da linguagem AspectJ. Abaixo apresentamos os principais elementos que compõem o esquema de AspectJML, e em seguida mostramos um exemplo de código AspectJ e partes de sua representação em AspectJML.

3.4.1 Esquema

O esquema de AspectJML foi definido através da evolução direta do esquema de JavaML 2.0. Conforme apresentado em [30], a linguagem AspectJ é composta por três diferentes (sub)linguagens: (1) a linguagem Java original; (2) uma linguagem de declaração de aspectos; e (3) uma linguagem de definição de conjuntos de junção. Cada uma dessas linguagens tem sua própria estrutura léxica e sintática. Neste sentido, o esquema de AspectJML foi construído com base no esquema definido em JavaML 2.0, e nas gramáticas das linguagens de declaração de aspectos e de definição de expressões de conjuntos de junção.

A primeira decisão de projeto na construção do esquema foi determinar, no esquema de JavaML 2.0, onde acrescentar os elementos relativos as declarações de aspectos e as definições de conjuntos de junção. Em seguida, determinou-se como representar aspectos e conjuntos de junção em XML, gerando a hierarquia de elementos de AspectJML. Para manter a compatibilidade com a representação de Java em JavaML 2.0, foram criados, em AspectJML, os elementos *aspectj-source-program* e *aspectj-file*, similares aos elementos *java-source-program* e *java-class-file*, de JavaML 2.0.

3.4.1.1 Declaração de aspectos

A definição de um aspecto representa a segunda linguagem que compõe o AspectJ. De acordo com a gramática de AspectJ, um aspecto pode ser declarado de duas maneiras: (1) como uma declaração de tipo, ligado diretamente a uma unidade de compilação; ou (2) como um membro de uma classe, de uma interface ou de outro aspecto. Para AspectJML suportar a declaração de aspectos no caso (1), é suficiente incluir um elemento do tipo *aspect* como um sub-elemento do elemento *aspectj-file*. No caso (2), basta adicionar um elemento *aspect* como um sub-elemento dos elementos *class*, *interface* e do próprio *aspect*.

O elemento *aspect* é composto pelos seguintes tipos de elemento: *modifiers*, *interface*, *perclause*, *aspect*, *class*, *method*, *declare*, *advice*, *pointcut*, *field*, *intertype-member-*

declaration e um elemento *superaspect* ou *superclass*. O elemento *superaspect*, a exemplo do elemento *superclass*, representa a relação de herança e é uma simples referência a um aspecto. O elemento *intertype-member-declaration*, linha 22 da Figura 12, define um método ou atributo em uma determinada classe. Detalharemos a seguir os elementos mais relevantes da representação. A Figura 12 ilustra o código em XML Schema [72] do elemento *aspect*.

Os elementos *modifiers* (linha 05), *interface* (linha 15), *class* (linha 14) e *method* (linha 16) fazem parte da especificação original de Java e, portanto, já estão definidos no esquema de JavaML 2.0. O elemento *advice* (linha 19), referente a um adendo em AspectJ, é composto por elementos do tipo *throws*, *pointcut-expr*, *block*, *formal-arguments* e um *type*. Os elementos *pointcut* (linha 20) e *pointcut-expr* fazem parte da linguagem de declaração de conjuntos de junção e serão detalhados a seguir.

```
01 <xs:element name="aspect">
02   <xs:complexType mixed="true">
03     <xs:sequence>
04       <xs:element ref="doc-comment" minOccurs="0"/>
05       <xs:element ref="modifiers" minOccurs="0"/>
06       <xs:element ref="perclause" minOccurs="0" maxOccurs="1"/>
07       <xs:choice minOccurs="0" maxOccurs="1">
08         <xs:element ref="superaspect" minOccurs="0"/>
09         <xs:element ref="superclass" minOccurs="0"/>
10       </xs:choice>
11       <xs:element ref="implement" minOccurs="0" maxOccurs="unbounded"/>
12       <xs:choice minOccurs="0" maxOccurs="unbounded">
13         <xs:element ref="aspect"/>
14         <xs:element ref="class"/>
15         <xs:element ref="interface"/>
16         <xs:element ref="method"/>
17         <xs:element ref="constructor"/>
18         <xs:element ref="declare"/>
19         <xs:element ref="advice"/>
20         <xs:element ref="pointcut"/>
21         <xs:element ref="field"/>
22         <xs:element ref="intertype-member-declaration"/>
23       </xs:choice>
24     </xs:sequence>
25     <xs:attribute name="name" type="xs:string" use="required"/>
26     <xs:attribute name="privileged" type="xs:boolean" default="false"/>
27     <xs:attributeGroup ref="location"/>
28     <xs:attribute name="id" type="xs:string"/>
29   </xs:complexType>
30 </xs:element>
```

Figura 12 – Representação da estrutura do elemento *aspect* em XML Schema.

3.4.1.2 Definição de conjuntos de junção

De maneira análoga ao elemento *aspect*, um conjunto de junção pode ser definido como membro de uma classe, de uma interface ou de um aspecto. Desta forma, para AspectJML suportar a definição de conjuntos de junção, basta adicionar um elemento do tipo *pointcut*, que representará um conjunto de junção em AspectJML, com sub-elemento dos

elementos class, interface e aspect. O elemento *pointcut* é composto por elementos do tipo *modifiers*, *formal-arguments* e *pointcut-expr*. Além destes elementos, um conjunto de junção também tem os atributos *name* e *id*.

A estrutura do elemento *pointcut-expr* compreende a terceira linguagem que compõe o AspectJ. São expressões de conjuntos de junção os elementos iniciados pelas palavras reservadas *call*, *execution*, *initialization*, *preinitialization*, *staticinitialization*, *get*, *set*, *handler*, *adviceexecution*, *within*, *withincode*, *cflow*, *cflowbelow*, *if*, *this*, *target*, *args*, uma referência a um conjunto de junção existente, e uma expressão de conjunto de junção entre parênteses. Estes 19 tipos de expressões de conjuntos de junção foram classificados em 9 elementos do AspectJML. São eles: *if*, *method-constructor-pointcut-expr*, *constructor-pointcut-expr*, *classname-pointcut-expr*, *field-pointcut-expr*, *pointcut-pointcut-expr*, *formal-argument-pointcut-expr*, *formal-arguments-pointcut-expr* e *advice-execution-expr*. Esta classificação agrupou elementos que necessitavam informações da mesma natureza. Por exemplo, os elementos *cflow* e *cflowbelow* foram representados pelo mesmo elemento por necessitarem de uma expressão de conjunto de junção para completar sua definição. A categorização realizada e as informações requeridas por cada tipo são mostradas na Tabela 1.

Elementos de AspectJ	Elementos de AspectJML	Informações Necessárias
If	If	Expressão binária do Java
call, execution, withincode	method-constructor-pointcut-expr	Padrão de construtor ou padrão de método
initialization, preinitialization	constructor-pointcut-expr	Padrão de construtor
staticinitialization, handler, within	classname-pointcut-expr	Padrão de nome de classe
get, set	field-pointcut-expr	Padrão de atributo
cflow, cflowbelow e expressão de pointcut entre parênteses	pointcut-pointcut-expr	Expressão de pointcut
this, target	formal-argument-pointcut-expr	Padrão de parâmetro específico de conjuntos de junção
args, referência a um pointcut existente	formal-arguments-pointcut-expr	Padrão de lista de parâmetros específicos de conjuntos de junção
adviceexecution	advice-execution-expr	Não exige informações complementares

Tabela 1 – Categorização dos tipos de expressões de conjuntos de junção em AspectJML.

Conforme exibido na coluna “Informações Requeridas” da Tabela 1, expressões de conjuntos de junção exigem padrões de elementos de linguagem. Para tratar esta questão, foram criados diversos tipos de padrões de elementos em AspectJML. Para exemplificar a definição dos tipos de padrões de elementos, será mostrada, a seguir, a estrutura do *class-type-*

dot-id, elemento que compõe a definição das expressões dos conjuntos de junção *method-constructor-pointcut-expr* e *field-pointcut-expr*.

O elemento *class-type-dot-id* é composto por um padrão de tipo de dado seguido de um padrão de identificador. O padrão de tipo de dado pode ser um simples padrão de nome (*name-pattern*) ou uma expressão de padrão de tipo (*type-patt-expr*) seguido de um padrão de nome simples (representado pelo atributo *simple-name-patt*). Além destes elementos, há também um qualificador de padrão (representado pelo atributo *qualifier*) que, neste caso, pode assumir os seguintes valores: '.', '+' e '..'. A definição do elemento *class-type-dot-id* baseou-se na expressão regular que determina o elemento *classtype_dot_id*, da linguagem AspectJ, ilustrada na Figura 13.

```
<clastype_dot_id> ::=
        <simple_name_pattern>
      | <name_pattern> '.' <simple_name_pattern>
      | <name_pattern> '+' '.' <simple_name_pattern>
      | <name_pattern> '..' <simple_name_pattern>
      | (<type_pattern_expr>) '.' <simple_name_pattern>
```

Figura 13 - Estrutura do elemento *classtype_dot_id*.

3.4.2 Exemplo

A Figura 14 mostra um exemplo bastante simples de um aspecto definido em AspectJ, denominado *ChecaExpoente*. Este aspecto tem como única finalidade alterar o comportamento do método *Math.pow* (que calcula a potência entre dois valores do tipo *double* passados como parâmetro) para não calcular potências de expoentes negativos. A Figura 15 mostra a representação em AspectJML do código exibido na Figura 14.

```
01  public aspect ChecaExpoente extends Object{
02    pointcut potenciacaopc(double arg0, double arg1):
03          args(arg0, arg1) && (call (double Math.po*(doub*, *)));
04    double around(double arg0, double arg1): potenciacaopc(arg0, arg1) {
05        if (arg1 < 0) {
06          System.out.println("Expoente negativo será considerado zero.");
07          return 1.; //Considera expoente arg1 = 0
08        }
09       else
10          return proceed(arg0, arg1);
11    }
12  }
```

Figura 14 - Exemplo de um aspecto em AspectJ.

```
01  <aspectj-source-program
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="apectjavaml-2.V.17.6.xsd">
02    <aspectj-file name=" ChecaExpoente.aj">
03      <aspect id="LChecaExpoente;" idkind="type"
name="ChecaExpoente">
04        <modifiers> <modifier name="public"/> </modifiers>
05        <superclass idkind="type" idref="Ljava/lang/Object;"
name="Object"/>
06        <constructor id="LChecaExpoente;&amp;lt;init&amp;gt;()V"
idkind="constructor" name="ChecaExpoente">
07          <modifiers> <modifier name="public"/> </modifiers>
08          <block> <super-call> <arguments/> </super-call> </block>
09        </constructor>
10        <pointcut id="LChecaExpoente;potenciacaopc" idkind="pointcut"
name="potenciacaopc">
11          <formal-arguments>
12            <formal-argument id="LChecaExpoente;arg9097070"
idkind="formal" name="arg0">
13              <type name="double" primitive="true"/>
14            </formal-argument>
15            <formal-argument id="LChecaExpoente;arg20474136"
idkind="formal" name="arg1">
16              <type name="double" primitive="true"/>
17            </formal-argument>
18          </formal-arguments>
19          <binary-pointcut-expr idkind="pointcut-expr"
op="&amp;amp;&amp;amp;">
20            <formal-arguments-pointcut-expr idkind="pointcut-expr"
type="args">
21              <formal-arguments-pointcut-patt>
22                <formal-argument-pointcut-patt>
23                  <type idkind="type" name="double"/>
24                </formal-argument-pointcut-patt>
25                <formal-argument-pointcut-patt>
26                  <type idkind="type" name="double"/>
27                </formal-argument-pointcut-patt>
28              </formal-arguments-pointcut-patt>
29            </formal-arguments-pointcut-expr>
30            <method-constructor-pointcut-expr idkind="pointcut-expr"
type="method-call">
31              <method-patt>
32                <type idkind="type" name="double"/>
33                <class-type-dot-id simple-name-patt="po*">
34                  <type idkind="type" name="Math"/>
35                </class-type-dot-id>
36                <formal-arguments-patt>
37                  <formal-argument-patt>
38                    <name-patt simple-name-patt="doub*"/>
39                  </formal-argument-patt>
40                  <formal-argument-patt>
41                    <name-patt simple-name-patt="*"/>
42                  </formal-argument-patt>
43                </formal-arguments-patt>
44              </method-patt>
45            </method-constructor-pointcut-expr>
46          </binary-pointcut-expr>
47        </pointcut>
48        <advice
id="LChecaExpoente;ajc$around$ChecaExpoente$1$28e821df(DDLorg/as
pectj/runtime/internal/AroundClosure;)D" idkind="advice" kind="around"
name="ajc$around$ChecaExpoente$1$28e821df">
49          <formal-arguments-pointcut-expr idkind="pointcut-expr"
type="pointcut-ref">
50            <pointcut-ref idref="LChecaExpoente;potenciacaopc"
name="potenciacaopc"/>
51            <formal-arguments-pointcut-patt>
52              <formal-argument-pointcut-patt>
53                <type idkind="type" name="double"/>
54              </formal-argument-pointcut-patt>
55              <formal-argument-pointcut-patt>
56                <type idkind="type" name="double"/>
57              </formal-argument-pointcut-patt>
58            </formal-arguments-pointcut-patt>
59          </formal-arguments-pointcut-expr>
60          <type name="double" primitive="true"/>
61          <formal-arguments>
62            <formal-argument id="LChecaExpoente;arg8106640"
idkind="formal" name="arg0">
63              <type name="double" primitive="true"/>
64            </formal-argument>
65            <formal-argument id="LChecaExpoente;arg17320380"
idkind="formal" name="arg1">
66              <type name="double" primitive="true"/>
67            </formal-argument>
68          </formal-arguments>
69          <block>
70            <if>
71              <test>
72                <binary-expr op="&amp;lt;">
73                  <formal-ref idkind="formal"
idref="LChecaExpoente;arg17320380" name="arg1"/>
74                  <literal-number kind="integer" value="0"/>
75                </binary-expr>
76              </test>
77              <true-case> <block>
78                <send idkind="method"
idref="Ljava/io/PrintStream;println(Ljava/lang/String;)V" message="println">
79                  <target>
80                    <field-ref idkind="field" idref="Ljava/lang/System;out"
name="out"/>
81                  </target>
82                  <arguments>
83                    <literal-string value="Expoente negativo ser
considerado zero."/>
84                  </arguments> </send>
85                <return>
86                  <literal-number kind="double" value="1."/>
87                </return>
88              </block>
89              </true-case>
90              <false-case>
91                <return>
92                  <send idkind="method"
idref="LChecaExpoente;ajc$around$ChecaExpoente$1$28e821dfproceed(
DDLorg/aspectj/runtime/internal/AroundClosure;)D" message="proceed">
93                    <target> <this/> </target>
94                    <arguments>
95                      <formal-ref idkind="formal"
idref="LChecaExpoente;arg8106640" name="arg0"/>
96                      <formal-ref idkind="formal"
idref="LChecaExpoente;arg17320380" name="arg1"/>
97                    </arguments>
98                  </send> </return> </false-case> </if> </block>
99        </advice>
100       </aspect>
101     </aspectj-file>
102     <type-dependences>
103       <type-dependence filename="C:\AspectJML\ChecaExpoente.aj"
signature="LChecaExpoente;">
104         <type-ref signature="LChecaExpoente;"/>
105         <type-ref signature="Ljava/io/PrintStream;"/>
106       </type-dependence>
107       <type-dependence filename="java/io/PrintStream"
signature="Ljava/io/PrintStream;"/>
108     </type-dependences>
109   </aspectj-source-program>
```

Figura 15 - Representação em AspectJML do Aspecto apresentado na Figura 14.

O conjunto de junção *potenciacaopc* está representado na Figura 15 pelas linhas 10 a 47. A representação do adendo está entre as linhas 48 a 99 da Figura 15. Outra característica importante do AspectJML pode ser observada entre as linhas 102 e 108 da Figura 15. São as referências cruzadas desta unidade de compilação. Este aspecto tem apenas uma referencia externa, a classe java.io.PrintStream. Observe que a aplicação do aspecto definido na Figura 14 ao código exibido na Figura 5 alteraria o comportamento do programa.

3.5 Ferramentas de Suporte

Como o AspectJML foi projetado para ser manipulado por ferramentas, é necessário que ele tenha ferramentas automáticas de conversão (de AspectJ para AspectJML) e de reversão (de AspectJML para AspectJ). O conversor de AspectJ para AspectJML foi construído alterando o compilador AspectJ ajc [7], versão 1.5. O reversor de AspectJML para AspectJ foi construído em XSLT, a partir do reversor distribuído junto com o JavaML. Detalharemos, a seguir, a construção do conversor, parte do trabalho desta dissertação, e do reversor, descrito originalmente em [57].

3.5.1 Conversor

Diferentemente do conversor do JavaML 2.0, que pôde ser construído estendendo o conversor do JavaML, que, por sua vez, foi implementado modificando o compilador Java da IBM, Jikes [39], o conversor do AspectJML não pôde evoluir a versão anterior existente, por o Jikes não suportar AspectJ.

Dado que o projeto de construção do conversor para AspectJML teria que iniciar do zero, o primeiro passo para a sua construção foi a seleção do compilador AspectJ que seria alterado. Foram avaliados o *ajc* [7] e o *abc* [1]. Apesar de o compilador *abc* ter seu projeto de software voltado para extensões e gerar binários com melhor desempenho [11], decidimos utilizar o *ajc* por ele ser o compilador oficial do AspectJ e ter um plano estabelecido de atualizações e correções. A Tabela 2 ressalta as principais vantagens e desvantagens identificadas em cada compilador. A seguir, detalharemos o processo de desenvolvimento do conversor AspectJML, a partir das modificações realizadas no compilador *ajc*.

