DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

“Modelagem e Cômputo de Métricas de Interesses no Contexto de Modernização de Sistemas Legados”

ALUNO: Raphael Rodrigues Honda

ORIENTADOR: Prof. Dr. Valter Vieira de Camargo

São Carlos
Setembro/2014

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS - SP
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MODELAGEM E CÔMPUTO DE MÉTRICAS DE INTERESSESNO CONTEXTO DE MODERNIZAÇÃO DE SISTEMAS LEGADOS

RAPHAEL RODRIGUES HONDA

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP
Setembro/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MODELAGEM E CÔMPUTO DE MÉTRICAS DE INTERESSE NO CONTEXTO DE MODERNIZAÇÃO DE SISTEMAS LEGADOS

RAPHAEL RODRIGUES HONDA

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientador: Dr. Valter Vieira de Camargo

São Carlos - SP
Setembro/2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

H771mc

Honda, Raphael Rodrigues.
Modelagem e cômputo de métricas de interesse no contexto de modernização de sistemas legados / Raphael Rodrigues Honda. -- São Carlos : UFSCar, 2014.
119 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2014.

1. Engenharia de software. 2. Reengenharia orientada à aspectos. 3. Medição. 4. *Architecture-Driven Modernization* - ADM. 5. Modelos. 6. *Knowledge Discovery Metamodel* - KDM. I. Título.

CDD: 005.1 (20ª)

Universidade Federal de São Carlos

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

"Modelagem e Cômputo de Métricas de Interesses no Contexto de Modernização de Sistemas Legados"

Raphael Rodrigues Honda

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação

Membros da Banca:

Prof. Dr. Valter Vieira de Camargo
(Orientador - DC/UFSCar)

Prof. Dr. Daniel Lucrédio
(DC/UFSCar)

Prof. Dr. Cláudio Nogueira Sant'Anna
(UFBA)

São Carlos
Outubro/2014

Pela memória de Irene Ernestina Werkling da Silva (a “Vó”).
(1921-2010)

AGRADECIMENTO

Agradeço a todos que, direta ou indiretamente, acreditaram em mim quando ingressei no curso de mestrado. Agradeço em especial a minha mãe Vera, por ser um exemplo de perseverança em minha vida, às minhas tias Rosa Maria e Aparecida, pelo apoio desde que eu era uma criança, sempre me tratando como um filho e a toda a minha família.

Agradeço aos amigos de república em São Carlos (Davi, Fábio e Fernando), aos amigos do grupo **SemDisquete**, ao Rogério da Silva Maciel e ao Marcelo (e todo o pessoal de São Paulo que já comprou minhas rifas - Massae, Ivete e Queico), ajudando desde sempre. Aos amigos de graduação: Rafael Benito, José Cimino, Márcio, Laura e Jonathan, aos colegas do Departamento de Computação da Universidade Federal de São Carlos, ao Paulinho da Ad/Mustache, ao Dadinho, Ana Paula, Larissa Kinoshita, Dani, Dieguinho, Gordo, Alice, Curee e a galera toda que eu esqueci.

Agradeço ao meu orientador Prof. Dr. Valter Vieira de Camargo pela orientação e exigência durante meu período como seu orientando. Muitas vezes estivemos distantes, mas sempre trocamos ideias e tivemos conversas construtivas sobre este trabalho, tendo em mente que a qualidade da pesquisa é muito importante. Agradeço também aos amigos de laboratório *AdvanSE*: Bruno Marinho e Rafael Durelli, pela parceria.

Agradeço também à minha namorada Annelise. Anne, obrigado pela paciência nos dias de queixa, reclamações e cansaço na fase final do mestrado, nunca esquecerei!

Agradeço à minha antiga coordenadora e professora de matemática do ensino médio, profª. Ana Dolores. Obrigado pelo dia em que, ao invés de me mandar para a Diretoria (depois de cometer mais um ato impensado), me chamou em um canto e me disse palavras motivadoras, isso fez diferença na minha vida.

Aviso ao leitor: a escrita científica reprime sentimentos, por isso, caso sinta indiferença ou imparcialidade demais, desculpe-me e lembre-se de que sempre podemos conversar melhor e discutir as ideias aqui propostas por outros meios de comunicação. Não hesite em entrar em contato caso tenha alguma sugestão para trabalhos futuros ou errata em relação a esse trabalho.

Essa pesquisa foi realizada com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

"A Felicidade é uma questão de ignorância"

Raphael Rodrigues Honda

Resumo

Manter sistemas legados é uma atividade complexa e cara para muitas empresas. Uma alternativa para este problema é a Modernização Dirigida à Arquitetura (*Architecture-Driven Modernization* - ADM), proposta pelo OMG (*Object Management Group*). A ADM consiste em um conjunto de princípios que apoiam a modernização de sistemas utilizando modelos. O *Knowledge Discovery Metamodel* (KDM) é o principal metamodelo da ADM e é capaz de representar diversas características de um sistema, como código-fonte, arquivos de configuração e de interface gráfica. Por meio de um processo de engenharia reversa apoiado por ferramentas é possível extrair conhecimento do código-fonte legado e armazená-lo em instâncias do metamodelo KDM. Outro metamodelo da ADM pertinente a este projeto é o *Structured Metrics Metamodel* (SMM) que torna possível a especificação de métricas e também a representação dos resultados de medições realizadas em modelos KDM. Quando decide-se modernizar um sistema legado, uma alternativa que procura melhorar o nível de modularização dos interesses de um sistema é a orientação a aspetos. Considerando essa alternativa, o objetivo deste projeto é apresentar uma abordagem para definição e aplicação de métricas de interesse em instâncias do metamodelo KDM. Esse tipo de medição precisa de uma mineração de interesses prévia, que realiza anotações nos componentes do sistema indicando qual interesse ele implementa. Para alcançar o objetivo do projeto, foi desenvolvida uma abordagem completa de medição de interesses utilizando modelos da ADM, composta por uma extensão do KDM para a representação de software orientado a aspectos (AO-KDM), uma biblioteca de métricas de interesses no formato SMM (CCML) desenvolvida com o intuito de ser parametrizável pelo Engenheiro de Modernização. Portanto, as métricas definidas neste projeto podem ser reusadas em outros projetos. Além disso, foi desenvolvida uma ferramenta de apoio computacional (CMEE) capaz de lidar com parametrização de anotações (anotações de interesses realizadas por ferramentas de mineração) que permite que modelos anotados com diferentes ferramentas de mineração possam ser medidos por métricas SMM.

Palavras-chave: Modernização, Medição, Métricas, ADM, SMM, KDM, Aspectos, Evolução, Interesses Transversais, Reengenharia, Modelos.

Abstract

Maintaining legacy systems is a complex and expensive activity for many companies. An alternative to this problem is the Architecture-Driven Modernization (ADM), proposed by the OMG (Object Management Group). ADM is a set of principles that support the modernization of systems using models. The Knowledge Discovery Metamodel (KDM) is the main ADM metamodel and it is able to represent various characteristics of a system, such as source code, configuration files and GUI. Through a reverse engineering process supported by tools is possible to extract knowledge from legacy source code and store it in KDM metamodel instances. Another metamodel that is important to this project is the Structured Metrics Metamodel (SMM) that allows the specification of metrics and also the representation of the measurements results performed on KDM models. When we decide to modernize a legacy system, an alternative that aims to improve concerns modularization of a system is the Aspect-Oriented Programming. Considering this alternative, the main goal of this project is to present an approach to defining and computing concern metrics in instances of KDM metamodel. This kind of measurement needs a prior concern mining that make notes on system components indicating concerns which it implements. To achieve the project objective, a complete approach to measure concerns using ADM models was developed, this approached is composed by an extension of KDM metamodel for representing Aspect-Oriented Software (AO-KDM), a concern metrics library in SMM format (CCML) developed in order to be parameterized by the Modernization Engineer. Therefore, the metrics defined in this project can be reused in other projects. Furthermore, we have developed a tool (CMEE) capable of handling parameterization annotations (notes about concerns made by the mining tools) that allows that models annotated by different mining tools could be measured by SMM metrics.

Keywords: Modernization, Measurement, Metric, ADM, SMM, KDM, Aspects, Evolution, Crosscutting Concerns, Reengineering, Model.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACD –*Advice Crosscutting Degree*

ADM–*Architecture Driven Modernization*

ADMPR - *ADM Pattern Recognition*

ADMTF – *ADM Task Force*

ADMVS - *ADM Visualization Specification*

ADMRS - *ADM Refactoring Specification*

AO-KDM –*Aspect-Oriented KDM*

ASTM – *Abstract Syntax Tree Metamodel*

CCML – *Croscutting Concern Metrics Library*

CIM – *Computing Independent Model*

CDA –*Crosscutting Degree of an Aspect*

CDC – *Concern Diffusion over Components*

CDO – *Concern Diffusion over Operations*

CDLOC – *Concern Diffusion over Lines of Code*

CMEE–*Concern Metrics Execution Engine*

COMO –*Component-Oriented Modernization*

CSV – *Comma-Separeted Value*

DOF –*Degree of Focus*

DOS –*Degree of Scattering*

DSL –*Domain Specific Language*

FCD –*Feature Crosscutting Degree*

FTs –*Frameworks Transversais*

KDM – *Knowledge Discovery Metamodel*

LCOO – *Lack of Cohesion in Operations*

LCOM –*Lack of Cohesion in Methods*

MAMBA – *Measurement Architecture for Model-Based Analysis*

MDA – *Model-Driven Architecture*

MDD – *Model-Driven Development*

MDL – *Metrics Definition Language*

MOF– *Meta Object Facility*

MQL – *Metrics Query Language*

M2M– *Model To Model*

NOF– *Number of Features*

OA – *Orientação a Aspectos*

OCL – *Object Constraint Language*

OO – *Orientação a Objetos*

OMG – *Object Management Group*

PIM – *Platform Independent Model*

POA – *Programação Orientada a Aspectos*

POO – *Programação Orientada a Objetos*

PSM – *Platform Specific Model*

SMM– *Structured Metrics Metamodel*

XMI– *XML Metadata Interchange*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contextualização

Sistemas legados são aqueles cujos custos de manutenção e evolução encontram-se fora dos níveis aceitáveis para uma organização, mas que ainda são úteis para apoiar seus processos internos. Frequentemente, esse tipo de software possui sua estrutura inconsistente com sua documentação, o que dificulta sua manutenção. Porém, substituí-lo completamente pode ser uma tarefa muito cara e propensa a erros (Sommerville, 2011). Segundo Pressman (2011), uma empresa de software pode despender de 60% a 70% de todos os seus recursos com manutenção de software.

Uma alternativa à substituição de um sistema legado são os tradicionais processos de reengenharia. Entretanto, um termo mais recente e adotado pelo OMG é a "Modernização Dirigida à Arquitetura" (*Architecture-driven Modernization - ADM)*, que consiste basicamente em uma Reengenharia Apoiada por Modelos. A ADM começou em 2003 com a criação da ADMTF (*Architecture-Driven Modernization Task Force*), uma iniciativa para a utilização dos conceitos da MDA (*Model-Driven Architecture*) no contexto de modernização de software (OMG, 2010). O objetivo da ADM é definir um conjunto de metamodelos padrões para representar a informação envolvida em um processo de modernização e facilitar a interoperabilidade entre ferramentas (Izquierdo, 2010).

Dentre os metamodelos propostos estão o metamodelo KDM (K*nowledge Discovery Metamodel*) e o Metamodelo de Métricas Estruturadas (SMM - *Structured*

Metrics Metamodel). O objetivo do KDM é representar por meio de metaclasses, diferentes artefatos de um sistema legado, como código-fonte, interfaces de usuário, base de dados, arquitetura e conceitos de negócio (Pérez-Castillo *et al.,2*011). Ele também permite representar níveis baixos e altos de abstração e serve como base para a descoberta e representação do conhecimento contido em sistemas.

O metamodelo SMM, por sua vez, tem o objetivo de permitir a especificação de métricas para o metamodelo KDM e a representação do resultado da aplicação dessas métricas. A especificação de uma métrica SMM inclui detalhes estruturais como nome, escopo, tipo e também a parte comportamental, representada por uma operação escrita em OCL, XPath ou alguma outra linguagem voltada a modelos. É nesse código que está descrito como a métrica será calculada. Também é importante ressaltar que esse código menciona/usa nomes de metaclasses do metamodelo KDM envolvidas na medição, tornando a especificação da métrica dependente e específica do modelo alvo que será medido. Para facilitar a leitura deste projeto, quando o objetivo for mencionar "instâncias do metamodelo KDM que representam aplicações" ou “instâncias do metamodelo SMM”, será usado apenas o termo "modelo KDM” e “modelo SMM”.

Em uma atividade de modernização, geralmente usa-se o termo “*as-is*” para representar o sistema “como está” (legado) e o termo "*to-be*” para representar o sistema alvo, ou seja, como o sistema ficará após a modernização. Realizar medições é uma tarefa importante para analisar as deficiências do sistema “*as-is*”, estimar e planejar como o sistema alvo será, além de avaliar se o novo sistema realmente se encontra como o planejado.

A POA (Programação Orientada a Aspectos) é uma alternativa promissora quando o objetivo é realizar uma modernização que visa tratar interesses que ficam espalhados e entrelaçados pelo sistema, conhecidos como "interesses transversais". Um interesse é um conceito abstrato relacionado com requisitos funcionais ou não funcionais de um sistema, como persistência, *logging* e segurança. A POA visa separar esses interesses transversais e agrupá-los em aspectos, melhorando assim a modularização do software. Assim, a modernização de um sistema OO legado para um sistema OA é uma alternativa interessante que tende a diminuir os custos com manutenções futuras. Esse tipo de modernização é denominado, no contexto deste trabalho, de "modernização orientada a aspectos" ou “modernização orientada a modularização”.

Uma atividade importante na modernização orientada a aspectos é medir o espalhamento e o entrelaçamento de interesses usando métricas de interesse. Várias métricas de interesse já foram propostas na literatura e podemos citar como exemplo: CDC (*Concern Diffusion over Components*), CDO (*Concern Diffusion over Operations*) e CDLOC (*Concern Diffusion over Lines of Code*), LCOO (*Lack of Cohesion in Operations*) (Sant'Anna *et al.*, 2003). *Disparidade*, *Concentração* e *Dedicação* (Wong *et al.*, 2000), dentre outras. As métricas acima citadas foram definidas para o contexto de orientação a aspectos e servem para avaliar quantitativamente o nível de espalhamento dos interesses transversais no sistema.

A proposta original da ADM, principalmente o KDM, restringe-se (propositalmente) à programação orientada a objetos. Modernizações que tem como alvo outros domínios, tecnologias e paradigmas mais específicos, devem estender o metamodelo KDM para conseguir representar detalhes desses domínios. Uma forma de fazer essa extensão é alterando, removendo ou inserindo novas metaclasses no KDM, algo conhecido como extensão pesada (*heavy weight*). Na proposta original da ADM não é possível conduzir uma modernização orientada a aspectos porque o metamodelo KDM não possui metaclasses que representem as abstrações da POA. Dessa forma, modernizações orientadas a aspectos precisam da existência de um KDM estendido, que represente os conceitos desse paradigma. Uma extensão orientada a aspectos pesada foi elaborada no contexto deste trabalho.

O metamodelo SMM também está restrito a medir apenas os elementos (metaclasses) que existem no KDM, por exemplo, classes, métodos e atributos, camadas, regiões de código e interfaces de usuário. Esse metamodelo não prevê a especificação de métricas para medir interesses ou aspectos, já que "interesses" não são abstrações existentes em um código fonte de uma linguagem de programação. Geralmente, uma técnica que tem sido usada pela comunidade para representar interesses em modelos é por meio de anotações, em que elementos básicos de código-fonte são anotados com etiquetas (*tags*).

Como o paradigma OA tem sido constantemente referenciado como uma alternativa interessante para se modularizar interesses transversais, e considerando que esses são problemas inerentes de sistemas legados, identificou-se uma oportunidade de pesquisa que este projeto visa contribuir.

Para um melhor entendimento do contexto em que este projeto se insere, a Figura 1.1 mostra esquematicamente um processo de modernização orientado a

aspectos e as etapas em que medições de interesses devem ser feitas. O processo tem início quando o código-fonte legado é transformado em modelos KDM. Para isso, faz-se uso do *MoDisco Framework* (Bruneliere *et al.*, 2010), que possui um *parser* que realiza uma transformação código-para-modelo.

Em seguida alguma ferramenta de mineração pode ser utilizada para anotar o modelo KDM, apontando claramente quais elementos desse modelo (atributos, métodos, classes, etc.) contribuem para a implementação de determinados interesses transversais. Para este trabalho, foi utilizada a ferramenta CCKDM, desenvolvida no grupo de pesquisa AdvanSE (Santibánez, 2013).

Após a etapa de mineração/anotação, a ferramenta CMEE (desenvolvida no contexto deste projeto) é usada para medir o modelo que representa o sistema “as-is”. Para este trabalho, foi utilizada a ferramenta CMEE em conjunto com a biblioteca CCML. Para a fase de medição, o primeiro passo do usuário é escolher as métricas de interesses que ele deseja aplicar. As métricas utilizadas como exemplos na Figura 1.1 são a CDO e CDC e tais métricas encontram-se especificadas e disponibilizadas na (CCML – *Croscutting Concern Metrics Library*). Em seguida, o usuário precisa informar para a ferramenta como o modelo de entrada está anotado, ou seja, qual o padrão de anotação que ele está usando (mecanismo de parametrização de Tags). Por fim, a CMEE mostra para o usuário os valores resultantes da medição.

Na sequência, os modelos passam por um processo de refatoração, o qual não está no escopo desse trabalho. O processo de refatoração gera modelos KDM Orientados a Aspectos, que são novamente minerados pela CCKDM e geram como saída modelos KDM orientados a aspectos anotados. Esses modelos KDM orientados a aspectos são representados por uma extensão orientada a aspectos do metamodelo KDM, também criada também no contexto deste projeto e denominada AO-KDM.

Em seguida, novamente, a ferramenta CMEE mede os modelos anotados e coleta novos índices relativos às métricas CDO e CDC. Então o engenheiro de modernização verifica se houve melhoria nos índices coletados pelas métricas e, caso esteja satisfeito com o resultado das refatorações, uma nova transformação é realizada para gerar o código-fonte da aplicação final. Caso o Engenheiro não fique satisfeito com o resultado das métricas aplicadas no modelo que representa o

sistema "*to-be*", deve-se então voltar à fase de refatorações e aplicar novas mudanças, medir novamente, e assim por diante.

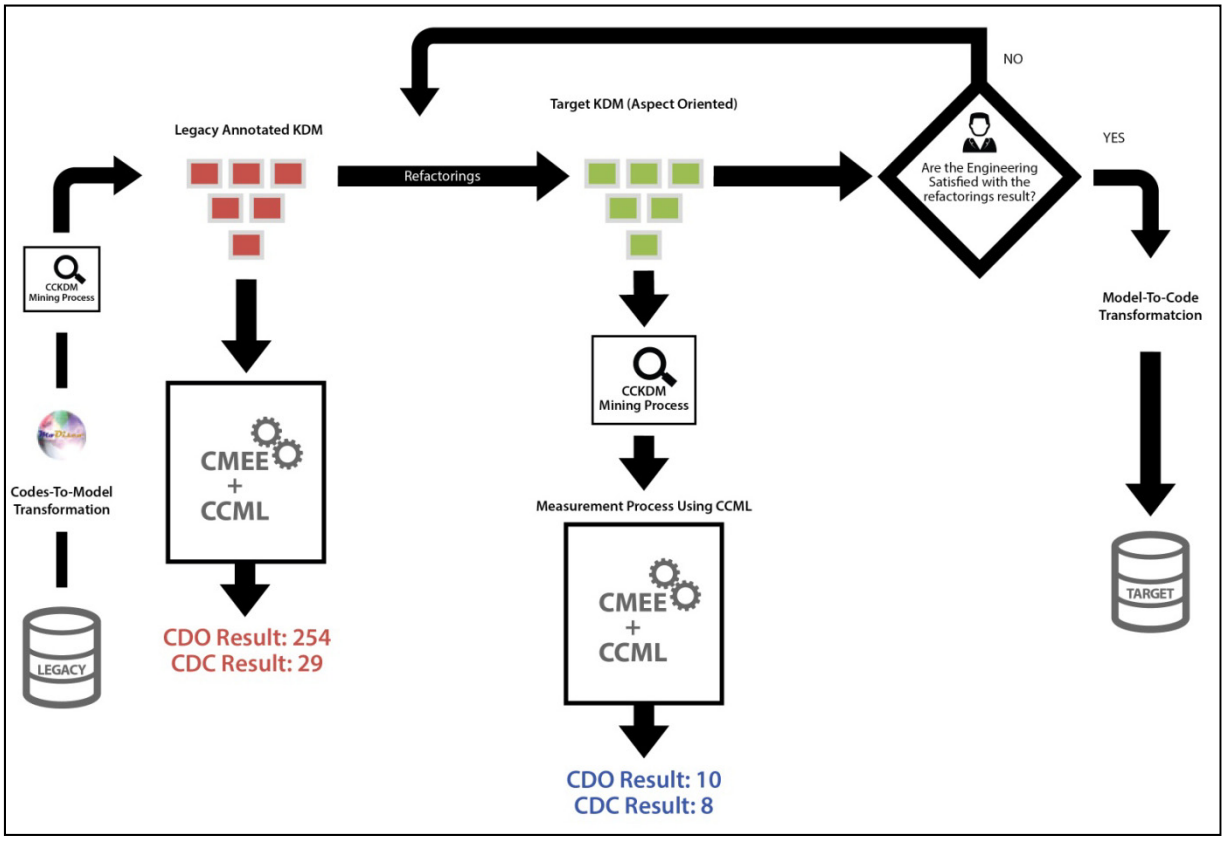

Figura 1.1 - Contextualização da CMEE em um processo de modernização apoiado pela ADM.

Percebe-se então que a ferramenta CMEE se faz importante em um processo de modernização para ajudar o Engenheiro de Modernização a decidir quais refatorações devem ser aplicadas, e depois, as medições também são úteis para saber se as refatorações foram bem sucedidas. Caso as refatorações não tenham surtido efeito, elas podem ser reaplicadas quantas vezes forem necessárias, até que o Engenheiro consiga o resultado esperado para o software modernizado.

A ferramenta foi construída na forma de *Plug-in* do Eclipse IDE, ou seja, um componente que pode ser adicionado ao Eclipse e é útil em um processo de modernização apoiado pela ADM. Sua localização no processo de modernização sempre será antes e depois das refatorações, como pode ser visto na Figura 1.1.

Um detalhe importante a ser mencionado é que as métricas disponibilizadas na biblioteca são independentes de plataforma. Assim, desenvolvedores de *plug-ins*

para outros ambientes/plataformas também podem se beneficiar dela para construir suas ferramentas de medição. O trabalho principal consiste em construir um módulo que carregue o modelo XMI (biblioteca de métricas) e consiga executar o código *XPath* que representa o comportamento da métrica.

1.2 Motivação

Considerando que uma característica intrínseca de sistemas legados é a presença de interesses transversais em sua arquitetura e que modernizações que visam melhorar a modularidade de sistemas são relevantes em contextos reais, as principais motivações aqui são:

i) ausência de estudos sobre a especificação de métricas de interesse usando o metamodelo SMM. A especificação de uma métrica de interesse deve ser genérica o bastante para propiciar seu reuso.

ii) ausência de ferramentas que apóiam o cômputo de tais métricas em modelos KDM que representam sistemas OO e OA. Sem o suporte de ferramentas, o trabalho de medição com as métricas SMM se torna complexo e propenso a erros.

iii) ausência de uma biblioteca de métricas de interesse SMM. A inexistência de uma biblioteca desse tipo dificulta a disseminação do metamodelo SMM e também leva engenheiros de modernização a criarem soluções que já foram criadas por outros desenvolvedores.

iv) ausência de estudos sobre a parametrização de métricas SMM para interesses. Em geral, métricas SMM são dependentes de abstrações existentes no metamodelo KDM, pois suas especificações mencionam nomes de metaclasses desse metamodelo. Entretanto, quando o objetivo é especificar métricas para medir abstrações não existentes em linguagens de programação, como "interesses", uma prática comum é anotar o modelo a ser medido. Entretanto, não há padrão para isso e cada engenheiro de software pode usar palavras-chave diferentes. Assim, é importante especificar uma métrica SMM parametrizada, em que os nomes das *Tags* que representam os interesses devem ser informados no momento da medição.

v) utilização de modelos permite uma representação do software em diversos níveis de abstração.

1.3 Objetivos

O objetivo mais amplo deste projeto é viabilizar a medição de espalhamento de interesses em modelos KDM que representem tanto sistemas orientados a objetos (OO) quanto sistemas orientados a aspectos (OA). Para atingir esse objetivo mais amplo, há outros objetivos mais específicos que precisaram ser atingidos:

1. Apresentar uma estratégia de parametrização de métricas SMM de forma a aumentar a reusabilidade das métricas de interesse especificadas. Particularmente, pretende-se garantir que as métricas propostas não sejam dependentes da forma como os modelos KDM alvo da medição estejam anotados;
2. Modelar uma biblioteca de métricas SMM de interesse que poderá ser utilizada por outros engenheiros de modernização. A ideia é evoluir essa biblioteca para que ela se torne um referencial para outros pesquisadores e para aqueles que pretendem aplicar as métricas na prática.
3. Facilitar a aplicação das métricas de interesse por meio do fornecimento de um *plug-in* para o ambiente Eclipse.

1.4 Contribuições

As principais contribuições deste projeto são:

1. Apresentação de uma forma parametrizada de especificação de métricas SMM que pode ser usada para medição de interesses (Objetivo 1).
2. Desenvolvimento e disponibilização de uma extensão do KDM para representar sistemas orientados a aspectos (Objetivo 1).
3. Desenvolvimento e disponibilização de uma Biblioteca de Métricas parametrizadas de interesse especificada em SMM, denominada CCML (*Crosscutting Concern Metrics Library*) (Objetivo 2).
4. Desenvolvimento e disponibilização de uma ferramenta (*Plug-in* do ambiente Eclipse) que aplica métricas de interesse em modelos KDM e

AO-KDM, gerando como saída modelos SMM com os resultados das medições (Objetivo 3).

1.5 Colaboração no Grupo de Pesquisa

Para a condução deste projeto foi necessário inicialmente o desenvolvimento de uma extensão orientada a aspetos do metamodelo KDM (Santos *et al*., 2014). Sem essa extensão, não seria possível representar um sistema orientado a aspectos usando esse metamodelo, e consequentemente, não teria como testar as métricas de interesse em modelos desse tipo. Assim, uma extensão pesada do metamodelo KDM foi criada em conjunto com o aluno de mestrado Bruno Marinho Santos, pois tal extensão também seria importante no contexto de seu trabalho. A extensão criada gerou um artigo publicado no Simpósio Brasileiro de Engenharia de Software (SBES), realizado em Maceió, neste ano de 2014.

Esse trabalho é mais uma contribuição para o grupo de pesquisa *AdvanSE* (*AdvancedResearch on Software Engineering*), da Universidade Federal de São Carlos (UFSCar). O grupo possui pesquisas em andamento sobre extensões, refatorações, mineração, métricas e validações de arquitetura em modelos KDM.

A ferramenta de mineração CCKDM, desenvolvida por Daniel SanMartín, também membro do grupo *AdvanSE* foi utilizada neste projeto para minerar os modelos KDM a serem medidos. A ferramenta CCKDM foi publicada no workshop LA-WASP de 2013 e conquistou o prêmio de melhor artigo desse workshop.

1.6 Organização do Trabalho

As seções do Capítulo 2 descrevem a ADM e seus objetivos, o metamodelo KDM e o metamodelo SMM. O Capítulo 3 introduz o conceito de programação orientada a aspectos e de medições de interesses.

O Capítulo 3 fala sobre a medição no contexto de separação de interesses. Seguido do Capítulo 4, que após uma fundamentação teórica dos assuntos

envolvidos na pesquisa, apresenta uma listagem e classificação dos trabalhos relacionados.

O Capítulo 5 fala sobre a extensão KDM-AO desenvolvida especialmente para apoiar esse trabalho, mostrando como é possível representar software orientado a aspectos com o KDM.

No Capítulo 6 é apresentada a definição de métricas de interesse utilizando o metamodelo SMM, suas limitações e possíveis adaptações de modelagem. Depois, no Capítulo 7, a ferramenta de apoio computacional da abordagem, denominada "CMEE: *Concern Metrics Execution Engine*" é apresentada em detalhes.

A avaliação da Ferramenta e abordagem é apresentada no Capítulo 8 e o Capítulo 9 apresentam as conclusões obtidas com este trabalho.

Capítulo 2

MODERNIZAÇÃO DIRIGIDA À ARQUITETURA (ADM)

2.1 Considerações Iniciais

A Modernização Dirigida à Arquitetura é a base para o desenvolvimento deste projeto e, devido à importância dos metamodelos *Knowledge Discovery Metamodel* (KDM) e *Structured Metrics Metamodel* (SMM), essa seção é dedicada para uma apresentação desses assuntos e também da ADM de uma forma geral. A Seção 2.2 explica como surgiu a iniciativa ADM, quais são seus objetivos e como é o processo de modernização proposto pela OMG. As Seções 2.3 e 2.4 apresentam os metamodelos KDM e SMM, respectivamente. A Seção 2.5 faz as considerações finais deste capítulo.

2.2 Modernização Dirigida a Arquitetura (ADM)

Em 2003, o OMG (*Object Management Group*) iniciou uma força tarefa denominada ADMTF (*Architecture-driven Modernization Task Force*) para criar especificações padronizadas e promover um consenso no contexto de modernização de aplicações existentes. ADM é um processo de compreensão e evolução de software existente com o propósito de prover melhorias no software (OMG, 2012),

que visa facilitar a interoperabilidade na manipulação de modelos envolvidos em seu processo.

Um dos pilares da ADM é o uso de modelos como artefatos primários, ou seja, a ADM é uma abordagem baseada no MDD (*Model-Driven Development*) e no MDA (*Model-Driven Architecture*), o que pode diminuir o nível de complexidade comparando-o com um processo de reengenharia comum, em que todos os esforços são direcionados ao código fonte, onde o nível de abstração é baixo.

Segundo Pérez-Castillo *et al.*(2011), o fluxo de um processo de modernização apoiado pela ADM possui três fases e é semelhante ao contorno de uma ferradura, conforme pode ser visto na Figura 2.1.

