UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
DEPARTAMENTO DE INFORMÁTICA E MATEMÁTICA APLICADA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

Dissertação de Mestrado

MaRiSA-AOCode: Uma Abordagem Genérica para Geração de Código Orientado a Aspectos

Everton Tavares Guimarães

Natal – RN

Março/2010

EVERTON TAVARES GUIMARÃES

MaRiSA-AOCode: Uma Abordagem Genérica para Geração de Código Orientado a Aspectos

Dissertação submetida ao Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte como requisito para a obtenção do grau de Mestre em Sistemas e Computação (MSc.).

Orientador:
Profª. Dra. Thaís Vasconcelos Batista

Co-Orientador
Profº Dr. Nélio Cacho

Natal – RN

Março/2010

Agradecimentos

A Deus, por sempre ter-me abençoado e me dado forças em todos os momentos de minha vida e por me permitir transpor todas as barreiras e obstáculos com os quais me deparei durante o mestrado.

A Thaís, pelas orientações, pela disponibilidade, pela competência, pelos conhecimentos fornecidos e por ter confiado e acreditado na minha capacidade de concretizar esse trabalho. Pelas muitas oportunidades que foram dadas, pela paciência e compreensão.

A Nelio, pela co-orientação, pela disponibilidade, paciência, aliás muita paciência, pelos conhecimentos fornecidos e pelas contribuições em meu trabalho. Agradeço também por ter acreditado no trabalho e na minha capacidade.

A meus pais, Álvaro e Marilsa, que embora à distância, sempre me apoiaram em todos os momentos. Obrigado por tudo que fizeram por mim e por todos os incentivos e esforços que sempre me deram.

Aos meus amigos, Ana Luisa, Heitor, Cássio, Lucas, Matheus, Cleverton, dentre tantos outros, por estarem sempre presentes e me apoiarem durante todo o curso. Pelo companheirismo e pelo incentivo na conclusão desse trabalho.

Ao Minora e Renato, meus quase irmãos, agradeço pela paciência, pelo apoio, pelo incentivo na conclusão deste trabalho. Obrigado pelo companheirismo, pela camaradagem, pela consideração e pela amizade.

Aos colegas do projeto CENPES/SOLAR, Jair, Gabriel, Eiji, André, Lirisnei pelo profissionalismo, companheirismo, amizade, camaradagem. Agradeço pelo apoio, incentivo e suporte que me deram durante o mestrado.

A UFRN e seus professores, pelo ensino, aprendizado e pelo amadurecimento que obtive enquanto pessoa e profissional.

RESUMO

Atualmente existem diversas abordagens orientadas a aspectos que estão relacionadas às diferentes fases do processo de desenvolvimento de software. Essas abordagens geralmente não possuem integração entre si e seus modelos e artefatos não estão alinhados dentro de um processo coerente. A integração entre o desenvolvimento de software orientado a aspectos (DSOA) e o desenvolvimento dirigido a modelos (MDD) permite propagação automática entre modelos de uma fase para outra, evitando perda de informações e decisões importantes estabelecidas em cada uma delas. Este trabalho apresenta uma abordagem dirigida a modelos, denominada MaRiSA-AOCode, que suporta a transformação de artefatos de projeto detalhado em código para diferentes linguagens de Programação Orientada a Aspectos. A abordagem proposta por MaRiSA-AOCode define regras de transformação entre *aSideML*, uma linguagem de modelagem para projeto detalhado orientado a aspectos, e *Metaspin*, um metamodelo genérico para linguagens de programação orientadas a aspectos. A instanciação do metamodelo genérico (Metaspin) provido pela abordagem de MaRiSA-AOCode é ilustrada através da transformação do Metaspin para duas linguagens: AspectLua e CaesarJ. Ilustramos a abordagem com um estudo de caso baseado no sistema *Health Watcher*.

Palavras-chave: Desenvolvimento Orientado a Aspectos, Desenvolvimento Dirigido a Modelos, Projeto Detalhado, Programação Orientada a Aspectos, Metaspin, aSideML.

ABSTRACT

Currently there are several aspect-oriented approaches that are related to different stages of software development process. These approaches often lack integration with each other and their models and artifacts are not aligned in a coherent process. The integration of Aspect-Oriented Software development (AOSD) and Model-Driven Development (MDD) enables automatic propagation of models from one phase to another, avoiding loss of important information and decisions established in each. This paper presents a model driven approach, called Marisa-AOCode, which supports the processing of detailed design artifacts to code in different Aspect-Oriented Programming languages. The approach proposed by Maris-AOCode defines transformation rules between aSideML, a modeling language for aspect-oriented detailed design, and Metaspin, a generic metamodel for aspect-oriented programming languages. The instantiation of the generic metamodel (Metaspin) provided by the approach of Maris-AOCode is illustrated by the transformation of Metaspin for two languages: AspectLua and CaesarJ. We illustrate the approach with a case study based on the Health Watcher System.

Keywords: Aspect-Oriented Software Development, Model-Driven Development, Detailed Project, Aspect-Oriented Programming, Metaspin, aSideML.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Nas últimas décadas, a engenharia de software vem pesquisando diversas formas de melhorar a qualidade e reduzir o custo do desenvolvimento de sistemas de software. Além de ter que lidar com constantes mudanças nos requisitos, projetos de software geralmente precisam tratar de mudanças de tecnologias. Embora a engenharia de software tenha tido diversos avanços, ainda existem discussões acerca da qualidade dos artefatos produzidos nas diversas fases do processo de desenvolvimento de software, bem como do quão as atividades do processo de desenvolvimento estão integradas, de forma a permitir melhor manutenibilidade e evolução dos artefatos, preservação das informações e a identificação de características transversais ao longo das fases do ciclo de desenvolvimento de software. Nesse contexto surgiram novos paradigmas que endereçam problemas relacionados ao projeto e a evolução de sistemas de software. Dentre esses estão o *Desenvolvimento de Software Orientado a Aspectos* (DSOA) e o *Desenvolvimento Dirigido a Modelos* (MDD).

O Desenvolvimento de Software Orientado a Aspectos (DSOA) [Filman et al, 2005] permite melhorar a modularização de sistemas de software por meio do encapsulamento de características transversais em abstrações denominadas *aspectos*. Características transversais são aquelas que se encontram entrelaçadas ou sobrepostas aos elementos base do sistema, de forma a restringir ou influenciar umas as outras, tornando o sistema complexo e de difícil análise. Distribuição, sincronização, tratamento de erros, depuração, e monitoramento são exemplos de características transversais comumente encontradas em sistemas de software. Existem várias propostas no contexto do DSOA que provêm meios sistemáticos para a identificação, modularização, representação e composição de características transversais [Rashid et al, 2003]. Embora a maioria dos trabalhos tenha enfoque na construção de linguagens de programação orientadas a aspectos (POA), frameworks e plataformas, há um numero de métodos e técnicas também focados em endereçar características transversais no nível de análise e projeto [Chitchyan et al, 2005].

O Desenvolvimento Dirigido por Modelos (MDD) [Stahl et al, 2006] utiliza uma abordagem interativa e *top-down* para desenvolver sistemas, onde modelos são os principais artefatos de desenvolvimento. Os modelos são associados a diferentes fases do processo de software, sendo especificados com base em rigores semânticos definidos em alguma linguagem de descrição de modelos. Adicionalmente, modelos fornecem visões e abstrações para representar diferentes elementos de diversas camadas de abstração e permitem a integração das fases do processo de desenvolvimento através da derivação de modelos a partir

de outros modelos utilizando transformações e especificação de regras de transformações entre tais modelos. As tecnologias que provêem suporte a MDD buscam produzir automaticamente ou semi-automaticamente, artefatos de software executáveis a partir de modelos.

Embora DSOA e MDD sejam paradigmas distintos em diversos aspectos, atualmente existem alguns trabalhos [Stein et al, 2002], [Baniassad and Clarke, 2004], [Chavez e Lucena, 2002], [Batista et al, 2008] [Alves et al, 2008] que integram DSOA e MDD, explorando a combinação entre ambos de forma a promover uma melhor integração entre as fases de desenvolvimento de software. A combinação entre tais paradigmas pode trazer diversos benefícios, visto que ambas as tecnologias têm por objetivo melhorar a modularidade e manutenibilidade de sistemas de software. Como exemplo, DSOA pode ser utilizado para separar (em nível de modelos) as características transversais da funcionalidade base da aplicação. Por outro lado, o MDD permite a especificação de regras de mapeamento com o intuito de gerar artefatos executáveis a partir das características base e transversais definidos nos níveis superiores. Dessa forma, a separação de características transversais pode ser mantida em todos os níveis de abstração, garantindo uma melhor modularidade e manutenibilidade do sistema de software.

A utilização de transformação de modelos pode trazer diversos benefícios, dentre eles: (i) consistência, encapsulamento, modularização e reutilização dos artefatos produzidos; (ii) redução do *gap* entre modelos envolvidos na transformação; (iii) rastreabilidade de alterações entre os modelos; (iv) automatização da transformação entre modelos, simplificando o processo e diminuindo os erros gerados por fatores humanos; e (v) redução do retrabalho na modelagem e modificação de modelos. No entanto, a definição de regras de mapeamento entre modelos abstratos e artefatos OA executáveis não é uma tarefa trivial. De forma geral, isto ocorre em virtude: (i) da necessidade de gerar código OA que respeite a complexa sintaxe/semântica das linguagens de programação, (ii) do fato das diferentes linguagens de programação orientadas a aspectos (OA) fornecerem diferentes abstrações e mecanismos de composição, o que na prática complica ainda mais o processo de geração das regras de mapeamento [Guimarães et al, 2009].

O presente trabalho tem como enfoque as fases finais do desenvolvimento de software orientado a aspectos visando integrar e efetivar a comunicação entre as fases de projeto detalhado e codificação utilizando a representatividade de modelos e recursos para transformações dirigidas a modelos. O presente trabalho está inserido em um contexto mais amplo que visa integrar e comunicar as fases de desenvolvimento de software. Em [Medeiros,

2007] e [Medeiros, 2008] foi desenvolvido o MARISA (***Ma**pping **R**equ**i**rements do **S**oftware **A**rchicteture*), uma abordagem dirigida a modelos que automatiza o processo de mapeamento e integração entre as fases de requisitos e arquitetura, onde foram utilizados AOV-Graph, uma linguagem de requisitos orientados a aspectos e AspectualACME, uma linguagem de descrição arquitetural orientada a aspectos, respectivamente. Em [Medeiros, 2008] e [Medeiros, 2009] é apresentada MARISA-DP, um abordagem baseada em modelos que integra as fases de arquitetura e projeto detalhado orientado a aspectos, a qual recebe uma descrição arquitetural originada de uma especificação de requisitos. Esse modelo de entrada utilizado por MARISA-DP é resultado do mapeamento de requisitos para arquitetura apresentado em [Medeiros et al, 2007]. Em [Guimarães, 2009a] apresentamos MARISA-Lua, uma abordagem dirigida a modelos que integra as fases de projeto detalhado e codificação orientadas a aspectos. Nesse trabalho foi utilizada a linguagem de modelagem aSideML para a representação do projeto detalhado, e AspectLua para a codificação OA. A especificação de projeto detalhado utilizado em MARISA-Lua é advinda dos modelos OA resultantes do processo de mapeamento definido em [Medeiros, 2008]. Logo, MARISA-Lua mantém as informações e decisões importantes realizadas desde a fase de requisitos, visto que as abordagens de MARISA apresentadas anteriormente estão integradas de forma a representar o processo de desenvolvimento de software orientado a aspectos como um todo.

MaRiSA-Lua permite a geração automatizada de código orientado a aspectos na linguagem AspectLua a partir da especificação de um projeto detalhado, através de regras de mapeamento especificadas entre os elementos definidos nesses dois níveis de abstração. No entanto, para fazer geração automática para uma outra linguagem de programação, a partir de um mesmo projeto detalhado utilizado em MaRiSA-Lua, é necessária a redefinição das regras de mapeamento para os elementos dessa outra linguagem. Nesse sentido, como forma de prover reusabilidade de regras de mapeamento em [Guimarães et al, 2009b] definimos um nível de abstração intermediário e regras de mapeamento entre o projeto detalhado para esse nível abstrato, e deste ultimo para uma linguagem de programação específica. Esse nível corresponde a um nível abstrato de linguagens de programação OA, com o objetivo de permitir a modelagem orientada a aspectos independente de plataforma e promover a reusabilidade de regras de mapeamento entre os níveis de projeto detalhado e a programação OA. A idéia consiste em reduzir o *gap* conceitual entre os níveis de projeto e de programação OA, fornecendo uma maior facilidade na definição de regras de mapeamento entre um nível abstrato de programação para uma linguagem específica na qual se deseja obter modelos significativos, gerando assim código OA de forma automatizada.

1.1 Motivação

Atualmente, existem diversas abordagens relacionadas às fases de projeto detalhado e codificação orientada a aspectos [Evermann, 2007], [Fuentes e Sánchez, 2007] e [Groher e Baum'garth, 2004], que promovem a integração entre essas fases trazendo melhorias na manutenibilidade e rastreabilidade das informações no processo de desenvolvimento. A integração e mapeamento entre elementos das fases de projeto detalhado e codificação são necessários para manter decisões tomadas em cada uma destas etapas, e garantir a correspondência e a diminuição da distância entre modelos e artefatos produzidos durante estas fases. Além disso, o mapeamento reduz o esforço na geração do código inicial derivado do projeto detalhado, e permite a propagação de informações advinda desde a fase de requisitos, arquitetura, e o projeto detalhado. As decisões tomadas desde as primeiras fases são propagadas para as fases subseqüentes do desenvolvimento. A integração das fases de projeto detalhado e codificação realizada através do mapeamento intermediário que provê um nível abstrato de programação pode trazer diversos benefícios: dentre eles: (i) redução do *gap* entre os modelos gerados em cada fase; (ii) diminuição do retrabalho na manutenção de modelos através da reutilização de regras de mapeamento e dos artefatos gerados no nível abstrato de programação; (iii) possibilidade de rastreamento entre as características transversais e elementos dos modelos; e (iv) evita a perda de informações e decisões importantes tomadas em fases iniciais do desenvolvimento de software.

A definição do mapeamento entre modelos abstratos e artefatos executáveis não é uma tarefa trivial, pois: (i) a necessidade de geração de código que possua sintaxe e semântica coerente com as linguagens de programação requer grande esforço na definição de um número elevado de regras de mapeamento; e (ii) as linguagens de programação existentes oferecem diversas abstrações para a modularização e composição de características transversais, complicando de certa forma a definição de regras de mapeamento. Logo, o mapeamento para linguagens de programação específicas possui um nível de reusabilidade muito baixo, visto que para diferentes linguagens o mapeamento deve ser refeito.

Nesse contexto, surge a necessidade da definição de um nível de abstração que permita o mapeamento do projeto detalhado para um metamodelo genérico de linguagens de programação orientadas a aspectos como forma de: (i) promover a reusabilidade das regras de mapeamento entre o projeto detalhado e a codificação OA; (ii) permitir rastreabilidade entre os modelos gerados; (iii) facilitar a localização de modificações nos modelos; (iv) reduzir o esforço na modelagem de transformações entre o projeto detalhado para diversas linguagens

de programação OA; (v) reduzir o *gap* entre a modelagem do nível de projeto detalhado para linguagens de programação OA; (vi) facilitar a geração de código a partir de um nível de implementação abstrato para uma ou mais linguagens de programação OA específicas de plataforma utilizando templates predefinidos para cada linguagem; e (vii) diminuir o esforço do desenvolvedor na construção de regras de mapeamento para uma nova linguagem orientada a aspectos específica.

Como forma de abstrair a complexidade entre modelos definidos em nível de projeto detalhado e a geração de artefatos executáveis para diversas linguagens OA, o presente trabalho apresenta uma abordagem baseada em modelos, que possibilita a geração de código para diversas linguagens de programação orientadas a aspectos a partir de um projeto detalhado OA. Isso é possível através da definição de um nível de mapeamento intermediário onde é utilizado um metamodelo independente de plataforma que encapsula as principais abstrações de diversas linguagens de programação OA.

Esse metamodelo, o Metaspin [Brichau et al, 2005], consiste em um metamodelo genérico de linguagens de programação orientadas a aspectos que representa as características essenciais de orientação a aspectos presentes nessas linguagens. O Metaspin é resultado de estudos [Brichau e Haupt, 2005] realizados entre diversas linguagens de programação, analisando as similaridades e diferenças entre elas. Atualmente, existem outras abordagens de metamodelos abstratos para a representação de linguagens de programação OA, tais como [Uetanabara et al, 2009] e [Havinga et al, 2006], no entanto tais abordagens não consideram uma grande variedade de linguagens orientadas a aspectos, bem como não avaliam os modelos de execução de cada uma delas

Metaspin foi escolhido pelos seguintes motivos: (i) prover um nível independente de plataforma para representação de conceitos de linguagens de programação OA; e (ii) ter sido resultado de estudos de similaridades e diferenças de mais de 27 linguagens de programação OA, bem como o modelo de execução de cada uma delas. Além disso, não foi encontrado na literatura nenhum outro estudo propondo um metamodelo genérico e extensível para diversas linguagens de programação orientadas a aspectos e com diferentes paradigmas de programação.

Nossa abordagem tem por objetivo tornar possível o mapeamento entre as atividades do ciclo de Desenvolvimento de Software Orientado a Aspectos, bem como a propagação de informações entre elas, definindo transformações dos artefatos da fase de projeto detalhado e codificação, através de regras de mapeamento que definimos em [Guimarães et al, 2009].

Para a descrição do projeto detalhado foi escolhida a linguagem de modelagem aSideML, uma extensão não conservativa da UML que provê semântica, notação e regras para dar suporte a Modelagem Orientada a Aspectos (MOA), e trata aspectos e *crosscutting* como elementos de primeira classe [Chavez, 2004]. Além disso, como citado anteriormente a linguagem aSideML tem sido utilizada em outros trabalhos que fazem parte do macro processo de modelagem do desenvolvimento de software orientado a aspectos no qual esse trabalho está inserido. Além disso, em virtude de aSIdeML ser uma extensão da UML, é Possível que algumas regras de mapeamento entre o nível de projeto e o nível abstrato de programação sejam reusáveis para outras linguagens que também são extensões da UML.

Para a representação de modelos em um nível abstrato de programação utilizamos o Metaspin, por ser um metamodelo genérico e extensível para diversas linguagens de POA. Além disso, o Metaspin provê um nível independente de plataforma para a representação de aspectos e permite representar aspectos em diferentes paradigmas de programação.

Para a codificação OA foram escolhidas as linguagens de programação AspectLua [Cacho et al, 2005] e CaesarJ [Aracic et al, 2006]. Embora essas linguagens de POA possuam mecanismos baseados em AspectJ [REFERENCIA]para o encapsulamento de características transversais, seus paradigmas de programação são diferentes. AspectLua é uma linguagem de *scripting*, simples, dinamicamente tipada e que possui paradigma estrutural. Como linguagem base utiliza Lua [Ierusalimschy, 2006]. Por sua vez, CaesarJ [Aracic et al, 2005] é uma linguagem de POA com forte suporte a reusabilidade, que combina construções orientadas a aspectos, com mecanismos avançados de modularização orientada a objetos. Tem como linguagem base a linguagem Java [Sun, 2009].

O mapeamento entre projeto detalhado e código fonte é realizado através da especificação de transformações entre modelos aSideML e Metaspin e deste para as linguagens AspectLua e CaesarJ. Em se tratando de MDA, o nosso trabalho contém modelos que estão inseridos em três, dos quatro níveis que a arquitetura dirigida a modelos possui (ver Figura 1).

No nível PIM estão os modelos aSideML e Metaspin (em conformidade com seus respectivos metamodelos), visto que são modelos independentes de plataforma, não possuindo qualquer descrição de tecnologia específica. No nível PIM temos os modelos AspectLua e CaesarJ (em conformidade com seus respectivos metamodelos), visto que são modelos relacionados a linguagens orientadas a aspectos específicas de plataforma. Por fim no nível CODE, temos as descrições textuais, ou seja, os códigos base e orientados a aspectos gerados por meio das transformações especificadas em nosso trabalho.

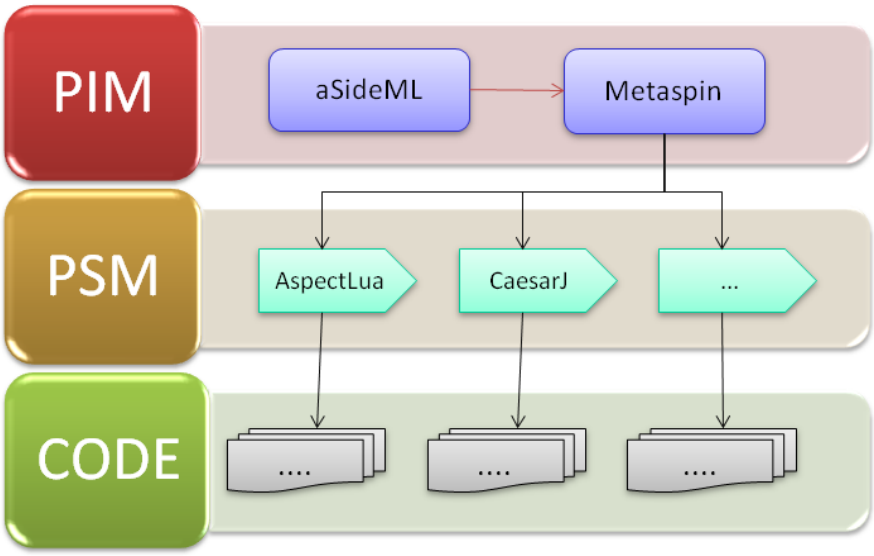

Figura 1 - mapeamento dos modelos na arquitetura MDA

Esse trabalho utiliza técnicas MDD para a integração entre modelos orientados a aspectos, das fases de projeto detalhado e codificação OA, trazendo diversos benefícios tais como: (i) geração automatizada de código OA em AspectLua e CaesarJ, a partir de modelos de projeto detalhado em aSideML, através do uso de transformações dirigidas a modelos e templates de geração de código. Com a utilização de técnicas MDD para a automatização de transformações entre modelos, o processo torna-se menos repetitivo e menos suscetível a falhas ou erros causados por fatores humanos; (ii) redução do retrabalho de modelagem através da reutilização de algumas regras de mapeamento e mudanças entre os modelos aSideML, Metaspin, AspectLua e CaesarJ; (iii) facilidade de rastrear mudanças e características transversais tanto em modelos de projeto quanto no código; (iv) redução do *gap* entre estas etapas de desenvolvimento e entre os modelos construídos em cada uma delas; (v) análise qualitativa dos modelos resultantes das transformações quanto a rastreabilidade, completude e corretude.

1.2 Objetivos

O presente trabalho tem por objetivo definir transformações entre modelos orientados a aspectos nas fases de projeto detalhado, representada por aSideML, o modelo intermediário de abstração que representa as principais características de linguagens de programação OA, representado pelo Metaspin, e código fonte, representado por AspectLua e CaesarJ. As

transformações são realizadas do nível de projeto detalhado para o nível abstrato que visa representar os principais conceitos de orientações a aspectos, tais como *advice*, *pointcut*, *join points* e declaração inter-tipos. A partir do nível de abstração proporcionado pelo Metaspin, existe um mapeamento do Metaspin para os conceitos da linguagem de programação OA em que se deseja obter o código, de forma que as informações contidas em cada abstração podem ser de fato transformadas em código OA.

Os objetivos específicos deste trabalho são:

- Especificação de metamodelos de aSideML, Metaspin, AspectLua e CaesarJ.
- Implementação e validação dos metamodelos aSideML e Metaspin.
- Especificação em ATL (*ATLAS Transformation Language*) das transformações entre os modelos de aSideML, Metaspin, sendo essas transformações baseadas nos respectivos metamodelos de cada linguagem
- Especificação de regras de transformação ATL entre o Metaspin e as linguagens de POA especificas de plataforma AspectLua e CaesarJ.
- Especificação de templates textuais para geração de código AspectLua e CaesarJ a partir dos seus respectivos modelos.
- Implementação das transformações entre modelos OA, utilizando ATL, bem como o ambiente MDA oferecido pela IDE Eclipse.
- Geração de código nas linguagens AspectLua e CaesarJ a partir do modelo Metaspin e dos templates textuais de cada linguagem de programação orientada a aspectos, utilizando o plugin Acceleo [Acceleo, 2009].
- Sincronização das regras de mapeamento dos modelos gerados em cada etapa concluindo a abordagem de MaRiSA-AOCode.
- Ilustração das transformações entre modelos através do estudo de caso do sistema Health Watcher.

1.3 Estrutura do trabalho

O restante desse trabalho está organizado em seis capítulos. O capítulo 2 apresenta os principais conceitos envolvidos no trabalho, que são: Desenvolvimento de Software Orientado a Aspectos, Projeto Detalhado, Codificação e Linguagens de Programação Orientadas a Aspectos, Metaspin e por fim, Desenvolvimento Dirigido a Modelos. O capítulo 3 apresenta a definição das regras de transformações entre elementos dos modelos de projeto

detalhado (em aSideML, modelos abstratos de programação (expressos pelo Metaspin) e modelos de linguagens orientadas a aspectos específicas de plataforma, tais como AspectLua e CaesarJ. O capitulo 4 introduz a implementação das regras de mapeamento entre os níveis de projeto detalhado e programação orientado a aspectos, por meio de regras de transformação dirigidas a modelos utilizando para tanto, a linguagem de transformação ATL. Adicionalmente, o capitulo mostra como os modelos são gerados por meio das regras e como esses modelos são avaliados segundo critérios de qualidade bem definidos. O capítulo 5 apresenta trabalhos relacionados ao contexto de integração entre desenvolvimento de software orientado a aspectos e o desenvolvimento dirigido a modelos, bem como um breve comparativo entre tais abordagens e a abordagem apresentada neste trabalho. Por fim, o capítulo 6 apresenta as considerações finais, contribuições e trabalhos futuros..

2 Fundamentação Teórica

O objetivo deste capítulo é apresentar, de forma resumida, os principais conceitos que estão envolvidos neste trabalho. Na Seção 2.1 apresentamos uma visão do Desenvolvimento Orientado a Aspectos (DSOA). Na Seção 2.2 apresentamos os conceitos relacionados à Modelagem Orientada a aspectos e a linguagem aSideML. A Seção 2.3 trata de conceitos relacionados à Programação Orientada a Aspectos e as linguagens de Programação Orientadas a Aspectos, AspectLua e CaesarJ. Na Seção 2.4 apresentamos o Metaspin, um metamodelo genérico para linguagens de programação OA. Por fim, a Seção 2.5 apresenta o MDD (*Model Driven Development)* e as ferramentas que serão utilizadas neste trabalho para transformação de modelos.

2.1 Desenvolvimento de Software Orientado a Aspectos (DSOA)

O Desenvolvimento de Software Orientado a Aspectos tem se mostrado uma alternativa viável para a modularização e composição de características transversais ao longo de todo o processo. Dessa forma, o DSOA demonstra sua grande importância no que se refere à qualidade e nível de complexidade dos sistemas de software, nas diversas fases que constituem o seu desenvolvimento. Isso demonstra a evolução constante e o surgimento de novos paradigmas e linguagens de programação. Como exemplo, a Programação Orientada a Objetos (POO) [Rentsch, 1982] surgiu com o objetivo de encapsular dados e comportamento em uma abstração denominada objeto e, permitir a interação entre tais objetos por meio de operações disponibilizadas por suas respectivas interfaces.

Com o advento da POO o desenvolvimento de sistemas de software tornou-se mais reutilizável, flexível, com maior facilidade na manutenção e no desenvolvimento de módulos que encapsulam as funcionalidades específicas do sistema [Grott, 2006]. Embora a POO tenha representado uma evolução no desenvolvimento de software, ainda existem problemas que podem ser identificados.

O desenvolvimento de um sistema baseado no paradigma de orientação a objetos geralmente é formado por: conceitos fundamentais e características transversais. Os conceitos fundamentais do sistema, o código base, estão relacionado ao propósito básico do sistema. As características transversais (crosscutting concerns) [Kiczales et al, 1997] estão relacionadas ao código que se encontra espalhado e entrelaçado em vários pontos do código base. Esse código acaba por ultrapassar a responsabilidade de uma classe de objetos. O entrelaçamento e

espalhamento de código dificultam a compreensão e manutenção do sistema de software. Através da separação de características transversais é possível prover melhorias na manutenção, modularização, reutilização e evita a replicação de código.

Nesse sentido, surgiu a Programação Orientada a Aspectos (POA) com o objetivo de preencher a lacuna da POO, através de uma nova abordagem de desenvolvimento. A POA utiliza aspectos para a separação entre código base e o código que implementa as características transversais do sistema. Embora a maioria dos trabalhos iniciais no DSOA tenha se focado no desenvolvimento de linguagens de programação orientadas a aspectos, frameworks e plataformas, um número de técnicas e métodos também tem sido propostos para endereçar características transversais no nível de analise e projeto [Chitchyan et al, 2005]. A idéia da POA tem sido refletida para as demais fases do ciclo de desenvolvimento de software, tais como arquitetura e projeto, definindo estratégias de DSOA para todo o ciclo de vida do software.

2.2 Programação Orientada a Aspectos – POA

A POA oferece suporte à separação de características transversais no nível de codificação. A POA considera melhorias na separação de características proporcionadas por paradigmas anteriores como o de Orientação a Objetos, além de fornecer suporte a novos mecanismos para tratar características transversais que não são tratadas adequadamente.

A idéia central da POA é que enquanto os mecanismos de modularização hierárquica de linguagens orientadas a objetos são extremamente úteis, eles são intrinsecamente incapazes de modularizar todas as características transversais em sistemas complexos [Kiczales et al, 2001].

A POA propõe o conceito de aspectos [Bakker et al, 2005] para a representação das características transversais (*crosscutting concerns*) [Kiczales et al, 1997] de um sistema e, tem por objetivo ser uma nova técnica que deve ser utilizada em conjunto com linguagens de programação, de forma a construir sistemas de software de melhor arquitetura, auxiliando na manutenção dos vários interesses e na compreensão do software. Os conceitos comumente definidos na POA são: aspectos, *join points*, *pointcuts*, *advice* e *introduction*.

Aspectos são definidos como unidades modulares, designadas para implementar e encapsular características transversais por meio de instruções sobre onde, quando e como eles são invocados [Filman et al, 2005]. Dessa forma, o aspecto promete uma maior modularidade e reduz o espalhamento e entrelaçamento do código base com a parte transversal do sistema

tornando-o, assim, mais fácil de manter. Segundo Filman, dependendo da linguagem de aspecto utilizada, os aspectos podem ser construídos de forma hierarquica, e essa linguagem pode prover meios para a separação de interesses de forma a permitir a definição do aspecto e especificação a interação do aspecto com os componentes do sistema em questão.

Os *join points* são lugares bem definidos na estrutura ou fluxo de execução de um programa, onde comportamentos adicionais podem ser inseridos [Filman et al, 2005]. Em relação a programação OA, podemos dividir os *join points* em estáticos e dinâmicos. Em [Chavez, 2004] define-se que: um ponto de junção estático (*static join point*) é um local na estrutura estática de um componente, enquanto que um ponto de junção dinâmico (*dynamic join point*) é um local no comportamento dinâmico de um programa de componentes.

Os *pointcuts* reúnem um conjunto de *join points* definindo o conceito de quantificação, que é um caminho para relacionar algo importante em muitos locais de um programa com uma simples declaração [Filman et al, 2005]. Ou seja, eles indicam um conjunto de *join points* que são afetados por uma ou mais características transversais.

O *advice* é um conjunto de operações do programa que são executadas quando *join points* especificados são alcançados por um *pointcut* [Filman et al. 2005]. Normalmente, cada *advice* possui um *pointcut* a ele associado. Tal *pointcut* determina os join points onde o *advice* será executado. As declarações de *advice* podem ser de três tipos: (i) *before*, executa no momento em que um *join point* é alcançado; (ii) *after*, executa no momento em que o controle retorna através do ponto de junção; e (iii) *around*, executa quando o *pointcut* é alcançado e tem controle explícito sobre quando o próprio método afetado deve ser executado.

O *introduction* ou declaração inter-tipos é um mecanismo utilizado pelo aspecto para introduzir novos elementos no sistema, fornecendo simplicidade, reflexão e modularização ao código, já que novos membros podem ser adicionados ao código sem a necessidade de se fazer alterações diretamente na especificação do programa. Esses novos elementos a serem introduzidos no sistema podem ser atributos, métodos, declaração de implementação de interface ou extensão de classes.

Atualmente existem diversas linguagens de POA. Algumas delas podem ser encontradas em [Brichau e Haupt, 2005], onde é apresentado um estudo analisando as linguagens e seus respectivos modelos de execução. Nas seções 2.3.1 e 2.3.2 são apresentadas as linguagens de programação OA que são utilizadas nesse trabalho, AspectLua e CaesarJ, respectivamente.

2.2.1 AspectLua

Em [Cacho et al, 2005] é apresentada AspectLua, uma extensão da linguagem de programação Lua [Ierusalimschy, 2006] para dar suporte a POA. AspectLua foi desenvolvida utilizando mecanismos básicos da própria linguagem sem a necessidade de alteração do seu interpretador e provê suporte à criação de aspectos, *pointcuts*, *join points* e *advices*. AspectLua herda algumas características não-convencionais de Lua, tais como: (i) as funções são valores de primeira classe e podem retornar vários valores, eliminando a necessidade de passar parâmetros por referência e; (ii) as tabelas são o principal mecanismo de estruturação de dados. Elas implementam *arrays* associativos, criam objetos dinamicamente e podem ser indexadas por qualquer valor da linguagem (exceto nulo).

```
1. a = Aspect:new()
2. a : aspect ({name = ‘valor},
3. {pointcutname = ‘valor’, designator =’valor’, list = {‘valor’}},
4. {type = ‘valor’, action = ‘valor’})
```

Figura 2 – Declaração de um aspecto em AspectLua

Para se criar um aspecto (ver Figura 4) usa-se a função *new ()* (linha 1) que a cria de uma instância de uma classe AspectLua. Após isso, define-se uma tabela Lua contendo os elementos do aspecto (*name*, *pointcut* e *advice*). A criação de aspectos em AspectLua depende de três parâmetros: (i) O primeiro parâmetro do método define o nome do aspecto (linha 2); (ii) O segundo parâmetro define os elementos de *pointcut* (linha 3): seu nome, seu *designator* e as funções e variáveis que devem ser interceptadas. O *designator* define o tipo de pointcuts. AspectLua suporta os seguintes tipos: *call* para as chamadas de função; *callone* para a chamada de aspectos que devem ser executados somente uma vez; *introduction* para introduzir funções em tabelas (objetos em Lua); e métodos *get* e *set* aplicado sobre variáveis. O campo *list* define funções ou variáveis que deve ser interceptadas, podendo fazer o uso de *wildcards*; e (iii) o terceiro parâmetro é uma tabela Lua que define os elementos do *advice* (linha 4): o tipo (*after*, *before*, e *around*) e a ação que deve ser tomada quando um *pointcut* é atingido.

```
1.  asp : aspect = new();
2.  asp : (name = 'Persistence_in_DB',
3.  {name = 'pointcut-template1',   designator = 'call', list = {'Usability.configurability*'}},
4.  {type = 'after', action = 'Set_DB'})
```

S

Figura 3 – Representação da instanciação de um aspecto em AspectLua

A Figura 5 ilustra um exemplo da declaração do aspecto *Persistence_in_DB*, responsável pela configuração da persistência em banco de dados no *Health Watcher System* [Soares, 2002]. Na declaração do aspecto, além da instanciação do aspecto e da definição do seu nome, também são definidos o *pointcut* e o *advice*. O *pointcut* '*pointcut-template1*', define um conjunto de *join points* que é representado no parâmetro *list* (linha 3).O campo de list 'ConFigurability.*' denota que todos os métodos da classe *ConFigurability* serão *join points* e que estão contidos no *pointcut* 'pointcut-template1'. No *advice* (linha 4) o valor de seu tipo definido *after* denota que a ação de tal *advice* será executada após a execução de um *join point*. Por fim, define-se qual a ação o advice deve executar quando um *join point* for atingido que neste caso é a operação *Set_BD*. A operação *Set_DB* define qual o banco de dados a ser utilizado pelo sistema.

2.2.2 CaesarJ

Em [Aracic et al, 2005] é apresentada CaesarJ, uma linguagem orientada a aspectos que possui forte suporte a reusabilidade. A linguagem combina construções de orientação a aspectos, tais como *pointcuts* e *advices*, com avançados mecanismos de modularização orientados a objetos. A linguagem CaesarJ compartilha algumas similaridades importantes com AspectJ [Kiczales et al, 2001], sendo esta última atualmente a mais amplamente utilizada linguagem de programação orientada a aspectos.

Embora CaesarJ possua algumas similaridades em relação a AspectJ existem várias diferenças importantes. Tanto AspectJ, quanto CaesarJ são extensões de Java que dão suporte a programação OA. Entretanto, CaesarJ estende Java 1.4, enquanto que AspectJ mantém-se atualizada com as últimas versões do Java. AspectJ, por exemplo já está disponível como uma extensão do Java 6. Adicionalmente, CaesarJ não suporta construções introduzidas em versões acima do Java 2, tais como anotações, tipos genéricos e generalizados para loop.

Além das classes Java, CaesarJ provê um segundo tipo de classe, a *CaesarJ class*, a qual é equivalente aos aspectos de AspectJ. Para o CaesarJ, o aspecto é um componente como outro do paradigma Orientado a Objetos e, por tal motivo, difere de uma classe Java, inicialmente, apenas pela substituição da palavra chave class pela ***cclass*** [Aracic et al, 2005]. Na Figura 6 é ilustrada a declaração genérica de um aspecto (linha 2) em CaesarJ utilizando a palavra chave ***cclass***. A instanciação de um objeto aspecto pode ser automática, ao utilizar a palavra chave ***deployed*** (linha 2) antes de cclass, ou programada utilizando o comando new.