3.5.1.1 Levantamento de Requisitos Funcionais e Suplementares

Identificaram-se dois requisitos funcionais e dois requisitos suplementares no projeto de desenvolvimento do conversor AspectJML. O estabelecimento destes requisitos visou manter a maior proximidade possível com a linguagem JavaML 2.0 e facilitar a manutenção do código produzido.

Compilador	**Vantagens**	**Desvantagens**
ajc	Menor tempo de compilação.	Não é construído sobre um arcabouço de compilação extensível.
	É o compilador "oficial" de AspectJ.	
abc	Construído sobre um arcabouço extensível, o Polyglot [60].	Maior tempo de compilação.
	Projetado para construção de extensões.	Diferenças em relação ao ajc.

Tabela 2 - Vantagens e desvantagens dos compiladores AspectJ ajc e abc.

Os dois requisitos funcionais identificados foram: (1) construir representações em AspectJML para os elementos da linguagem AspectJ, conforme especificado na seção 3.2.3; e (2) manter para a linguagem Java a mesma representação XML existente em JavaML 2.0, acrescendo elementos para os modificadores adicionados à linguagem Java após a confecção do conversor JavaML 2.0.

Os dois requisitos suplementares identificados foram: (1) permitir que o conversor AspectJML seja facilmente migrado para novas versões do *ajc*; e (2) tentar não alterar o código fonte das classes do compilador *ajc*.

3.5.1.2 Análise

O desdobramento da análise da mudança no compilador ajc indicou a necessidade de três classes e um aspecto. A primeira, chamada *XMLGenerator*, é responsável por armazenar e gerar o documento XML com a representação do código fonte na formatação estabelecida. A segunda classe, chamada *AJML*, é responsável por navegar na árvore sintática da linguagem Java e nos nós de declaração de aspectos, adendos, declarações (*declares*) e conjuntos de junção da unidade de código e produzir o documento DOM [33] correspondente. A terceira classe, chamada *AJMLPatternNode*, é responsável por navegar na árvore sintática da linguagem de definição de expressões de conjuntos de junção e produzir os nós (elementos) DOM correspondentes. Por fim, o aspecto *AJMLCompilerAdapter* é responsável

por interceptar o método *org.aspectj.ajdt.internal.compilerAjCompilerAdapter.afterProcessing*, passo do compilador que já tem a árvore sintática montada, e acionar o método *visit* da unidade de compilação usando a classe *AJML*, que cria um documento XML correspondente a árvore sintática da unidade de compilação processada. A Figura 16 ilustra o modelo de análise do conversor AspectJML.

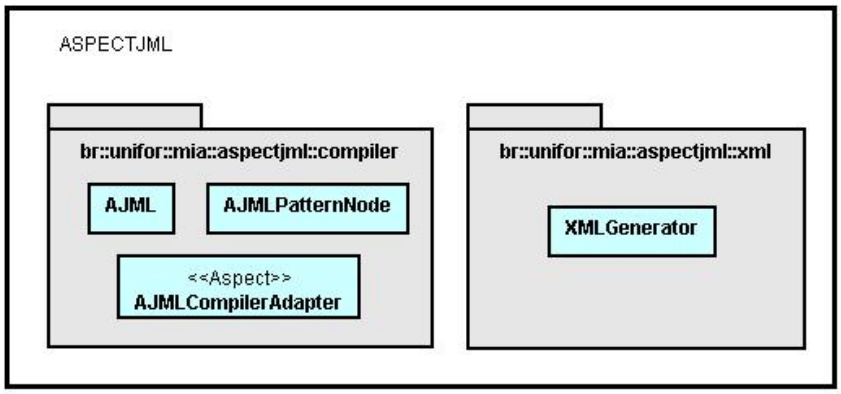

Figura 16 - Modelo de análise do conversor AspectJML

3.5.1.3 Projeto

O projeto do conversor buscou ser o menos intrusivo possível ao código fonte do ajc. O compilador ajc é composto por 24 projetos Eclipse [18]. A idéia inicial de construção do conversor AspectJML foi adicionar a estes projetos um outro projeto Eclipse chamado *aspectjml*, que tivesse as classes e aspectos necessários para produzir a representação de código em XML. No entanto, por conta de dependências entre projetos geradas, ainda foi necessário estender alguns dos projetos existentes do ajc. A fase de projeto identificou a necessidade de criação de mais duas classes e uma interface, deixando a solução final com cinco classes, um aspecto e uma interface.

A classe de análise *XMLGenerator* foi desdobrada em duas, a *XMLGenerator* e a *AJMLDocumentFactory*. A *XMLGenerator* foi projetada com apenas um único método estático que recebe um objeto do tipo documento DOM e gera sua representação em XML. A outra classe, AJMLDocumentFactory, ficou responsável por fabricar e manter documentos DOM. Esta classe implementa os padrões de projeto *singleton* [23] e *factory* [23], e disponibiliza para o resto da aplicação um objeto do tipo documento DOM para o compilador. Estas classes pertencem ao pacote *br.unifor.mia.aspectjml.xml* do projeto Eclipse aspectjml, criado especificamente para o conversor.

A classe AJML foi projetada como a implementação do padrão de projeto *Visitor* [63] e teve seu nome alterado para *AJMLVisitor*. Ela herda indiretamente da classe *ASTVisitor* do pacote *org.aspectj.org.eclipse.jdt.internal.compiler*. Para implementar esta herança, foi criada uma classe abstrata intermediária chamada *AbstractAJMLVisitor*, que adiciona métodos de navegação no documento DOM. Esta classe abstrata também é necessária para viabilizar a interação mútua entre a classe *AJMLVisitor* e a classe *AJMLPatternNode*, que fazem parte de projetos Eclipse distintos. A função da classe *AJMLVisitor* consiste em montar os elementos XML correspondentes para cada nó da árvore sintática da linguagem Java, de declaração de aspectos, adendos, declarações e conjuntos de junção.

A classe *AJMLPatternNode* também foi projetada como a implementação do padrão de projeto *Visitor* e teve seu nome alterado para *AJMLPatternNodeVisitor*. Ela implementa a interface *PatternNodeVisitor* do pacote *org.aspectj.weaver.patterns*. No entanto, a implementação desta interface não é direta. Criou-se uma interface intermediária chamada *AJMLPatternNodeVisitorInterface* para permitir a interação da classe *AJMLVisitor* com a classe *AJMLPatternNodeVisitor*. A implementação da classe *AJMLPatternNodeVisitor* consiste em, para cada nó da árvore sintática, montar os elemento XML correspondentes.

O aspecto *AJMLCompilerAdapter* foi criado para interceptar a execução do passo imediatamente anterior à aplicação da mesclagem do código AspectJ no código Java do compilador ajc. Este aspecto contém um adendo do tipo *before* que intercepta o método *AjCompilerAdapter.afterProcessing* do pacote *org.aspectj.ajdt.internal.compiler*. O método interceptado tem um argumento chamado *unit* que contém a unidade de compilação processada. O adendo cria um objeto chamado *visitor* da classe *AJMLVisitor*; executa o método *traverse* do argumento *unit*, passando objeto *visitor* como parâmetro, e converte o documento DOM produzido em um documento XML, utilizando a classe *XMLGenerator*.

A Figura 17 ilustra o diagrama de classes de projeto do conversor AspectJML. A construção deste conversor alterou dois dos 24 projetos Eclipse que compõem o compilador ajc, *weaver* e *org.aspectj.ajdt.core*, e acrescentou um novo projeto, *aspectjml*. Em cada um destes projetos foi adicionado o pacote *br.unifor.mia.aspectjml.compiler*. Além disso, também se adicionou o pacote *br.unifor.mia.aspectjml.xml* no projeto *aspectjml*. Estes pacotes estão ilustrados em cinza na Figura 17. A Figura 17 também apresenta em verde os pacotes existentes no compilador ajc que foram modificados ou referenciados pelas classes do

conversor. As classes criadas pelo conversor AspectJML estão indicadas em azul e as classes existentes do compilador ajc estão hachuradas em amarelo.

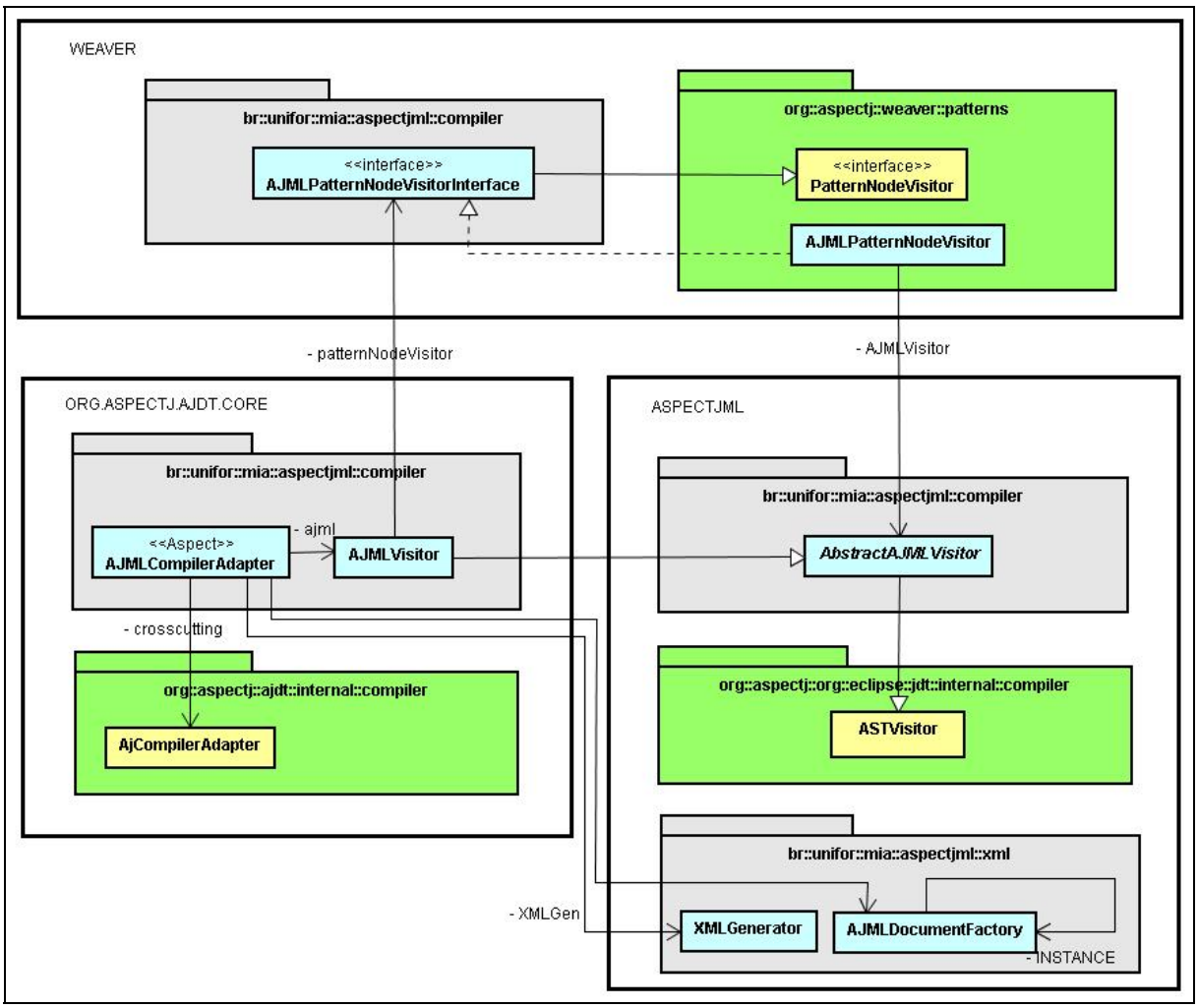

Figura 17 - Modelo de projeto conversor do AspectJML

3.5.1.4 Desenvolvimento

A implementação do conversor *AspectJML* adicionou pouco mais de 3300 linhas de código à implementação original do compilador ajc, distribuídas em 5 classes, 1 interface e 1 aspecto, explanados acima. A Tabela 3 mostra a quantidade de linhas de código de cada um destes elementos. Comentaremos, a seguir, a construção das 2 maiores classes deste conversor, *AJMLVisitor* e *AJMLPatternNodeVisitor*.

3.5.1.4.1 Classe AJMLVisitor

Conforme exposto anteriormente, a função da classe *AJMLVisitor* é produzir nós XML para as estruturas da linguagem Java e para as estruturas aspectos, adendos, declarações

e conjuntos de junção. Para cada nó da árvore sintática, este método extrai informações do nó e grava no documento DOM que representa a unidade de compilação. Isso é feito com o uso do padrão de projeto *Visitor*.

Unidade de Compilação	**Qtd. Linhas**
AbstractAJMLVisitor.java	16
AJMLPatternNodeVisitorInterface.java	30
AJMLCompilerAdapter.aj	32
XMLGenerator.java	42
AJMLDocumentFactory.java	44
AJMLPatternNodeVisitor.java	688
AJMLVisitor.java	2.459
Total	**3.311**

Tabela 3 – Quantitativo de linhas de código por unidade de compilação

O padrão de projeto *Visitor* é empregado quando se deseja visitar uma estrutura de entidades em forma de árvore. Para implementá-lo, cada nó da árvore deve conter um método para atravessar o nó, normalmente chamado *traverse*. A primeira instrução deste método é acionar um método da classe visitante, normalmente chamado *visit*. Dependendo do retorno do método *visit*, ele aciona ou não os métodos atravessar (*traverse*) dos seus nós filho. Assim, com as chamadas sucessivas de métodos 'atravessar' e 'visitar', a classe visitante percorre a estrutura de árvore obtendo as informações desejadas.

No ajc, cada nó da AST da linguagem Java e da declaração de aspetos, adendos, declarações e conjuntos de junção é representado por uma classe Java que implementa o padrão de projeto *Visitor*. Desta forma, a implementação da classe *AJMLVisitor* consiste em construir métodos *visit* que extraiam as informações dos nós da AST e adicionem em um documento DOM. Houve dois tipos de implementação do método *visit*. Explanaremos a seguir cada um.

No primeiro tipo, o corpo do método *traverse* do nó é compatível com a estrutura do AspectJML. Ou seja, a seqüência de acionamento dos métodos *traverse* dos nós filhos é a mesma do esquema XML da linguagem AspectJML. Neste caso, a construção do método *visit* resumiu-se a adicionar informações do nó corrente no documento DOM. A Figura 18 apresenta a implementação do método *visit* para os elementos 'argumento' (*argument*) e 'expressão binária ou' (*OR_OR_Expression*). Nestes dois casos, as informações dos nós

foram adicionadas ao documento DOM e pôde-se continuar navegando pela AST a partir do método *traverse* implementado nas classes *argument* e *OR_OR_Expression*. O retorno verdadeiro destes métodos indica que o método *traverse* deve processar os nós filhos normalmente.

```
01  public boolean visit(Argument argument, BlockScope scope) {
02
03    AbstractMethodDeclaration lMethod;
04    this.createVisitorElement("formal-argument");
05    currentNode.setAttribute("name", new String(argument.name));
06    lMethod = ((AbstractMethodDeclaration)argument.binding.declaringScope.referenceContext());
07    currentNode.setAttribute("id", new String(lMethod.binding.declaringClass.signature()) +
08                          "arg" + argument.binding.hashCode());
09    currentNode.setAttribute("idkind", "formal");
10    return true; // keep traversing
11  }
12
13  public boolean visit(OR_OR_Expression or_or_Expression, BlockScope scope) {
14    this.createVisitorElement("paren");
15    this.createBinaryExpressionElement("||");
16    return true; // keep traversing
17  }
```

Figura 18 – Implementação do método *visit* para os elementos *argument* e *OR_OR_Expression*

No segundo tipo, o corpo do método *traverse* do nó é incompatível com a estrutura do AspectJML. Neste caso, o método *visit* deve adicionar informações do nó corrente no documento DOM e coordenar o acionamento do método *traverse* dos nós filhos do nó processado. A implementação do método *visit* do nó 'atributo' (*field*) apresentada na Figura 20 enquadra-se neste caso. A estrutura de árvore implementada para este nó exibida na Figura 19 é incompatível com a estrutura do AspectJML por causa do nó *var-initializer* do AspectJML. Por isso, o retorno deste método *visit* é falso e o próprio método *visit* fica encarregado de acionar os métodos *traverse* dos filhos deste nó.

```
01  public void traverse(ASTVisitor visitor, MethodScope scope) {
02
03    if (visitor.visit(this, scope)) {
04      if (this.annotations != null) {
05        int annotationsLength = this.annotations.length;
06        for (int i = 0; i < annotationsLength; i++)
07          this.annotations[i].traverse(visitor, scope);
08      }
09      if (this.type != null) {
10        this.type.traverse(visitor, scope);
11      }
12      if (this.initialization != null)
13        this.initialization.traverse(visitor, scope);
14    }
15    visitor.endVisit(this, scope);
16  }
```

Figura 19 - Implementação do método *traverse* da classe FieldDeclaration

```
01  public boolean visit(FieldDeclaration fieldDeclaration, MethodScope scope) {
02     this.createVisitorElement("field");
03     this.currentNode.setAttribute("name", new String(fieldDeclaration.name));
04     this.currentNode.setAttribute("id", getFieldDeclarationId(fieldDeclaration.binding));
05     this.currentNode.setAttribute("idkind", "field");
06     if (fieldDeclaration.modifiers != 0)
07        this.currentNode.appendChild(this.createModifiersElement(fieldDeclaration.modifiers));
08     if (fieldDeclaration.annotations != null) {
09        int annotationsLength = fieldDeclaration.annotations.length;
10        for (int i = 0; i < annotationsLength; i++)
11           fieldDeclaration.annotations[i].traverse(this, scope);
12     }
13     if (fieldDeclaration.type != null) {
14        fieldDeclaration.type.traverse(this, scope);
15     }
16     if (fieldDeclaration.initialization != null) {
17        this.createVisitorElement("var-initializer");
18        fieldDeclaration.initialization.traverse(this, scope);
19        this.getParent();
20     }
21     return false; // do nothing by default, keep traversing
22  }
```

Figura 20 - Implementação do método *visit* para o elemento *field* da classe AJMLVisitor

3.5.1.4.2 Classe AJMLPatternNodeVisitor

A implementação da classe AJMLPatternNodeVisitor foi similar à construção de AJMLVisitor. No entanto, os elementos tratados por esta classe são as expressões de conjunto de junção. A Figura 21 ilustra a implementação dos métodos *visit* dos nós 'expressão de conjunto de junção binária *e*' (*AndPointcut*) e *CflowPointcut* da classe AJMLPatternNodeVisitor.

```
01  public Object visit(AndPointcut node, Object data) {
02     Element currentNode;
03     currentNode = this.createVisitorElement("binary-pointcut-expr", data);
04     currentNode.setAttribute("op", "&&");
05     currentNode.setAttribute("idkind", "pointcut-expr");
06     return currentNode;
07  }
08
09  public Object visit(CflowPointcut node, Object data) {
10     Element currentNode;
11     currentNode = this.createVisitorElement("cflow-pointcut-expr", data);
12     currentNode.setAttribute("below", + new Boolean(node.isCflowBelow()).toString());
13     currentNode.setAttribute("idkind", "pointcut-expr");
14     return currentNode;
15  }
```

Figura 21 – Implementação do método *visit* para os elementos *AndPointcut* e *CflowPointcut*

3.5.2 Reversor

O reversor de AspectJML para Java [57] foi construído utilizando a tecnologia de transformação XSLT [73]. Um dos fortes motivos para esta escolha foi a facilidade de manuseio que XSLT oferece sobre XML. O reversor foi criado com base no reversor proposto

por Badros para a linguagem JavaML [12], acrescentando novos *templates* para transformação de estruturas específicas de AspectJ.