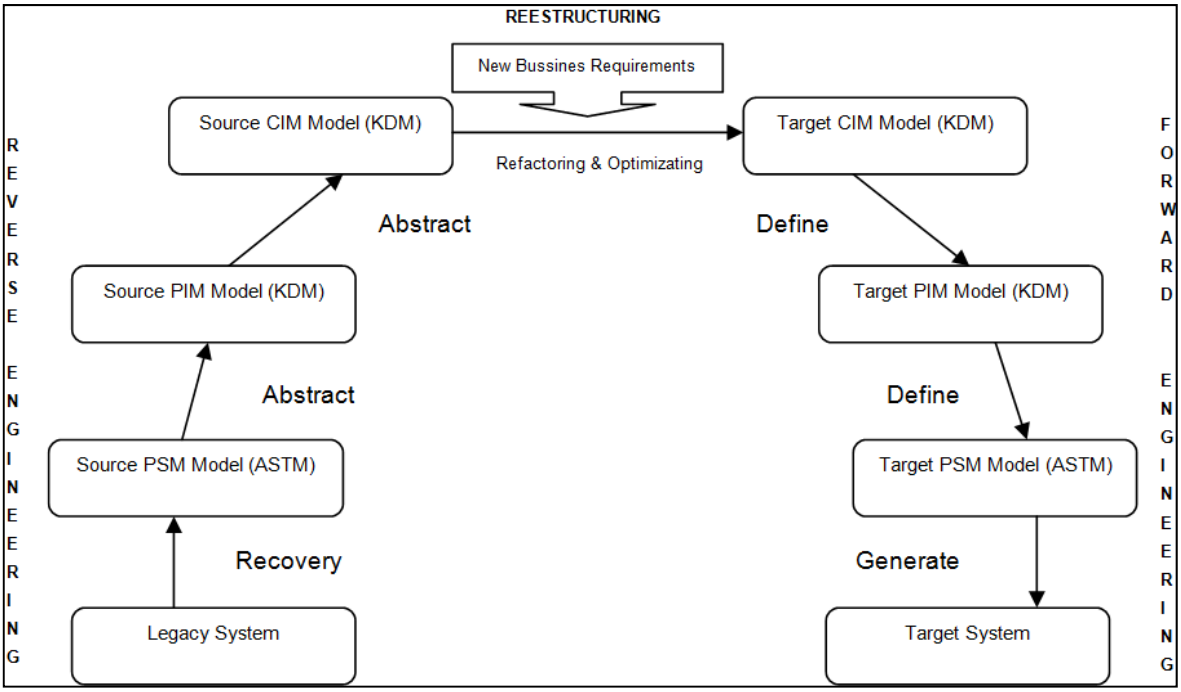

Figura 2.1 -Modelo de Modernização Ferradura (Pérez-Castillo *et al.*, 2011)

As fases são: Engenharia reversa, Reengenharia e Engenharia Avante. Partindo do lado inferior esquerdo, na parte da engenharia reversa, o conhecimento é extraído do sistema legado e um modelo PSM (*Platform Specific Model*) é gerado. O modelo PSM serve como base para a abstração e geração de um modelo PIM (*Platform Independent Model*), que serve como base para a abstração e geração do modelo CIM (*Computing Independent Model*), ou seja, durante a fase de Engenharia Reversa, transformações são feitas com o intuito de se obter uma representação de alto nível do sistema, independentemente da plataforma utilizada anteriormente.

O modelo PSM, representado por uma instância do metamodelo *Abstract Syntax Tree Metamodel* (ASTM), é um modelo específico de uma plataforma, ou seja, nele existem metadados relacionados a uma plataforma ou linguagem de programação específica. O modelo PIM, representado por modelos KDM, possui um nível de abstração maior, pois não contém informação específica de uma plataforma. Por último, temos o modelo CIM, também representado por modelos KDM, que é o modelo de mais alto nível no processo de modernização, esse modelo representa um ponto de vista do sistema independente de computação, ou seja, bastante abstrato.

Na fase de Reengenharia, refatorações, melhorias e novas regras de negócios podem ser introduzidas no sistema, e, com uma representação independente de plataforma do sistema modernizado, segue-se para a fase de Engenharia Avante, em que os modelos são novamente submetidos a uma série de transformações para chegar ao nível de código-fonte novamente.

Segundo Pérez-Castillo *et al.*(2011) e Sadovykh *et al.*(2009), as refatorações e melhorias podem ser realizadas nos três níveis de abstração: PSM, PIM e CIM, ou seja, existem cenários que é interessante realizar um processo de modernização a partir de modelos PSM ou PIM ou CIM, dependendo da necessidade de abstração.

A indústria de software possui uma demanda de evolução de sistemas legados, porém, antes da ADM, não haviam especificações padronizadas para suprir essa demanda de mercado.

Algumas vantagens da ADM são:

- Pesquisadores podem definir catálogos de refatoração independentes de linguagem e plataforma (Pérez-Castillo *et al.*, 2011). Isso é possível porque agora as refatorações devem ser elaboradas levando em conta apenas o metamodelo KDM. Como esse metamodelo é um padrão independente de plataforma, todas as refatorações elaboradas sobre ele podem ser aplicadas a quaisquer instâncias desse metamodelo, mesmo que elas representem sistemas de diferentes linguagens e plataformas, algo que não seria possível se as refatorações tivessem sido elaboradas para uma linguagem específica.
- O metamodelo KDM apóia a “descoberta do conhecimento” do sistema legado. Isso pode ser realizado por meio de ferramentas de terceiros,

tal como o *MoDisco* (Bruneliere *et al.*, 2010). Esse processo é possível mesmo tendo como documentação apenas o código-fonte.

- O metamodelo SMM permite que métricas sejam representadas como modelos, permitindo que engenheiros de software possam reusar métricas (representadas em modelos) em outros projetos em que os artefatos de software sejam modelos KDM. Além disso, os resultados das medições também são modelos, o que permite o desenvolvimento de ferramentas para cálculos estatísticos de projetos e uma base de dados de medições que pode ser compartilhada e reusada.

Para dar suporte aos objetivos da ADM, foram planejados diversos metamodelos, alguns desses estão em fase de desenvolvimento e outros já possuem versões disponíveis para uso: *Knowledge Discovery Metamodel* (KDM), descrito na Seção 2.3, *Abstract Syntax Tree Metamodel* (ASTM), que é um metamodelo complementar do KDM e pode representar o código em nível de árvore de sintaxe abstrata. *Structured Metrics Metamodel* (SMM) é um metamodelo para representação de métricas e resultados de medições, descrito na Seção 2.4. *ADM Pattern Recognition* (ADMPR), que facilita a busca de padrões em um software. *ADM Visualization Specification* (ADMVS), que tem como objetivo representar visualmente metadados de uma aplicação representada em KDM. *ADM Refactoring Specification* (ADMRS), que define maneiras nos quais modelos KDM podem ser usados para refatorar aplicações.

O estado atual de cada metamodelo pode ser visto na Tabela 2.1. Alguns metamodelos (em fase de desenvolvimento) possuem versões disponíveis apenas para membros do OMG.

Tabela 2.1- Situação dos Metamodelos da ADM

Metamodelo	Situação	Versão	Data
ADMPR (*ADM Pattern Recognition*)	Em desenvolvimento		
ADMRS (*ADM Refactoring Specification*)	Em desenvolvimento		
ADMVS (*ADM Visualization Specification*)	Em desenvolvimento		
ASTM (*Abstract Syntax Tree Metamodel*)	Disponível	1.0	Janeiro/2011
KDM (*Knowledge Discovery Metamodel*)	Disponível	1.3	Agosto/2011
SMM (*Structured Metrics Metamodel*)	Disponível	1.0	Janeiro/2012
	Em desenvolvimento	1.1	Novembro/2013
Transformation Metamodel	Em desenvolvimento		

2.3 Metamodelo de Descoberta de Conhecimento– *KDM (Knowledge Discovery Metamodel)*

Assim como os outros metamodelos da ADM, o KDM é baseado no metamodelo MOF (*Meta Object Facility*) e é dividido em quatro camadas, representando tanto os artefatos físicos quanto os lógicos. Cada camada de abstração separa o conhecimento sobre o sistema legado em vários interesses ortogonais que são bem conhecidos dentro do contexto da Engenharia de Software. (Pérez-Castillo *et al.*, 2011).

Como pode ser observado na Figura 2.2, o KDM possui quatro camadas (Pérez-Castillo *et al.*, 2011). A camada de infraestrutura (*Infrastructure Layer*) está no mais baixo nível de abstração e contém pacotes que são usados por toda a especificação do KDM. Essa camada é dividida em três pacotes: O *Core*, o *KDM* e o *Source*. O pacote *Core* define as construções básicas para criar e representar os elementos do metamodelo em todos os pacotes do KDM. O pacote *KDM* também possui construções básicas para criar e representar os elementos do metamodelo. Além disso, permite a introdução de novos elementos por meio de estereótipos, os quais podem modificar a semântica de um elemento. Já o pacote *Source* define o primeiro e mais básico modelo do KDM; o modelo de inventário. Com esse modelo é possível enumerar os artefatos físicos do sistema, como arquivos de código fonte, imagens e arquivos de configuração.

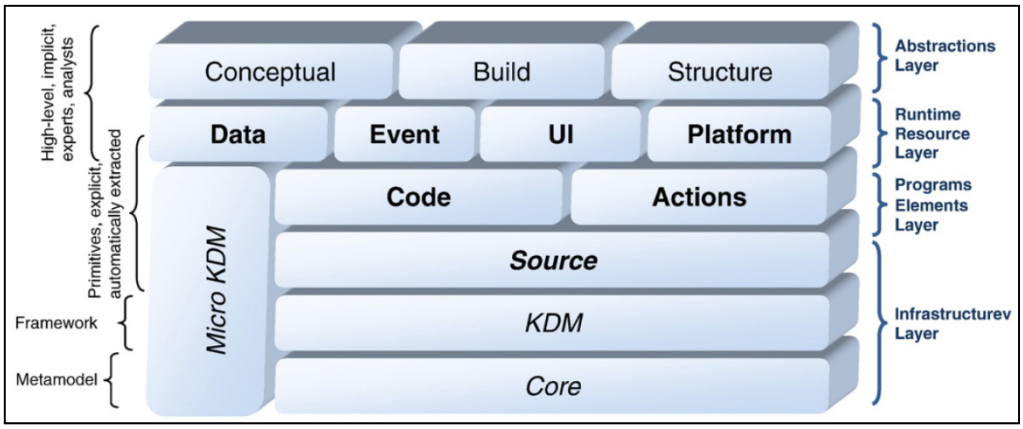

Figura 2.2 - Camadas, pacotes e interesses no KDM (Pérez-Castillo *et al.*, 2011)

A Camada de Elementos do Programa (*Program Elements Layer*) oferece um conjunto de elementos de metamodelo a fim de prover uma representação intermediária independente de linguagem para várias construções comuns de linguagens de programação. É dividido em dois pacotes: *Code* e *Action*. O pacote *Code* define um conjunto de elementos que representam elementos comuns em códigos-fontes definidos em diferentes linguagens de programação, tais como: classes, procedimentos, métodos, *templates* e interfaces.

A Figura 2.3 apresenta o modelo do pacote *Code*. Esse modelo representa informações referentes a itens de um trecho de código-fonte. O elemento *CodeItem* é especializado em outros três elementos: *Module*, *ComputationalObject* e *DataType*. O *Module* representa unidades do programa que contém outros elementos e podem ser usados como componente lógico do software. Um elemento *Module* pode ser, por exemplo, uma referência para um pacote ou para um arquivo do modelo de inventário. Elementos *ComputationalObject* representam objetos chamáveis, como *MethodUnits* e *CallableUnits*. Elementos *DataType* representam itens de dados do sistema legado, como variáveis e arquivos de registro. (Pérez-Castillo *et al.*, 2011).

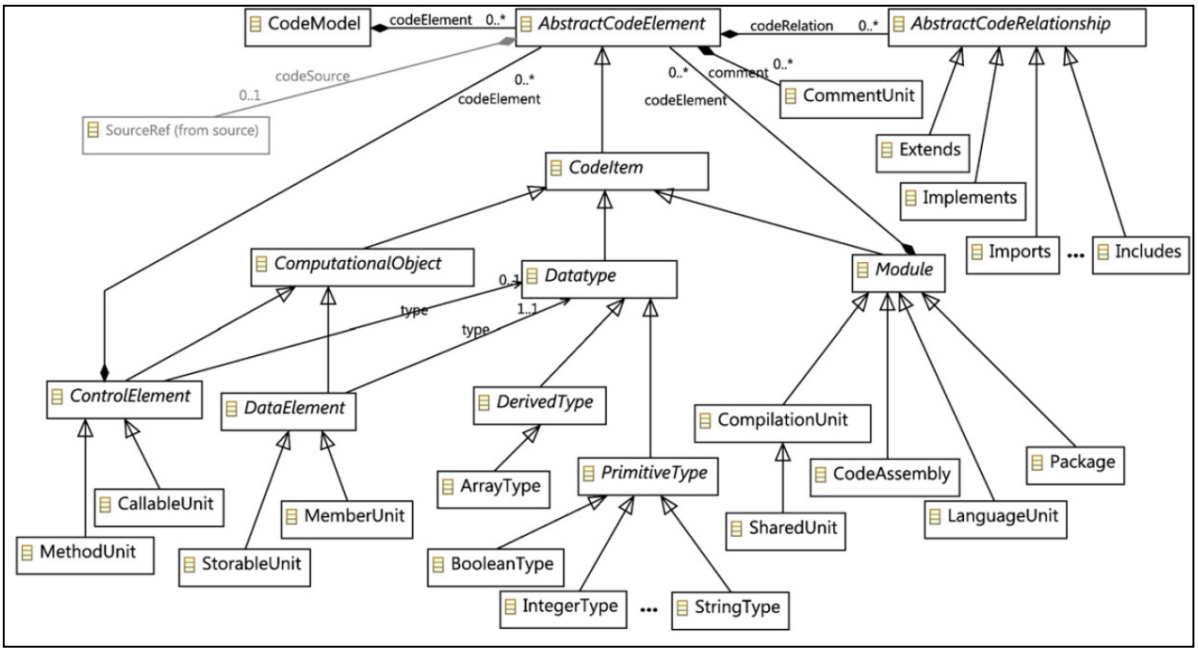

Figura 2.3 - Pacote Code da Camada de Elementos de Programa (Pérez-Castillo*et al.*, 2011)

O pacote *Action* estende o *Code* e define elementos que representam comportamentos de uma linguagem de programação e seus relacionamentos.

A camada de recursos de tempo de execução (*RuntimeResourceLayer*) permite a representação de elementos como dados, eventos, interface gráfica, processos, *threads*, etc. A divisão desta camada é feita em quatro pacotes: *Data*, *Event*, *UI* e *Platform*. O pacote *Data* pode representar artefatos relacionados com a persistência do sistema legado. O pacote *Event* define transições entre eventos durante a execução do sistema. O pacote *UI* permite representar todos os elementos de interface gráfica de usuário do sistema. E o pacote *Platform* permite representar a plataforma de execução do sistema legado.

A camada de abstrações (*Abstractions Layer*) define um conjunto de metamodelos com o propósito de representar conhecimento específico de domínio, bem como fornecer uma visão geral do sistema legado. Esta camada é dividia em três pacotes: *Conceptual*, *Build* e *Structure*. O pacote *Conceptual* define um metamodelo que permite a representação de informação específica de domínio, no qual o sistema legado faz parte. O pacote *Build* define um metamodelo que permite a representação de artefatos gerados pelo compilador durante o processo de compilação. E o pacote *Structure* define um metamodelo que permite a representação da arquitetura e estrutura do sistema legado.

Instâncias do KDM, assim como de outros metamodelos da ADM, são representadas por arquivos do tipo *XML MetadataInterchange (XMI)*. O XMI é um formato de troca de metadados para modelos que utilizam o XML, permitindo a interconexão de objetos no mesmo arquivo ou em arquivos diferentes por meio de identificadores (OMG XMI Specification, 2011).

2.4 Metamodelo de Métricas Estruturadas - SMM

O Metamodelo de Métricas Estruturadas (*Structured Metrics Metamodel - SMM*) é um metamodelo que proporciona um formato padrão tanto para definir métricas quanto para representar os resultados das medições. Os principais termos no SMM são *Measures* (métricas) e *measurements* (medições). Por *measure*, pode-

se entender um modelo de métrica que descreve uma maneira para calcular propriedades de um sistema. *Measurements* são medições, mais especificamente resultados da aplicação de uma *measure* em um modelo alvo a ser medido.

Para aplicar métricas SMM em modelos KDM, é necessária uma ferramenta para a automatização. Na literatura foram encontradas abordagens como a de Izquierdo (2010) e Frey *et al.*(2012), ambas sem suporte para softwares orientados a aspectos. O processo ocorre como ilustrado na Figura 2.4, a ferramenta deve receber um modelo de métricas SMM e um modelo KDM que representa um sistema ou parte dele, depois, um modelo SMM é gerado como saída. O arquivo de saída também é um do tipo SMM e contém o resultado das medições aplicadas.

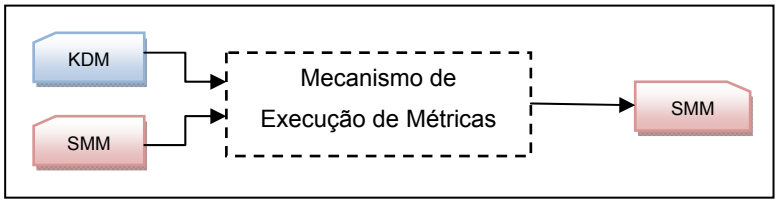

Figura 2.4 - Aplicação de Métricas SMM em Modelos KDM

Existem algumas vantagens no uso do SMM. Além de ser o modelo para métricas e medições na ADM, ele se faz vantajoso pelo fato de ser um padrão, logo, métricas definidas em conformidade com o metamodelo SMM podem ser testadas, reusadas e compartilhadas. Além disso, o uso de um padrão facilita o desenvolvimento de ferramentas compatíveis capazes de automatizar o processo de medição.

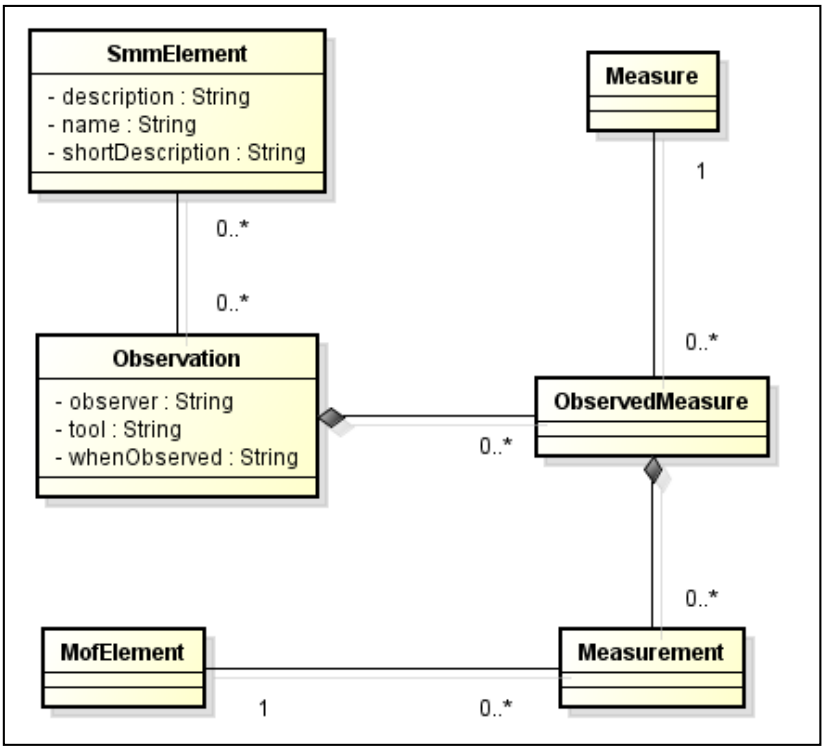

Figura 2.5–Abordagem principal do metamodelo SMM. OMG (2012) (Adaptada).

A Figura 2.5 mostra um diagrama de classes com as principais metaclasses do metamodelo SMM. O objetivo dessa figura é representar os principais elementos que são criados para representar uma ou mais métricas e também os resultados da aplicação das mesmas.

Os principais elementos do processo de medição são: *Observation,* que armazena em seus atributos dados como nome do engenheiro de modernização, ferramenta utilizada para a medição e data da medição. Pode-se também reparar que todo processo de medição possui uma instância da classe *ObservedMeasure*, que é o elo de ligação com a métrica a ser aplicada (elemento *Measure*) e o elemento *MOF-Based* (Baseado (desenvolvido) com base no metamodelo MOF) a ser medido (KDM é um exemplo de modelo *MOF-Based*). A instância da classe *Measurement* armazena os resultados das medições aplicadas no elemento alvo. Resumindo, em todo processo de medição, precisamos de um *Observation*, uma métrica, um elemento a ser medido e um elemento *measurement* para armazenar os resultados. Essas são as principais metaclasses do metamodelo SMM e obrigatórias em todo processo de medição.

O metamodelo SMM, como dito anteriormente, pode representar métricas e resultados de medições. As métricas representadas pelo SMM podem ser de

diversos tipos: dimensionais (resultados comparados de acordo com sua magnitude), contadoras (reconhecem uma determinada entidade, retornando *true)*, diretas (mais genérica, aplicáveis a outros casos), coletivas (passível de aplicação de função aritmética ao resultado da medição), binárias (associa duas métricas base) e *ranking* (trata os resultados de acordo com um ranking pré-estabelecido).

Métrica Dimensional é todo tipo de métrica cujo objetivo é coletar índices numéricos e ordená-los por sua magnitude. Métricas Dimensionais, como pode ser visto na Figura 2.6, possuem os seguintes atributos: *Name* (nome da métrica), *Label* (rótulo da métrica, ou seja, nome que será renderizado em uma ferramenta em conformidade com o metamodelo SMM), *Description* (descrição detalhada da métrica), *Trait* (campo para especificar que tipo de característica a métrica medirá), *Scope* (Elemento onde estão os elementos alvo da métrica), *LabelFormat* (campo onde é possível definir um formato com variáveis para a renderização do *Label* da métrica), *Categories* (campo onde é possível adicionar categorias – previamente cadastradas – para a métrica em questão) e *Unit* (unidade de medida da métrica, por exemplo, linhas de código.).

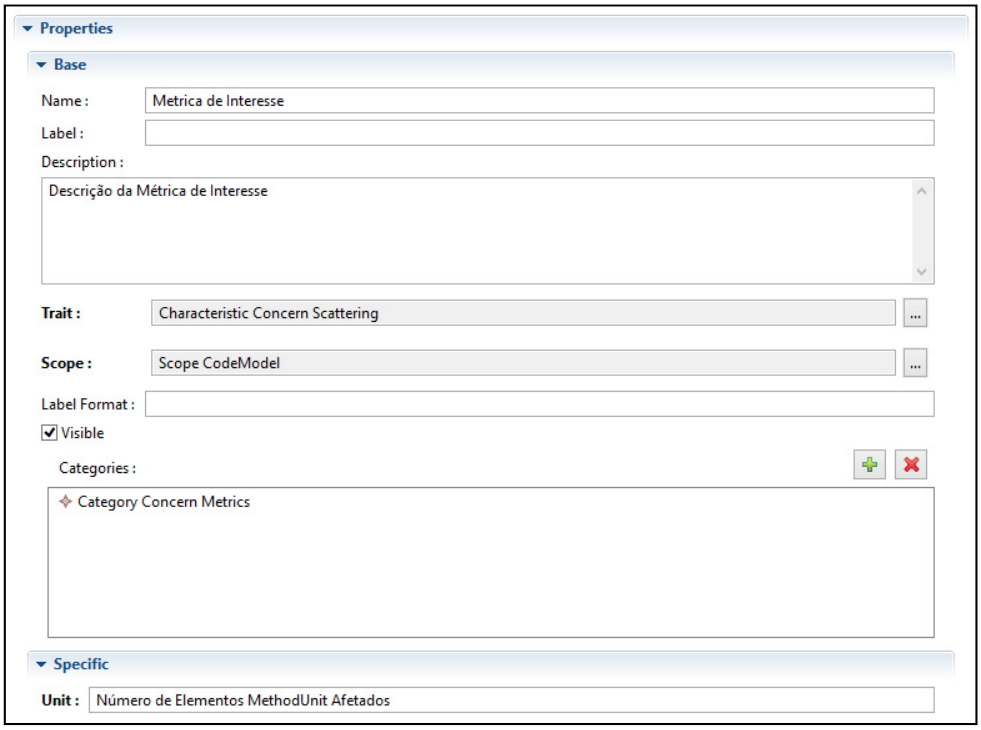

Figura 2.6 - Tela de Criação de uma Métrica Dimensional no Modisco SMM Editor.

A Métrica Dimensional é a subclasse da classe abstrata *Measure* e é a superclasse de vários outros tipos de métricas,ou seja, as métricas do tipo Direta, Binária, Contadora, e Coletiva possuem todos os atributos de uma métrica Dimensional, além de outros atributos que as tornam específicas para determinadas situações.

A Métrica Direta possui um atributo chamado *Operation*. Um *Operation* é um código que pode ser aplicado em um determinado modelo alvo. Todo elemento *Operation* possui nome, *label*, descrição, um campo para escolher a linguagem que será utilizada e outro para inserir o código a ser aplicado em um modelo alvo. Toda métrica direta possui um elemento *Operation* associado a ela. Um código de *Operation* geralmente possui menções aos elementos do modelo alvo a ser medido. Por exemplo, uma métrica que mede algo relacionado a interesses transversais pode mencionar em seu código o termo *AspectUnit*. Isso torna o código do *Operation* dependente de modelo alvo. No caso de *AspectUnit*, estaria citando um elemento que não existe no KDM original (sem suporte a POA), o que tornaria o código do elemento *Operation* dependente de modelo alvo (instância de um metamodelo específico que estende o KDM original). Por isso, um dos objetivos desse trabalho é desenvolver/definir métricas com códigos parametrizáveis, para que fique independente de nomenclatura de modelo alvo.

A Métrica Coletiva é uma subclasse da Métrica Dimensional e possui como diferencial um atributo chamado *Accumulator*. Uma Métrica Coletiva tem como objetivo medir vários modelos alvo, ou seja, aplicar um *Operation* em vários modelos que representam sistemas ou partes de sistemas e então realizar uma operação matemática por meio de seu atributo *Accumulator*, que pode ser: soma (soma todos os índices coletados durante a aplicação do *Operation* nos modelos alvo), valor máximo (compara os resultados das medições e retorna o maior valor), valor mínimo (compara os índices coletados e retorna o menor valor), média (retorna a média dos resultados obtidos com as medições) e desvio padrão (retorna o desvio padrão dos resultados obtidos).

A Métrica Contadora é também uma subclasse de Métrica Dimensional e é muito semelhante à Métrica Direta, porém, a Métrica Contadora possui uma restrição. Apenas resultados “1” ou “0” podem ser retornados. Para exemplificar o uso de uma métrica contadora, imagine um caso onde o Engenheiro de Modernização queira saber quantos métodos existem em um software. Para isso,

bastaria utilizar uma métrica contadora que faça o reconhecimento de elementos *MethodUnit*. Assim, caso o *Operation* da Métrica Contadora identifique um elemento como sendo *MethodUnit*, ele retorna “1”, caso contrário, ele retorna “0”.

A Métrica Binária se difere das demais porque possui o objetivo de realizar alguma função (operação matemática ou algébrica) com os resultados de duas métricas bases. Ou seja, ao criar uma Métrica Binária, é necessário associá-la à outras duas métricas bases. As métricas bases são métricas que podem ser de quaisquer outros tipos, como Dimensional, Direta e Contadora. Conforme a medição é processada pela ferramenta, a função (atributo *Function*) realiza alguma operação com os dois resultados. Por exemplo: uma métrica binária quer saber qual a área das telas de um *software* (em *pixels*). Para isso, ela pode possuir duas métricas base: uma para coletar a largura da tela e outra para coletar a altura. Nesse caso, a *Function* da Métrica Binária seria multiplicar os resultados das métricas base. Considerando o exemplo, se a Métrica Base 1 retorna {10px, 12px, 30px, 40px} e a Métrica Base 2 retorna {10px, 08px, 05px, 05px}, então a área das telas seriam, respectivamente: 400px, 96px, 150px, 200px.

A Métrica Nomeada (*NamedMeasure*) é um tipo de métrica especial. Nela, o atributo *Name* é obrigatório. O Motivo disso é que a Métrica Nomeada é para ser utilizada em casos que se deseja definir uma métrica muito conhecida na literatura/mercado. Por exemplo, *McCabe’sCyclomaticComplexity* é uma métrica muito difundida, por isso, é possível apenas definir uma Métrica Nomeada com o nome “*McCabe’sCyclomaticComplexity*”, sem precisar especificar demais elementos, como *Scope*, *Trait* e *Category*. A ideia de se utilizar Métricas Nomeadas é que, caso a métrica seja reutilizada, ela seria compreendida apenas pelo seu atributo *Name*.

A Métrica Reescalável (*RescaledMeasure*) pode ser utilizada em casos que se desejam realizar conversões de unidades de medida. Por exemplo: Um Engenheiro de Modernização quer reutilizar a Métrica Binária (citada anteriormente) que coleta a área das telas do sistema utilizando *Pixels* como unidade de medida. Porém, o Engenheiro quer que a métrica colete a área das telas em centímetros cúbicos. Para isso, basta que ele crie uma Métrica Reescalável, associe ela à Métrica Binária supracitada e então escreva uma fórmula (no campo *Formula*) de conversão de unidade: Pixel-para-CM³.

A Métrica de Razão (*Ratio Measure*), subclasse de Métrica Dimensional, tem como objetivo manipular resultados de duas métricas base e obter a razão. Ou seja,

uma *Ratio Measure* precisa ter relação com outras duas métricas, que são chamadas de Métricas Base. A Métrica de Razão irá dividir o Dividendo pelo Divisor e obter a Razão. Essa métrica é bastante semelhante à Métrica Binária, porém, ela possui uma restrição (c*onstraint*) em sua especificação que diz que a operação que ela realizará com o resultado das medições das métricas base será sempre a operação de divisão.

Métrica de *Ranking* é uma subclasse de Métrica Dimensional. Uma Métrica *Ranking* pode ser associada a qualquer métrica derivada de Métrica Dimensional. Para criar uma Métrica *Ranking*, é necessário instanciar também outra classe chamada *RankingInterval*. O *RankingInterval* determina quais intervalos serão rotulados. Por exemplo, podemos definir que um sistema com 1000 linhas de código é “Pequeno”, mas, se ele tiver entre 1001 até 5000, trata-se de um sistema “Médio”, e se ele tiver mais de 5000, então ele é "Grande". Um *Ranking* é um elemento que pode ser associado a qualquer Métrica Dimensional ou qualquer Métrica que seja subclasse de Métrica Dimensional.