Utilizando a forma programada, o desenvolvedor deve lembrar de colocar a instrução que coloca o objeto do aspecto na lista de monitores de chamadas e execução de métodos.

```
1. package umPacote;
2. public deployed cclass Persistence_in_DB () {
3.
4. pointcut pc1 (): call (void Usability.configurability(..))
5.
6. before ():pc1 () {
7.     System.out.println ("before advice");
8. }
9. after (): pc1 () {
10.     System.out.println ("after advice");
11. }
12. around (Usability usability): pc1 () && target (usability) {
13.     System.out.println ("Aspectual");
14.     proceed(usability);
15. }
```

Figura 4 – Declaração de um aspecto genérico com definição de pointcut e advice

A declaração de pointcuts e advices é feita de forma similar a AspectJ. Um *pointcut* é definido pela palavra chave ***pointcut*** (linha 4 e linha 5) seguida do nome do pointcut e a declaração do conjunto de dos *join points a* que o mesmo faz referência. O tipo do *pointcut* define a forma que o aspecto acessa os *join points.* O acesso aos *join points* podem ser realizado através de primitivas: *get*, *set*, *call* e *callone.*

Assim como AspectJ, a linguagem CaesarJ declara *advices* como métodos que devem ser executados em cada join point de um *pointcut*, quando o mesmo é acessado pelo aspecto. Dessa forma, CaesarJ suporta *advices* do tipo *before* (linha 6 a linha 8), *after* (linha 9 a linha 11) e *around* (linha 12 a 15). Uma particularidade do *advice* do tipo *around* é que este pode fazer o uso ou não da chamada ***proceed*** (linha 14). A utilização da *around advice* define que quando um dado *pointcut* é atingido, a execução antes e depois do mesmo tem seu controle passado para o aspecto. Dessa forma, o aspecto é quem define se após a execução do *advice* ligado a este *pointcut* o controle da execução é devolvido para o código base ou não. Quando o programador utiliza a instrução *proceed* ele força o aspecto a devolver o controle da execução ao código base da aplicação após a execução da ação do *advice*.

Adicionalmente, CaesarJ utiliza um modelo de join point similar ao utilizado por AspectJ, com a exceção de que designador de pointcut de aspectos "if" não é suportado. Ao contrário de AspectJ, portanto, CaesarJ suporta a **implantação dinâmica** (*dynamic deployment*) de advice, o que significa que todos os advices de um aspecto em CaesarJ podem ser ativados e desativados. Por fim, o CaesarJ não tem o conceito de *Intertype declaration* como definido em AspecJ [5]. Em lugar desse conceito é utilizado o conceito de *mixin* onde é possivel apenas realizar inserções em Aspectos.

2.3 Projeto Detalhado Orientado a Aspectos

A atividade de projeto detalhado no processo de desenvolvimento de software dá ao projetista a oportunidade de pensar sobre um sistema de software a partir de um dado conjunto de requisitos [Chitchyan et al, 2005]. O projeto detalhado é uma importante fase do processo de desenvolvimento, pois é a partir dele que os engenheiros de software irão produzir diversos modelos que refletem um esboço da solução a ser implementada. Dessa forma, tais modelos podem ser analisados e avaliados para determinar se eles satisfazem ou não os requisitos do sistema. Ademais, os modelos resultantes do projeto detalhado podem ser utilizados para o planejamento de atividades subseqüentes, bem como podem ser utilizados na construção do sistema e na definição de casos de testes de software [Swebok, 2000]. A saída resultante de uma atividade de projeto é um conjunto de modelos que caracterizam e especificam o comportamento e estrutura de um sistema. Esses modelos podem estar em diferentes níveis de abstração dependendo do nível de detalhamento que o projetista deseja.

O projeto detalhado orientado a aspectos possui os mesmos objetivos de qualquer atividade de projeto: caracterizar e especificar o comportamento e estrutura do sistema de software. O projeto OA tem como foco o uso de técnicas e ferramentas para identificação, representação, estruturação e gerenciamento de características transversais. Portanto, o projeto detalhado OA visa identificar, representar e gerenciar os conceitos e propriedades relacionadas ao desenvolvimento de software orientado a aspectos que auxiliam na compreensão, evolução e reutilização de modelos orientados a aspectos. A sua contribuição para o projeto de software diz respeito às extensões da capacidade de modularização, de forma a representar características transversais de forma modular.

Uma abordagem de projeto detalhado OA pode incluir tanto uma linguagem, quanto um processo. Um processo de modelagem de projeto detalhado é aquele que recebe como entrada um conjunto de requisitos e produz um modelo de projeto que deve representar

parcialmente a arquitetura. O modelo de projeto detalhado OA produzido durante o processo representa preocupações separadas e relações entre tais preocupações. Esse modelo é uma especificação abstrata para a implementação que pode ocorrer em uma plataforma de programação orientada a aspectos e mecanismos de composição.

Uma linguagem de projeto detalhado OA é uma linguagem que inclui construções que podem descrever os elementos a serem representados em um projeto e os relacionamentos que podem existir entre tais elementos. Em particular, as linguagens de projeto detalhado OA possuem construções para suportar a modularização de características transversais, bem como a especificação de sua composição.

2.3.1 Linguagem de Modelagem Orientada a Aspectos - aSideML

Estudos realizados em [Chitchyan et al, 2005] mostram que diversas abordagens foram criadas com o objetivo de representar e modelar projetos orientados a aspectos. Dentre essas abordagens podemos citar: *Composition Patterns* [Clarke e Walker, 2001], AODM [Stein et al, 2002], Theme/UML [Baniassad e Clarke, 2004], AML [Groher e Baumgarth, 2004], UMLAUT [Coelho e Murphy, 2004], dentre outros.

Em [Chavez, 2004] é apresentada aSideML, uma linguagem de modelagem construída com o objetivo de permitir a especificação e comunicação de projetos OA. A linguagem aSideML tem como base o metamodelo aSideML, que utiliza o metamodelo de UML (*Unified Modeling Language*) [UML, 2009] como base e provê extensões (novas metaclasses e metassociações) para descrição de aspectos, características transversais e outros elementos de DSOA. Na linguagem aSideML, os aspectos e características transversais são explicitamente tratados como elementos de primeira classe. Esses modelos servem como projetos preliminares que devem ser desenvolvidos na direção dos modelos de implementação de ferramentas e linguagens de POA.

As principais características providas por aSideML são: (i) suporte a interfaces transversais (*crosscutting interfaces*) de forma explícita; (ii) suporte à descrição de aspectos como elementos de modelagem parametrizados, promovendo seu reuso; (iii) suporte à descrição explícita de relacionamento entre aspectos e os elementos que ele afeta, chamado de *relacionamento de crosscutting*; e (iv) suporte à descrição explícita de relacionamentos de dependência entre aspectos, dentre outros. Adicionalmente, aSideML provê suporte a

elementos de modelagem estruturais, comportamentais, e composicionais, relacionados a modelagem orientada a aspectos.

Tabela 1 - Dimensões da modelagem orientada a aspectos com aSideML [Chavez 2004]

Modelo	Diagramas	Perspectivas	Elementos
Estrutural	Diagrama de aspectos		Aspecto, Interface Transversal, Característica Transversal
	Diagrama de classe estendido	Centrado em aspecto	Aspecto, *Crosscutting*, Interface transversal, *Order*
		Centrado em base	
Comportamental	Diagrama de seqüência estendido	Ponto de combinação	Ponto de combinação dinâmico
	Diagrama de colaboração aspectual		Instância de aspecto, colaboração aspectual
	Diagrama de sequência		Instância de aspecto, interação aspectual
Composicional	Diagrama de classes combinadas		Classe combinada
	Diagrama de colaboração combinada		Colaboração combinada
	Diagrama de seqüência combinada		Interação combinada

A Tabela 1 mostra uma visão geral a respeito dos tipos de modelagem a que aSideML provê suporte, relacionando os modelos, diagramas, perspectivas e elementos de cada modelagem. A modelagem estrutural fornece a visão estática de um sistema na presença de aspectos. Embora aSideML ofereça suporte a três tipos de modelagem, nosso trabalho irá contemplar somente a modelagem estrutural, a qual será melhor descrita na seção 2.1.1.1.

A *modelagem comportamental* oferece a visão de interação de um sistema na presença de aspectos. Os principais elementos constituintes de modelos comportamentais são instâncias de aspectos, interações aspectuais e colaborações aspectuais. De acordo com [Chavez, 2004], a colaboração aspectual pode ser entendida como a descrição de uma organização geral de objetos e instâncias de aspectos que interagem dentro de um contexto a fim de implementar o comportamento crosscutting de uma característica transversal comportamental.

Por sua vez, a *modelagem composicional* descreve visões estáticas e de interação de um sistema após a combinação de modelos de objetos e modelos de aspectos. O resultado do processo de combinação depende da estratégia de combinação adotada. Por sua vez, a estratégia de combinação depende do modelo de implementação suportado pela ferramenta ou linguagem de programação orientada a aspectos. Os principais elementos de modelagem composicional são classes combinadas, colaborações combinadas e interações combinadas.

2.3.1.1 Modelagem Estrutural

A modelagem estrutural de aSideML tem como principais elementos os aspectos, os elementos base que são afetados pelos aspectos, bem como seus relacionamentos. O aspecto é constituído por um conjunto de características que melhoram a estrutura e o comportamento de classes por meio de *crosscutting* de forma sistêmica. A linguagem aSideML provê notação gráfica para representação e uso de aspectos. Os aspectos podem ter estrutura de dados e comportamento local. Ademais, eles definem a estrutura e comportamento organizados em interfaces transversais, as quais serão combinadas com o comportamento de classes por meio de *crosscutting.* Os aspectos são representados por meio de diagramas de aspectos e podem ser utilizados em outros diagramas, como por exemplo, o diagrama de classe entendido [Chavez, 2004].

As interfaces transversais providas por aSideML são conjuntos de características transversais com nome associado, que caracterizam o comportamento transversal de aspectos. As interfaces transversais são declaradas dentro de aspectos, nos diagramas de aspectos. Uma característica transversal descreve uma propriedade nomeada (atributo ou operação) definida em um aspecto que pode afetar um ou mais elementos base em locais específicos por meio de crosscutting.

Em aSideML, podemos ter a definição de dois tipos de características transversais: (i) a característica transversal estrutural (*Additions*) que é uma especificação de um atributo que será estaticamente introduzido na interface de uma ou mais classes base e, (ii) a característica transversal comportamental é uma especificação de um comportamento que será adicionada a uma ou mais classes base, ou para refinar (*Refinement*) ou redefinir (*Redefinition*) uma operação de uma ou mais classes base. A característica transversal comportamental possui em seu compartimento nomes da operação (adornos) representados com o símbolo _, e podem ter diversas combinações indicando se o comportamento *crosscutting* deve ocorrer antes *(_opName*), depois (*opName_*) ou durante (*_opName_*) a operação base. Adicionalmente,

existe a característica requisitada comportamental (*Uses*) que é uma operação base que será usada dentro do espaço dos nomes dos aspectos [Chavez, 2004].

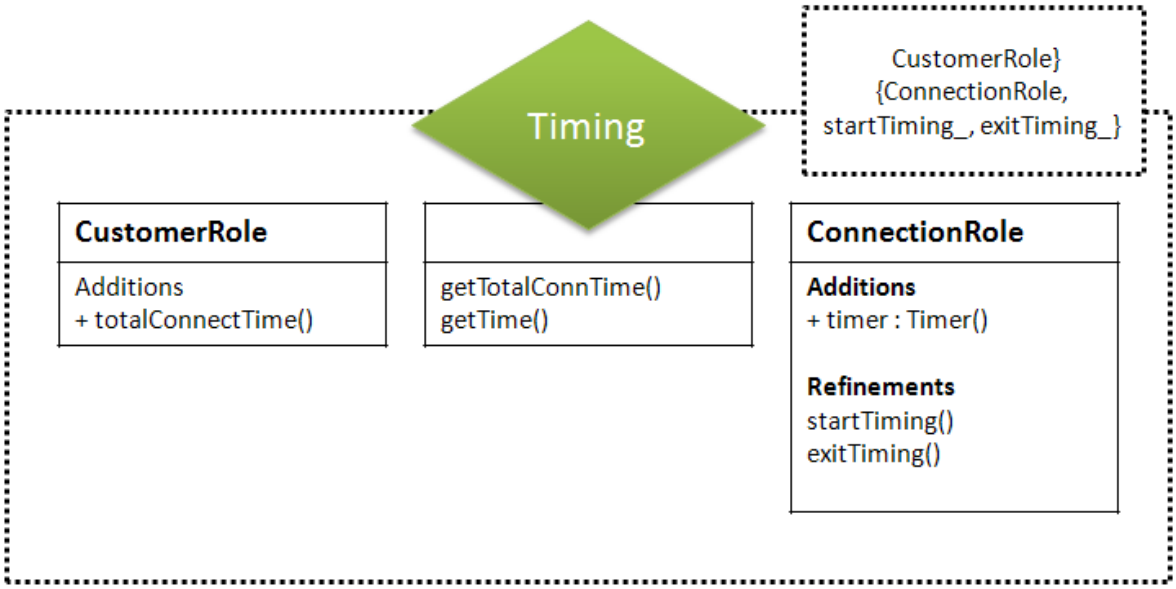

Figura 5 - Representação da declaração do aspecto Timing [Chavez, 2004]

A Figura 2 ilustra a representação do aspecto Timing com suas respectivas interfaces transversais. O Aspecto Timing é representado por um losango, contendo duas interfaces transversais, *CustomerRole* e *ConnectionRole*. No canto direito superior do aspecto é ilustrado um retângulo tracejado que corresponde a caixa de parâmetros de template (*Template Parameter*). Os parâmetros de template representam uma lista de parâmetros formais, com uma lista para cada interface transversal do aspecto. Cada uma das interfaces transversais declaradas no aspecto possui suas respectivas características transversais. O aspecto ainda apresenta comportamento local, representado pela declaração das operações *getTotalConnTime()* e *getTimer()*.

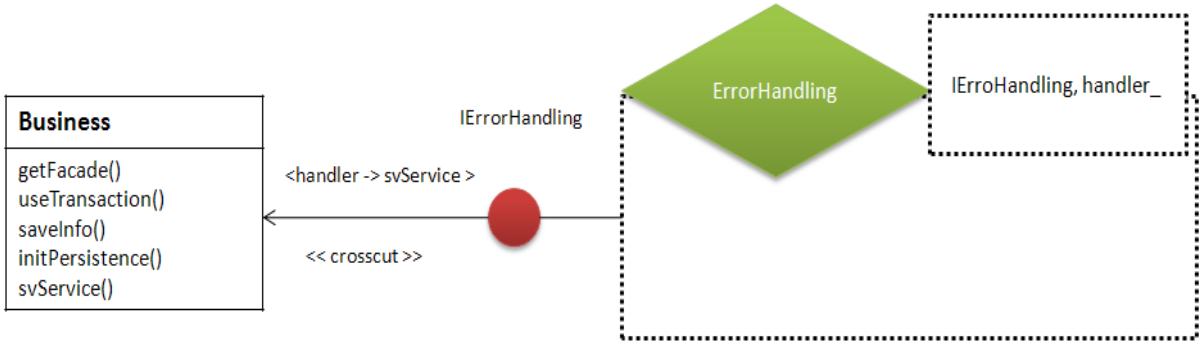

Figura 6 – Diagrama de classe estendido em aSideML [Chavez, 2004]

2.3.1.2 Diagramas

Além de fornecer novos diagramas, aSideML enriquece alguns diagramas da UML, com o intuito de apresentar elementos de *crosscutting*, bem como seus relacionamentos com os elementos base. Dentre os diagramas fornecidos por aSideML [Chavez, 2004] temos, diagramas de aspecto, diagrama de colaborações aspectuais, diagramas de sequência aspectuais, diagramas de processo de combinação e diagramas de classes entendidos. Cada um deles será melhor descrito abaixo:

a) Diagramas de aspecto: O diagrama de aspecto fornece uma descrição completa de um aspecto. A descrição incorpora as interfaces transversais, características locais e relacionamentos de herança. Cada característica transversal comportamental pode ser visualizada em um digrama de colaboração aspectual.

b) Diagramas de colaboração aspectual: Uma colaboração aspectual oferece uma apresentação gráfica de uma colaboração aspectual. Um colaboração aspectual é um tipo especial de colaboração que models a realização de uma operação transversal definida dentro de um aspecto. Esse diagrama oferece suporte à visão de interação que envolve instâncias de aspectos e elementos base.

c) Diagramas de sequência aspectuais: O diagrama de sequência aspectual oferece uma apresentação gráfica de um conjunto de mensagens organizadas em seqüências temporais, de tal forma que essas mensagens denotam invocações a operações de aspectos, sob a perspectiva de aspectos. Tal diagrama oferece suporte à visão de interação que envolve instâncias de aspectos e elementos base.

d) Diagramas de processo de combinação: Um diagrama de processo de combinação oferece uma apresentação gráfica para um grupo de elementos combinados. Os elementos combinados são elementos base adornados de forma a enfatizar as melhorias proporcionadas pelos elementos *crosscutting* [Chavez, 2004]. Os elementos combinados podem ser especializados para cada modelo de implementação disponível, tais como AspectLua, CaesarJ, AspectJ, dentre outros.

e) Diagramas de classes estendidos: Um diagrama de classes estendido oferece uma apresentação gráfica da visão de projeto estático de um sistema em que as classes e os aspectos residem como cidadãos de primeira classe. Cada aspecto pode ser visualizado em detalhe e separadamente em um diagrama de aspecto correspondente [Chavez, 2004].

Embora aSideML possua semântica e notação para a representação de características transversais, bem como sua interação com as características não-transversais de um sistema, em nosso trabalho utilizaremos uma descrição textual de aSideML. Essa decisão se dá em virtude de atualmente não haver ferramentas que permitam a modelagem visual com a notação disponibilizada pela linguagem. Além disso, existe uma sintaxe textual para a representação dos elementos de aSideML.

2.4 Metaspin

O Metaspin [Brichau et al, 2006] é um metamodelo que foi concebido como resultado do estudo [Brichau, Haupt, 2005] de diversas linguagens de POA [Kiczales et al, 97], onde foram analisadas as semelhanças fundamentais, bem como as variabilidades entre as linguagens. O metamodelo visa representar os conceitos essenciais das linguagens de programação orientadas a aspectos (POA), tais como *join point, pointcut*, *advice*, e *weaving*.

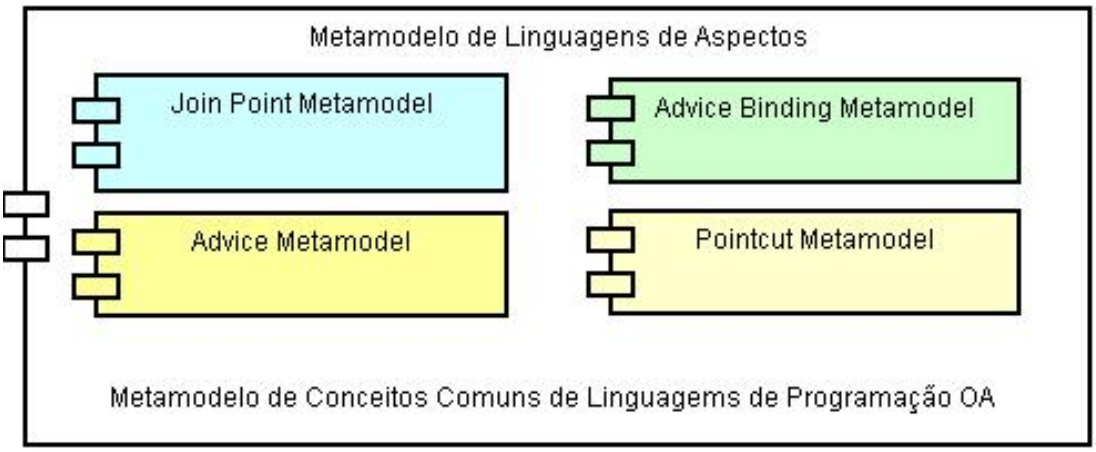

Figura 7 - Representação da Estrutura do Metaspin

A principal característica do Metaspin é a possibilidade de representar linguagem de POA (e.g. AspectJ, CaesarJ, dentre outras). A representação de uma linguagem de POA no Metaspin pode ocorrer através da tradução de características de linguagens de aspectos

específicas para os conceitos do metamodelo ou através da especialização dos conceitos comuns do metamodelo. O Metaspin é dividido em quatro sub-metamodelos, de acordo com o ilustrado na Figura 7: metamodelo de *join point*, metamodelo de *advice*, metamodelo de linguagem de *pointcut* e metamodelo de ligação do *advice*, os quais serão detalhados nas seções 2.4.1, 2.4.2, 2.4.3 e 2.4.4 respectivamente.

2.4.1 Metamodelo de *Join Point*

Representa o conceito de *join points,* permitindo a representação de *join points* estáticos e dinâmicos. O *join point* dinâmico é aquele disponível a partir da execução de um programa, enquanto que o *join point* estático é aquele disponível a partir do texto do código fonte.

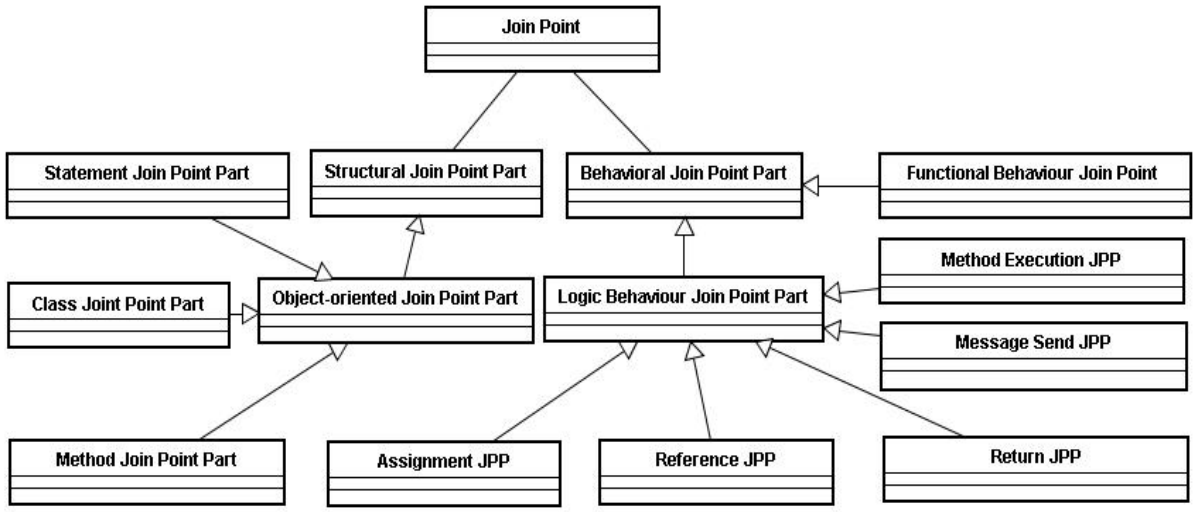

Figura 8 - Representação do Metamodelo de *Joinpoint*

O metamodelo de *join point* mostrado na Figura 8 define que um *join point* é formado por uma parte estrutural (*Structural Join Point Part*) e uma parte comportamental (*Behavioral Join Point Part*). A parte estrutural refere-se a um local no código fonte de um programa onde o aspecto irá atuar, que podem ser: (i) classes (*Class Join Point Part*); (ii) métodos (*Method Join Point Part);* e (iii) declarações (*Statement Join Point Part*). Por sua vez, a parte comportamental refere-se ao estado de execução de um programa. No entanto, não detalhatemos os elementos das parte comportamental visto que ela não será utilizada em nossa abordagem.. Em alguns casos existe a necessidade de se especializar a noção genérica de *join points*, no sentido de refletir em diversos tipos de *join point* existentes em diferentes linguagens de POA.

2.4.2 Metamodelo de *Advice*

As ações que podem ser disparadas através de aspectos em *join points* particulares são descritos usando esse metamodelo. O *advice* expressa a funcionalidade que precisa ser invocada por um aspecto. O Metaspin modela *advices* concretos usando ações de *advice* (*Advice Action*) que são compostos em uma estrutura de árvore, representando o *advice* (ver Figura 9).

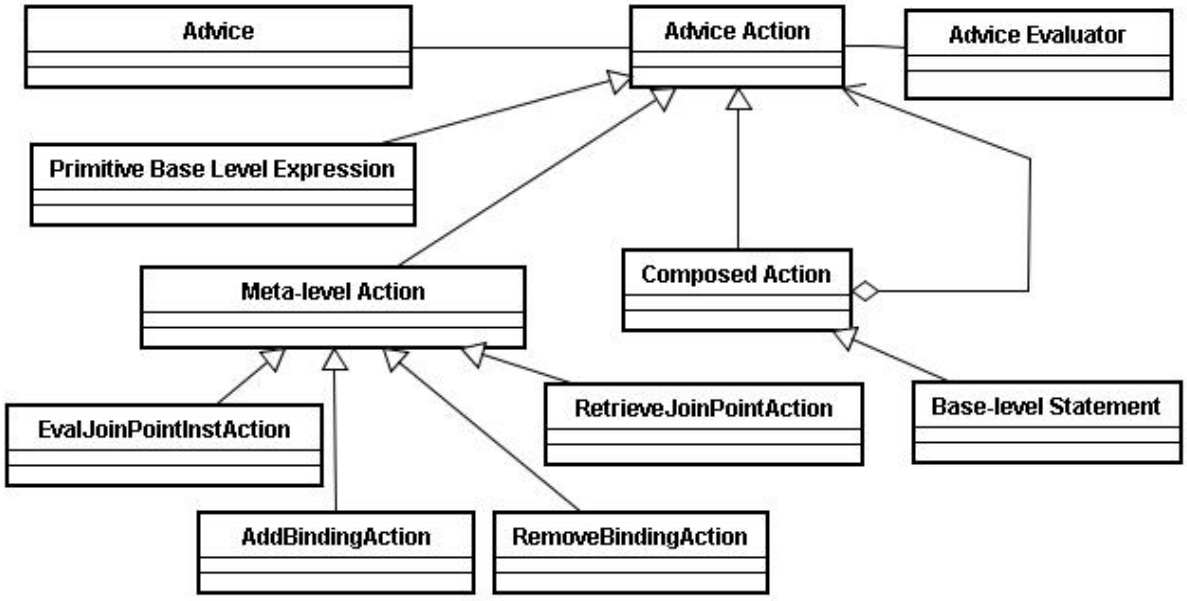

Figura 9 - Representação do Metamodelo de Advice

Tal estrutura pode ser composta por: expressões primitivas (*Primitive Base Level Expression*); (ii) expressões compostas no nível base (*Composed* Action) e expressões em meta-nível (*Meta-level Action*). Enquanto que as ações de nível base estão relacionadas com expressões e declarações normais do programa, as ações de meta-nível são ações específicas que podem estar contidas somente no *advice*. As ações de meta-nível definem como instruções específicas que ocorrem no *advice* precisam ser executadas.

2.4.3 Metamodelo de Linguagem de *Pointcut*

Esse metamodelo representa a abstração de *pointcut*. Os *pointcuts* são representados como predicados sobre os *join points*. Isso significa que expressões de pointcut avaliam (*evaluate*) os *join points*. O conceito de *pointcut* é representado no metamodelo como um *Join Point Selector* (ver Figura 10). Um seletor de *join point* (*Join Point Selector*) pode ser um

seletor do tipo primitivo (*Primitive Selector*) ao qual aplica um predicado simples em um *join point* ou pode ser um seletor do tipo composto (*Composed Selector*) que aplica múltiplos predicados a um *join point*.

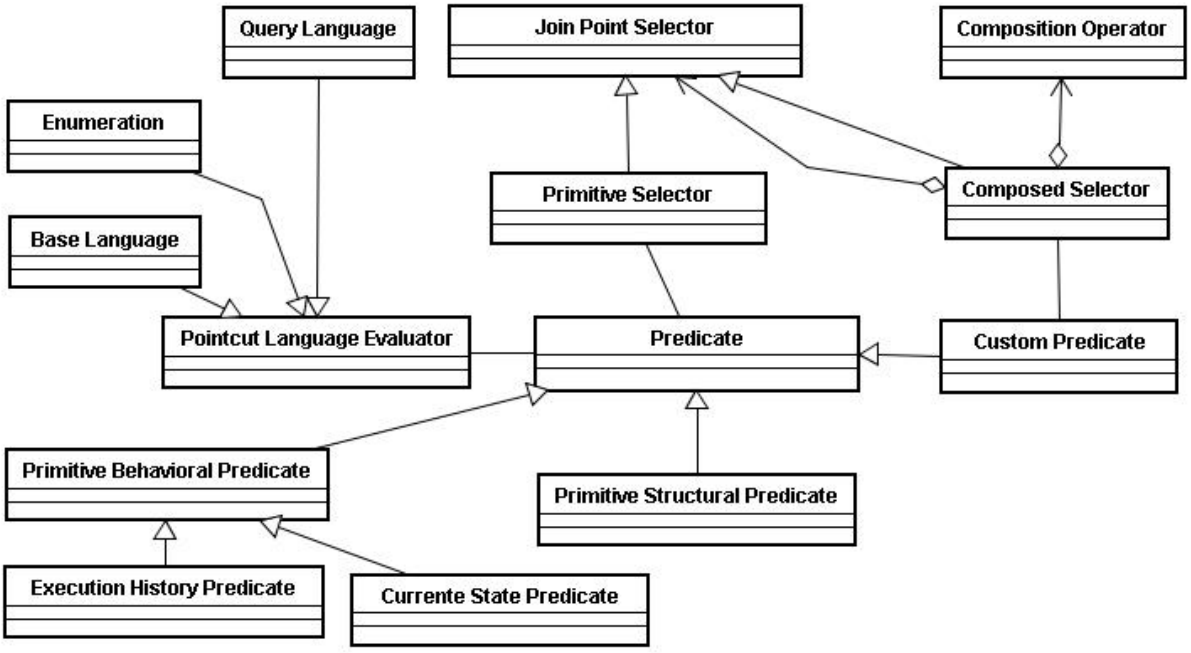

Figura 10 - Representação do metamodelo de Linguagem de Pointcut

Cada um dos múltiplos predicados é chamado em um seletor separado. Posteriormente, esses seletores são unidos usando operadores de composição (*Composition Operators*). Uma expressão de *pointcut* pode então ser formada a partir da aplicação de um predicado simples ou da aplicação de múltiplos predicados. O Metaspin classifica os predicados em estruturais e comportamentais. O predicado estrutural (*Primitive Structural Predicate*) está relacionado às propriedades estruturais do *join point*, enquanto que o predicado comportamental (*Primitive* Behavioral Predicate) está relacionado às propriedades comportamentais do *join point*. O avaliador de linguagem de pointcut (*Pointcut Language Evaluator*) é uma propriedade do *Join Point Selector.* O Metaspin já provê especializações do avaliador de linguagem de *pointcut*, como enumeração (*Enumeration*), linguagem de consulta (*Query Language*) e protocolos de conexão presentes na linguagem base (*Base Language*).

2.4.4 Metamodelo de Ligação de *Advice*

O metamodelo de ligação (b*inding*) de *advice*, ilustrado na Figura 11 define como os aspectos são instanciados, modularizados e como os *advices* são vinculados aos *pointcuts.* Os

aspectos são formados de *pointcuts*, *advices* e definições de variáveis. Um aspecto contém seletor de ligações de advice (*Selector Advice Binding*). Os aspectos contem definições de variáveis que são responsáveis por definir o estado da instancia de um aspecto. Um *StateSelector* é associado com cara variável de um aspecto, tal que cada *StateSelector* define o escopo de cada variável, ou seja, ele define como um estado particular é selecionado ou criado por cada variável, de forma a executar um advice. Isto torna possível para representar que, para a execução de um conselho aspecto, um estado especial, que podem ser selecionados para cada uma das variáveis. Na maioria dos casos, todas as variáveis do aspecto terão o mesmo escopo mais o metamodelo permite a especificação de uma granularidade mais final.

Finalmente, um seletor de ligação (*Binding Selector*) representa a composição de advices quando múltiplos aspectos e/ou advices são aplicados no mesmo join point. O seletor define a ordem de aplicação dos advices, selecionando um como sendo o primeiro.

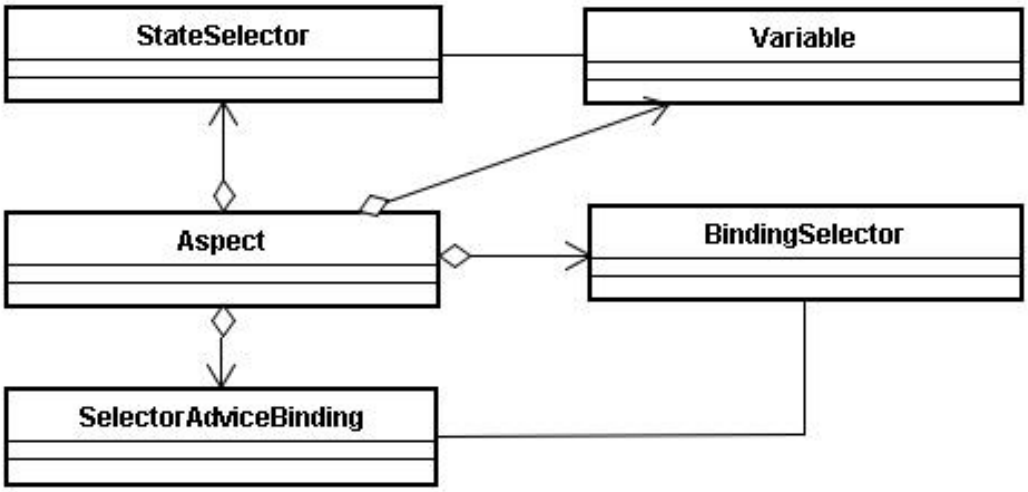

Figura 11 - Representação do Metamodelo de Ligação de Advice

2.5 MDD (Model Driven Development)

A essência do Desenvolvimento Dirigido a Modelos (MDD - *Model Driven Development*) [Stahl et al., 2006] está na especificação de modelos e na transformação desses modelos em outros modelos ou em artefatos de software, de forma a permitir a comunicação de diferentes fases do processo de desenvolvimento de software, onde os artefatos produzidos em cada fase podem ser representados por modelos. Na Abordagem MDD os modelos não são apenas veículos para descrever sistemas de software ou facilitar a comunicação entre as partes do sistema, mas também podem ser associados a diferentes fases do processo de desenvolvimento de software, como ilustrado na Figura 12.

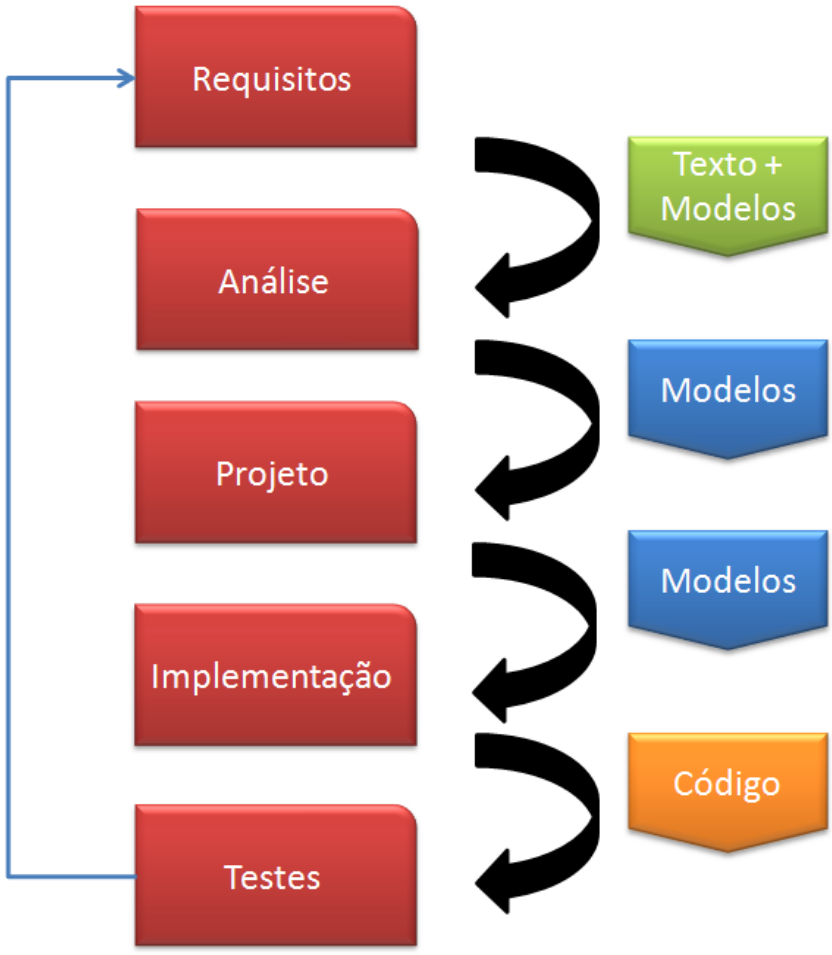

Figura 12 - Processo MDD

A Arquitetura Dirigida a Modelos (MDA) [OMG/MDA, 2003], a qual está inserida no contexto MDD, propõe a separação da especificação do funcionamento de um sistema, dos detalhes de como tal sistema utiliza a capacidade de uma plataforma específica. A MDA tem como principais objetivos a interoperabilidade, portabilidade e reusabilidade por meio da separação de conceitos arquiteturais. Adicionalmente, a MDA provê 4 níveis de abstração da arquitetura de um sistema (ver Figura 13), são eles: (i) Modelo Independente de Computação (CIM); (ii) Modelo Independente de Plataforma (PIM); e Modelo Específico de Plataforma (PSM) e CODE. Os três primeiros níveis trabalham com artefatos em nível de modelos, enquanto que o último trabalha com artefatos textuais.

Geralmente, a modelagem de um sistema a partir de um ponto inicial, utilizando-se uma abordagem dirigida a modelos ocorre em quatro fases. Inicialmente é criado um modelo independente de computação CIM (*Computational Independent Model*). Normalmente, o CIM é responsávek por descrever o sistema em nível de requisitos em um processo de

desenvolvimento de software dirigido a modelos. O CIM não mostra detalhes da estrutura do sistema. Os principais usuários do CIM, os analistas de domínio, não conhecem os modelos ou artefatos utilizados para descrever a funcionalidade para o qual os requisitos são articulados no CIM. Ademais, o CIM desempenha um papel importante diminuindo o *gap* entre os especialistas em domínio e seus requisitos, e os especialistas que desenvolvem e constroem artefatos, que em conjunto satisfazem os requisitos de domínio [OMG/MDA, 2003]. O CIM é transformado em um modelo independente de plataforma (PIM - *Plataform Independent Model*) responsável por descrever (i.e. diagramas de classe, diagramas de componentes e diagramas de estado) o sistema, abstraindo a tecnologia a ser utilizada. Em outras palavras, o PIM exibe um grau específico de independência, possibilitando seu uso em um variado número de plataformas.