A implementação do reversor consiste de quase 100 *templates* XSLT e aproximadamente 800 linhas de código. O primeiro *template* a ser executado é o que faz ligação com o nó <*aspectj-source-program*>. Todo documento AspectJML iniciará com este nó, pois ele é a raiz do documento XML. A execução deste *template* desencadeia a execução de todos os outros *templates* de transformação. Basicamente, esse *template* executa um laço com a instrução <*xsl:for-each*>, passando por todos os elementos filhos do nó raiz. Dentro do corpo desta instrução é inserido um comentário com o nome do arquivo AspectJ original, apenas a título informativo. Para isso, são executadas as seguintes instruções: <*xsl:text*> com o texto "//FILE:"; <*xsl:value-of*>, que seleciona e escreve o atributo *name* do nó <*aspectj-file*>, e, por último, a instrução <*xsl:text*>, que acrescenta uma linha no arquivo revertido. Posteriormente, a instrução <*xsl:apply-templates*> é executada, selecionando o nó atual e aplicando o respectivo *template*. Finalmente, quando todos os nós tiverem sido executados, verificação implementada com a instrução <*xsl:if*>, um comentário final é inserido. A Figura 22 ilustra a código de tratamento do nó *aspectj-source-program*.

```
01 <xsl:template match="aspectj-source-program">
02   <xsl:for-each select="*">
03     <xsl:text>// FILE: </xsl:text>
04     <xsl:value-of select="@name"/>
05     <xsl:text>&#xA;</xsl:text>
06     <xsl:apply-templates select="."/>
07     <xsl:if test="not(position()=last())">
08       <xsl:text>//----------------&#xA;</xsl:text>
09     </xsl:if>
10   </xsl:for-each>
11 </xsl:template>
```

Figura 22 - *Template* de transformação do nó raiz (*aspectj-source-program*)

A instrução <*xsl:apply-templates* select="."/> (linha 06 na Figura 22) aplica o *template* que faz ligação com o nó <*aspectj-file*>. Este, por sua vez, transforma seus filhos na ordem em que se encontram no documento AspectJML. Nesse caso, o próximo *template* a ser executado é o do elemento <*import*>. Após o término da transformação de todos os nós <*import*>, o processador passará para o elemento <*aspect*>. A Figura 23 mostra o *template* para transformação deste nó.

```
01 <xsl:template match="aspect">
02   <saxon:assign name="cchIndent" select="$cchIndent + 4"/>
03   <xsl:call-template name="modificadores"/>
04   <xsl:text>aspect </xsl:text>
05   <xsl:value-of select="@name"/>
06   <xsl:if test="not(superclass)">
07              <xsl:call-template name="open-brace"/>
08   </xsl:if>
09   <xsl:apply-templates select="superclass"/>
10   <xsl:call-template name="newline"/>
11   <xsl:for-each select="aspect">
12     <xsl:call-template name="newline"/>
13     <xsl:call-template name="modificadores"/>
14     <xsl:text>aspect </xsl:text>
15       <xsl:value-of select="@name"/>
16       <xsl:apply-templates select="perclause"/>
17     <xsl:call-template name="open-brace-newline"/>
18     <xsl:if test="constructor/modifiers">
19              <xsl:apply-templates select="constructor"/>
20     </xsl:if>
21     <xsl:apply-templates select="advice"/>
22     <xsl:call-template name="close-brace-newline"/>
23   </xsl:for-each>
24   <xsl:call-template name="newline"/>
25   <xsl:if test="constructor/modifiers">
26              <xsl:apply-templates select="constructor"/>
27   </xsl:if>
28   <xsl:apply-templates select="declare|intertype-member-declaration|
29 pointcut|advice|field|method|class"/>
30   <xsl:call-template name="newline"/>
31   <saxon:assign name="cchIndent" select="$cchIndent - 4"/>
32   <xsl:call-template name="newline"/>
33   <xsl:call-template name="close-brace"/>
34 </xsl:template>
```

Figura 23 - *Template* para transformação dos aspectos

Inicialmente, o *template* acrescenta a identação utilizando a instrução <saxon:assign> (linha 02), específica do processador Saxon [66]. Em seguida, o aspecto é declarado com as seguintes instruções: o *template* "modificadores" (linha 03) acrescenta os modificadores; a *tag* <xsl:text> (linha 04) coloca na declaração a palavra reservada *aspect*; a *tag* <xsl:value-of> (linha 05) inclui o nome do aspecto através do atributo *name*; e, por último, a instrução <xsl:apply-templates> (linha 09) acrescenta o *extends* na declaração do aspecto. Após incluir uma linha com <xsl:call-template> chamando o *template* "*newline*" (linha 10), uma instrução <xsl:for-each> (linha 11) é executada. Se existir declaração de aspectos internos, esta instrução irá tratar. Posteriormente, são aplicados os *templates*: *declare*, *intertype-member-declaration*, *pointcut*, *advice*, *field*, *method* e *class* (linhas 28 e 29). Finalmente, é acrescentado um recuo na identação (linha 31) e a declaração do aspecto geral é fechada com uma chamada para o *template* "*close-brace*" (linha 33).

A seguir serão mostrados dois importantes *templates* ligados diretamente aos conceitos da POA com suas respectivas implementações. São eles: *pointcut* e *advice*.

O *template pointcut* é responsável pelas transformações referentes a conjuntos de junção. Inicialmente os modificadores são inseridos. Logo em seguida, são montados os argumentos do ponto de junção com o uso da instrução <xsl:apply-templates select="*formal-arguments*"/>. Finalmente, são montadas as expressões de conjuntos de junção através do <*xsl:apply-templates* select="*binary-pointcut-expr*"/>. A Figura 24 apresenta o código deste *template*.

```
01 <xsl:template match="pointcut">
02   <xsl:call-template name="newline"/>
03   <xsl:call-template name="modificadores"/>
04   <xsl:text>pointcut </xsl:text>
05   <xsl:value-of select="@name"/>
06   <xsl:apply-templates select="formal-arguments"/>
07   <xsl:if test="binary-pointcut-expr">
08      <xsl:text>: </xsl:text>
09      <xsl:call-template name="newline"/>
10      <xsl:apply-templates select="binary-pointcut-expr"/>
11   </xsl:if>
12   <xsl:call-template name="semicolon"/>
13   <xsl:call-template name="newline"/>
14 </xsl:template>
```

Figura 24 - *Template* para transformação de conjuntos de junção

O *template advice* é encarregado de realizar as transformações dos adendos. Assim como no *template* anterior, inicialmente os modificadores são inseridos. Logo em seguida, o tipo do adendo é acrescido através do atributo *kind* da instrução <xsl:value-of>. Então, são montados os argumentos do adendo com o uso da instrução <xsl:apply-templates select="formal-arguments"/>. Posteriormente, são montados o ponto de atuação e o bloco de execução do adendo através do <xsl:apply-templates select="binary-pointcut-exp|unary-pointcut-expr|block"/>. A Figura 25 apresenta o código deste *template*.

```
01 <xsl:template match="advice">
02     <xsl:call-template name="newline"/>
03     <xsl:call-template name="modificadores"/>
04     <xsl:apply-templates select="type"/>
05     <xsl:text> </xsl:text>
06     <xsl:value-of select="@kind"/>
07     <xsl:apply-templates select="formal-arguments"/>
08     <xsl:text>: </xsl:text>
09     <xsl:apply-templates select="binary-pointcut-expr|unary-pointcut-expr|block"/>
10     <xsl:call-template name="newline"/>
11 </xsl:template>
```

Figura 25 - *Template* para transformação de adendos

A Tabela 4 lista alguns *templates* utilizados como apoio durante as transformações e suas respectivas responsabilidades. Maiores detalhes sobre a implementação do reversor e seu desempenho podem ser obtidos em [57].

Nome	**Responsabilidade**
field	Transformação das variáveis de classe/aspecto.
method	Transformação da construção dos métodos da classe/aspecto.
formal-arguments	Transformação dos argumentos dos métodos, conjuntos de junção, adendos e construtores.
block	Transformação dos blocos de código dentro de métodos, adendos, condições e construtores.
if	Transformação da construção das condições.
Send	Transformação da chamada de um método.
class	Transformação da estrutura de classes.
constructor	Transformação de construtores, utilizado dentro de classes e aspectos.
declare	Transformação do elemento do tipo *declare*.
formal-arguments-pointcut-patt	Similar a *formal-arguments*. Aplica as transformações dos argumentos das expressões dos pontos de atuação.
method-patt	Transformação dos métodos para *declare*.
constructor-patt	Transformação dos construtores para *declare*.
modifiers-patt	Transformação dos modificadores para *method-patt* e *constructor-patt*.
formal-arguments-patt	Transformação dos argumentos para *method-patt* e *constructor-patt*.
binary-expr	Transformação das expressões de condições.
binary-type-patt-expr	Transformação das expressões utilizadas dentro de *method-patt*.
binary-pointcut-expr	Transformação das expressões utilizadas dentro de *pointcut* e *advice*.

Tabela 4 – *Templates* de apoio

3.6 Conclusão

Este capítulo apresentou AspectJML, uma linguagem de marcação baseada em XML para AspectJ. AspectJML tem como principal vantagem facilitar a identificação e manipulação de elementos estruturais de um programa AspectJ. Este benefício resulta da habilidade de AspectJML representar mais diretamente a estrutura sintática do programa, o que não ocorre na representação textual clássica.

Dentre as principais aplicações para AspectJML, destaca-se o suporte ao rápido desenvolvimento de ferramentas de análise e manipulação de código tirando proveito da grande variedade de tecnologias XML já disponíveis. Neste contexto, ferramentas para a

conversão entre AspectJ e AspectJML estão disponibilizadas ao público [9], permitindo que os benefícios oferecidos por essa forma de representação sejam estendidos a um maior número de pesquisadores e desenvolvedores de ferramentas de engenharia de software.

O próximo capítulo apresenta o processo de refatoração para AspectJ proposta nessa dissertação, o qual utiliza o AspectJML como base para a construção e customização de refatorações.

Capítulo 4

Um Processo de Refatoração para AspectJ utilizando Leis de Programação, AspectJML e XSLT

Este capítulo apresenta um processo de desenvolvimento de refatorações para AspectJ baseado no uso de Leis de Programação, AspectJML e XSLT.

4.1 Introdução

O uso de refatorações no desenvolvimento de software é cada vez mais comum. No entanto, mesmo as refatorações convencionais existentes nas atuais ferramentas de desenvolvimento mostram-se incorretas em algumas situações [19]. Isto porque muitas dessas refatorações exigem transformações não triviais na estrutura sintática do código, o que pode tornar a sua implementação uma tarefa complexa. A necessidade de automatizar operações de refatoração complexas requer a construção de soluções que dêem ao usuário o poder de implementá-las e customizá-las de acordo com as suas necessidades e preferências.

Uma alternativa para a construção de refatorações customizáveis é a utilização da linguagem XML como formato de representação da estrutura sintática do código [50]. Dessa forma, é possível construir refatorações a partir de transformações definidas sobre a representação do código em XML [56]. A principal vantagem dessa abordagem é a possibilidade de utilizar as diversas implementações dos padrões de consulta e atualização de XML. Outra vantagem é a possibilidade de tirar proveito das inúmeras linguagens declarativas existentes para manipulação de dados XML, como XSLT [73], para especificar e customizar refatorações em um nível de abstração muito maior do que seria possível utilizando linguagens de programação imperativas.

Este capítulo descreve uma estratégia de refatoração para AspectJ que permite construir novas refatorações utilizando somente linguagens declarativas. Ele utiliza tecnologias XML como mecanismo de representação e manipulação do código, a fim de

auxiliar a construção de ferramentas de refatoração mais facilmente reutilizáveis, customizáveis e extensíveis. Este processo de construção de refatorações utiliza AspectJML [53], uma representação XML para código fonte AspectJ, XSLT, uma linguagem de transformação para XML, e leis de programação para derivar refatorações para AspectJ [16].

A proposta apresentada neste trabalho se beneficia do XSLT pela sua facilidade de: (1) casamento de padrões; (2) especificação de *templates*; e (3) navegação entre nós XML. Como será mostrado mais adiante, estas funcionalidades são essenciais para construção de refatorações. A proposta foi validada com a construção de duas refatorações para AspectJ, *Extract Pointcut* [34] e *Extract Method Calls* [45].

Para implementar esta estratégia, realizaram-se basicamente duas atividades. A primeira foi a definição de um processo de construção de refatorações que permite a produção de novas refatorações somente com codificação em linguagens declarativas [55]. A segunda foi a adaptação de um arcabouço de refatoração ao processo de desenvolvimento proposto e a construção de uma ferramenta que instancia este arcabouço [54].

O processo de construção de refatorações segue o processo de refatoração de código baseado em XML definido por Mendonça *et al.* [56], e define diretrizes e guias para implementação de refatorações com a linguagem XSLT.

Em seguida, adaptou-se o arcabouço RefaX [49], que suporta o processo de refatoração de código baseado em XML descrito em [56], para suportar o processo de construção de refatorações com XSLT. A customização consistiu em reimplementar as abstrações originalmente implementadas em Java para a linguagem XSLT.

Concluída a definição de estratégia, ela foi validada com a implementação de uma instância desta nova versão do arcabouço RefaX. Esta instância, denominada RefaX-AOP [54], foi utilizada para implementar as duas refatorações para POA mencionadas anteriormente.

O restante deste capítulo está organizado da seguinte forma: a Seção 4.2 dá uma visão geral do processo de refatoração definido por Mendonça *et al.* [56], utilizado como base para construção do RefaX-AOP. A Seção 4.3 discute os requisitos estabelecidos para realização deste trabalho. A Seção 4.4 apresenta o processo de construção de refatorações

proposto por este trabalho. A Seção 4.5 descreve a implementação do processo. A Seção 4.6 mostra um exemplo de uso do processo com a implementação de duas refatorações. Por fim, a Seção 4.7 apresenta uma avaliação preliminar da ferramenta implementada e a Seção 4.84.7.2 sumariza as conclusões do capítulo.

4.2 Um Processo de Refatoração de Código Baseado em XML

Mens e Tourwé [58] definiram um processo composto de 5 passos para aplicar refatorações de código. São eles:

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

Baseado no processo enunciado acima, Mendonça *et al.* [56] definiram um processo de refatoração de código baseado em XML. Neste novo processo, os passos 3 a 5 foram redefinidos para se adaptar ao conjunto de modelos de dados e tecnologias de processamento XML utilizados. O processo resultante foi então reorganizado dentro dos seguintes novos passos:

1) Conversão do código fonte para XML;
2) Armazenamento dos dados XML gerados;
3) Aplicação de refatoração via manipulação de dados XML;
4) Conversão de XML para código fonte.

A Figura 26 ilustra os relacionamentos entre esses quatro passos. Cada passo será brevemente explanado a seguir.

4.2.1 Conversão de Código Fonte para XML

Este passo inicia quando o usuário seleciona os arquivos do código fonte que serão convertidos para a representação XML escolhida. O processo de conversão é geralmente realizado através de ferramentas automatizadas, como *parsers* de linguagem. A ferramenta responsável por este passo é chamada de *conversor XML*. Na Figura 26, o elemento retulado *Código para XML* representa o conversor utilizado no processo.