Neste trabalho foram modeladas e utilizadas métricas de interesse. Dizer que uma métrica é uma “métrica de interesse” não quer dizer que ela é de um tipo específico, tal como direta, coletiva ou binária. Dizer que uma métrica é uma “métrica de interesse” significa que ela se enquadra na categoria de métricas cujo objetivo é medir índices relativos a interesses transversais. As Métricas de Interesse de Cláudio Sant’Anna (2003), por exemplo, são Métricas que se enquadram na categoria “Métricas de Interesse” (Exclusivas de interesse ou não), mas que, no SMM, podem ser definidas como Métricas Contadoras, já que o objetivo é contar quantos *Operations* (CDO) ou Classes/Interfaces/Aspectos (CDC) implementam um determinado interesse. Então, se as métricas visam contar elementos, podemos dizer que são do tipo Métricas Contadoras (*CountingMeasure*).

O metamodelo SMM, como dito anteriormente, é capaz de representar os resultados de medições por meio de sua classe *Measurement*. Ou seja, os índices coletados ao aplicar uma métrica em um modelo alvo são armazenados em instâncias da classe *Measurement* do metamodelo SMM. A ideia apresentada pelos especificadores do metamodelo SMM é que esses modelos que representam resultados de medições possam ser usados como base de conhecimento para futuras tomadas de decisão e/ou estimativas.

A tela do *MoDisco SMM Editor* mostrada anteriormente para exemplificar uma métrica Dimensional é uma forma amigável de se definir métricas SMM utilizando interface gráfica de usuário (GUI). Porém, por trás dessas interfaces, o que é gerado é um arquivo XMI com todos os elementos criados. A Figura 2.7 ilustra um modelo SMM que representa uma métrica para contar o número de métodos em uma classe e também seu respectivo modelo SMM com o resultado da medição após ser aplicada. A métrica do exemplo (parte superior da Figura) é uma métrica direta que tem como objetivo contabilizar os métodos encontrados em um modelo KDM. Para isso, foi necessário desenvolver um *Operation*. Um *Operation* é um elemento do modelo SMM que armazena um código. O código de um elemento *Operation* pode ser escrito em diversas linguagens, tais como OCL ou *XPath*. Para o exemplo da Figura 2.7, foi utilizada a linguagem OCL (*Object Constraint Language*). "*oclIsTypeOf(code::MethodUnit)*", significa que devem ser identificados somente elementos *MethodUnit* (*MethodUnit* é o elemento do KDM que representa um Método).

Figura 2.7 - Trecho de um modelo SMM que conta o número de métodos em uma classe e seu respectivo modelo com os resultados de uma medição.

2.5 Considerações Finais

O Capítulo 2 abordou a ADM e seus metamodelos KDM e SMM. Foi visto que a ADM surgiu com o intuito de promover um consenso na indústria de modernização de sistemas legado, e para isso foram desenvolvidos (e alguns ainda estão em desenvolvimento) metamodelos padrões que podem representar os artefatos de software envolvidos em um processo de modernização.

Capítulo 3

MEDIÇÕES NO CONTEXTO DE SEPARAÇÃO DE INTERESSES

3.1 Considerações Iniciais

Este trabalho tem como principal objetivo viabilizar a medição de interesses em sistemas representados pelo KDM estejam eles no paradigma OO ou OA. A medição no contexto de separação de interesses visa medir índices no sistema (dentre outros) relacionados com o espalhamento (quando um interesse transversal possui partes de sua implementação espalhadas por vários componentes do *software*) ou entrelaçamento (quando os componentes se dedicam a implementar mais de um interesse, entrelaçando a implementação dos interesses) dos interesses nos diversos componentes do *software*, por isso, o objetivo desse capítulo é introduzir os conceitos da programação orientada a aspectos e também discorrer sobre as principais métricas de interesse encontradas na literatura.

Esse Capítulo está organizado da seguinte maneira: A Seção 3.2 introduz o conceito de interesses transversais e programação orientada a aspectos. A Seção 3.3 explica o conceito de métricas de interesse e as principais métricas de interesse encontradas na literatura. A Seção 3.4 finaliza o Capítulo 3 com algumas considerações finais.

3.2 Interesses Transversais e a Programação Orientada a Aspectos

A Programação Orientada a Objetos (POO) trouxe avanços à Engenharia de *Software* por ser um paradigma que se ajusta muito bem aos problemas de domínio real. Porém, existem certas decisões de projeto que as técnicas OO não são suficientes para capturar. Essas dificuldades conduzem o programador a realizar uma implementação espalhada e entrelaçada (explicação na Seção 3.1), o que piora a manutenção do *software* (Kiczales *et al.*, 1997).

Para suprir as necessidades encontradas na Programação Orientada a Objetos, o conceito de Programação Orientada a Aspectos (POA) foi introduzido por *Kiczales et al.*(1997) no final dos anos 90 com o intuito de modularizar os chamados "interesses transversais". Interesse (*concern*) foi um termo introduzido por *Dijkstra* no ano de 1976 e se refere a dados, operações ou requisitos (funcionais ou não funcionais) de um *software*. Interesses transversais (C*rosscutting Concerns*) são interesses do sistema que entrecortam seus componentes, ou seja, que estão espalhados em vários componentes. A orientação a aspectos visa separar esses interesses e agrupá-los, melhorando a modularidade do sistema, o que consequentemente facilita futuras manutenções.

A modularização de interesses que entrecortam os componentes do sistema pode causar um efeito positivo nas classes e métodos por deixá-los apenas com a codificação referente à sua função.

A compreensão de termos como: Interesse Transversal, Aspecto, Ponto de Junção, Conjunto de Junção, Adendo, Declaração Inter Tipo e Combinação são importantes para a compreensão do Paradigma Orientado a Aspectos e para este projeto, que pretende medir modelos KDM de *softwares* orientados a aspectos.

Um Interesse Transversal (*Crosscutting Concern*) é um requisito do *software*, seja ele funcional ou não funcional, que entrecorta a implementação do sistema, de forma que seu código fique espalhado e/ou entrelaçado em vários componentes do sistema. Esses interesses transversais podem ser modularizados em Aspectos (*Aspects*), que é possível incluir instruções para serem executadas e especificar onde e quando elas serão invocadas.

Uma vez que existem interesses modularizados em aspectos, podem-se definir Conjuntos de Junção (*Pointcuts*) baseando-se nos Pontos de Junção (*Join*

Points) do sistema. Pontos de junção são pontos identificáveis durante a execução de um sistema. Por exemplo: chamadas a métodos, tratamento de exceções, acessos a construtores, etc. Já os Conjuntos de Junção permitem especificar um conjunto de pontos de junção em que deve haver a execução de um código.

O código executado quando o fluxo do programa chega até um Conjunto de Junção é chamado de Adendo (*Advice*). Um Adendo pode ser executado antes, depois, ou mesmo no lugar de um Conjunto de Junção. Os Adendos são construções parecidas com os métodos no paradigma OO.

Outro recurso encontrado na POA é a Declaração Inter-Tipo (*Inter-Type Declaration*). Esse recurso serve para fazer modificações na estrutura de uma classe, introduzindo atributos e/ou métodos (construtores, *getters*, *setters*, heranças e interfaces).

Para exemplificar alguns termos da POA, a Figura 3.1 apresenta o trecho do código de um aspecto responsável pelo interesse de *Logging* em um sistema. Assim como em uma classe, declaramos um aspecto especificando seu modificador de acesso, nesse caso "*public*". Na linha "2", um *Pointcut* é criado, declarando o ponto de junção do programa em que esse *Pointcut* faz referência, que nesse caso é a chamada do método *setNome()*, da classe Cliente. Na linha "4", há um *Advice* que executa algum comando de *logging* após (*after*) a chamada do método *setNome()*.

```
1  public aspect Logging {
2      pointcut clienteAlterado():
3          call(void Cliente.setNome(String));
4      after() returning: clienteAlterado() {
5          ... Código a ser executado
6      }
7
8      ...
```

Figura 3.1- Código de um aspecto responsável pelo *Logging* do sistema

Para auxiliar o desenvolvimento de *softwares* orientados a aspectos, foi criada a linguagem *AspectJ* (Kiczales, 2001). *AspectJ* é uma extensão da linguagem Java para o desenvolvimento de *software* OA. Essa extensão dá suporte a criação de aspectos e seus componentes e realiza a combinação de componentes OO (da linguagem Java) e OA.

3.3 Métricas de interesse

Métricas de *software* podem ser usadas como indicadores dos pontos fracos e fortes de uma abordagem em questão (Sant'Anna *et al.*, 2003). Ao se medir um *software*, pode-se obter índices úteis para atividades da Engenharia de *Software*, como estimativa de custo/esforço e gerenciamento de qualidade (Fenton e Pfleeger, 1997).

Segundo *Chimdambere Kemerer* (1994), no contexto da POO existem muitas possibilidades de projeto para um mesmo *software*, as métricas podem ajudar nas escolhas mais adequadas, auxiliando na redução de custo de desenvolvimento, testes e manutenção ao longo da vida do *software*. De uma forma geral, utilizam-se métricas para melhorar o processo de desenvolvimento de um *software*.

As métricas são mais comumente utilizadas em nível de código, nesse caso o artefato medido é o código-fonte do sistema. No contexto da ADM, a medição é realizada em nível de modelo, por isso, a medição pode ser feita em vários níveis de abstração, podendo ser realizada em modelos PSM (*Platform Specific Model*), PIM (*Platform Independent Model*) ou CIM (*Computation Idependent Model*). Neste trabalho, serão estudadas métricas e formas de medições mais focadas em modelos PSM.

A medição de *software* utilizando como artefato primário o código-fonte do sistema é um assunto antigo e vasto na literatura (*Fenton e Pfleeger*, 1997) (*Chidamber e Kemerer*, 1994), porém, no contexto deste trabalho, é necessário pensar em métricas e medições em que os alvos são artefatos de softwares representados por instâncias de metamodelos da ADM. Modelos esses que representam *softwares* orientados a objetos e *softwares* orientados a aspectos.

As métricas de interesse encontradas na literatura e seus respectivos objetivos estão listados na Tabela 3.1, que possui as seguintes colunas: Nome (nome da métrica de interesse), Sigla da métrica, Autor ou Autores, Objetivo (Breve explicação sobre o objetivo da métrica e seu funcionamento), se a métrica é ou não é modelável com o metamodelo SMM e se a mesma está na CCML. Algumas métricas não são exclusivas de interesses, podendo ser aplicadas também em software OO. Por outro lado, existem métricas que só podem ser aplicadas no contexto da POA.

Tabela 3.1 - Métricas de Interesse encontradas na literatura

Nome	Sigla	Autor	Breve Descrição	Modelável com SMM?	Está na CCML?
Concern Diffusion over Components	CDC	Sant'Anna *et al.*, 2003	Mede o espalhamento de um interesse em componentes do sistema. No contexto da ADM, pode ser aplicada antes e depois do processo de modernização.	Sim	Sim
Concern Diffusion over Operations	CDO		Mede o espalhamento de um interesse em métodos do sistema. No contexto da ADM, pode ser aplicada antes e depois do processo de modernização.	Sim	Sim
Lack of Cohesion in Operations	LCOO		Mede o grau de coesão dos Operations (Métodos e *Advices*) de um componente do sistema. Essa métrica é uma extensão da métrica orientada a objetos LCOM (*Lack of Cohesion in Methods*) (Chidamber; Kemerer,1994). No contexto da ADM, pode ser aplicada antes e depois do processo de modernização.	Sim	Não
Concern Diffusion over Lines of Code	CDLOC		Mede o espalhamento de um interesse sobre as linhas de código do sistema. Essa métrica não é modelável por manipular linhas de código, que não são representadas de forma fiel em modelos KDM.	Não	Não
Disparidade	$DISP_{CF}$	Wong; Gokhale; Horgan, 2000.	Objetivo da métrica é saber o quão próximo está uma *Feature* F de um Componente C. Quanto menor for este grau de aproximação, maior será a disparidade. (Bruno Silva, 2009). Aplicável antes e depois do processo de modernização.	Sim	Sim
Concentração	$CONC_{FC}$		Mede a concentração de blocos que implementam uma *Feature* F em um Componente C.Aplicável antes e depois do processo de modernização.	Sim	Sim
Dedicação	$DEDI_{CF}$		Dedicação "DEDI(C,I)" mede a proporção em que os blocos do componente C estão dedicados à implementação do Interesse I em relação ao conjunto de todos os blocos de C (Bruno Silva, 2009).Aplicável antes e depois do processo de modernização.	Sim	Não
*Concentração**	*CONC(s,t) (CONC Estendida)*	Eaddy *et al*., 2007	Extensão da métrica CONCENTRAÇÃO (Wong *et al* ,2000). Aqui, ao invés de utilizar *Features*, são utilizados Linhas de Código (LOC) de Interesses, e ao invés de se utilizar Blocos, são utilizadas Linhas de Código (LOC) de Componentes.Aplicável antes e depois do processo de modernização.	Sim	Sim
*Dedicação**	*DEDI(t,s) (DEDI estendida)*		Extensão da métrica DEDICAÇÃO (Wong *et al*, 2000). Aqui, ao invés de se utilizar Blocos de código, são utilizados Linhas de Código (LOC) do componente, e, ao invés de se utilizar *Feature*, utiliza-se Linhas de Código (LOC) de um Interesse.Aplicável antes e depois do processo de modernização.	Sim	Sim
Degree of Scattering	DOS		*Degree of Scattering* (Grau de Espalhamento) deriva de (CONC(s,t) e DEDI(t,s,)). DOS mede o	Sim	Não

			grau de espalhamento de um interesse entre os componentes do sistema (Bruno Silva, 2009). Aplicável antes e depois do processo de modernização.		
Degree of Focus	DOF		*Degree of Focus* (Grau de foco) mede o grau de proximidade de um componente em relação aos interesses do sistema, isto é, quão bem separados em componentes estão os interesses em questão (Bruno Silva, 2009).	Sim	Não
Crosscutting Degree of na Aspect	CDA	Ceccato e Tonella, 2004	O objetivo dessa métrica é calcular quantos componentes são afetados por um determinado Aspecto. Se um aspecto entrecorta, de alguma maneira, 5 classes e 3 interfaces, logo, seu índice CDA será 8. Aplicável somente em sistemas orientados a aspectos.	Sim	Não
Number of Features	NOF	Lopez *et al.*, 2007	Mede o número de *Features* em um sistema. Segundo Figueiredo e Cacho (2008), Podemos entender *Feature* como sendo um Interesse.Aplicável antes e depois do processo de modernização.	Sim	Não
Feature Crosscutting Degree	FCD		Mede o número de classes que são afetadas por todos os *advices* de um determinado interesse. Aplicável somente em sistemas orientados a aspectos.	Sim	Não
Advice Crosscutting Degree	ACD		Mede o número de classes afetadas por um único *Advice* específico. Aplicável somente em sistemas orientados a aspectos.	Sim	Não
Size		Ducasse *et al.*, 2006	A técnica *Distribution Map*, desenvolvida por Ducasse *et al.* (2006) para analisar propriedades de sistemas possui dois conceitos; Partição de referência (Classes, Interfaces, Métodos, etc) e Partição de Comparação (Interesses do sistema). A métrica *Size* se baseia nesses conceitos e seu objetivo é contar o número de Partições de Comparação, ou seja, o número de interesses dentro das Partições de Referência.	Sim	Não
Touch			Medea proporção entre o valor da métrica *Size* e o número de subpartes encontradas nas partições de referência do sistema (Bruno Silva, 2009). Ou seja, nessa equação o dividendo é o índice *Size* e o divisor é o número de subpartes encontradas nas partições de referência do sistema que – não – tenham sido previamente contabilizadas pela métrica *Size* cujo papel é o de dividendo.	Sim	Não
Spread			*Spread* conta o número de Partições de Referência que possuem alguma implementação de uma determinada Partição de Comparação (Interesse)	Sim	Não
Focus			Mede a proximidade entre a partição de referência (Classes, Interfaces, Métodos, etc) e a partição de comparação (Interesse). Ou seja, quanto maior o número, maior o foco de uma partição de referência na implementação de uma partição de comparação (interesse).	Sim	Não

3.3.1 MétricasCDO, CDC, CDLOC e LCOO.

O objetivo da métrica CDO (*Concern Diffusion over Operations*) é medir o espalhamento de um interesse sobre os métodos de um sistema. Segundo Sant'Anna *et al.* (2003), nessa métrica, primeiro é contabilizado o número de métodos que contribuem diretamente para a implementação de um interesse, depois, são contados os métodos e *advices* que acessam os métodos contabilizados inicialmente por meio de chamadas, parâmetros formais, tipos de retorno, lançamento de exceções e variáveis locais. Além disso, métodos construtores também são considerados.

A Figura 3.2 exemplifica as métricas CDC e CDO. Trata-se de um diagrama de classes do sistema *Health Watcher*. O *Health Watcher* é um sistema *web* de apoio ao sistema público de saúde ou órgãos de vigilância sanitária. Na Figura 3.2, as linhas escurecidas significam que o item está envolvido na implementação de um interesse. Nesse caso, o interesse em foco é o padrão de projeto *Observer* (Gamma *et al.*, 1995). Ao contar os métodos que tem alguma relação com o padrão de projeto *Observer*, nota-se que o índice CDO é de dezessete (17), ou seja, há dezessete (17) métodos envolvidos na implementação do interesse. Quanto à métrica CDC, seu índice é sete (7), considerando que todos os componentes do diagrama possuem ao menos alguma contribuição para a implementação do interesse.

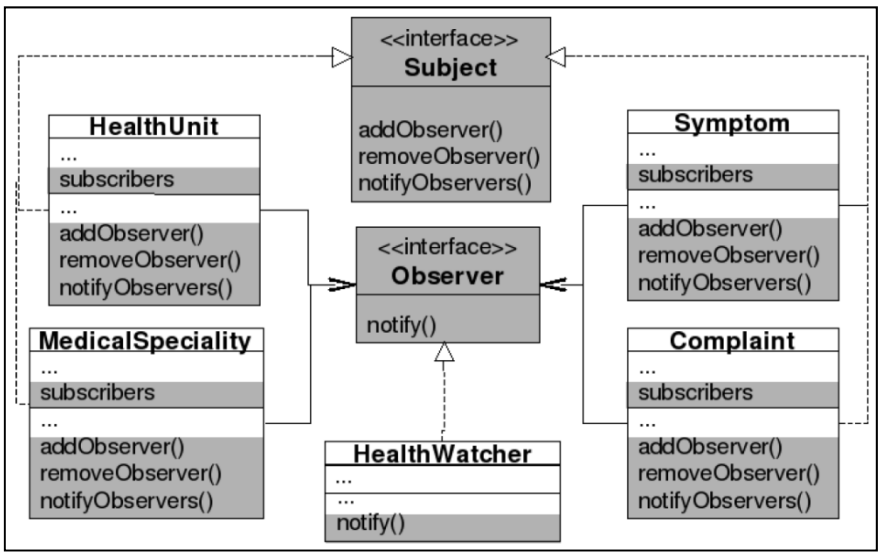

Figura 3.2- Impacto do padrão *Observer*nos métodos e componentes do sistema *Health Watcher* (Silva, 2009)

A CDLOC (*Concern Diffusion over Lines of Code*) contabiliza a alternância em que as linhas de código se dedicam à implementação de um determinado interesse. A Figura 3.3 é um trecho da implementação da classe *HealthUnit.* Nela é possível notar que há oito (8) pontos de alternância de interesse em relação ao padrão *Observer*.

```
public class HealthUnit
                    implements Subject {
  private int code;
  private String description;
  private List specialities;
  private List subscribers = new ArrayList();

     ...

  public void setDescription(String descricao) {
    this.description = descricao;
    notifyObservers();
  }
     ...

  public void addObserver(Observer observer) {
    subscribers.add(observer);
  }

  public void removeObserver(Observer observer) {
    subscribers.remove(observer);
  }

  public void notifyObservers() {
    ...
  }
}
```

Figura 3.3 - Alternância de interesse (Silva, 2009)

LCOO mede a quantidade de pares método/método, método/*advice* ou *advice*/*advice* que não acessam as mesmas instâncias de variáveis. LCOO é uma métrica que mede a falta de coesão (Sant'Anna *et al*., 2003). LCOO estende a métrica LCOM (*Lack of Cohesion in Methods*) (Chidamber e Kemerer, 1994), que é voltada para a orientação a objetos, pois não inclui *advices*.

Para entender a métrica LCOO, imagine o seguinte cenário: considere que um Aspecto possui um método e um *advice* chamados, método_1 e *advice*_1, além de oito atributos, atributo_1 até o atributo_8. Deve-se realizar uma intersecção entre o uso de atributos do par {método_1; *advice*_1} e coletar o resultado. Por exemplo, considere que método_01 utilize os atributos atributo_1, atributo_2, atributo_3 e atributo_4. Considere também que o *advice*_01 utiliza o atributo_5, atributo_6, atributo_7 e atributo_8. Ao fazer uma operação de intersecção, teremos o valor zero.

Isso significa que há baixa coesão, ou seja, não faz sentido o método_1 e o *advice*_1 estarem no mesmo Aspecto, pois provavelmente eles implementam um interesse diferente.

3.3.2 Métricas Disparidade, Concentração e Dedicação.

As métricas Disparidade, Concentração e Dedicação foram idealizadas por Wong; Gokhale; Horgan (2000) e visam medir a disparidade, concentração e dedicação dos blocos do sistema em relação às *features* do sistema. Por BLOCOS, entenda: sequência de linhas de código que não realizam transferência de controle para outro ponto do sistema, ou seja, não realiza chamada para nenhum outro componente ou operação. Por *FEATURE* entenda: um requisito ou um interesse do sistema. Para contextualizar melhor as explicações dessas métricas para o contexto deste trabalho, será utilizado o termo "interesse" ao invés de "*feature*".

Disparidade "DISP(C,I)" mede quantos blocos estão relacionados ao interesse I que estão localizados em um determinado componente chamado de C. Ou seja, o Objetivo da métrica é saber o quão próximo está um interesse I de um Componente C. Quanto menor for este grau de aproximação, maior será a disparidade.

Concentração "CONC(I,C)" mede a proporção de blocos relacionados à um interesse I em um Componente C. Ou seja, mede a quantidade de blocos que implementam um interesse I em um Componente C. Quanto maior o índice, maior a concentração do interesse naquele Componente. Quando menor o índice, menor a concentração do interesse *e* no Componente. A métrica Concentração é o inverso da Disparidade, ou seja, se temos um índice de concentração alto, podemos dizer que a disparidade é baixa.

Dedicação "DEDI(C,I)" mede a proporção em que os blocos do componente C estão dedicados à implementação do Interesse I em relação ao conjunto de todos os blocos de C (Bruno Silva, 2009). Podemos então entender que a métrica Dedicação verifica o grau de proporcionalidade de elementos *SourceRegion* que implementam I em relação aos que não implementam. Logo, se o índice é proporcional, o componente está coeso, caso contrário, ele pode ter interesses transversais em seu corpo.

3.3.3 Concentração*, Dedicação*, DOS, DOF.

A métrica Concentração* (Eaddy *et al.*, 2007) é uma extensão da métrica Concentração proposta por Wong *et al.* (2000). Aqui, ao invés de utilizar *Features* (Interesses), são utilizados Linhas de Código (LOC) que implementam Interesses, e ao invés de se utilizar Blocos, são utilizadas Linhas de Código (LOC) de Componentes. Para se calcular essa métrica, podemos assumir o interesse como sendo a variável "I" e o componente como sendo a variável "C". A fórmula da métrica Concentração* pode ser vista na Figura 3.4.

$$CONC^*(I,C) = \frac{\text{LOC no componente C relacionado ao interesse I}}{\text{LOC relacionadas ao interesse I}}$$

Figura 3.4 - Fórmula da métrica Concentração (Eaddy *et al.*, 2007).

Dedicação* (Figura 3.5) é uma extensão da métrica proposta por Wong *et al.* (2000). Aqui, ao invés de se utilizar Blocos de código, são utilizados Linhas de Código (LOC) de um componente C, e, ao invés de se utilizar *Feature* (Interesse I), utilizam-se Linhas de Código (LOC) de um Interesse, Logo:

$$DEDI^*(C,I) = \frac{\text{LOC no componente C relacionado ao interesse I}}{\text{LOC no componente C}}$$

Figura 3.5 - Fórmula da métrica Dedicação (Eaddy *et al.*, 2007)

DOS (*Degree of Scattering*) ou Grau de Espalhamento, em português, é uma métrica derivada a partir das definições anteriores (CONC*(I,C) e DEDI*(C,I)). DOS mede o grau de espalhamento de um interesse entre os componentes do sistema (Bruno Silva, 2009). O cálculo se dá pela fórmula apresentada na Figura 3.6.

$$DOS(i) = 1 - \frac{|T| \sum_{c}^{C} \left(CONC^*(i,c) - \frac{1}{|C|}\right)^2}{|C| - 1}$$

Figura 3.6 - Fórmula da Métrica DOS (Eaddy *et al.*, 2007)

A média dos resultados de DOS de todos os interesses do sistema é inversamente proporcional ao grau de modularização do mesmo [...] dessa maneira, é desejável um baixo valor na média DOS. (Eaddy *et al.*, 2007apud Bruno Silva, 2009). Os índices de DOS variam de 0 (zero), quando o interesse está totalmente implementado dentro de apenas um componente e 1 (um), quando o interesse está igualmente espalhado entre os componentes do sistema.

Degree of Focus (Grau de foco) mede o grau de proximidade de um componente em relação aos interesses do sistema, isto é, quão bem separados em componentes estão os interesses em questão (Bruno Silva, 2009). O cálculo se dá pela fórmula apresentada na Figura 3.7.

$$DOF(c) = \frac{|S| \sum_{s}^{S} (DEDI^*(c,i) - \frac{1}{|S|})^2}{|S| - 1}$$

Figura 3.7 - Fórmula da Métrica DOF

A média dos resultados DOF de todos os componentes do sistema é diretamente proporcional (Eaddy *et al.*, 2007 apud Bruno Silva, 2009). Os índices DOF variam de 0 (zero), desfocado, ou seja, o componente implementa múltiplos interesses e 1 (um), quando o componente está focado em apenas 1 interesse.

3.3.4 CDA

O objetivo dessa métrica é calcular quantos componentes são afetados por um determinado Aspecto. Se um aspecto entrecorta, de alguma maneira, 5 classes e 3 interfaces, logo, seu índice CDA será 8. É importante lembrar que essa métrica é aplicável somente após a modernização, pois há a necessidade que o sistema já esteja implementado com POA.

3.3.5 NOF, FCD, ACD.

O objetivo da métrica NOF (*Number of Features*) ou, Número de Interesses, é medir o número de Interesses em um sistema. Segundo Figueiredo e Cacho (2008), Podemos entender *Feature* como sendo um Interesse, por isso, neste trabalho, "*feature*" será traduzida para o português como "interesse".

O objetivo da métrica FCD (*Feature Crosscutting Degree*) ou "Grau de Entrelaçamento do Interesse" é medir o número de classes que são afetadas por todos os *advices* de um determinado interesse.

O objetivo da métrica ACD (*Advice Crosscutting Degree*) ou "Grau de Entrelaçamento do *Advice*" é medir o número de classes afetadas por um único *Advice* específico.

3.3.6 *Size, Touch, Spread, Focus.*

Primeiramente, antes de explicar as métricas de Ducasse *et al.*(2006), é preciso entender uma técnica criada pelos autores para auxiliar na medição de interesses: *DistributionMap*.

A técnica *DistributionMap*, serve para analisar propriedades de sistemas e possui dois conceitos; Partição de referência (Classes, Interfaces, Métodos, etc.) e Partição de Comparação (Interesses do sistema). A métrica *Size* se baseia nesses conceitos e seu objetivo é contar o número de Partições de Comparação, ou seja, o número de interesses dentro das Partições de Referência.

A métrica *Touch* define a proporção entre o valor da métrica *Size* e o número de subpartes encontradas nas partições de referência do sistema (Bruno Silva, 2009). Ou seja, nessa equação o dividendo é o índice *Size* e o divisor é o número de subpartes encontradas nas partições de referência do sistema que – não – tenham sido previamente contabilizadas pela métrica *Size* cujo papel é o de dividendo.

Spread conta o número de partições de referência que é tocada por uma determinada propriedade (Bruno Silva, 2009). Ou seja, *Spread* conta o número de Partições de Referência que possuem alguma implementação de uma determinada Partição de Comparação (Interesse).

Focus mede a proximidade entre a partição de referência (Classes, Interfaces, Métodos, etc.) e a partição de comparação (Interesse). Ou seja, quanto maior o número, maior o foco de uma partição de referência na implementação de uma partição de comparação (interesse).

3.4 Considerações Finais

Esse capítulo abordou a questão da medição no contexto da separação de interesses. Foram apresentados conceitos de POA (Programação Orientada a Aspectos) e as principais métricas de interesse encontradas na literatura

As métricas apresentadas na Tabela 3.1 são, em sua maioria, modeláveis por meio do metamodelo SMM. Porém, vale ressaltar que existem algumas adaptações. Esses detalhes podem ser vistos no Capítulo 5 “Definindo Métricas de Interesse com o Metamodelo SMM”.

O Capítulo 5 “AO-KDM: Uma extensão Orientada a Aspectos Para o KDM” irá detalhar a extensão desenvolvida para que o KDM pudesse representar software orientado a aspectos. Isso foi essencial para este trabalho pelo fato de que, para definir e aplicar métricas de interesse no contexto da ADM é necessário que o modelo alvo possa representar tanto o paradigma de desenvolvimento de software orientado a objetos quanto o paradigma orientado a aspectos. Em um cenário de modernização POO para POA, as medições são realizadas em ambos os modelos e paradigmas, o que exige que o KDM consiga representar software orientado a aspectos.

Capítulo 4

TRABALHOS RELACIONADOS

Essa seção apresenta uma síntese do levantamento bibliográfico acerca do tema deste projeto. O levantamento foi realizado nos moldes de um mapeamento sistemático e o alvo foram estudos relacionados à modernização de sistemas que utilizam a ADM. Parte desse mapeamento se dedicou ao contexto de medição de modelos KDM, e são esses estudos os apresentados na Seção 4.3.

4.1 Medição de Interesses no contexto da ADM

Não foram encontrados trabalhos relacionados à medição de *softwares* orientados a aspectos representados pelo KDM por meio do uso de métricas SMM. Em consequência da escassez de abordagens desta natureza, foi feito um mapeamento sistemático de abordagens relacionadas à medição de *software* utilizando os metamodelos SMM e KDM sem especificar demais detalhes. Um resumo desse mapeamento contendo os detalhes das abordagens encontradas é descrito a seguir.

4.2 Mapeamento Sistemático

Para este trabalho, um mapeamento sistemático conjunto (Durelli *et al.,* 2014a) sobre modernização de sistemas legados foi realizada em conjunto com demais pesquisadores do laboratório *AdvanSE* (*Advanced Research on Software*

Engineering), localizado na Universidade Federal de São Carlos, no Departamento de Computação. O mapeamento descrito nesta seção é parte integrante de um mapeamento sistemático conjunto sobre modernização de sistemas legados no contexto da ADM e está focada em abordagens que utilizem o SMM para medir modelos KDM.