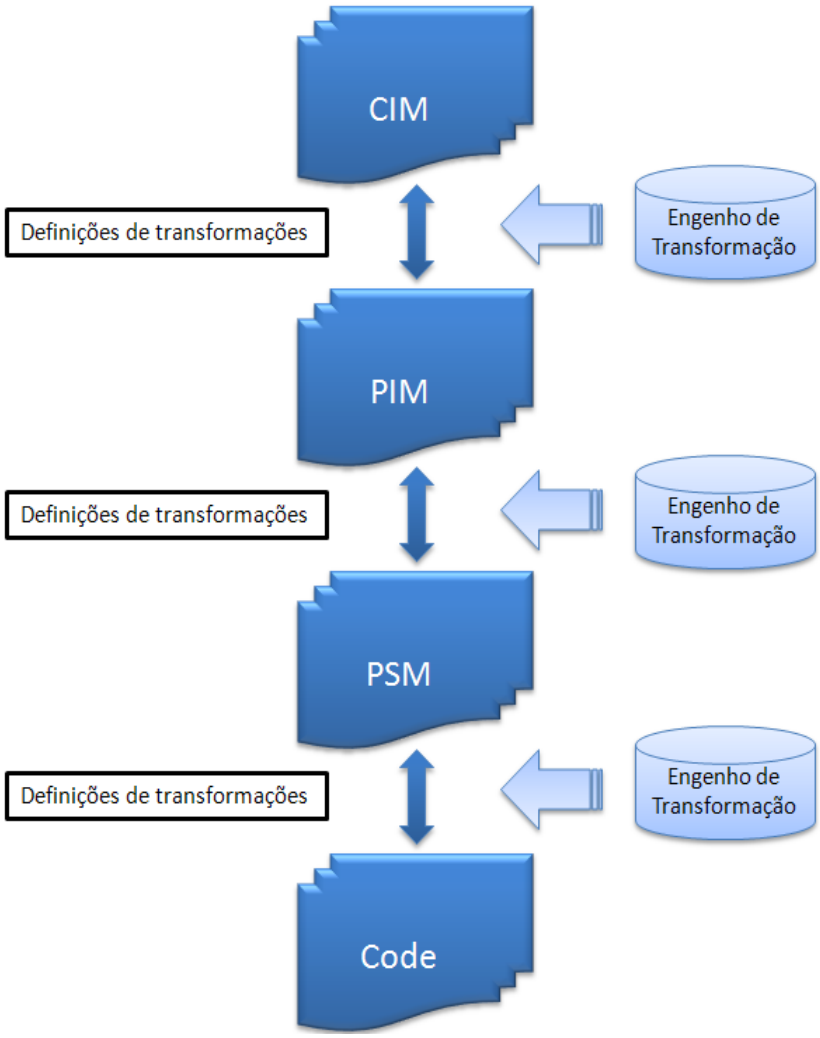

Figura 13 - Níveis de Abstração da Arquitetura Dirigida a Modelos

Finalmente, o PIM é transformado em um modelo especifico de plataforma (PSM - *Plataform Specific Model*). O PSM combina especificações do nível PIM com os detalhes que

especificam como o sistema utiliza uma plataforma específica. Por fim, o PSM é transformado em código fonte em uma linguagem de programação.

Os passos da transformação entre modelos podem ser realizados por ferramentas automatizadas, as quais reduzem os erros de programação e os custos de desenvolvimento. O mapeamento dos modelos de uma abstração para outra é feito através da definição de transformações e aplicação dos mecanismos de transformação para executar tais transformações. Para definir transformações entre modelos são usadas linguagens de transformação.

2.5.1 ATLAS Transformation Language (ATL)

ATL [Jouault et al, 2005] é uma linguagem de transformação entre modelos especificada a partir de um metamodelo e de uma sintaxe textual concreta. Um programa especificado em ATL é composto de regras (ATL *rules*) que definem como os elementos de um modelo de entrada são alcançados e navegados de forma a se criar e inicializar elementos de um modelo de saída. Além do modelo básico de transformação. A linguagem ATL também provê mecanismos chamados *helpers,* que permitem navegar no modelo de forma a obter informações sobre elementos do modelo de entrada..

Além disso, ATL permite a modularização de código através de bibliotecas (ATL *Libraries*). As transformações ATL são definidas em forma de módulos (module). Um ATL *module* contém uma seção de cabeçalho (*Header Section*) obrigatória, uma seção de import e um número de *helpers* e regras de transformação (*transformation rules*). A seção do cabeçalho dá o nome do módulo da transformação e declara os modelos de entrada e de saída. A seção de cabeçalho inicia com a palavra-chave *module* seguida pelo nome do módulo ou transformação. Os modelos de entrada e saída são declarados como variáveis tipadas por seus metamodelos. A palavra-chave *create* indica o modelo alvo. A palavra-chave *from* indica o modelo de entrada.

No exemplo da Figura 14 o modelo de saída vinculado à variável OUT é criado a partir de um modelo de entrada IN. O modelo de entrada e o modelo de saída estão de acordo com seus metamodelos.

```
1. module aSideML2Metaspin;
2. create  OUT : Metaspin from IN : aSideML ;
3.
4. rule AspectaSideML2AspectMetaspin{
5.          from
6.          aspect : aSideML!Aspect
7.          to
8.                    asp : Metaspin!"AdviceBinding::Aspect"(
9.                              name <- aspect.name,
10.                             variables <- vars
11.                   ),
12.
13. vars: distinct Metaspin!"AdviceBinding::Variable"
14.       foreach (var in asp.attributes) (
15.                   name <-attributes.name,
16.                   ),
17.       }
18.
19. helper context aSideML!Aspect def : allAttributes :
20.      Sequence(aSideML!Attribute) =
21.        self.attrs  ;
```

Figura 14 – Representação de declarações ATL

Geralmente, mais de uma modelo de entrada e de saída podem ser enumerados na seção do cabeçalho. O *helpers* e *transformation rules* são construções usadas para especificar a transformação. A Figura 14 contém um exemplo que será usado na definição da transformação do estudo de caso. Neste exemplo, é declarada a transformação *aSideML2Metaspin* (linha 1), que irá especificar a transformação um aspecto do modelo de entrada aSideML descrito em aSideML produzindo um aspecto no modelo de saída Metaspin.

A regra *AspectaSideML2AspectMetaspin* (linhas 4-17) define uma transformação simples do elemento *Aspect* pertencente ao modelo de entrada aSideML para o elemento *Aspect* pertencente ao modelo Metaspin de saída. A palavra-chave *from* define o elemento do modelo de entrada. Ao definir-se o elemento de uma regra de transformação deve-se criar uma variável e instanciá-la informando o metamodelo e o elemento que se deseja transformar. A palavra chave *to* é utilizada para indicar qual elemento do modelo de saída deverá ser criado. Assim como o elemento de entrada deve-se especificar uma variável e instanciá-la informando o metamodelo e o elemento que será criado. Depois de especificados os elementos de entrada e saída e seus respectivos metamodelos de origem, define-se a lógica da transformação (linhas 8-17).

Adicionalmente, na Figura 14 é definido o *helper* *allAttributes* A palavra-chave *context* define a partir de qual elemento o *helper* pode ser invocado. Neste caso o contexto é o elemento do tipo *Aspect* a partir do qual se deseja obter um conjunto de elementos do tipo

Attribute. O nome do *helper* é introduzido pela palavra-chave *def*. Por fim, após a declaração do nome é especificado o tipo de retorno. No exemplo, o *helper* deverá retornar um conjunto de elementos do tipo *Attribute*, pertencente ao modelo de entrada aSideML que é denotado pela expressão *Sequence*(aSideML!Attribute).

2.5.2 *Kernel MetaMetaModel* (KM3)

Nesse trabalho utilizamos o KM3 [Jouault et al, 2003] na especificação de metamodelos. O KM3 é uma linguagem definida pelo grupo INRIA, que provê mecanismos para a descrição de metamodelos e definição de linguagens específicas de domínio DSL [Deursen et al, 2000], tendo sua terminologia baseada em Ecore [Budinsk, 2004]. O KM3 surgiu como resposta a freqüentes questionamentos de usuários que trabalhavam com transformações em modelos utilizando a linguagem ATL. O *Object Management Group* (OMG) tem especificado como padrão para a definição de metamodelos o *Meta-Object Facility* (MOF) [OMG/MOF]. Embora a OMG defina o MOF como padrão para a especificação de DSL's, em nosso trabalho foi utilizada a linguagem KM3. Essa decisão foi tomada visto que o KM3 já vem sendo utilizado em outras abordagens que constituem um projeto mais amplo no qual nosso trabalho também está inserido. A sintaxe de KM3 é simples, bem definida e apresenta algumas similaridades com a notação do Java [SUN, 2009]. Os arquivos definidos com a extensão .km3 podem ser transformados em um metamodelo e serializados no formato XMI [OMG/XMI, 2003]. Além disso, o KM3 é suportado pela IDE Eclipse

```
package PrimitiveTypes {
        datatype Boolean;
        datatype Integer;
        datatype String;
}
package KM3 {

abstract class LocatedElement {
        attribute location : String;
}
abstract class ModelElement extends LocatedElement {
        attribute name : String;
        reference "package" : Package oppositeOf contents;
}
class Classifier extends ModelElement {
}
class DataType extends Classifier {
}
class Enumeration extends Classifier {
        reference literals[*] ordered container : EnumLiteral oppositeOf enum;
}
```

```
class EnumLiteral extends ModelElement {
        reference enum : Enumeration oppositeOf literals;
}

class TemplateParameter extends Classifier {
}
class Class extends Classifier {
        reference parameters [*] ordered container: TemplateParameter;
        attribute   isAbstract : Boolean;
        reference supertypes[*] : Class;
        reference structuralFeatures[*] ordered container : StructuralFeature oppositeOf owner;
        reference operations[*] ordered container : Operation oppositeOf owner;
}
class TypedElement extends ModelElement {
        attribute lower : Integer;
        attribute upper : Integer;
        attribute isOrdered : Boolean;
        attribute isUnique : Boolean;
        reference type : Classifier;
}
class StructuralFeature extends TypedElement {
        reference owner : Class oppositeOf structuralFeatures;
        reference subsetOf [*]: StructuralFeature oppositeOf derivedFrom;
        reference derivedFrom [*]: StructuralFeature oppositeOf subsetOf;
}
class Attribute extends StructuralFeature {
}
class Reference extends StructuralFeature {
        attribute   isContainer : Boolean;
        reference opposite[0-1] : Reference;
}
class Operation extends TypedElement {
        reference owner : Class oppositeOf operations;
        reference parameters[*] ordered container : Parameter oppositeOf owner;
}
class Parameter extends TypedElement {
        reference owner : Operation oppositeOf parameters;
}
class Package extends ModelElement {
        reference contents[*] ordered container : ModelElement oppositeOf "package";
        reference metamodel : Metamodel oppositeOf contents;
}
class Metamodel extends LocatedElement {
        reference contents[*] ordered container : Package oppositeOf metamodel;
 }
}
```

Figura 15 – Descrição do metamodelo KM3 descrito em KM3

A Figura 15 ilustra a definição do metamodelo de KM3 especificado na sintaxe textual da própria linguagem. Nessa definição, vemos que o KM3 é composto por elementos que estão inseridos dentro de um *package*, tais como *ModelElement*, *TypedElement*, *Classifier*, *EnumLiteral*, dentre outros. Além disso, o KM3 permite: (i) herança entre os elementos definidos no metamodelo, através do uso da palavra *extend*; e (ii) definir que determinados

elementos podem conter conjuntos ordenados ou não, de outros elementos através do uso. Por exemplo, o elemento *Class* (destacado em cinza), mostra que este herda do elemento *Classifier* (utilizando-se o comando *extends*) previamente definido no metamodelo. Ademais, define que o elemento *Class* contém um conjunto ordenado (uso das palavras-chave *ordered* e *container*) dos elementos *Operation*, *TemplateParameter* e *StructuralFeature*.

2.5.3 *Textual Concrete Syntax* (TCS)

No contexto do desenvolvimento dirigido a modelos, quando trabalhamos com a definição de linguagens específicas de domínio (DSL) [Deursen et al, 2000], temos a necessidade de representar suas abstrações seja visualmente ou textualmente. Mecanismos como o *Human-Usable Textual Notation* (HUTN) [HUTN/OMG, 2004] proposto pela OMG e o TCS proposto em [Jouault et al, 2006], permitem a definição de sintaxes textuais para a representação dos elementos de DSL, bem como a derivação de modelos a partir uma descrição textual de uma linguagem, e vice-versa. Embora a OMG defina como padrão a HUTN, resolvemos adotar o TCS, pois este já vem sendo utilizado em outros trabalhos que fazem parte de um contexto mais amplo no qual este está inserido.

O TCS [Jouault et al, 2006] é utilizado em nossa abordagem tanto para a definição da sintaxe textual concreta dos elementos do metamodelo aSideML, quanto para transformações de *text-to-model* (T2M). O TCS é um componente do Eclipse/GMT que permite a especificação de sintaxes textuais concretas para linguagens específicas de domínio (DSL – Domain Specific Language), bem como definição de transformações do tipo *text-2-model* e *model-to-text*. A Figura 16 ilustra um exemplo de como pode ser definida a sintaxe textual para o elemento A de um metamodelo. Neste exemplo temos a definição dos tipos primitivos que são utilizados na linguagem, bem como o elemento base do modelo que pode conter os elementos do tipo *Aspect, Relationship Crosscutting, Crosscutting Element, classes e interfaces.*

Em relação às transformações *text-to-model* e *model-to-text* o TCS provê dois mecanismos para fornecer suporte a esse dois tipos de transformação, o *injector* e o *extractor*, respectivamente. O *injector* recebe como entrada uma descrição textual de acordo com a sintaxe predefinida pelo TCS e a partir dela insere elementos em um modelo que deverá esta em conformidade com seu respectivo metamodelo. O *injector* consiste de um *parser* gerado por ferramentas providas pela tecnologia ANTLR [Parr, 2007] que é parte do mecanismo TCS. Por sua vez, o *extractor* gera a representação textual de uma linguagem recebendo como

entrada um modelo XMI bem formado em conformidade com o metamodelo da respectiva linguagem. O *extractor* trabalha com a representação interna de modelos expressa em uma linguagem e cria sua representação textual.

```
Syntax aSideML {
-- BEGIN Primitive templates
-- Specifies representation of primitive types.
-- Only needs modification when default lexer is not satisfactory.
-- Generally modified along with the lexer.
        primitiveTemplate identifier for String default using NAME:
                value = "%token%";

        primitiveTemplate stringSymbol for String using STRING:
                value = "ei.unescapeString(%token%, 1)",
                serializer="'\" + %value%.toCString() + '\"";

        primitiveTemplate integerSymbol for Integer default using INT:
                value = "Integer.valueOf(%token%)";

        primitiveTemplate floatSymbol for Double default using FLOAT:
                value = "Double.valueOf(%token%)";
-- END Primitive templates
-- BEGIN Class templates
-- Specifies representation of classes.
-- This is the main section to work on.
    template Model main
        :       "Model aSideML" modelName"{"
                relationshipCrosscutting
                aspect
          crosscuttingElement
                classes
                interfaces;         ;
```

Figura 16 - exemplo de definição de sintaxe textual com TCS

O uso de TCS é tipicamente mais simples que o desenvolvimento de injetores e extratores de forma *ad-hoc*. Além disso, a redundância entre modelos TCS e seus modelos correspondentes é reduzida. Embora o TCS permita a simplificação na construção de injetores e extratores, tem-se certo custo por isso. Ou a sintaxe textual é adaptada de acordo com as possibilidades do TCS, ou o metamodelo é simplificado. Ademais, uma restrição importante imposta pelo TCS nos metamodelos é que eles devem possuir um elemento raiz (adicionando a primitiva *main context* na declaração de um elemento).

2.5.4 Acceleo

O Acceleo [Acceleo, 2009] é uma ferramenta de geração de código desenvolvida pela empresa francesa Obeo, que realiza a transformação de modelos em código, seguindo uma estratégia MDA. O Acceleo foi desenvolvido como forma de melhorar a produtividade do

desenvolvimento de software lidando com diversos conceitos agrupados. A ferramenta é baseada nos últimos avanços da pesquisa e melhores práticas industriais, e oferece vantagens como: alta personalização, interoperabilidade de modelos, e gerenciamento de rastreabilidade entre modelos.

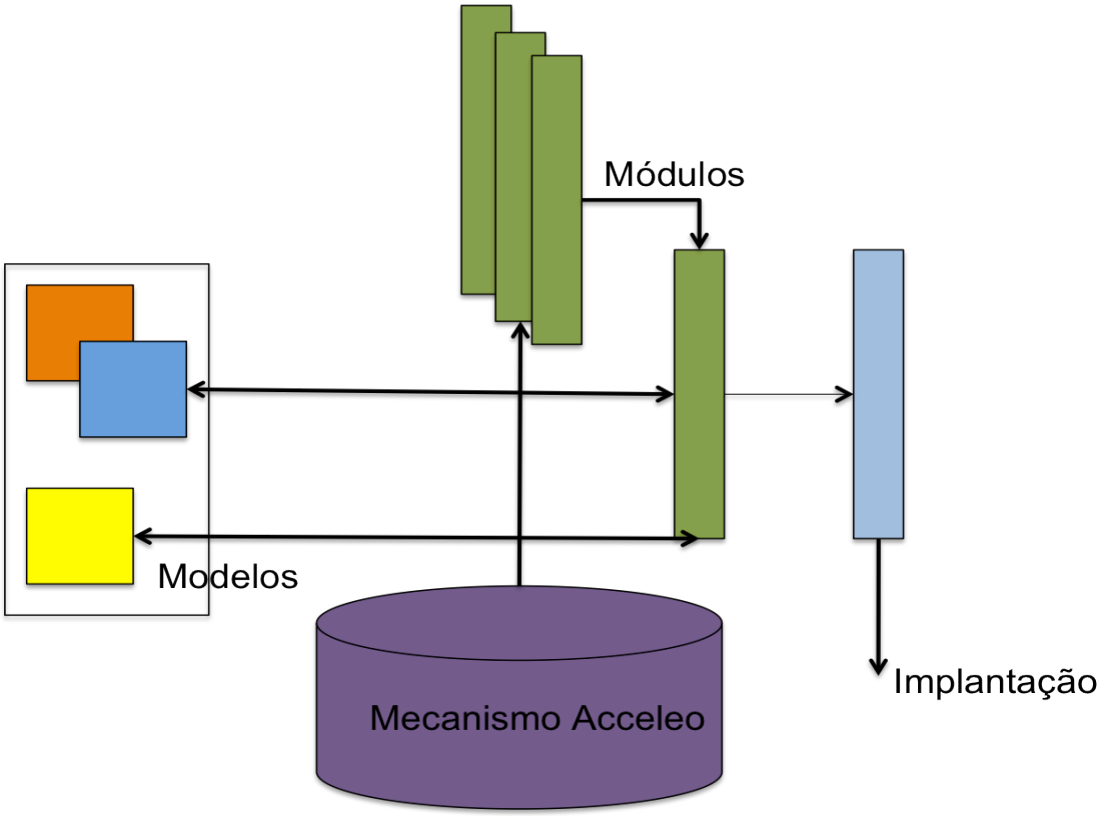

Figura 17 - Representação da arquitetura do Acceleo [Acceleo, 2009]

O Acceleo apresenta diversas funcionalidades tais como: integração total com eclipse, uso nativo de *Eclipse Modeling Framework* (EMF) [Budinsk, 2004], ambiente de desenvolvimento e edição, editor com destaque colorido de acordo com a sintaxe, detecção de erros, gerenciamento de templates e habilidade de trabalhar com qualquer tecnologia. Ademais, a ferramenta é compatível com modelos UML2, UML1. Adicionalmente, o Acceleo trabalha com o conceito de módulos (module) (ver Figura 17), onde os módulos são *plugins* Acceleo para a geração de código. Cada módulo representa uma tecnologia específica. Os Acceleo *modules* provêem soluções com alta confiabilidade e rápida implementação. Um *module* é composto por diversos templates que descrevem uma informação requisitada para gerar um código fonte de um metamodelo.

As transformações de modelos para texto (M2T) são construídas a partir de templates. Os templates são arquivos com script's contendo códigos de acesso ao modelo de origem que

geram a estrutura dinâmica, especificados entre os terminadores <% e %> ou [<% e %>], e estrutura dos arquivos a serem gerados.

Na Figura 18 observa-se a maioria dos elementos de definição de um código Acceleo, através da definição de um template para a geração de código. A linha 1 e 6 mostra como acessar a parte dinâmica da programação, por meio dos delimitadores <% e %>. Na linha 2 é definido o metamodelo a ser utilizado, onde a palavra chave metamodel é seguida por uma URI indicando qual o metamodelo. Nas linhas 4 e 5, é realizado a importação de código escritos na linguagem Java SE, e assim permitir que acesse métodos de dentro do template. Na linha 8 e 9, é especificado um script para processar elementos do modelo de um determinado tipo que é especificado pelo atributo type, no exemplo processar elementos do tipo *Class*; Ainda é fornecido um nome para o script no atributo name; por fim, para cada elemento, o script será executado e irá ter como resultado um arquivo texto com o nome determinado no atributo file.

```
1.<%
2.Metamodel http://www.eclipse.org/uml2/2.0.0/UML
3.
4. import org.acceleo.modules.uml2.services.Uml2Services
5. import org.acceleo.modules.uml2.services.StringServices
6. %>
7.
8. <%script type=”Class” name=”generate”
9.   file=”/<%jdbcPackage.toPath() %/Jdbc<%name%> Dao.java%>”
10. package <%jdbcPackage%>;
11.
12. import java.sql.Connection;
13. import java.sql.Date;
14. import java.sql.PreparedStatement;
15. import java.sql.ResultSet;
16. import java.sql.SQLException;
17. import java.sql.Statement;
18. import java.util.ArrayList;
19. import java.util.List;
20.
21. import org.acceleo.fwk.dao.DaoException;
22. import org.acceleo.fwk.dao.jdbc.JdbcConnectionUtils;
23. import org.acceleo.sample.dto.<%name%>Dto;
24.
25. public class Jdbc<%name%Dao> {
26.
```

Figura 18 – Representação do funcionamento do TCS.

3 Transformações baseadas em modelos orientados a aspectos: do Projeto Detalhado ao Código Fonte.

Neste capítulo são ilustradas as regras de mapeamento que foram especificadas, visando estabelecer a comunicação entre o nível de projeto detalhado, para um nível abstrato de linguagens de programação OA e, deste último, para uma ou mais linguagens de programação OA específicas de plataforma. As seção 3.1 e 3.2 ilustram as regras de mapeamento entre os elementos do nível projeto detalhado descrito em aSideML para um nível abstrato de programação conferido pelo Metaspin e CoreElement. Nas seções 3.3 e 3.4 são apresentadas as regras de mapeamento entre o nível abstrato de programação para as linguagens de programação OA específicas de plataforma, AspectLua e CaesarJ, respectivamente.

Essas regras de transfromação são especificadas sob a perspectiva do desenvolvimento dirigido a modelos, onde o mapeamento entre as linguagens ocorre através da rastreabilidade das informações dos elementos pertencentes aos metamodelos de cada linguagem. Adicionalmente, as transformações dirigidas a modelos orientados a aspectos permitem a propagação de informações pelo ciclo de vida de desenvolvimento de software. O Metaspin é utilizado especificamente para representar o nível abstrato de implementação de linguagens de aspectos. Para a representação de conceitos não OA, tais como classes, interfaces, atributos e métodos, utilizamos o pacote *core* da UML (que aqui denominamos *CoreElement*) visto que ele já fornece essas abstrações.

O processo de mapeamento pode ser dividido em 3 fases: (i) mapeamento dos conceitos relacionados a descrição aspectual de aSideML para o Metaspin, onde os conceitos em nível de projeto detalhado providos por aSideML são mapeados para um nível de implementação abstrato conferido pelo Metaspin; e (ii) mapeamento dos conceitos relacionados a descrição dos elementos base de aSideML para o metamodelo *CoreElement;* (iii) mapeamento dos elementos do Metaspin para os elementos do metamodelo da linguagem específica OA (AspectLua, CaesarJ, dentre outras); e (iv) mapeamento dos elementos do *CoreElement* para o código base em uma linguagem específica. A primeira atividade é realizada em um nível PIM (*Plataform Independent Model*) onde a partir de um modelo de projeto detalhado independente de plataforma expresso em aSideML é gerado um modelo de implementação abstrato expresso pelo Metaspin. As fase 2 parte de um modelo de

implementação abstrato no nível PIM da arquitetura MDA e gera modelos para linguagens especificas de plataforma correspondendo ao nível PSM (*Plataform Specific Model*) da arquitetura MDA. Por fim, a fase 3 trabalha em um nível PSM, onde será realizada a geração de código para diferentes linguagens de programação (neste caso AspectLua e CaesarJ) a partir de modelos OA específicos de plataforma. A Figura 19 resume onde as atividades do processo de mapeamento encaixam-se na arquitetura dirigida a modelos.

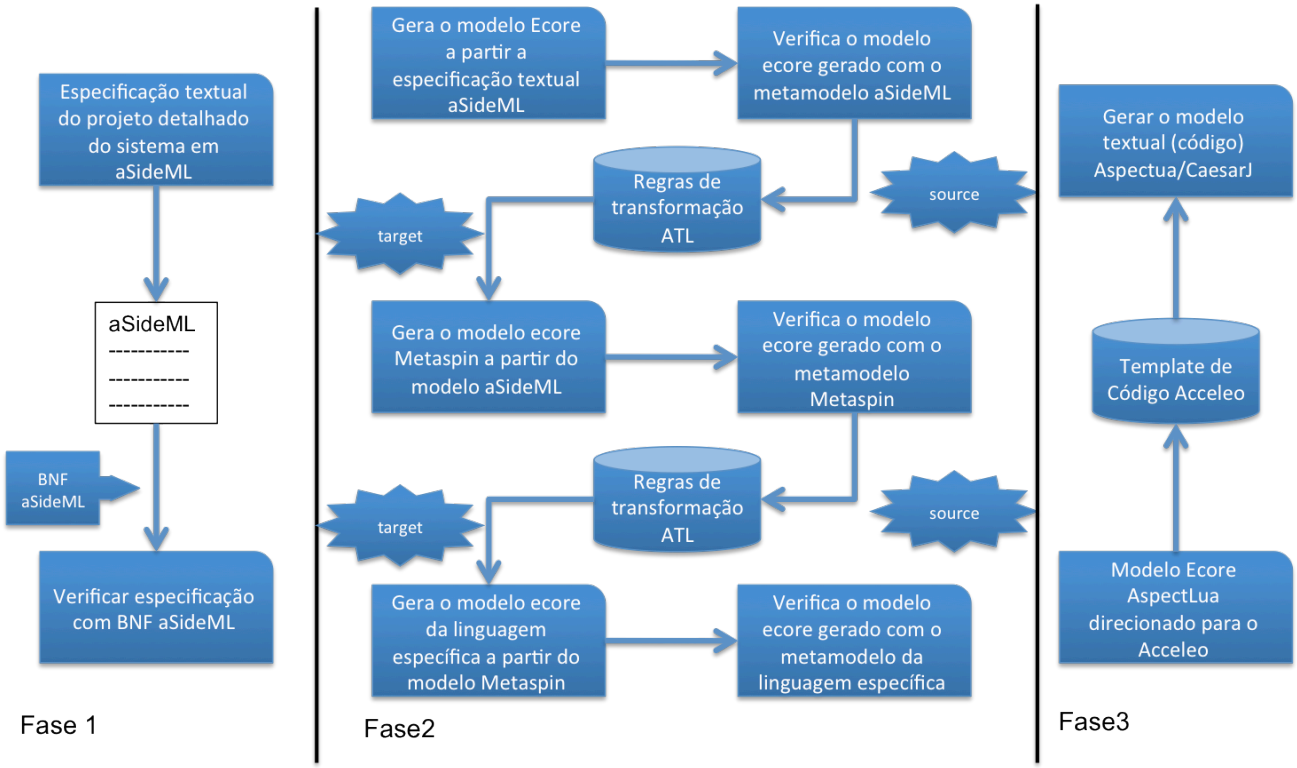

Figura 19 – Mapeamento de atividades de transformação entre modelos no Marisa-AOCode

O Metaspin provê suporte somente para abstrações ligadas `a orientação a aspectos, ou seja, não contempla abstrações para a representação dos elementos base de um sistema. Dessa forma, com a necessidade de endereçar o mapeamento dos elementos base de uma aplicação representado pelo projeto detalhado em aSideML foi utilizado o metamodelo *CoreElement*. As etapas desse processo de mapeamento serão detalhadas nas seções 3.1, 3.2, 3.3 e 3.4, respectivamente.

3.1 Mapeamento entre aSideML e Metaspin.

Nesta seção é ilustrado o mapeamento entre os conceitos de orientação a aspectos providos por aSideML, no nível de projeto detalhado, para os conceitos providos pelo Metaspin, no nível de implementação abstrato de aspectos.

i. **Mapeamento de Aspectos de aSideML para Aspectos no Metaspin:** um aspecto em aSideML é representado como um conjunto de características que enriquecem a estrutura e o comportamento de classes por meio de *crosscutting*. Dessa forma, os aspectos em aSideML são definidos usando-se argumentos fornecidos para **parâmetros de template** (*Template Parameter*) através de relacionamento *crosscutting*. Por sua vez, o Metaspin define um aspecto como um conjunto de seletores *join point selectors*, *advices* e *variables* e *operations*. Os aspectos de aSideML tem seu nome, atributos e conjunto de operações mapeados para o aspecto no Metaspin. A Figura 20 ilustra o mapeamento dos elementos de um aspecto aSideML para o Metaspin.

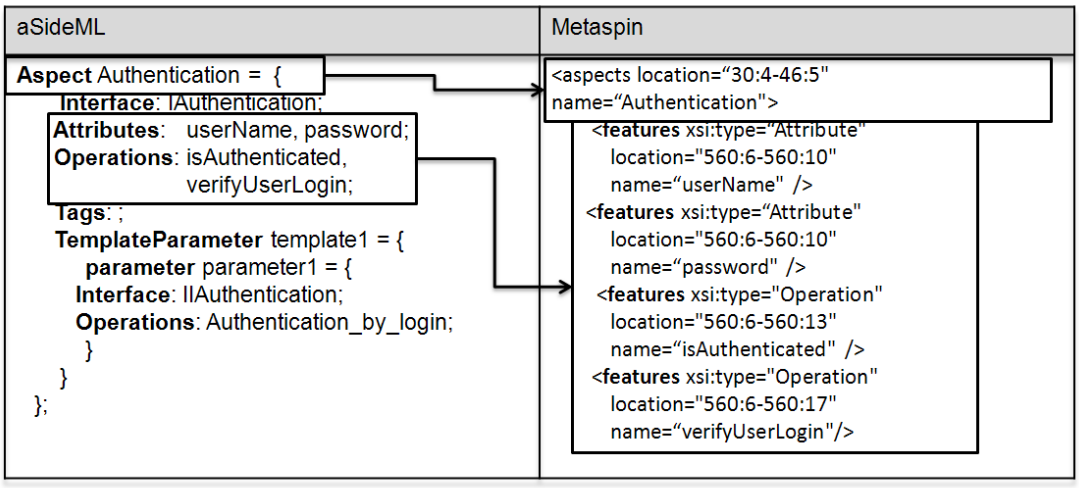

Figura 20 - Mapeamento de Aspecto em aSideML para Aspecto no Metaspin

ii. **Mapeamento de Parâmetros de Template de aSideML para Join Point Selector no Metaspin:** As operações de parâmetros de template definidos por aSideML são *placeholders* para operações. Esses parâmetros de template de operação podem ser definidos e referenciados dentro de especificações de interação, denotando que são *pointcuts* para o comportamento *crosscutting*. O Metaspin representa o conceito de *pointcut* através de *Join Point Selectors*. Dessa forma, os parâmetros de template de aSideML são mapeados (ver Figura 21) para os *join point selectors do Metaspin.* O *Join Point Selector* possui uma expressão de *pointcut.* Essa expressão pode ser formada pela aplicação de um predicado simples ou ser composta através da aplicação de múltiplos predicados. A expressão de *pointcut* é representada no Metaspin como um predicado (*Predicate*) que possui a referencia aos *join point* que irão compor o *pointcut* do aspecto.

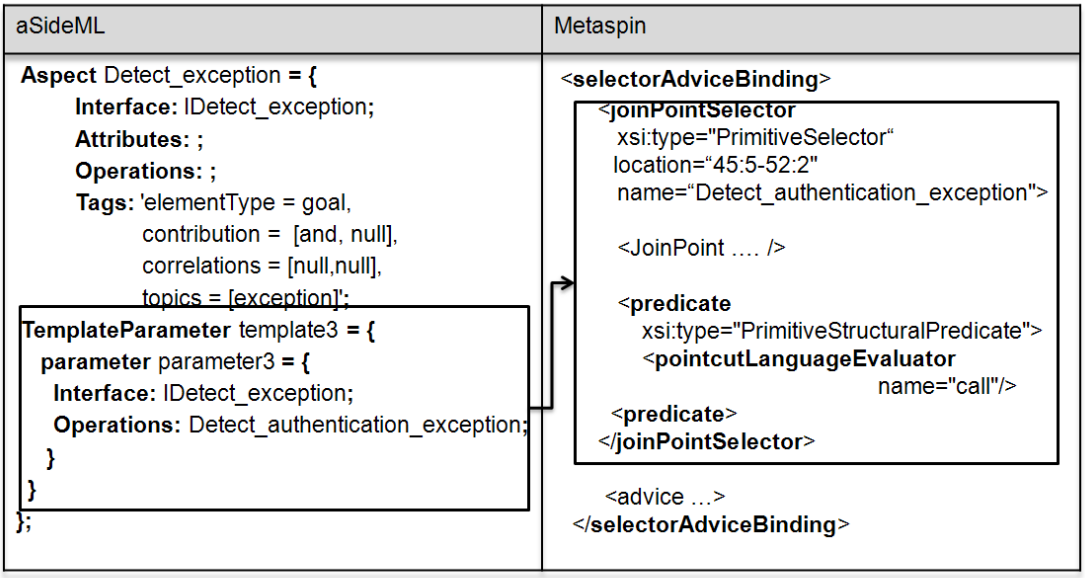

Figura 21 - Mapeamento de *Template Parameter* para *Join Point Selector*

iii. **Mapeamento de *Template Match* de aSideML para *Join Points* no Metaspin:** Os **casamentos de template** (Template Match) definidos em aSideML são um conjunto de operações a serem substituídas pelos advices dos aspectos. Essas operações estão relacionadas com os join points que determinado aspecto trata. Os elementos definidos no join point do Template match (classes, métodos ou declarações) são mapeados diretamente para o conceito de join points no Metaspin. O Metaspin provê representação de join points estruturais e comportamentais. No entanto, em virtude de o nosso trabalho foca em modelagem estrutural utilizou-se somente a representação de join points estruturais. Logo, as classes, métodos ou declarações definidas no *join point* do Template Match de aSideML são mapeados (ver Figura 22) no Metaspin para Class *Join Point Part*, *Method Join Point Part* e *Statement Join Point Part,* respectivamente.

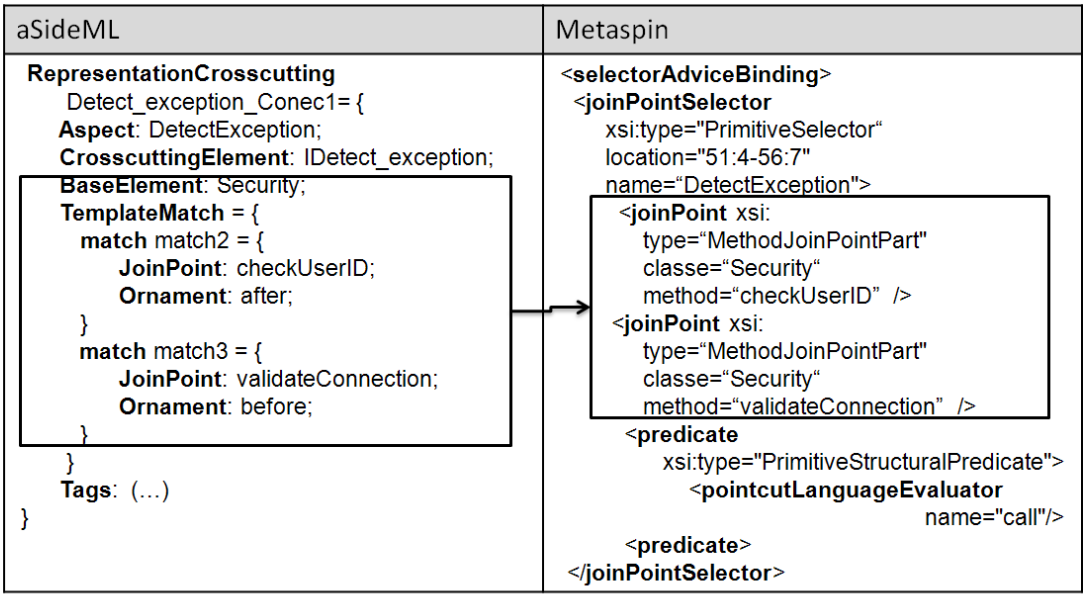

Figura 22 - Mapeamento de TemplateMatch para *Join Points*

iv. **Mapeamento de características transversais estruturais para *Intertype Declarations*:** em aSideML, uma característica transversal estrutural é a especificação de um atributo que será introduzido na interface de uma ou mais classes base [Chavez, 2004]. A linguagem aSideML define que o elemento Additions deverá conter assinaturas de operações ou novos atributos, de forma que essas características possam ser incorporadas pelo aspecto em classes e interfaces que representam o código base. O Metaspin não provê elementos para o tratamento deste tipo de informação. Como forma de evitar perda de informações dos modelos estruturais, foi introduzido o elemento *Intertype Declaration* no metamodelo do Metaspin. Assim, os elementos *Additions* agora são mapeados para *Intertype Declaration* no Metaspin (Ver Figura 23).Além disso, os elementos que são incorporados nas interfaces ou classes são anotados com comentário do tipo "From aSideML – Crosscutting Structural Feature", informando qual o elemento de Additions de aSideML foi mapeado para o *Intertype Declaration* a ser introduzido em determinada classe ou Interface.