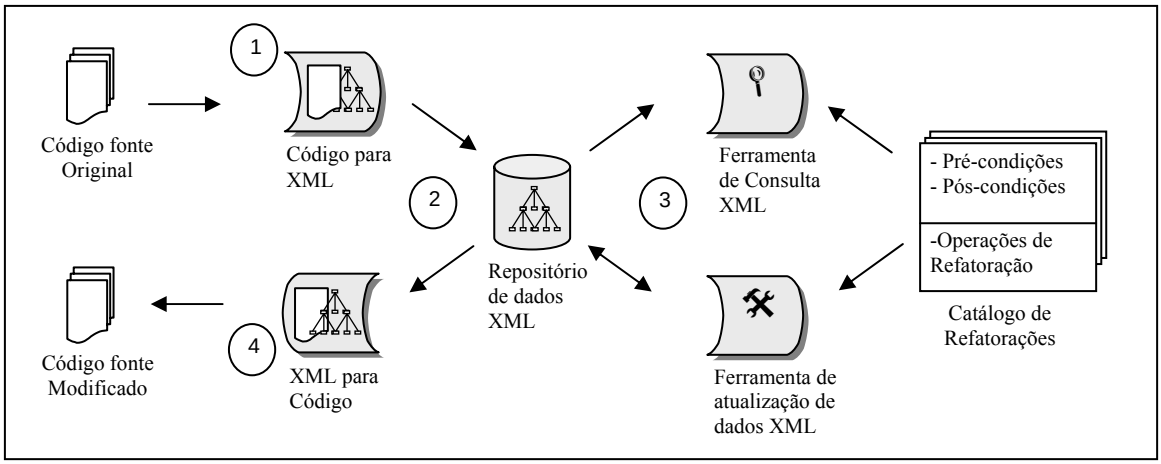

Figura 26 - Processo de refatoração de código baseado em XML (extraído de [56])

4.2.2 Armazenamento dos Dados XML Gerados

Uma vez que os arquivos do código fonte foram convertidos para a correspondente representação XML, esses dados devem ser armazenados em algum tipo de repositório. Ele está representado na Figura 26 pelo elemento *Repositório de Dados XML*, para que eles possam ser eficientemente acessados nos passos subseqüentes.

Esse repositório pode variar desde uma simples pasta em um sistema de arquivo até a um robusto banco de dados nativo XML. A escolha dependerá do tipo de tecnologia de processamento XML utilizada nos passos de refatoração.

4.2.3 Aplicação de Refatoração via Manipulação de Dados XML

Nesse passo, o usuário escolhe a refatoração que será aplicada ao código fonte em XML convertido no passo 1. Esse passo é subdividido em três etapas: teste de pré-condições, aplicação das operações de refatoração, e verificação da preservação do comportamento. Essas etapas são descritas a seguir.

4.2.3.1 Teste de pré-condições

As pré-condições são formadas por combinações de *Funções de Análise*, como proposto por Roberts [65]. *Funções de análise* são funções de avaliação de código e são classificadas em primitivas e derivadas. Um exemplo de função de análise primitiva é a *Superclass(className)*, que retorna a superclasse de *className*, caso *className* estenda outra classe ou Ø, caso contrário. As funções de análise derivadas são descritas como combinações de funções de análise primitivas. Por exemplo, a função *AllSuperclasses(className)*, que retorna o conjunto de todas as superclasses de *className,* pode ser descrita como:

- Ø, caso *Superclass(className)* = Ø; ou
- *Superclass(className)* U *AllSuperclasses(Superclass(className))*, caso contrário.

Na Figura 26, essa etapa é representada pelo fluxo indicado entre *Repositório de Dados XML*, *Ferramenta de Consulta XML* e *Pré-condições*.

4.2.3.2 Aplicação das operações de refatoração

Uma vez que todas as pré-condições foram validadas, a próxima atividade consiste em aplicar as operações de refatoração propriamente ditas. Uma única refatoração pode ser especificada em termos de múltiplas operações. Essas características não apenas incrementam a modularidade e o reuso das refatorações, mas também fornecem um poderoso mecanismo de abstração que permite descrever refatorações em um nível mais alto, de forma que sejam independentes de linguagem de programação e modelo de código fonte.

Como cada operação de refatoração altera o código fonte, neste processo de refatoração baseado em XML elas devem ser expressas usando alguma linguagem de transformação ou atualização de dados XML apropriada. Na Figura 26, essa etapa consiste no fluxo de informações entre os elementos *Repositório de Dados XML*, *Ferramenta de Atualização de Dados XML*, e *Operações de Refatoração*.

4.2.3.3 **Verificação da preservação do comportamento**

Após todas as operações de refatoração terem sido executadas, é necessário verificar se o código alterado ainda preserva seu comportamento externo original. Para isso,

este processo utiliza as pós-condições propostas por Roberts [65] e Tichelaar *et al*. [69]. Assim como as pré-condições, as pós-condições variam de acordo com o tipo e o propósito de cada refatoração. Como as pós-condições são limitadas a acessar apenas informações do código fonte, neste processo de refatoração baseado em XML elas também são expressas em termos de funções de análise, usando uma linguagem de consulta XML e uma ferramenta de consulta XML apropriada. Essa etapa é representada na Figura 26 pelo fluxo de informações entre os elementos *Repositório de Dados XML*, *Ferramenta de Consulta XML*, e *Pós-condições*.

4.2.4 Conversão de XML para Código Fonte

Uma vez que todas as transformações foram aplicadas aos devidos elementos do código fonte no repositório, seus resultados devem ser refletidos nos artefatos do código fonte original. Isso é feito geralmente utilizando uma ferramenta de transformação XML, que processa os elementos selecionados do código fonte de acordo com o esquema XML subjacente, gerando uma representação textual equivalente para esses elementos na linguagem de programação original. Esse segundo passo de conversão (de XML para o código fonte) é chamado de *reversão*, e a ferramenta de conversão utilizada é chamada *reversor XML*. Na Figura 26, o elemento *XML para código* representa o reversor utilizado no processo.

4.3 Requisitos de Construção e a Solução Adotada

A estratégia de refatoração de código AspectJ proposta neste trabalho teve como base três requisitos principais: (1) simplificar a personalização de novas refatorações; e (2) utilizar linguagens e padrões consolidados e amplamente utilizados. Estes dois requisitos são discutidos nas subseções a seguir.

4.3.1 Simplificar a customização de novas refatorações

Este requisito aborda a facilidade de construção de refatorações. Manipulação de árvores sintáticas e alto poder de análise de código são elementos fundamentais para implementação refatorações. A solução proposta deve fornecer uma abordagem simples de manipulação das estruturas sintáticas do código.

4.3.2 Utilizar linguagens e padrões consagrados

Este requisito pressupõe que o uso de linguagens e padrões estabelecidos e adotados por entidades confiáveis da comunidade acadêmica aumenta a chance da solução ser amplamente utilizada. Este requisito também previne que o desconhecimento da tecnologia utilizada prejudique a utilização da ferramenta.

Este requisito também favorece abordagens de refatoração de código que não obrigue o aprendizado de uma nova linguagem de programação ou de uma extensão de uma linguagem existente. Extensões ou novas linguagens aumentam a curva de aprendizado, reduzindo a produtividade na realização de refatorações.

4.3.3 Solução adotada

Para atender os dois requisitos discutidos acima, a solução adotada neste trabalho utiliza a linguagem de marcação AspectJML [53] como representação XML do código fonte AspectJ. Isto permite utilizar XSLT [73] para codificar as refatorações. Esta linguagem de transformação XML tem recursos bastante favoráveis para construção de refatorações, como casamento de padrões, especificação de *templates*, e facilidade de navegação entre nós (elementos da representação XML).

As características do XSLT apresentadas acima atendem ao requisito (1). Como o XSLT é um padrão estabelecido e não será preciso construir extensões para customizar refatorações, o requisito (2) também é atendido. Além de AspectJML e XSLT, a solução proposta utiliza ainda o conceito de leis de programação [31] para decompor as refatorações em transformações menores e, assim, facilitar a sua implementação [16]. A próxima seção apresenta o processo de construção de refatorações proposto.

4.4 Um Processo de Construção de Refatorações com Leis de Programação, AspectJML e XSLT

O Processo proposto é composto por quatro etapas: (i) identificar pré-condições, transformações e pós-condições; (ii) codificar pré e pós-condições, e as transformações em XSLT; (iii) ajustar os argumentos necessários pelo código XSLT; e (iv) incorporar o código XSLT gerado no arquivo de catálogo de refatorações da ferramenta de refatoração. Detalharemos a seguir cada etapa deste processo de desenvolvimento.

4.4.1 Identificar pré-condições, transformações e pós-condições

Uma refatoração é composta por pré-condições, transformações e pós-condições. Seguindo o processo de derivações de refatorações proposto por Cole e Borba [16], deve-se relacionar as pré-condições das transformações e as transformações de cada lei de programação que compõem a refatoração. Além disso, deve-se determinar, também, as pós-condições da refatoração.

As pré-condições de uma refatoração são derivadas das pré-condições das leis envolvidas. Diferentemente da maioria das abordagens de refatoração, nesta, nem todas as pré-condições podem ser verificadas no início. Como uma refatoração é o resultado da aplicação de uma seqüência de leis, a verificação das pré-condições de uma lei pode requisitar o resultado produzido por uma lei anterior.

As transformações são modificações definidas em cada lei de programação. A mudança definida por uma lei de programação é bem mais simples que a mudança de uma refatoração. Isso decompõe a construção de uma refatoração em blocos menores e mais fáceis de construir.

Pós-condições, conceito introduzido por Roberts [65], são verificações realizadas sobre o código refatorado e descrevem o que deve ou não haver depois da aplicação da refatoração. Elas são aplicadas após o término da última transformação da última lei de programação de uma refatoração.

4.4.2 Codificar pré e pós-condições, e as transformações em XSLT

O passo seguinte à identificação das pré e pós-condições, e transformações é a construção dos procedimentos XSLT. Para cada lei de programação, deve-se implementar as pré-condições e transformações necessárias.

Este procedimento é facilitado pela capacidade de tratamento de padrões do XSLT. Além disso, a quebra de uma refatoração em partes menores favorece o seu desenvolvimento nesta linguagem. Detalharemos a seguir guias de construção em XSLT de código de validação, pré e pós-condições, e de transformações.

As atividades básicas para construção dos procedimentos XSLT de verificação são as seguintes: (1) construir a expressão XPath [71] que determina a estrutura que será

verificada; e (2) extrair o resultado da verificação a partir das informações contidas nos nós. Por exemplo, caso se deseje verificar se um aspecto já tem um conjunto de junção (*pointcut*) com um determinado nome, deve-se selecionar o elemento *aspect* de AspectJML na expressão XPath. Com isso, basta percorrer as declarações de conjuntos de junção para obter o resultado da verificação.

Já as atividades para construção dos procedimentos XSLT de transformação são as seguintes: (1) construir a expressão XPath que determina a estrutura a ser modificada, e (2) construir o novo nó modificado. Quando uma transformação altera mais de um nó XML, recomenda-se a construção de uma transformação para cada nó XML. Assim, cada mudança fica fácil de ser controlada. Observe que, desta forma, quebramos ainda mais a granularidade das transformações de código, reduzindo a complexidade de sua construção.

4.4.3 Ajustar os argumentos requeridos pelo código XSLT

Após a conclusão da codificação em XSLT, deve-se mapear os argumentos necessários para funcionamento da refatoração. Estes argumentos são marcados com as *tags* ##RefaX_Param_NN##, onde NN é o seqüencial do parâmetro na refatoração. Estas *tags* são substituídas pela instância do RefaX em tempo de execução pelos parâmetros reais da refatoração. A Figura 27 ilustra o uso das *tags* com o código XSLT da pré-condição *Check pointcut name* da refatoração *Extract pointcut* [34].

```
01 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
02   <xsl:template match="/">
03     <xsl:variable name="result">
04       <xsl:for-each select="//pointcut">
05         <xsl:if test="@name='##RefaX_Param_01##'">
06           <xsl:value-of select="@name='##RefaX_Param_01##'"/>
07         </xsl:if>
08       </xsl:for-each>
09     </xsl:variable>
10     <xsl:value-of select="$result='true'"/>
11   </xsl:template>
12 </xsl:stylesheet>
```

Figura 27 – Uso de argumentos em procedimentos XSLT

4.4.4 Incorporar o código XSLT ao catálogo da ferramenta de refatoração

Após a substituição dos argumentos da refatoração, o código XSLT deve ser acoplado ao arquivo de catálogo de refatorações da ferramenta de refatoração. A Figura 28

ilustra a representação da refatoração *Extract Pointcut* e mostra a inserção da pré-condição *Check pointcut name*, apresentada na Figura 27, neste arquivo de catálogo.

4.5 Implementação do Processo

O RefaX [56] é um arcabouço de refatoração escrito em Java que implementa o processo de refatoração de código baseado em XML descrito na Seção 4.2. Este arcabouço foi projetado para atender cinco principais requisitos. São eles: (1) independência de linguagem; (2) independência de modelo de dados (ou esquema da representação XML); (3) independência de tecnologia de manipulação de dados XML; (4) confiabilidade; e (5) escalabilidade.

```
01 <rf:refactoring name="Extract_Pointcut">
02   <rf:law name="Extract_named_Pointcut">
03     <rf:pre-conditions>
04       <rf:pre-condition name="Check_pointcut_name">
05         <rf:stylesheet><![CDATA[
06           <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
07             <xsl:output method="text"/>
08             <xsl:template match="/">
09               <xsl:variable name="result">
10                 <xsl:for-each select="//pointcut">
11                   <xsl:if test="@name='##RefaX_Param_02##'">
12                     <xsl:value-of select="@name='##RefaX_Param_02##'"/>
13                   </xsl:if>
14                 </xsl:for-each>
15               </xsl:variable>
16               <xsl:value-of select="$result=''"/>
17             </xsl:template>
18           </xsl:stylesheet> ]]>
19         </rf:stylesheet>
20       </rf:pre-condition>
21     </rf:pre-conditions>
22     <rf:transformations>
23       <rf:transformation name="Create_pointcut"> ...
24       </rf:transformation>
25     </rf:transformations>
26     <rf:post-conditions/>
27   </rf:law>
28   <rf:law name="Use_named_Pointcut">
29     <rf:pre-conditions/>
30     <rf:transformations>
31       <rf:transformation name="Replace_pointcut_expr"> ...
32       </rf:transformation>
33     </rf:transformations>
34     <rf:post-conditions/>
35   </rf:law>
36 </rf:refactoring>
```

Figura 28 – Representação da refatoração *Extract Pointcut*

Entretanto, foi necessário realizar algumas modificações arquiteturais neste arcabouço para facilitar a construção de novas refatorações com XSLT. Para construir uma nova refatoração com o RefaX4Java [49] (uma implementação existente do arcabouço RefaX), por exemplo, é necessário codificar classes Java, procedimentos em alguma

linguagem de processamento de XML e a integração entre o XML e o Java. O RefaX-AOP substitui todo este esforço de desenvolvimento somente por codificação em XSLT.

A seguir apresentaremos a arquitetura do RefaX e a revisão realizada nessa arquitetura para instanciação do processo de desenvolvimento de refatorações descrito na Seção 4.4.

4.5.1 Arquitetura do RefaX

A arquitetura do RefaX está ilustrada na Figura 29 e seus elementos serão detalhados a seguir.

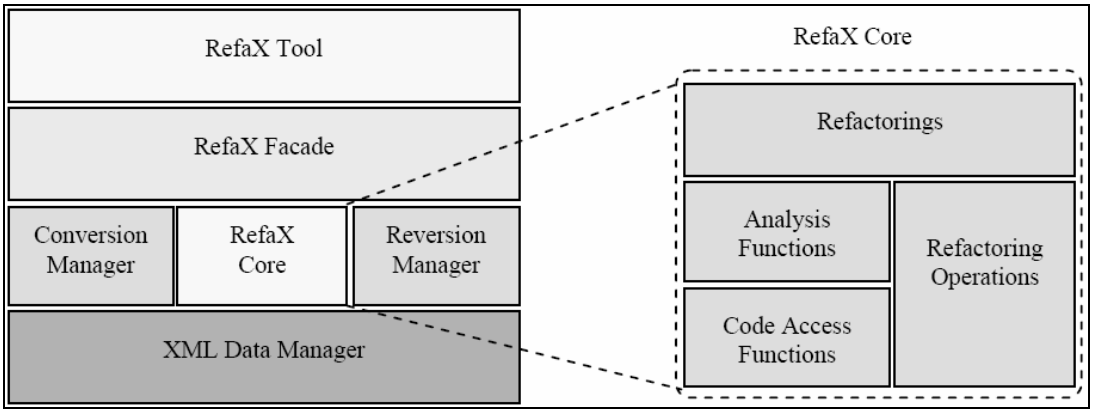

Figura 29 – Arquitetura original do arcabouço RefaX

4.5.1.1 Ferramenta RefaX

Esta camada representa qualquer ferramenta de refatoração desenvolvida através da extensão dos serviços oferecidos pelo arcabouço, e com a qual o usuário irá interagir para aplicar as refatorações disponíveis ao código fonte. Esta interação poderá acontecer através de uma interface própria, seja gráfica ou de programação, ou na forma de *plug-ins* para algum ambiente de desenvolvimento (IDE) existente.

4.5.1.2 RefaX Facade

Esta camada é responsável por fornecer uma interface unificada para o conjunto de serviços implementados nas camadas mais internas. Como o próprio nome indica, utilizamos o padrão de projeto *Facade* [23] para oferecer uma única forma de acesso às classes mais internas do arcabouço.

4.5.1.3 Gerenciador de Conversão

Esta camada é responsável por oferecer as operações necessárias para converter os arquivos originais do código fonte para a representação XML escolhida pelo desenvolvedor, através da utilização de um conversor.

4.5.1.4 Gerenciador de Reversão

Esta camada fornece as operações que o desenvolvedor irá utilizar para transformar o código em XML de volta para a sua representação original.

4.5.1.5 Gerenciador de Dados XML

Esta camada tem como função armazenar o código em XML no repositório de dados, bem como fornecer mecanismos para as outras camadas manipularem os dados armazenados.