Foram realizadas buscas por artigos científicos nas seguintes máquinas de busca: *IEEExplore*, *ACM*, *Scopus*, *Springer*, *WebofScience*, *ScienceDirect* e *EngineeringVillage*. Após definir as máquinas de busca, a seguinte *string* de busca (Figura 4.1) foi utilizada para buscar por artigos relacionados ao tema de estudo geral do trabalho:

```
((Software Metrics Meta-model) OR (Software Metrics Metamodel) OR
(Structured Metrics Meta-Model) OR (Structured Metrics Metamodel) OR
(Abstract Syntax Tree Metamodel) OR (Abstract Syntax Tree Meta-model) OR
(Knowledge Discovery Metamodel) OR (Knowledge-Discovery Metamodel) OR
(Knowledge-Discovery Meta-model) OR (Knowledge Discovery Meta-model) OR
(Architecture Driven Modernization) OR (Architecture-Driven
Modernization) OR (Model Driven Modernization) OR (Model-Driven
Modernization) OR (Model-driven software modernization) OR (ADM Pattern
Recognition specification) OR (ADM Visualization specification) OR (ADM
Refactoring specification) OR (ADM Transformation specification) OR ("KDM
Metamodel) OR (KDM Meta-model)).
```

Figura 4.1 - *String* de busca do mapeamento sistemático

O resultado da primeira busca, somado aos artigos que foram adicionados manualmente na ferramenta de apoio para revisões sistemáticas *Start* (*State of the Artthrough Systematic Review*) somaram 349 artigos. Desses, apenas quatro estão diretamente ligados ao tema deste projeto e foram aprovados nos critérios de inclusão.

A Tabela 4.1 mostra os critérios de inclusão e exclusão utilizados no mapeamento sistemático.

Tabela 4.1 - Critérios de inclusão e exclusão do mapeamento sistemático

Inclusão	Exclusão
Apresenta de alguma maneira uma modernização de sistema legado	Artigo completo indisponível
O estudo primário relata algo sobre ADM	Artigo não está em Português, Inglês ou Espanhol
Descreve a ADM ou ao menos um de seus metamodelos	

Após a leitura dos artigos selecionados e a extração de dados, algumas informações relevantes foram levantadas e são descritas a seguir.

Izquierdo *et al.* (2009), desenvolveram uma ferramenta para o cálculo de métricas dentro do contexto da ADM. Nesta abordagem, como pode ser visto na Figura 4.2, o autor extrai o conhecimento de código legado na linguagem PL/SQL e gera modelos ASTM. Esses modelos ASTM, por sua vez, são submetidos a uma série de transformações M2M (*Model-to-model*) por meio da linguagem *RubyTL*, gerando então um modelo KDM, que é medido automaticamente por modelos SMM. Por fim, para melhor visualização dos resultados, um arquivo CSV (*comma-separeted value*) é gerado para o usuário. No trabalho em questão, nada é comentado sobre representar sobre orientado a aspectos com o metamodelo KDM. A abordagem em questão também não cita métricas de interesse.

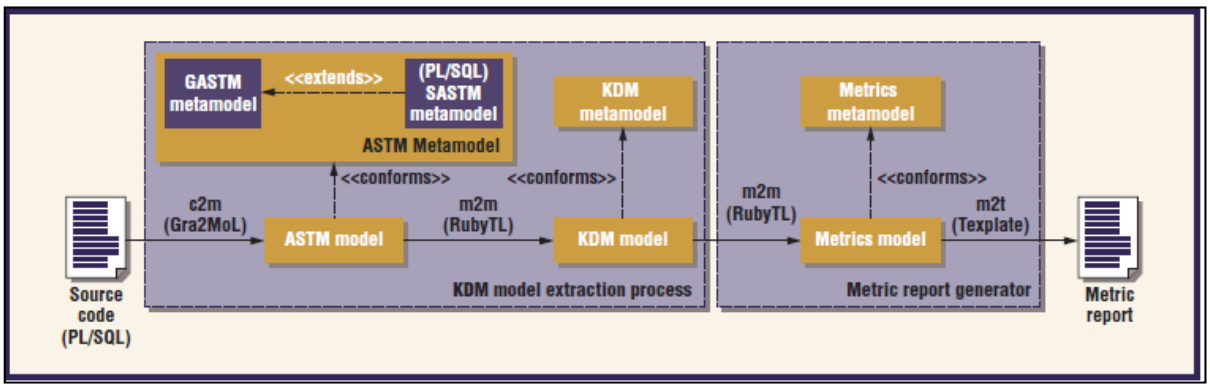

Figura 4.2 - Abordagem de medição de Izquierdo *et al.* (2009)

No mesmo ano, Engelhardt *et al.* (2009) descrevem uma ferramenta chamada *Metrino*, que também é utilizada para aplicação de métricas SMM (métricas essas que não são de interesse e não se aplicam em modelos KDM orientados a aspectos) em modelos KDM. A ferramenta é baseada em regras (*Rules*) escritas na linguagem

OCL que servem como base para a geração de modelos de métricas SMM. A correlação entre regras, modelos e métricas é apresentada na Figura 4.3.

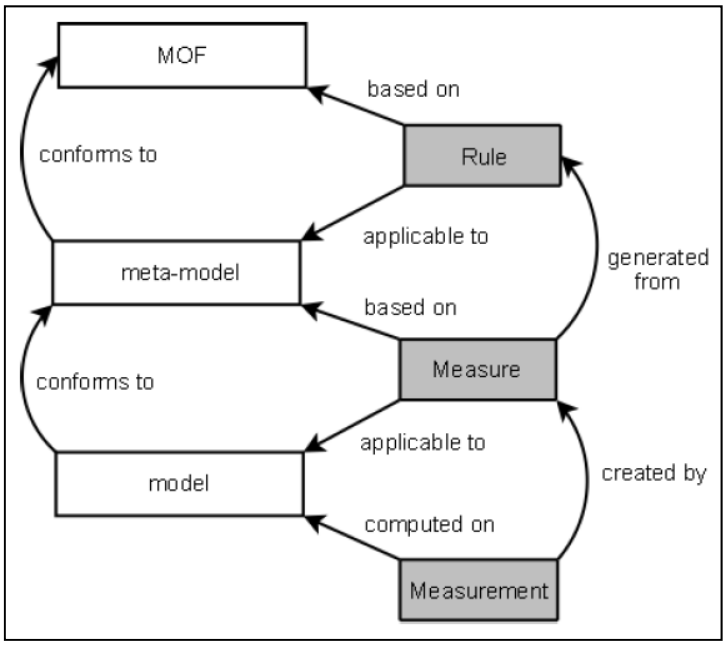

Figura 4.3 - Correlação entre modelos, regras e métricas (Engelhardt *et al.*, 2009)

Conforme visto na Figura 4.3, na abordagem de Engelhardt *et al.* (2009), regras (*rules*) são criadas com base no metamodelo MOF e são aplicáveis em metamodelos em conformidade com o MOF. Métricas (*measures*) são geradas a partir de regras, baseadas em metamodelos em conformidade com o MOF e aplicáveis em modelos em conformidade com metamodelos que estejam em conformidade com o MOF. Medições (*measurements*) são criadas a partir da aplicação das métricas em modelos alvo e são computadas em novos modelos com resultados de medições.

Os autores da ferramenta *Metrino* utilizaram uma estratégia mais amigável para definição de métricas. Essa estratégia foi permitir ao usuário criar modelos de métricas apenas escrevendo código OCL, que, posteriormente é transformado em modelos SMM, por meio de transformação *code-to-model* (código-para-modelo).

A Figura 4.4 ilustra o fluxo da ferramenta *Metrino*, que basicamente acontece em quatro fases: Na primeira fase (*Rule Management*), o usuário pode definir uma regra ou utilizar uma que já foi definida. Na segunda fase (*Measure Generation*), são gerados os modelos de métricas. A terceira fase (*Measure Management*) permite que modificações sejam feitas nos modelos de métricas. Também há a possibilidade do usuário agrupar métricas em conjuntos para que sejam executadas de uma só

vez, unindo assim métricas que avaliam um mesmo aspecto do modelo. Por fim, na quarta fase (*Measure Evaluation*) a ferramenta aplica as métricas selecionadas pelo usuário nos modelos que representam o sistema e então gera novos modelos com os resultados das medições.

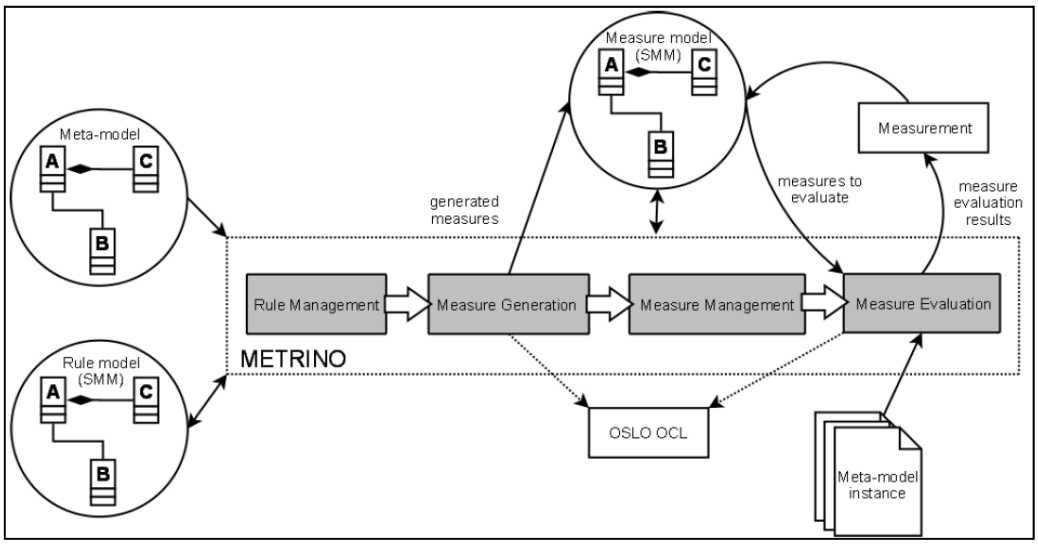

Figura 4.4 - Ferramenta de medição *Metrino* (Engelhardt *et al*., 2009)

Em 2010, uma nova versão da ferramenta de Izquierdo *et al.* foi publicada. Essa versão tem como vantagem um motor de execução mais genérico, diferente do anterior, onde o trabalho foi mais focado para um problema específico de medição, que no caso, eram medições relacionadas à bases de dados.

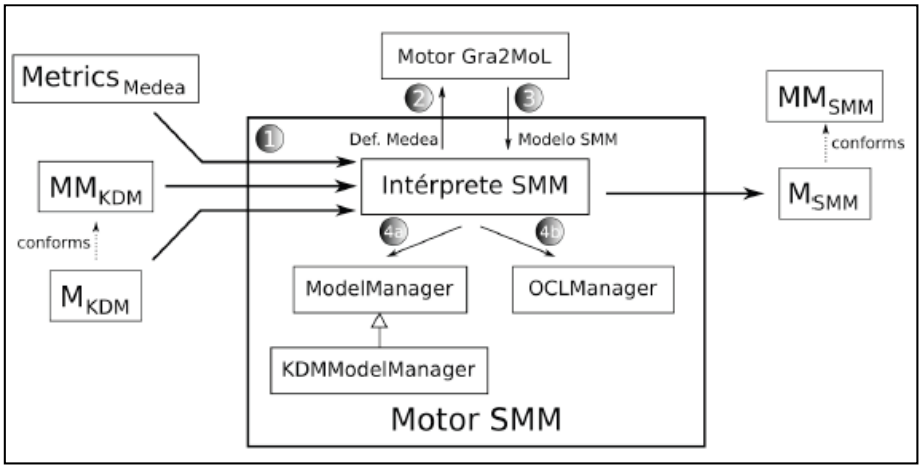

Figura 4.5 - Abordagem de medição de Izquierdo *et al.* (2010)

A Figura 4.5 ilustra a abordagem apresentada por Izquierdo *et al.* (2010). O mecanismo de execução recebe como entrada métricas definidas em uma

linguagem específica de domínio (*DSL - Domain Specific Language*) denominada *Medea,* além de modelos KDM em conformidade com o metamodelo KDM. Depois, o mecanismo interpreta as métricas definidas na linguagem *Medea* e gera modelos SMM que então são aplicados nos modelos KDM, gerando como saída um modelo SMM de resultado de medições.

A publicação mais recente de Izquierdo *et al*. (2010) também não faz nenhuma referência à medição de interesses ou sobre a representação de software orientado a aspectos em modelos KDM.

O artigo mais recente encontrado pela busca foi sobre a abordagem MAMBA (*Measurement Architecture for Model-BasedAnalysis*), (Frey *et al.*, 2012). Nesse artigo, um mecanismo de execução de métricas em modelos *Ecore*/KDM foi apresentado. A ferramenta possui uma linguagem de definição de métricas própria, chamada MDL *(Metrics Definition Language*), e uma linguagem para filtragem de resultados chamada MQL (*Metrics Query Language*). Para esta abordagem, os autores estenderam o metamodelo SMM para adicionar novos recursos ao mesmo.

A Figura 4.6 ilustra o funcionamento da ferramenta MAMBA. A ferramenta requer, como entrada, métricas definidas em modelos SMM ou definidas na linguagem MDL (que depois serão traduzidos para modelos SMM), além disso, métricas auxiliares podem ser definidas em arquivos MQL (que também serão transformados em arquivos SMM). Também é necessário entrar com um arquivo a ser medido, como um modelo KDM, por exemplo. Com os arquivos de entrada, o componente *Measurement Controller* verifica se existe a necessidade de importar dados externos para auxiliar a medição (como um arquivo CSV com resultados de medições manuais) e então o *Execution Engine* aplica as métricas SMM nos modelos KDM, gerando como saída arquivos SMM de medições.

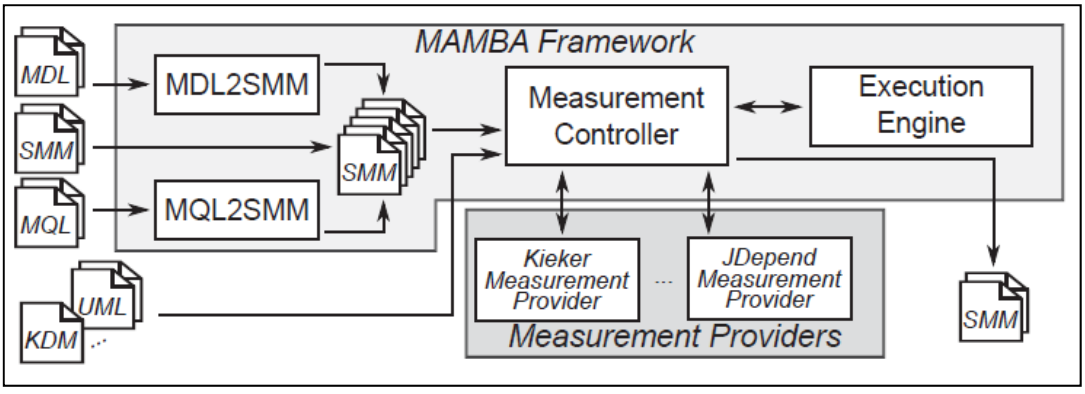

Figura 4.6 - Ferramenta MAMBA - *Measurement Architecture for Model-Based Analysis* (Frey *et al.*, 2012)

Apesar de bem evoluída, a ferramenta MAMBA não possui, ou não foram publicadas menções sobre recursos de medição de interesses em modelos KDM que representem *software* Orientado a Aspectos.

A Tabela 4.2 apresenta um resumo quanto ao uso dos metamodelos nas abordagens citadas. Das quatro abordagens estudadas, todas utilizam o metamodelo KDM original, sem modificações. Uma utiliza o metamodelo SMM estendido e todas possuem mecanismo de execução automatizado. Outro ponto importante em comum nas abordagens é que todas utilizam uma linguagem própria para definição das métricas. Isso ocorre porque definir um modelo de métricas SMM manualmente é muito trabalho e passível de erros.

Tabela 4.2 - Classificação das abordagens quanto ao uso dos metamodelos da ADM

Abordagem/Detalhes	CMEE (2014)	Frey *et al.* (2012)	Izquierdo *et al.* (2009) e (2010)	Engelhardt *et al.* (2009)
Metamodelo utilizado para representar o sistema	KDM/AO-KDM	KDM	KDM	KDM
Metamodelo utilizado para representar as métricas e medições	SMM	SMM	SMM	SMM
Possui mecanismo de execução automatizado?	Sim	Sim	Sim	Sim
Estende o metamodelo KDM?	Sim	Não	Não	Não
Estende o metamodelo SMM?	Não	Sim	Não	Não
Possui linguagem de definição de métricas SMM?	Não	Sim	Sim	Sim
Representa Software Orientado a Aspectos?	Sim	Não	Não	Não
É dependente de anotações prévias no modelo alvo?	Sim	Não	Não	Não

O estudo das abordagens existentes é útil para compreender o estado da arte quanto à aplicação de métricas em modelos da ADM. Nota-se uma carência de trabalhos relacionados a este tema, e a ausência no que diz respeito a este mapeamento sistemático, de trabalhos relacionados à medição de *software* OA utilizando padrões da ADM.

4.3 Trabalhos Relacionados sobre Extensões do KDM

Essa subseção dedica-se a explicar trabalhos relacionados à extensão AO-KDM (Desenvolvida para este projeto). O trabalho de pesquisa mais relacionado à

extensão KDM-AO, é a extensão apresentada no trabalho de Mirshams (Mirshams, 2011). Como também foi feito neste trabalho, a autora criou uma extensão pesada (*HeavyWeight*) do metamodelo KDM para o paradigma orientado a aspectos.

Existem três principais diferenças entre este trabalho e o trabalho de Mirshams. Primeiramente, Mirshams baseou sua extensão em um modelo criado por ela própria, já o AO-KDM foi baseado em um conhecido perfil da UML para a programação orientada a aspectos (Evermann, 2007). O perfil de Evermann engloba todos os conceitos da Orientação a Aspectos utilizados na linguagem *AspectJ* e outras linguagens menos conhecidas, como *AspectC++* e *AspectS* (Evermann, 2007). A segunda diferença é o nível de abstração das extensões. A extensão de Mirshams contém menos elementos que o perfil de Evermann. Isso significa que o AO-KDM está apto a representar tanto uma visão de alto nível (usando as metaclasses mais genéricas), quanto uma visão de baixo nível de abstração. A terceira diferença é que o trabalho de Mirshams é limitado à "*dynamic crosscutting*", pois não há elementos para a representação de *IntertypeDeclarations*. No entanto. Apesar dessas diferenças, a principal similaridade é que o AO-KDM utiliza as mesmas metaclasses que a autora utilizou.

Outra extensão KDM é apresentada por Baresi e Miraz (2011). Eles propuseram uma extensão pesada para apoiar o COMO (Component-Oriented Modernization). COMO é um metamodelo que suporta conceitos tradicionais de uma arquitetura de *software*, permitindo anexar componentes de *software* no KDM. Usando essa extensão é possível substituir ou adicionar partes de um sistema. Ao contrário do que foi feito nesse trabalho, no trabalho de Baresi e Miraz (2011) não usaram um perfil existente como um ponto de partida para criar uma extensão, eles combinaram outro metamodelo ao KDM. O metamodelo COMO estende algumas classes de alto nível do KDM, como *kdm::KDMModel*, *core::KDMEntity*, e*core::KDMRelationship*. Essas classes são a base entre a extensão proposta por esses autores e são um link entre o metamodelo KDM e o metamodelo COMO. A maior similaridade com o AO-KDM é o fato de eles também terem realizado uma extensão pesada do metamodelo KDM. Como maior diferença, pode-se citar o fato de que os autores estenderam apenas elementos de alto nível do KDM, enquanto o AO-KDM estende também elementos de baixo nível, como elementos de código-fonte; *ClassUnit*, *MethodUnit*, *CommentUnit*, etc.

4.4 Considerações Finais

Nota-se que embora existam algumas poucas abordagens relacionadas a este projeto, nenhuma dá suporte à medição de interesses em modelos KDM orientado a aspectos, motivando assim a criação de uma nova abordagem para suprir tal necessidade.

Capítulo 5

AO-KDM: UMA EXTENSÃO ORIENTADA A ASPECTOS PARA O KDM

5.1 Considerações Iniciais

Para a realização desse trabalho foi preciso desenvolver uma extensão do metamodelo KDM. Isso foi necessário para tornar possível a representação de *software* orientado a aspectos em modelos KDM, que, após o processo de modernização, são medidos por métricas de interesse definidas em modelos SMM. Esse capítulo descreve como a extensão foi feita e mostra sua aplicação em um sistema. O Capítulo 5 está organizado na seguinte maneira: a Seção 5.2 explica em detalhes como foi realizada a extensão do metamodelo KDM original, gerando o AO-KDM, além de mostrar um estudo de caso utilizando a extensão AO-KDM. Por fim, a Seção 5.3 apresenta algumas considerações finais sobre o Capítulo 5.

5.2 Extensões Para o Metamodelo KDM e a Extensão AO-KDM

A especificação do KDM (OMG, 2012) originalmente não provê suporte para elementos de *softwares* orientados a aspecto. Segundo Pérez-Castillo *et al.* (2011), o pacote *KDM,* localizado na camada de infraestrutura do metamodelo KDM define

um mecanismo de extensão leve para a adição de novos elementos com o objetivo de estender a semântica em modelos KDM. Por “mecanismo de extensão leve” entende-se que não haverá modificações nas classes do metamodelo, mas sim a utilização das próprias classes existentes marcadas com estereótipos.

Estereótipos são como “anotações” que mudam a semântica de um determinado elemento, ou seja, uma possível forma de extensão do KDM consiste em introduzir estereótipos aos elementos do metamodelo.

A Figura 5.1 mostra um exemplo de extensão leve que modifica a semântica de um elemento do modelo de classe da UML (*Unified Modeling Language*) para que o mesmo passe a representar um Aspecto.

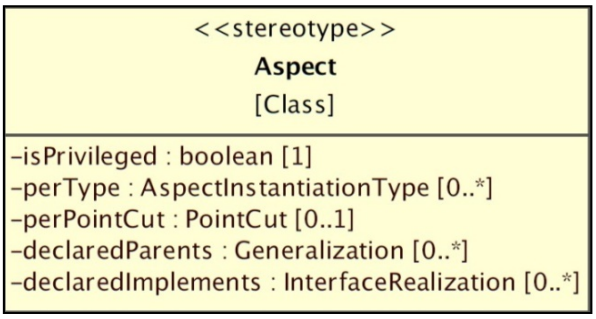

Figura 5.1 - Extensão leve do modelo de classe da UML (Evermann, 2007).

A Figura 5.1 faz parte da proposta de Evermann (2007) que consiste em estender o modelo de classes da UML para que o mesmo possa representar *softwares* orientados a aspectos. Nesse caso, foi criado um estereótipo de *Class* (metaclasse original) chamado “*Aspect*”, mudando assim a semântica da metaclasse original.

Outra forma de se representar *software* orientado à aspectos com o metamodelo KDM é realizando uma extensão “pesada”. Nesse tipo de extensão, que foi escolhida para este trabalho, é necessário criar novas metaclasses para representar os elementos da orientação à aspectos, editando o metamodelo KDM existente. Diferentemente da extensão leve, nessa técnica é necessário alterar o metamodelo original, adicionando novas metaclasses, e não apenas marcá-las com estereótipos.

Para esse projeto, a extensão do metamodelo foi realizada utilizando a tecnica de extensão pesada, ou seja, novas classes foram adicionadas ao

metamodelo KDM para que o mesmo pudesse representar *software* OA. A extensão criada foi batizada com o nome de AO-KDM (*Aspect-Oriented KDM*).

A principal decisão antes da criação da extensão AO-KDM foi escolher um *profile* UML que fosse abrangente o suficiente para representar todos os elementos do paradigma OA. Nesse sentido, foi conduzida uma revisão da literatura para identificar *profiles* UML OA que pudessem ser considerados bons candidatos. Foram analizadas várias propostas (M. Kande *et al.*, 2002) (R.Pawlak *et al.*, 2002) (D. Stein *et al.*, 2002) (M. Basch e A. Sanchez, 2003) (L. Fuentes e P. Sanchez, 2006) (E. Barra, *et al.*, 2004) (J Grundy e R. Patel, 2001) (C. Chavez e C. Lucena, 2002) (H. Yan *et al.*, 2004), mas o *profile* de Evermann foi considerado o mais versátil por incorporar um maior nível de detalhes.

Embora este *profile* tenha sido proposto principalmente para a linguagem *AspectJ*, ele incorpora todos os conceitos genéricos da POA. Observando linguagens orientadas a aspectos, como *AspectJ*, *AspectC++* e *AspectS*, é possível notar que o perfil de Evermann engloba todos os detalhes apresentados nessas principais linguagens, obviamente usando uma terminologia diferente. É como um super conjunto para a programação orientada a aspectos. Isso não é um problema porque, caso seja necessário representar um programa em *AspectS* usando os elementos criatos por Evermann, o único possível problema é que alguns elementos não vão ser preenchidos, pois pode não ser necessário. No entanto, isso é aceitável na filosofia do KDM, uma vez que existe um elemento chamado *ClassUnit* para representar classes, mas também podemos criar instâncias KDM para sistemas processuais. Assim, este tipo de elemento não seria criado.

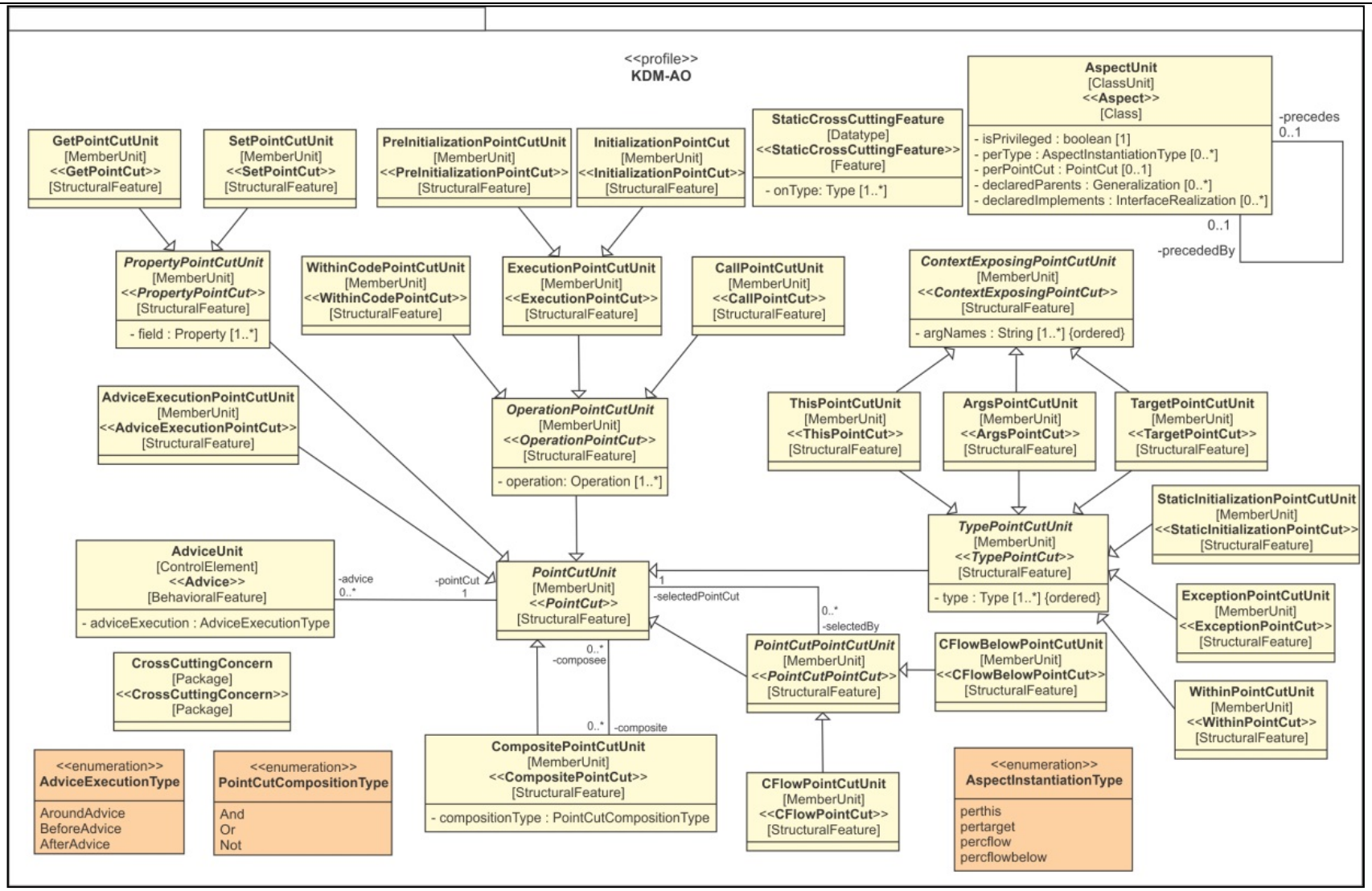
<<profile>>
KDM-AO
AspectUnit
[ClassUnit]
<<Aspect>>
[Class]
- isPrivileged : boolean [1]
- perType : AspectInstantiationType [0..*]
- perPointCut : PointCut [0..1]
- declaredParents : Generalization [0..*]
- declaredImplements : InterfaceRealization [0..*]
-precedes
0..1
0..1
-precededBy
StaticCrossCuttingFeature
[Datatype]
<<StaticCrossCuttingFeature>>
[Feature]
- onType: Type [1..*]
GetPointCutUnit
[MemberUnit]
<<GetPointCut>>
[StructuralFeature]
SetPointCutUnit
[MemberUnit]
<<SetPointCut>>
[StructuralFeature]
PreInitializationPointCutUnit
[MemberUnit]
<<PreInitializationPointCut>>
[StructuralFeature]
InitializationPointCut
[MemberUnit]
<<InitializationPointCut>>
[StructuralFeature]
PropertyPointCutUnit
[MemberUnit]
<<PropertyPointCut>>
[StructuralFeature]
- field : Property [1..*]
WithinCodePointCutUnit
[MemberUnit]
<<WithinCodePointCut>>
[StructuralFeature]
ExecutionPointCutUnit
[MemberUnit]
<<ExecutionPointCut>>
[StructuralFeature]
CallPointCutUnit
[MemberUnit]
<<CallPointCut>>
[StructuralFeature]
ContextExposingPointCutUnit
[MemberUnit]
<<ContextExposingPointCut>>
[StructuralFeature]
- argNames : String [1..*] {ordered}
AdviceExecutionPointCutUnit
[MemberUnit]
<<AdviceExecutionPointCut>>
[StructuralFeature]
OperationPointCutUnit
[MemberUnit]
<<OperationPointCut>>
[StructuralFeature]
- operation: Operation [1..*]
ThisPointCutUnit
[MemberUnit]
<<ThisPointCut>>
[StructuralFeature]
ArgsPointCutUnit
[MemberUnit]
<<ArgsPointCut>>
[StructuralFeature]
TargetPointCutUnit
[MemberUnit]
<<TargetPointCut>>
[StructuralFeature]
StaticInitializationPointCutUnit
[MemberUnit]
<<StaticInitializationPointCut>>
[StructuralFeature]
TypePointCutUnit
[MemberUnit]
<<TypePointCut>>
[StructuralFeature]
- type : Type [1..*] {ordered}
AdviceUnit
[ControlElement]
<<Advice>>
[BehavioralFeature]
- adviceExecution : AdviceExecutionType
-advice
0..*
-pointCut
1
PointCutUnit
[MemberUnit]
<<PointCut>>
[StructuralFeature]
1
-selectedPointCut
0..*
-selectedBy
ExceptionPointCutUnit
[MemberUnit]
<<ExceptionPointCut>>
[StructuralFeature]
CrossCuttingConcern
[Package]
<<CrossCuttingConcern>>
[Package]
PointCutPointCutUnit
[MemberUnit]
<<PointCutPointCut>>
[StructuralFeature]
CFlowBelowPointCutUnit
[MemberUnit]
<<CFlowBelowPointCut>>
[StructuralFeature]
WithinPointCutUnit
[MemberUnit]
<<WithinPointCut>>
[StructuralFeature]
0..*
-composee
0..*
-composite
<<enumeration>>
AdviceExecutionType
AroundAdvice
BeforeAdvice
AfterAdvice
<<enumeration>>
PointCutCompositionType
And
Or
Not
CompositePointCutUnit
[MemberUnit]
<<CompositePointCut>>
[StructuralFeature]
- compositionType : PointCutCompositionType
CFlowPointCutUnit
[MemberUnit]
<<CFlowPointCut>>
[StructuralFeature]
<<enumeration>>
AspectInstantiationType
perthis
pertarget
percflow
percflowbelow

Figura 5.2 - Perfil de Evermann e AO-KDM

A Figura 5.2 mostra tanto o perfil de Evermann quanto a estrutura do AO-KDM. Cada classe/elemento possui quatro palavras no seu primeiro compartimento. A primeira palavra (em negrito) representa o nome da metaclasse criada para a extensão AO-KDM, por exemplo, *AspectUnit*. A segunda palavra, entre colchetes, é a superclasse do KDM que foi estendida pelo elemento do AO-KDM. Por exemplo, a metaclasse *AspectUnit* (AO-KDM) estende a metaclasse *ClassUnit* (KDM). Abaixo dos elementos mencionados, existem mais duas palavras representando o perfil de Evermann. Por exemplo, o estereótipo <<*Aspect*>>, criado por Evermann estende a metaclasse *Class* da UML. Então, seguindo esse padrão, a Figura 5.2 mostra cada elemento/classe tanto para o KDM-AO quanto para o perfil de Evermann, ou um estereótipo, ou uma lista enumerada.