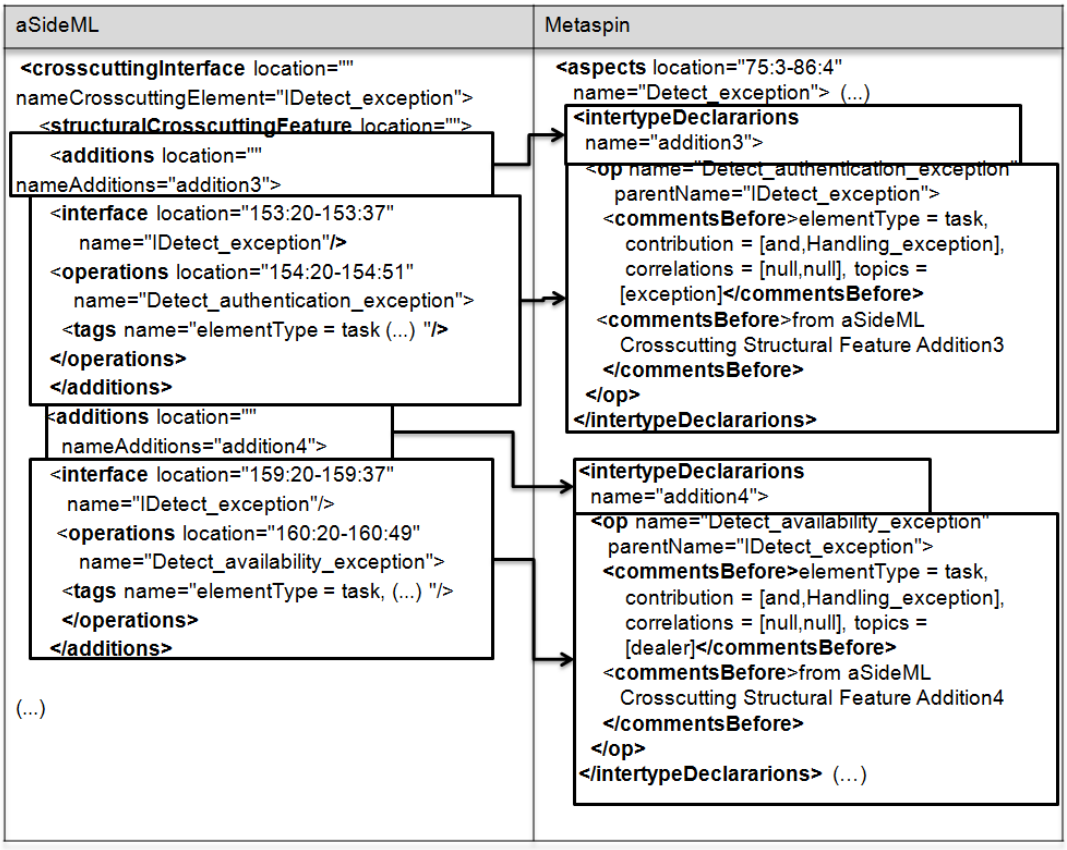

Figura 23 - Mapeamento de *Additions* em aSideML para *Intertype Declarations* no Metaspin

v. **Mapeamento de características transversais comportamentais para *Advice Action* e *Advice Type*:** em aSideML, uma característica transversal comportamental é uma especificação de parte do comportamento que será adicionado a uma ou mais classes do código base da aplicação. As características transversais comportamentais podem ser listadas em diferentes compartimentos de uma interface transversal de aSideML, tais como redefinições e refinamentos. As redefinições (*Redefinitions*) e refinamentos (*Refinements*) possuem elementos do tipo Adorno e *Operation*. Tais elementos irão compor o *advice* do metamodelo Metaspin. O elemento *Operation* é mapeado para a ação do *advice* (*Advice Action)*, que neste caso é especializado para uma ação de meta-nível (*Metalevel Action*), correspondendo a um elemento do tipo *EvalJoinPointInstruction*. Adicionalmente, cada operação correspondente a uma característica transversal comportamental deve vir comentada indicando se tal característica sofreu um refinamento ou redefinição. Por outro lado, o elemento Adorno é mapeado para um avaliador de advice (*Advice Evaluator*), visto que cada ação de *advice* deve ter um avaliador relacionado. O avaliador de *advice* geralmente possui valores do tipo *before*, *after* e *around*.

aSideML	Metaspin

CrosscuttingInterface IAuthentication= {
Redefinitions = {
Redefinition redefinition1 = {
Interface: IAuthentication;
Adorno: before;
Operation: Authentication_by_Login;
Operation Tags: elementType = task,
contribution = [and, Authentication],
correlations = [null, null], topics = [null];
}
Refinement = {
Refinement refinement1 = {
Interface: IAuthentication;
Adorno: before;
Operation:
Authentication_by_Fingerprint;
Operation Tags: elementType = task,
contribution = [and, Authentication],
correlations = [null, null], topics = [null];
}
}
	</**selectorAdviceBinding**>
	(other elements…)
	advice>
	<**adviceAction**
	xsi:type="EvalJoinPointInstructionAction"
	name="Authentication_by_Login">
	comments = 'From aSideML – Redefinition"
	<**adviceEvaluator** name="before"/>
	</**adviceAction**>
	</**advice**>
	advice>
	<**adviceAction**
	xsi:type="EvalJoinPointInstructionAction"
	name="Authentication_by_Fingerprint">
	comments = 'From aSideML – Refinement"
	<**adviceEvaluator** name="before"/>
	</**adviceAction**>
	</**advice**>
	</**selectorAdviceBinding**>

Figura 24 - Mapeamento de *Redefinition* e *Refinement* em aSideML para *advice* no Metaspin

A Figura 24 ilustra o mapeamento entre as características transversais do tipo *redefinition* e *refinement* pertencentes à uma interface transversal para o *advice* de um aspecto expresso em um modelo Metaspin. A Tabela 2 resume o mapeamento entre os elementos de aSideML, que representam os conceitos de orientação a aspectos em nível de projeto detalhado, para os elementos do Metaspin.

Tabela 2 - Mapeamento entre aSideML e Metaspin

aSideML			**Metaspin**
Aspect			*Aspect*
Template Parameter			*Join Point Selector*
Template Match	*Joinpoint*		*Joinpoint* (conjunto de operações afetadas pelo aspecto).
Crosscutting Interface	*Additions*		*Intertype Declaration* (operações ou atributos que serão inseridos em interfaces ou classes do código base)
	Redefinition / Refinement	*Operation*	*Advice Action*: corresponde à ação a ser executada em um advice.
		Adorno	Advice Evaluator: corresponde a quando a ação do advice deverá ser executada (*before, after, around*)

3.2 Mapeamento de aSideML e *CoreElement*

Para o mapeamento dos elementos de aSideML que denotam o código base de um sistema de software utilizamos o pacote *core* do metamodelo da UML, que por questão de praticidade aqui o denominamos por CoreElement. A tabela 3 ilustra o resumo do mapeamento dos elementos de aSideML para o *CoreElement*.

(i) **Mapeamento de características comportamentais requisitadas em aSideML para *Operation* do *CoreElement*:** em aSideML uma característica comportamental requisitada (*Uses*) consiste numa operação do código base que o aspecto utiliza. Essas características são denominadas *Uses* em aSideML e em virtude de representarem operações do código base da aplicação que o aspecto utiliza, elas são mapeadas (ver Figura 25) para o conceito de *Operation* no *CoreElement*. As operações que denotam características comportamentais requisitadas de aSideML, devem vir anotadas pelo comentário "*from aSideML – Behavioral Required Feature*", indicando que tal operação do código base é utilizada pelo aspecto.

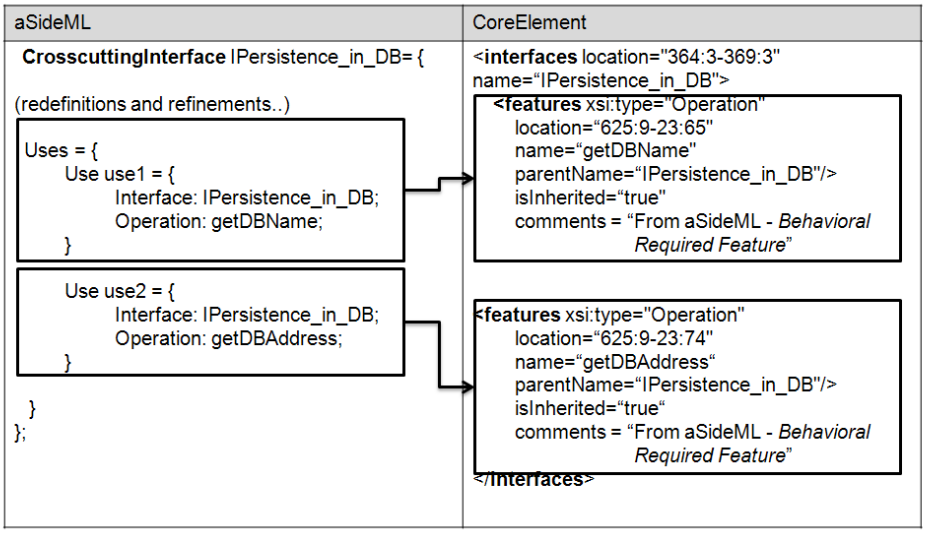

Figura 25 - Mapeamento de Uses em aSideML para *Operation* no *CoreElement*

(ii) **Mapeamento de Classes em aSideML para *Class* do CoreElement:** a abstração de classes em aSideML é mapeada diretamente para *Class* no CoreElement, sendo seus atributos e operações mapeados (ver Figura 26) para os elementos do tipo *Attribute* e *Operation,* respectivamente.

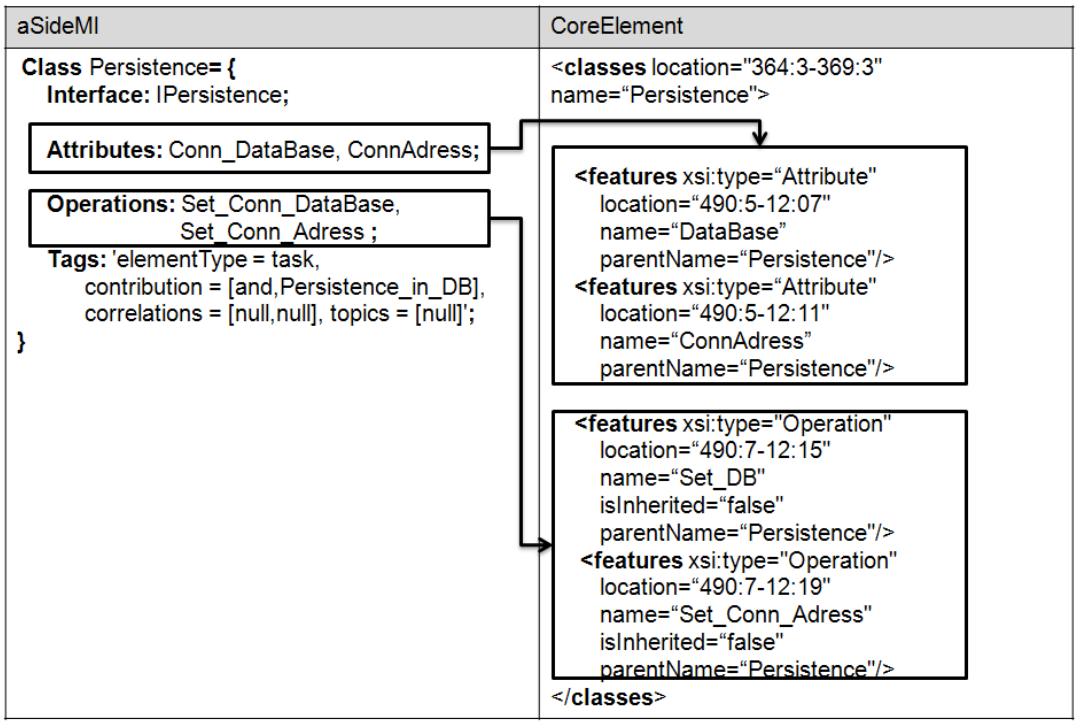

Figura 26 - Mapeamento de Classe aSideML para *Class CoreElement*

(iii) **Mapeamento de Interface em aSideML para Interface do CoreElement:** As interfaces de aSideML são mapeadas diretamente para Interface no CoreElement. As assinaturas das operações declaradas em uma interface aSideML são mapeadas para *Operation* no *CoreElement* e inserida em suas respectivas interfaces.

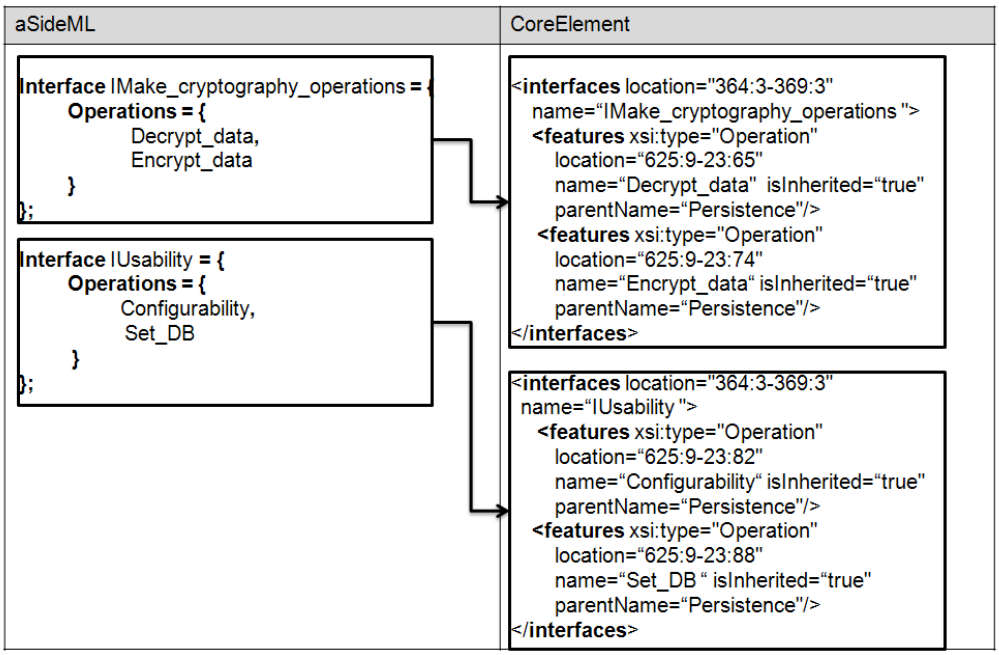

Figura 27 - Mapeamento de Interface aSideML para Interface no Metaspin

O Mapeamento entre Interfaces e as operações expressas em aSideML para o modelo *CoreElement* é ilustrado na Figura 27. A tabela 3 mostra o resumo do mapeamento entre os elementos aSideML para o modelo CoreElement.

Tabela 3 - Mapeamento entre aSideML e *CoreElement*

aSideML		**CoreElement**
Classe		*Class*
Attribute		Attribute
CrosscuttingInterface	*Uses*	*Operation*: precedida do comentário "from aSideML – Behavioral Required Feature
Interface		*Interface*

3.3 Mapeamento entre Metaspin e Linguagens de Programação orientadas a aspectos específicas de plataforma

Esta seção mostra como as abstrações de orientação a aspectos providos pelo Metaspin são mapeadas para uma linguagem de programação orientada a aspectos específica. Em nosso trabalho utilizamos AspectLua e CaesarJ como linguagens de programação OA específicas de plataforma de forma a ilustrar as regras de mapeamento. Os mapeamentos entre Metaspin e AspectLua e, entre Metaspin e CaesarJ serão detalhados nas subseções 3.3.1 e 3.3.2 respectivamente.

3.3.1 Mapeamento entre Metaspin e AspectLua

(i) **Mapeamento de Aspect do Metaspin para Aspect em AspectLua:** o elemento aspecto do Metaspin é mapeado diretamente para o conceito de aspecto em AspectLua. Além do nome, os atributos do aspecto do Metaspin são mapeados (ver Figura 28) para as variáveis de um aspecto AspectLua.

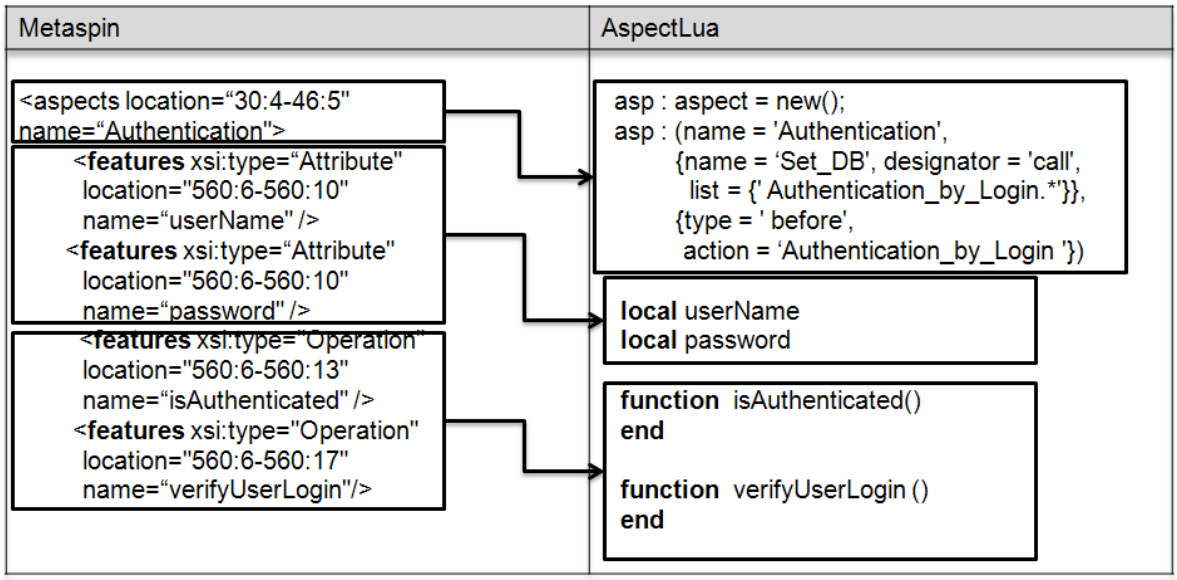

Figura 28 - Mapeamento entre Aspecto no Metaspin para aspecto em AspectLua

(ii) **Mapeamento de *Join Point Selector* do Metaspin para o *Pointcut* de AspectLua -** o *JoinPointSelector* representa um seletor de pontos de junção que irão compor o *pointcut* do aspecto. Dessa forma, ele é mapeado para o conceito de *pointcut* em AspectLua.

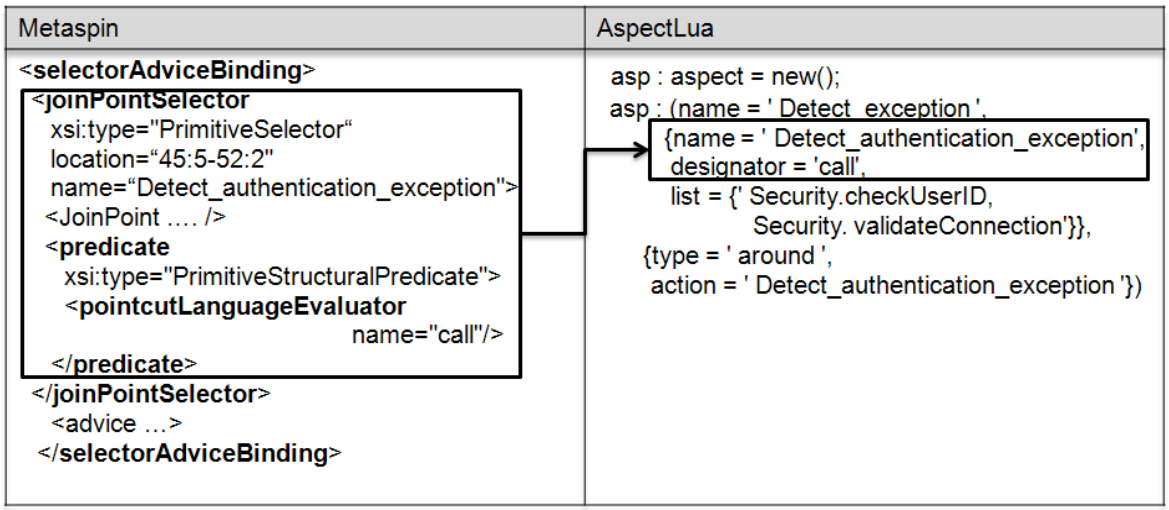

Figura 29 - Mapeamento de *Join Point Selector* para *pointcut*

O elemento *Predicate* pertencente a um *JoinPointSelector* corresponde a um predicado a ser aplicado sobre o conjunto de *join points* pertencentes a um *pointcut.* Para cada *Predicate* é definido um *Pointcut Language Evaluator* que define o tipo de chamada sob a qual os *join points* são acessados. Dessa forma, esse elemento é mapeado (ver Figura 29) para

o elemento *designator* do *pointcut* em AspectLua, podendo assumir como valor as primitivas *call*, *callone* e *execution*.

(iii) **Mapeamento de *JoinPoint* do Metaspin para conjunto de *join points* em AspectLua:** Os join points estruturais do Metaspin podem ser expressos em forma de classes, métodos e declarações. Aqui tratamos os *join points* estruturais do tipo *ClassJoinPointPart* e *MethodJoinPointPart,* os quais são mapeados para o conjunto de *join points* que compõe o *pointcut* de um aspecto, expressos pelo parâmetro *list* na declaração do aspecto. A Figura 30 ilustra o mapeamento de dois *join points* (*MethodJoinPointPart*) expressos no Metaspin, que são incorporados na lista de *join points* na declaração do aspecto em AspectLua.

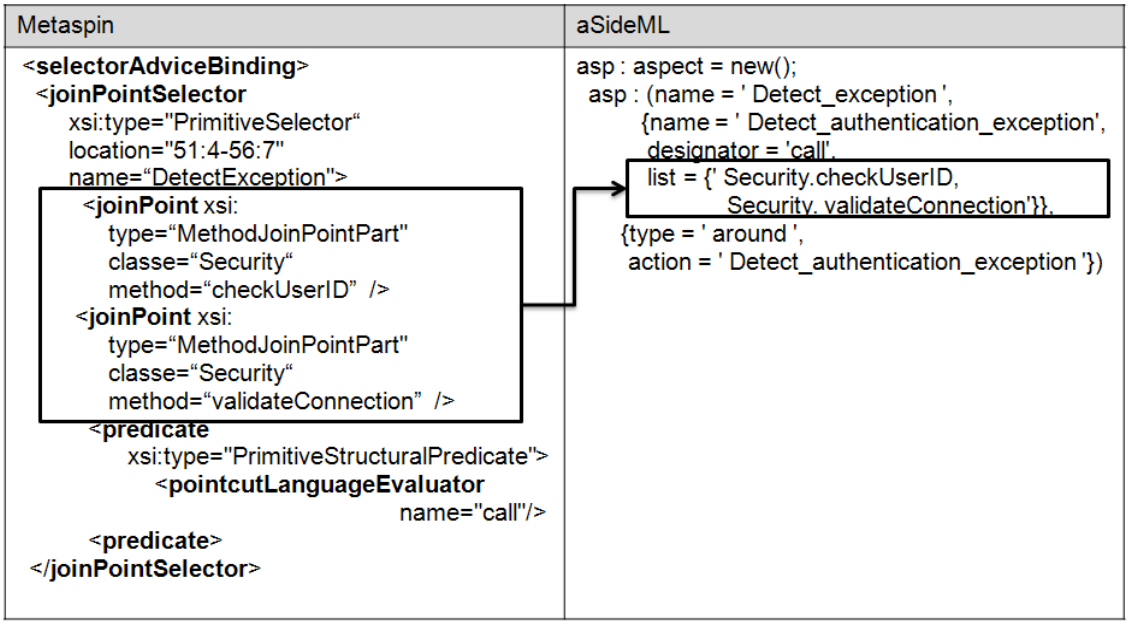

Figura 30 - Mapeamento de *join points* entre Metaspin e AspectLua

(iv) **Mapeamento de *Intertype Declaration* do Metaspin para *Introduction* em AspectLua:** O elemento *Intertype Declaration* do Metaspin é mapeado para o o conceito de *Introduction* em AspectLua (ver Figura 31). O conceito de *introduction* em AspectLua estende as características de entidades do tipo *function* e são representadas dessa forma, pois elas designam operações a serem adicionadas pelo aspecto aos elementos base com o objetivo de melhorar seu comportamento.Adicionalmente, as operações representadas como declarações inter-tipos além de serem inseridas na metatabela da(s) classe(s) que o aspecto afeta, também são anotadas com comentário do tipo "from aSideML – Crosscutting Structural Feature <nome_elemento>" de forma a permitir a rastreabilidade desses elementos.

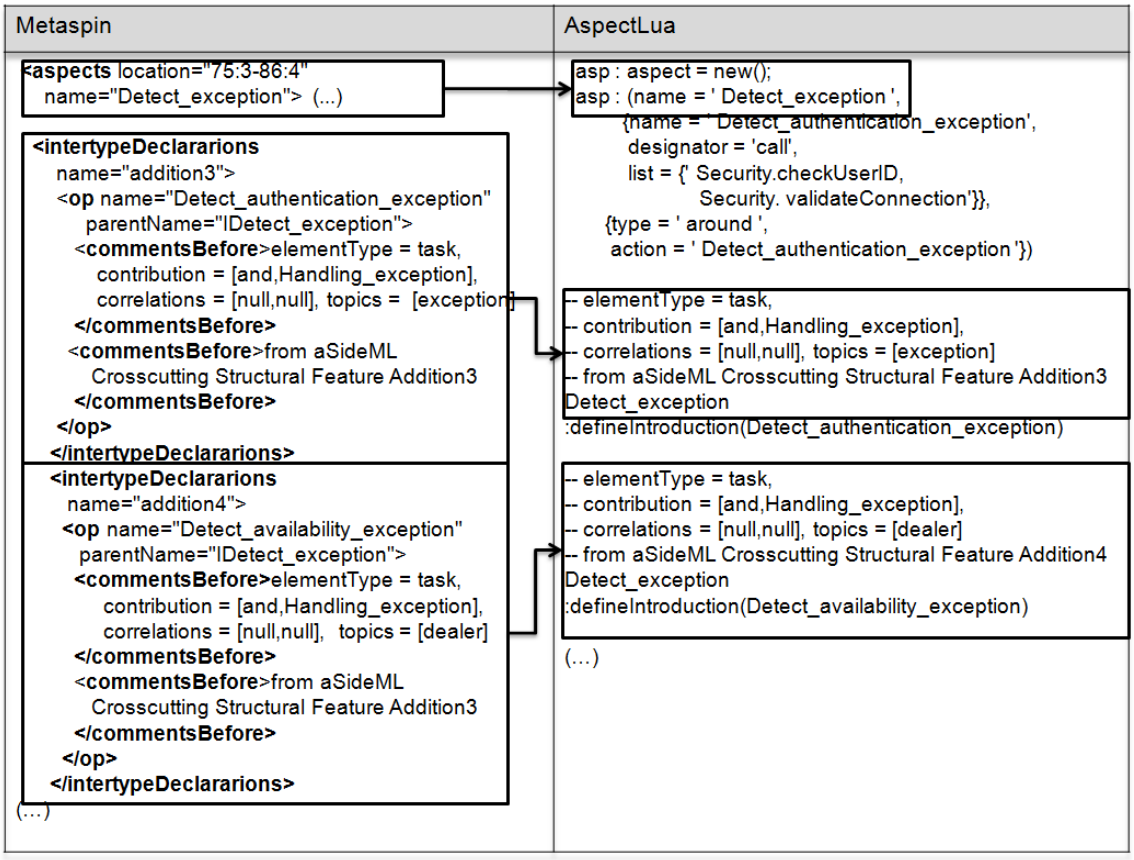

Figura 31 - Mapeamento entre *Intertype Declaration* no Metaspin para *Introduction* em AspectLua

(v) **Mapeamento de *Advice* do Metaspin para *Advice* em AspectLua:** o avaliador de *advice* (*AdviceEvaluator*) do Metaspin representa "quando" (*before*, *after* e *around*) a ação do *advice* de um determinado aspecto deve ser executada. Dessa forma, essa informação é mapeada para o campo *AdviceType*, na declaração do *advice* de um aspecto em AspectLua. Adicionalmente, o *advice* possui uma ação que será executa quando um dos *join points* pertencentes a um *pointcut* for atingido. Essa ação é representada no Metaspin pelo elemento *Advice Action*. Dessa forma, esse elemento é mapeado diretamente para o campo *action* na declaração do *advice* do aspecto em AspectLua. A Figura 32 ilustra o mapeamento do *advice* do nível abstrato de programação para o *advice* do aspecto em AspectLua.

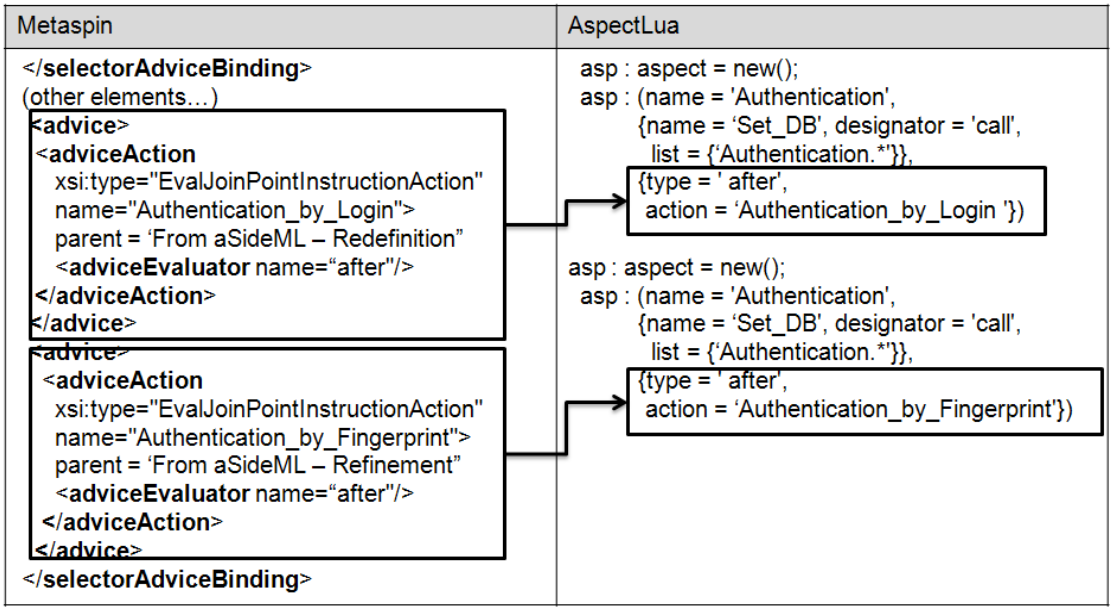

Figura 32 - Mapeamento de *advice* entre Metaspin e AspectLua

Na tabela 4 é mostrado um resumo do mapeamento entre os elementos aspectuais de Metaspin e AspectLua. O metamodelo AspectLua estende os conceitos do Metamodelo Metaspin, culminando em um mapeamento de um para um entre todos os elementos utilizados na construção de um aspecto em AspectLua.

Tabela 4 - Mapeamento entre Metaspin e AspectLua

Metaspin	**AspectLua**
Aspect	*Aspect*
Join Point Selector	*Pointcut*
PointcutLanguageEvaluator	*PoincutExpression*
Joinpoint	*Joinpoint*
Advice Evaluator	Advice Type: corresponde a quando a ação do advice deverá ser executada (*before, after, around*)
Advice Action	*Advice Action*: corresponde à ação a ser executada em um advice.

3.3.2 Mapeamento entre Metaspin e CaesarJ

(i) **Mapeamento de Aspect do Metaspin para Aspect em CaesarJ:** o aspecto do Metaspin possui correspondência direta para um aspecto em CaesarJ. Neste caso, o aspecto

tem seu nome, seus atributos e suas operações mapeados para o aspecto em CaesarJ, como ilustrado na Figura 33.

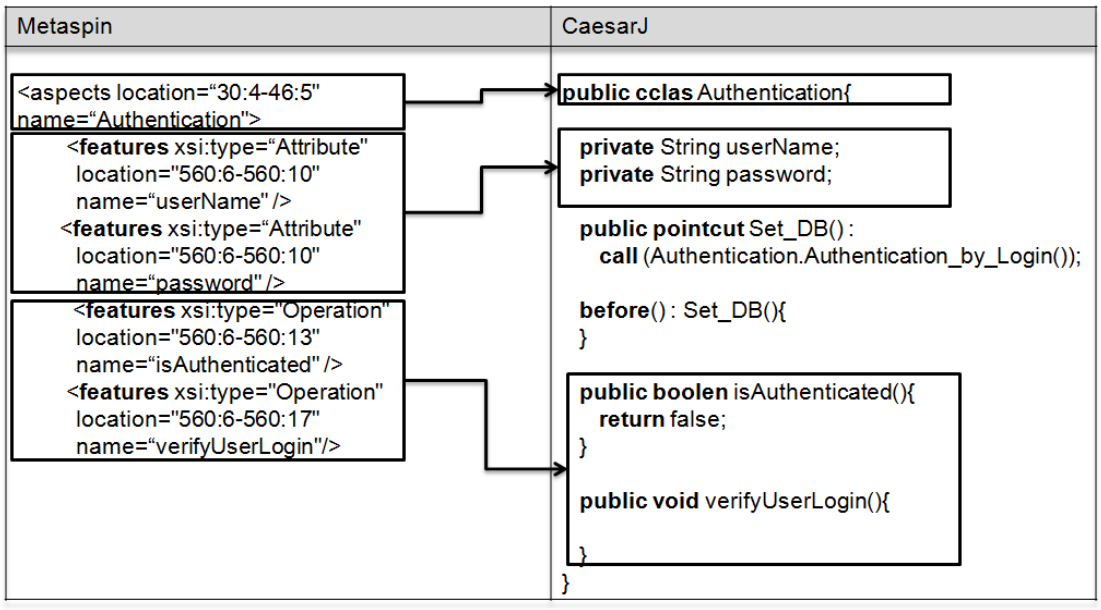

Figura 33 - mapeamento entre aspecto no Metaspin e aspecto em CaesarJ

Por convenção tratamos os atributos declarados em um aspecto com uma visibilidade privada (*private*), podendo ser acessados segundo a definição de métodos *getters* e *setters* gerados automaticamente para cada atributo durante a execução da regra de transformação.

(ii) **Mapeamento de *Join Point Selector* do Metaspin para o *Pointcut* de CaesarJ:** os elementos *JoinPointSelector* do Metaspin são aqui mapeados para *pointcut* pertencentes a um determinado aspecto CaesarJ (ver Figura 34). Os elementos *PoincutLanguageEvaluator* definidos dentro de um predicados de um aspecto do Metaspin é mapeado em CaesarJ para o tipo de chamada executada sob um determinado conjunto de *join points*. Em CaesarJ podemos definir chamadas do tipo *call,*e execution que estão associadas ao *pointcut* de um determinado aspecto.

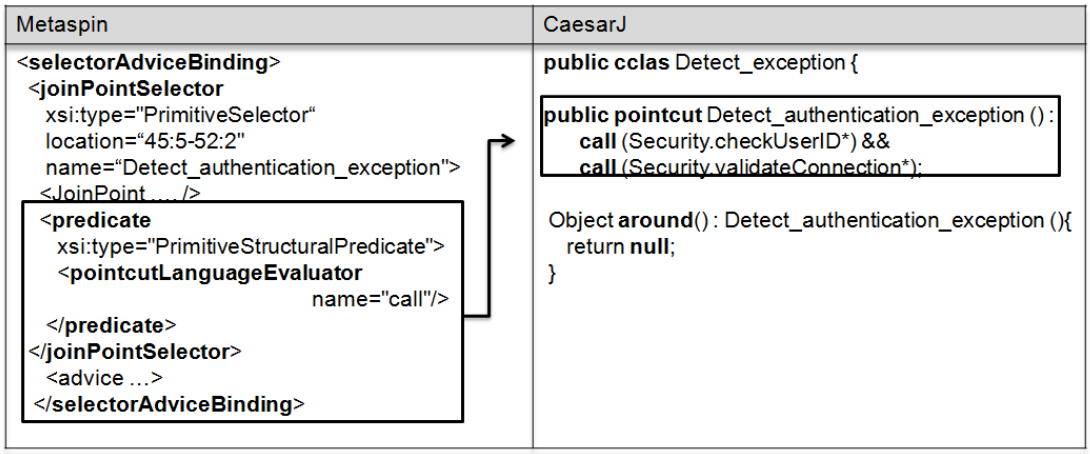

Figura 34 - mapeamento de join point selector para pointcut

(iii) **Mapeamento de *JoinPoint* do Metaspin para conjunto de *join points* em CaesarJ:** para o mapeamento de *join points* do Metaspin consideraremos os j*oin points* do tipo *ClassJoinPoint e MetlhodJoinPoint*. Dessa forma, cada um dentre esses tipos de join points são mapeados para o conjunto de *join points* que irão compor o *pointcut* de um aspecto. A Figura 35 ilustra o mapeamento de elementos do tipo *MethodJoinPointPart* expressos no Metaspin, os quais são representados como declarações de *join points* dentro dos *pointcuts* do aspecto em CaesarJ.