4.5.1.6 Núcleo RefaX

A camada destacada na parte direita da Figura 29 é o Núcleo RefaX (*RefaX Core*). Esta camada é composta por quatro subcamadas: Refatorações (*Refactorings*), Funções de Análise (*Analysis Functions*), Funções de Acesso ao Código (*Code Acces Functions*) e Operações de Refatoração (*Refactoring Operations*). Elas são responsáveis pelas refatorações propriamente ditas (teste das pré-condições, aplicação das operações de refatoração e verificação das pós-condições). O lado direito da Figura 29 mostra como essas subcamadas estão estruturadas. Abaixo descrevemos o papel de cada uma na arquitetura.

- **Refatorações**: representa as refatorações disponíveis em RefaX.
- **Funções de análise**: representa as funções de análise (*Analysis Functions - AFs*) utilizadas para compor as pré e pós-condições das refatorações.
- **Funções de acesso ao código:** representa o conjunto de funções responsáveis por acessarem cada elemento do código em XML. É ela que implementa o requisito de independência de esquema XML, pois sempre que um desenvolvedor for instanciar RefaX para uma determinada representação XML, cada função de acesso ao código (*Code Access Functions - CAFs*) deve ser especificada para essa representação.

- **Operações de refatoração**: representa cada operação que altera o código XML a fim de aplicar a refatoração desejada. As operações de refatoração (*Refactoring Operations - ROs*) não utilizam as funções de acesso ao código. Cada uma utiliza uma tecnologia XML diferente: as funções de acesso são especificadas usando uma linguagem de consulta, pois seu objetivo é pesquisar elementos de código, enquanto que para especificar as operações de refatoração utiliza-se uma linguagem de atualização ou transformação.

4.5.2 Revisão da Arquitetura

Esta subseção apresenta a mudança realizada no Arcabouço RefaX e os impactos nas funcionalidades do arcabouço.

A alteração consistiu em modificar a camada Núcleo RefaX para suportar a estratégia de refatoração baseada em XSTL. Com a estratégia de transferir para o XSLT a responsabilidade de acessar e transformar o código fonte, a subcamada Funções de Acesso ao Código, responsável pela abstração de acesso ao código, tornou-se desnecessária. No seu lugar, foi construída a subcamada Definições de Refatorações (RefactoringDefinition), que fornece para ao arcabouço as implementações em XSLT. A Figura 30 ilustra a arquitetura revisada do Arcabouço RefaX.

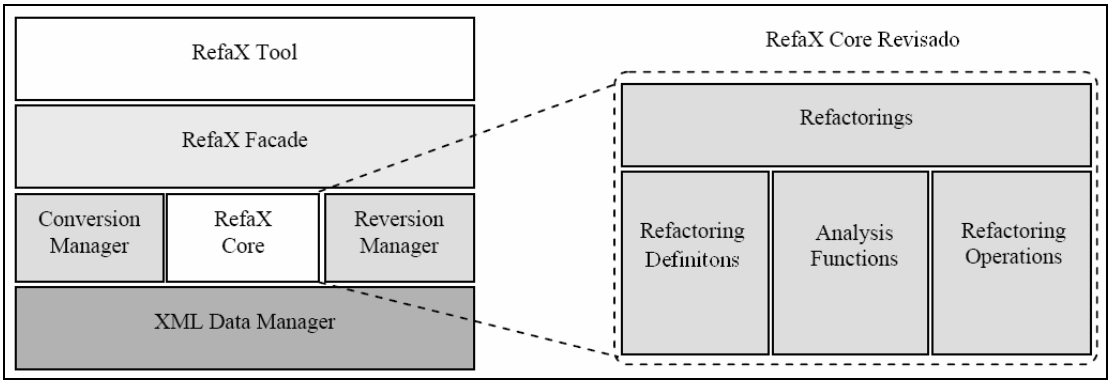

Figura 30 – Arquitetura revisada do arcabouço RefaX

Com esta mudança, os papéis das subcamadas do Núcleo RefaX foram revisados. Abaixo descrevemos o papel de cada uma na arquitetura.

- **Definições de Refatorações**: contém um catálogo de refatorações disponíveis em RefaX. Fornece as implementações (pré-condições, transformações e pós condições) das refatorações disponíveis.

- **Refatorações**: é responsável por coordenar a execução da refatoração através dos seguintes passos: (1) obter da subcamada Definição de Refatorações as implementações de cada refatoração; (2) verificar, com o auxílio da subcamada Funções de Análise, se as pré-condições são atendidas; (3) efetuar, com o auxílio da subcamada Operações de Refatoração, as transformações de código necessárias; e (4) verificar, com o auxílio da subcamada Funções de Análise, se as pós-condições são atendidas.

- **Funções de análise**: é responsável por verificar as pré e pós-condições das refatorações fornecidas pela subcamada Definições de Refatorações.

- **Operações de refatoração**: é responsável por aplicar as transformações das refatorações fornecidas pela subcamada Definições de Refatorações.

Esta mudança arquitetural afetou os requisitos independência de linguagem, independência de esquema XML e independência de tecnologia. Comentaremos a seguir as vantagens e desvantagens desta mudança.

A independência de linguagem e a independência de esquema XML eram garantidas pela subcamada Funções de Acesso ao Código. Esta subcamada abstrai do resto do arcabouço os detalhes da linguagem de programação alvo da refatoração e a forma de armazenamento da sua representação em XML. Com a eliminação desta subcamada e a inclusão da subcamada Definições de Refatoração, o código XSLT produzido para implementação das refatorações será dependente da linguagem de programação e de sua representação em XML. Esta dependência constitui uma fraqueza desta nova versão da arquitetura. No entanto, se considerarmos o cenário de funcionamento apenas para AspectJ, única linguagem de programação OA utilizada em larga escala, este prejuízo será minimizado.

Com a adoção da linguagem XSLT, o requisito de independência de tecnologia também é violado. No entanto, se considerarmos o ganho de produtividade atingido com o uso de apenas uma linguagem de programação para implementar pré e pós-condições e transformações da refatoração, o prejuízo da ausência deste requisito também é minimizado.

4.5.3 RefaX-AOP: Uma Ferramenta de Customização de Refatorações para AspectJ

Para demonstrar a viabilidade do processo de refatoração proposto neste trabalho, e da revisão de arquitetura realizada no arcabouço RefaX, foi implementada uma ferramenta de refatoração de código AspectJ a partir de uma instanciação desta versão do arcabouço, denominada RefaX-AOP [54]. Esta ferramenta constitui o núcleo de um ambiente de refatoração, a partir do qual é possível construir diferentes interfaces para a interação do usuário com a ferramenta.

No entanto, para que a ferramenta possa ser usada, é necessário que ela esteja integrada a um ambiente de desenvolvimento, ou então ela própria seja uma IDE. Desta forma, construiu-se o RefaX-AOP na forma de um plug-in para o ambiente de desenvolvimento Eclipse [18]. A seguir, mostramos como RefaX-AOP foi implementado instanciando a versão modificada do RefaX e como a ferramenta interage com o plug-in, criando um ambiente de refatoração para código AspectJ.

4.5.3.1 Escolha dos Pontos de Extensão do RefaX

Para instanciar o RefaX, é necessário escolher um conjunto de tecnologias XML para preencher seus pontos de extensão. As escolhas feitas para RefaX-AOP foram:

- Representação de código Java em XML: AspectJML [53][9], descrito no capítulo anterior;
- Conversor: foi utilizado o conversor desenvolvido para o AspectJML [9];
- Repositório de dados XML: eXist [20], um banco de dados nativo XML gratuito;
- Ferramenta de atualização XML: motor embutido em eXist [20].
- Reversor: foi utilizado o reversor escrito em XSLT também fornecido juntamente com o AspectJML [9].

É importante salientar que, diferentemente da versão original do RefaX, esta versão do arcabouço não exige a realização da escolha da ferramenta de consulta XML e da linguagem de atualização XML, uma vez que ele utiliza por padrão o XSLT.

4.5.3.2 Detalhes de Implementação

Como toda instância do RefaX, o RefaX-AOP foi construído em Java. Esta instância construiu as classes *RDXSLTRefactoring*, *GenericAnalysisFunction*, *GenericRefactoringOperation* e *GenericRefactoring*. Utilizamos o prefixo *Generic* para indicar que as classes não estão ligadas diretamente a uma refatoração específica. Esta abordagem foi adotada originalmente pelos criadores do RefaX. Detalharemos a seguir o papel de cada classe construída.

A classe *RDXSLTRefactoring* é responsável por fornecer as implementações das refatorações. O construtor desta classe recebe o nome de uma refatoração como parâmetro e recupera de um catálogo de refatorações, a refatoração correspondente. Esta refatoração é representada por um conjunto de classes *RefactoringLaw* e um procedimento de verificação de pós-condições.

A classe *GenericRefactoring* é responsável por obter as refatorações e aplicar as pré-condições, transformações e pós-condições. Esta classe coordena a realização da refatoração. Ela obtém e aplica as transformações especificadas nas leis de programação na ordem definida. Para cada lei, as pré-condições são verificadas e, caso sejam atendidas, as transformações são aplicadas. Após a aplicação com sucesso de todas as leis que compõem uma refatoração, as pós-condições da refatoração são verificadas, concluindo o processo de refatoração.

As classes *GenericAnalysisFunction* e *GenericRefactoringOperation* são responsáveis por, respectivamente, verificar pré e pós-condições, e aplicar as transformações no código. Elas recebem um código XSLT e aplicam em uma unidade de compilação. A primeira retorna um valor binário indicando se a pré-condição foi ou não atendida. A segunda classe recebe uma unidade de compilação e devolve uma unidade de compilação com uma transformação de uma lei aplicada. A Figura 31 ilustra o diagrama das classes construídas e sua distribuição nas subcamadas do RefaX Core do RefaX-AOP.

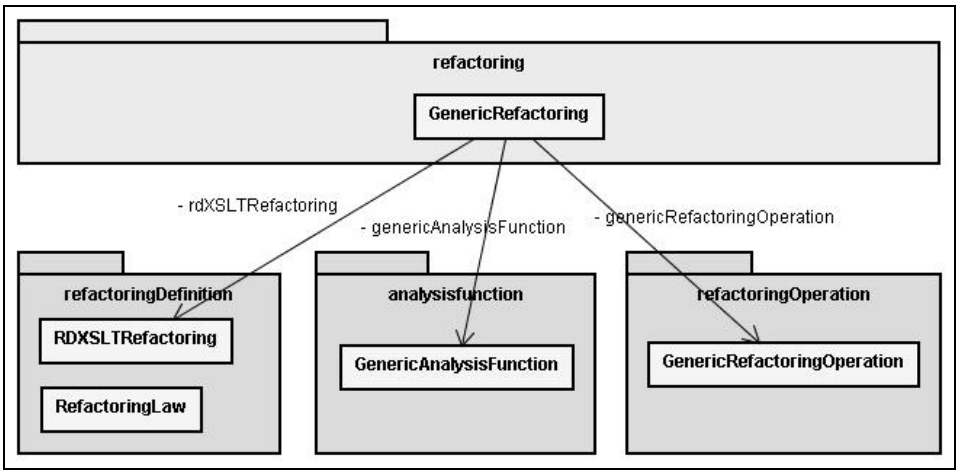

Figura 31 – Diagrama de Classes do pacote RefaX Core do RefaX-AOP

4.5.3.3 RefaX4AJ: Um Plug-in para o Eclipse

Foi construído um *plug-in* para o Eclipse para facilitar a aplicação das refatorações implementadas pelo RefaX-AOP. A interface gráfica do *plug-in* é ilustrada na Figura 32. O uso do *plug-in* é bastante simples e consiste em: (i) selecionar uma refatoração dentre as existentes no arquivo de configuração da ferramenta; (ii) informar a lista de parâmetros requisitados; e (iii) confirmar a operação.

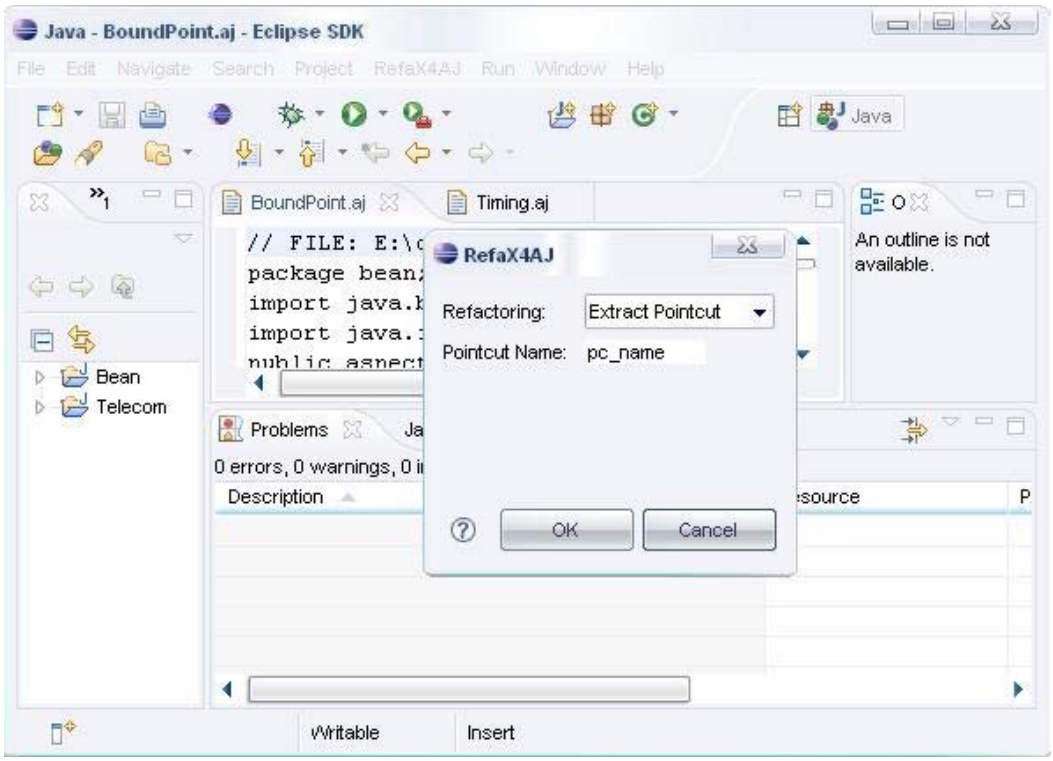

Figura 32 – Interface Gráfica do RefaX-AOP

4.6 Exemplo de Uso

O RefaX-AOP foi validado preliminarmente com a implementação de duas refatorações derivadas em [16] como combinação de leis de programação: *Extract pointcut* [34] e *Extract Method Calls* [45]. A seguir, descreve-se como estas duas refatorações foram construídas utilizando o processo de construção de refatorações proposto.

4.6.1 Construção da Refatoração *Extract Pointcut*

A refatoração *Extract pointcut* [34] descreve uma refatoração de AspectJ para AspectJ que melhora a qualidade do código. Ela cria um conjunto de junção nomeado a partir de expressões de conjuntos de junção existentes em adendos (*advices*), promovendo reuso de código. Esta refatoração foi derivada por Cole e Borba a partir da aplicação da seguinte seqüência de leis de programação [16]: *Extract Named Pointcut* e *Use Named Pointcut*.

4.6.1.1 Identificar pré-condições, transformações e pós-condições

Conforme o descrito em 4.2.2, o primeiro passo do processo consiste em identificar as pré-condições e transformações de cada lei de programação e as pós-condições da refatoração. A Tabela 5 relaciona as pré-condições e transformações das leis de programação desta refatoração.

Leis	**Pré-condições**	**Transformações**
Extract Named Pointcut	[C1] Não existir pointcut de nome ‘p’	[T1] Criar um conjunto de junção nomeado após a lista de conjuntos de junção do aspecto contendo a expressão de conjunto de junção fornecida na refatoração.
Use Named Pointcut	Nenhuma	[T2] Substituir a expressão de conjunto de junção por uma referência ao conjunto de junção criado com a aplicação da lei *Extract Named Pointcut*.

Tabela 5 – Leis e pré-condições da refatoração *Extract Pointcut*

4.6.1.2 Codificar pré e pós-condições, e as transformações em XSLT

O próximo passo do processo é codificar as verificações e transformações identificadas no passo anterior em XSLT. Na refatoração *Extract pointcut*, esta atividade consiste em codificar a pré-condição [C1] e as transformações [T1] e [T2] indicadas na Tabela 5. O Apêndice A apresenta o código XSLT dessa refatoração.

A construção do procedimento XSLT da pré-condição [C1] consiste em buscar no conteúdo do nó AspectJML que representa o aspecto (*<xsl:template match= "//aspect[@id='{identificador do aspecto}']">*) um nó *pontcut* que tenha o atributo *name* igual a 'p' (*<xsl:if test="@name='{nome do pointcut}'">*). Caso a busca encontre uma ocorrência, o procedimento XSLT retorna falso, caso contrário, ele retorna verdadeiro.