Como mencionado anteriormente, o *profile* proposto por Evermann usa elementos específicos da linguagem *AspectJ*. No entanto, os elementos de alto nível, como *Aspects*, *Advices* e *Pointcuts* são comuns a todas as linguagens de programação Orientada a Aspectos. Outro ponto que também influenciou na decisão pela escolha do *profile* de Evermann foi o fato do mesmo já ter sido revisado e modificado por Thiago Gottard (2011), o que permitiu uma melhor visão de sua construção.

5.2.1 Estendendo o KDM para a Orientaçãoà Aspectos

Nessa seção são apresentados alguns detalhes do KDM-AO, que podem ser vistos na Figura 5.2. O primeiro par de colchetes ([ ]) abaixo do nome de cada elemento exibe o nome da metaclasse do KDM que foi estendida. Um dos maiores desafios ao estender o metamodelo KDM foi saber qual a melhor metaclasse do metamodelo original para estender ao criar o novo elemento.

Como os elementos do perfil de Evermann já foram previamente mapeados para metaclasses da UML (estendendo-as por meio de estereótipos), a maior dificuldade foi identificar metaclasses do KDM que tivessem características similares às metaclasses da UML utilizadas por Evermann. Devido a isso, foi necessário desenvolver um mapeamento entre ambos os metamodelos (UML e KDM), que pode

ser visto na Tabela 5.1. Essa tabela de mapeamento identifica metaclasses do KDM que possuem características similares com metaclasses da UML. Algumas metaclasses podem ser mapeadas de forma direta, como por exemplo a metaclasse Class, da UML, que pode ser facilmente mapeada para a metaclasse *ClassUnit*, do KDM. Ambas as metaclasses apresentam o mesmo objetivo: representar classes em um contexto orientado a objetos. No entanto, como o KDM visa representar mais detalhes de baixo nível que a UML, algumas metaclasses da UML não possuem apenas um candidato ao mapeamento no metamodelo KDM. Esse é o caso da metaclasse *Property*. Esse elemento da UML possui três possíveis metaclasses equivalentes no metamodelo KDM: *StorableUnit*, *ItemUnit* ou *MemberUnit*. *StorableUnit* representa tipos primitivos de variáveis, *ItemUnit* representa registros e vetores e *MemberUnit* representa associações com outras classes. Essa é uma lacuna que ocorre porque o pacote *Code* do KDM está em um nível mais baixo de abstração que a UML.

Tabela 5.1 - Mapeamento UML para KDM

METACLASSE UML	METACLASSE KDM	DIFERENÇAS
Class	*ClassUnit*	A metaclasse *Class* (UML/Pacote *Basics*) possui quatro atributos: *isAbstract*, *ownedProperty[*]*, *ownedOperation[*]* e *superClass*. A metaclasse *ClassUnit*, do pacote *Code* engloba todos esses atributos por meio da metaclasse *AbstractCodeElement*. Uma *ClassUnit* pode ter qualquer atributo cujo tipo é uma concretização de *AbstractCodeElement*, como *StorableUnit*, *MemberUnit*, *ItemUnit*, *MethodUnit*, *CommentUnit*, *KDMRelationships*, etc.
Operation	*MethodUnit*	*Operation* (UML/Pacote *Basics*)é uma metaclasse que representa um elemento comportamental e possui os seguintes atributos: *class* (especifica à que classe ele pertence), *ownedParameter* (parâmetros da operação), e *raisedException* (exceções da operação). A metaclasse *MethodUnit* é ideal para representar um *Operations* porque também representa um elemento comportamental capaz de representar operações definidas nas mais diversas linguagens de programação. A metaclasse *MethodUnit* possui atributos como: *kind* (define o tipo de *Operation*, por exemplo; abstrato, construtor, destrutor, virtual, etc) e *export* (define o modificador de acesso do *Operation*, como; publico, privado e protegido).
Property	*StorableUnit; ItemUnit; MemberUnit*	*Property* (UML), representa variáveis de uma formal geral (local, global, vetores, associações, etc.), enquanto o KDM possui um elemento para cada tipo de *Property*; tipos primitivos (*StorableUnit*), registros e vetores (*ItemUnit*), membros de classes (*MemberUnit*).
Package	*Package*	Um *Package* na UML (Pacote *Basics*) é muito similar à um KDM *Package* (Pacote *Code*). Ambos são containeres para elementos de programa, como classes e outros elementos de código. Um *Package* pode ter uma ou mais classes, e a classe pode ter muitos outros elementos, como métodos, atributos, comentários, etc.
StructuralFeature	*DataElement*	*StructuralFeature* (UML/Pacote *Code::Abstractions*) é uma metaclasse abstrata que pode ser especializada para representar um membro estrutural de uma classe, como um *Property*. O KDM possui a metaclasse *DataElement* (Pacote *Code*), que também pode ser especializada para classes que representam membros estruturais.
BehavioralFeature	*ControlElement*	*BehavioralFeature* (UML/ Pacote *Core::Abstractions*) é uma metaclasse abstrata que pode ser especializada para representar membros comportamentais de uma classe. O KDM possui uma metaclasse abstrata equivalente chamada *ControlElement*, que pode ser especializada para representar elementos “chamáveis”, como por exemplo, métodos.
Parameter	*ParameterUnit*	*Parameter* (UML/Pacote *Core::Abstractions*) é uma metaclasse abstrata para representar o nome e o tipo do elemento que vai ser passado por parâmetro para um elemento comportamental. No KDM, pode-se utilizar a metaclasse *ParameterUnit*. *ParameterUnit* pode representar o nome, tipo e a posição dos parâmetros em uma assinatura, além de permitir o tipo do parâmetro (valor ou referência).
Relationship	*KDMRelationship*	Ambas as metaclasses *Relationship* e *KDMRelationship* são abstratas e podem ser especializadas para representar algum tipo de relacionamento entre dois elementos, como agregação, generalização, etc.

Na Tabela 5.1 é possível ver a relação existente entre as metaclasses e também comentários sobre as mesmas. Como o KDM é um metamodelo mais abrangente que a UML, muitas das relações consideram apenas o pacote *Code* do KDM, pois esse pacote é o único que visa representar classes, atributos, métodos e relacionamentos entre outros elementos com características estáticas. Os outros pacotes do KDM são mais concentrados em detalhes como Interface Gráfica de Usuário, arquitetura e elementos conceituais.

Baseado no mapeamento realizado, o desenvolvimento do KDM-AO foi feito por meio da criação de novas metaclasses para o metamodelo KDM. Uma nova metaclasse foi criada para cada estereótipo presente no perfil de Evermann, mas trocando a metaclasse estendida pela sua equivalente no metamodelo KDM. Por exemplo, se o estereótipo no perfil de Evermann estende a metaclasse *Class*, então, no KDM-AO deverá estender a metaclasse *ClassUnit*. Como pode ser visto na Figura 5.2, os principais elementos orientados a aspectos do perfil de Evermann são representados em um alto nível de classes/estereótipos, são eles: *CrosscuttingConcern*, *Aspect*, *Advice*, *Pointcut* e *StaticCrosscuttingFeature*. Os elementos restantes são subclasses das classes supracitadas. A seguir, serão descritos cada um dos principais elementos que foram criados para a extensão KDM-AO. Como já apresentado anteriormente, a extensão KDM-AO utiliza um padrão de nomes onde os elementos terminam com a palavra *Unit*, como por exemplo: *AspectUnit*, *AdviceUnit* e *PointcutUnit*. Essa foi a maneira usada para diferenciar os elementos do KDM-AO e os elementos do perfil de Evermann. No perfil de Evermann, o elemento *CrosscutingConcern* estende o elemento *Package*, da UML, e visa representar a existência de um interesse transversal, como persistência, segurança, concorrencia, etc. No KDM-AO, esse elemento estende a metaclasse *Package*, do metamodelo KDM original. Essa metaclasse do KDM representa um pacote onde é possível criar Aspectos, Classes e outros elementos.

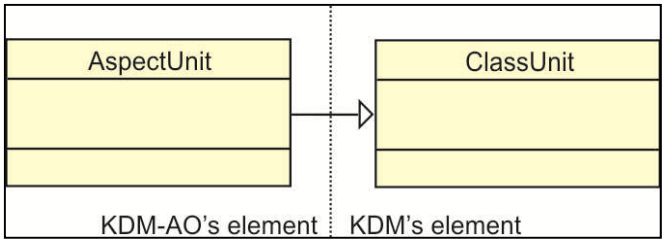

Figura 5.3 - *AspectUnit* herdando da metaclasse *ClassUnit*

AspectUnit é o elemento da extensão para a representação de aspectos, que foi estendido da metaclasse *ClassUnit* (Figura 5.3). A decisão de estender a metaclasse *ClassUnit* se deu por ela possuir todos as características que um Aspecto pode ter, além de poder suportar novos elementos como pointcuts, advices e declarações inter-tipo. Nessa e nas próximas figuras, alguns atributos foram omitidos porque todos eles foram descritos na Figura 5.2.

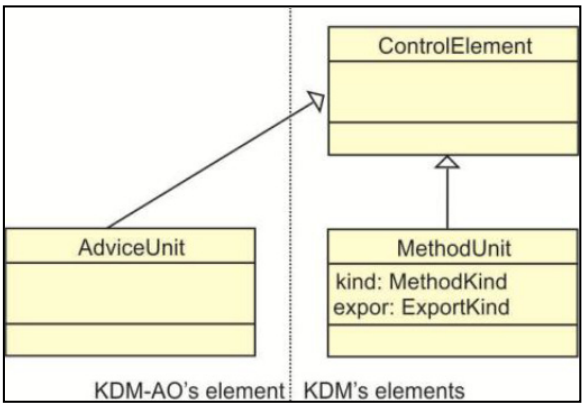

Figura 5.4 - AdviceUnit herdando da metaclasse ControlElement

AdviceUnit herda a metaclasse *ControlElement*, ou seja, *AdviceUnit* herda da superclasse de *MethodUnit*. *AdviceUnit* e *MethodUnit* possuem comportamentos parecidos, ambos possuem blocos de código e comandos de controle relacionados ao comportamento do sistema, por isso, *AdviceUnit* herda de *ControlElement* (Figura 5.4).

PoincutUnit é o elemento para a representação de *pointcuts*. De acordo com o profile de Evermann, um *pointcut* é um elemento estrutural e estende a metaclasse da *StructuralFeature*, da UML. O KDM também possui uma metaclasse para representar características estruturais chamada *DataElement*, que é uma metaclasse abstrata. Suas sub-metaclasses são *StorableUnit*, *MemberUnit* e *ItemUnit*. Como um *pointcut* pode ser abstrato, e as metaclasses *StorableUnit* e *ItemUnit* não podem, *MemberUnit* foi escolhido para ser a super-metaclasse de *PointcutUnit*. Além disso, outro motivo que influenciou na escolha de utilizar o *MemberUnit* como super-metaclasse foi o fato de que *pointcuts* entrecortam outras classes, e o *MemberUnit* é a metaclasse do KDM que é utilizada para referenciar membros de outras classes dentro de uma determinada classe. As relações entre as metaclasses envolvidas nessa discussão podem ser vistas na Figura 5.5.

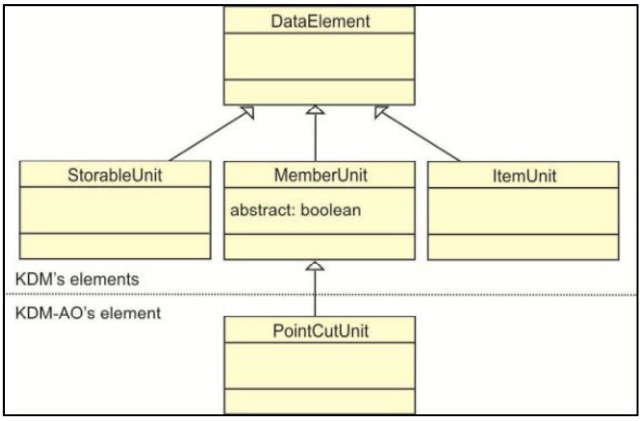

Figura 5.5 - *PointCutUnit* herdando da metaclasse *MemberUnit*

StaticCrosscuttingFeature é o elemento para representar declarações inter-tipo. Na extensão AO-KDM esse elemento pode estender duas meta-classes: *StorableUnit* e MethodUnit. Dessa maneira, *StaticCrossCuttingFeature* é capaz de representar tanto características estruturais quanto características comportamentais. Portanto, uma instância de *StaticCrossCuttingFeature* pode ser um atributo ou um método que será inserido de uma determinada classe (Figura 5.6).

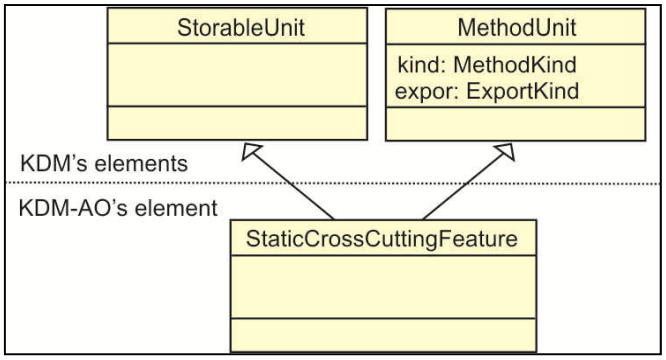

Figura 5.6 - *StaticCrossCuttingFeature* (*Inter-Type Declaration*)

Para a criação do AO-KDM, foi utilizada o Eclipse IDE e o *Framework* EMF (*EclipseModelingFramework*), que permitiu a visualização e edição do metamodelo KDM original, em formato *.ECORE*, disponibilizado pelo OMG em seu website oficial.

Cada metaclasse do profile é representada por um elemento do EMF, como: *Eclass* (Classe), *EEnum* (Lista enumerada), *EPackage* (Pacote), *EAtribute* (Atributo/Propriedade) e *EReference* (Referência). Na Figura 5.7, quase toda classe é representada dentro do metamodelo por um elemento *EClass*. Os elementos

denotados como <<*enumeration*>> são representados por elementos *EEnum*. Os atributos dentro das classes são recriados por elementos *EAttribute* e os relacionamentos entre as classes do perfil são especificadas por elementos *EReference*. A Figura 5.7 mostra a metaclasse *AspectUnit* representada no metamodelo KDM. É possível ver na parte A da Figura 5.7, os atributos da metaclasse (*isPrivileged*, *perType*, *perPointCut*, *declaredParents* e *declaredImplements*) e os relacionamentos (*precedes* e *precededBy*). A parte B mostra a metaclasse já introduzida no metamodelo KDM, assim como todos os seus atributos.

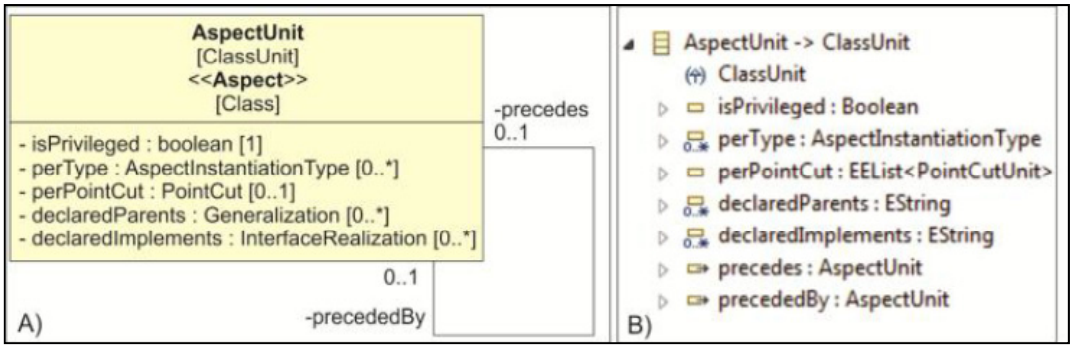

Figura 5.7 - KDM-AO na visão do Eclipse Modeling Framework (EMF)

A cada novo elemento adicionado, há um conjunto de propriedades que podem ter valores padrão ou que precisam ser preenchidas. Por exemplo, quando adicionada uma nova *EClass*, as principais propriedades que devem ser informadas são: *Name* e *ESuperTypes* (super classes herdadas pelo novo elemento). Na Figura 5.8 são mostradas as propriedades pertencentes à metaclasse *AspectUnit*. Após a criação de todas as novas metaclasses no metamodelo KDM, o *plug-in* KDM-AO foi gerado, permitindo a criação de instâncias orientadas a aspectos do metamodelo KDM.

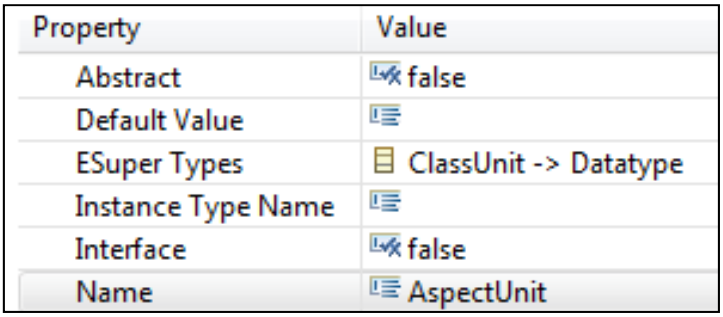

Figura 5.8 - Propriedades do AspectUnit

5.2.2 Estudo de Caso: Representando uma Aplicação Real com o KDM-AO

Nessa seção é apresentado um estudo caso mostrando que o KDM-AO pode ser usado para representar uma aplicação real, ou seja, o KDM-AO pode ser utilizado em um processo de modernização apoiado pela ADM, nesse caso, um processo de modernização envolvendo um *Framework* Transversal de Persistência (Camargo e Masiero, 2005). FTs (*Frameworks* Transversais) são *frameworks* orientados a aspecto que encapsulam um interesse de uma forma genérica, como persistência, segurança e criptografia (Camargo e Masiero, 2005). FTs são compostos por aspectos e classes (concretos e abstratos). A maioria deles dependem fortemente de declarações *intertype*e expressões idiomáticas orientadas a aspectos bem conhecidos como *Container* Introdução e marcador de (Hanenberg, 2000). O cenário de modernização que será utilizado aqui considera a existência de: i) uma instância do KDM que represente um sistema legado (aqui chamado de “KDM Legado” ou "Modelo Base"), que precisa ser modernizado; ii) uma ou mais instâncias do KDM que representem FTs, disponíveis em um repositório; e iii) uma ou mais instâncias do KDM que representem os elementos de instanciação, ou seja, classes e aspectos concretos criados pelo engenheiro de aplicação para acoplar o FT ao código base ou modelo base (nesse caso, KDM Legado).

Nesse estudo de caso, foi modernizado um sistema de gerenciamento de uma loja de CD/DVD. O objetivo da modernização foi modularizar o interesse de Persistência utilizando Aspectos. Como os membros do laboratório AdvanSE possuem experiência com *Frameworks* Transversais, a ideia foi usar um FT de Persistência previamente desenvolvido nesse processo. Fazendo isso, validamos o KDM-AO para a representação de Aspectos e também para a representação de *Frameworks* Transversais. O foco dessa seção e desse estudo de caso é mostrar que é possível representar conceitos da POA com a extensão do KDM que foi desenvolvida. Está fora do escopo minerar o KDM Legado a procura de interesses transversais, removê-los ou mesmo prover uma ferramenta que facilita o acoplamento de FTs.

Portanto, o primeiro passo foi obter uma instância do KDM representando um sistema gerenciador de loja de CDC/DVD. Isso foi feito utilizando o MoDisco *Framework* (Bruneliere *et al.,* 2010), que possui um *parser* que transforma

automaticamente código-fonte Java em instâncias do metamodelo KDM, no formato .XMI. O segundo passo foi obter uma instância do KDM-AO (Metamodelo Estendido) que representasse o *Framework* Transversal de Persistência. Isso foi feito utilizando o *plug-in* desenvolvido a partir da extensão KDM-AO, onde classes e aspectos foram convertidos em modelos KDM que representam o FT.

A seguir, serão mostrados somente os principais Aspectos do FT e os principais elementos existentes no KDM-AO, ou seja, os novos elementos que foram inseridos no metamodelo KDM.

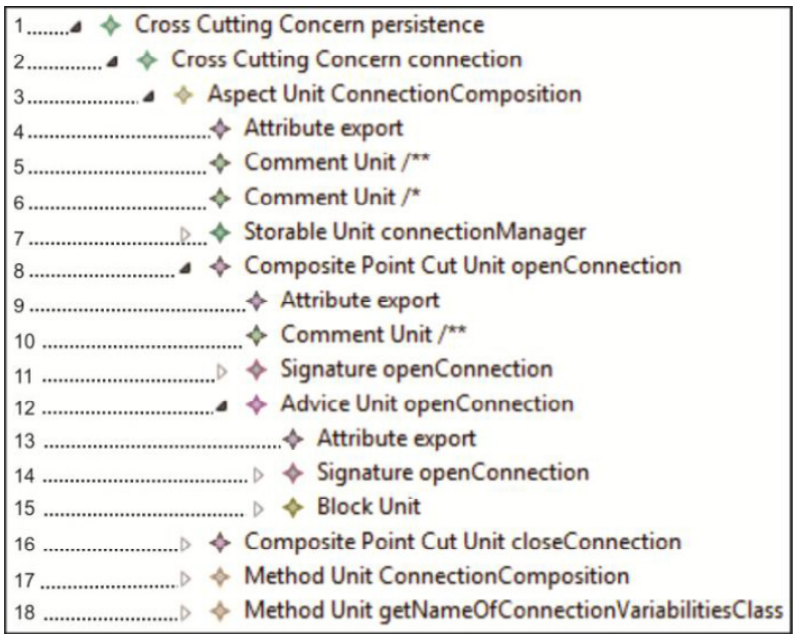

Figura 5.9 - *ConnectionComposition.aj* no KDM-AO

A Figura 5.9 apresenta um trecho de um modelo desenvolvido manualmente com a ferramenta Eclipse Modeling Framework. Na Figura 5.9 é possível observar um Aspecto abstrato do FT chamado *connectionComposition.aj*, que está localizado dentro do pacote "*persistence.connection*". Na extensão desenolvida, um elemento "Cross Cutting Concern" representa um pacote para elementos da POA. O propósito desse *Aspect* é prover uma base comportamental para a abertura e fechamento de conexão com base de dados. Durante a instanciação, é preciso prover implementações concretas para os pointcuts abstratos *openConnection()* e *closeConnection()*. Esse Aspecto tem em seu corpo um atributo, dois *pointcuts* abstratos, um *pointcut* concreto, um método abstrato e dois *advices*.

A visualização mostrada na Figura 5.9 é possível por causa de uma ferramenta existente chamada KDM-SDK Plugin, que permite ao usuário criar/editar modelos XMI em conformidade com o metamodelo KDM (KDM Guide, 2011).

No entanto, também é possível visualizar os arquivos gerados pelas ferramentas KDM-SDK *Plugin* e KDM-AO *Plugin* como um arquivo XMI comum.

Cada linha na Figura 5.9 contém um tipo de elemento e seu valor. Por exemplo, na primeira linha é possível visualizar a existência de um elemento *CrossCuttingConcern* cujo valor é persistence, ou seja, essa é uma instância da metaclasse *CrossCuttingConcern*. A linha 3 exibe o nome de um Aspecto modelado da seguinte forma: Inicialmente o tipo (*AspectUnit*), então, seu valor (*ConnectionComposition*). O elemento "*Attributeexport*" (linha 4) é usado para armazenar a visibilidade (público, privado, protegido). Esse elemento é usado em classes, aspectos, métodos, *pointcuts*, *advices* e outros elementos que permitem esse tipo de sentença. O elemento *StorableUnit* (linha 7) é usado para declarar variáveis e *PointCutCompositeUnit* (linhas 8 e 16) é usado para representar *pointcuts* concretos ou abstratos de um aspecto.

O elemento Signature (linha 11) recebe o mesmo nome que o elemento Signature (linha 14) e tem a função de armazenar os parâmetros que são passados aos *Pointcuts*, *MethodseAdvices*. *AdviceUnit* (linha 12) representa um *Advice* que foi declarado no Aspecto. Na Figura 5.9, o elemento *BlockUnit* (linha 15) é o corpo do *advice*e pode representar trechos de código como *try/catch*, entre outros. *CommentUnit* (linhas 5, 6 e 10) armazena comentários que tenham sido feito no código fonte e *MethodUnit* (linhas 17 e 18) permite a representação de métodos dentro de um Aspecto.

É importante dizer que um código-fonte em *AspectJ* consiste de código Java com elementos da POA. Elementos como *MethodUnit*, *CommentUnit*, *Signature*, *BlockUnit* e *AttributeExport* já existem no metamodelo KDM e estão sendo usados para fazer a representação dos elementos comuns da linguagem Java dentro do Aspecto.

```
1  <codeElement xsi:type="code:AspectUnit" name="OORelationalMapping">
2    <attribute tag="export" value="public abstract"/>
3    <attribute tag="export" value="private"/>
4    <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="" ext="">
5      <ownedElement xsi:type="code:StorableUnit" name="tableName" type="/0/@model.0/@codeE
6        <attribute tag="export" value="public"/>
7        <ownedRelation xsi:type="code:HasValue" to="/0/@model.1/@codeElement.42"/>
8      </ownedElement>
9      <onType>PersistentRoot</onType>
10   </ownedElement>
11   <ownedElement xsi:type="code:StaticCrossCuttingFeature" name="">
12     <ownedElement xsi:type="code:MethodUnit" name="getID" type="/0/@model.0/@codeElement
13       <attribute tag="export" value="public abstract"/>
14       <codeElement xsi:type="code:Signature" name="getID">
15         <parameterUnit type="/0/@model.0/@codeElement.2/@codeElement.0" kind="return"/>
16       </codeElement>
17     </ownedElement>
18     <onType>PersistentRoot</onType>
19   </ownedElement>
```

Figura 5.10 - Um trecho do aspecto *OORelationalMapping.aj* no formato XMI

Na Figura 5.10 é mostrado o Aspecto *OORelationalMapping* como uma instância do KDM-AO, no formato XMI.Esse Aspecto visa introduzir (via Declaração Inter-tipo) dezenas de métodos de persistência em classes de persistência da aplicação. Na linha 1, há uma declaração do*OORelationaMapping*, que é um *AspectUnit*. Dentro dele, existem dois *IntertypeDeclarations* dentro do elemento *StaticCrossCuttingFeature*, (linhas 4 e 11). Esse tipo de declaração permite alguém inserir Atributos/Propriedades e Métodos/Operações em outros elementos, como Interfaces, Aspectos e Classes preenchendo os valores no atributo *onType* (linhas 9 e 18). O primeiro *StaticCrossCuttingFeature* (linha 4) que aparece está inserindo um *StorableUnit* (linha 5) chamado *tableName* na *InterfacePersistentRoot*. O segundo (linha 11), está inserindo um *MethodUnit* (linha 12) chamado *getID* na mesma interface supracitada (*PersistentRoot*). Na Figura 5.11 está o código-fonte equivalente ao modelo escrito na linguagem *AspectJ*.

As Figuras 5.9 e 5.10, mostraram que é possível representar e armazenar instâncias do KDM que representam Aspectos de um FT ou Aspectos de uma aplicação convencional. Outra atividade essencial durante o reuso do FT é a realização do processo de instanciação e acoplamento do FT ao código-base da aplicação. Isso foi feito especializando Métodos e *Poincuts* concretos.

```
public abstract aspect OORelationalMapping {

   public String PersistentRoot.tableName = "";
```

Figura 5.11 - Um trecho do código-fonte do Aspecto *OORelationalMapping.aj*

Nesse estudo de caso, foi necessário criar quatro aspectos concretos e uma classe para realizar o acoplamento do FT ao sistema de gerenciamento da loja de CD/DVD. Os aspectos criados foram *MyOORelationalMapping* (Figura 5.11), *MyConnectionCompositionRules*, *MyDirty* e *MyAspect* e a classe criada foi *MyConnectionVariabilities*. A classe *myConnectionsVariabilities* armazena informação sobre a base de dados, o aspecto *MyOORelationalMapping* declara classes da aplicação base que devem receber métodos de persistencia, o aspecto *myConnectionCompositionRules* especifica os pontos de abertura e fechamento de conexão à base de dados. Finalmente, o aspecto *myDirty* e *myAspect* que são abstratos e estendem aspectos do FT. A Figura 5.12 mostra o aspecto *MyOORelationalMapping* (linhas 1, 2 e 3), cujos nomes podem ser vistos na linha 2. Dentro do aspecto criado pelo engenheiro de aplicação existem sentenças "*declareparents*" informando as classes da aplicação que devem estender uma interface no FT chamada *PersistentRoot*. Isso é feito para que todas as classes possam receber operações de persistência definidos nessa interface.

```
1  <codeElement xsi:type="code:AspectUnit"
2                  name="MyOORelationalMapping"
3                  isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedRelation xsi:type="code:Imports"
6                      to="CrossCuttingConcern persistence"
7                      from="AspectUnit MyOORelationalMapping"/>
8    <ownedRelation xsi:type="code:Imports"
9                      to="CrossCuttingConcern application"
10                     from="AspectUnit MyOORelationalMapping"/>
11   <ownedRelation xsi:type="code:Extends"
12                     to="AspectUnit OORelationalMapping"
13                     from="AspectUnit MyOORelationalMapping"/>
14   <ownedRelation xsi:type="code:InterfaceRealization"
15                     to="InterfaceUnit PersistentRoot"
16                     from="ClassUnit Music"
17                     Name="Music-PersistentRoot"/>
18   <ownedRelation xsi:type="code:InterfaceRealization"
19                     to="InterfaceUnit PersistentRoot"
20                     from="ClassUnit Purchase"
21                     Name="Purchase-PersistentRoot"/>
22   <declaredParents>Music-PersistentRoot</declaredParents>
23   <declaredParents>Purchase-PersistentRoot</declaredParents>
24 </codeElement>
```

Figura 5.12 - Trecho do arquivo XMI do aspecto *MyOORelationalMapping.aj*

Das linhas 5 a 7 e 8 a 10 são mostrados dois *Imports*. O primeiro *import* faz referência ao Pacote do FT de persistência e o segundo faz referência ao pacote da aplicação base. As linhas 11, 12 e 13 representam o elemento *Extends*, que armazena informaçãoes referentes à de herança existente entre o aspecto *MyOORelationalMapping* e *OORelationalMapping*, presentes no FT. Nas linhas 22 e 23 desse aspecto pode ser visualizado o uso do elemento "*declareParents*", esse elemento armazena o nome do relacionamento entre a classe da aplicação base e o aspecto ou interface do FT. Para especificar esse relacionamento em um modelo, é necessário usar um elemento que pode armazenar o nome da classe da aplicação base, o nome do aspecto/interface do FT e o nome do relacionamento entre eles. No KDM, o elemento capaz de fazer essa representação é o "*Implements*", no entanto, ele não tem o atributo *name*, portanto, foi necessário estender esse elemento, adicionando o atributo "*name*". O novo elemento criado a partir do elemento "*Implements*" foi chamado de *InterfaceRealization*.

As linhas 14, 15, 16 e 17 representam uma instância do elemento *InterfaceRealization* onde, na linha 15 é mostrada a interface *PersistentRoot*, do FT. A linha 16 mostra a classe *Music* da aplicação base e o nome do relacionamento

existente entre eles pode ser visto na linha 17. Outra instância do *InterfaceRealization* pode ser vista entre as linhas 18 a 21. O segundo trecho no código-fonte de instanciação a ser visto são os *pointcuts* que abrem e fecham a conexão com a base de dados, presentes no aspecto *myConnectionCompositionRules*. Na Figura 5.13 é mostrado um trecho de um arquivo XMI que contém o elemento *pointCutopenConnection*, onde é possível ver seus principais elementos, como *CompositePointCutUnit*, *ExecutionPointCutUnit* e *ParameterUnit*. O elemento *CompostitePointCutUnit* (linha 5) é a encapsulação de todos os *pointcuts* que representam o *openConnection* (linha 6). O *ExecutionPointCut* (Linha 9) é o *pointcut* que entrecorta o método principal (linha 13) da classe *FindSomeCDsclass* (Linha 11). Por fim, o *ParameterUnit* (linhas 16 a 19) armazenam os parâmetros do *pointCut*.

```
1  <ownedElement xsi:type="code:AspectUnit"
2                name="myConnectionCompositionRules"
3                isAbstract="false">
4    <attribute tag="export" value="public"/>
5    <ownedElement xsi:type="code:CompositePointCutUnit"
6                  name="openConnection"
7                  compositeType="OR">
8      <attribute tag="export" value="public"/>
9      <ownedElement xsi:type="code:ExecutionPointCutUnit"
10                   type="//@model.0/@codeElement.1/@codeEleme
11       <codeElement xsi:type="code:ClassUnit" name="FindSomeCDs">
12         <attribute tag="export" value="public"/>
13         <codeElement xsi:type="code:MethodUnit" name="main">
14           <attribute     tag="export" value="public static"/>
15           <codeElement xsi:type="code:Signature" name="main">
16             <ownedElement xsi:type="code:ParameterUnit"
17                           type="//@model.0/@codeElement.1/@c
18                           kind="return"/>
19             <parameterUnit name=".." ext="" kind="unknown"/>
```

Figura 5.13 - Um trecho do aspecto MyConnectionCompositionRules.aj em formato XMI

Com a realização deste estudo de caso foi possível determinar a adaptabilidade da extensão desenvolvida para representar as mais importantes características e especificidades de uma aplicação implementada em *AspectJ*, nesse caso, um *Framework* Transversal implementado em *AspectJ*. Para representar um *pointCut* genérico com a extensão AO-KDM, é necessário somente criar uma

instância da metaclasse *OperationPointCutUnit* e informar os parâmetros que entrecortam o sistema base. Mas, se for necessário instânciar um *pointCut* mais específico, também é possível, pois o mesmo possui campos para a especificação de detalhes.

Outro exemplo é o fluxo de controle de um *JoinPoint*. Um *control-flowpointcut* sempre especifica outro pointcut como seu argumento. Existem dois tipos de *control-flowpointcuts*, e na extensão AO-KDM eles são representados pelos elementos *CFlowPointCutUnit* e *CFlowBelowPointCut*. O primeiro pointcut captura todos os *OperationPointCutUnit* no *control-flow* do PointCutUnit especificado, incluindo o *OperationPointCutUnitmatching* o *PointCutUnititself*. O segundo *PointCutUnit* exclui o *OperationPointCutUnit* no *PointCutUnit* especificado (Laddad, 2003). A Figura 5.14 mostra como os *PointCutUnits* mencionados podem ser representados utilizando o KDM-AO. As linhas 2 e 3 representam *GetPointCutUnit* e *SetPointCutUnit*, representando que o campo *accountBalance* vai ser entrecurtado quando lido ou escrito. O *CompositePointCutUnit* (linhas 4 e 7) encapsula os *PointCutUnits*, permitindo o engenheiro de aplicação especificar os pontos do sistemas base que vão ser afetados pelo *PointCutUnits*. A linha 6 mostra o *CallPointCutUnit* que é modificado por um *CFlowPointCutUnit* (linha 5) ea linha 9 mostra um *ExecutionPointCutUnit* que é modificado por um *CFlowBelowPointCutUnit* (linha 8).

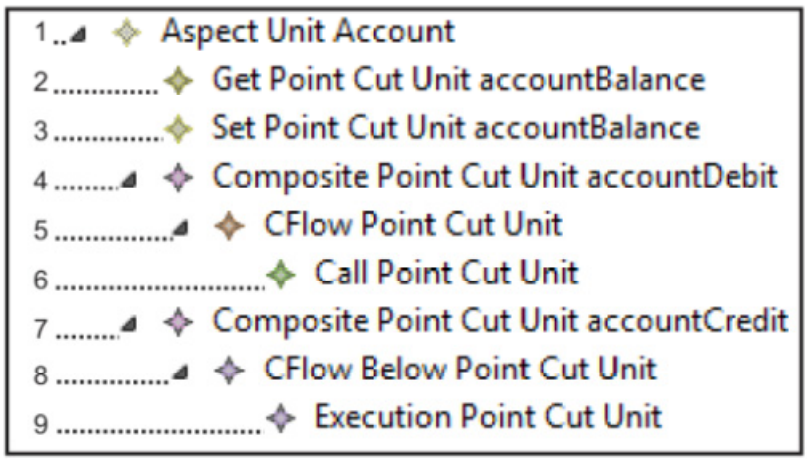

Figura 5.14 - Exemplo de especificação de baixo nível utilizando o AO-KDM Plug-in

Existem outras representações possíveis de *pointcuts* suportadas pela extensão, e o nível de detalhes da instância do AO-KDM depende principalmente do engenheiro de aplicação e do *parser*, que criam a instância.

5.3 Considerações Finais

Como comentado anteriormente, uma extensão pesada pode mudar o metamodelo original, ou simplesmente adicionar novas metaclasses. Ao criar o AO-KDM, optou-se pela segunda opção, pois manter a base original do metamodelo facilita o reuso da extensão em outros contextos. Além disso, com extensões pesadas é possível aumentar o nível de consistência na definição de modelos.

Com exceção do trabalho de Mirsham (2011), não foi encontrado qualquer outro trabalho que tenha estendido o KDM para o paradigma Orientado a Aspectos. Sua extensão também é uma extensão pesada, no entanto, ela não permite a representação de conceitos de baixo nível. Outra contribuição dessa extensão é mostrar um mapeamento preliminar entre metaclasses da UML e do KDM, que pode ser usada para converter/transformar perfis UML em extensões do KDM. No caso do KDM-AO, transformou-se um *profile* UML em uma extensão pesada orientada a aspectos do KDM. Portanto, considerando o mapeamento mostrado na Tabela 4.1, qualquer perfil UML pode ser transformado. Isso é útil porque em ambientes orientados a modelos (*Model-Driven Environments*), sistemas que são representados como instâncias KDM vão precisar ser mostrados como diagramas de classes, por exemplo. Nesse caso, seriam necessários os mesmos conceitos em ambos os modelos. Embora esse estudo de caso do AO-KDM tenha apenas mostrado a capacidade do AO-KDM de representar código-fonte existente OA (engenharia reversa), isso pode também ser usado na engenharia avante, para a geração de código-fonte. Nesse caso, o AO-KDM seria mais apropriado que a extensão de Mirsham, já que o AO-KDM possui maior número de detalhes de baixo nível.

Capítulo 6

DEFINIÇÃO DE MÉTRICAS DE INTERESSE UTILIZANDO O METAMODELO SMM

6.1 Considerações Iniciais

O foco neste capítulo é discutir como definir uma métrica de interesse utilizando o metamodelo SMM. É mostrado também como modelar algumas das métricas de interesse apresentadas no Capítulo 3, além de comentar sobre as métricas que possuem limitações e as que não são passíveis de modelagem utilizando o metamodelo SMM.

6.2 Definições de Métricas de Interesse com o metamodelo SMM

Em geral, o conceito de métricas para modelos é entendido simplesmente pela definição de códigos OCL que são aplicados sobre modelos. Entretanto, métricas SMM possuem um conjunto mais rico de informações, descrevendo detalhes da métrica, seu escopo, categoria, unidade de medida e a operação a ser realizada, que pode ser um código OCL, *XPath* ou algum outro código em linguagem compatível com o propósito.

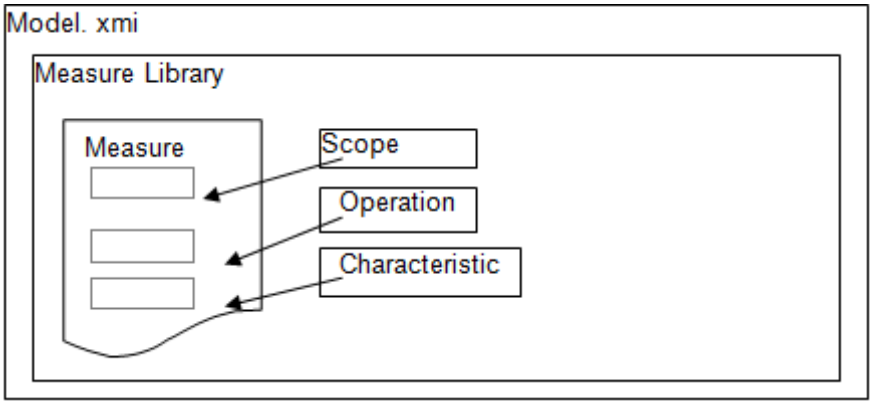

Figura 6.1 - Estrutura de uma métrica SMM

As métricas SMM são estruturadas como mostrado na Figura 6.1. É possível observar que uma métrica SMM precisa estar dentro de um elemento *MeasureLibrary* (Biblioteca de métricas). Uma *MeasureLibrary* pode possuir uma ou mais métricas, geralmente uma biblioteca possui métricas com características em comum, mas isso não é algo obrigatório.

Dentro de um elemento *MeasureLibrary*, o desenvolvedor da métrica deve definir todos os elementos que uma métrica deve possuir, são eles: *Scope* (Escopo da métrica, ou seja, onde ela será aplicada), *Characteristic* (Que tipo de característica ela mede? Exemplo: tamanho), *Unit* (Qual a unidade de medida utilizada? Exemplo: Número de linhas de código, número de classes, etc.) e *Operation* (Elemento que define a operação que será aplicada no elemento alvo, geralmente um código em alguma linguagem de programação compatível). Assim, como todos os elementos do metamodelo SMM, uma métrica e seus elementos devem conter alguns atributos básicos, como *Name*, *Description* e *LabelFormat* (define o formato em que o nome do elemento deverá ser renderizado).

O formato utilizado para representar uma métrica é o XMI (*XMLMetadata*Interc*h*ange), utilizando o esquema definido no metamodelo SMM. Manipular arquivos XMI é muito trabalhoso e propenso a erros. Por isso, o MoDisco *Framework* possui um editor chamado *SMMEditor*, que permite a criação de modelos SMM de forma gráfica, como pode ser visto na Figura 6.2. O MoDisco SMM Editor foi desenvolvido utilizando como base o EMF (*EclipseModelingFramework*) e segue as regras do metamodelo SMM.

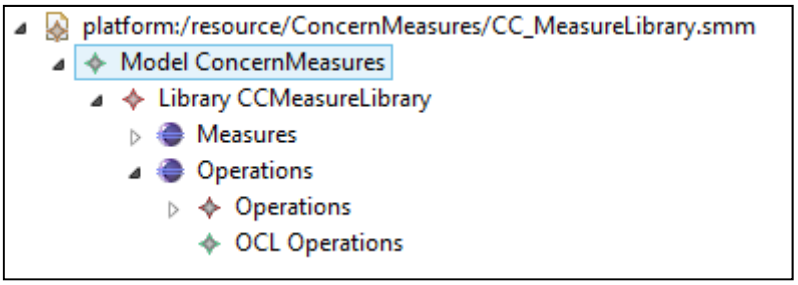

Figura 6.2 - SMM Editor

Como pode ser visto na Figura 6.2, o SMM Editor permite ao usuário definir uma biblioteca de métricas de forma amigável, além de validar o modelo para verificar se o mesmo está em conformidade com o metamodelo.

A definição de métricas de interesse utilizando o metamodelo SMM se difere de métricas tradicionais pelo seguinte motivo: para medir índices relacionados a interesses, especificamente índices de espalhamento e entrelaçamento de interesses, antes, precisamos saber quem (qual componente ou operação) implementa o quê (qual interesse).

Um *software* possui diversos elementos, tais como componentes (classes, interfaces, aspectos), operações (métodos e *advices*), atributos, variáveis, etc. Para medir o espalhamento de um interesse transversal, por exemplo: *Logging*, precisamos saber quais elementos de *software* implementam algo relacionado à *Logging*. Para descobrir que interesses um determinado elemento implementa, é necessário realizar previamente (antes da medição) uma mineração de interesses nos modelos KDM que representam o *software*.

Dado um modelo KDM, uma ferramenta de mineração de interesses irá marcá-lo com anotações a fim de identificar quais interesses os elementos de *software* implementam. Por exemplo: se um método persiste dados no banco de dados, então ele deve ser marcado com a anotação "*concern=persistence*" (ou algo semanticamente equivalente). Essas anotações serão utilizadas para realizar a medição de interesses no modelo.

Dessa forma, a particularidade/diferença que as métricas de interesse possuem é que o elemento que deve ser computado não existe diretamente como uma metaclasse do metamodelo a ser medido. Por exemplo, o interesse de consumo de energia em um sistema robótico pode ser representado por um conjunto de cinco métodos, três atributos e duas classes. Geralmente, esse elemento é

representado de alguma outra forma. Uma das formas mais usadas é com anotações nos elementos de modelo.

```
<codeElement concern="Persistence" xsi:type="code:MethodUnit" name="ConnectionDB"...
    <attribute tag="export" value="private"/>
    <source language="java">
        <region file="/0/@model.2/@inventoryElement.16 language="java"/>
    </source>
    <codeElement xsi:type="code:Signature" name="ConnectionDB">
        <parameterUnit type="/0/@model.0/@codeElement.0/@codeElement.0/@codeEle...
            <source language="java">
                <region language="java"/>
            </source>
        </parameterUnit>
    </codeElement>
```

Figura 6.3 - Exemplo de Modelo KDM Anotado pela Ferramenta CCKDM

Como pode ser visto na Figura 6.3, um modelo KDM pode ser minerado e anotado de acordo o interesse que o mesmo implementa. No caso da Figura 6.3, o elemento anotado é um Método (*MethodUnit*) que implementa o interesse de Persistência, sendo anotado pela ferramenta CCKDM (Santínbáñes, 2013) com a marcação "*concern=persistence*". No caso da ferramenta utilizada nesse projeto, o padrão de anotação é o da Figura 6.3, ou seja; "*concern=nome_do_concern*". Porém, isso pode variar e depende muito de como a ferramenta de mineração foi desenvolvida. Como o padrão de anotação é variável, a Biblioteca de Métricas de Interesse CCML, que é apresentada na Seção seguinte, utiliza um mecanismo de Parametrização de *Tags* de anotação. Logo, as métricas da CCML podem ser reutilizadas em projetos que utilizaram outras ferramentas de mineração e utilizam outros padrões de anotações. Para que isso seja possível, basta o Engenheiro de Modernização especificar o padrão de anotação. Alguns exemplos de anotações são: "*concern*=nomeDoConcern","*implements*=nomeDoInteresse", "nomeDoInteresse=*true*/*false*".

6.3 CCML – *Crosscutting Concern Measure Library* (Biblioteca de Métricas de Interesses Transversais)

Para esse trabalho, foram estudadas e/ou implementadas as seguintes métricas de interesse:

- CDO
- CDC
- CDLOC (não pode ser representada com o SMM)
- LCOO
- Disparidade
- Concentração
- Dedicação
- Concentração*
- Dedicação*
- DOS
- DOF
- CDA
- NOF
- FCD
- ACD
- *Size*
- *Touch*
- *Spread*
- *Focus*

Essas métricas foram agrupadas em uma Biblioteca de Métricas que foi batizada de “CCML – *Crosscutting Concern Measures Library*". A Figura 6.4 apresenta a estrutura da CCML.

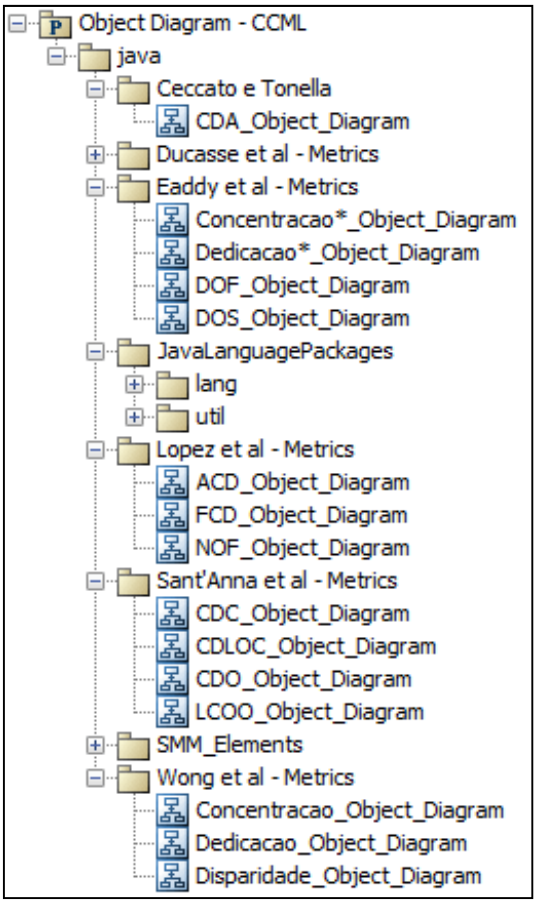

Figura 6.4 - Estrutura da Biblioteca CCML na Ferramenta de Modelagem *AstahCommunity*.

As métricas que foram definidas e estão contidas na Biblioteca CCML serão explicadas detalhadamente das subseções seguintes. A CCML está disponível na *internet* em formato padronizado XMI, em conformidade com o metamodelo SMM, e também em formato *Astah* (.astah) para a ferramenta de modelagem *AstahCommunity* ou *Professional*.As métricas possuem um Diagrama de Objetos da UML para exemplificar sua estrutura, além do código de seu elemento *Operation* (quando implementado) escrito na linguagem *XPath*.

A Tabela 6.1 apresenta as métricas contidas na CCML. A primeira coluna mostra o nome da métrica de interesse encontrada na literatura. A segunda coluna diz se a métrica foi implementada, se ela não foi implementada ou se ela foi implementada de forma parcial. A terceira coluna apresenta as adaptações necessárias para a modelagem da métrica no padrão SMM. As métricas apresentadas na tabela que foram implementadas "Parcialmente" estão ainda em fase de implementação, por isso, na coluna "Adaptações" elas possuem o valor

"N/A" (Não se Aplica), porque ainda não é possível definir as adaptações realizadas antes de terminar sua implementação.

Tabela 6.1 - Conteúdo da Biblioteca CCML, Adaptações e Dificuldades ao modelar métricas de interesse.

Métricas de Interesse Encontradas na Literatura	Foi Modelada?	Adaptações/Dificuldades
CDO	Sim	Não
CDC	Sim	Não
CDLOC	Não	Métrica não aplicável ao contexto de modelos
LCOO	Parcialmente	N/A
Disparidade	Sim	Blocos são representadas como regiões de código-fonte (*Blocks* ->*SourceRegion*)
Concentração	Sim	Blocos são representadas como regiões de código-fonte (*Blocks* ->*SourceRegion*)
Dedicação	Parcialmente	N/A
Concentração*	Sim	Blocos são representadas como regiões de código-fonte (*Blocks* ->*SourceRegion*)
Dedicação*	Sim	Blocos são representadas como regiões de código-fonte (*Blocks* ->*SourceRegion*)

6.3.1 Métrica CDO

A métrica CDO foi definida da seguinte maneira: visto que seu objetivo é medir o espalhamento de um determinado interesse sob *Operations* (Métodos e *Advices*) do software, então, podemos descrevê-la como uma métrica *Counting*, ou seja, uma métrica cujo objetivo é contar quantos Métodos e/ou *Advices* implementam um determinado interesse. Por exemplo: se um *software* possui quatro métodos que implementam *Logging*, então, seu índice CDO é igual a 4 (CDO=4).

No caso da métrica CDO definida na CCML, também podemos dizer que a mesma é uma *CollectiveMeasure*, pois pode-se querer medir mais de um modelo alvo. Logo, trata-se de uma *CollectiveMeasure* associada à uma métrica *Counting*. Após definir o tipo da métrica CDO (*Counting*), foi preciso definir o Escopo. Como o objetivo é medir o espalhamento de um interesse sobre Operações do sistema, então pode-se dizer que o escopo da CDO é o pacote *Code*, onde estão Métodos e

Advices, ou, no KDM, *MethodUnit* e *AdviceUnit*. Seu *Trait* (*Characteristic*) é "*ConcernScattering*", ou seja, a característica medida pela métrica é o espalhamento de interesses. Como *Unit* (unidade de medida) pode-se definir que a CDO utiliza "*NumberOf Affected Operations*", ou seja, quantidade de operações (métodos ou *advices*) anotadas que implementam um determinado interesse).

Também se definiu seu elemento *Accumulator* (obrigatório para métricas do tipo *CollectiveMeasure*), que nesse caso é soma "sum", pois queremos somar o índice CDO de todos os modelos KDM inseridos como entrada para a medição. A Figura 6.5 exemplifica uma instância da métrica CDO definida na Biblioteca CCML.

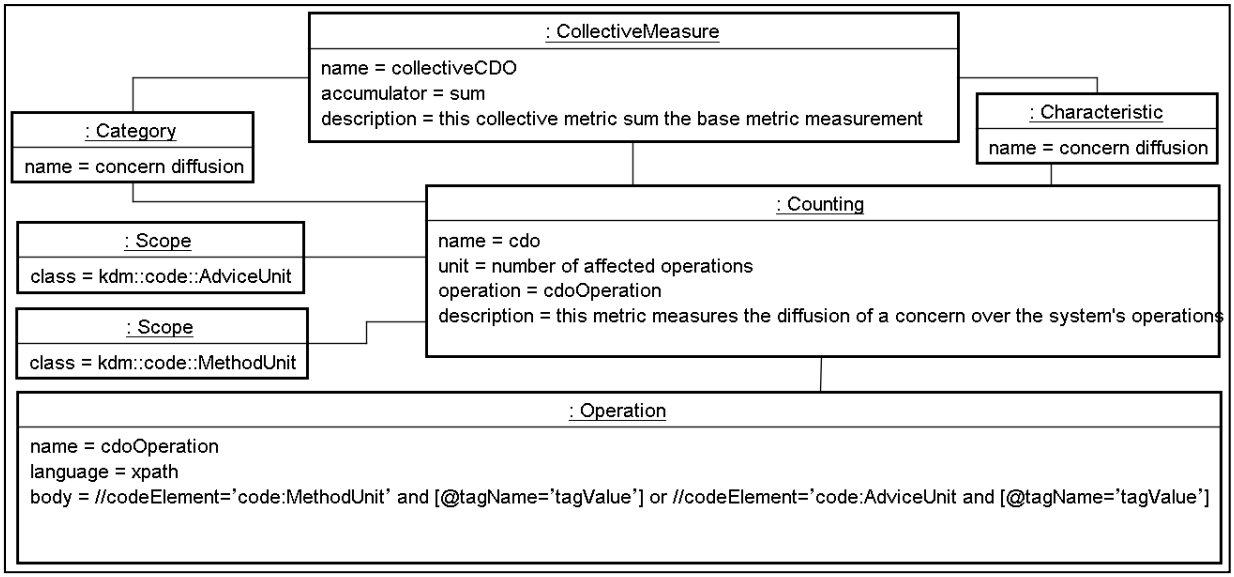

Figura 6.5 - CDO/CCML (Diagrama de Objetos)

Por último, o *Operation* da métrica CDO ficou como mostra a Figura 6.5. O código do *Operation* da métrica CDO, mostrado na Figura 6.6 está escrito na linguagem *XPath* e pode ser interpretado da seguinte maneira:

Procure em todo o arquivo XMI um elemento chamado "*codeElement*" cujo valor é "*code:MethodUnit*" e que possua um atributo chamado "*tagName*" cujo valor é "*tagValue*" ou, procure em todo o arquivo XMI um elemento chamado "*codeElement*" cujo valor é "*code:AdviceUnit*" e que possua um atributo chamado "*tagName*" cujo valor é igual a "*tagValue*".

```
        //codeElement='code:MethodUnit'  and  [@tagName='tagValue']
or //codeElement='code:AdviceUnit and [@tagName='tagValue']
```

Figura 6.6 - CDO/CCML (Diagrama de Objetos)

Lembrando que os valores "*tagName*" e "*tagValue*" são variáveis que podem ser substituídas pelo padrão de anotação utilizado pela ferramenta de mineração. Por exemplo: "*tagName*" pode ser "*concern*" e "*tagValue*" pode ser "*persistence*", sendo assim "*concern=persistence*".

Repare que os termos *tagName* e *tagValue* são utilizados no código do *Operation* da métrica CDO, e também nas métricas que ainda serão apresentadas. Isso ocorre pois a ferramenta desenvolvida para este projeto trabalha com um mecanismo de parametrização de anotações.

Como um Engenheiro de Modernização tem a escolha de usar qualquer ferramenta de mineração de interesses, ou até mesmo minerar manualmente, a ferramenta CMEE, desenvolvida para este projeto, possui um mecanismo que permite ao Engenheiro especificar um código de *Operation* utilizando variáveis como: *tagName*_01, *tagValue*_01, *tagName*_02, *tagValue*_02. A vantagem é que a ferramenta é totalmente independente de técnica de mineração de interesses.

Ferramentas de mineração podem utilizar vários padrões de anotação, tais como: "*concern=persistence*", "interesse=*persistence*", "cc=*persistence*", etc. Ou seja, utilizando variáveis é possível se "adequar" a qualquer padrão de anotação utilizado por qualquer ferramenta de mineração de interesses.

6.3.2 Métrica CDC

Para a definição da métrica CDC, cujo objetivo é medir o espalhamento de um determinado interesse sobre os componentes do sistema, podemos também defini-la como uma métrica do tipo *Counting*, ou seja, uma métrica que irá contar quantos componentes implementam um interesse. Seu escopo é o pacote *Code*, pois este possui elementos que representam componentes, como *ClassUnit*, *AspectUnit* e *InterfaceUnit*. O *trait* (ou *characteristic*) de CDC é "*ConcernScattering*", sua unidade

de medida (*unit*) é “número de componentes” e seu *Operation* pode ser visto na Figura 6.7.

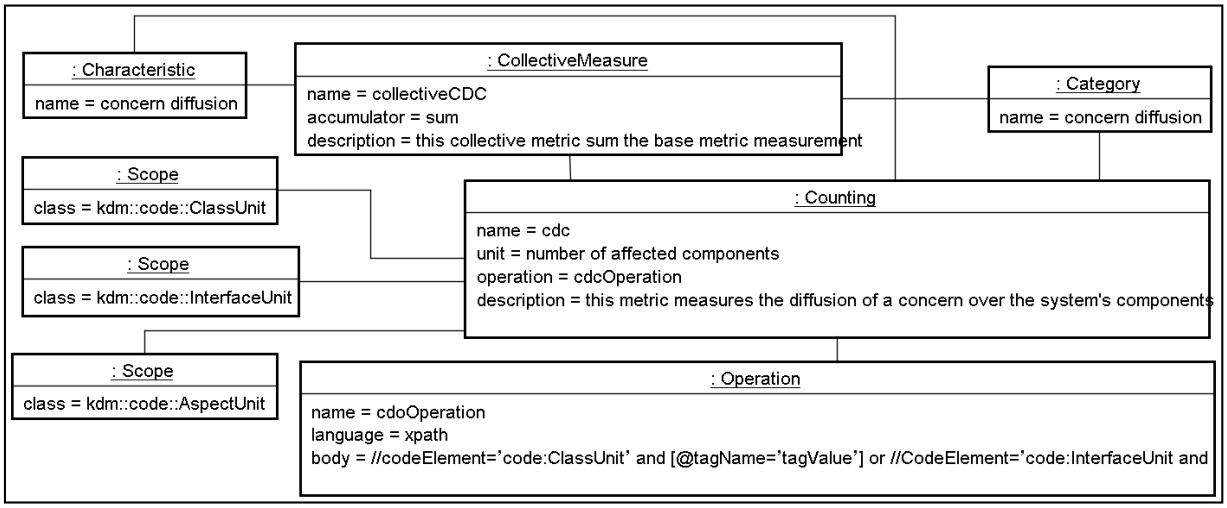

Figura 6.7 - CDC/CCML (Diagrama de Objetos)

O código do *Operation* da métrica CDC, mostrado na Figura 6.7 está escrito na linguagem *XPath* e pode ser interpretado da seguinte maneira.

Procure em todo o arquivo XMI um elemento chamado “*codeElement*” cujo valor é “*code:ClassUnit*” e que possua um atributo chamado “*tagName*” cujo valor é “*tagValue*” ou, procure em todo o arquivo XMI um elemento chamado “*codeElement*” cujo valor é “*code:InterfaceUnit*” e que possua um atributo chamado “*tagName*” cujo valor é igual a “*tagValue*” ou procure em todo o arquivo XMI um elemento chamado “*codeElement*” cujo valor é “*code:AspectUnit*” e que possua um atributo chamado “*tagName*” cujo valor é igual a “*tagValue*”. Lembrando que os valores “*tagName*” e “*tagValue*” são variáveis que podem ser substituídas pelo padrão de anotação utilizado pela ferramenta de mineração. Por exemplo: “*tagName*” pode ser “*concern*” e “*tagValue*” pode ser “*logging*”.

6.3.3 Métrica CDLOC

A métrica CDLOC é diferente, pois mede o espalhamento de um determinado interesse sobre as linhas de código do sistema. Para atingir seu objetivo, a métrica CDLOC conta a alternância de implementação de interesses nas linhas do sistema.

Tal operação não pode ser realizada em modelos KDM, pois o mesmo não permite a representação individual de linhas de código, mas apenas de regiões/trechos de código-fonte,ou seja, a métrica CDLOC. Mesmo que a métrica CDLOC fosse adaptada para contabilizar a alternância de elementos *SourceRegion*, não seria possível, pois um elemento *SourceRegion* pode representar trechos de código-fonte, e não linhas individuais.

6.3.4 Métrica LCOO

A Métrica LCOO (*Lack of Cohesion in Operations*) mede o grau de coesão dos *Operations* (Métodos e *Advices*) de um componente do sistema. Essa métrica é uma extensão da métrica orientada a objetos LCOM (*Lack of Cohesion in Methods*) (Chidamber; Kemerr, 1994). Uma maneira de modelar a métrica LCOO utilizando o metamodelo SMM é classificá-la como uma métrica Contadora, ou seja, uma vez que temos os *Operations* do sistema anotados com o interesse que cada um deles implementa, basta aplicar a lógica de funcionamento da métrica LCOO.

6.3.5 Métrica Disparidade

Objetivo da métrica Disparidade (Wong *et al.*, 2000) é saber o quão próximo está uma *Feature* F (Interesse) de um Componente C (Classes, Interfaces, Aspectos), ou seja, quanto de um interesse é implementado em um Componente C. Para isso, mede-se quantos blocos de código do Componente C implementam um determinado interesse. Quanto menor for este grau de aproximação, maior será a disparidade. (Bruno Silva, 2009). Para melhor compreender e modelar a métrica Disparidade, consideremos que uma *Feature* F é um Interesse I.

Para modelar essa métrica, é necessário que elementos *SourceRegion*, do pacote *Source* do KDM sejam previamente anotados. Para o cenário desse trabalho, temos que considerar que Blocos de código são na verdade Regiões de Código-Fonte, ou seja, elementos *SourceRegion*. Logo, se um elemento *SourceRegion*

possui uma anotação do tipo "*concern=persistence*" podemos contabilizar essa região de código-fonte como implementadora de um determinado interesse.

O Diagrama de Objetos da métrica Disparidade é apresentado na Figura 6.9. Repare que assim como as outras métricas do tipo *Counting*, foi necessário utilizar uma métrica coletiva para somar (função "sum") o resultados coletados pela métrica base. A característica dessa métrica é "*concerndisparity*", sua categoria, no contexto da biblioteca, é "*concerndisparity*" também. Além disso, podemos citar como informações importantes que o escopo dessa métrica é o pacote Source, que possui os elementos *SourceRegion*.

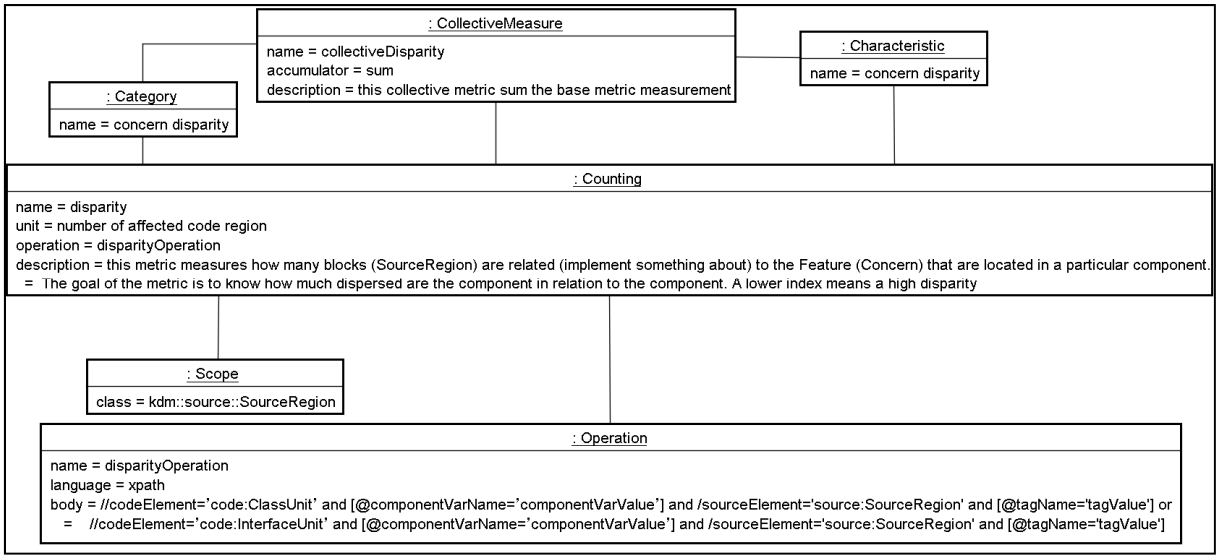

Figura 6.8 - Disparidade/CCML (Diagrama de Objetos)

O código do elemento *Operation* da Métrica Disparidade teve de ser adaptado. Na versão original usa-se "blocos de código-fonte", porém, no caso da ADM, temos que considerar que "blocos de código-fonte" são "*sourceRegion*". Logo, o código pode ser interpretado da seguinte maneira: Procure em todo o arquivo KDM um elemento chamado "*codeElement*" chamado "*ClassUnit*" ou "*InterfaceUnit*" ou "*AspectUnit*" que possuam elementos *SourceRegion* anotados com o valor da variável que será parametrizada pelo usuário da ferramenta CMEE. Repare que *tagName* e *tagValue* são variáveis, ou seja, o usuário pode configurá-las para medir quaisquer tipos de interesses minerados por qualquer ferramenta de mineração.

6.3.6 Métrica Concentração

A Métrica Concentração (Wong *et al.*, 2000) é inversamente proporcional à métrica Disparidade, ou seja, o que muda é a forma como interpretamos o resultado. Um índice de disparidade baixo significa alta disparidade porque as regiões de código-fonte não estão implementando o interesse em questão dentro do componente alvo, logo, há bastante disparidade entre o componente e o interesse. No caso da concentração, temos o inverso, um alto índice de concentração significa que os elementos *SourceRegion* do componente estão sim implementando um interesse específico, ou seja, existe concentração de implementação de um interesse no componente em questão.

Por razões de igualdade, o diagrama de objetos da CCML relacionado à métrica Concentração não é apresentado aqui, mas está presente na biblioteca.

6.3.7 Métrica Dedicação

A métrica Dedicação “DEDI(C,I)” (Wong *et al.*, 2000) mede a proporção em que os blocos do componente C estão dedicados à implementação do Interesse I em relação ao conjunto de todos os blocos de C (Wong *et al*., 2000 apud Bruno Silva, 2009). Com a explicação anterior, podemos considerar então que, se o conjunto dos elementos *SourceRegion* que implementam o interesse I é proporcional ao conjunto dos elementos *SourceRegion* que não implementam o interesse I, então, temos um componente dividido, que possui a mesma quantidade de elementos *SourceRegion* que implementam o interesse I que elementos *SourceRegion* que não Implementam. Nesse caso, o Componente C não está totalmente dedicado à implementação do Interesse I.

No caso de os valores não serem proporcionais, pode-se dizer ou não se um componente está dedicado à implementação de um interesse. No caso do número de elementos *SourceRegion* que implementam o interesse I for igual ao total de elementos *SourceRegion* do Componente C, podem dizer que ele está dedicado à implementação do Interesse I, caso o número de elementos *SourceRegion* que

implementam o interesse I seja 0, podemos dizer que ele não está dedicado à implementar tal interesse.

Pode-se citar como informações importantes que o escopo dessa métrica é o pacote *Source*, do metamodelo KDM.

6.3.8 Métricas Concentração*, Dedicação*, DOS e DOF

Em métricas que se avaliam os interesses que as Linhas de Código (LOC) implementam, caso das Métricas Concentração*, Dedicação*, DOS (*Degree of Scattering*) e DOF (*Degree of Focus*) (Eaddy *et al.*, 2007), temos uma limitação em relação ao KDM. A limitação não é do SMM, e sim do metamodelo KDM, que não representa uma linha de código de forma fiel, e sim, as representa em forma de "trechos de código", por meio de elementos chamados *SourceRegion*. Portanto, podemos afirmar que métricas que envolvem Linhas de Código não podem ser modeladas de forma fiel à sua definição original, porém, podem-se definir modelos SMM que as representam com algumas pequenas adaptações. No caso das métricas de Eaddy*et al.* (2007), temos a seguinte situação: as métricas originais de Wong *et al.* (2000), Concentração e Dedicação já foram adaptadas para a Biblioteca CCML e são explicadas nas Seções 6.3.6 e 6.3.7, respectivamente. Portanto, não há a necessidade de explicá-las novamente e apresentar novos diagramas de objeto.

A Métrica DOS visa medir o espalhamento de um interesse nos componentes de um sistema. O escopo da métrica é o pacote *Code*, que possui elementos como: Classes (*ClassUnit*), Aspectos (*AspectUnit*), Interfaces (*InterfaceUnit*). A característica que a métrica mede é "*ConcernScattering*" e a categoria também é "*ConcernScattering*". A unidade de medida da métrica DOS é o número de componentes afetados por um determinado interesse.

Já a métrica DOF mede o foco do componente em implementar um determinado interesse. Essa métrica tem como Escopo os componentes do sistema (contidos no pacote *Code*), assim como a métrica DOS. A diferença entre DOS e DOF são somente suas respectivas fórmulas, que podem ser vistas na Seção 3.3.3.

6.3.9 Métrica CDA (*Crosscutting Degree of an Aspect*)

O objetivo dessa métrica é calcular quantos componentes são afetados por um determinado Aspecto. Se um aspecto entrecorta, de alguma maneira, 5 classes e 3 interfaces, logo, seu índice CDA será 8.

Uma possível maneira de modelar a métrica CDA com metamodelo SMM é contar os pontos de junção (*JoinPoints*) do aspecto, depois, pode-se filtrá-los de forma que não hajam repetidos, ou seja, no fim, teremos o índice CDA. Portanto, a Métrica CDA é modelável com o metamodelo SMM. A estrutura da métrica é a seguinte: seu escopo é o pacote *Code*, pois é ele que possui componentes como *ClassUnit*, *InterfaceUnit* e *AspectUnit.* A característica que a métrica mede é o impacto, ou grau de espalhamento de um aspecto sobre os componentes do sistema, sua unidade de medida, que são os componentes afetados pelo aspecto e sua categoria é a difusão de um aspecto (*concern diffusion*).

6.3.10 Métrica NOF (*Number of Features*)

A Métrica NOF (*Number of Features*) mede o número de *Features* em um sistema. Segundo Figueiredo e Cacho (2008), Podemos entender *Feature* como sendo um Interesse. Logo, NOF mede o número de Interesses em um sistema.

Essa métrica é simples de se modelar utilizando SMM pelo fato de que basta que os componentes *AspectUnit* sejam previamente anotados. Uma vez que os componentes *AspectUnit* estão anotados com o interesse que o mesmo está dedicado à implementar, podemos contabilizar +1 para o índice NOF. Lembrando que se dois *AspectUnit* implementarem um mesmo interesse, os dois juntos contabilizam apenas 1 no resultado final, ou seja, são contabilizados apenas anotações únicas.

A característica dessa métrica é a quantidade de interesses. Sua unidade de medida é o número de interesses. Sua categoria dentro da Biblioteca CCML é "*concern quantitative Metrics*" e seu escopo é o pacote *Code*, que possui os elementos *AspectUnit*.

Uma maneira de codificar seu *Operation* é identificar todos os *AspectUnit* que implementam interesses transversais (sem repetição) e somá-los. Dessa forma teríamos uma versão da métrica NOF para modelos SMM.

6.3.11 Métrica FCD (*Feature Crosscutting Degree*)

A métrica FCD (*Feature Crosscutting Degree*) mede o número de classes que são afetadas por todos os *advices* de um determinado *Feature* (Interesse). Então, se uma *Feature* é um Interesse, e um Interesse Transversal pode ser representado por um ou mais Aspectos, basta contar o número de classes que são entrecortadas por *advices* de Aspectos que implementam um determinado interesse. Pode-se dizer então que, o índice FCD é o número de elementos *ClassUnit* afetados de alguma forma por elementos *AspectUnit* que implementam um mesmo interesse. Um Interesse pode ser representado por um ou mais elementos *AspectUnit*, então, se contabilizarmos os pontos de junção de todos os *AspectUnit* que entrecortam classes do sistema e implementam o mesmo interesse, teremos o índice FCD.

O Escopo dessa métrica está focado nos seguintes elementos: *AspectUnit*, *AdviceUnit*, *JoinPointUnit* (todos pertencem ao pacote *Code*). A Característica da métrica é medir o espalhamento de um interesse sobre as classes do sistema por meio do entrecorte de classes (elementos *ClassUnit*). Sua categoria na Biblioteca CCML é "*concern quantitative Metrics*". Uma maneira de implementar o código de seu *Operation* seria identificar anotações nos elementos *JoinPointUnit* e verificar quais classes eles entrecortam, realizando assim uma contagem para se obter o índice FCD

6.3.12 Métrica ACD (*Advice Crosscutting Degree*)

A métrica ACD (*Advice Crosscutting Degree*) visa medir o número de classes afetadas por um único *Advice* específico. Para isso basta contar o número de Pontos de Corte do *Advice* (sem repetições) e teremos o índice ACD. Pode-se dizer então que a métrica ACD visa medir o número de elementos *ClassUnit* afetados por um

AdviceUnit específico (especificado na definição do *Operation* da métrica). O Escopo dessa métrica é o pacote Code, que possui os elementos *AdviceUnit* e também os elementos *ClassUnit*.

A característica da métrica é medir o espalhamento de um interesse através de pontos de corte dos *AdviceUnit* do sistema. Os Escopos são: *AdviceUnit* e *JoinPointUnit* e a categoria da métrica ACD na Biblioteca CCML é "*concernquantitativeMetrics*".

Uma possível maneira de implementar o código do *Operation* da métrica ACD seria identificar anotações nos elementos *AdviceUnit* e verificar quais classes eles manipulam, realizando assim uma contagem para se obter o índice ACD

6.3.13 Métricas *Size, Touch, Spread, Focus*

As métricas de Ducasse *et al.,* (2006) possuem termos que não podemos encontrar no metamodelo KDM, como Partição de Referência e Partição de Comparação, porém, nesse caso, podemos modelar as métricas *Size*, *Touch*, *Spread* e *Focus* com adaptações. Uma vez que podemos considerar e definir que Partição de Referência, são os elementos de programa Classes (*ClassUnit*), Métodos (*MethodUnit*), Interfaces (*InterfaceUnit*). Partição de Comparação são os interesses de um sistema, ou seja, uma Partição de Comparação pode ser, por exemplo, o interesse de *Logging*.

No KDM, uma Partição de Comparação são anotações realizadas por ferramentas de mineração de interesses. *Logging* em um sistema é pode ser considerado uma Partição de Comparação.

As Métricas de Ducasse *et al.,* (2006) não foram modeladas e não estão na Biblioteca CCML porque ainda serão melhor estudadas, de forma que sua modelagem seja feita da forma mais fiel possível, em um trabalho futuro.

6.4 Considerações Finais

O Capítulo 6 apresentou uma explicação sobre a definição de métricas de interesse utilizando o metamodelo SMM. Além disso, explicou em detalhes a modelagem das métricas de interesse (quando possível) encontradas na literatura utilizando o SMM, apresentando seus respectivos Diagramas de Objetos da UML (para fins de exemplificação e para ajudar o leitor a compreender cada modelo de métrica) e Código da Operação (somente em alguns casos). Também foi apresentada a estrutura de uma Biblioteca de Métricas SMM.

Métricas definidas com o metamodelo SMM são padronizadas. Isso significa que podem ser reusadas por outros Engenheiros, em outros projetos de modernização e em outros cenários de modernização. Para que o reuso das métricas seja possível, é importante que a Ferramenta de medição esteja em conformidade com o metamodelo SMM, ou seja, que seja capaz de receber como entrada um modelo de métricas SMM e modelos alvo, depois, a ferramenta precisa compreender os modelos, aplicar as métricas selecionadas pelo Engenheiro de modernização e gerar um modelo SMM de saída com os resultados da medição, além disso, é preciso que a ferramenta saiba interpretar a linguagem utilizada para a definição dos códigos dos elementos *Operation*. No caso deste trabalho, até a versão 0.1 da ferramenta CMEE, apenas a linguagem *XPath* é suportada.

Este trabalho focou em medições de índices de interesses em modelos KDM anotados, porém, a anotação gera a seguinte limitação: O KDM “puro” não é suficiente para ser alvo de medições de interesse (ao menos para as métricas apresentadas neste trabalho). Isso abre uma lacuna a ser estudada: “Medição de Interesses em modelos KDM não anotados”.

Capítulo 7

CMEE: UM APOIO COMPUTACIONAL PARA APLICAÇÃO DE MÉTRICAS SMM EM MODELOS

7.1 Considerações Iniciais

Este Capítulo mostra uma ferramenta de apoio computacional desenvolvida especialmente para este projeto, para que fosse possível a aplicação de métricas de interesse SMM em modelos KDM de forma automatizada. Os detalhes da ferramenta são mostrados nas seções que seguem.

7.2 Funcionamento da Ferramenta

A ferramenta CMEE foi desenvolvida utilizando a linguagem Java SE (*Standard Edition*) e se integra ao ambiente de desenvolvimento integrado Eclipse na forma de *Plug-in*. Suas atividades e interação com o usuário podem ser vistas no diagrama de atividades da UML apresentado na Figura 7.1.

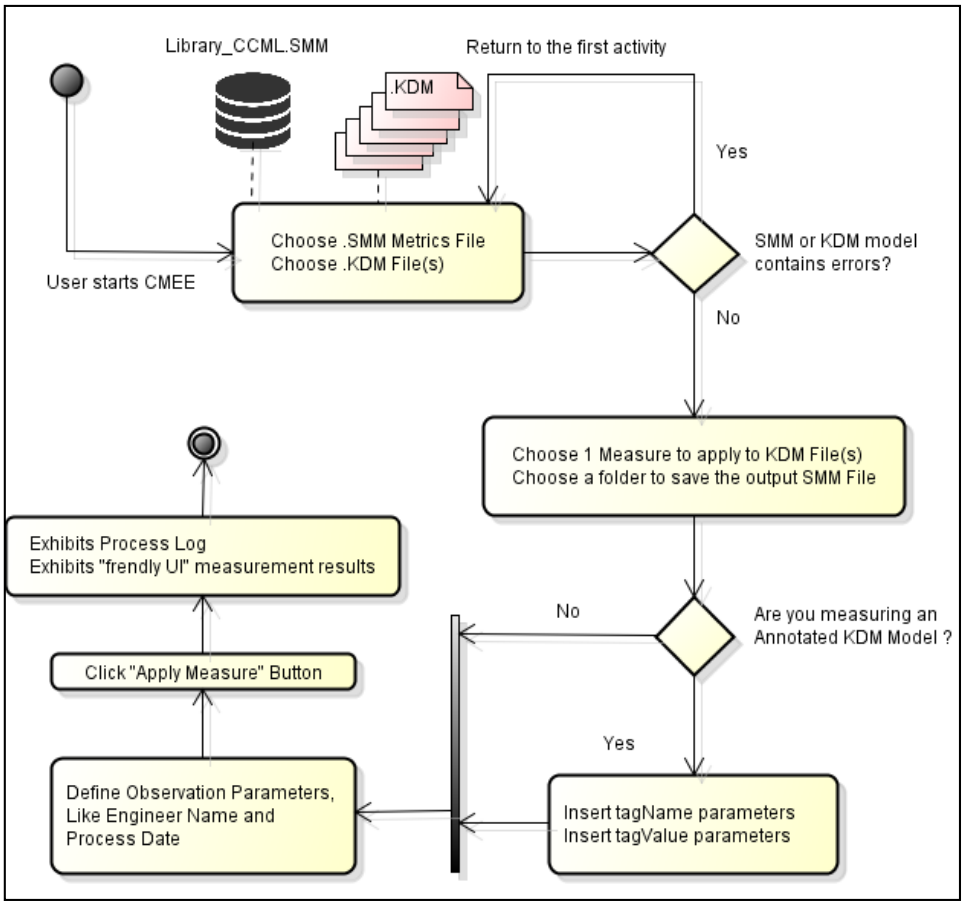

Figura 7.1 - Diagrama de Atividades da UML – CMEE

A interação se inicia quando o usuário clica no botão “CMEE”, na interface do Eclipse IDE. Uma janela se abrirá (Figura 7.2) e o usuário deverá informar quais os modelos de entrada. São necessários no mínimo dois modelos: Um modelo SMM que contém a biblioteca de métricas (neste trabalho foi criada e utilizada a biblioteca de métricas SMM CCML – *Crosscutting Concern Measures Library*) e um modelo KDM que será o alvo das métricas.

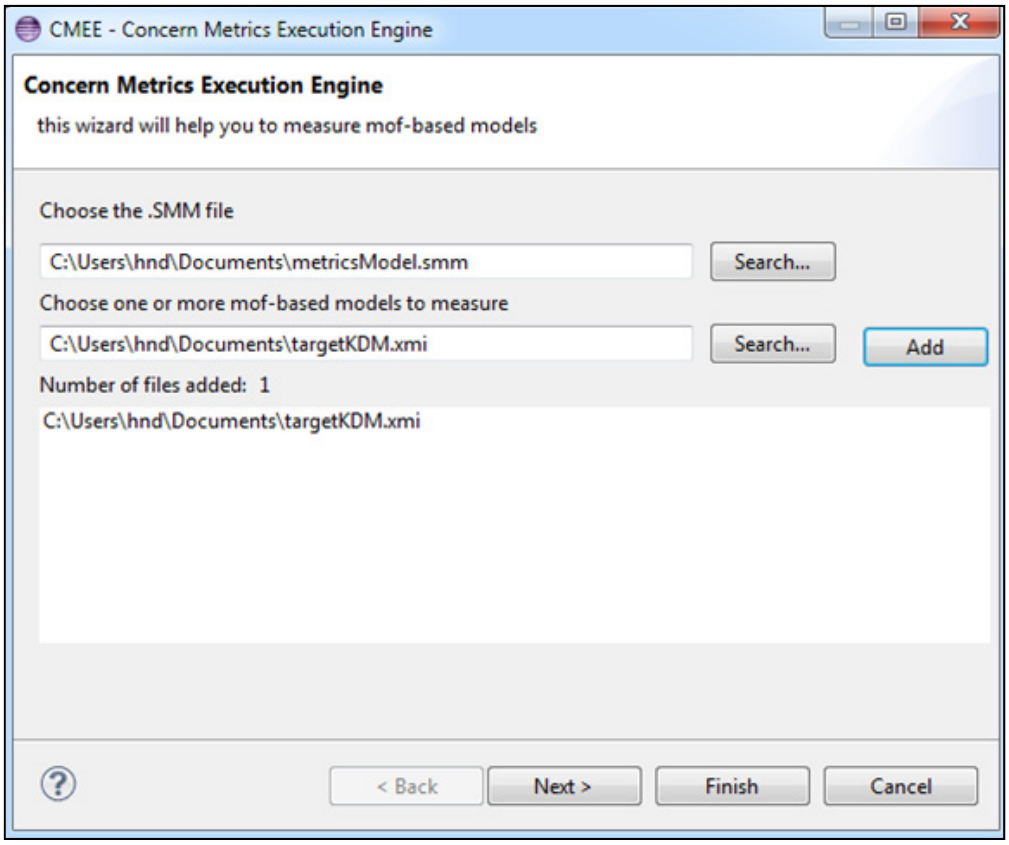

Figura 7.2 - CMEE: Tela de escolha dos modelos de entrada

Após o usuário informar os modelos e entrada e clicar em "*Next*", as métricas encontradas no modelo de métricas SMM serão listadas (Figura 7.3). Com a lista de métricas, o usuário deverá escolher qual delas ele quer que seja aplicada ao modelo KDM alvo. Para isso, basta clicar no nome da métrica e clicar em "*Next*" novamente.

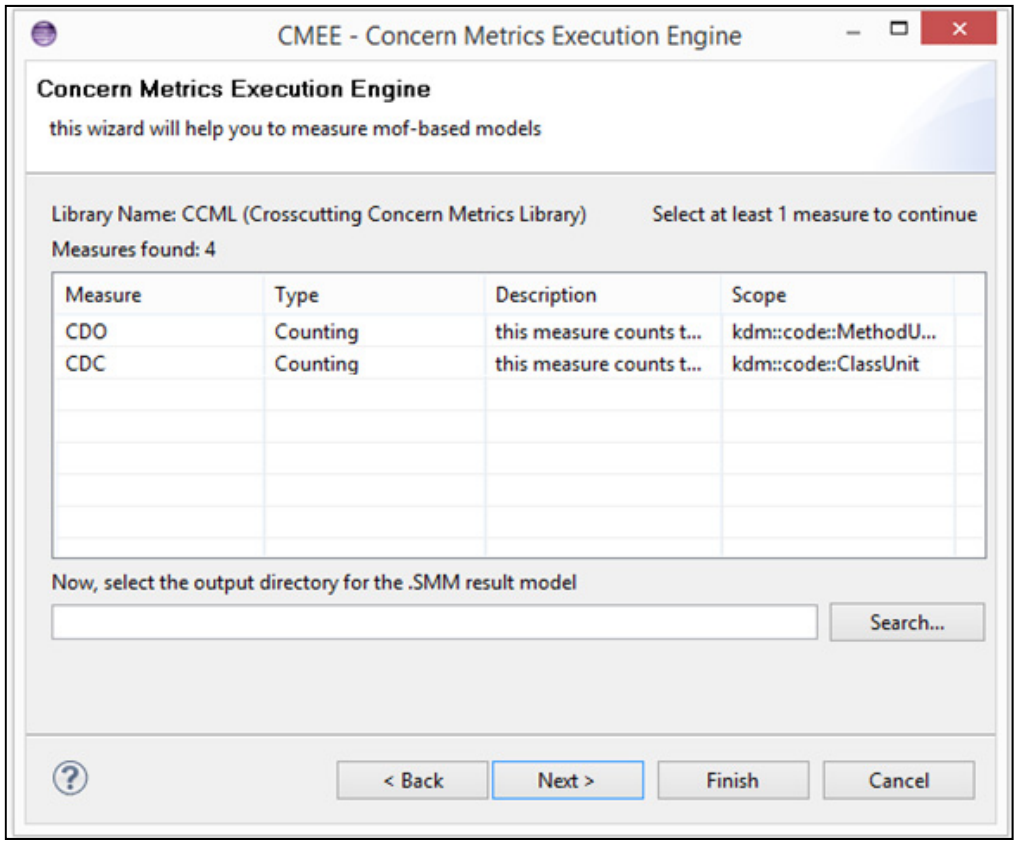

Figura 7.3 - CMEE: Tela de escolha da métrica a ser aplicada no modelo alvo

Após selecionar a métrica a ser aplicada, o usuário tem a possibilidade de configurar os parâmetros das anotações. Como cada KDM a ser medido pode usar um padrão diferente de anotação, a CMEE permite que o usuário ajuste o código do *Operation* da métrica para que ele funcione com qualquer anotação. Então, caso o usuário desejar medir um modelo anotado, ele deve selecionar a caixa "*Are you measuring an annotated model?*". Depois, ele deve informar quais os nomes e valores das meta *Tags* utilizadas no código do *Operation*. Meta *Tags* são espécies de variáveis declaradas no código do elemento *Operation*. Chamá-las de variáveis faz sentido pelo fato da ferramenta CMEE substituí-las por parâmetros definidos pelo Engenheiro de Modernização. Os valores que as meta *Tags* (*metaTagName* e *metaTagValue*) recebem são os definidos pelo Engenheiro nos campos "*TagName*" e "*TagValue*" na tela da CMEE apresentada na Figura 7.4. Repare na Figura 7.4 que é possível adicionar vários parâmetros, ou seja, é possível medir, ao mesmo tempo, anotações com diferentes padrões de nomes, por exemplo: "*concern=persistence*" e "interesse=persistência" ou "cc=*persistence*" e "it=*persistência*". É importante lembrar que a ferramenta CMEE também é capaz de aplicar métricas tradicionais (que não

medem interesse e nem usam modelos KDM anotados), por isso há a necessidade de dizer para a ferramenta que se quer medir um modelo anotado.

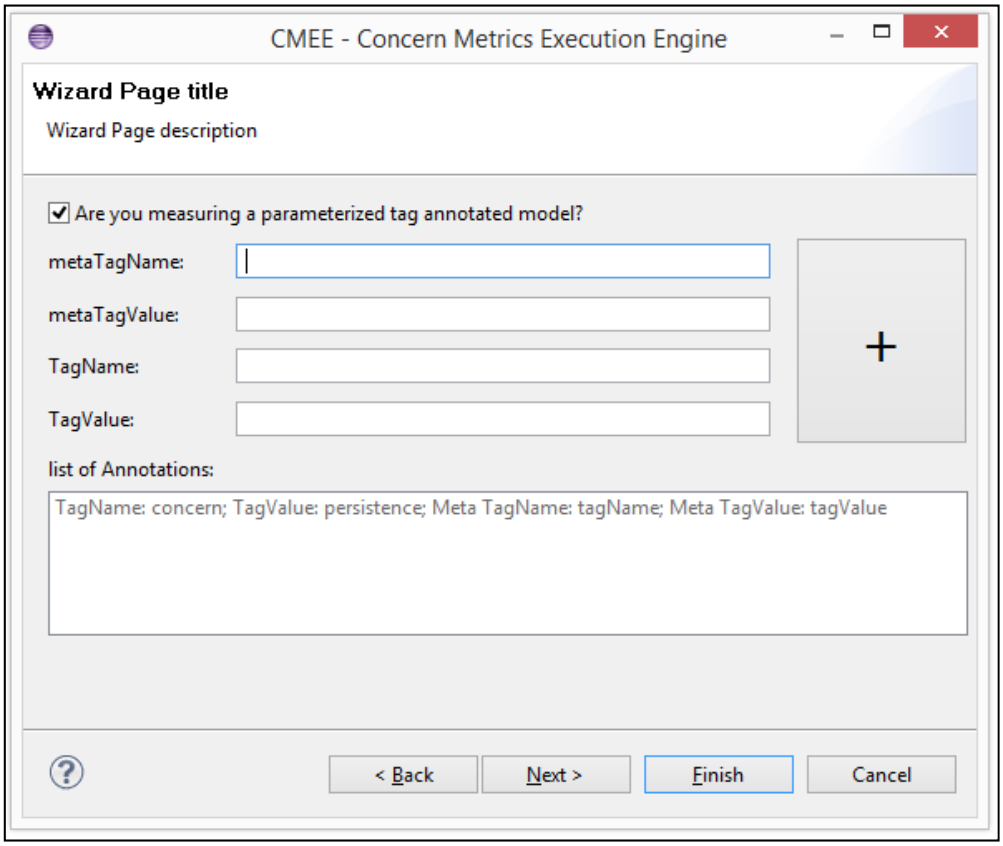

Figura 7.4 - CMEE: Tela de parametrização das *Tags* de anotação

Após configurar os parâmetros das *Tags*, o usuário deve informar seu nome e uma descrição para o processo de medição (Figura 7.5). Depois, basta clicar em "*ApplyMeasures*" para iniciar o processo de medição.

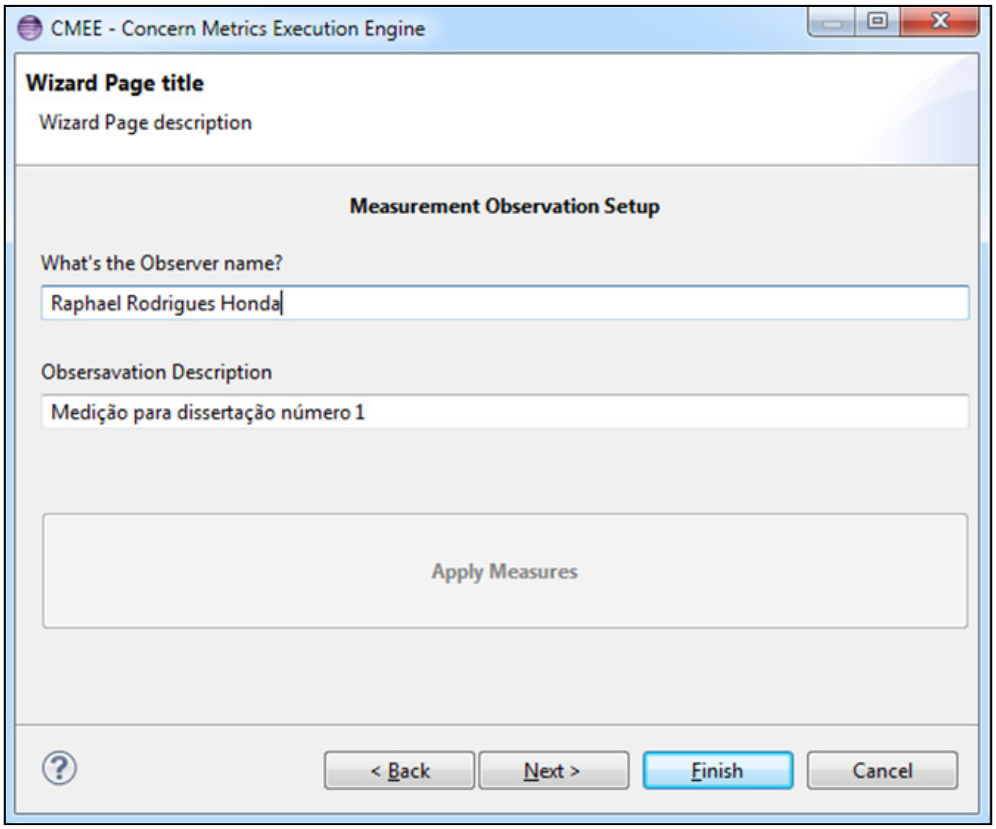

Figura 7.5 - CMEE: Tela de configuração da medição

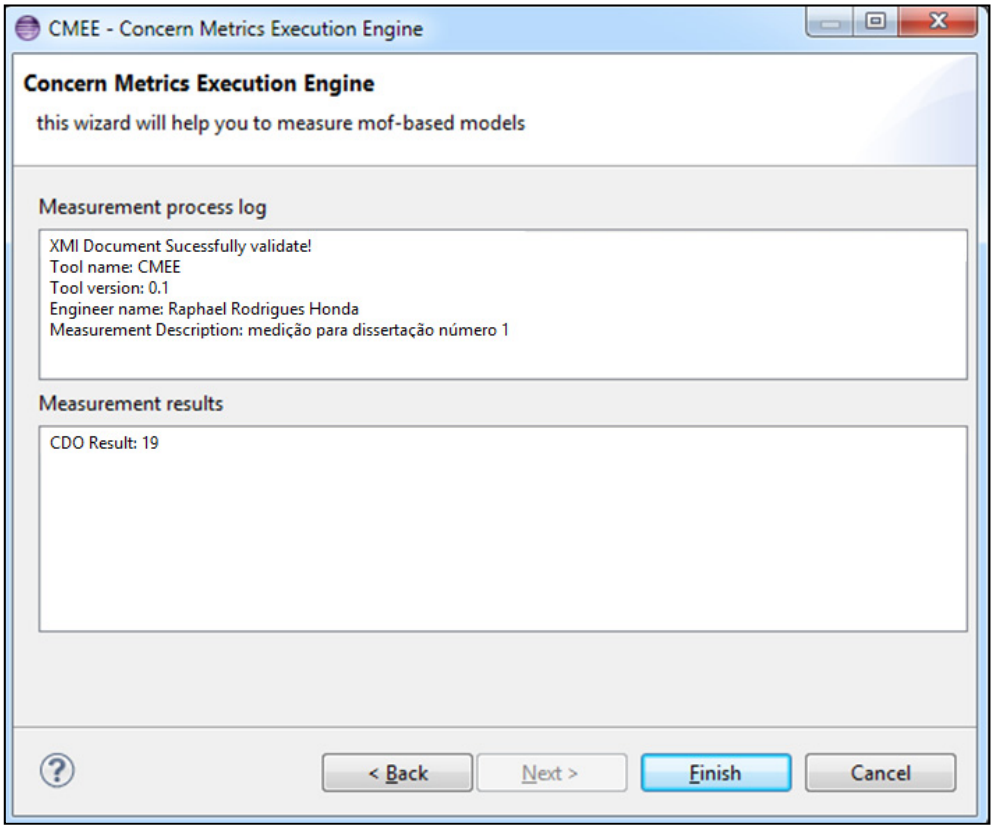

Figura 7.6 - CMEE: Tela de apresentação dos resultados

Na última tela da ferramenta (Figura 7.6), é exibido o *Log* do processo com as mensagens de “Sucesso” ou “Erro” e também o resultado das medições de forma amigável para o usuário. Além disso, a versão completa dos resultados das medições é gerada e salva no diretório escolhido pelo usuário.

7.3 Exemplo de Aplicação da CMEE

Para a avaliação da ferramenta no contexto da abordagem trabalhada, concluiu-se que uma das maneiras de fazê-la seria aplicando a CMEE e a CCML em um *software* que precisasse passar por um processo de modernização. Portanto, a avaliação da abordagem foi realizada aplicando a ferramenta CMEE em modelos KDM que representam um sistema legado em relação ao interesse transversal de persistência e em modelos KDM que representam o sistema legado utilizando um *framework* transversal de persistência (Camargo e Masiero, 2005). A Seção 7.2 descreve a aplicação da abordagem em uma aplicação legada e a Seção 7.3 descreve a aplicação da abordagem em uma aplicação modernizada. A Seção 7.4 mostra os resultados obtidos com a aplicação das métricas CDO e CDC. A Seção 7.5 finaliza o capítulo.

O *Software* escolhido para essa avaliação foi o CD/DVD *Store*. Trata-se de uma aplicação escrita na linguagem de programação Java e que possui um *framework* de persistência acoplado. O *Framework* de persistência acoplado também está escrito em Java e utiliza o paradigma Orientado e Objetos. O Problema é que linhas de código-fonte relacionadas à persistência ou à ligação da aplicação base com o *Framework* estão em praticamente todas as classes.

Pode-se dizer que esse é um exemplo de que, por mais que a orientação a objetos seja um paradigma que permita uma boa modularização, ainda possui alguns problemas de entrelaçamento. Observe no código abaixo um exemplo do problema supracitado

```
}
catch(SQLException e){
    System.out.println("CATCH: SQLException "+ e.getMessage());
}
```

Figura 7.7 - Trecho de código que implementa o interesse de persistência na aplicação CD/DVD *Store*

Na Figura 7.7 é possível observar que existe código-fonte de implementação relacionado ao interesse de persistência. Por isso, podemos dizer que esse componente (Classe) implementa de alguma forma o interesse de persistência. Para resolver esse problema e melhorar a modularização do sistema, foi realizada uma modernização a fim de remover essa implementação de persistência da aplicação base e deixar essa tarefa a cargo do *Framework* Transversal Orientado a Aspectos.

Um *Framework* Transversal OA de persistência implementa toda a parte de persistência da aplicação de forma totalmente modular. Ou seja, atributos, métodos e *advices* serão executados para realizar a persistência, mas isso não estará na aplicação base, e sim no *Framework* Transversal.

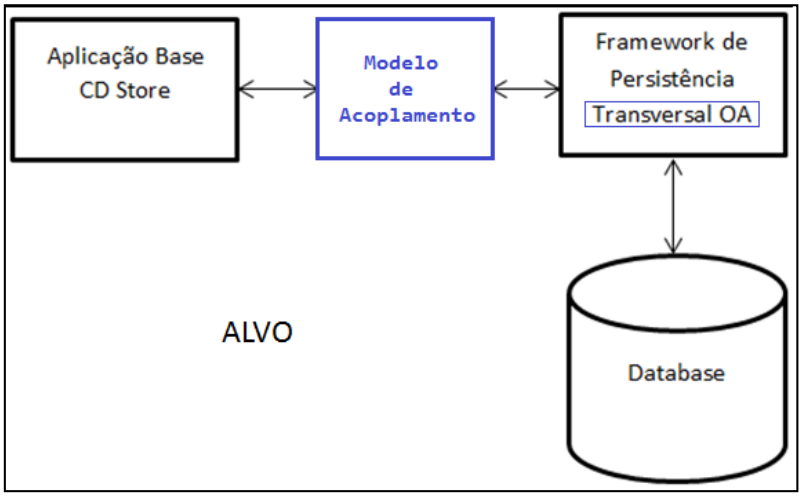

Figura 7.8 - Acoplamento de um *Framework* de Persistência à uma aplicação base (Antes e Depois)

Para que o interesse de persistência seja totalmente removido da aplicação base, é necessária a instanciação de um modelo de acoplamento (Figura 7.8). Esse modelo de acoplamento possui informações como, por exemplo, a *String* de conexão, usuário e senha do banco de dados. É o modelo de acoplamento que

possibilita a comunicação entre a aplicação e o *Framework* de Persistência Transversal.

Para a avaliação, foi preciso então realizar os passos de uma modernização apoiada pela ADM, isso foi feito como apresentado no roteiro abaixo.

1. Transformação do código-fonte Java da aplicação CD/DVD *Store* para modelos KDM (Transformação Código-para-Modelo) por meio do MoDisco.
2. Mineração dos interesses transversais por meio da ferramenta CCKDM
3. Medição dos índices CDO e CDC (Antes do processo de Refatoração)
4. Acoplamento do *Framework* Transversal de Persistência
5. Medição dos interesses transversais por meio da ferramenta CCKDM
6. Avaliação dos resultados (Tabela 7.1)

Os resultados das medições dos passos 3 e 5 podem ser vistos na Tabela 7.1, que apresenta os índices coletados para as métricas CDO e CDC, antes e depois do acoplamento do FT de Persistência à aplicação CD/DVD *Store*.

Tabela 7.1- Tabela de resultados das medições (CDO e CDC)

ÍNDICES CDO E CDC	CDO	CDC
APLICAÇÃO BASE LEGADA COM FT OO	**13**	**97**
APLICAÇÃO BASE LEGADA COM FT OA	**1**	**19**

Observando a Tabela 7.1, é possível ver que os índices coletados depois do acoplamento do FT de Persistência Transversal são consideravelmente mais baixos. O espalhamento do interesse de persistência sobre as operações da aplicação base, que antes era de 13, caiu para 1. E o espalhamento do interesse transversal de persistência sobre os componentes, que era de 87, caiu para 19.

Para verificar a veracidade desses números, foi realizada uma contagem manual, que conferiu com os dados da ferramenta CMEE.

Com isso, considera-se que, além de validar a viabilidade do cenário de modernização em questão, também se validou a utilidade de uma ferramenta de medição de interesses em modelos, como a CMEE.

Considerando o cenário de modernização utilizado nesse trabalho, que chamamos de "*Aspect-Oriented Modernization*", a abordagem funcionou bem, entregando ao Engenheiro de Modernização uma ferramenta computacional que permite a coleta de dados por meio de métricas de interesse. Por ser dinâmica e parametrizável, a CMEE permite que novas métricas sejam criadas e medidas por meio de seu mecanismo.

7.4 Limitações da Ferramenta CMEE

Pode-se citar como limitação da ferramenta o fato de ela somente medir modelos previamente minerados e anotados (marcados) por uma ferramenta de mineração de interesses.

A CMEE também se limita ao fato de somente executar métricas cujos códigos de Operação (*Operation*) foram escritos utilizando-se da linguagem *XPath* (até Setembro de 2014).

7.5 Considerações Finais

O apoio computacional desenvolvido para esse trabalho foi apresentado visando mostrar suas principais funções. Pôde ser observado que a ferramenta consegue medir modelos KDM e KDM Orientado a Aspectos por meio de modelos de Métricas SMM, de forma automatizada, desde que os modelos tenham sido previamente minerados e anotados por uma ferramenta de mineração de interesses, sendo essa uma limitação da abordagem.

Capítulo 8

CONCLUSÕES

Este capítulo tem como objetivo finalizar a discussão levantada nesse trabalho e também deixar algumas sugestões de lacunas de pesquisa para trabalhos futuros relacionados ao tema.

Como resultados obtidos com o desenvolvimento deste trabalho, pode-se citar uma abordagem completa de medição de interesses utilizando modelos da ADM, composta por uma extensão do KDM para a representação de *software* orientado a aspectos (AO-KDM) e uma biblioteca de métricas de interesses no formato SMM (CCML) desenvolvida com o intuito de ser parametrizável pelo Engenheiro de Modernização, ou seja, com o intuito de poder ser reusada em outros projetos.

Além disso, foi desenvolvida uma ferramenta de apoio computacional (CMEE) capaz de lidar com parametrização de anotações (padrões de anotações de interesses realizadas por ferramentas de mineração) que permite que modelos anotados com diferentes ferramentas de mineração possam ser medidos por métricas SMM.

Essa abordagem visa o cenário de modernizações apoiadas pelos padrões da ADM cujo objetivo é melhorar a modularização dos interesses transversais de sistemas legados, porém, a ferramenta CMEE possui a capacidade de automatizar também a medição de métricas SMM que não visam medição de interesses, tornando-a mais genérica e abrangente para ser utilizada por Engenheiros também em outros cenários de modernização.

Pode-se dizer também que uma lacuna observada no mapeamento sistemático realizado para este trabalho foi preenchida. O estado da arte da área de medição de interesses utilizando modelos da ADM avançou, pois não foram encontrados trabalhos semelhantes. Além disso, o fato de desenvolver uma

abordagem de medição onde os códigos a serem aplicados nos modelos são parametrizáveis aumenta a chance desta abordagem ser reusada entre a comunidade e indústria.

Este trabalho também gerou alguns artigos científicos, como um mapeamento sistemático sobre ADM e seus metamodelos, denominado "*A Mapping Study on Architecture-Driven Modernization*" publicado na Conferência de Integração e Reuso da Informação (15º *IEEE International Conference on Information Reuse and Integration*) (Durelli *et al.,* 2014a) e um estudo sobre extensões para o metamodelo KDM foi publicado no Simpósio Brasileiro de Engenharia de *Software* (SBES 2014) (Bruno Santos *et al.*, 2014b).

8.1 Limitações da Abordagem e do Metamodelo SMM

Como limitações desta abordagem, podemos citar a ausência de uma ferramenta de transformação código-para-modelo (*parser*) que gere modelos baseados no AO-KDM. Isso dificulta testar sistemas grandes, uma vez que modelá-los manualmente seria exaustivo e propenso a erros. Para gerar modelos baseados no metamodelo KDM original, foi utilizado o MoDisco, mas que também possui certas limitações, uma vez que não contempla todos os pacotes do metamodelo KDM.

A Ferramenta CMEE (*Concern Metrics Execution Engine*), desenvolvida especialmente para este trabalho é, de certa forma, dependente da extensão AO-KDM ou de extensões que usem a mesma nomenclatura para os elementos, como *AspectUnit*, *AdviceUnit*, *PointCutUnit*. Além disso, até a data da entrega desse trabalho (setembro de 2014), a ferramenta CMEE interpreta somente métricas cujo código do elemento *Operation* é definido com a linguagem *XPath*.

Também pode ser citado como limitação do metamodelo SMM o fato de não permitir a representação totalmente fiel das métricas de interesses aqui estudadas, uma vez que, por exemplo, não consegue representar linhas de código-fonte, e sim, regiões de código-fonte.

8.2 Sugestões Para Trabalhos Futuros

Como trabalho futuro pode-se listar algumas melhorias e estudos a serem realizados:

- Modelar as métricas de interesses estudadas que ainda não estão na biblioteca CCML
- Estudar a viabilidade da criação de métricas de interesse que sejam independentes de anotações/marcações
- Propor ao grupo OMG integrar o KDM-AO ao metamodelo KDM original, para que se torne um padrão.
- Implementar uma versão *web* da ferramenta CMEE
- Implementar um módulo de avaliação de código baseado na detecção de *bad smells* na CMEE.
- Permitir a execução de códigos de elementos *Operation* escritos em outras linguagens, como: OCL, Java, C, PHP, *Python*.
- Permitir a criação e edição de métricas SMM dentro da própria ferramenta CMEE.

REFERÊNCIAS

ADM Task Force – Standards Roadmap. Object Management Group. **ADM White Papers and Roadmap**. Disponível em: <http://adm.omg.org/ADMTF%20Roadmap.pdf>Acesso em 10 de Fevereiro de 2013.

ALI, H.; BIÇER, V. **Modern Software Engineering Concepts and Practices: Advanced Approaches**, IGI Global, December 2010, ISBN 1-60960-215-3 ISBN 13: 978-1-60960-215-4.

Architecture-Driven Modernization, 2014.**Document omg**. Disponível em: <http://adm.omg.org>.Acesso em: 10 de Agosto de 2014.

BARESI, L.; MIRAZ, M. **"A Component-oriented Metamodel for the Modernization of Software Applications,"** 16th IEEE International Conference on Engineering of Complex Computer Systems. 2011.

BARRA, E.; GENOVA, G.; LLORENS, J. **An approach to aspect modelling with UML 2.0**.In Proceedings of the AOM workshop at AOSD, 2004, 2004.

BASCH, M.; SANCHEZ, A. **Incorporating aspects into the UML.In Proceedings of the AOM** workshop at AOSD, 2003, 2003.

BRUNELIERE H., J. CABOT, F. JOUAULT, F. MADIOT, **"MoDisco: A generic and extensible framework for model driven reverse engineering,"** IEEE/ACM international conference on Automated software engineering, ACM New York, NY, USA, 2010, pp. 173-174.

CAMARGO, V. V. de; RAMOS, R. A.; PENTEADO, R.; MASIERO, P. C. **"Projeto Orientado a Aspectos do Padrão Camada de Persistência,"** 17º Simpósio Brasileiro de Engenharia de Software (SBES), Manaus-Amazonas, Brasil, outubro, 2003, pp. 114-129.

CAMARGO, V. V. de, MASIERO, P. C., **"Frameworks Orientados a Aspectos,"** XIX Simpósio Brasileiro de Engenharia de Software, Uberlândia. 2005, pp. 200-216.

CAMARGO, V. V. de; MASIERO, P. C. **"An Approach to Design Crosscutting Framework Families,"** ACP4IS 08, Brussels, Belgium, 2008.

CHAVEZ, C.; LUCENA, C.. **A metamodel for aspect-oriented modeling.** In Proceedings of the AOM with UML workshop at AOSD, 2002, 2002.

CHIDAMBER, S.; KEMERER, C. **"A Metrics Suite for Object Oriented Design"**. IEEETransactions on Software Engineering, 20 (6), June 1994, pp. 476-493.

COUTO, C. F. M.; VALENTE, M. T. O.; BIGONHA, R. da S. **“Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência,”** 2º. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP’05), evento satélite do XIX SBES, Uberlândia, MG, Brasil, outubro, 2005.
ENGELHARDT, M. *et al*. Generation of Formal Model Metrics for MOF based Domain Specific Languages. **The Pragmatics of OCL and Other Textual Specification Languages.**V. 24. 2009

EVERMANN, J. **"A meta-level specification and profile for AspectJ in UML."** Proceedings of the 10th international workshop on Aspect-oriented modeling. ACM, 2007.

EVERMANN, J. **“An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks”**, Workshop AOM ’07, Vancouver, British Columbia, Canada, 2007.

FENTON, N.;PFLEEGER, S. **“Software Metrics: A Rigorous and Practical Approach”**. 2.ed.London: PWS, 1997.

FUENTES, L.; SANCHEZ, P.**Elaborating UML 2.0 profiles for AO design**.In Proceedings of the AOM workshop at AOSD, 2006, 2006.

GAMMA, E. *et al.***Design Patterns: Elements of Reusable Object-Oriented Software.**Addison-Wesley, First Edition, 1995

GOTTARDI, T.; PENTEADO, R. A. D.; CAMARGO, V. V. de. **A Process for Aspect-Oriented Platform-Specific Profile Checking.**In Proceedings of the 2011 International Workshop on Early Aspects. New York, NY , USA. 2011.

GRUNDY, J.; PATEL,R. **Developing software components with the UML, Enterprise Java Beans and aspects**.In Proceedings of ASWEC 2001, Canberra, Australia, 2001.

HANENBERG, S.**“Multi-Design Application Frameworks,”** Generative and Component-Based Software Engeneering Young Reaearchers Workshop, Erfurt, October 10, 2000.

HANNEMANN, J. ; KICZALES, G. **Overcoming the prevalent decomposition of legacy code.** In Proc. of Workshop on Advanced Separation of Concerns at the International Conference on Software Engineering (ICSE), Toronto, Canada, 2001.

IZQUIERDO, J.;MOLINA, J. An Architecture-Driven Modernization Tool for Calculating Metrics, **Software, IEEE, v. 27 p. 37-43,** 2009

IZQUIERDO, J.; ZAPATA, B.; MOLINA, J.Definición y ejecución de métricas enel contexto de ADM, **VII Workshop sobre desenvolvimento de software dirigido por modelos** (DSDM), 2010

KANDE, M.; KIENZLE, J.; STROHMEIER, A. **From AOP to UML - a bottom-up approach.** In Proceedings of the AOM with UML workshop at AOSD, 2002, 2002.

KDM Guide.**Knowledge Discovery Meta-Model**, August 2011. Document omg/formal/2011-08-04.

KICZALES, G. *et al.***"Aspect-Oriented Programming, In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland."**Springer-Verlag LNCS 1241 (1997) p. 220-242.

LADDAD, R.**AspectJ in Action: Practical Aspect-Oriented Programming**, Manning Publications, Greenwich (74° w. long.), 2003, pp.75-77.

MIRSHAMS, P. S. **"Extendingthe Knowledge Discovery Metamodel to SupportAspect-OrientedProgramming"**, 79 f. Dissertação (Mestrado em Ciência aplicada em Engenharia de Software) – Departamento de Ciência da Computação e Engenharia de Software, Universide de Montreal, Quebec, Canada, 2011, unpublished.

NORMANTAS, K.; SOSUNOVAS, S.; VASILECAS, O. **"An Overview of the Knowledge Discovery Meta-Model,"** International Conference on Computer Systems and Technologies - CompSysTech'12, 2012.

Object Management Group. **Architecture-Driven Modernization (ADM): Abstract Syntax Tree Metamodel (ASTM) Version 1.0.** Disponível em: <http://www.omg.org/spec/ASTM/1.0/>. Acesso em: 10 de Agosto de 2014.

Object Management Group. **Architecture-Driven Modernization (ADM): Structured Metrics Meta-Model (SMM) Version 1.0**. Disponível em: <http://www.omg.org/spec/XMI/2.4.1/>, 2011. Acesso em: 10 de Agosto de 2014.

Object Management Group. OMG **MOF 2 XMI Mapping Specification Version 2.4.1**.Disponível em: <http://www.omg.org/spec/SMM/1.0/>. Acesso em: 10 de Agosto de 2014.

Object Management Group. **MDA Specifications.**Disponível em: <http://www.omg.org/mda/specs.htm>Acesso em:10 de Agosto de 2014.

Object Management Group. **OMG Specifications**, April 2014. Documents omg. Disponível em: <http://www.omg.org/spec/>. Acesso em: 10 de Agosto de 2014.

PAWLAK, R.; DUCHIEN, L., FLORIN, G.; LEGOND-AUBRY, F.; SEINTURIER, L.; MARTELLI, L.**AUML notation for aspect-oriented software design**.In Proceedings of the AOM with UML workshop at AOSD, 2002, 2002.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. de, PIATTINI, M. **"Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems,"** Computer Standards & Interfaces 33, pp. 519–532.Elsevier B.V, 2011.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. de; PIATTINI, M."**Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems".** Computer Standards & Interfaces, v. 33, n. 6, p. 519-532, 2011b

PRESSMAN, R. **Engenharia de Software**, 7ª edição. São Paulo: McGraw-Hill, 2011.

RASHID, A.; CHITCHYAN, R. **"Persistence as an Aspect,"** 2nd International Conference on Aspect Oriented Software Development (AOSD), Boston–USA, March, 2003.

RAUSCH, A.; RUMPE, B.; HOOGENDOORN L. **"Aspect-Oriented Framework Modeling,"** 4th AOSD Modeling with UML Workshop (UML Conference 2003) October, 2003.

SADOVYKH, A. *et al*. **Architecture Driven Modernization in Practice – Study Results**. 14th IEEE International Conference on Engineering of Complex Computer Systems. 2009

SANTIBANEZ, D. G. S. M. ; DURELLI, R. S. ; CAMARGO, Valter Vieira. **CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process.** Em: Session Tools - CBSoft 2013 (Congresso Brasileiro de Software), 2013.

SANT'ANNA, C. *et al.* **On the Reuse and Maintenance of Aspect-Oriented Software: Na Assessment Framework.** In Proceedings XVII Brazilian Symposium on Software Engineering. 2003

SANT'ANNA, C. *et al.* **On the Modularity of Software Architectures: A Concern-Driven Measurement Framework.** Lecture Notes in Computer Science Volume 4758, pp 207-224. 2007

SANTOS *et al.*, 2014a – WMOD **"Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization"**, Bruno Santos, Rafael S. Durelli, Raphael R. Honda, Valter V. Camargo

SANTOS *et al.*, 2014b – SBES (SIMPÓSIO BRASILEIRO DE Engenharia de software) **"KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel"**, Bruno Santos, Rafael S. Durelli, Raphael R. Honda, Valter V. Camargo

SILVA, B. C. **Um Método de Refatoração para Modularização de Interesses Transversais**, Dissertação de Mestrado. Universidade Federal do Rio Grande do Sul. 2009.

SHAHSHAHANI, M. P. **Extending The Knowledge Discovery Metamodel to Support Aspect-Oriented Programming**, Dissertação de Mestrado. Concordia University. 2011

SOARES, S.; LAUREANO, E.; BORBA, P. **"Implementing Distribution and Persistence Aspects with AspectJ,"** 17th ACM Conference on Object-Oriented

Programming, Systems, Languages, and Applications (OOPSLA), November, 2002, pp 174-190.

SOREN, F *et al*. **MAMBA: A Measurement Architecture for Model-Based Analysis**, Department of Computer Science, University of Kiel, Germany, number TR-1112, 2011.

SOREN, F *et al*. MAMBA: Model-Based Analysis Utilizing OMG's SMM, in: **Proceedings of the 14. Workshop Software-Reengineering (WSR '12)**, Bad Honnef, Germany, May 2-4, 2012, pages 37-38, 2012

STEIN, D.; HANENBERG, S.; UNLAND, R. **Designing aspect-oriented crosscutting in UML.**In Proceedings of the AOM with UML workshop at AOSD, 2002, 2002.

VISAGGIO,G., **"Ageing of a data-intensive legacy system: symptoms and remedies,"** Journal of Software Maintenance 13. 2001, pp. 281–308.

YAN,H.;KNIESEL,G.;CREMERS,A. **A meta model and modeling notation for AspectJ.** In Proceedings of the AOM workshop at AOSD, 2004, 2004.