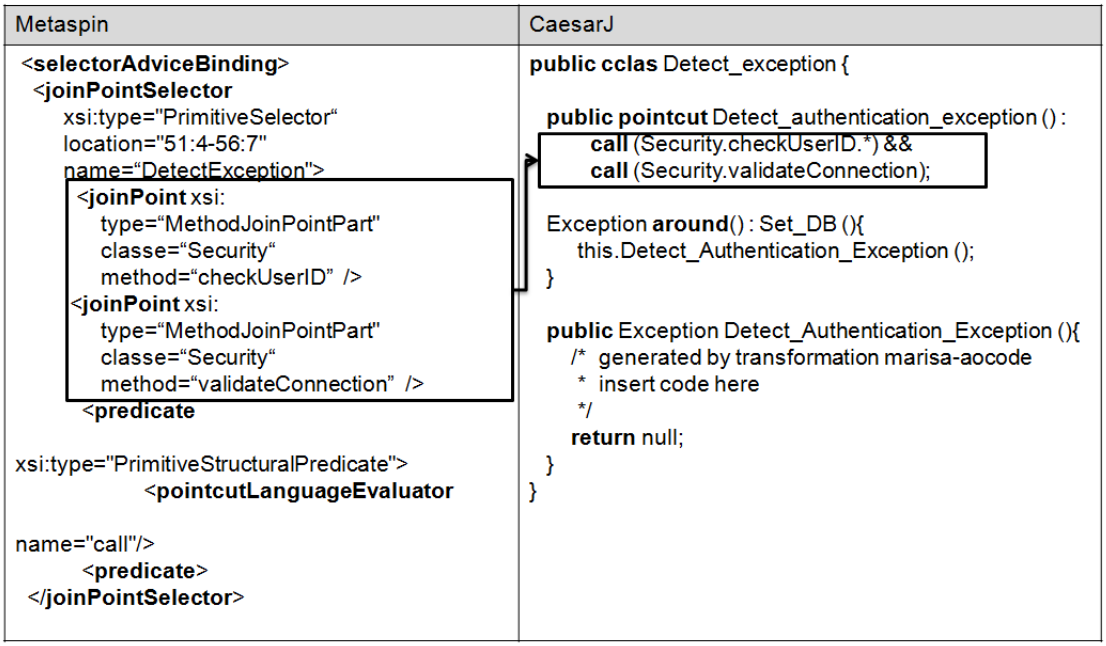

Figura 35 - Mapeamento de join points no Metaspin para join points em CaesarJ

Para este mapeamento definiu-se que: (i) se o elemento for um *ClassJoinPointPart* , o *join point* é definido como “with(nomeDaClasse)”, onde adicionamos os operadores lógicos do tipo “&&” ou ”||” no caso de haver mais de uma classe como *join point*. Se o *join point* for do tipo *MethodJoinPointPart*, definimos a declaração do *joinpoint* como sendo “(nomeDaClasse.nomeDoMetodo(..)*)”, onde os tipos de chamadas podem ser do tipo *execution* ou *call*.

(iv) **Mapeamento de *Advice* do Metaspin para *Advice* em CaesarJ** – os dois elementos que formam o *advice* em um aspecto no Metaspin, o *AdviceEvaluator e o AdviceType* são mapeados para a declaração do *advice* em CaesarJ. No caso, *AdviceEvaluator* do Metaspin é mapeado para o tipo do *advice*, sendo esta uma palavra reservada em CaesarJ que pode assumir os valores *before, after e around*. Por sua vez o *advice* é mapeado para um método na declaração do aspecto e este é referenciado dentro da declaração do *advice* do aspecto.

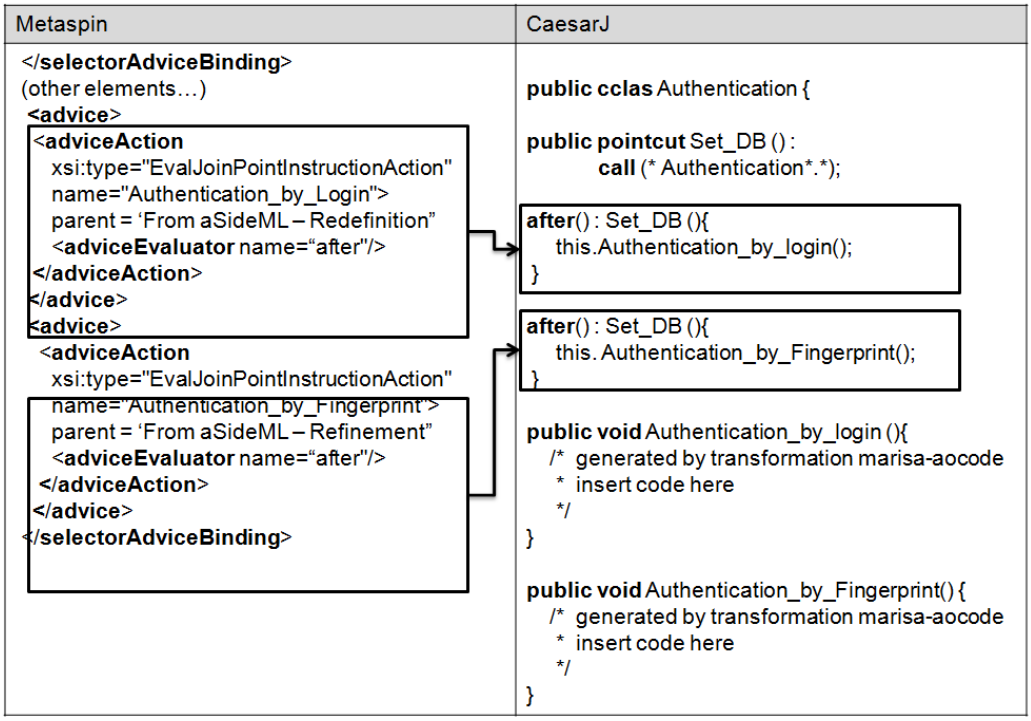

Figura 36 - Mapeamento de *AdviceAction* e *AdviceType* no Metaspin para *advice* em CaesarJ

Adicionalmente, o *advice* possui uma referência para o *pointcut* que ele afeta. Tal referência é proveniente do *JoinPoinSelector* do Metaspin ao qual esse *advice* esta ligado. A Figura 36 ilustra o mapeamento do *advice* do nível abstrato de programação para o *advice* do aspecto em CaesarJ. Na tabela 5 é mostrado um resumo do mapeamento entre os elementos aspectuais de Metaspin e AspectLua. O metamodelo CaesarJ estende os conceitos do

Metamodelo Metaspin, culminando em um mapeamento de um para um entre todos os elementos utilizados na construção de um aspecto em CaesarJ.

Tabela 5 - Mapeamento de Metaspin e CaesarJ

Metaspin		**CaesarJ**
Aspect		*Aspect*
Join Point Selector		*Pointcut*
PointcutLanguageEvaluator		Mapeado para o tipo de chamada a um determinado *join point* contido em um pointcut, podendo assumir valores do tipo *call, execution*
Joinpoint	ClassJoinPointPart	Declaração de *join poin em CaesarJ como uma expressão com tipo* with(nomeDaClasse)
	MethodJoinPointPart	Declaração de um *join point* contendo a classe e o método a que o mesmo pertence, em uma expressãp do tipo (nomeDaClasse.nomeDoMetodo(..)*)
Advice Evaluator		Advice Type: corresponde a quando a ação do advice deverá ser executada (*before, after, around*)
Advice Action		*Advice Action*: corresponde à ação a ser executada em um advice.

3.4 Mapeamento entre *CoreElement* e os modelos base específicos de plataforma

Nesta seção mostramos o mapeamento entre os elementos do *CoreElement* e os elementos de base expressos em uma linguagem específica. Para AspectLua, utilizamos como linguagem base a linguagem de *scripting* Lua. Por sua vez, para CaesarJ utilizamos como linguagem base, a linguagem Java, ambas com o intuito de representar o código base de uma aplicação OA.

3.4.1 Mapeamento entre *CoreElement* e Lua

(i) **Mapeamento de *Class*, *Attribute* e *Operation* do CoreElement para *Class*, *Attribute* e *Function***. Os elementos *Class* e os elementos *features* do tipo *Attribute* e *Operation* do metamodelo CoreElement possuem correspondentes diretos no metamodelo de AspectLua, sendo mapeados (ver Figura 37) para o conceito de *Class*, *Variable* e *Function* respectivamente. Por ser dinamicamente tipada, AspectLua não leva em consideração os tipos dos atributos de uma classe.

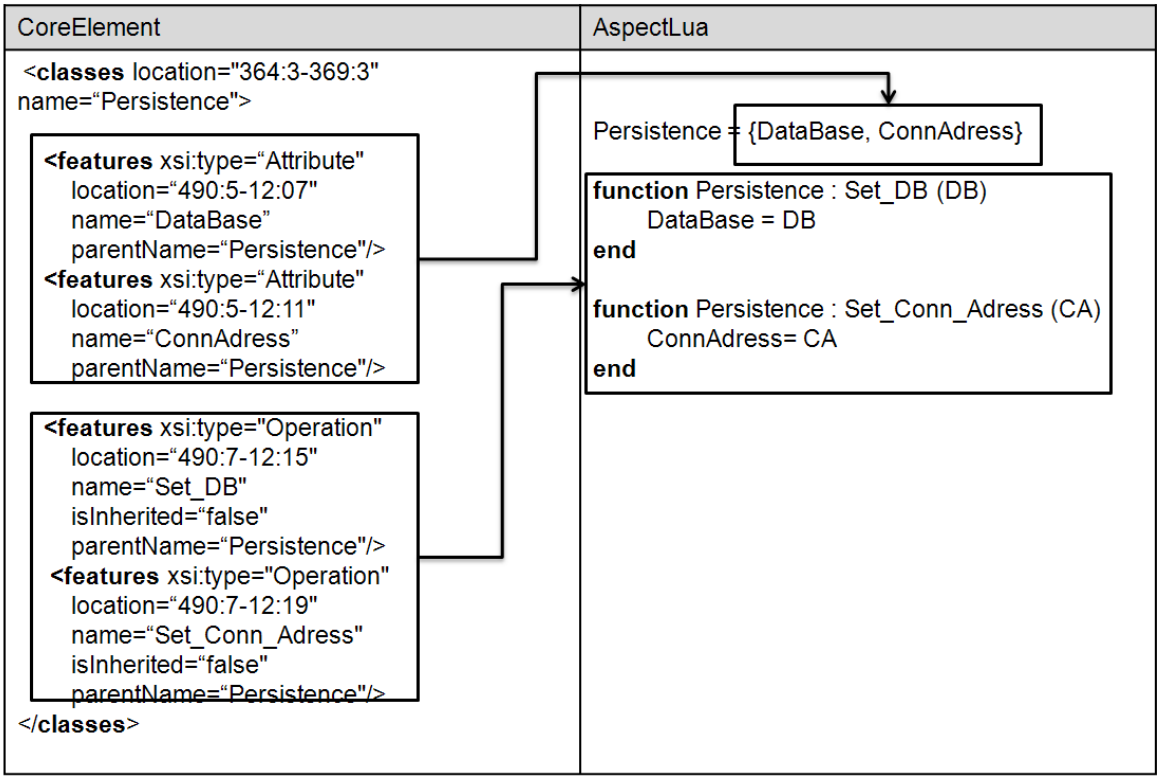

Figura 37 - Mapeamento de *Class*, *Attribute* e *Operation* no Metaspin para *Class*, *Variable* e *Function* em AspectLua

(ii) **Mapeamento de Interface do *CoreElement* para *Function* em AspectLua** – os elementos do tipo interface presentes no Metaspin não possuem mapeamento direto para um elemento correspondente na linguagem AspectLua. Dessa forma, em virtude de AspectLua não possuir o conceito de interface e para que não haja perda de informações na transformação entre os modelos, uma *Interface* tem os seus elementos *Operation* mapeados *function* em AspectLua. As assinaturas de operações advindas de interfaces são inseridos nas classes que implementam tais interfaces. Além disso, as operações devem vir precedidas por um comentário do tipo "-- Function inherited from interface nomeDaInterface", bem como por suas propriedades (Tags aSideML) que guardam informações advindas desde a fase de requisitos.

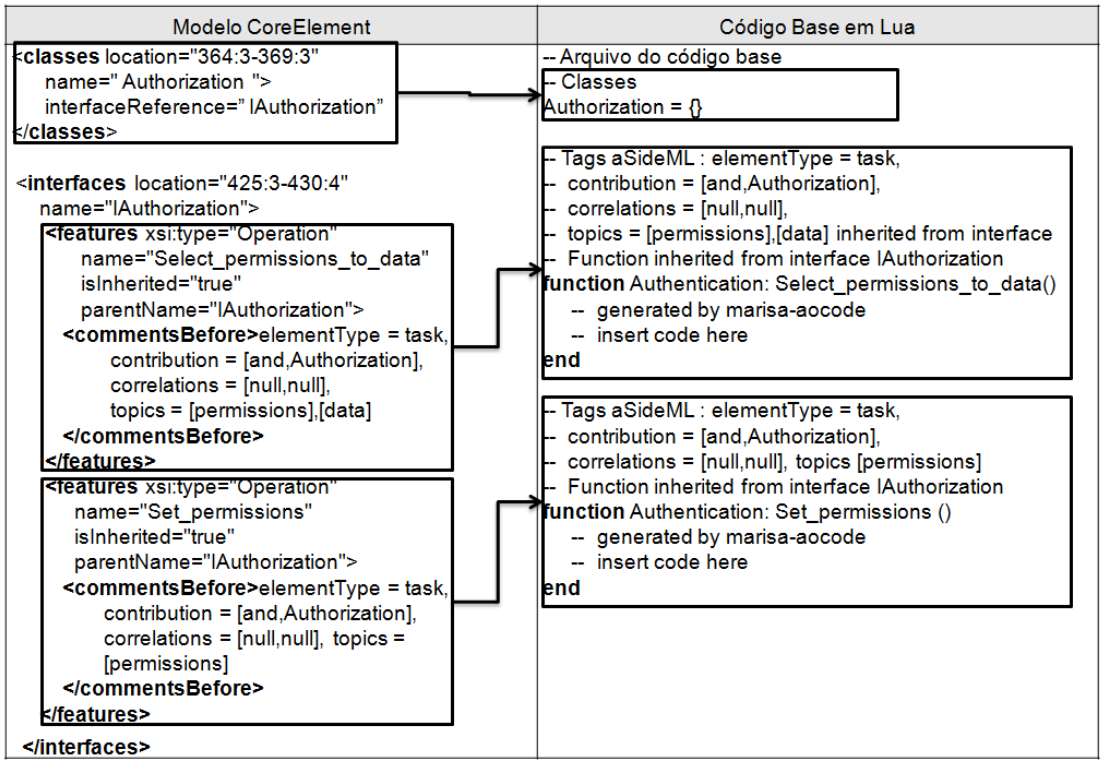

Figura 38 - Mapeamento de Interfaces para funções em classes AspectLua

A Figura 38 ilustra as operações de uma interface no modelo Metaspin sendo mapeadas para funções, que são inseridas na classe que implementa tal interface. A tabela 6 resume o mapeamento entre o *CoreElement* e AspectLua.

Tabela 6 - Mapeamento entre *CoreElement* e AspectLua

CoreElement	**AspectLua**
Class	Class
Attribute	Variable
Operation: precedida de uma notação especificando qual o tipo de características transversal essa operação representa, seja ela comportamental, estrutural ou requisitada.	Function
Interface	Function

3.4.2 Mapeamento entre *CoreElement* e Java

(i) **Mapeamento de Class, Attribute e Operation do CoreElement para Classe, *Attribute* e *Method* em Java -** Os elementos *Class*, *Attribute* e *Operation* do CoreElement são mapeados para classes, atributos e métodos no metamodelo CaesarJ. Para cada classe é gerada

uma classe em Java, bem como o seu construtor padrão e seu construtor com campos, caso esta classe possua atributos. Para cada *feature* do tipo Operation ou do tipo *Attribute* é criado um atributo corresponde na classe Java sendo este de visibilidade privada. Como forma de padronizar a definição de atributos foi decidido que todos eles serão declarados como privado (*private*). Além disso, para cada atributo serão gerados os respectivos métodos *get* e *set*, como forma de prover acesso a estes elementos. Por fim, para cada *feature* do tipo Operation no modelo *CoreElement* é criado um método na classe Java, com o mesmo tipo de retorno. A Figura 39 ilustra um exemplo de uma classe expressa no CoreElement, sendo mapeada para uma classe em Java.

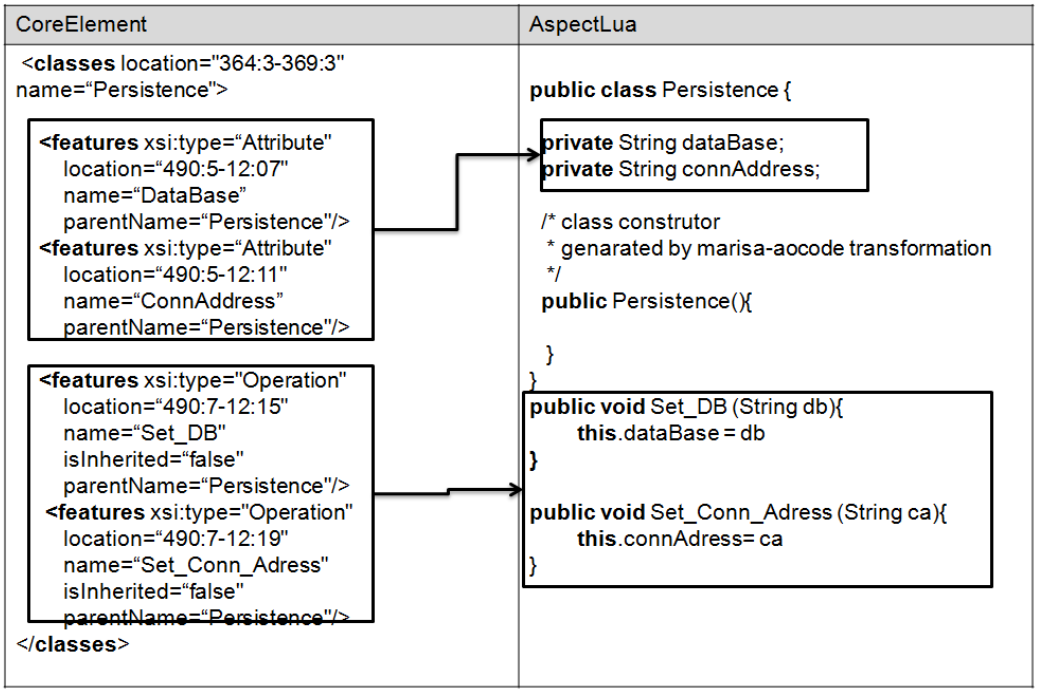

Figura 39 - Mapeamento de Class, Attribute e Operation no Metaspin para Class, Attribute e Operation em Java

(ii) **Mapeamento de Interface do CoreElement para Interface em Java**: o elemento interface no CoreElement possui correspondencia direta com elementos do metamodelo CaesarJ. Cada interface é mapeada para uma interface correspondente no metamodelo CaesarJ. O conjunto de *features* Operation da interface *CoreElement* é mapeado para assinaturas de métodos nas interfaces Java. Além disso nos casos em que uma classe possui referência para um ou mais interfaces no modelo CoreElement, são adicionadas as operações da(s) interface(s) que esta classe implementa e deve vir na declaração da classe a palavra reservada “implements” seguido do nome das respectivas interfaces. A Figura 40 ilustra um

exemplo onde uma classe implementa um interface e como esse mapeamento é realizado a geração de classes e interfaces Java.

Modelo CoreElement	Código Base em Java

<classes location="364:3-369:3"
name="Authentication">
interfaceReference="iAuthentication"
</classes>
 	
<interfaces location="425:3-430:4"
name="IAuthorization">
<features xsi:type="Operation"
name="Select_permissions_to_data"
isInherited="true"
parentName="IAuthorization">
<commentsBefore>elementType = task,
contribution = [and,Authorization],
correlations = [null,null],
topics = [permissions],[data]
</commentsBefore>
</features>
<features xsi:type="Operation"
name="Set_permissions"
isInherited="true"
parentName="IAuthorization">
<commentsBefore>elementType = task,
contribution = [and,Authorization],
correlations = [null,null], topics =
[permissions]
</commentsBefore>
</features>
</interfaces>
	public class Authentication
	implements iAuthentication {
	
	public Authentication () {
	
	}
	
	/* Tags aSideML : elementType = task,
	contribution = [and,Authorization], correlations = [null,null],
	topics = [permissions],[data] inherited from interface */
	/* Method inherited from interface IAuthorization */
	public void Select_permissions_to_data () {
	/* generated by marisa-aocode */
	/* insert code here */
	}
	
	/* Tags aSideML : elementType = task,
	contribution = [and,Authorization],
	correlations = [null,null], topics [permissions] */
	/* Function inherited from interface IAuthorization */
	public void Set_permissions () {
	/* generated by marisa-aocode */
	/* insert code here */
	}
	}
	
	public interface IAuthentication {
	
	public void Set_permissions (/*Parameters*/);
	public void Select_permissions_to_data (/*Parameters*/)
	}

Figura 40 - Mapeamento de Interfaces CoreElement para interfaces em Java

Além disso, as operações devem vir precedidas por um comentário do tipo "-- Function inherited from interface nomeDaInterface", bem como por suas propriedades (Tags aSideML) que guardam informações advindas desde a fase de requisitos. A tabela 7 ilustra o resumo do mapeamento entre os elementos do CoreElement e os elementos base do metamodelo CaesarJ, que no caso corresponde a linguagem Java.

Tabela 7 - Mapeamento de CoreElement para Java

CoreElement	**Java**
Class	Classe
Attribute	Attribute, onde os atributos são declarados com visibilidade privada e paa cada atributo pertencente a uma classe são criados métodos *get* e *set*.
Operation: precedida de uma notação especificando qual o tipo de características transversal essa operação representa, seja ela comportamental, estrutural ou requisitada.	Function
Interface	Interfaces Java. Cada interface é declarada na classe que a implementa, e as assinaturas de suas operações são inseridas dentro de tal classe. Além disso, os métodos provindos de uma interface são anotados com um comentário do tipo "Method inherited from interface nomeDaInterface".

4 Implementação Das Regras de Mapeamento Genéricas do Projeto Detalhado para Código Fonte em AspectLua e CaesarJ

Este capítulo tem por objetivo mostrar a implemetação das regras de mapeamento definidas no capítulo 3. Aqui iremos mostrar como as principais regras de mapeamento foram implementadas, ilustrando a correspondência entre os modelos de entrada e saída de cada fase definida no processo de transformação entre o projeto detalhado, o nível abstrato de programação e os modelos orientados a aspectos específicos de plataforma.

O processo de desenvolvimento de nossa abordagem, denominada MaRiSA-AOCode, inclui a especificação dos metamodelos e modelos das linguagens utilizadas em nível de projeto detalhado e codificação OA, assim como define um conjunto de regras de transformação seguindo o mapeamento entre os elementos de cada metamodelo de acordo com as regras definidas no capítulo 3. Além disso, como forma de validar as regras de transformação definimos um estudo de caso, baseado no sistema Health Watcher analisamos os modelos, bem como o código gerado como saída da aplicaçnao das transformações definidas em MaRiSA-AOCode.

A especificação dessas regras de mapeamento é realizada utilizando a linguagem de transformação ATL. A escolha da linguagem ATL (ver seção 2.5.1) se dá em virtude de esta já ter sido utilizada em outros trabalhos que fazem parte de um contexto mais amplo no qual este também está inserido. Para a transformação entre os modelos das fases de projeto detalhado e codificação foram utilizadas algumas tecnologias de desenvolvimento dirigido a modelos, de forma a facilitar e automatizar as transformações entre tais fases. Tais tecnologias foram apresentadasno capítulo 2, tais como ATL (2.5.1), KM3 (2.5.2) e TCS (2.5.3).

Na seção 4.1 é apresentada uma visão geral da abordagem, descrevendo os passos do processo de transformações entre aSideML, Metaspin e *CoreElement*, e AspectLua. A seção 4.2 apresenta as transformações dirigidas a modelos, entre os níveis de projeto detalhado OA, especificado em aSideML, e o nível de programação abstrato que é fornecido pelo Metaspin e *CoreElement*, respectivamente. O Metaspin é responsável por representar as abstrações de Orientação a Aspectos, como aspectos, *advice*, *pointcut* e *join point*. Por sua vez, o *CoreElement* é responsável por modelar os elementos do código base, como classes, operações, atributos. Essas transformações são realizadas em um nível de modelo independente de plataforma (PIM) da arquitetura MDA.

Nas seções 4.3 e 4.4 são ilustradas as regras de transformação dirigidas a modelos entre o nível de implementação abstrato de linguagens OA, para as linguagens de programação OA específicas de plataforma, AspectLua e CaesarJ, respectivamente. Neste caso, temos transformações entre os níveis PIM, onde se tem o Metaspin conferindo um nível abstrato de programação OA, e o nível PSM representado pelos modelos AspectLua e CaesarJ, a partir dos quais será gerado o código OA nas respectivas linguagens. A seção 4.5 apresenta um estudo de caso como forma de ilustrar o funcionamento da ferramenta, bem como os modelos gerados em cada fase do processo de MaRiSA-AOCode e o código fonte OA em AspectLua e CaesarJ. A seção 4.6 ilustra a utilização da ferramenta MaRiSA-AOCode.

Por fim, na seção 4.7 é apresentada uma análise dos modelos gerados em cada fase do processo de mapeamento do projeto detalhado para o código fonte em AspectLua e CaesarJ. Ou seja, são analisados os modelos os modelos abstratos gerados a partir de uma especificação de projeto detalhado em aSideML, bem como os modelos específicos de plataforma gerados a partir de um modelo Metaspin. Neste sentido, tanto os modelos quanto os códigos fonte gerados em cada linguagem de programação OA são analisados segundo critérios de completude, rastreabilidade e corretude.

4.1 Visão Geral da Abordagem

O processo de transformação (ver Figura 41) é organizado em três fases: (i) elaboração e verificação de modelos; (ii) Mapeamento intermediário entre modelos (aSideML, Metaspin e, AspectLua ou CaesarJ); e (iii) Geração de código em AspectLua ou CaesarJ. Adicionalmente, existe a fase de preparação, onde são definidos os metamodelos de cada linguagem, a sintaxe textual de acordo com a BNF das linguagens, bem como as regras de transformação especificadas em ATL. Como forma de simplificar a visualização do processo de mapeamento, não será ilustrada a fase de preparação.

A entrada do processo de transformação entre aSideML e Metaspin é uma representação textual de um modelo aSideML. Esse modelo textual pode ser obtido de duas formas: (i) manualmente; e (ii) automaticamente, através da transformação entre AOV-Graph [Silva,2006] e aSideML [Chavez, 2004], realizada pela ferramenta MaRiSA-MDD [Medeiros, 2008]. Quando o modelo textual aSideML é obtido manualmente, ele deve ser verificado de forma que somente os modelos bem-formados sejam utilizados no processo de transformação. Em contrapartida, quando o modelo é obtido automaticamente, assume-se que ele seja bem-

formado, uma vez que foi obtido de um processo de transformação que possui uma rigorosa verificação da semântica dos elementos do modelo. Essa verificação é necessária para garantir que nenhum erro foi introduzido por falha humana.

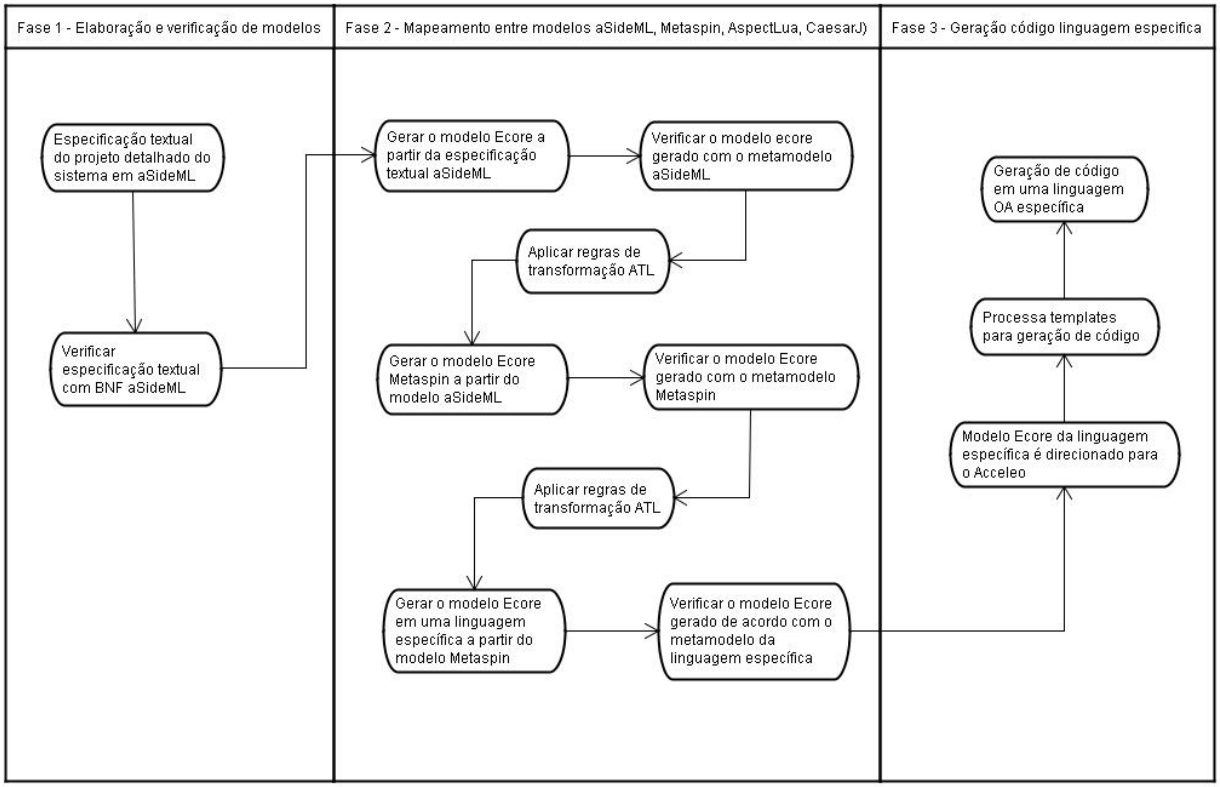

Figura 41 – Processo de Transformação de MaRiSA-AOCode

Na Fase 1 é realizada a especificação do projeto detalhado do sistema. A especificação textual aSideML é transformada em um modelo Ecore aSideML. Nessa transformação é verificado se a especificação textual está de acordo com a sintaxe textual descrita com a utilização do componente TCS (ver seção 2.5.3), de forma a validar se o modelo gerado a partir da especificação textual é um modelo bem formado e se este esta em conformidade com o metamodelo definido para a linguagem aSideML.

Na Fase 2 são realizadas as transformações entre os modelos aSideML e Metaspin e, deste último para AspectLua. Adicionalmente, é realizada a verificação da conformidade de cada um desses modelos com seus respectivos metamodelos. Para cada etapa de mapeamento são definidas regras de mapeamento especificadas com a linguagem ATL. A linguagem ATL atua entre o modelo Ecore de aSideML para o modelo Ecore Metaspin, verificado de acordo com o modelo Metaspin (definido em KM3). Em seguida ocorre a transformação entre o modelo Ecore Metaspin para o modelo Ecore AspectLua ou CaesarJ, dependendo de qual linguagem o desenvolvedor esteja trabalhando. Os passos de verificação entre os modelos das

linguagens são necessários visto que o modelo Ecore pode ter sido gerado a partir de especificações mal-formadas.

Na primeira transformação é realizado o mapeamento entre os elementos de aSideML e do Metaspin. Para esta transformação é utilizado um modelo de entrada bem-formado, provindo da fase 1 do processo de transformação, onde a especificação textual aSideML foi transformada em um modelo aSideML. Uma vez realizada essa primeira transformação temos como resultado um modelo de saída Metaspin, que está em conformidade com seu respectivo metamodelo (definido em KM3). Essa conformidade é garantida pelas regras de transformação ATL que verificam a conformidade do modelo gerado para cada elemento que as regras geram para o modelo de saída. O modelo Metaspin gerado é, por sua vez, utilizado como modelo de entrada para a segunda transformação que visa gerar um modelo AspectLua. Da mesma forma que a primeira transformação da fase 2, o modelo de AspectLua gerado deve estar em conformidade com seu metamodelo.

Na fase 3 é feita a geração do código a partir do modelo AspectLua ou CaesarJ gerado como resultado final das transformações que foram definidas na fase 2. O modelo AspectLua ou CaesarJ gerado é direcionado como modelo de entrada para o Acceleo. O Acceleo processa esse modelo através de seus scripts definidos para cada caso. Os templates são pré-definidos pelo desenvolvedor de acordo com o interesse em uma linguagem específica que ele deseje gerar código orientado a aspectos. A geração do código de acordo com os respectivos metamodelos, AspectLua ou CaesarJ é feita de forma particular para cada caso.

Quando o desenvolvedor opta pela geração de um código em AspectLua são gerados dois arquivos com extensão lua, são eles: (i) arquivo *aspecto.lua*, onde estará contido o código referente aos aspectos que foram gerados através do processo de transformação; e (ii) *base.lua*, onde se encontra as classes que correspondem a parte base de uma aplicação. O *weaving* é realizado com o interpretador Lua. Não é necessário um interpretador específico para tratar o *weaving* de aspectos e código base, visto que AspectLua foi construída com a própria linguagem Lua, sem a necessidade de alterações no interpretador.

Por outro lado, o desenvolvedor pode optar pela geração de código em CaesarJ. Neste caso, o código base é gerado em forma de pacotes, classes e interfaces Java, por meio dos scripts de transformação do Acceleo. Para o código aspectual são gerados os aspectos de acordo com a modelagem de aspectos feita em aSideML, e estes também são gerados em seus devidos pacotes. O *weaving* neste caso é realizado em tempo de compilação utilizando um compilador CaesarJ para realizar o *binding* entre o código base e o código do aspecto.

4.2 Transformação de aSideML para Metaspin

Esta seção trata da implementação das regras de transformação entre aSideML e Metaspin, que correspondem a Fase 1 e a Fase 2 (primeira transformação) de acordo com o processo de mapeamento ilustrado na Figura 41 ilustrada no inicio deste capítulo. Para exemplificar o processo de mapeamento, as Figuras 42 a 43 ilustram as principais regras de transformação.

Na Figura 42 é definida a regra *AspectaSideML2AspectMetaspin*, responsável por realizar o principal mapeamento dentre as transformações de aSideML para Metaspin. Esta regra realiza o mapeamento de praticamente todos os elementos que compõe um aspecto no Metaspin, tais como *advice* (*Advice Action* e *Advice Evaluator*), *Join points*, *pointcuts* (*Join Point Selectors*), expressão de *pointcut* (*Predicate*), dentre outros.

Para cada aspecto em um modelo em aSideML é criado um aspecto correspondente no Metaspin. Para as regras de transformação, um aspecto no modelo Metaspin é expresso como um conjunto de *variables*, *functions*, e *selector advice bindings*. Os *selector advice binding* são elementos do modelo Metaspin responsáveis por encapsular um conjunto de pares (*advice*, *pointcut*) pertencentes a um determinado aspecto. Para a regra de mapeamento *Aspect2Aspect*, um aspecto tem um conjunto de elementos *attributes* e *operations*, que de um aspecto pertencente a um modelo aSideML são consultados no modelo através de laços de interação, de forma que tenha-se conjuntos de variables e functions correspondentes expressos no modelo Metaspin.

Os *advices*, bem como suas ações são obtidos através da consulta de *redefinitions* e *refinements* contidos em um ou mais interfaces transversais (*Crosscutting Interface*) a que o aspecto faz referência. Para a consulta de interfaces transversais é necessário saber quais os relacionamentos transversais em que o aspecto está envolvido, através da utilização de helpers ATL. Dessa forma, para cada *redefinition* ou *refinement* encontrados nas interfaces transversais pertencentes a um relacionamento em que o aspecto está relacionado em aSideML, é dado origem a um elemento *advice action* e *advice evaluator* que irão compor um dos *advices* do aspecto no Metaspin.

De forma similar ocorre com o *join points*. Para cada relacionamento crosscutting (*Crosscutting Relationship*) é definido um elemento Base afetado pelo aspecto e o conjunto e *matches* relacionados a esse elemento base. Cada um dos *matches* sejam eles classes, operações ou declarações pertencentes aos componentes base, são transformado para um join

point, que no final irão compor a lista de *join points* a que o *pointcut* do aspecto fará referência. Os pointcuts (*Join Point Selector*) gerados para o modelo Metaspin são provenientes dos elementos *Template Parameter* de aSideML. Logo, para cada *Template Parameter* que um aspecto em aSideML fizer referência será originado um *pointcut* correspondente para este aspecto no modelo Metaspin.

Transformação AspectaSideML2AspectMetaspin.atl

```
rule AspectaSideML2AspectMetaspin{
  from
    sapect: aSideML!Aspect
  to
          asp : metaspin!"metaspin::Aspect"(
          name <- aspect.nameCrosscuttingElement,
          variable <- variables,
          function <- operations,
          location <- aspect.location,
          selectorAdviceBinding <- sab
  ),

  variables : distinct metaspin!"metaspin::Variable"
      foreach (attr in aspect.attributes)(
          name <- attr.name,
           type <- attr.type
  ),
  operations : distinct metaspin!"metaspin::Function"
            foreach (opr in aspect.operations)(
             name <- opr.name,
             type <- opr.type,
             return <- opr.return
            ),
  sab : metaspin!"metaspin::SelectorAdviceBinding"(
          advice <- thisModule.Refinement2Advice(
                    thisModule.getCrosscuttingRefinements(
            thisModule.getRelationshipCrosscutting(
            asp.nameCrosscuttingElement)
                              .interface.nameCrosscuttingElement)),
          jps <- thisModule.TempateParameter2JoinPointSelector()

          )
}

rule RelationshipCrosscutting2SelectorAdviceBinding{
  from
          cross_rel : aSideML!RelationshipCrosscutting
  to
    sa_binding : metaspin!"metaspin::SelectorAdviceBinding"(
          name <- cross_rel.name
   )
}

rule TempateParameter2JoinPointSelector{
  from
          template : aSideML!TemplateParameter
  to
          jps : metaspin!"metaspin::PrimitiveSelector"(
                    name <- template.operations.name,
                    location <- template.location,
```

```
                crosscuttingReference <- template.interface.name,
                predicate <- pse
        ),
        pse : metaspin!"metaspin::PrimitiveStructuralPredicate"(
                name <- template.nameTemplateParameter,
                pointcutLanguageEvaluator <- ple
        ),
        ple : metaspin!"metaspin::PointcutLanguageEvaluator"(
                name <- 'call'
        )
}

rule Redefinition2Advice {
  from
        red: aSideML!Redefinitions
  to
        advice : metaspin!"metaspin::Advice"(
                adviceAction <- ejpi,
                commentsBefore <- 'From aSideML - Redefinition',
                crosscuttingReference <- red.interface.name
        ),

        ejpi : metaspin!"metaspin::EvalJoinPointInstructionAction"(
                name <- red.operationRedefinitions.name,
                adviceEvaluator <- ae
        ),

        ae : metaspin!"metaspin::AdviceEvaluator"(
                name <- red.adornoTextual.name
        )
}

(continua...)
```

Figura 42 – Transformação ATL de Aspecto em aSideML para Aspecto no Metaspin

A Figura 43 ilustra definição de duas regras *Class2Class* e *Interface2Interface*, responsáveis por realizar o mapeamento direto de Classes e Interfaces em aSideML, para os elementos *Class* e *Interface no CoreElement*.

Para cada classe em um modelo de entrada aSideML, as transformações ATL verificam seus atributos e operações armazenando-os em um conjunto de features que serão atribuídos a classes no modelo de saída Metaspin. Cada operação tem seu campo *isInherited* atribuído com o valor *false*, visto que essas operações são da própria classe e não foram herdadas de outra interface. A regra *Interface2Interface* também organiza as operações das interfaces provindas do modelo aSideML em um conjunto de features, que serão atribuídas a uma interface correspondente no modelo de saída Metaspin.

Transformação aSideML2CoreElement.atl

```
rule Class2Class{
 from
        classe : aSideML!Class
 to
        class: metaspin!"CoreElement::Class"(
                name <- classe.name,
                location <- classe.location,
                features <- attributes->union(operations)
        ),
        attributes: distinct metaspin!"CoreElement::Attribute"
            foreach (attr1 in classe.operations)(
                 name <- attr 1.name,
        ),
        operations: distinct metaspin!"CoreElement::Operation"
            foreach (op1 in classe.operations)(
                name <- op1.name,
                parentName      <- classe.name,
                isInherited <- false
                )
  }

rule Interface2Interface{
 from
        inter : aSideML!Interface
 to
        interface: metaspin!"CoreElement::Interface"(
                name <- inter.name,
                operations <- ops
        ),

        ops: distinct metaspin!"CoreElement::Operation"
              foreach (op2 in thisModule.
          getInterfaceOperations(interface.name))(
                        name     <- op2.name,
                        location <- op2.location
        )
}
```

Figura 43 – Transformação ATL de Classes e Interfaces em aSideML para Classes e Interfaces no Metaspin

4.3 Transformação de Metaspin para AspectLua

Nas Figuras 44 e 45 são definidas as principais regras de transformação correspondentes a segunda transformação da Fase 2 do processo, a qual contempla a transformação do modelo Metaspin / *CoreElement* para um modelo AspectLua, a partir do qual será gerado o código orientado a aspectos nas respectivas linguagens, dependendo de qual linguagem o desenvolvedor deseje utilizar..

A Figura 44 ilustra a regra *MetaspinAspect2AspectLuaAspect*, responsável por transformar um aspecto e seus elementos expressos em um nível abstrato de programação suportado pelo Metaspin, para elementos de um modelo AspectLua especifico de plataforma.

Diferentemente de aSideML, onde os elementos necessários para compor um aspecto no Metaspin estão dispersos, na transformação entre os modelos Metaspin e AspectLua, o mapeamento ocorre praticamente de forma direta.

Transformação Metaspin2AspectLua.atl

```
rule MetaspinAspect2AspectLuaAspect{
 from
        aspect : metaspin!Aspect
 to
        asp : AspectLua!Aspect(
                name <- aspect.name,
                location <- aspect.location,
                advice <- adv,
                pointcut <- pcut
        ),
        adv : AspectLua!Advice(
                adviceType <- aType,
                adviceAction <- aAction
        ),
        aType : AspectLua!AdviceType(
                name <- aspect.selectorAdviceBinding.
            advice.adviceAction.adviceEvaluator.name,
                location <- aspect.selectorAdviceBinding.
            advice.adviceAction.adviceEvaluator.location
        ),
        aAction : AspectLua!AdviceAction(
                name <- aspect.selectorAdviceBinding.
            advice.adviceAction.name,
                location <- aspect.selectorAdviceBinding.
              advice.adviceAction.location
        ),
        pcut : AspectLua!Pointcut(
                pointcutname <- aspect.selectorAdviceBinding.
             joinPointSelector.joinPoint.classe + 'pointcut',
                location <- aspect.selectorAdviceBinding.
             joinPointSelector.joinPoint.location,
                designator <- designador,
                joinpoints   <- joinPoint
        ),
        joinPoint : AspectLua!JoinPoint(
                 pointcutExpression <- pointcutExpress
        ),
        pointcutExpress : AspectLua!PointcutExpression(
              regularExpression <- aspect.selectorAdviceBinding.
               joinPointSelector.joinPoint.classe +'.*'
        ),
        designador : AspectLua!PointcutDesignator(
        name <- 'call'
        )
}
```

Figura 44 – Transformação ATL de Aspecto em Metaspin para Aspecto em AspectLua

A regra *MetaspinAspect2AspectLuaAspect* considera um aspecto em AspectLua como um elemento que contem um *name*, e um conjunto de *advices* e *pointcuts*. Em virtude da

declaração de um aspecto em AspectLua ter um acoplamento alto entre o *advice* e o *pointcut*, para cada aspecto existente em um modelo de entrada Metaspin é gerado um aspecto no modelo AspectLua, e para cada *Selector Advice Binding* é replicada a declaração do aspecto, alterando-se somente o pointcut e o *advice* de acordo com os elementos definidos em cada *Advice Binding Selector* a que o aspecto no modelo Metaspin fizer referência.

Para cada elemento *Advice Evaluator* e *Advice Action* que compõe o *advice* no modelo Metaspin é criado um elemento *Advice Type* e *Advice Action* correspondentes no modelo AspectLua de saída. De forma similar, para cada elemento *Join Point Selector* é criado um elemento *pointcut* correspondente no modelo AspectLua.

A representação de *join points* no Metaspin pode ser realizada por elementos distintos, tais como *ClassJoinPointPart* e *MethodJoinPointPart*, que representam *join points* como classes e métodos respectivamente. Cada um desses elementos é incorporado a uma expressão de pointcut (*Pointcut Expression*), que ao final deverá conter um conjunto dos *join points* que o aspecto deve afetar. Para cada elemento *join point* que for representado por um tipo *ClassJoinPointPart*, deve ser adicionado ao final do nome da classe a expressão ".*" quando forma mapeado para os *join points* de AspectLua, de forma a denotar que todas as operações daquela dada classe serão afetadas pelo aspecto. Depois de formada a expressão de *pointcut*, esta deve ser inserida no elemento *pointcut* correspondente.

A Figura 45 ilustra a regra de mapeamento *CoreElement2AspectLua*, responsável por mapear as classes e interfaces de aSideML para a abstração de classes provida por AspectLua. Nesta regra de transformação as classes providas por AspectLua são elementos que possuem um conjunto de *variables* e *functions*. Para cada Attribute de uma classe *CoreElement* é criado um elemento do tipo *Variable* em AspectLua, sento este atribuído a respectiva classe no modelo AspectLua gerado. Para cada Operation proveniente do *CoreElement*, temos duas situações: (i) caso a operação seja advinda de uma classe, o seu atributo *isInherited* tem o valor *false* atribuído e o seu atributo *parentName* recebe o nome da classe; e (ii) caso a operação seja proveniente da interface que a classe *CoreElement* herda, seu atributo *isInherited* tem o valor *true* atribuído e o seu atributo *parentName* recebe o nome da respectiva interface. Ao final, esse conjunto de *functions* são atribuídos a uma classe AspectLua.

Transformação CoreElement2AspectLua.atl

```
rule Class2Class{
 from
    classe: metaspin!"CoreElement::Class"
 to
          class: AspectLua!Class(
                    name <- classe.name,
                    location <- classe.location,
         variables < - vars,
                    functions <- functions1->union(functions2)->flatten()
          ),

    vars: distinct AspectLua!Function
              foreach (op1 in classe.operations)(
                    name <- op1.name,
                    location <- op1.location,
          ),

          functions1: distinct AspectLua!Function
              foreach (op1 in classe.operations)(
                     name <- op1.name,
                    location <- op1.location,
                    parentName        <- classe.name,
                    isInherited <- op1.isInherited
          ),

          functions2: distinct AspectLua!Function
                    foreach (op2 in thisModule.getInterfaceOperations(
                    classe.interface.name))(
                    name      <- op2.name,
                    location <- op2.location,
                    parentName  <- classe.name,
                    isInherited <- true
          )
  }
```

Figura 45 – Transformação ATL de Classes e Interfaces no CoreElement para Classes em AspectLua

4.4 Transformação de Metaspin para CaesarJ

Nas Figuras 46, 47 e 48 são definidas as principais regras de transformação correspondentes a segunda transformação da Fase 2 do processo, a qual contempla a transformação do modelo Metaspin para um modelo CaesarJ, a partir do qual será gerado o código orientado a aspectos.

A Figura 46 ilustra a regra *MetaspinAspect2CaesarJAspect.atl* que tem por objetivo compor os aspectos na linguagem CaesarJ com seus respectivos elementos a partir de um modelo abstrato de programação expresso pelo Metaspin. A transformação de um modelo Metaspin para o modelo CaesarJ é feita praticamente de forma direta. Isso se deve em virtude do metamodelo CaesarJ ter sido construído com base em estudos das principais abstrações de orientação a aspectos providas pelo CaesarJ, ou seja, as abstrações providas pela linguagem

foram mapeadas para elementos específicos do Metaspin, de forma que cada conceito em CaesarJ estende um elemento do Metaspin. A regra considera um aspecto em CaesarJ como um elemento que contém um *name*, e um conjunto de *advices* e *pointcuts*. Para cada elemento aspecto no Metaspin é gerado um aspecto correspondente no modelo CaesarJ. Diferentemente de AspectLua não é necessário gerar uma nova declaração de um aspecto para diferentes *binding* entre os *pointcuts* e *advices* que esse aspecto pode ter. A declaração do aspecto é feita somente uma vez para cada elemento e o aspecto tem diversas declarações de *pointcuts* e *advices* pertencentes *ao mesmo.* Para cada aspecto do Metaspin é gerado um aspecto correspondente em CaesarJ herdando o nome, bem como seu conjunto de operações e atributos. Para cada *Selector Advice Binding* é gerado um relacionamento entre os *pointcuts* e *advices* que esse aspecto contém. Os elementos *Join Point Selector* e os elementos *Advice* são mapeados para as declarações de *pointcut* e *advice* em CaesarJ, respectivamente.

No caso do *Advice,* cada elemento *Advice Evaluator* e *Advice Action* que compõe o *advice* no modelo Metaspin é criado um elemento *Advice Type* e *Advice Action* correspondentes no modelo CaesarJ. Por fim, para cada tipo de *join point* tratado no modelo Metaspin é gerada uma declaração apropriada para cada um deles (ver regras de mapeamento do capitulo 3).

Transformação MetaspinAspect2CaesarJAspect.atl

```
rule AspectMetaspin2AspectCaesarJ{
  from
    aspect: metaspin!"metaspin::Aspect"
  to
          asp : Caesar!Aspect(
          name <- aspect.name,
          attributes <- attributesList,
          methods <- methodsList,
          advices <- CaesarJ!Advice.AllInstances(),
     pointcuts <- CaesarJ!Pointcut.AllInstances()
  ),
  attributesList: distinct CaesarJ!Attribute
      foreach (attr in aspect.attributes)(
                    name <- attr.name,
      type <- thisModule.vefificarTipoAtributo(attr)
  ),

  methodsList: distinct CaesarJ!Method
            foreach (opr in aspect.operations)(
                     name <- opr.name,
       type <- thisModule.vefificarTipoMetodo(opr),
       return <- thisModule.verificarRetornoMetodo(opr)
  )
}

rule JoinPointSelector2Pointcut{
  from
          jps: metaspin!"metaspin::JoinPointSelector"
```

```
  to
        pointcut: CaesarJ!Pointcut(
                name <- jps.name,
                interfaceReference <- jps.crosscuttingReference.name,
                joinpoints <- thisModule.ClassJoinPoint2JoinPoint(jps.name)
                        ->union (thisModule.MethodJoinPoint2JoinPoint(jps.name))
        )
}

rule ClassJoinPointClass2JoinPoint{
        from
                cjp : metaspin!"metaspin::ClassJoinPointPart"
        to
                joinpoint : CaesarJ!JoinPoint(
                        name <- cjp.classe,
                        callType <- thisModule.obterTipoChamadaJoinPoint(cjp)
                )
}

rule MethodJoinPointClass2JoinPoint{
        from
                mjp : metaspin!"metaspin::MethodJoinPointPart"
        to
                joinpoint : CaesarJ!JoinPoint(
                        className <- mjp.class,
                        methodName <- mjp.operation,
                        callType <- thisModule.obterTipoChamadaJoinPoint(mjp)
                )
}

rule AdviceMetaspin2AdviceCaesarJ {
  from
        jpi: metaspin!"metaspin::Advice"
  to
        advice : CaesarJ!Advice(
                adviceAction <- jpi.obterAcaoAdvice(),
                adviceType <- jpi.obterAcaoAdvice(),
                commentsBefore <- 'From aSideML - Redefinition',
                interfaceReference <- jpi.crosscuttingReference.name,
                pointcutReference <- jpi.joinPoinSelectorReference
        )
}
```

Figura 46 – Transformação ATL de Aspecto em Metaspin para Aspecto em CaesarJ

As Figuras 47 e 48 ilustram as duas principais regras de mapeamento *ClassCoreElement2ClassCaesarJ* e *InterfaceCoreElement2InterfaceCaesarJ*, responsáveis por mapear as classes e interfaces de um modelo Metaspin para a classes e interfaces Java. Na regra *ClassCoreElement2ClassCaesarJ* (ver Figura 47) de transformação as classes providas Java são elementos que possuem um conjunto de *atributos* e métodos. Para cada *Attribute* de uma classe *CoreElement* é criado um atributo correspondente na classe Java. Além disso, para cada atributo são criados os respectivos métodos *get* e *set*. Por sua vez, para cada *Operation*

no modelo *CoreElement* é criado o método correspondente na classe Java. Por padrão adotamos que as operações terão visibilidade do tipo *public* e que terão retorno do tipo *void*. Essa decisão foi tomada visto que os modelos aSideML gerados automaticamente pela ferramenta MaRiSA-DP [Montenegro, 2008] não definem tipo de retorno para as operações expressas nos modelos aSideML.

Transformação CoreElement2CaesarJClass.atl

```
rule ClassCoreElement2ClassCaesarJ{
 from
  classe: metaspin!"CoreElement::Class"
 to
   class : CaesarJ!Class(
            name <- classe.name,
            location <- classe.location,
            featuresAttribute <- atributos,
    featuresMethods <- metodos,
    visibility <- 'public'
    ),
            atributos : distinct CaesarJ!Attributo
          foreach (attr in classe.attributes)(
                        name                <- attr.name,
                        location <- attr.location,
       visibility  <-              'public'
            ),
            metodos: distinct CaesarJ!Method
          foreach (op1 in classe.operations)(
                        name                <- op1.name,
                        location <- op1.location,
                        parentName        <- classe.name,
                        isInherited <- false,
       visibility  <- 'public',
       return      <- op1.verifyMethodReturn()
            )
}
```

Figura 47 – Transformação ATL de Classes e Interfaces no CoreElement para Classes em CaesarJ

A regra de mapeamento *InterfaceCoreElement2InterfaceCaesarJ.atl* ilustrada na Figura 48 é responsável pelo mapeamento das interfaces do modelo CoreElement para interfaces em Java. Para cada interface Metaspin criamos uma interface Java correspondente. Cada interface herda o nome e os métodos da interface que lhe deu origem. Adicionalmente, as assinaturas dos métodos serão incorporadas na classe que implementa a interface em tempo de geração de código a partir do modelo CaesarJ. Da mesma forma como definimos para classes, cada *Operation* no modelo CoreElement é criado o método correspondente na classe Java, que por padrão terá visibilidade do tipo *public* e que seu retorno do tipo *void*.

Transformação *InterfaceCoreElement2InterfaceCaesarJ*.atl

```
from
  interfaceCE : CaesarJ!Interface
to
  interface : CaesarJ!Class(
          name <- interfaceCE.name,
          location <- interfaceCE.location,
    featuresMethods <- metodos,
    visibility <- 'public'
  ),

   metodos: distinct CaesarJ!Method
     foreach (op1 in interfaceCE.operations)(
           name              <- op1.name,
           location          <- op1.location,
           parentName        <- interfaceCE.name,
           isInherited <- true,
     visibility  <- 'public',
     returnType  <- op1.verifyMethodReturn()
          )
}
```

Figura 48 - transformação de interfaces CoreElement para interfaces Java

4.5 Estudo de Caso

Como forma de ilustrar e avaliar as regras de mapeamento e os modelos gerados usamos o sistema Health Watcher (HW) [Soares, 2002], visto que o mesmo é bastante utilizado em pesquisas da área de Desenvolvimento de Software Orientado a Aspectos. O HW tem por objetivo realizar a coleta e gerenciar denúncias e notificações relacionadas à area de saúde pública, bem como notificar a população a respeito de informações gerais sobre saúde pública [Soares, 2002]. As Figuras 49 a 52 mostram os resultados obtidos em cada etapa do processo de mapeamento entre aSideML, Metaspin e AspectLua.

A Figura 49 ilustra a descrição textual em aSideML e o modelo Metaspin gerado, resultante do processo de transformação de modelos. Na especificação textual aSideML é descrito o aspecto *Persistence_in_DB* e sua interface transversal *IPersistence_in_DB*. Na especificação textual aSideML podemos visualizar o aspecto sendo constituído por um conjunto de *operations* (*Connect_DB*, *Disconnect_DB*, *Initiate_DB*, *Verify_if_DB_is_connected*, dentre outras) que são mapeadas para um conjunto de *features* do tipo *Operation* no modelo Metaspin. Similarmente, o *TemplateParameter template1* de aSideML é representado como um *SelectorAdviceBinding* que irá armazenar um conjunto de *join points*. A *Redefinition redefinition1* ilustrada na interface transversal de aSideML, a qual o aspecto *Persistence_in_DB* faz referência, irá compor o *advice* do aspecto. O seu elemento adorno *after* é representado como um *AdviceEvaluator* no modelo Metaspin e, o elemento

Operation Authentication_by_login da *Redefinition redefinifion1* é mapeado para o elemento *AdviceAction* no modelo Metaspin que corresponde a ação executada no *advice*.

Textual aSideML	Modelo Metaspin

Aspect Persistence_in_DB = {
Interface: IPersistence_in_DB;
Attributes: ;
Operations: self, Name, Connect_DB,
Disconnect_DB,
Persistence_in_Oracle,
Persistence_in_MySQL,
Persistence_in_Microsoft_Access,
Verify_if_DB_is_connected, Initiate_DB;
Tags: 'elementType = task,
contribution = [and, Persistence],
correlations = [null, null], topics = [DB]';
TemplateParameter template1 = {
parameter parameter1 = {
Interface: IPersistence_in_DB;
Operations: _Set_DB_;
}
}
};
CrosscuttingInterface IPersistence_in_DB = {
Redefinitions = {
Redefinition redefinition1 = {
Interface: IAuthentication;
Adorno: after;
Operation: Authentication_by_Login;
Operation Tags: elementType = task,
contribution = [and, Authentication],
correlations = [null, null], topics = [null];
}
}
(...)
	<aspects location="46:3-57:7"
	name="Persistence_in_DB">
	<**features** xsi:type="Operation"
	location="571:4-571:19"
	name="Connect_DB" />
	<**features** xsi:type="Operation"
	location="572:5-572:15"
	name="Disconnect_DB"/>
	<**features** xsi:type="Operation"
	location="572:5-572:11"
	name="Initiate_DB"/>
	(…demais operações)
	<**selectorAdviceBinding**>
	<**joinPointSelector** xsi:type="PrimitiveSelector“
	location="51:4-56:7"
	name="parameter1">
	<**joinPoint** xsi: type="ClassJoinPointPart"
	classe="ConFigurability"/>
	<**predicate**
	xsi:type="PrimitiveStructuralPredicate">
	<**pointcutLanguageEvaluator**
	name="call"/>
	<**predicate**>
	</**joinPointSelector**>
	<**advice**>
	<**adviceAction**
	xsi:type="EvalJoinPointInstructionAction"
	name="Authentication_by_Login">
	parent = ‘From aSideML – Redefinition”
	<**adviceEvaluator** name="after"/>
	</**adviceAction**>
	</**advice**>
	</**selectorAdviceBinding**>
	</**aspects**>

Figura 49 - Mapeamento entre aSideML e Metaspin

A Figura 50 ilustra a descrição textual em aSideML e o modelo *CoreElement* gerado. Na especificação textual aSideML é descrito o relacionamento transversal *Persistence_in_DB_Connec2_1* e os elementos base que participam deste relacionamento, que são as classes *Usability* e *ConFigurability*.

Na descrição aSideML, o código fonte afetado é representado pela classe *ConFigurability*, que está representada no elemento *JoinPoint* pertencente ao elemento *TemplateMatch match1*. Por se tratar de uma classe, esse *join point* é representado como um *ClassJoinPointPart* que consiste numa especialização do elemento *JoinPoint* no modelo *CoreElement*. Adicionalmente, a classe *ConFigurability* e seu conjunto de attributes e *operations* são mapeados para a abstração *Class e* para *features* do tipo *Attribute* e *Operation* no *CoreElement*, respectivamente.

Textual aSideML	Modelo CoreElement

RepresentationCrosscutting
Persistence_in_DB_Conec2_1 = {
Aspect: Persistence_in_DB;
CrosscuttingElement: IPersistence_in_DB;
BaseElement: Usability;
TemplateMatch = {
match match1 = {
JoinPoint: ConFigurability;
Ornament: around;
}
}
Tags: (…)
}
-- Classes de aSideML
Class Usability = {
Interface : IUsability**;**
Attributes :;
Operations :;
Tags : 'elementType = softgoal,
contribution = [and,null],
correlations = [null,null], topics = [null]'**;}**
Interface IUsability = {
Operations = {
ConFigurability, Set_DB
}
};
	<aspects location="46:3-57:7"
	name="Persistence_in_DB">
	(…)
	<**joinPointSelector** xsi:type="PrimitiveSelector“
	location="51:4-56:7"
	name="parameter1">
	<**joinPoint** xsi: type="ClassJoinPointPart"
	classe="ConFigurability"/>
	<**predicate**
	xsi:type="PrimitiveStructuralPredicate">
	<**pointcutLanguageEvaluator**
	name="call"/>
	<**predicate**>
	</**joinPointSelector**> (…)
	</**aspects**>
	<**classes** location="364:3-369:3" name="Usability">
	<**features** xsi:type="Operation"
	location="571:4-571:21"
	name="ConFigurability" isInherited="true"
	parentName=“Usability"/>
	<**features** xsi:type="Operation"
	location="572:5-572:27"
	name="Set_DB" isInherited="true"
	parentName=“Usability"/>
	
	(...)
	</**classes**>

Figura 50 - Mapeamento entre aSideML e *CoreElement*

A Figura 51 ilustra o resultado do mapeamento entre o modelo Metaspin e a geração do código OA em AspectLua. O aspecto *Persistence_in_DB* é mapeado diretamente para a declaração de um aspecto em AspectLua. O conjunto de *features* do tipo *Operation* pertencentes a esse aspecto no modelo Metaspin são mapeadas para *functions* no arquivo que contém a declaração do mesmo. O elemento Selector Advice Binding irá dar origem a declaração do *pointcut*, *join points* e *advice*. O *Join Point Selector* **é** mapeado para o *pointcut* do aspecto, e seu elemento *Join Point Configurability* do tipo *ClassJoinPointPart* é mapeado para o campo *list* na declaração do aspecto no código AspectLua. Por fim, os elementos do *advice* irão compor os parâmetros *type* e *action* na declaração do aspecto em AspectLua. O *EvalJoinPointInstructionAction Authentication_by_login* que corresponde a uma especialização de um *Advice Action* é mapeado para o campo *action* e, o seu *Advice Evaluator* é mapeado para o campo *type* na declaração do aspecto AspectLua. Em virtude de essa operação ter vindo de uma *Redefinition* de aSideML, ela é anotada com o comentário “*From aSideML – Redefinition*”, de forma a facilitar a rastreabilidade entre o elemento no nível de projeto detalhado e de implementação. O código aspectual na linguagem AspectLua é gerado em um arquivo denominado aspecto.lua, onde são declarados todos os aspectos implementados para uma dada aplicação. Foi utilizado essa padronização de forma a facilitar o *weaving* do código em um segundo momento, mas dependendo da estratégia que se deseje

utilizar é possível gerar um arquivo para cada aspecto através da reconFiguração do script Acceleo.

Modelo Metaspin	Código Aspectual em AspectLua

<aspects location="46:3-57:7"
name="Persistence_in_DB">
  <**features** xsi:type="Operation"
  location="571:4-571:19"
  name="Connect_DB" />
  <**features** xsi:type="Operation"
  location="572:5-572:15"
  name="Disconnect_DB"/>
  <**features** xsi:type="Operation"
  location="572:5-572:11"
  name="Initiate_DB"/>
(…demais operações)
<**selectorAdviceBinding**>
<**joinPointSelector** xsi:type="PrimitiveSelector“
  location="51:4-56:7"
  name="parameter1">
<**joinPoint** xsi: type="ClassJoinPointPart"
  classe="ConFigurability"/>
<**predicate**
xsi:type="PrimitiveStructuralPredicate">
<**pointcutLanguageEvaluator**
name="call"/>
<**predicate**>
</**joinPointSelector**>
<**advice**>
<**adviceAction**
xsi:type="EvalJoinPointInstructionAction"
  name="Authentication_by_Login">
  parent = ‘From aSideML – Redefinition”
<**adviceEvaluator** name="after"/>
</**adviceAction**>
</**advice**>
</**selectorAdviceBinding**>
</**aspects**>
	< - Arquivo do aspecto
	
	asp = Aspect: new()
	asp: aspect({name = 'Persistence_in_DB'},
	{name= 'pointcut-Persistence_in_DB',
	designator = 'call', list = {ConFigurability.*}},
	{type ='after', action = Authentication_by_Login}
	)
	
	-- From aSideML - Redefinition
	function Authentication_by_Login()
	  -- insert code here
	end
	
	-- funções do aspecto
	function Persistence_in_DB ()
	-- insert code here
	end
	
	function Connect_DB()
	-- insert code here
	end
	
	function Disconnect_DB ()
	-- insert code here
	end
	
	(demais funções)

Figura 51 - Mapeamento entre Metaspin e AspectLua

Na Figura 52 temos o exemplo do aspecto *Persistence_in_DB* expresso no Metaspin sendo mapeado para um aspecto em CaesarJ. O conjunto de *features* do tipo *Operation* pertencentes a esse aspecto no modelo Metaspin são mapeadas para métodos nas classes Java. O elemento Selector Advice Binding é mapeado para a declaração do *pointcut*, *join points* e *advice* do aspecto *Persistence_in_DB*.

O *Join Point Selector* é mapeado para o pointcut do aspecto. O elemento e seu elemento *ClassJoinPointPart Configurability* é mapeado para uma chama de *join point* dentro da declaração do *pointcut* a que ele pertence. O caractere ‘*” é utilizado como forma de representar que a chamada a qualquer operação da classe ConFigurability será capturada pelo aspecto. Os elementos do *Advice* são mapeados diretamente para a declaração do *advice* do aspecto. O *AdviceAction* é mapeado para um método que é chamado dentro da declaração do *advice* do aspecto. Essa operação deverá posteriormente implementada e parametrizada de

acordo com a necessidade do desenvolvedor. Além disso, essa operação é anotada com o comentário "*From aSideML – Redefinition*", visto ser proveniente de uma *Redefinition* no nível de projeto detalhado, de forma a facilitar a rastreabilidade entre o elemento no nível de projeto detalhado e de implementação. O *AdviceType* define quando a ação do *advice* deve ser executada e pode ser representado no código CaesarJ pela palavras reservadas *before*, *after* e *around* utilizadas na declaração do *advice.*

Modelo Metaspin	Código CaesarJ

<aspects location="46:3-57:7"
name="Persistence_in_DB">
  <**features** xsi:type="Operation"
  location="571:4-571:19"
  name="Connect_DB" />
  <**features** xsi:type="Operation"
  location="572:5-572:15"
  name="Disconnect_DB"/>
  <**features** xsi:type="Operation"
  location="572:5-572:11"
  name="Initiate_DB"/>
(…demais operações)
<**selectorAdviceBinding**>
<**joinPointSelector** xsi:type="PrimitiveSelector"
  location="51:4-56:7"
  name="parameter1">
<**joinPoint** xsi: type="ClassJoinPointPart"
  classe="ConFigurability"/>
<**predicate**
xsi:type="PrimitiveStructuralPredicate">
<**pointcutLanguageEvaluator**
name="call"/>
<**predicate**>
</**joinPointSelector**>
<**advice**>
<**adviceAction**
xsi:type="EvalJoinPointInstructionAction"
  name="Authentication_by_Login">
  parent = 'From aSideML – Redefinition"
<**adviceEvaluator** name="after"/>
</**adviceAction**>
</**advice**>
</**selectorAdviceBinding**>
</**aspects**>
	< - Arquivo do aspecto
	public aspect Persistence_in_DB {
	
	pointcut Persistence_in_DB :
	  call (**public void** Configurability.*)
	
	after : Persistence_in_DB(/*parameters*/) {
	  Authentication_by_login(/*parameters*/)
	}
	
	}
	
	/* From aSideML – Redefinition */
	public void Authentication_by_Login(/*paraketers*/) {
	  /* generated by marisa-aocode */
	  /* insert code here */
	}
	
	/* aspect methods */
	public void Persistence_in_DB() {
	  /* generated by marisa-aocode */
	  /* insert code here */
	}
	
	public void Connect_DB(){
	  /* generated by marisa-aocode */
	  /* insert code here */
	}
	
	public void Disconnect_DB() {
	  /* generated by marisa-aocode */
	  /* insert code here */
	}
	
	(demais métodos)

Figura 52 - mapeamento entre Metaspin e CaesarJ

As Figuras 53 e 54 ilustram o resultado do mapeamento entre o modelo *CoreElement* e a geração do código base em Lua e Java, respectivamente. As classes geradas através do mapeamento da especificação textual aSideML para o modelo *CoreElement* possuem mapeamento direto para a representação de uma classe tanto em Lua, quanto em Java. Para cada classe mapeamos seu nome, seu conjunto de *features* do tipo Operation e Attribute para um conjunto de *functions* e *variables* dentro das classes lua, respectivamente. Em virtude de AspectLua não ter uma abstração correspondente à interface, as interfaces provenientes de um

modelo *CoreElement* têm seu conjunto de assinaturas de operações mapeadas para *functions* dentro de uma classe Lua, tendo sua propriedade *isInherited* com valor igual a *true.*

As classes do CoreElement são transformadas em classes Java e as suas *features* do tipo *Attribute* e *Operation* são mapeados para atributos e métodos, respectivamente. Os atributos possuem visibilidade do tipo privada (*private*) e para cada atributo são criados métodos *get* e *set*. As interfaces do CoreElement são mapeadas para interfaces Java e são declaradas com a palavra reservada ***implements*** nas classes que as referenciam. Adicionalmente, o conjunto de métodos de uma interface Java gerada no processo de transformação de MaRiSA-AOCode são inseridas nas classes que implementam essas interfaces.

Modelo CoreElement	Código Base em Lua
<**classes** location="364:3-369:3" name="Usability"> <**features** xsi:type="Operation" location="571:4-571:21" name="ConFigurability" isInherited="true" parentName="Usability"/> <**features** xsi:type="Operation" location="572:5-572:27" name="Set_DB" isInherited="true" parentName="Usability"/> </**classes**> <**classes** location="364:3-369:9" name="ConFigurability"> <**features** xsi:type="Operation" location="571:4-571:21" name="self" isInherited="true" parentName="ConFigurability "/> </**classes**>	-- Arquivo do código base -- Classes Usability = {} **function** Usability: conFigurability() -- insert code here **end** **function** Usability: Set_DB () -- insert code here **end** ConFigurability = {} **function** ConFigurability: self) -- insert code here **end**

Figura 53 - Mapeamento de CoreElement para código Lua

Modelo CoreElement	Código Base em Java

<**classes** location="364:3-369:3" name="Usability">
<**features** xsi:type="Operation"
location="571:4-571:21"
name="ConFigurability" isInherited="true"
parentName="Usability"/>
<**features** xsi:type="Operation"
location="572:5-572:27"
name="Set_DB" isInherited="true"
parentName="Usability"/>
</**classes**>
	public class Usability **implements** IUsability {
	
	public Usability () {
	
	}
	
	public void ConFigurability() {
	/* generated by marisa-aocode */
	/* insert code here */
	}
	
	public void Set_DB () {
	/* generated by marisa-aocode */
	/* insert code here */
	}
	}

Figura 54 - Mapeamento de CoreElement para código Java

4.6 MaRISA-AOCode

Essa seção tem por objetivo ilustrar a utilização da ferramenta MaRiSA-AOCode. MaRiSA-AOCode é uma abordagem para transformação entre modelos, segundo regras de mapeamento definidas no capítulo 3 deste documento, que provê a integração entre as fases de projeto detalhado e codificação permitindo a geração de código orientado a aspectos para mais de uma linguagem POA. As Figuras 55 a 60 ilustram os projetos que compõem o ambiente MaRiSA-AOCode. Os projetos são organizados da seguinte forma.

- Projeto de metamodelos: aSideML-metamodel, metamodel-Metaspin, metamodel-AspectLua e metamodel-CaesarJ.
- Projeto de transformação model-2-text: aSideML2Metaspin, Metaspin2AspectLua e Metaspin2CaesarJ.
- Projetos de tranformação text-2-model: AspectLua-Model2Text e CaesarJ-Model2Text.

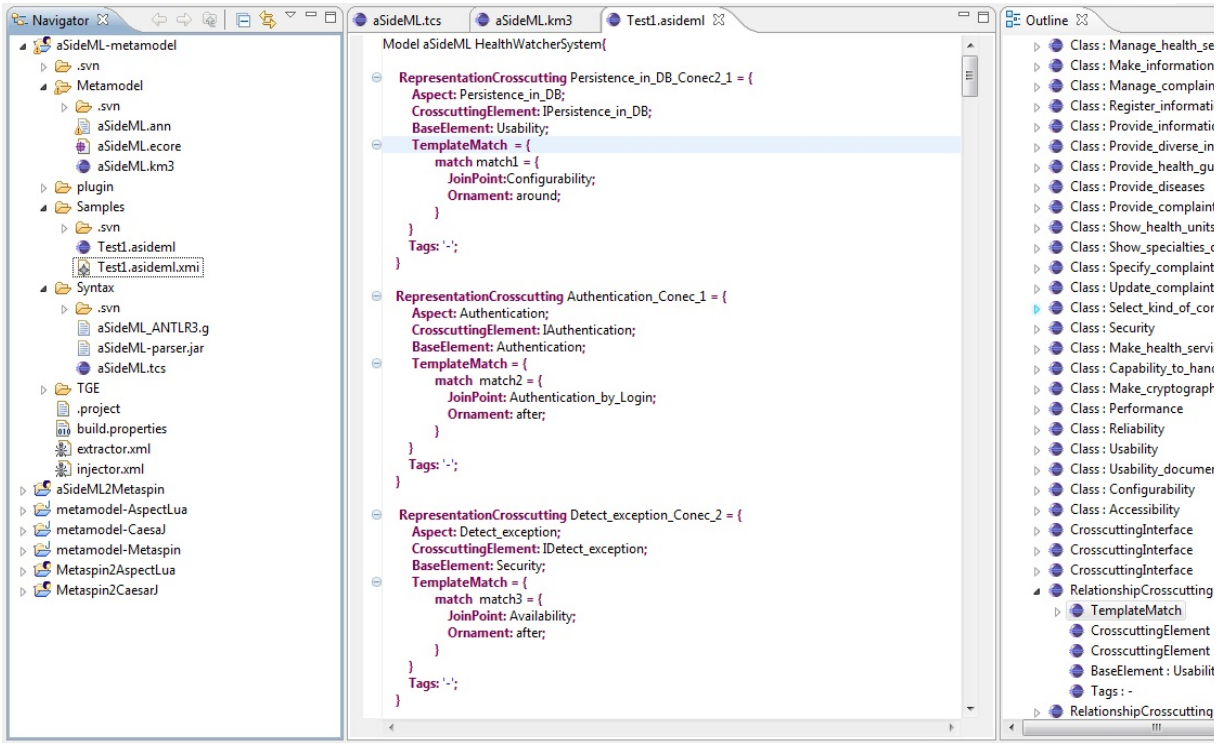

Figura 55 - Modelo textual aSideML

A Figura 55 ilustra o projeto aSideML-metamodel onde é realizada a descrição textual aSideML de acordo com a sintaxe textual descrita em um arquivo aSideML.tcs.Sempre que novos elementos são adicinados em uma descrição textual, esses elementos são verificados

pelo TCS para garantir que o relacionamento entre eles está em conformidade com o metamodelo da linguagem.

Uma vez definida uma descrição textual bem formada, a descrição textual é transformada em um modelo aSideML (ver Figura 56), o qual é representado em forma de árvore através do editor da IDE Eclipse. Uma vez que alteramos uma descrição textual ela deve ser injetada novamente de forma a garantir que o modelo de entrada (o qual servirá de entrada para todo o processo de transformação de MaRiSA-AOCode) esta está em conformidade com o metamodelo aSideML. Isso ajuda a garantir que estejamos sempre trabalhando com modelos bem formados evitando erros na execução das transformações.

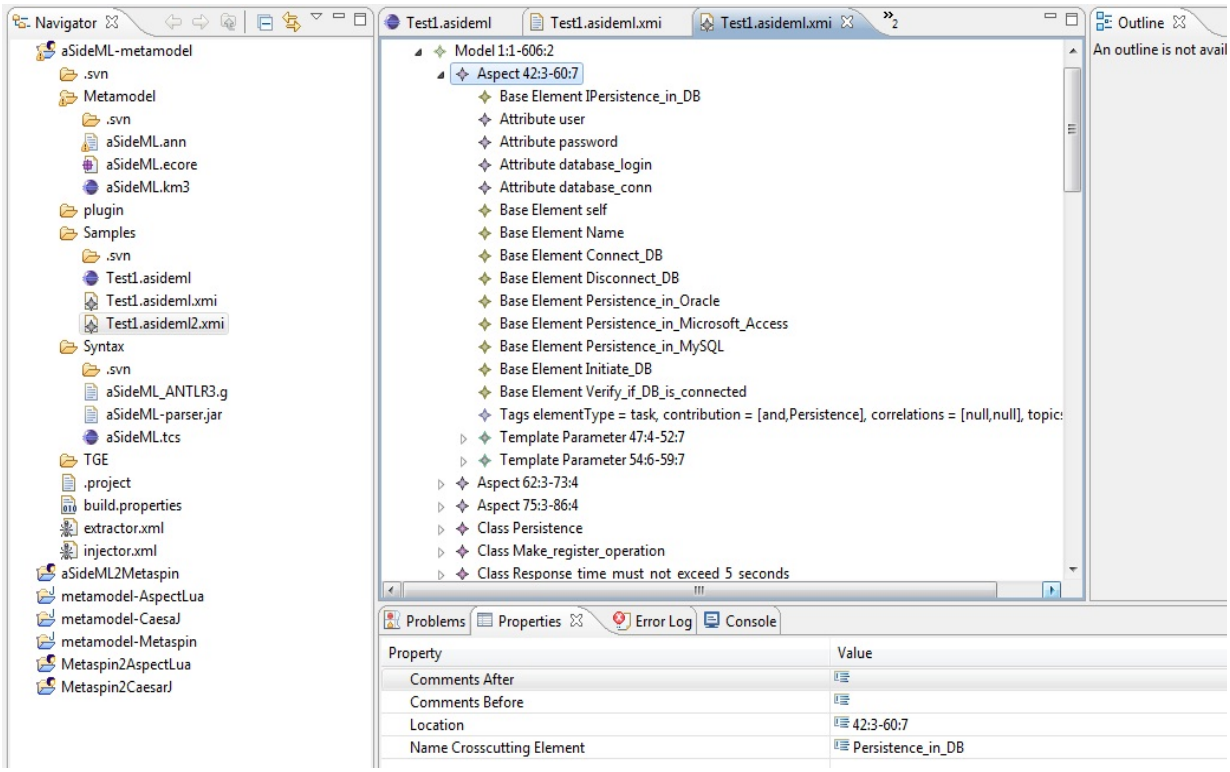

Figura 56 - modelo aSideML injetado a partir da descrição textual

A Figura 57 ilustra o projeto aSideML2Metaspin, responsável por realizar a transformação do modelo aSideML bem formado, proveniente da Fase 1 do processo de transformação, o qual será transformado em modelos Metaspin e CoreElement. Esse projeto é composto basicamente por 3 diretórios: (i) lib, onde são implementados os ATL *Helpers* utilizados para auxiliar na navegação e busca de informações em elementos do modelo aSideML; (ii) modelos, onde são armazenados os modelos de entrada aSideML que serão consumidos pelas regras de transformação especificadas nesse projeto; e (iii) transformações, contém os

arquivos ATL responsáveis por realizar a transformação de aSideML para Metaspin (*aSideML2Metaspin*.atl) e para CoreElement (*aSideML2CoreElement*.atl).

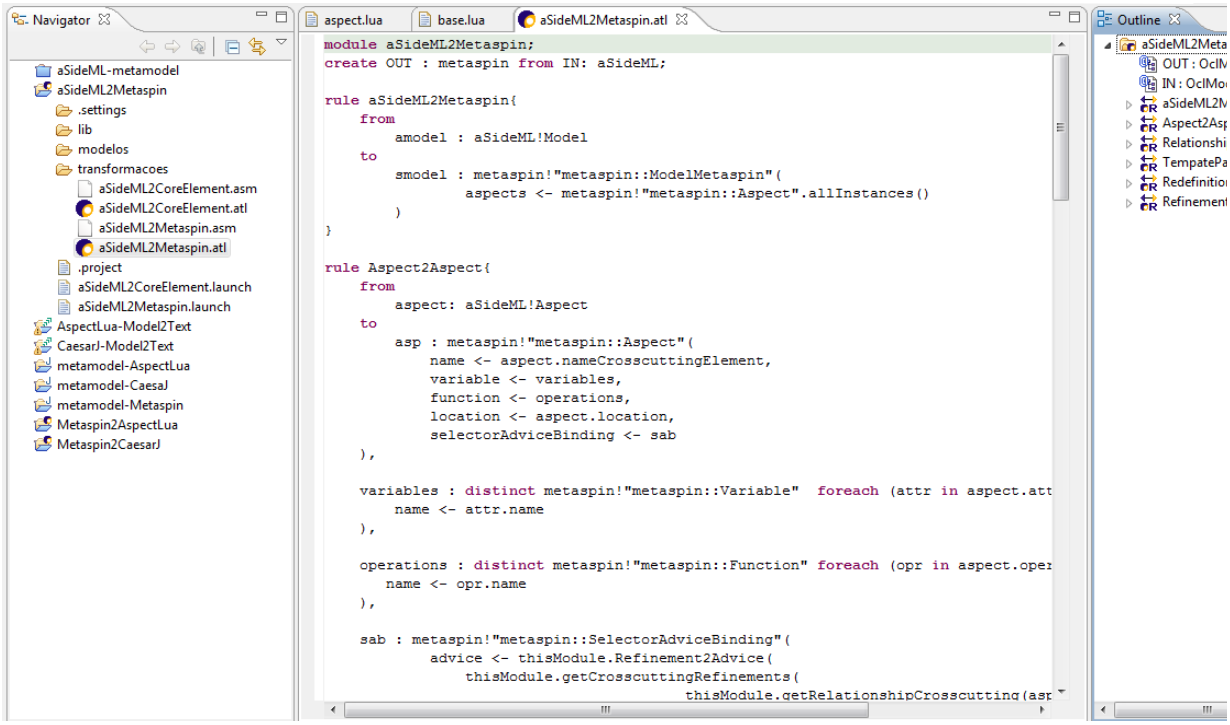

Figura 57 - projeto de transformação de aSideML para Metaspin

Nas Figuras 58 e 59 ilustramos os projetos Metaspin2AspectLua e Metaspin2CaesarJ, responsáveis por transformar os modelos aSideML e CoreElement em modelos AspectLua e CaesarJ. Esses projetos seguem a mesma estrutura de diretórios que o projeto aSideML2Metaspin. Em ambos os casos, os modelos Aspectlua e CaesarJ gerados a partir de modelos Metaspin e CoreElement bem formados, são verificados e validados com seus respectivos metamodelos de forma a garantir que os modelos de saída também serão bem formados e portanto podem ser utilizados na fase seguinte, onde serão processados por templates de geração de código.

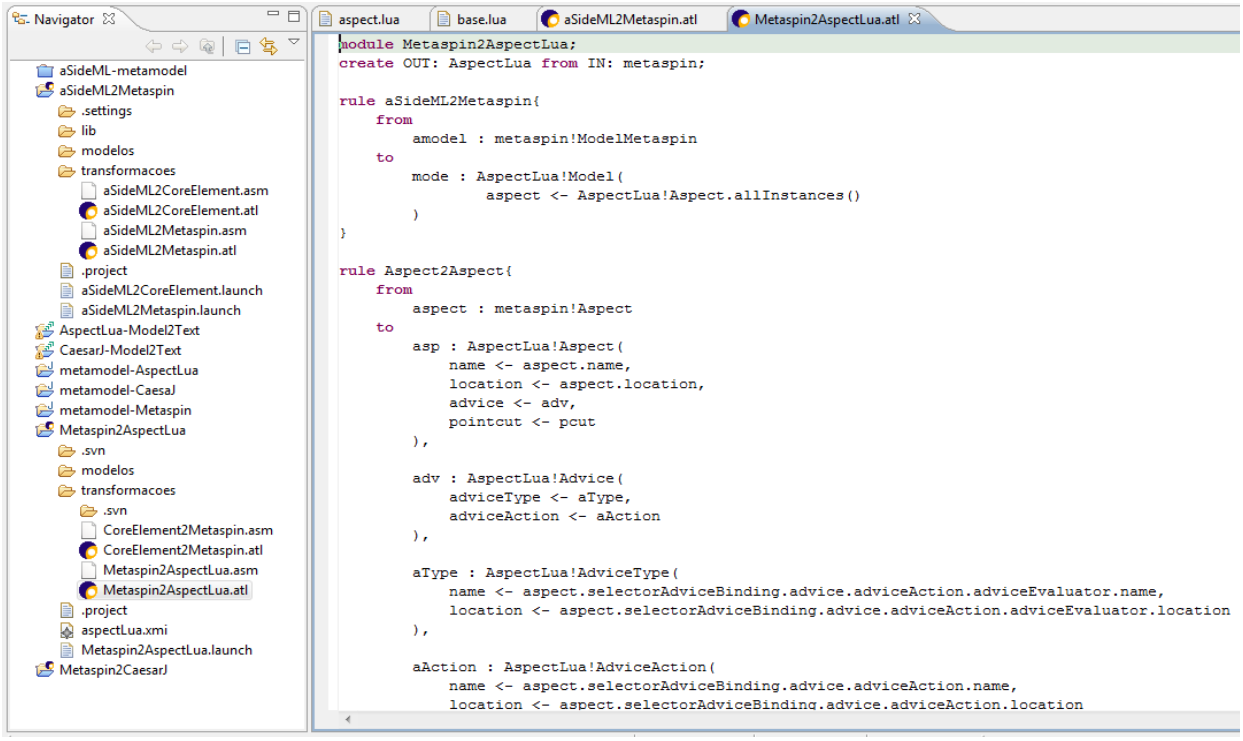

Figura 58 - projeto de transformação de Metaspin para AspectLua

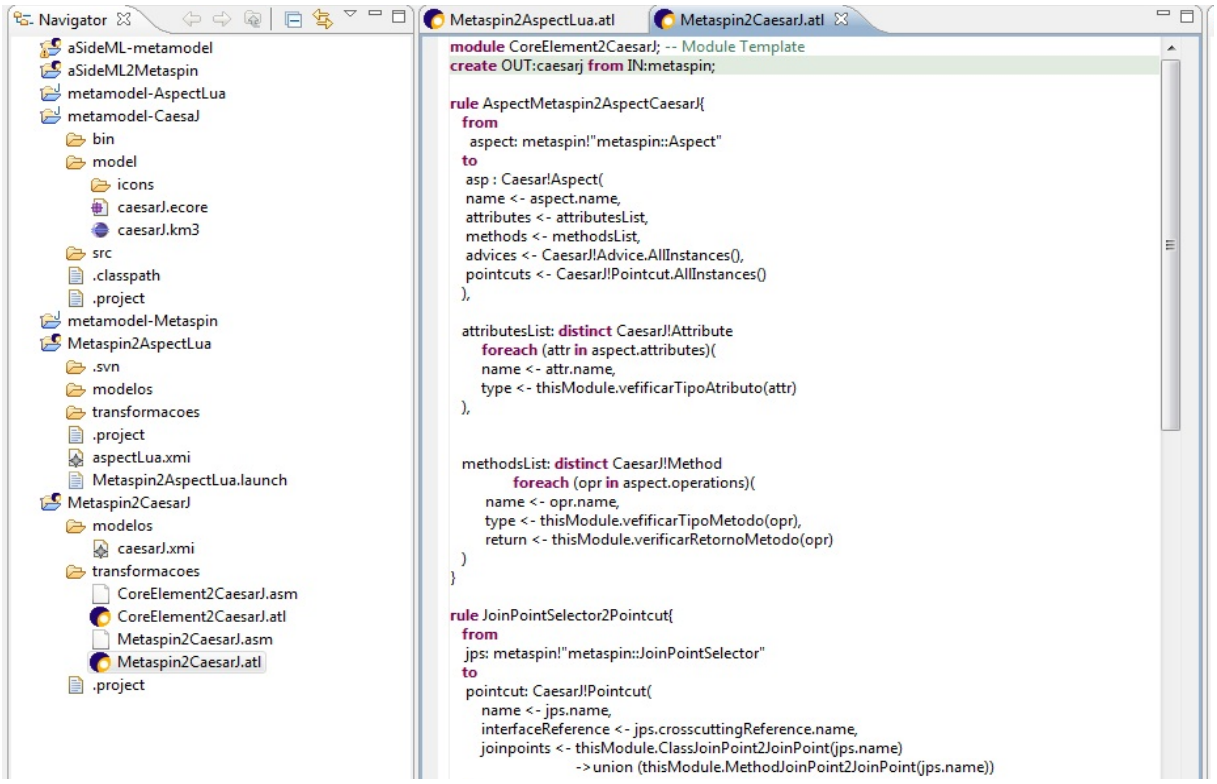

Figura 59 - projeto de transformação de Metaspin para CaesarJ

Para a geração automatizada de código OA a partir dos modelos AspectLua e CaesarJ foram criados projetos Acceleo para cada uma dessas transformações. Na Figura 60 ilustramos o exemplo do projeto AspectLua-Model2Text. Os principais arquivos para

transformação definidos são: (i) AspectLua.mt e Lua.mt, os quais são templates responsáveis por processar os modelos AspectLua e transformá-los nos arquivos aspect.lua e base.lua, respectivamente. Nesses templates é possivel definir como cada elemento do modelo de entrada será representado textualmente na linguagem em que estamos trabalhando. Além disso, é possível dizer em que pacote queremos que esses códigos sejam gerados; (ii) aspectlua.chain, o qual é um script que contém quais arquivos de transformação devem ser executados e qual a sequência deles.

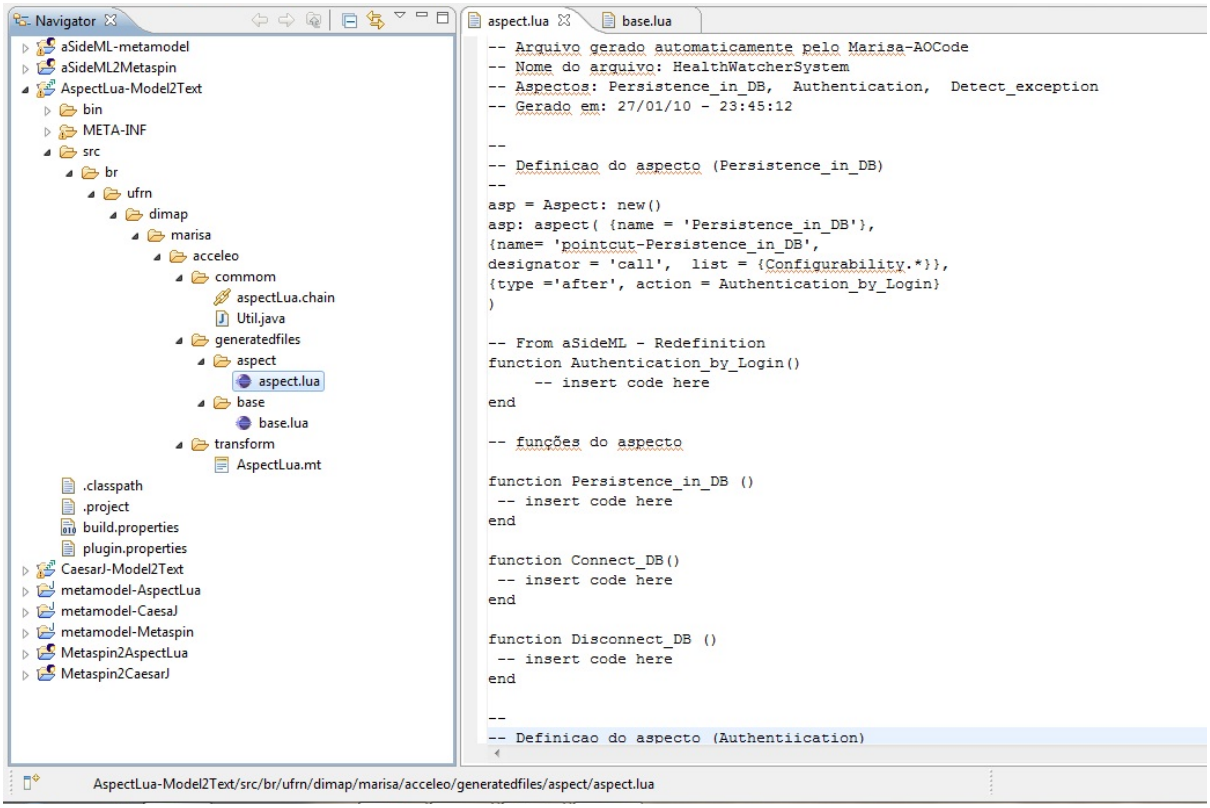

Figura 60 - código AspectLua gerado pela ferramenta

Adicionalmente, no arquivo aspectlua.chain (ver Figura 60) são definidos os templates de geração de código aspectlua.mt e base.mt que devem ser executados para gerar os arquivos aspect.lua (código aspectual) e base.lua (código base), respectivamente.

4.7 Análise dos modelos obtidos de MaRiSA-AOCode

Nesta seção relatamos a experiência envolvida na transformação entre o modelo de projeto detalhado (aSideML) e o modelo abstrato de programação (Metaspin), e a transformação deste último para modelos específicos de plataforma (AspectLua e CaesarJ), realizando uma análise qualitativa que considera os seguintes parâmetros:

- Completude: dado um processo de transformação entre modelosm, esse critério visa considerar se existem elementos em um modelo de entrada que não foram mapeados para um elementos correspondentes no modelos de saída.
- Rastreabilidade: nesse critério avaliamos se o modelo de saída do processo de transformação representa todos os elementos especificados no modelo entrada, e se é possível identificar quais elementos do modelo de entrada originaram um elemento específico no modelo saída.
- Corretude: esse critério avalia se os elementos do(s) modelo(s) de saída foram corretamente gerados através do processo de mapeamento, não apresentando erros sintáticos de acordo com os templates definidos para cada linguagem .

4.7.1 Avaliação do modelo abstrato de programação gerado a partir do projeto detalhado

a) Completude: os modelos usados como entrada do processo de transformação de MaRiSA-AOCode vêm da abordagem de MaRiSA-DP [Montenegro, 2008]. Embora aSideML permita a modelagem estrutural, comportamental e composicional de aspectos, através do modelo de projeto detalhado gerado por MaRiSA-DP, consideramos somente a modelagem estrutural. Nesse contexto, as regras de mapeamento utilizam os principais elementos estruturais de aSideML: *Aspect*, *Base Element*, *Crosscutting Interface*, *Template Match*, *Join Point*, *Template Parameter*, *Redefinition*, *Refinement*, *Additions*, *Uses*, *Operation* e *Tags*. No entanto, existem elementos de aSideML que não tem correspondente direto no Metaspin. Por exemplo, o elemento Crosscutting Relationship não tem correspondência, porém seus elementos possuem correspondentes diretos espalhados pelo modelo Metaspin. Dessa forma, a completude entre os modelos ocorre através do mapeamento direto ou indireto entre os modelos. Outro exemplo relacionado é o valor do elemento *PointcutLanguageEvaluator* no contido no *Predicate* do *JoinPointSelector* do aspecto no Metaspin. A linguagem aSideML define essa informação somente em modelos comportamentais, os quais não são tratados em nossa abordagem. Para os modelos comportamentais adota-se como valor padrão para essa informação a primitiva *execution*. No entanto, por tratarmos somente da modelagem estrutural de aSideML em nossa abordagem, decidimos por utilizar como valor padrão para esta informação a primitiva *call* visto ser o tipo de chamada de interceptação de *join points* mais comum entre as linguagens de programação OA. Podemos considerar que houve completude no mapeamento visto que os elementos

estruturais de aSideML foram mapeados direta ou indiretamente para um elemento no Metaspin.

b) Rastreabilidade: nesse critério avaliamos se o modelo Metaspin representa todos os elementos especificados no modelo aSideML, e se é possível identificar quais elementos de aSideML originaram um elemento específico no modelo Metaspin. Em virtude de os elementos estruturais serem mapeados para um elemento correspondente no Metaspin, podemos afirmar que as informações contidas em um modelo aSideML são todas salvas. Como forma de permitir a visualização do relacionamento origem/fonte de cada elemento gerado no modelo Metaspin são utilizados comentários em grande parte dos elementos, como é o caso de operações pertencentes a classes e interfaces, bem como propriedades do aspecto que são provenientes de relacionamentos transversais e interfaces transversais de aSideML. Adicionalmente, nossa abordagem permite a propagação de informações desde a modelagem de requisitos em AOV-Graph para descrição arquitetural em AspectualACME (definido em MaRiSA), e descrição arquitetural em AspectualACME para o projeto detalhado em aSideML (definido em MaRiSA-DP). Por exemplo, as propriedades de AspectualACME, as quais mantêm informações adicionais de requisitos que não são representados por elementos de primeira classe, são mapeados para Tags aSideML, que por sua vez, são armazenadas em forma de comentário no modelo Metaspin.

4.7.2 Avaliação dos modelos específicos de plataforma gerados a partir do modelo abstrato de programação

a) Completude: avaliamos nesse critérios se existem elementos do Metaspin que não foram mapeados para AspectLua e/ou CaesarJ. O Metaspin é um metamodelo que visa atender a diversas linguagens de programação orientadas a aspectos. Dessa forma, os elementos atendem a diversos paradigmas de programação. Para o nosso mapeamento, foram escolhidos elementos que atendessem a necessidade do mapeamento para a linguagem AspectLua, que embora sejam uma linguagem de *scripting* possui mecanismos que permitem simular a orientação a objetos, e a linguagem CaesarJ, que tem como linguagem o Java, uma linguagem orientada objetos.

Tabela 8 - quantificação dos elementos do metaspin

Metamodelo	nº elementos existentes	nº elementos utilizados
Pointcut	**14**	**9**
Join Point	**14**	**6**
Advice	**11**	**6**
Advice Binding	**8**	**6**
TOTAL	**45**	**25**

Na tabela 8 quantificamos os elementos existentes no Metaspin organizando-os de acordo com cada sub-metamodelo. No metamodelo de *Pointcut* foram utilizados 9 elementos dentre os 14 existentes, são eles: *Pointcut Language Evaluator*, *Base Language*, *Predicate*, *Primitive Selector, Join Point Selector* e *Custom Predicate*. No metamodelo de Join Point utilizamos apenas 6 elementos dos 14 disponíveis, visto que tratamos somente *join* points estáticos em nossa abordagem, são eles: *Join Point*, *Structural Join Point Part*, *Object Oriented Join Point Part*, *Class Join Point Part*, *Method Join Point Part* e *Statement Join Point Part*. O restante dos elementos desse metamodelo corresponde a elementos utilizados para representação de *join points* dinâmicos, os quais não são tratados nesse trabalho. No metamodelo de *Advice* foram utilizados 6 elementos dentre os 11 existentes, são eles: *Advice*, *Advice Action*, *Meta-level Action, EvalJoinPointInstructio, Primitive Base Action*, *Advice Evaluator*. Os elementos utilizados representam a contento a abstração de advice para AspectLua e CaesarJ. Por fim, o metamodelo *Advice Binding* foram utilizados 4 elementos dos 6 existentes, são eles: *Aspect* , *Variable*, *Selector Advice Binding* e *Intertype Declaration*. Dos 45 elementos existentes de todo o Metaspin, para o tratamento de informações estruturais de aSideML e geração de código nas linguagens CaesarJ e AspectLua foram utilizados 25 elementos.

O Metaspin possui um grande número de elementos de forma a ser genérico o suficiente para abranger os conceitos relacionados à orientação a aspectos de diversas linguagens de POA. No entanto, para o escopo de nosso trabalho onde serão utilizadas duas linguagens de programação OA não foi necessário a utilização de todos os elementos presentes no metamodelo.

Além disso, foram utilizados somente elementos estruturais, visto que neste trabalho tratamos somente a modelagem estrutural de aspectos descrita em aSideML. Nesse sentido, não foram utilizados elementos que representam características comportamentais de aspectos, nem tampouco elementos que tratam *join points* dinâmicos. Dessa forma, podemos dizer que

não houve completude no mapeamento dos elementos dos modelos, embora os elementos utilizados do Metaspin tenham sido suficientes para representar tanto as abstrações de orientação a aspectos em um nível abstrato de programação, quanto em um nível específico de plataforma (AspectLua e CaesarJ). Neste caso, conclui-se que a utilização de um número reduzido de elementos do Metaspin deve-se a abrangência que este metamodelo tem, bem como pelo fato de estar sendo utilizado em nosso trabalho em um contexto específico. Ou seja, embora o Metaspin disponibilize uma grande diversidade de elementos no seu metamodelo, foram utilizados somente alguns elementos, os quais foram suficientes para a representação de aspectos e a geração de código para AspectLua e CaesarJ.

b) Rastreabilidade: os elementos dos modelos AspectLua e CaesarJ gerados pelas regras de transformação são em sua grande maioria anotados com comentários de forma a permitir sabem qual elemento do projeto detalhado deu origem a determinado elemento tanto no modelo Metaspin, quanto no modelo AspectLua e/ou CaesarJ onde essas informações são propagadas. Um exemplo disso são operações de classes e interfaces, operações advindas de redefinições e refinamentos que são anotadas em nível de modelos e em nível de código. Além disso, é realizada a propagação de informações desde a fase de requisitos que chegam até o nível de Tags em aSideML. Essas Tags são também armazenadas em forma de comentário nas operações a que elas pertencem. Dessa forma existe a rastreabilidade de informações desde a fase de projeto detalhado aSideML, sendo estas propagadas no nível abstrato de programação do Metaspin, bem como no nível específico de programação de AspectLua e CaesarJ.

c) Corretude: avaliamos se os elementos dos modelos de saída (AspectLua e CaesarJ) foram corretamente gerados através do processo de mapeamento descrito em MaRiSA-AOCode. Embora os aspectos e código base sejam gerados automaticamente, é necessário a intervenção do programador para a definição da lógica de negócio a ser implementada, visto que o código gerado é proveniente de modelos estruturais de projeto detalhado e não comportamentais. Além disso, MaRiSA-AOCode garante a corretude sintática do código gerado, em virtude de a geração de código ser feita através de templates pré-definidos baseado na sintaxe textual tanto de AspectLua, quanto de CaesarJ para os respectivos códigos. Como exemplo, dentre as 31 classes e interfaces geradas a partir do modelo base do Health Watcher, nenhuma delas apresentou problemas de sintaxe.

5 Trabalhos Relacionados

Este capítulo apresenta algumas abordagens atuais para a integração das fases de desenvolvimento de software orientado a aspectos, que fazem uso de técnicas MDD (i.e. transformações entre modelos). Além de ressaltar as contribuições de cada abordagem em relação a integração desses dois paradigmas, realizamos uma breve análise comparativa dessas abordagens em relação a apresentada em nosso trabalho.

5.1 Cross-MDA – a model driven approach for aspect management

O trabalho desenvolvido em [Alves et al, 2008] apresenta o CrossMDA, um framework que lida com a separação horizontal e vertical de características. O CrossMDA permite o tratamento de aspectos no nível de modelagem e provê mecanismos que permitem a separação de interesses tanto sobre a dimensão horizontal, entre modelos do mesmo nível de abstração, bem como a separação de interesses na dimensão vertical, entre modelos de diferentes níveis de abstração. A separação de interesses na dimensão horizontal é obtida através da adoção de um processo que modela aspectos independentemente de elementos de negócio no nível PIM. O modelo PIM do aspecto é uma representação abstrata de um interesse transversal particular, permitindo esconder detalhes de implementação do desenvolvedor de negócio e elevando o nível de abstração da modelagem no nível do PIM.

O CrossMDA incorpora um processo de transformação que visa integrar características transversais em sistemas dirigidos a modelos. Tal processo é realizado através da combinação de tecnicas existentes nas abordagen da Arquitetura Dirigida a Modelos (MDA), bem como na Programação Orientada a Aspectos (POA). Adicionalmente, o framework utiliza o conceito de separação horizontal de características para criar aspectos de forma independentes e modelos de negócio, integrando-os por meio de transformações MDD. O CrossMDA provê um processo de desenvolvimento e um conjunto de serviços e ferramental para dar suporte a tal processo.

Segundo [Alves et al, 2008], os objetivos do CrossMDA são: (i) elevar o nível de abstração na modelagem orientada a aspectos através do uso de modelos PIM de características transversais independentes de modelos de negócio; (ii) reusar artefatos de características transversais no nível de modelos PIM; (iii) automatizar o mapeamento de artefatos de transformação MDA relacionados a características transversais; (iv) facilitar o

reuso de artefatos de transformação MDA relacionados a características transversais; e (v) favorecer o reuso de modelos PIM de negócio.

O processo do CrossMDA pode ser dividido em 4 fases (ver Figura 61). A fase inicial ou fase 0, engloba duas diferentes visões de modelagem de artefatos: (i) modelagem de aspecto; e (ii) modelagem de negócio. O modelo de aspecto é uma representação abstrata de características transversais. Assim, as características transversais são modeladas como classes anotadas com o estereótipo <<aspect>> e organizadas em pacotes. No CrossMDA, um pacote aspecto é uma entidade que agrega aspectos relacionados, ou seja, aqueles que lidam com a mesma categoria de requisitos. De forma similar ao modelo de aspecto, o modelo de negócio também apresenta uma visão independente de plataforma, mas de processo de negócio.

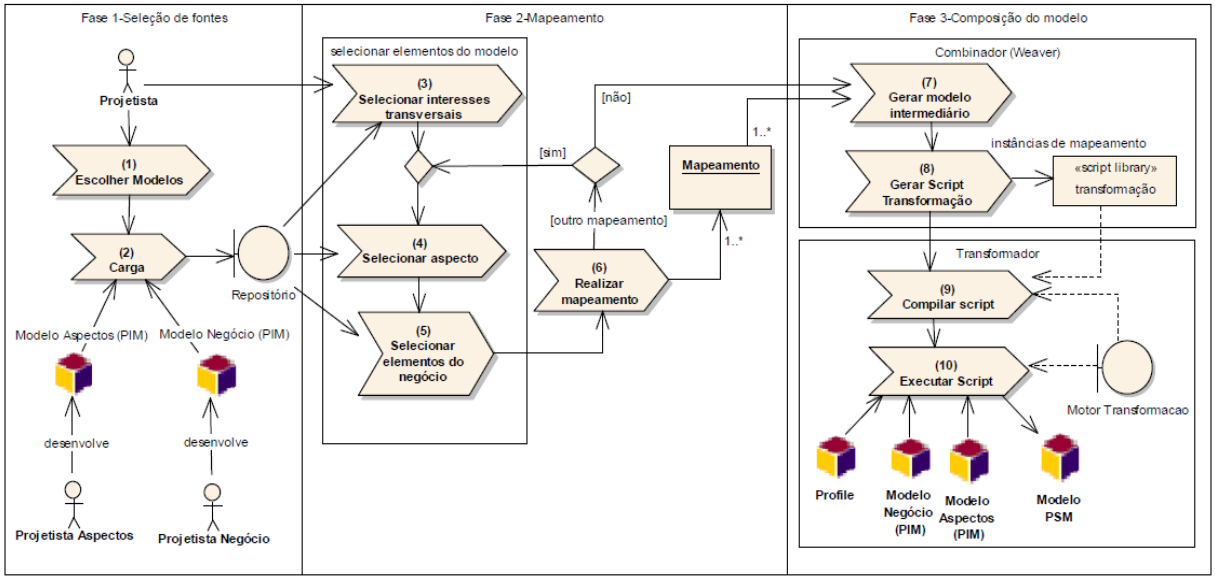

Figura 61 - Processo do CrossMDA

O processo do CrossMDA não impõe nenhuma restrição a modelagem de negócio. Logo, o modelo de negócio pode ser composto por qualquer elemento válido da UML para a modelagem de entidades de negócio e seus relacionamentos. Os modelos de aspecto e de negócio podem ser desenvolvidos independentemente por dois diferentes atores: o arquiteto do aspecto e o arquiteto de negócio, respectivamente. Os modelos construídos nessa fase são armazenados em um repositório para posterior utilização nas próximas fases.

A Fase 1 é realizada pelo arquiteto do sistema de forma a aumentar um dado modelo de negócio com as características transversais necessárias para endereçar os requisitos do sistema. Essa fase é composta de duas atividades: (i) seleção de modelos; e (ii) carregamento do modelo. A seleção do modelo consiste na seleção dos modelos fontes do aspecto e do modelo PIM de negócio que serão usados no processo de transformação. A atividade de

carregamento do modelo é responsável por carregar e persistir os modelos selecionados em um repositório de metadados.

Na Fase 2 o arquiteto do sistema inicia o mapeamento que consiste na especificação dos relacionamentos entre aspectos e elementos do modelo de negócio. Essa fase se inicia com a seleção de pacotes de características transversais que são relevantes para o domínio da aplicação ser modelado. A seguir é realizado um processo iterativo de definição de relacionamentos entre aspectos e elementos de negócio, no qual o arquiteto do sistema seleciona os aspectos que devem ser aplicados a um conjunto de elementos de negócio. Quando esse processo iterativo acaba, o CrossMDA gera um conjunto de mapeamentos que representam as definições de *pointcuts* e declarações intertipos geradas de acordo com relacionamentos entre aspectos e elementos de negócio especificados através do arquiteto do sistema.

Na fase 3 é gerado o modelo PSM, o qual representa o refinamento do modelo fonte de negócio adicionados com aspectos.Essa fase é composta 4 atividades organizada em dois sub-processos, a saber: (i) *weaving* de modelo; e (ii) geração de modelo.

O sub-processo de *weaving* de modelo inicia com a geração de um modelo intermediário de um conjunto de mapeamentos produzidos como saída da fase 2. Tal modelo intermediário é uma representação que contém cada instância de classe de aspecto e suas respectivas dependências para os elementos do modelo de negócio. O modelo intermediário é então transformado em uma especificação formal através da geração de um programa de transformação baseado na especificação OMG MOF QVT (Query/View/Transformation) [OMG/QVT, 09]. Esse programa de transformação é então compilado e executado no sub-processo de geração do modelo gerando como saída o modelo PSM. Adicionalmente, o CrossMDA provê diversos serviços: (i) persistência de modelos; (ii) mapeamento de modelos; (iii) combinador; e (iv) transformador de modelos.

A persistência de modelos é responsável por implementar as operações básicas de forma a permitir o carregamento e a persistência de modelos, bem como as operações de navegação, recuperação e instanciação de novos elementos do modelo existente. A realização da persistência é realizada por um serviço de repositório para a persistência de metamodelos. Essa tarefa é realizada por um serviço de repositório para persistência de metadados.

O mapeamento de modelos provê mecanismos para o gerenciamento do mapeamento dos relacionamentos entre os aspectos e os elementos de negócio, a qual consiste em uma atividade chave do processo do CrossMDA. Os mapeamentos suportados pelo CrossMDA

são: (i) pontos de atuação, que seguem o padrão de especificação dos pontos de junção da abordagem POA e da linguagem AspectJ, e (ii) declaração intertipos.

O combinador responsável por integrar os modelos de aspectos ao modelo de negócio gerando instâncias dos aspectos selecionados e as associações destas com os elementos de negócio.

Por fim, o transformador de modelos, essa atividade é iniciada quando um programa de transformação, gerado pelo combinador, necessita ser compilado e executado. O CrossMDA fornece um serviço para compilar e executar o programa de transformação e dessa forma gerar o novo modelo, através do motor de transformação de ATL, que inclui uma máquina virtual (ATLvm) e um compilador.

5.2 Aspect-Orientation from Design to Code

Este trabalho [Groher e Baumgarth, 2004] tem como foco a separação de características transversais na fase de projeto arquitetural oferecendo a AML (*Architecture Modeling Language*), uma notação para modelagem de projeto arquitetural que segue o padrão UML. Ou seja, a abordagem oferece a separação de características transversais em uma arquitetura de alto nível, apresentando uma abordagem para a modelagem orientada a aspectos e a geração automatizada de código OA.

Com a notação, artefatos transversais são claramente encapsulados e completamente isolados da lógica de negócio para forçar o seu reuso. A notação fornece uma separação clara de linguagem dependente de orientação a aspectos de partes independentes de orientação a aspectos simplifica o suporte a um numero diferente de linguagens OA e conceitos dos aspectos e da lógica de negócio.

Para estender o suporte, além da fase de arquitetura o gerador de código apresentado endereça o suporte ao projeto de baixo nível oferecendo um mapeamento automatizado de modelos de projeto para modelos de programação para prevenir inconsistências entre o projeto e a implementação. Adicionalmente, como linguagem para geração de código foi utilizado o AspectJ, sendo utilizado como critério de seleção o grau de maturidade da linguagem.

A abordagem utiliza a UML como linguagem de modelagem. Quando se usa UML para a modelagem orientada a aspectos, os desenvolvedores realizam a modelagem usando ferramentas e ambiente familiares para ganhar todos os benefícios que são usados no projeto orientado a objetos.

Adicionalmente, a abordagem visa trazer benefícios relacionados tanto ao reuso do código gerado quanto do projeto de software OA, incluindo a habilidade para reusar aspetos e elementos base separadamente. Dessa forma, os aspectos e elementos base são completamente separado e independente de tecnologia de implementação.

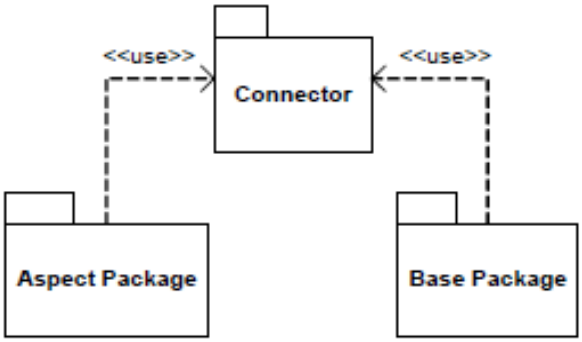

Figura 62 - (De) composição em nível de pacote

A linguagem proposta na abordagem, a AML (*Architecture Modeling Language*), considera o fato de conceitos transversais tenderem a afetar múltiplas classes em um sistema. Desde que um conceito transversal pode ser representado por várias classes e todas essas classes podem estar associadas com a classe que tal conceito entrecorta, o módulo de construção de conceito deve ser de mais alto nível que uma classe. Na Figura 62 é ilustrada uma visão geral da notação e seu foco na decomposição em nível de pacotes.

A AML inclui um pacote base (contendo lógica de negócio), um pacote aspecto (contendo o conceito transversal) e um *connector* para ligar os aspectos aos elementos base. Essa separação provida pela AML permite uma alta reusabilidade do aspecto e elementos base visto que o *connector* é o único elemento transversal. O suporte diferentes tecnologias OA é, portanto, bastante simples e direta já que apenas a sintaxe do *connector* deve ser mudada.

- O pacote aspecto provê uma representação gráfica da visão estática de um conceito transversal em particular e é, junto com o pacote base, uma das partes OO do modelo OA.
- O pacote base contém a lógica de negócio do sistema e pode ser modelado considerando qualquer interesse transversal que potencialmente pode afetar o sistema. Similar ao pacote aspecto, o pacote base pode contem qualquer modelo UML valido que descreve lógica de negócio do sistema desejado.

Na AML não existe relacionamento direto entre o pacote aspecto e o pacote base. O relacionamento é definido somente através de conector de pacote contendo regras para a posterior recomposição de aspectos e elementos base. Toda a semântica de conector apresentada na abordagem foi desenvolvida de acordo com o modelo de conexão de AspectJ. Na Figura 62 é ilustrado o pacote *connector*, que contém as classes que estão em conformidade com os conceitos oferecidos por AspectJ para a especificação de regras de *weaving*:

i. *Introduction*, que define regras para o mecanismo de *introduction*.
ii. *Pointcut*, que define os pontos de execução no fluxo de controle do programa.
iii. *Advice*, que define o código a ser executado nos *pointcuts* definidos na classe *Pointcut.*

Na AML, todas as classes podem conter operações com semânticas especiais para especificar como o aspecto e os elementos base devem ser recompostos. Adicionalmente, o gerador de código realiza validação de corretude sintaxe e semântica antes de realizar a geração automática do código a partir de um modelo de programação. O desenvolvimento desse gerador é dividido em duas partes:

- Validação de modelo: valida os modelos de projeto OA quanto à corretude sintática e semântica.
- Geração de código: gera o código fonte AspectJ para os modelos OA validados.

Para o desenvolvimento do gerador de código foi utilizada a ferramenta CASE Together. O Together possui uma API aberta que oferece uma grande quantidade de conceitos para a manipulação de modelos UML. A ferramenta valida e gera automaticamente as partes OO do modelo gerado (aspectos e elementos base), a validação e geração de código das partes OA é implementada como módulos que são plugados dentro da plataforma Together. Os elementos aspecto e os elementos base são mapeados para código fonte Java. O pacote aspecto e o pacote base são partes OO da notação. Os elementos *connector* são mapeados para código fonte AspectJ. O pacote *connector* consiste na parte OA da notação, ligando o pacote aspecto e o pacote base. Como forma de garantir a corretude sintática e semântica os arquivos AspectJ que podem ser compilados com um compilador AspectJ mapeando mapeiam regras que foram definidas entre a notação e os conceitos AspectJ.

5.3 A Meta-Level Specification and Profile for AspectJ in UML.

Este Trabalho [Evermann, 2007] apresenta um metamodelo para a linguagem AspectJ, utilizando Profiles como mecanismo de extensão da UML. Ademais, utiliza uma modelagem baseada no padrão XMI permitindo o uso do Profile em ferramentas CASE disponíveis comercialmente e suporte a geração de código. Com a utilização de mecanismos de extensão UML, o metamodelo resultante é também um Profile UML para suportar a modelagem de AspectJ na UML.

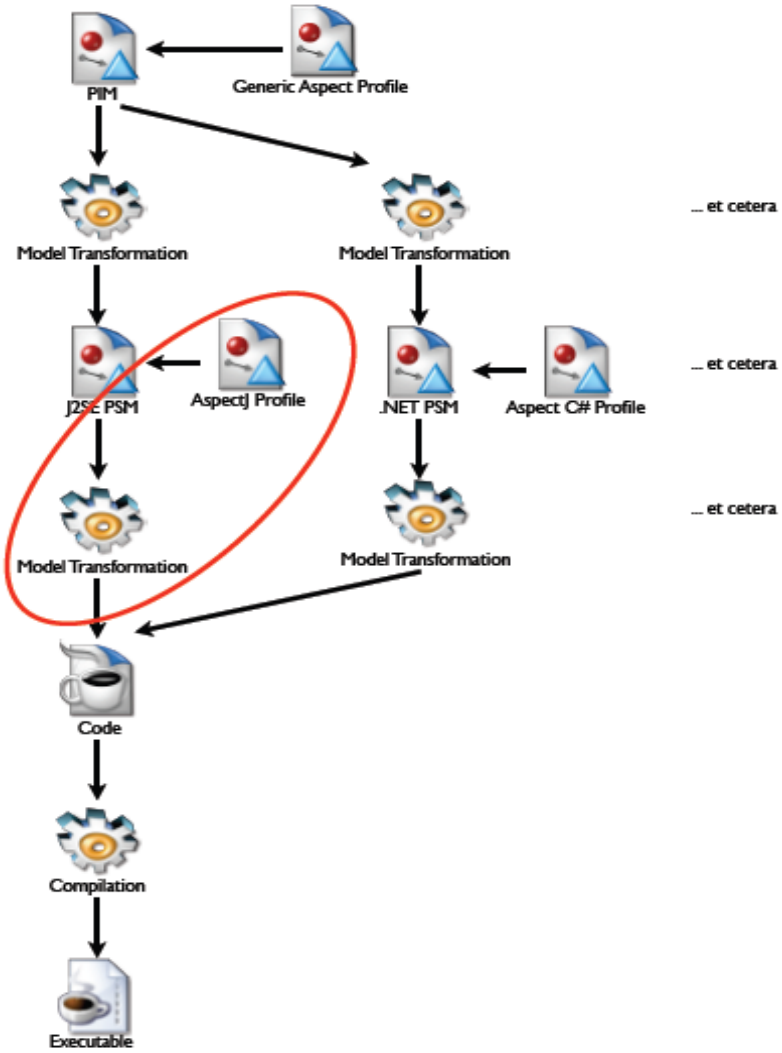

Figura 63 - Uso do Profile com desenvolvimento baseado em MDA [Evermann, 2007

A proposta está situada na parte destacada da Figura 63 dentro do contexto da arquitetura MDA. O perfil utilizado permite a especificação de modelo específicos de plataforma, ou seja, específico para a plataforma Java e AspectJ, bem como a geração de

código fonte a partir de tais modelos. O Profile apresentado no trabalho não é uma extensão genérica de modelagem orientada a aspectos, pois as diferenças conceituais entre diferentes implementações de aspectos são substanciais e não podem ser capturadas em um único modelo. Profiles genéricos podem ser utilizados para a especificação de um modelo independente de plataforma (PIM), o qual serve de base para o modelo específico de plataforma (PSM) na arquitetura MDA.

O trabalho apresenta uma técnica para a criação de modelos orientados a aspectos que são convertidos em código OA, o qual pode ser combinado (*woven*) por um compilador orientado a aspectos. O *weaving* do código gerado é feito em nível de código. O foco do *weaving* em nível de código ao invés do *weaving* em nível de modelo é útil para o desenvolvimento de projetos onde o comportamento do sistema não é especificado em diagramas comportamentais UML

Dentre os pontos positivos dessa abordagem estão:

- Suporte a ferramentas de compatíveis com UML 2.0. A extensão não requer suporte de software especial e permite que a modelagem de aspectos seja usada em ferramentas de software maduras.
- A técnica proposta é suportada por modelos de intercâmbio UML XMI, permitindo a utilização de diferentes modelos MOF em conformidade com ferramentas de modelagem UML.
- A proposta do trabalho permite que todos os conceitos relacionados a aspectos sejam especificados em termos de metamodelo, não havendo necessidade da especificação de palavras chaves.
- Mantém a separação estrita do modelo base e de características transversais nos modelos em que ela é aplicada.

Adicionalmente, o trabalho apresenta um metamodelo com os conceitos de AspectJ. Esse metamodelo é modelado no meta-nível da UML, por meio da utilização de Profiles. Ao invés de especializar *metaclasses* UML, é utilizada a extensão de metaclasses já existentes.

Um estereótipo UML é definido como uma *metaclass* que entra em um relacionamento do tipo *extends* com *metaclasses* existentes. Visualmente ele é mostrado como as classes estendidas. Esse mecanismo de extensão na UML 2.0 é, portanto, uma forma eficiente na qual qualquer modelo de meta-nível imediatamente torna-se usável como Profile.

Em virtude de o modelo estar em conformidade com o formato padrão UML XMI e ser completamente especificado em termos de metamodelo, o código pode ser facilmente

gerado. Na proposta do trabalho foi implementado um XTSL para a geração de uma código AspectJ válido. As transformações especificadas têm um alto grau de complexidade decorrente da tentativa de garantia da robustez que a abordagem visa prover. A geração de código atualmente depende de o projetista desenvolver modelos que são representações validas AspectJ.

O trabalho apresenta dois pontos fortes, um sob a perspectiva teórica e outro sob a perspectiva prática:

- Na perspectiva teórica, o ponto forte da proposta é uma especificação completa de AspectJ na UML. O modelo especifica completamente AspectJ em termos do metamodelo da UML e não lida com descrições textuais ou anotações que devem ser processadas para aplicação do modelo ou verificação.
- Na perspectiva prática, os aspectos podem ser trocados usando o mecanismo de intercâmbio de modelos UML XMI e aplicados a novos modelos UML existentes. A forma modular na qual a UML 2.0 e o MOF permitem que Profiles sejam trocados e aplicados significa que extensões do modelo AspectJ podem ser aplicados em modelos UML existentes, apenas como extensões AspectJ que podem ser combinadas em software Java existentes.

5.4 Towards Executable Aspect-Oriented UML models

Este trabalho [Fuentes e Sánchez, 2007] tem por objetivo obter modelos OA executáveis, utilizando extensões de semânticas de ação da UML. A abordagem usa padrões conhecidos e amplamente utilizados (como a UML), de forma a obter soluções abertas e prevenir a necessidade de aprender novas linguagens e notações. A abordagem define um processo para a construção de modelos UML executáveis. O trabalho apresenta duas principais contribuições, nos quais baseia o processo de sua abordagem:

1. Um Profile UML 2.0 para a especificação de modelos executáveis, que são chamados de AOEM (*Aspect-Oriented Executable Modeling*). O Profile é utilizado para a modelagem comportamental OA, através da extensão das Semânticas de Ação (*Action Semantics*) da UML.
2. Um modelo de *Weaver* para os modelos OA que está em conformidade com o Profile AOEM. O mecanismo de *weaving* permite adicionar automaticamente aspectos aos módulos do projeto que ele entrecorta.

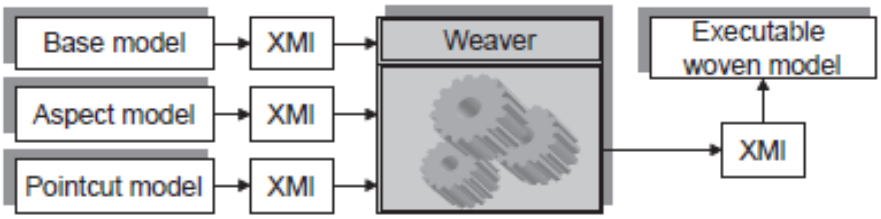

Figura 64 - Cenário de modelos OA Executáveis

Utilizando esses dois elementos o processo é definido (ver Figura 64) de acordo com as seguintes atividades:

- Construção de um modelo UML executável para a representação de características não-transversais, ou seja, construção do *base model*.
- As características transversais, incluindo seu comportamento, são modeladas como aspectos usando o Profile AOEM. Isso produz o *aspect model*.
- A forma como as características transversais devem ser compostas com outras características é especificado através do *pointcut model*.
- Os modelos base e de aspecto são compostos, o que irá produzir o *woven model*. Esse modelo é um modelo comum UML executável.
- Para executar o modelo OA, o *woven model* é importado em uma ferramenta UML com capacidades de execução.

Como dito anteriormente, a abordagem utiliza extensões de semânticas de ação (*Action Semantics*) da UML. *Action Semantics* são procedimentos, onde uma ação é a unidade fundamental de especificação de um comportamento que consome um conjunto de entrada e converte em um conjunto de saídas. De forma a prevenir o uso de notações que trabalham especificamente com ferramentas proprietárias, foi desenvolvido um Profile UML, para especificação de um conjunto de ações compatíveis com qualquer ferramenta que suporte diagrama de atividades e ações abstratas. O Profile trabalha da seguinte forma:

- *Procedures* são representadas através de diagramas de atividades.
- Ações são nós de diagramas de atividades.
- As entradas e saídas são representados como *pins*.
- Para distinguir cada ação especifica, elas são estereotipadas com seus respectivos nomes.
- Deve ter o mesmo numero de *pins* de entrada/saída especificados, o que é garantido através de restrições OCL.

Para permitir que o *weaving* e a importação do *woven model* em alguma ferramenta UML com capacidades de execução, os modelos UML deve estar disponíveis no formato XMI (XML *Metadata Interchange*) [OMG/XMI, 2009] padrão. O XMI permite serializar um modelo UML num documento XML, que pode ser facilmente manipulado. O *Weaver* de modelos proposto na abordagem utiliza representações XMI do modelo base, modelo de aspecto e modelo de *pointcut* e produz como saída uma representação XMI de um *weaving model* do sistema. Por fim, essa representação XMI do sistema poderá executada por qualquer ferramenta que suporte diagrama de atividades UML de forma.

5.5 *Comparação*

Como forma de estabelecer um comparativo entre as diferentes abordagens relacionadas mostradas anteriormente foram categorizadas as principais informações envolvidas em tais trabalhos visando permitir avaliar cada abordagem.

Tabela 9 - Comparativo entre as abordagens de trabalhos relacionados

Abordagem	*Utiliza metamodelos*	*Automatização do processo de transformação*	*Integração dos artefatos das fases de desenvolvimento de software*	*Geração de código OA*	*Direcionada a MDD*	*Direcionada a DSOA*
Aspect-Orientation from Design to Code [Groher e Baumgarth, 2006]	*Sim*	*Sim*	*Sim*	*Sim* (*AspectJ)*	*Não*	*Sim*
A Meta-Level Specification and Profile for AspectJ in UML [Evermann, 2007]	*Sim*	*Sim*	*Sim*	*Sim* (AspectJ)	*Sim*	*Sim*
Towards Executable Aspect-Oriented UML models [Fuentes e Sánchez, 2007]	*Não*	*Sim*	*Não. Somente artefatos em nível de projeto detalhado.*	*Não*	*Sim*	*Sim*
CrossMDA [Alves et al, 2007]	*Sim*	*Sim*	*Não. Somente artefatos em nível de projeto detalhado.*	*Não*	*Sim*	*Sim*
Transformações entre modelos Orientados a aspectos do projeto detalhado a codificação	*Sim*	*Sim*	*Sim. Suporte artefatos das fases de projeto detalhado e modelos específico de plataforma para geração de codigo OA.*	*Sim (AspectLua e CaesarJ)*	*Sim*	*Sim*

Tais informações estão organizadas na tabela 9, são elas: (i) utilização de metamodelos; (ii) automatização do processo de transformação; (iii) transformação entre fases finais do processo de desenvolvimento de software; (iv) geração de código orientado a

aspectos; (v) se a abordagem é direcionada para o desenvolvimento dirigido a modelos; e (vi) se a abordagem é direcionada para o desenvolvimento de software orientado a aspectos.

O CrossMDA [Alves, 2007] é um arcabouço para dar suporte à integração entre modelagem com aspectos e MDA. Em CrossMDA, aspectos são modelados explicitamente apenas no nível de projeto, enquanto que MaRiSA-AOCode trabalha tanto no nível de projeto detalhado quanto na geração de código OA. Além disso, o CrossMDA adota um modelo de aspectos simples, porém baseado em "Profiles" de UML, para prover uma representação abstrata e independente de plataforma (PIM) e usa outra notação, AODM (Stein et al, 2002), para descrição de modelos PSM. Nossa abordagem permite a propagação de informações vindas desde a fase de requisitos através da integração com outras ferramentas que vem sendo desenvolvidas no projeto mais amplo no qual este faz parte. Ou seja, além de permitir a modelagem em nível de projeto detalhado e a geração automatizada de código para mais de uma linguagem de programação OA, ela armazena informações de modelos desde a fase de requisitos e arquitetura, os quais são tratados em MaRiSA [Medeiros, 2007] e MaRiSA-DP [Montenegro, 2008]. Tanto o CrossMDA e MaRiSA- AOCode utilizam templates ATL para dar suporte às transformações. Porém, CrossMDA dá apoio à transformação apenas entre modelos no nível de projeto. Adicionalmente, ambas as abordagens definem um processo de transformação entre modelos.

O trabalho definido em [Groher e Baumgarth, 2007] foca na modelagem de alto nível da arquitetura realizada por meio da AML, permitindo a partir de uma especificação a geração de código fonte para Java e AspectJ. Em nossa abordagem, a modelagem é realizada em um nível de projeto detalhado independente de plataforma, onde através de um mapeamento intermediário no nível abstrato de programação permitindo o mapeamento para diferentes linguagens de programação, sendo neste caso para Java e CaesarJ, e para Lua e AspectLua. Para a geração de código, nossa abordagem realizada a separação do modelo base (parte OO) e dos modelos de aspectos (parte OA) a partir da descrição de projeto detalhado em aSideML, no nível independente de plataforma. Uma vez gerados os modelos OO e OA é realizado o mapeamento e geração de código para uma ou mais linguagens de programação específicas de plataforma. Além disso, a abordagem de [Groher e Baumgarth, 2007] utiliza a CASE *tool* Together [Together, 2009], a qual é extensível através de API Java, para o gerador de código. Em nossa abordagem utilizamos regras de mapeamento e templates baseado em modelos, o que garante uma maior corretude e validação dos modelos gerados de acordo com os respectivos metamodelos de cada linguagem

Em [Evermann, 2007] foi definido um metamodelo completo para a linguagem AspectJ através do uso de Profile UML, onde a partir desse Profile é possível realizar a geração de código Java e AspectJ. O trabalho foca no nível de modelos específicos de plataforma dentro da arquitetura MDA. Além disso, com a utilização de Profiles UML se torna possível aplicá-lo em outras ferramentas que suportem UML. No entanto, O Profile apresentado no trabalho não é uma extensão genérica de modelagem orientada a aspectos, pois as diferenças conceituais entre diferentes implementações de aspectos são substanciais e não podem ser capturadas em um único modelo. Em nossa abordagem foi realizado a descrição dos metamodelos das linguagens utilizadas e descrito um mapeamento intermediário para dar suporte para a geração de código para mais de uma linguagem de programação a partir da definição de um projeto detalhado descrito em aSideML. Dessa forma, nossa abordagem define regras de transformação entre um modelo de projeto detalhado e um modelo abstrato de programação, permitindo a partir deste último a geração de código para mais de uma linguagem OA. A modelagem de nossa abordagem abrange os níveis PIM, PSM e CODE da arquitetura MDA, pois define a comunicação entre o projeto detalhado especificado em aSideML e a geração de código fonte em diferentes linguagens através do mapeamento intermediário suportado pelo Metaspin. Adicionalmente, enquanto na abordagem de [Evermann, 2007] foram definidas regras de mapeamento extensas descritas em XSTL com alto grau de complexidade tornando difícil seu reuso, nossa abordagem utilizou a linguagem ATL para a descrição de regras entre o projeto detalhado, o nível abstrato de programação e a linguagem específica de plataforma. A maioria das regras ATL podem ser reutilizadas em outras abordagens dirigidas a modelos que realizem o mapeamento linguagens de modelagem de projeto detalhado que estendam a UML para um nível abstrato de programação. A linguagem ATL facilita a descrição de regras de mapeamento entre modelos, pois fornece mecanismos (*rules*, *helpers*, *modules*) para a transformação entre modelos. Além disso, possui uma sintaxe amigável, facilitando o entendimento e reuso de regras de mapeamento definidas.

Em [Fuentes e Sanchez, 2007] é definida uma abordagem para a modelagem em nível de projeto detalhado com a utilização da UML e suas semânticas de ação, com o objetivo de produzir modelos OA executáveis que possam ser interpretados por ferramentas que tenha capacidade de executar tais modelos. O modelo final gerado nessa abordagem é um modelo UML executável, resultante da composição dos modelos *base*, *aspect* e *pointcut*. Para executar o modelo OA resultante é necessário importá-lo em alguma ferramenta que tenha suporte a capacidade de execução de modelos UML. Em nossa abordagem foi utilizado o

desenvolvimento baseado em metamodelos e modelos em formato Ecore, tendo sido desenvolvida dentro do ambiente Eclipse. A partir de modelos Ecore gerados é possível trabalhar com diferentes *plugins* suportados pelo Eclipse para o desenvolvimento de modelos. Adicionalmente, os modelos podem ser disponibilizados em formato XMI permitindo a interoperabilidade com outras ferramentas. Quanto a geração de código, a partir de regras de mapeamento bem definidas é possível realizar a geração de código para mais de uma plataforma e reutilizar parte dessas regras para dar suporte a outras linguagens de programação sem que seja necessário gerar código a partir de uma especificação de projeto detalhado independente de plataforma. Em ambas as abordagens é definido um processo de transformação de modelos, onde é possível a modelagem de projeto detalhado. No entanto, enquanto que a abordagem de [Fuentes e Sanchez, 2007] define a geração de modelos executáveis UML e realiza *weaving* em nível de modelos, nossa abordagem realiza a geração de código OA para mais de uma linguagem de programação e o *weaving* dos modelos gerados é realizado em nível de código, seja por compiladores especiais ou interpretadores.

6 Conclusão

De maneira geral, embora existam atualmente diversas abordagens orientadas a aspectos que associam diferentes atividades do processo de desenvolvimento, elas não possibilitam integração entre si. Além disso, os modelos associados a cada uma das atividades não estão naturalmente alinhados ou inseridos em um processo coerente. Nesse sentido, os desenvolvedores acabam por se sobrecarregar de atividades, visto que eles devem: (i) ter conhecimento de diversas abordagens OA, bem como de seus respectivos modelos e artefatos associados; (ii) ser capazes de compreender como aspectos são representados em cada uma das atividades; e (iii) definir a correspondência entre artefatos OA gerados em cada atividade. Adicionalmente, caso a correspondência entre modelos e artefatos OA associados às diversas atividades não estiver bem definida, pode haver perda de informações e/ou decisões importantes, ou mesmo introduzir erros de uma atividade para outra.

Como forma de endereçar tal problema, nesse trabalho foi proposta uma abordagem genérica que promove a integração entre o desenvolvimento orientado a aspectos (DSOA) e desenvolvimento baseado em modelos (MDD) onde foram definidos modelos OA de cada atividade do processo de desenvolvimento, bem como a correspondência entre eles. O trabalho foca nas atividades de projeto detalhado e codificação OA, utilizando as linguagens aSideML e, AspectLua e CaesarJ, respectivamente. Esse trabalho define um nível de programação abstrato como forma de permitir a geração de código OA para mais de uma linguagem de programação, através da definição de um conjunto de regras entre o nível de projeto detalhado e o nível abstrato de programação. Esse nível abstrato de programação é provido pelo Metaspin, um metamodelo que provê as principais abstrações para orientação a aspectos, sendo construído com base em estudos de diversas linguagens de programação OA, analisando suas diferenças e similaridades. . Além disso, definimos um processo baseado em modelos, onde foram especificadas regras de mapeamento entre os elementos de cada linguagem, permitindo a propagação automática de modelos entre uma atividade e outra.

Os modelos associados às atividades de projeto detalhado e codificação foram construídos com base em rigores semânticos especificados por linguagens para descrição de modelos. Dessa forma, foi utilizada a linguagem KM3 (*Kernel MetaMetaModel*) para descrição de metamodelos para aSideML, AspectLua e CaesarJ. Para a representação dos modelos aSideML para Metaspin utilizamos o componente TCS (*Textual Concret Syntax*). Além disso, para a especificação de regras de transformação utilizamos a linguagem de

transformação ATL. Essa decisão pela utilização dessas tecnologias da-se em virtude de já estarem sendo utilizadas em outros projetos em um contexto mais amplo no qual este também faz parte. As transformações foram implementadas na IDE Eclipse.

Através da geração automatizada é possível obter uma estrutura de código inicial referente a implementação dos elementos base e dos respectivos aspectos que os afetam. No entanto, em virtude de tratarmos somente a modelagem estrutural de aSideML, existe a necessidade de intervenção do programador de forma a inserir comportamento nos modelos gerados. Além disso, em virtude dos modelos aSideML utilizados serem advindos de informações desde fase de requisitos e arquitetura, alguns detalhes de tipo de propriedades ou atributos não são tratados. No entanto, em caso de definição manual de um modelo de projeto detalhado em aSideML, os tipos de atributos e retornos de métodos são tratados. Todavia, grande parte do trabalho do desenvolvedor é realizado automaticamente pela ferramenta, portanto, a função deles será ajustar e incluir comportamento nos códigos gerados, seja em AspectLua ou CaesarJ.

Adicionalmente, realizamos uma breve análise dos modelos gerados, bem como do código OA gerados nas linguagens AspectLua e CaesarJ. Nessa análise, tentamos mostrar a qualidade dos modelos quanto a completude e rastreabilidade, bem como a corretude sintática do código gerado. Além disso, pudemos verificar a abrangência do Metaspin em relação a diversas linguagens de programação. Em nossa abordagem, por se tratar de um contexto mais específico de utilização do Metaspin e pelo fato de tratarmos somente a modelagem estrutural de aspectos não foram utilizados todos os elemetnos disponíveis no Metaspin. No entanto, os elementos utilizados conseguiram de forma satisfatória representar as principais abstrações de orientação a aspectos em um nível abstrato de programação e, permitiram propagar as informações do projeto detalhado especificado em aSideML para linguagens de programação orientadas a aspectos especificas, AspectLua e CaesarJ. Alem disso, houve a necessidade da criação de um elemento para tratar casos de *Intertype Declaration*, visto que o metamodelo do Metaspin não trata essa característica de orientação a aspectos.

Com o ambiente MaRiSA-AOCode garante-se que as informações contidas na modelagem estrutural do projeto detalhado estejam presentes no código gerado em mais de uma linguagem de programação OA. A geração do código para ambas as linguagens é feita de duas formas (i) para o código AspectLua, são gerados dois arquivos, um contendo o código base da aplicação e um segundo arquivo contendo a parte correspondente ao código aspectual; e (ii) para o código em CaesarJ é gerado um projeto onde são geradas as classes Java em seus respectivos pacotes, cujos nomes são pré-definidos, bem como os aspectos em seus devidos

pacotes. O *weaving* entre os códigos aspectuais e códigos base gerados para cada linguagem de programação é feito de acordo com compiladores ou interpretadores específicos para cada uma delas. A ferramenta MARISA-AOCode, as transformações e os exemplos estão disponíveis em: http://www.ppgsc.ufrn.br/~everton/marisaaocode.

6.1 Contribuições

As contribuições desse trabalho incluem:

- Integração entre as atividades de requisitos, arquitetura, projeto detalhado e codificação orientada a aspectos, por meio de transformações entre o projeto detalhado para diferentes linguagens de programação OA, mantendo-se informações provenientes desde modelos de requisitos, arquitetura e projeto.
- Definição dos metamodelos de aSideML, AspectLua e CaesarJ em KM3 e Ecore, através da extensão dos conceitos de programação orientadas a aspectos providos pelo Metaspin, bem como estudar a sintaxe textual das linguagens de programação orientadas a aspectos utilizadas em nosso estudo.
- Especificação de regras de transformação entre aSideML, Metaspin, AspectLua e CaesarJ, de forma a expressar nas fases seguintes as informações e decisões contidas nas atividades predecessoras. Como exemplo, podemos citar as propriedades provenientes de um modelo arquitetural descrito em AspectualACME produzido pela ferramenta MaRiSA-DP [Medeiros et al, 2009] as quais são armazenadas em forma de *Tags* em aSideML. Estas por sua vez, são mapeadas como comentários de código, seja em AspectLua ou CaesarJ.
- Implementação de regras de transformação em ATL permitindo o processo automatizado de transformações entre os modelos de projeto detalhado, modelos abstratos de programação OA e modelos de linguagens de programação OA específicas.
- Construção de um ambiente integrado, MaRiSA-AOCode, que permita a criação de modelos em conformidade com os metamodelos de cada linguagem utilizada e regras de transformação entre seus elementos. Esse ambiente facilita o acesso a informações e decisões em nível de projeto detalhado e codificação OA, permitindo uma melhor visualização da propagação de informações entre essas fases. Adicionalmente, em virtude de este trabalho estar inserido em um contexto mais amplo, pode utilizar modelos provenientes de MaRiSA-MDD, tornando possível a visualização de

informações desde requisitos, arquitetura, projeto e codificação dentro de um mesmo ambiente de desenvolvimento.

- Validação das regras de mapeamento utilizando o estudo de caso *Health Watcher,* o qual é comumente utilizado em avaliação de estratégias orientadas a aspectos, gerando código para duas linguagens de programação orientadas a aspectos, AspectLua e CaesarJ.
- Avaliação qualitativa preliminar dos modelos gerados das transformações propostas em nossa abordagem, bem como avaliação da generalidade e reusabilidade das regras de mapeamento.
- Validação preliminar do Metaspin, visto que ainda não existem outras abordagens que se utilizem do metamodelo.
- Publicação do artigo sobre MaRiSA-Lua no SBCARS 2009
- Publicação do artigo sobre MaRiSA-AOCode no LA-Wasp 2009.

6.2 Trabalhos Futuros

Com o intuito de dar continuidade a pesquisa desenvolvida nessa dissertação, alguns trabalhos futuros podem ser listados:

- Aplicação de um conjunto de métricas como forma de avaliar quantitativamente os modelos gerados.
- Utilização de outras linguagens de programação (por exemplo, como HyperJ) visando verificar a generalidade das regras de mapeamento, bem como sua reusabilidade para linguagens que não seguem a idéia de orientação a aspectos providas pela linguagem AspectJ.
- Utilizar outras linguagens representativas em nível de projeto detalhado (e.g. ThemeML) para a construção de regras de mapeamento genéricas com o nível abstrato de programação idealizado neste trabalho através do uso do Metaspin.
- Utilizar ao menos uma linguagem simétrica que permita a representação de projeto detalhado, de forma avaliar s eo Metaspin também suporta o mapeamento de linguagens simétricas de projeto detalhado apra o nível abstrato de programação.
- Definição e implementação de regras de transformação inversa de código OA para diagramas de projeto detalhado em aSideML.

- Implementação de regras de mapeamento que permitam a geração de código mais detalhado, a partir da modelagem comportamental de aSideML.
- Construção de um *plugin* que integre todas a ferramentas MaRiSA permitindo a sincronização automática entre os modelos de diferentes fases de desenvolvimento, bem como prover um processo completo dentro da IDE Eclipse que permite a modelagem de sistemas OA desde a fase de requisitos até a codificação.

7 Referências

Acceleo. Disponível em: <http://www.acceleo.org>. Acesso em outubro de 2009.

Alves, M.P., Pires, P.F, Delicato, F.C. e Campos, L.M. "CrossMDA: *a model driven approach for aspect management*". *In Journal of Universal Computer Science*, vol. 14, n. 8, 2008.

ATL. "ATL: Atlas Transformation Language". ATL User Manual 0.7, 2006.

Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K. "An Overview of CaesarJ", 2005.

Bakker, J et al. "Characterization of early aspects approaches". Chicago, USA: The Early Aspects Workshop, 2005.

Basch, M. e Sanchez, A. "Incorporating aspects into the UML". In Proceedings of the AOM workshop at AOSD, 2003.

Budinsk, F. "Eclipse Modeling Framework: a developer's guide". England , Addisson Wesley, 2004.

Batista, T., Bastarrica, C., Soares, S., e Fernandes, L. "A Marriage of MDD and Early Aspects in Software Product Line Development". In Early Aspects Workshop at SPLC 2008: Aspect-Oriented Requirements and Architecture for Product Lines co-located with the 12th International Software Product Line Conference 2008, pp. 97-103, Limerick - Ireland, ISBN 978-1-905952-06-9, Setembro, 2008.

Brichau et al, 2006. "An Initial Metamodel for Aspect-Oriented Programming Languages".Technical report of AOSD-Europe, Fevereiro, 2006.

Brichau, J., Haupt, M. "Survey of Aspect-Oriented Languages and Execution Models. Technical report of AOSD-Europe, Maio, 2005.

Cacho N. et al. 2005. "AspectLua: A Dynamic AOP Approach". Journal of Universal Computer Science, vol. 11, no. 7, 2005, 1177-1197.

Chavez, C. V. "Um Enfoque Baseado em Modelos para Design Orientado a Aspectos". Tese (Doutorado) – Pontifica Universidade Católica do Rio de Janeiro. Rio de Janeiro, 2004.

Chitchyan, et al. (2005) "Survey of Analysis and Design Approaches", Survey of AOSD-Europe Network of Excellence, Maio, 2005.

Cottenier, T., van den Berg, A., Elrad, T. "Motorola weaver: Model weaving in a large industrial context". In Proc. of the 6th Int. Conference on Aspect-Oriented Software Development, Industry Track (AOSD), British Columbia, Canada, 2007.

Deursen, A. van, Klint, P. e Visser, J. "Domain-specific Languages: an annotated bibliography", ACM SIGPLAN Notices, 2000.

Eclipse. Disponível em: <http://www.eclipse.org/>. Acesso em: junho de 2009.

EMF. Disponível em: < http://www.eclipse.org/modeling/emf/>. Acesso em: junho de 2009.

Evermann, J. 2007. “A meta-level specification and profile for AspectJ in UML”. In Proceedings of the 10th international Workshop on Aspect-Oriented Modeling (Vancouver, Canada, March 12 - 12, 2007). AOM '07, vol. 209. ACM, New York, NY, 21-27.

Fuentes, L e Sanchez, P. (2007). “Towards executable aspect-oriented UML models”. In Proceedings of the 10th international Workshop on Aspect-Oriented Modeling (Vancouver, Canada, March 12 - 12, 2007). AOM '07, vol. 209. ACM, New York, NY, 28-34.

Filman, R. E. et al. “Aspect-Oriented Software Development”. Boston, Addison Wesley, 2005.

Guimarães, E. T, Medeiros, A. L. , Batista, T. V. e Minora, L. A. “MARISA-LUA: A MDD approach to Integrate Detailed Project and Code in Aspect-Oriented Software Development”. In: *III Latin American Workshop on Aspect-Oriented Software Development* LA-WASP, 2009, Fortaleza. Anais do III Latin American Workshop on Aspect-Oriented Software Development LA-WASP, 2009a.

Guimarães, E.T, Cacho, N. e Batista, T.V. “Uma Estratégia baseada em Metamodelo para Geração de Código Orientado a Aspectos”.2009, Natal. III Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS ‘09), 2009b.

Groher, I. e Baumgarth, T. “Aspect-Orientation from Design to Code”. Munich, Alemanha. Proceedings of the Early Aspects 2004 - Aspect-Oriented Requirements Engineering and Architecture Design. Lancaster, Março, 2006.

Havinga, W., Staijen, T., Rensink, A., Bergmans, L., van den Berg, K. “An Abstract Metemodel for Aspect Languages”.

Ierusalimschy, R. Programming in Lua. Second Edition, Rio de Janeiro, 2006.

Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K. “Overview of CaesarJ”. In Transactions on Aspect-Oriented Software Development I, LNCS, vol. 3880, pp. 135-173, Fevereiro, 2006.

Jouault, F., Bézivin, J., Kurtev, I. “KM3: a DSL for Metamodel Specification”. In: IFIP International Conference on Formal Methods For Open Object-Based Distributed Systems, Bologna, Italy, 2003. Proceedings of 8th IFIP, p. 171-185. Bologna, Italy, 2003.

Jouault, F. e Kurtev, I. “Transforming Models with ATL”. In Proceedings of the Model Transformation in Practice Workshop, October 3rd 2005, part of the MoDELS 2005 Conference.

Jouault, F, Bézivin, J., Kurtev, I. “TCS: a DSL for the specification of textual concrete syntaxes in model engineering”. GPCE - Generative Programming and Component Engineering: pp. 249-254, 2006.

Kiczales, G. et al. “Aspect-Oriented Programming”. In Proceedings of the European Conference on Object-Oriented Programming, 1997.

Medeiros, A. L, “MARISA-MDD: uma abordagem para transformações entre Modelos Orientados à Aspectos: dos requisitos ao Projeto Detalhado”. Dissertação (Mestrado) – UFRN, 2008.

Medeiros, A. L., Fernandes, L., Batista, T., Minora, L. “Requisitos e Arquitetura de Software Orientada a Aspectos: Uma Integração Sinérgica”. XXI Simpósio Brasileiro de Engenharia de Software (SBES), João Pessoa, PB, Outubro, 2007.

Medeiros, A. L, “MARISA-MDD: uma abordagem para transformações entre Modelos
Orientados à Aspectos: dos requisitos ao Projeto Detalhado”. Dissertação (Mestrado) – UFRN, 2008.

Medeiros , A. L., Batista, T., Chavez, C. “MARISA-DP -- from Architecture to Design: an MDD approach”. Early Aspects Workshop at AOSD 2009: Aspect-Oriented Requirements Engineering and Architecture Design, Charlottesville - Virginia - USA , pp , March 2009.

Mosconi, M., Charfi, A., Svacina, J., Wloka, J. “Applying and evaluating AOM for platform independent behavioral UML models”. In Proceedings of the AOSD Workshop on Aspect-Oriented Modeling (Brussels, Belgium, April 01 - 01, 2008). AOM '08. ACM, New York, NY, 19-24.

OMG/XMI. Disponível em: < http://www.omg.org/technology/xml/> Acesso em dezembro de 2009.

OMG/MDA. Disponível em: <http:// www.omg.org/mda/>. Acesso em dezembro de 2009.

OMG/QVT. Disponível em: <http://www.omg.org/spec/QVT/1.0/>. Acesso em dezembro de 2009.

Parr, T. “The Definitive ANTLR Reference: Building Domain-Specific Languages”, The Pragmatic Bookshelf, maio, 2007.

Rashid, A., Moreira, R. e Araujo, J. “Modularisation and Composition of Aspectual Requirements”. In: 2nd International Conference on Aspect-Oriented Software Development, 2003.

Rentsch, T. Object-Oriented Programming. ACME SIGLAN Notices, v. 17 N^{o}9, 1982.

Silva, L. “Uma Estratégia Orientada a Aspectos para Modelagem de Requisitos”. Tese de Doutorado em Engenharia de Software - PUC-Rio. Rio de Janeiro, 220p. 2006.

Soares, S. et al. (2002). “Implementing Distribution and Persistence Aspects with AspectJ”. In Proceedings of the *Object Oriented Programming, Systems, Languages and Applications* (OOPSLA’02), pp. 174-190, 2002.

Stahl, T., Voelter, M., Czarnecki, K. “Model-Driven Software Development, Technology, Engineering, Management”. England: John Wiley & Sons, 2006.

Stein, D., Hanenberg, S., Unland, R. “Designing aspect-oriented crosscutting in UML”. In Proceedings of the AOM with UML workshop at AOSD, 2002.

Sun. Disponível em: <http://www.java.sun.com>. Acesso em outubro de 2009.

Together. Disponível em: < http://www.borland.com/us/products/together/>. Acesso em dezembro de 2009.

UML. Disponível em: < 2003http://www.uml.org/>. Acesso em: junho de 2009.

Uetanabara, J., Camargo, V.V., Christina, V.F. “UML-AOF: A Profile for Modeling Aspect-Oriented Framework”. In Proceedings of the 13th workshop on aspect-oriented modeling, Charlottesville, Virginia, EUA, 2009.

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