A construção da transformação [T1] consiste em criar um novo conjunto de junção com a expressão de conjunto de junção fornecida após o último conjunto de junção existente no aspecto refatorado (*<xsl:template match= "//aspect[@id='{identificador do aspecto}'] //pointcut[last()]">*). A Figura 33 ilustra o código em XSLT desta refatoração.

```
<xsl:template match="//aspect[@id='##RefaX_Param_01##']//pointcut[last()]">
  <xsl:copy-of select="../pointcut[last()]"/>
  <pointcut id="##RefaX_Param_01####RefaX_Param_02##" idkind="pointcut" name="##RefaX_Param_02##">
    <xsl:call-template name="argumentos"></xsl:call-template>
    <xsl:copy-of select="$RefaX_Param_03"/>
  </pointcut>
</xsl:template>

<xsl:template match="//pointcut[@id='##RefaX_Param_01####RefaX_Param_02##']/formal-arguments" name="argumentos">
  <xsl:call-template name="add-param">
    <xsl:with-param name="comp1" select="1"/>
  </xsl:call-template>
</xsl:template>
<xsl:template name="add-param">
  <xsl:param name="comp1"/>
  <xsl:if test="$comp1 = '1'">
    <xsl:for-each select="//advice/*[@idkind='pointcut-expr']">
      <xsl:variable name="data" select="../*[@idkind='pointcut-expr']"/>
      <xsl:variable name="identical">
        <xsl:call-template name="check_identical">
          <xsl:with-param name="comp1" select="$data"/>
          <xsl:with-param name="comp2" select="$RefaX_Param_03"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:choose>
        <xsl:when test="$identical='true'">
          <xsl:copy-of select="../formal-arguments"/>
        </xsl:when>
      </xsl:choose>
    </xsl:for-each>
  </xsl:if>
 <xsl:if test="$comp1 != '1'">
   <xsl:copy-of select="."/>
  </xsl:if>
</xsl:template>
```

Figura 33 – Código XSLT da transformação T1 – Criar conjunto de junção

A construção da transformação [T2] consiste em localizar e substituir as ocorrências da expressão de conjunto de junção (*<xsl:template match= "//aspect[@id='{identificador do aspecto}']//*[@idkind='pointcut-expr']">* e *<xsl:when test="../*[@idkind='pointcut-expr'] = {expressão de conjunto de junção}">*) por uma referência ao conjunto de junção criado na transformação [T1].

4.6.1.3 Ajustar os argumentos requeridos pelo código XSLT

Concluída a codificação descrita no passo anterior, o passo seguinte consiste em ajustar os parâmetros da refatoração. Na refatoração *Extract pointcut*, são requeridos três argumentos: (1) o identificador do aspecto; (2) o nome do conjunto de junção; e (3) a expressão de conjunto de junção. Cada parâmetro da refatoração será substituído pelo símbolo ##RefaX_Param_NN##, onde NN é o número do parâmetro. A ferramenta de refatoração substitui estes marcadores pelos parâmetros reais da refatoração em tempo de execução.

4.6.1.4 Incorporar o código XSLT ao arquivo de catálogo da ferramenta de refatoração

O último passo consiste em incorporar o código construído ao arquivo de catálogo do RefaX-AOP. Para isso, deve-se criar um nó *<rf:refactoring name="{Extract Pointcut}">* para representar a refatoração. Dentro deste nó, deve-se criar um nó *<rf:law name="{nome da lei de programação}">* para cada lei de programação. Em cada nó de lei, deve-se criar um nó *<rf:pre-conditions>* e um nó *<rf:transformations>*. No nó de pré-condições, deve-se criar um nó *<rf:pre-condition name="{nome da pré-condição}">* para cada pré-condição. No nó de transformações, deve-se criar um nó *<rf:transformation name="{nome da transformação}">* para cada transformação. Os nós *rf:pre-condition* e *rf:transformation* devem conter os procedimentos XSLT construídos nos passos anteriores.

4.6.2 Construção da Refatoração *Extract Method Calls*

A refatoração *Extract Method Calls* [45] descreve uma refatoração de AspectJ para Java que aumenta a modularidade de chamadas de métodos que aparecem em vários outros métodos. Ela cria um aspecto que define um adendo que realiza uma chamada ao método no momento apropriado. Esta refatoração foi derivada por Cole e Borba a partir da aplicação da seguinte seqüência de leis de programação [16]: *Add Before-Execution*, *Merge Before*, *Remove this Parameter*, *Remove Argument Parameter* e a aplicação da refatoração *Extract Pointcut*, detalhada no item anterior. Conforme se percebe pela escolha da lei *Add Before-Execution*, esta implementação da refatoração *Extract Method Calls* se aplica apenas para chamadas de métodos que estão posicionadas no início de um método. No entanto, Cole e Borba também definem leis para aplicação desta refatoração quando a chamada ocorre no final do corpo de um método, que tem implementação similar.

4.6.2.1 Identificar pré-condições, transformações e pós-condições

Conforme o processo proposto, deve-se identificar as pré-condições e transformações de cada lei de programação e as pós-condições da refatoração. A Tabela 6 relaciona as pré-condições e transformações das leis de programação desta refatoração.

4.6.2.2 Codificar pré e pós-condições, e as transformações em XSLT

O passo seguinte é codificar as verificações e transformações identificadas no passo anterior em XSLT. Na refatoração *Extract Method Calls*, esta atividade consiste em codificar as pré-condições [C1], [C2], [C3], [C4], [C5], [C6], [C7] e [C8]; e as transformações [T1], [T2], [T3] e [T4] indicadas na Tabela 6. O Apêndice A apresenta o código XSLT dessa refatoração.

A pré-condição [C1] não precisa ser implementada, uma vez que o trecho de código envolvido na movimentação para o adendo, uma chamada de um método, não declara variáveis.

Leis	**Pré-condições**	**Transformações**
Add Before-Execution	[C1] O trecho de código não declara variável nem chama o método super	[T1] Criar um adendo do tipo *before* para cada chamada do método especificado na entrada após o último adendo do tipo *before* do aspecto definido.
	[C2] O trecho de código não chama *return*	
Merge Before	[C3] Os pontos de junção (*joinpoints*) capturados pelas expressões de adendos criados em [T1] são disjuntos	[T2] Criar um adendo do tipo *before* que substitua todos os adendos adjacentes que: (i) contenha os mesmos parâmetros, (ii) tenha o corpo do adendo idêntico.
	[C4] Os adendos precisam ter a mesma lista de parâmetros	
	[C5] Não há nenhum adendo entre os adendos que serão mesclados	
	[C6] O corpo do adendo deve ser idêntico	
Remove this Parameter	[C7] Um parâmetro t do tipo T não ser referenciado pelo corpo do adendo	[T3] Excluir o parâmetro t do tipo T e substituir o *this(t)* por *this(T)*
Remove Argument Parameter	[C8] Um parâmetro t do tipo T não ser referenciado pelo corpo do adendo	[T4] Excluir o parâmetro t do tipo T da lista de parâmetros do adendo

Tabela 6 – Leis e pré-condições da refatoração *Extract Method Calls*

A pré-condição [C2] não deve ser implementada explicitamente e será implementada de maneira indireta na transformação [T1]. Caso implementada explicitamente, a pré-condição [C2] pode inviabilizar a refatoração por causa de uma ocorrência de chamada de método em uma expressão de *return*. Desta forma, [T1] irá selecionar apenas as ocorrências de chamadas de método que sejam a primeira sentença (*statement*) do corpo do método. Caso haja chamadas de método dentro de expressões de *return*, elas não serão selecionadas.

A construção da transformação [T1] consiste em criar um adendo *before* para cada chamada do método refatorado (*<xsl:for-each select="//send[@idref='{identificador do método}']">*). Este adendo será criado após o último adendo do tipo *before* do aspecto (*<xsl:template match="//aspect[@id='{identificador do aspecto}']//advice[@kind='before'][last()]">*). No entanto, conforme especificado na pré-condição [C2], isso só deve acontecer quando a chamada do método for a primeira sentença do corpo do método (*<xsl:when test="parent::node()/*[position()=1]=self::node()">*). Atendida as condições e identificados os locais para criação dos adendos, basta construir o código AspectJML que representa o novo adendo.

A pré-condição [C3] também não precisa ser verificada, pois cada expressão de conjunto de junção existente nos adendos criados em [T1] refere-se a uma chamada de método distinta. Desta forma, a intersecção entre as expressões de conjuntos de junção criadas em [T1] será sempre vazia.

De maneira análoga à pré-condição [C2], a pré-condição [C4] será verificada somente na execução da transformação [T2]. Desta forma, [T2] selecionará apenas adendos com a lista de parâmetros especificados na refatoração.

A construção da pré-condição [C5] consiste em verificar se não há nenhum adendo entre os adendos que serão mesclados. Esta pré-condição também deve ser implementada explicitamente porque será verificada na transformação [T2].

A pré-condição [C6] não precisa ser implementada, uma vez que o corpo dos adendos mesclados será a chamada do método alvo da refatoração. Isso também será verificado na implementação da transformação [T2].

Finalmente, a construção da transformação [T2] consiste em localizar e substituir os adendos criados em [T1] (*<xsl:template match="//aspect[@id='{identificador do aspecto}']//advice [@kind='before']">*) por um só adendo que realize o trabalho de todos os anteriores. Para realizar esta transformação, precisamos comparar a lista de parâmetros dos adendos, pré-condição [C4], e os corpos dos adendos, pré-condição [C6].

Estas comparações de estruturas se resumem em comparações de nós XML, efetuadas facilmente em XSLT. No entanto, o AspectJML tem uma peculiaridade que exigiu um tratamento adicional no procedimento de comparação. Esta linguagem utiliza a seguinte estrutura para os identificadores de argumentos: *L<pacote>/<pacote>/<classe>; arg<número>,* onde *<número>* é um identificador único do argumento. Com isso, caso os nós XML fossem comparados indistintamente, o resultado da comparação sempre seria falso, por conta da divergência dos números dos identificadores dos argumentos. Para contornar este problema, alteramos o procedimento de comparação de nós XML para ignorar o numeral existente depois de um ";arg" em um atributo.

Utilizando o procedimento de comparação discutido no parágrafo anterior, realizamos a comparação entre os argumentos do adendo (*<xsl:with-param name="comp1" select=".//formal-arguments"/>*) e os argumentos do adendo adjacente (*<xsl:with-param name="comp1" select="following-sibling::advice[@kind='before']//formal-arguments"/>*). Da mesma forma, realizamos a comparação entre o corpo do adendo (*<xsl:with-param name="comp1" select=".//send"/>*) e o corpo do adendo adjacente (*<xsl:with-param name="comp1" select="following-sibling::advice[@kind='before']//send"/>*). Sobre o corpo do adendo, precisamos verificar ainda se ele é igual à chamada do método alvo da refatoração informada no início da refatoração (*<xsl:with-param name="comp2" select= "$RefaX_Param_03"/>*).

A transformação propriamente dita consiste em alterar a expressão de conjunto de junção de um dos adendos pela união das expressões dos dois adendos. Para completar a transformação, basta suprimir o outro adendo. A Figura 34 ilustra a montagem da expressão de conjunto de junção do adendo resultante.

A pré-condição [C7] consiste em verificar no corpo do adendo (*<xsl:template match="//aspect[@id='{identificador do aspecto}']//advice[@kind='before']//send">*) se o argumento "t" do tipo "T" criado na transformação [T1] que referencia a classe T é utilizado.

Caso não seja utilizado, a transformação [T3] elimina o argumento do adendo (*<xsl:template match="//aspect[@id='{identificador do aspecto}']//advice[@kind='before']//formal-argument[1]">*) e substitui o parâmetro "t" da função reservada *this* pelo tipo "T" (*<xsl:template match="//aspect[@id='{identificador do aspecto}']//advice[@kind='before']//formal-argument-pointcut-expr[@type='this']">*). A implementação da pré-condição [C8] e a transformação [T4] são similares à pré-condição [C7] e da transformação [C3].

```
01  <binary-pointcut-expr idkind="pointcut-expr" op="||">
02     <xsl:copy-of select="//*[@idkind='pointcut-expr']"/>
03     <xsl:copy-of select="following-sibling::advice[@kind='before']//send/../../*[@idkind='pointcut-expr']"/>
04  </binary-pointcut-expr>
```

Figura 34 – Montagem da expressão de conjunto de junção contendo as expressões dos dois adendos da expressão de conjunto de junção contendo as expressões dos dois adendos

4.6.2.3 Ajustar os argumentos requeridos pelo código XSLT

Concluída a codificação descrita no passo anterior, o próximo passo consiste em ajustar os parâmetros da refatoração. Na refatoração *Extract Method Calls*, são requeridos 3 argumentos. São eles: (1) o identificador do aspecto onde o adendo e o conjunto de junção serão criados; (2) o nome do conjunto de junção; e (3) o trecho de código que representa chamada do método que será transportado para o adendo. Cada parâmetro da refatoração será substituído pelo símbolo ##RefaX_Param_NN##, onde NN é o seqüencial do argumento, no código XSLT e o RefaX-AOP irá substituí-lo pelo parâmetro da refatoração no momento da aplicação da refatoração.

4.6.2.4 Incorporar o código XSLT ao arquivo de catálogo da ferramenta de refatoração

O último passo, item 4.4.4, consiste em incorporar o código construído ao arquivo de catálogo do RefaX-AOP. Para isso, deve-se criar um nó *<rf:refactoring name="{Extract Method Calls}">* para representar a refatoração. Com este nó, deve-se seguir o mesmo procedimento descrito na refatoração anterior.

4.7 Avaliação Preliminar

Para validar o uso do RefaX-AOP, implementou-se e aplicou-se a refatoração *Extract Pointcut* em três pequenas aplicações implementadas em AspectJ, *bean*, *spacewar* e *telecom*.

A aplicação *bean* consiste basicamente de uma classe *Point* e um aspecto *BoundPoint* que adiciona suporte a mudança de propriedades e serialização à classe *Point*. Esta aplicação tem três unidades de compilação, sendo duas classes e um aspecto, e seu código fonte tem 239 linhas.

A aplicação *spacewar* é a implementação de um jogo. Esta aplicação tem 19 unidades de compilação, sendo 15 classes e quatro aspectos, e seu código fonte tem 2277 linhas.

A aplicação *telecom* é um simulador de sistema de telefonia. Esta aplicação tem 13 unidades de compilação, sendo 10 classes e três aspectos, e seu código tem 749 linhas.

Todas são aplicações exemplo distribuídas juntamente com o compilador AspectJ [7]. A Figura 35 exibe o código antes, na parte de cima, e depois, na parte de baixo, da refatoração *Extract Pointcut* aplicada na aplicação *bean*. Com esta refatoração, o conjunto de junção *pc* criado pode ser reusado em outros adendos, aumentando o reuso e a legibilidade do código.

As avaliações realizadas e os resultados encontrados são descritos nos parágrafos a seguir. A análise foi feita sob duas vertentes. A primeira analisou o desempenho da ferramenta construída. A segunda analisou a facilidade de implementação de novas refatorações utilizando o processo proposto e a ferramenta desenvolvida.

```
01  void around(Point p): execution(void Point.setX(int)) && target(p) {
02      int oldValue = p.getX();
03      proceed(p);
04      firePropertyChange(p, "x", oldValue, p.getX());
05  }
```

```
06  pointcut pc (Point p):
07      execution(void Point.setX(int) && target(p);
08  void around(Point p): pc(p) {
09      int oldValue = p.getX();
10      proceed(p);
11      firePropertyChange(p, "x", oldValue, p.getX());
12  }
```

Figura 35 – Exemplo da refatoração *Extract Pointcut* no aspecto BoundPoint.

4.7.1 Análise de Desempenho

A avaliação de desempenho consistiu em: (i) selecionar aspectos das aplicações; (ii) aplicar a refatoração; e (iii) medir o tempo de realização do procedimento. Todos os testes neste experimento foram realizados com o mesmo hardware, um laptop Pentium 4, com processador de 1.4 GHz e 512 MB de Memória DRAM.

A Tabela 7 apresenta o tempo de processamento das três fases da refatoração supracitada (*Converter para AspectJML*, *Refatorar*, e *Reverter para AspectJ*). A primeira fase, *Converter para AspectJML*, consiste em acionar o compilador *ajc* modificado para gerar a representação de código em AspectJML. A segunda fase, *Refatorar*, consiste em aplicar, usando o RefaX-AOP, os *templates* XSLT para modificar o código AspectJ representado em AspectJML. Por fim, a terceira fase, *Reverter para AspectJ*, consiste em aplicar os *templates* XSLT que transformam código representado em AspectJML em código AspectJ. Os tempos de execução de cada fase da refatoração foram medidos para as aplicações relacionadas e somados na coluna *Total* da Tabela 7.

Nos experimentos realizados nas aplicações descritas acima, obtivemos tempos totais de refatoração variando entre 4,6 e 9,8 segundos (coluna *Total* da Tabela 7). A parcela mais custosa das três fases é a aplicação da refatoração, que corresponde, em média, a 88,86% do tempo de processamento das refatorações (coluna *Refatorar* da Tabela 7). A etapa de conversão para AspectJML corresponde, em média, a 6,73% do tempo de processamento (coluna *Converter para AspectJML* da Tabela 7). Por fim, a etapa de reversão para AspectJ corresponde, em média, a apenas 4,41% do tempo de processamento (coluna *Reverter para AspectJ* da Tabela 7).

Embora não se tenha medido o desempenho de outras ferramentas [27][25][28][5], a experiência mostra que o desempenho medido pelas ferramentas implementadas está aquém do desempenho das ferramentas de refatoração existentes. Salienta-se ainda que as ferramentas implementadas podem ser otimizadas. Apesar disso, acredita-se que a facilidade de customizar novas refatorações oferecidas pelo RefaX-AOP compense suas limitações atuais. De qualquer forma, a avaliação realizada serviu apenas como indicativo de desempenho. Experimentos mais elaborados devem ser realizados.

Projeto	**Aspecto**	**Tempo (ms)**			
		Converter para AspectJML	**Refatorar**	**Reverter para AspectJ**	**Total**
bean	BoundPoint.aj	1549	6906	500	**8955**
spacewar	Display.aj	319	9219	271	**9809**
telecom	Billing.aj	411	6219	286	**6916**
	TimerLog.aj	135	4266	255	**4656**
	Timing.aj	278	5953	266	**6497**

Tabela 7 – Processamento da Refatoração *Extract Pointcut* com RefaX-AOP

4.7.2 Facilidade de Implementação de Refatorações

A implementação das refatorações *Extract Poincut* e *Extract Method Calls* permitiu avaliar a facilidade de construir refatorações com o processo e as ferramentas propostos. Percebeu-se que desenvolvedores com pouca ou nenhuma experiência em XSLT apresentaram dificuldades para desenvolver ou customizar refatorações utilizando a estratégia proposta. Por outro lado, desenvolvedores com domínio dos comandos básicos desta linguagem codificaram transformações de código e verificação de pré e pós-condições com grande rapidez e facilidade. Isto demonstrou que a estratégia proposta oferece um mecanismo fácil de desenvolvimento de refatorações desde que o desenvolvedor conheça a linguagem XSLT.

4.8 Conclusão

Este capítulo apresentou um processo de desenvolvimento de refatorações para AspectJ que exige apenas codificação em uma linguagem declarativa (XSLT) para definir novas refatorações. Este processo, implementado na ferramenta RefaX-AOP, utiliza o conceito de leis de programação e uma representação de código AspectJ em XML (AspectJML). Por fim, avaliou-se o desempenho e a facilidade de uso da ferramenta construída.

O próximo capítulo apresenta as considerações finais deste trabalho e identifica oportunidades de trabalhos futuros.

Capítulo 5

Conclusão

Este capítulo apresenta as conclusões deste trabalho e enumera oportunidades para pesquisas futuras.

5.1 Considerações Finais

Neste trabalho apresentamos uma estratégia de refatoração de código AspectJ que permite a customização de novas refatorações apenas com codificação em uma linguagem de programação declarativa, o XSLT. A realização deste trabalho envolveu: (i) a criação de uma representação XML para AspectJ, denominada AspectJML [53]; (ii) a definição de um processo de desenvolvimento de refatorações com base no AspectJML [53]; (iii) a adaptação de um arcabouço de refatoração Java, RefaX [56], para se adequar ao processo de desenvolvimento de refatorações proposto; (iv) a instanciação deste arcabouço modificado para construir uma nova ferramenta de refatoração para AspectJ, denominada RefaX-AOP [54]; e, por fim, (v) uma avaliação preliminar do processo através da construção de duas refatorações, *Extract Pointcut* e *Extract Method Calls*.

5.2 Trabalhos Futuros

Como uma interessante oportunidade para trabalho futuro, vislumbra-se a construção de uma ferramenta para geração automática do código XSLT a partir da definição das leis de programação definidas em cada refatoração. Além de abstrair do programador de refatorações o conhecimento sobre o esquema da representação de código em XML (no caso, o AspectJML), tal ferramenta permitiria a especificação das leis de programação em uma linguagem de mais alto nível. Outra vantagem seria o encapsulamento dos detalhes da representação de código em XML, permitindo ao programador se concentrar apenas na refatoração a ser desenvolvida. Essa característica também facilitaria uma eventual troca de

representação XML por outra equivalente, sem que a mudança de esquema XML afetasse a implementação das refatorações já existentes.

Neste trabalho construiu-se apenas duas refatorações pra exemplificar o uso e comprovar a eficácia da ferramenta RefaX-AOP. Sinaliza-se como mais uma linha de trabalho futuro a construção de novas refatorações para AspectJ. E, ainda, como o AspectJ é um superconjunto da linguagem Java, o RefaX-AOP está apto também suportar a construção de refatorações para a linguagem Java.

Um outro trabalho futuro seria melhorar a interface gráfica do *plug-in* desenvolvido para a ferramenta de refatoração. Uma sugestão para esta melhoria seria a inclusão de um *wizard* para acompanhamento do fluxo de aplicação das leis de programação.

Por fim, outro trabalho futuro importante seria a realização de uma avaliação mais sistemática das ferramentas implementadas.

Referências Bibliográficas

[1] abc, “abc: The AspectBench Compiler for AspectJ”. Disponível em http://abc.comlab.ox.ac.uk. Último acesso em 27/06/2007.

[2] Aguiar, A., David, G., Badros, G. J., “JavaML 2.0: Enriching the Markup Language for Java Source Code”. Em XML: Aplicações e Tecnologias Associadas (XATA'04), Porto, Portugal, Fevereiro de 2004.

[3] AJEFW, “AJEFW: AspectJ Exception FrameWork”. Disponível em http://sourceforge.net/projects/ajefw. Último acesso em 05/07/2007.

[4] Aksit, M., Bergmans, L. M. J., Vural S., “An Object-oriented Language Database Integration Model: The Composition-filters Approach”. In Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP ’92), Springer-Verlag Lecture Notes in Computer Science, vol. 615, pp.372-395, Junho/Julho 1992.

[5] Arcoverde, R., Soares, S., Lustosa, P., Borba, P., “AJaTS: AspectJ Transformation System”. In Proceedings of the 1th ECOOP Workshop on Refactoring Tools (WRT’07), Berlin, pp. 35-36. Springer-Verlag, 2007.

[6] AspectJ, “AspectJ Project Homepage”. Disponível em http://www.aspectj.org. Último acesso em 25/06/2007.

[7] ajc, “ajc, AspectJ Compiler”. Disponível em http://www.eclipse.org/aspectj/ downloads.php. Último acesso em 27/06/2007.

[8] AOP-Migrator, “AOP-Migrator Projec Page”. Disponível em http://se.itc.it/aop-migrator. Último acesso em 13/12/2007.

[9] AspectJML, “AspectJML Project Page”. Disponível em http://aspectjml.googlepages. com. Último acesso em 25/06/2007.

[10] AspectWerzk, “AspectWerkz Project”. Disponível em http://aspectwerkz.codehaus.org. Último acesso em 25/06/2007.

[11] Avgustinov. P., Christensen, A. S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., Moor, O., Sereni, D., Sittampalam, G., Tibble, J., “Building the abc AspectJ compiler with Polyglot and Soot”. Technical Report abc-2004-4, Oxford University, 2004. Disponível em aspectbench.org. Último acesso em 04/07/2007

[12] Badros, G. J., “JavaML: A Markup Language for Java Source Code”. In Proc. of the 9th Int. Conf. on the World Wide Web, Amsterdam, The Netherlands, May 2000. Disponível em http://www.badros.com/greg/JavaML. Último acesso em 04/09/2006.

[13] BeautyJ, “BeautyJ Home Page”. Disponível em http://beautyj.berlios.de/. Último acesso em 04/07/2007.

[14] Bergmans, L. M. J., Aksit M., “Composing Crosscutting Concerns using Composition Filters”. Communications of the ACM, 44(10), pp. 51-57, October 2001.

[15] Castor, F., Oliveira, K., Souza, A., Santos, G., Borba, P., “JaTS: A Java Transformation System”. Em anais do XV Simpósio Brasileiro de Engenharia de Software (SBES’01), pp. 374–379, 2001.

[16] Cole, L., Borba, P. “Deriving Refactorings for AspectJ”. In Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, March 2005. ACM Press.

[17] Dijkstra, E. W. “Selected Writings on Computing: A Personal Perspective, chapter EDW447: On the Role of Scientific Thought”, pages 60-66. Springer Verlag, August 1974. Disponível em http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html. Último acesso em 12/06/2007.

[18] Eclipse, “Eclipse Home Page”. Disponível em http://www.eclipse.org. Último acesso em 27/06/2007.

[19] Ettinger, R., Verbaere, M., “Refactoring bugs in Eclipse”, Intellij IDEA and Visual Studio. Technical report, Computing Laboratory. Oxford University, http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports. Último acesso em 12/06/2007.

[20] eXist, “eXist Home Page”. Disponível em http://exist.sourceforge.net/. Último acesso em 30/06/2007.

[21] Filman, R. E., Elrad, T. H., Clarke S., Aksit M., “Aspect-Oriented Software Development”. Addison-Wesley, 2005.

[22] Fowler, M., Beck, K., Opdyke, W., Roberts, D., “Refactoring: Improving the Design of Existing Programs”. Addison-Wesley,. 1999.

[23] Gamma, E., Helm. R., Johnson, R., Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1994.

[24] Guyomarc'h, J., Guéhéneuc, Y., “On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics”. In Proceedings of the 9th ECOOP Workshop on Quantitative Approaches to Object-Oriented Software Engineering (QAOOSE 2005), Glasgow, U.K; Julho 2005.

[25] Hanenberg, S., Oberschulte, C., and Unland, R., “Refactoring of Aspect-Oriented Software”. In Proceedings of the 4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), pages 19–35. Springer Verlag, 2003.

[26] Hannemann, J., “Aspect-Oriented Refactoring: Classification and Challenges”. In Linking Aspect Technology and Evolution (LATE) Workshop, 2006.

[27] Hannemann, J., Fritz, T., e Murphy, G. C., “Refactoring to Aspects: an Interactive Approach”. In Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology eXchange, Anaheim, California, USA, Outubro 2003.

[28] Hannemann, J., Murphy, G. C., Kiczales, G., “Role-based Refactoring of Crosscutting Concerns”. In Proceedings of the 4th International Conference on Aspect-oriented software development, p.135-146, Março 14-18, 2005, Chicago, Illinois

[29] Harrison, W., Ossher, H., “Subject-oriented Programming (a critique of pure objects)”. In Proceedings of the 8th Annual Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'93), Washington, D.C., USA, ACM press, pp. 411-428, Setembro 1993.

[30] Hendren, L., Moor, O., Christensen, A. S., e a equipe abc. “The abc Scanner and Parser, including an LALR(1) Grammar for AspectJ”. Techrep, Programming Tools Group, Oxford University and the Sable research group, McGill University, September 2004. Disponível em http://abc.comlab.ox.ac.uk/documents/scanparse/index.html. Último acesso em 25/06/2007.

[31] Hoare, C., Hayes, I. J., Jifeng, H., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., Sufrin, B. A., “Laws of Programming”. Commun. ACM, 30(8):672–686, 1987.

[32] Holt, R. C., Winter, A., Schürr, A., “GXL: Toward a Standard Exchange Format”. In Proc. of the 7th Working Conf. on Reverse Engineering (WCRE’00), Brisbane, Australia, pp. 162–171, November 2000.

[33] Hors, A. L., Hégaret, P. L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S., “Document Object Model (DOM) Level 3 Core Specification”. Disponível em http://www.w3.org/TR/DOM-Level-3-Core/. Último acesso em 27/06/2007.

[34] Iwamoto, M., Zhao, J., “Refactoring aspect-oriented programs”. In Faisal Akkawi, Omar Aldawud, Grady Booch, Siobh´an Clarke, Jeff Gray, Bill Harrison, Mohamed Kand´e, Dominik Stein, Peri Tarr, and Aida Zakaria, editors, The 4th AOSD Modeling With UML Workshop, 2003.

[35] IPSI-XQ, “IPSI-XQ Project Page”. Disponível em http://www.ipsi.fraunhofer.de/oasys/projects/ipsixq/index_e.html. Último acesso em 05/04/2006.

[36] JPA, “Java Persistence Aspect”. Disponível em http://sourceforge.net/projects/jpa/. Último acesso em 05/07/2007.

[37] JBoss Group, “JBoss Aspect-Oriented Programming”. Disponível em http://www.jboss.org/products/aop. Último acesso em 25/06/2007.

[38] JBoss Group, “The JBoss 4 Application Server Guide”. Disponível em http://docs.jboss.org/jbossas/jboss4guide/r4/html/index.html, Subitem 15.2.3.1. Último acesso em 25/06/2007.

[39] Jikes, “Jikes Home page”. Disponível em http://jikes.sourceforge.net. Último acesso em 27/06/2007.

[40] Kersten, M., “AOP tools comparison, parts 1 and 2”. DeveloperWorks, IBM, Fevereiro de 2005. Disponível em http://www.ibm.com/developerworks/java/library/j-aopwork1/. Último acesso em 10/01/2007.

[41] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., “An Overview of AspectJ”. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP’01), Budapest, Hungary, Springer Verlag Lecture Notes in Computer Science, vol. 2072, pp. 327-353, Julho, 2001.

[42] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., “Getting Started with AspectJ”. Communications of the ACM, 44(10):59-65, Outubro, 2001.

[43] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., Irwin, J., “Aspect-Oriented Programming”. In Proc. of ECOOP’97, pp. 220-242, 1997.

[44] Laddad, R., “AspectJ in Action – Practical Aspect-Oriented Programming”. Manning, 2003.

[45] Laddad, R., “Aspect-Oriented Refactoring Series”. TheServerSide.com, Dezembro, 2003.

[46] Lieberherr, K. J., “Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns”. PWS Publishing Company, Boston, 1996.

[47] Lieberherr, K. J., Orleans, D., Ovlinger, J., “Aspect-Oriented Programming with Adaptive Methods”. Communications of the ACM, 44(10), p.39-41, Outubro, 2001.

[48] Lopes, C. V., Ngo, T. C., “The Aspect Markup Language and its Support of Aspect Plugins”, ISR Technical Report UCI-ISR-04-8, Outubro 2004.

[49] Maia, P. H. M., Mendonça, N. C., Fonseca, L. A., Andrade, R. M. C. "Um Ambiente para Refatoração de Código Java Utilizando Tecnologias XML". Em IV Workshop de Desenvolvimento Baseado em Componentes (WDBC'04), João Pessoa - PB, Brasil, 2004.

[50] Mammas, E., Kontogiannis, C., "Towards Portable Source Code Representations Using XML". In Proc. of the 7th Working Conf. on Reverse Engineering (WCRE'00), Brisbane, Australia, pp. 172–18, November, 2000.

[51] Maruyama, K., Yamamoto, S., "A CASE tool platform using an XML representation of Java source code". In Proc. IEEE International Workshop on Source Code Analysis and Manipulation, pages 158–167, 2004.

[52] Maruyama, K., Yamamoto, S., "Design and Implementation of an Extensible and Modifiable Refactoring Tool". In proc. of the 13th IEEE International Workshop on Program Comprehension (IWPC'05), pages 195-204, 2005.

[53] Melo Junior, L. S., Mendonça, N. C., "AspectJML: Uma Linguagem de Marcação para AspectJ". Em Anais do 2º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), Uberlândia-MG, Brasil, Outubro, pp.51-58, 2005.

[54] Melo Junior, L. S., Mendonça, N. C., Menezes, R. S., "Uma Ferramenta de Refatoração para AspectJ utilizando AspectJML e XSLT". Em anais da Sessão de Ferramentas do XXI Simpósio Brasileiro de Engenharia de Software (SBES'07), João Pessoa-PB, Brasil. Outubro, 2007.

[55] Melo Junior, L. S., Mendonça, N. C., Menezes, R. S., Trinta, F., "Um Processo de Construção de Refatorações para AspectJ utilizando AspectJML e XSLT". Em Anais do 1º Workshop Latino-americano de Desenvolvimento de Software Orientado a Aspectos (LA-WASP'07), João Pessoa-PB, Brasil, Outubro, 2007.

[56] Mendonça, N. C., Maia, P. H. M., Fonseca, L. A., Andrade, R. M. C., "RefaX: A Refactoring Framework Based on XML". In Proc. of the 20th IEEE Int. Conf. on Software Maintenance (ICSM'04), Chicago, IL, EUA, September 2004, IEEE pp. 147-156. Computer Society Press.

[57] Menezes, R. S., “Proposta de Construção de um Reversor de Código para a Linguagem AspectJML”. Monografia de Bacharelado em Sistemas de Informação, Faculdade Integrada do Ceará, Maio, 2007.

[58] Mens, T., Tourwé, T., “A Survey of Software Refactoring”. IEEE Transactions on Software Engineering, Vol. 30, No. 2, February, 2004.

[59] Monteiro, M., Fernandes, J., “Towards a Catalog of Aspect-Oriented Refactorings”. In 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA, March 2005. ACM Press.

[60] Nystrom, N., Clarkson, M. R., Myers, A. C., “Polyglot: An Extensible Compiler Framework for Java”. In Proc. of the 12th International Conference on Compiler Construction, volume 2622 of LNCS, pages 138–152, 2003.

[61] Ossher, H., Tarr, P., “Multi-dimensional Separation of Concerns and the Hyperspace Approach”. Software Architectures and Component Technology (Aksit M., editor), Kluwer, pp.293-323, 2002.

[62] Ossher, H., Tarr, P., “Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software”. Communications of the ACM 44(10), pp. 43-50, October 2001.

[63] Palsberg, J., Jay, C. B., “The essence of the Visitor pattern”. Technical Report 05, University of Technology, Sydney, 1997.

[64] Refatorações, “Catálogo On-line de Refatorações”. Disponível em http://www.refactoring.com/catalog/index.html. Último acesso em 22/01/2007.

[65] Roberts, D. B. “Practical Analysis for Refactoring”. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1999.

[66] Saxon, “Saxon Home Page”. Disponível em http://saxon.sourceforge.net/. Último acesso em 27/06/2007.

[67] Simic, H., Topolnik, M., “Prospects of Encoding Java Source Code in XML”. In Proc. of the 7th Int. Conf. on Telecomm. (ConTel’2003), Zagreb, Croatia, June 2003.

[68] SourceForge, “SourceForge.net Home Page”. Disponível em www.sourceforge.net. Último acesso em 05/07/2007.

[69] Tichelaar, S., Ducasse, S., Demeyer, D., Nierstrasz, O., “A Meta-model for Language-Independent Refactoring”. In Proc. of the International Symposium on Principles of Software Evolution (ISPSE 2000). Kanazawa, Japan, November 2000.

[70] W3C, “Extensible Markup Language (XML)”. Disponível em http://www.w3.org/XML. Último acesso em 25/06/2007.

[71] W3C, “XML Path Language (XPath) Version 1.0”. W3C Recommendation November 1999. Disponível em http://www.w3.org/TR/xpath/. Último acesso em 06/05/2007.

[72] W3C, “XML Schema Part 1: Structures, Part 2: Datatypes, World Wide Web Consortium Working Draft, Technical Report, October-2004. Disponível em http://www.w3.org/TR/xmlschema-1 e http://www.w3.org/TR/xmlschema-2. Último acesso em 25/06/2007.

[73] W3C, “XSL Transformations (XSLT) Version 1.0”, November 1999. Disponível em http://www.w3.org/TR/xslt. Último acesso em 25/06/2007.

[74] W3C, “XQuery 1.0 and XPath 2.0 Data Model”, November 2002. Disponível em http://www.w3.org/TR/2002/WD-query-datamodel-20021115. Último acesso em 25/07/2005.

[75] Xalan-Java, “Xalan-Java Version 2.7.0”. Disponível em http://xml.apache.org/xalan-j/. Último acesso em 05/07/2007.

[76] XML:DB, “XML:DB. XUpdate – XML Update Language Working Draft”. Disponível em http://www.xmldb.org/xupdate/. Último acesso em 25/07/2005.

Apêndice A

Código XSLT das Refatorações *Extract Pointcut* e *Extract Method Calls*

Este apêndice apresenta a implementação em XSLT das refatorações Extract Pointcut e Extract Method Calls.

Refatoração *Extract Pointcut*

```
01 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
02   <xsl:template match="/">
03     <xsl:variable name="result">
04       <xsl:for-each select="//pointcut">
05         <xsl:if test="@name='##RefaX_Param_02##'">
06           <xsl:value-of select="@name='##RefaX_Param_02##'"/>
07         </xsl:if>
08       </xsl:for-each>
09     </xsl:variable>
10     <xsl:value-of select="$result='true'"/>
11   </xsl:template>
12 </xsl:stylesheet>
```

Código XSLT da pré-condição [C1] Verificar nome do conjunto de junção.

```
01  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
02    <xsl:template match="* | @*">
03      <xsl:copy>
04        <xsl:apply-templates select="* | @*"/>
05      </xsl:copy>
06    </xsl:template>
07      <xsl:variable name="RefaX_Param_03" select="document('RefaXParam.xml')//p3/*"/>
08    <xsl:template match="//aspect[@id='##RefaX_Param_01##']//pointcut[last()]">
09      <xsl:copy-of select="../pointcut[last()]"/>
10      <pointcut id="##RefaX_Param_01####RefaX_Param_02##" idkind="pointcut" name="##RefaX_Param_02##">
11        <formal-arguments/>
12        <xsl:copy-of select="$RefaX_Param_03"/>
13      </pointcut>
14    </xsl:template>
15  </xsl:stylesheet>
```

Código XSLT da transformação [T1] Criar conjunto de junção.

```
01  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
02    <xsl:template match="* | @*">
03      <xsl:copy>
04        <xsl:apply-templates select="@* | *"/>
05      </xsl:copy>
06    </xsl:template>
07    <xsl:variable name="RefaX_Param_03" select="document('RefaXParam.xml')//p3/*"/>
08      <xsl:template match="//*[@idkind='pointcut-expr']">
09      <xsl:variable name="data" select="../*[@idkind='pointcut-expr']"/>
10      <xsl:variable name="identical">
11        <xsl:call-template name="check_identical">
12          <xsl:with-param name="comp1" select="$data" />
13          <xsl:with-param name="comp2" select="$RefaX_Param_03" />
14        </xsl:call-template>
15      </xsl:variable>
16      <xsl:variable name="is_pc">
17        <xsl:value-of select="../../pointcut/@name='##RefaX_Param_02##'"/>
18      </xsl:variable>
19      <xsl:choose>
20        <xsl:when test="$identical='true' and $is_pc='false'">
21          <formal-arguments-pointcut-expr idkind="pointcut-expr" type="pointcut-ref">
22            <pointcut-ref idref="##RefaX_Param_01####RefaX_Param_02##">
23            </pointcut-ref>
24          </formal-arguments-pointcut-expr>
25        </xsl:when>
26        <xsl:otherwise>
27          <xsl:copy-of select="../*[@idkind='pointcut-expr']"/>
28        </xsl:otherwise>
29      </xsl:choose>
30    </xsl:template>
31    <xsl:template name="check_identical">
32      <xsl:param name="comp1"/>
33      <xsl:param name="comp2"/>
34      <xsl:variable name="string1">
35        <xsl:call-template name="stringify">
36          <xsl:with-param name="node" select="$comp1"/>
37        </xsl:call-template>
38      </xsl:variable>
39      <xsl:variable name="string2">
40        <xsl:call-template name="stringify">
41          <xsl:with-param name="node" select="$comp2"/>
42        </xsl:call-template>
43      </xsl:variable>
44      <xsl:value-of select="$string1=$string2"/>
45    </xsl:template>
46    <xsl:template name="stringify">
47      <xsl:param name="node"/>
48      <xsl:for-each select="$node/*">
49        <xsl:choose>
50          <xsl:when test="boolean(local-name())">&lt;
51            <xsl:value-of select="local-name()"/>
52            <xsl:variable name="pos" select="position()"/>
53            <xsl:for-each select="@*">
54              <xsl:text> </xsl:text>
55              <xsl:value-of select="local-name()"/>="<xsl:value-of select="."/>"</xsl:for-each>
56            <xsl:call-template name="stringify">
57              <xsl:with-param name="node" select="."/>
58            </xsl:call-template>&lt;/
59            <xsl:value-of select="local-name()"/>&gt;</xsl:when>
60          <xsl:otherwise>
61            <xsl:value-of select="normalize-space(.)"/>
62          </xsl:otherwise>
63        </xsl:choose>
64      </xsl:for-each>
65    </xsl:template>
66  </xsl:stylesheet>
```

Código XSLT da transformação [T2] Usar conjunto de junção.

Refatoração *Extract Method Calls*

```
01 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
02   <xsl:template match="node(  ) | @*">
03     <xsl:copy>
04       <xsl:apply-templates select="@* | node(  )"/>
05     </xsl:copy>
06   </xsl:template>
07   <xsl:template match="//aspect[@id='##RefaX_Param_02##']//advice[@kind='before'][last()]">
08     <xsl:copy-of select="//aspect[@id='##RefaX_Param_02##']//advice[@kind='before'][last()]"/>
09     <xsl:call-template name="replace-call">
10       <xsl:with-param name="comp1" select="1"/>
11     </xsl:call-template>
12   </xsl:template>
13   <xsl:template name="replace-call" match="//send[@idref='##RefaX_Param_03##']">
14     <xsl:param name="comp1"/>
15     <xsl:if test="$comp1 = '1'">
16       <xsl:for-each select="//send[@idref='##RefaX_Param_03##']">
17         <xsl:choose>
18           <xsl:when test="parent::node()/*[position()=1]=self::node()">
19           <advice id="##RefaX_Param_04##" idkind="advice" kind="before" name="##RefaX_Param_04##">
20             <binary-pointcut-expr idkind="pointcut-expr" op="&amp;amp;&amp;amp;">
21               <method-constructor-pointcut-expr idkind="pointcut-expr" type="execution">
22                 <method-patt>
23                   <type idkind="type" name="void"/>
24                   <!--Method Name -->
25                   <class-type-dot-id simple-name-patt="{../../@name}">
26                     <!--Class Name -->
27                     <type idkind="type" name="{../../../@name}"/>
28                   </class-type-dot-id>
29                   <formal-arguments-patt>
30                     <xsl:for-each select="../../formal-arguments/formal-argument">
31                       <formal-argument-patt>    <type idkind="type" name="{./type/@name}"/>    </formal-argument-patt>
32                     </xsl:for-each>
33                   </formal-arguments-patt>
34                 </method-patt>
35               </method-constructor-pointcut-expr>
36               <binary-pointcut-expr idkind="pointcut-expr" op="&amp;amp;&amp;amp;">
37                 <formal-argument-pointcut-expr idkind="pointcut-expr" type="this">
38                   <type idkind="type" name="{../../../@name}"/>
39                 </formal-argument-pointcut-expr>
40                 <formal-arguments-pointcut-expr idkind="pointcut-expr" type="args">
41                   <formal-arguments-pointcut-patt>
42                     <xsl:for-each select="../../formal-arguments/formal-argument">
43                       <formal-argument-pointcut-patt>
44                         <type idkind="type" name="{./type/@name}"/>
45                       </formal-argument-pointcut-patt>
46                     </xsl:for-each>
47                   </formal-arguments-pointcut-patt>
48                 </formal-arguments-pointcut-expr>
49               </binary-pointcut-expr>
50             </binary-pointcut-expr>
51             <modifiers>    <modifier name="public"/>   </modifiers>
52             <type name="void" primitive="true"/>
53             <formal-arguments>
54               <formal-argument id="{../../../@id}arg{(string-length(.)*string-length(../../../@id)) mod 999 + 1}" idkind="formal" name="t">
55                 <type idkind="type" idref="{../../../@id}" name="{../../../@name}"/>
56               </formal-argument>
57               <xsl:for-each select="../../formal-arguments/formal-argument">    <xsl:copy-of select="."/>   </xsl:for-each>
58             </formal-arguments>
59             <block>    <xsl:copy-of select="."/>    </block>
60           </advice>
61         </xsl:when>
62         </xsl:choose>
63       </xsl:for-each>    </xsl:if>    </xsl:template>   </xsl:stylesheet>
```

Código XSLT da transformação [T1] Criar adendo do tipo *before*.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <p3>
    <send idkind="method" idref="##RefaX_Param_02##" message="##RefaX_Param_07##">
      <target>    <type idkind="type" idref="##RefaX_Param_04##" name="##RefaX_Param_05##"/>    </target>
      <arguments>    <formal-ref idkind="formal" idref="##RefaX_Param_06##" name="i"/>    </arguments>
    </send>
  </p3>
  <xsl:variable name="RefaX_Param_03" select="document('')//p3/*"/>
  <xsl:template match="node( ) | @*">
    <xsl:copy>    <xsl:apply-templates select="@* | node( )"/>    </xsl:copy>
  </xsl:template>
  <xsl:template match="//aspect[@id='##RefaX_Param_02##']//advice[@kind='before']">
    <xsl:variable name="identical">
      <xsl:call-template name="check_identical">
        <xsl:with-param name="comp1" select=".//send"/>
        <xsl:with-param name="comp2" select="$RefaX_Param_03"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:if test="$identical = 'true'">
      <xsl:variable name="identical_next">
        <xsl:call-template name="check_identical">
          <xsl:with-param name="comp1" select="following-sibling::advice[@kind='before']//send"/>
          <xsl:with-param name="comp2" select="$RefaX_Param_03"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:if test="$identical_next = 'true'">
        <xsl:call-template name="merge-advices">
          <xsl:with-param name="comp1" select="//*[@idkind='pointcut-expr']"/>
          <xsl:with-param name="comp2" select="following-sibling::advice[@kind='before']//send/../../*[@idkind='pointcut-expr']"/>
        </xsl:call-template>
      </xsl:if>
    </xsl:if>
    <xsl:if test="$identical = 'false'">    <xsl:copy-of select="."/>    </xsl:if>
  </xsl:template>
  <xsl:template name="merge-advices">
    <xsl:param name="comp1"/>
    <xsl:param name="comp2"/>
    <binary-pointcut-expr idkind="pointcut-expr" op="||">    <xsl:copy-of select="$comp1"/>    <xsl:copy-of select="$comp2"/>
    </binary-pointcut-expr>
  </xsl:template>
  <xsl:template name="check_identical">
    <xsl:param name="comp1"/>
    <xsl:param name="comp2"/>
    <xsl:variable name="string1">
      <xsl:call-template name="stringify">    <xsl:with-param name="node" select="$comp1"/>    </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="string2">
      <xsl:call-template name="stringify">    <xsl:with-param name="node" select="$comp2"/>    </xsl:call-template>
    </xsl:variable>
    <xsl:value-of select="$string1=$string2"/>
  </xsl:template>
  <xsl:template name="stringify">
    <xsl:param name="node"/>
    <xsl:for-each select="$node/*/*">
      <xsl:choose>
        <xsl:when test="boolean(local-name())">&lt;
          <xsl:value-of select="local-name()"/>
          <xsl:variable name="pos" select="position()"/>
          <xsl:for-each select="@*">
            <xsl:text> </xsl:text>
            <xsl:value-of select="local-name()"/>="<xsl:if test="not(contains(self::node(), ';arg'))"><xsl:value-of select="."/></xsl:if><xsl:if
test="contains(., ';arg')"><xsl:value-of select="substring-before(., ';arg')"/></xsl:if>"</xsl:for-each>
          <xsl:call-template name="stringify">
            <xsl:with-param name="node" select="."/>
          </xsl:call-template>&lt;/
          <xsl:value-of select="local-name()"/>&gt;</xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="normalize-space(.)"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>    </xsl:stylesheet>
```

Código XSLT da transformação [T2] Mesclar adendos do tipo *before*.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <p3>
    <send idkind="method" idref="##RefaX_Param_02##" message="##RefaX_Param_07##">
      <target>
        <type idkind="type" idref="##RefaX_Param_04##" name="##RefaX_Param_05##"/>
      </target>
      <arguments>
        <formal-ref idkind="formal" idref="##RefaX_Param_06##" name="i"/>
      </arguments>
    </send>
  </p3>
  <xsl:variable name="RefaX_Param_03" select="document('')//p3/*"/>
  <xsl:template match="node( ) | @*">
    <xsl:copy>
      <xsl:apply-templates select="@* | node( )"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="//aspect[@id=##RefaX_Param_01##']//advice[@kind='before']//formal-argument[1]">
    <xsl:variable name="identical">
      <xsl:call-template name="check_identical">
        <xsl:with-param name="comp1" select="..//..//send"/>
        <xsl:with-param name="comp2" select="$RefaX_Param_03"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:if test="$identical = 'false'">
        <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:template>
  <xsl:template name="check_identical">
    <xsl:param name="comp1"/>
    <xsl:param name="comp2"/>
    <xsl:variable name="string1">
      <xsl:call-template name="stringify">   <xsl:with-param name="node" select="$comp1"/>   </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="string2">
      <xsl:call-template name="stringify">   <xsl:with-param name="node" select="$comp2"/>   </xsl:call-template>
    </xsl:variable>
    <xsl:value-of select="$string1=$string2"/>
  </xsl:template>
  <xsl:template name="stringify">
    <xsl:param name="node"/>
    <xsl:for-each select="$node/*/*">
      <xsl:choose>
        <xsl:when test="boolean(local-name())">&lt;
          <xsl:value-of select="local-name()"/>
          <xsl:variable name="pos" select="position()"/>
          <xsl:for-each select="@*">
            <xsl:text> </xsl:text>
            <xsl:value-of select="local-name()"/>="<xsl:if test="not(contains(self::node(), ';arg'))"><xsl:value-of select="."/></xsl:if><xsl:if
test="contains(., ';arg')"><xsl:value-of select="substring-before(., ';arg')"/></xsl:if>"</xsl:for-each>
          <xsl:call-template name="stringify">
            <xsl:with-param name="node" select="."/>
          </xsl:call-template>&lt;/
          <xsl:value-of select="local-name()"/>&gt;</xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="normalize-space(.)"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>   </xsl:stylesheet>
```

Código XSLT da transformação [T3] Remover parâmetro *this*.

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <p3>
    <send idkind="method" idref="##RefaX_Param_02##" message="##RefaX_Param_07##">
      <target>
        <type idkind="type" idref="##RefaX_Param_04##" name="##RefaX_Param_05##"/>
      </target>
      <arguments>
        <formal-ref idkind="formal" idref="##RefaX_Param_06##" name="i"/>
      </arguments>
    </send>
  </p3>
  <xsl:variable name="RefaX_Param_03" select="document('')//p3/*"/>
  <xsl:template match="node( ) | @*">
    <xsl:copy>
      <xsl:apply-templates select="@* | node( )"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="//aspect[@id='LAspectoMatematico;']//advice[@kind='before']//formal-argument[1]">
    <xsl:variable name="identical">
      <xsl:call-template name="check_identical">
        <xsl:with-param name="comp1" select="..//..//send"/>
        <xsl:with-param name="comp2" select="$RefaX_Param_03"/>
      </xsl:call-template>
    </xsl:variable>
    <xsl:if test="$identical = 'false'">
        <xsl:copy-of select="."/>
    </xsl:if>
  </xsl:template>
  <xsl:template name="check_identical">
    <xsl:param name="comp1"/>
    <xsl:param name="comp2"/>
    <xsl:variable name="string1">
      <xsl:call-template name="stringify">   <xsl:with-param name="node" select="$comp1"/>   </xsl:call-template>
    </xsl:variable>
    <xsl:variable name="string2">
      <xsl:call-template name="stringify">   <xsl:with-param name="node" select="$comp2"/>   </xsl:call-template>
    </xsl:variable>
    <xsl:value-of select="$string1=$string2"/>
  </xsl:template>
  <xsl:template name="stringify">
    <xsl:param name="node"/>
    <xsl:for-each select="$node/*/*">
      <xsl:choose>
        <xsl:when test="boolean(local-name())">&lt;
          <xsl:value-of select="local-name()"/>
          <xsl:variable name="pos" select="position()"/>
          <xsl:for-each select="@*">
            <xsl:text> </xsl:text>
            <xsl:value-of select="local-name()"/>="<xsl:if test="not(contains(self::node(), ';arg'))"><xsl:value-of select="."/></xsl:if><xsl:if
test="contains(., ';arg')"><xsl:value-of select="substring-before(., ';arg')"/></xsl:if>"</xsl:for-each>
          <xsl:call-template name="stringify">
            <xsl:with-param name="node" select="."/>
          </xsl:call-template>&lt;/
          <xsl:value-of select="local-name()"/>&gt;</xsl:when>
        <xsl:otherwise>
          <xsl:value-of select="normalize-space(.)"/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each>
  </xsl:template>   </xsl:stylesheet>
```

Código XSLT da transformação [T4] Remover parâmetro.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo