UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EXTENSÕES DO METAMODELO KDM PARA APOIAR MODERNIZAÇÕES ORIENTADAS A ASPECTOS DE SISTEMAS LEGADOS

ORIENTADOR: PROF. DR.VALTER VIEIRA DE CAMARGO

São Carlos - SP
Outubro/2014

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EXTENSÕES DO METAMODELO KDM PARA APOIAR MODERNIZAÇÕES ORIENTADAS A ASPECTOS DE SISTEMAS LEGADOS

BRUNO MARINHO SANTOS

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
Outubro/2014

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

S237em

Santos, Bruno Marinho.
Extensões do metamodelo KDM para apoiar modernizações orientadas a aspectos de sistemas legados / Bruno Marinho Santos. -- São Carlos : UFSCar, 2015.
124 f.

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

1. Engenharia de software. 2. *Architecture-Driven Modernization* - ADM. 3. Knowledge Discovery Metamodel - KDM. 4. Extensões leves e pesadas. 5. POA. 6. Estudo de casos. 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

"Extensões do Metamodelo KDM para Apoiar Modernizações Orientadas a Aspectos de Sistemas Legados"

Bruno Marinho Santos

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)

Profa. Dra. Maria Istela Cagnin Machado
(UFMS)

Profa. Dra. Rosângela Dellosso Penteado
(DC/UFSCar)

São Carlos
Outubro/2014

À minha mãe, irmã e irmão.

AGRADECIMENTO

Primeiramente a Deus.

Ao meu orientador, Prof. Dr. Valter Vieira de Camargo, pela confiança ao me orientar, pelas constantes lições, incentivando sempre a buscar algo a mais, pela paciência nas horas de discussão, pela excelente orientação e pelo exemplo de pessoa e profissional.

À minha mãe, pelo exemplo de luta, coragem, positivismo, apoio e alegria. Ao meu pai, por ter investido desde os primeiros anos de minha vida em uma educação de qualidade e pelo exemplo de caráter. À minha irmã, exemplo de sinceridade, esforço, alegria e dedicação. Ao meu irmãozinho que enche meu coração de alegria só em saber que ele existe. Aos meus avós, tios mais próximos e primos, em especial Isabela, Rafaela, Lais, Cristiano, Francisco, Roger, Vamberto, Teresa.

À toda minha família de Teresina, que me acolheram com muito amor, em especial tia Graça, tia Socorro, tio Chico, Hamilton, Deusa, Nilson, Raquel, Emanuel e Leomar.

Aos amigos que fiz no mestrado Amandia, André, Odair, Thiago Lacerda, Raphael Honda, Rafael Durelli, Victor, Daniel, Matheus Viana, Mirela, Mauro, Juliana e Isis. Aos colegas da turma de 2012 do PPGCC/DC.

Aos amigos de longa data Matheus Melo, Joyce, Jullyane, Ana, Willardy, Suanny, Lara, Talita, Karol, Karinny e Plínio.

Aos professores que me encorajaram de alguma forma, desde a graduação, a seguir meus estudos na pesquisa. Em especial aos professores Renata, Edileuza, Vinícios, Leonardo, Silvie, Ricardo Queiroz e Ricardo Albano.

Aos funcionários do Departamento de Computação da UFSCar, em especial a tia Vera e Augusto, por proporcionar serviços e infraestrutura adequada, de modo a possibilitar o desenvolvimento de nossos trabalhos.

À CAPES, pelo apoio financeiro.

À todos que, de alguma forma, contribuíram com a realização desse trabalho.

Obrigado!

RESUMO

Manter sistemas legados é uma atividade complexa e onerosa para muitas empresas. Uma proposta recente para esse problema é a Modernização Dirigida à Arquitetura (*Architecture-Driven Modernization* - ADM), proposta pela OMG (*Object Management Group*). A ADM consiste em um conjunto de princípios e metamodelos padrões que apoiam a modernização de sistemas utilizando modelos. O *Knowledge Discovery Metamodel* (KDM) é o principal metamodelo da ADM, podendo representar diversos artefatos de um sistema, como código-fonte, arquitetura, interface de usuário, arquivos de configuração e processos de negócio. Em geral, sistemas legados possuem interesses transversais, apresentando problemas de entrelaçamento e espalhamento de código, o que eleva os custos de manutenção. A orientação a aspectos é uma alternativa para melhorar a modularização de interesses transversais. Mediante isso, neste trabalho é apresentado o termo Modernização Orientada a Aspectos que utiliza os conceitos da orientação a aspectos na ADM. Essa modernização consiste em remodularizar sistemas legados utilizando aspectos representados em nível de modelo. Para atingir esse objetivo, foi realizada uma extensão leve e outra pesada do metamodelo KDM, para analisar em qual das duas o desempenho dos engenheiros de modernização seria melhor. Para fazer a avaliação das extensões, foi realizado um estudo de caso levando em consideração a modernização com aspectos em um sistema de pequeno porte. Com o objetivo de avaliar o estudo de caso usando as duas extensões, foram desenvolvidos critérios de comparação que auxiliassem os engenheiros de software a escolher qual dos dois mecanismos de extensão utilizar em seu projeto. Foi feito também um estudo experimental que buscou reproduzir os cenários em que engenheiros de modernização tivessem que realizar manutenções e desenvolver novas refatorações em um modelo KDM orientado a aspectos. Os dados do experimento foram avaliados em relação ao tempo de desenvolvimento das atividades e quantidade de erros encontrados. Por fim, percebeu-se que o mecanismo de extensão a ser utilizado vai depender do contexto em que ele será aplicado, mas, para o domínio aqui estudado a extensão que melhor atendeu aos requisitos foi a pesada.

Palavras-chave: ADM, KDM, Extensões, Leve, Pesada, POA, Engenheiro de modernização, Estudo de caso, Experimento.

ABSTRACT

Maintaining legacy systems is a complex and expensive activity for many companies. A recently proposal to solve this problem is Architecture-Driven Modernization (ADM), proposed by Object Management Group (OMG). The ADM consists of a set of concepts and standard metamodels that support systems modernization using models. The Knowledge Discovery Metamodel (KDM) is the main metamodel of ADM, it can represent many artifacts of a legacy system, such as source code, architecture, user interface, configuration files and business process. In general, legacy systems have crosscutting concerns, it can show source code problems like tangling and scattering, and it raises the maintenance costs. The aspect orientation is an alternative to improve crosscutting concerns modularization. Thus, in this dissertation is presented the term Aspect Oriented Modernization that uses the aspect oriented concepts in the ADM context. This modernization process consists in modularize legacy systems with aspects represented in model level. To achieve this goal, in this work were performed a lightweight and a heavyweight extension in the KDM metamodel, to analyze which one would present a better performance if used by Modernization Engineers. The evaluation of these extensions was performed by a case study that considered the modernization with aspects of a small-sized system. To evaluate the case study in both extensions, a set of comparison criteria were created to support the software engineers in choosing the best extension mechanism, according to their needs. In the context of this dissertation an experimental study were developed that aimed reproducing the scenarios that the modernization engineers had to perform maintenances and developing new refactorings in a aspect oriented KDM model. The experiment data considered the development time of the activities and the found number of errors. Finally, it was noticed that the extension mechanism to be choose will depend on the context that it will be applied, however, considering the approach studied here the best extension mechanism is the heavyweight one.

Keywords: ADM, KDM, Extension Mechanism, Lightweight, Heavyweight, Aspect Oriented Programming, Modernization Engineer, Study Case, Experiment.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADM - *Architecture-Driven Modernization*

ADMTF - *Architecture-Driven Modernization Task Force*

AMMA - ATLAS *Model Management Architecture*

AspeCiS - *An Aspect approach to develop Cooperative Information Systems*

ATL - ATLAS *Transformation Language*

CCKDM - *CrossCutting Concern Knowledge Discovery Metamodel*

CIM - *Computing Independent Model*

COMO - *Component-Oriented Modernization*

DSOA - Desenvolvimento de Software Orientado a Aspectos

EMF - *Eclipse Modeling Framework*

FOA - Framework Orientado a Aspectos

FTs - Frameworks Transversais

IDE - *Integrated Development Environment*

KDM - *Knowledge Discovery Metamodel*

KDM-OO - KDM Orientado a Objetos

KDM-AO - KDM *Aspect Oriented*

KDM-SDK – KDM - *Software Development Kit*

MDA - *Model-Driven Architecture*

OMG - *Object Management Group*

PIM - *Platform Independent Model*

POA - Programação Orientada a Aspectos

POO - Programação Orientada a Objetos

PSM - *Platform Specific Model*

RFP - *Request-For-Proposal*

UFSCar – Universidade Federal de São Carlos

UML - *Unified Modeling Language*

XMI - *Metadata Interchange*

XML - *eXtensible Markup Language*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Para que um sistema de informação continue satisfazendo seus requisitos previamente estabelecidos é necessário que seja evoluído continuamente ou deixará de cumprir seu papel adequadamente. Muitas empresas possuem sistemas que sofreram, com o passar do tempo, o fenômeno da erosão e envelhecimento, fazendo com que sejam considerados "legados". A erosão e o envelhecimento consistem no desgaste de um sistema em consequência de sucessivas alterações que sofreram, como por exemplo, funcionalidades que são retiradas, modificadas ou adicionadas e que comprometem assim sua qualidade (Visaggio, 2001).

Para muitas empresas, a completa substituição de seu sistema legado possui alto risco e consome uma grande quantidade de recursos que torna essa alternativa inviável. Já a reengenharia de sistemas é uma alternativa que tende a prolongar o tempo de vida útil do sistema e ser mais viável economicamente (Pérez-Castillo *et al*., 2011a). Entretanto, a reengenharia tradicional possui dois problemas principais. O primeiro é sua complexidade, tornando-a difícil de automatizar e dificultando bastante sua realização. Outro problema é que os tradicionais processos de reengenharia carecem de formalização e padronização, uma vez que não existe um guia de como executá-los passo a passo (Pérez-Castillo *et al*., 2011b).

Em 2003 a *Object Management Group*[1] (OMG) criou uma força tarefa para analisar e evoluir os tradicionais processos de reengenharia, formalizando-os e fazendo com que eles fossem apoiados por modelos (OMG, 2013). Logo, o conceito típico de reengenharia começou a mudar e o termo Modernização Dirigida à Arquitetura (*Architecture-Driven Modernization* - ADM) surgiu como uma solução para os problemas de padronização. A ADM é um processo de modernização de sistemas legados que utiliza um conjunto de metamodelos para descrever um sistema em diferentes representações arquiteturais (OMG, 2013).

A ADM preconiza que processos de modernização devem seguir os padrões já definidos da Arquitetura Orientada a Modelo (*Model-Driven Architecture* - MDA): *Platform Specific Model* (PSM), *Platform Independent Model* (PIM) e *Computational Independent Model* (CIM). Dessa forma, durante a modernização de um sistema são gerados vários modelos que representam diferentes partes do sistema, como: fluxo de dados, banco de dados, elementos de programação (métodos, classes, tipos de dados, etc.), arquitetura e processos de negócio (Pérez-Castillo *et al*., 2011a).

O *Knowledge Discovery Metamodel* (KDM) é o principal metamodelo da ADM e tem como objetivo representar as principais características de um sistema, desde detalhes de baixo nível, como linhas de código, até conceitos abstratos não presentes em uma linguagem de programação, como módulos arquiteturais e processos de negócio. Originalmente (e propositalmente) o KDM não inclui metaclasses específicas de domínios/subdomínios/tecnologias particulares, como serviços web (*web services*), sistemas multiagentes, nuvens e programação orientada a aspectos. A filosofia é permitir que engenheiros de modernização possam adaptar o metamodelo quando for necessário representar esses detalhes. Há dentro do KDM, pacotes dedicados à criação de extensões leves (*lightweight extensions*) desse metamodelo por meio do uso de estereótipos e etiquetas valoradas. Outra forma de estender o KDM é modificá-lo diretamente, criando novas metaclasses e/ou modificando as existentes. Esse tipo de extensão é conhecida como "pesada" (*heavyweight*).

Quando se trata de modernizar um sistema, uma possível tecnologia que pode ser usada é a programação orientada a aspectos, que possui abstrações para melhorar os níveis de modularização dos interesses transversais. Nesse contexto,

[1] OMG é uma organização internacional que aprova padrões abertos para aplicações orientadas a objetos desde 1989.

destacam-se os Frameworks Transversais (*Crosscuting Frameworks* ou FTs), que são frameworks orientados a aspectos e tratam de um único interesse transversal como persistência, segurança e concorrência (Camargo e Masiero, 2005). Esse tipo de framework possui mecanismos de composição abstratos como conjuntos de junção (*pointcuts*) da linguagem AspectJ e variabilidades correspondentes ao interesse que ele encapsula (Camargo e Masiero, 2005).

A proposta original da ADM, principalmente o KDM, não dá suporte à utilização dos conceitos da orientação a aspectos durante o processo de modernização. Como o paradigma Orientado a Aspectos tem sido constantemente referenciado como uma alternativa interessante para se modularizar interesses transversais, e considerando que esses são problemas inerentes de sistemas legados, vislumbrou-se uma oportunidade de conduzir essa pesquisa no contexto de "modernização orientada a aspectos".

1.2 Motivação

Considerando que interesses transversais são inerentes a muitos sistemas legados e que o paradigma orientado a aspectos é bom candidato para melhorar a modularização desses sistemas, considera-se importante viabilizar a condução de modernizações orientadas a aspectos no contexto da ADM.

Nesse sentido, pôde-se observar na literatura carência de estudos sobre i) representação dos conceitos da Programação Orientada a Aspectos (POA) no metamodelo KDM e ii) comparação entre os diferentes tipos de extensão do metamodelo KDM. Sem a adequada representação dos conceitos da orientação a aspectos nesse metamodelo, a realização de uma modernização desse tipo pode se tornar propensa a erros. Isso acontece porque as abstrações da POA teriam que ser representadas com conceitos Orientados a Objetos (OO), o que acarreta em perda de informação semântica. Por exemplo, é difícil representar um conjunto de junção somente com conceitos OO.

Com relação à abordagens que apresentam extensões do KDM para aspectos, foi encontrado apenas um trabalho na literatura. Mirshams (2011) usa um compilador existente como apoio para a recuperação automática de modelos KDM a

partir de código em AspectJ. Entretanto, a extensão apresentada por ela é mais genérica do que a apresentada aqui e muitos conceitos são representados de forma mais simples.

O segundo fator motivador deste trabalho é a ausência de trabalhos que comparam extensões leves e pesadas do KDM e fornecem diretrizes de uso de ambas.

1.3 Objetivos e Contribuições

Neste trabalho de mestrado, o principal objetivo é viabilizar a condução de modernizações orientadas a aspectos por meio do fornecimento de duas extensões do metamodelo KDM para representar os conceitos desse paradigma. Duas extensões, uma leve (*lightweight*) e uma pesada (*heavyweight*) foram criadas permitindo que as abstrações da POA, como pontos de junção (*join points*), conjuntos de junção (*pointcuts*), adendos (*advices*) e outras possam ser representados nesse metamodelo.

O segundo objetivo é auxiliar engenheiros de modernização na escolha de qual tipo de extensão (leve ou pesada) melhor atende as suas necessidades. Assim, realizou-se um estudo comparativo para delinear as vantagens/desvantagens e principais diferenças entre ambas.

Um terceiro objetivo neste projeto é apresentar uma forma de estender o metamodelo KDM com base em perfis existentes da *Unified Modeling Language* (UML) (*UML profiles*). Assim como a POA é uma alternativa interessante de modernização, outras tecnologias também são, como por exemplo, serviços web, computação em nuvem, agentes, componentes, sistemas adaptativos e sistemas embarcados. Dessa forma, perfis UML existentes para essas tecnologias podem ser usados como base para a criação de extensões do KDM. Para isso foi disponibilizado neste projeto uma tabela de mapeamento entre o metamodelo da UML e o metamodelo KDM, que facilita a criação de extensões.

1.4 Cenário de Modernização com FTs

O cenário de modernização em que as extensões OAs do KDM se tornam necessárias é representado esquematicamente pela Figura 1.1 - Modernização Orientada a Aspectos. Na parte inferior esquerda da figura encontra-se um sistema legado que possui problemas de modularização, por exemplo, com a presença de um interesse transversal de persistência. O objetivo da modernização é obter uma nova versão orientada a aspectos desse sistema em que os problemas de modularização estejam resolvidos. Considere também a existência de um repositório em que vários FTs estejam disponíveis na forma de instâncias KDM-OA.

Por meio de um processo automático de engenharia reversa, realizado por uma ferramenta chamada MoDisco (Bruneliere *et al*., 2010), obtém-se uma instância do metamodelo KDM que representa o sistema "as is", isto é, com os mesmos problemas de entrelaçamento e espalhamento presentes no código-fonte. Em vários pontos deste texto esse modelo também será chamado de "KDM legado". Em seguida, há a necessidade de um processo de mineração de interesses para marcar os elementos de modelo que contribuem para a implementação dos interesses transversais existentes, que pode ser feito com auxílio da ferramenta *CrossCutting Concern Knowledge Discovery Metamodel* (CCKDM) proposta por Santibáñez *et al*. (2013). A saída desse processo é um outro KDM com determinados interesses claramente anotados.

O próximo passo, denominado "Refatorações", consiste em uma transformação de modelos que i) remove os elementos marcados do KDM legado e ii) realiza o processo de reúso de um dos FTs existentes por meio da criação de um "modelo de instanciação". Na figura, ilustra-se a escolha de um FT de Persistência do repositório. O modelo de instanciação faz a ligação entre a aplicação base e o(s) FT(s) escolhido(s). Essa representação pode ser vista logo abaixo da palavra "Refatorações" e consiste em criar uma instância KDM que armazene as informações de quais classes, interfaces, métodos e atributos do KDM legado serão afetados pelo FT escolhido. Vale lembrar que cada FT exige a criação de um modelo de instanciação, dessa forma, se o engenheiro de aplicação escolher três FTs, existirão três modelos de instanciação.

Em seguida, gera-se uma nova versão modernizada do KDM, denominada aqui de KDM modernizado, com aquele determinado interesse transversal modularizado.

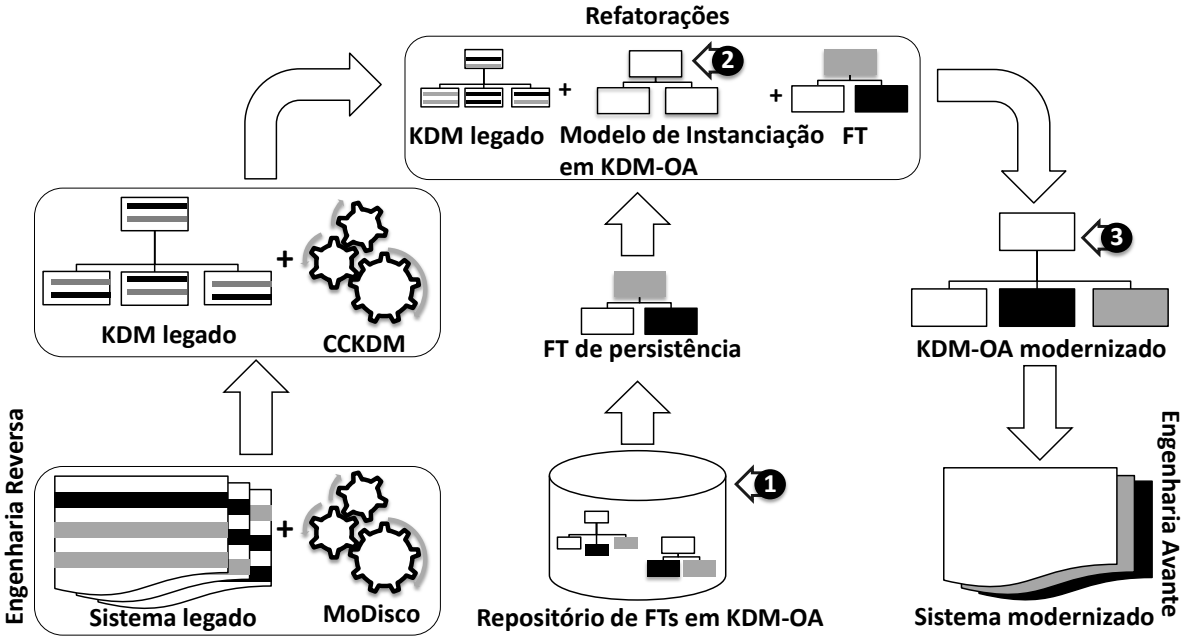


Figura 1.1 - Modernização Orientada a Aspectos

Nesse cenário apresentado, as extensões do KDM (pesada ou leve) são necessárias em três momentos; marcadas na figura com o símbolo . O primeiro é para representar os FTs disponíveis do repositório. Como esses frameworks são orientados a aspectos, eles contém elementos aspectuais e apenas com um KDM estendido é possível fazer essa representação. O segundo é para representar o modelo de instanciação, já que ele também contém elementos aspectuais e o terceiro é no KDM modernizado, que também contém elementos aspectuais.

Note-se que as demais partes não são tratadas neste projeto. O processo de mineração do KDM foi desenvolvido em um trabalho anterior (Santibáñez *et al*., 2013) e o processo de refatoração está sendo desenvolvido por um aluno de doutorado (Durelli *et al*., 2014b).

1.5 Organização do Trabalho

Esta monografia de dissertação está organizada em oito capítulos. No primeiro capítulo estão apresentados o contexto, a motivação e os objetivos do trabalho. No Capítulo 2 são enfatizados os conceitos do desenvolvimento de software orientado a objetos e Frameworks Transversais.

No Capítulo 3 é ilustrada uma contextualização sobre a modernização de sistemas legados com a utilização dos padrões propostos pela OMG, ou seja, ADM e KDM, em que seus conceitos e particularidades são observados. Também é apresentada a ferramenta MoDisco, utilizada neste trabalho para auxiliar no desenvolvimento das abordagens aqui propostas.

No Capítulo 4 são discutidas as duas formas de extensão do metamodelo KDM (Leve e Pesada) e também é mostrado como essas duas extensões foram desenvolvidas para suportarem os conceitos da orientação a aspectos.

Já no Capítulo 5 é mostrada a primeira parte da avaliação do trabalho, que foi feita por meio de um estudo de caso para as duas extensões. No mesmo capítulo é mostrada uma avaliação comparativa das duas abordagens por meio da criação de alguns critérios de avaliação.

No Capítulo 6 são mostrados o planejamento, execução e análise dos dados de um experimento que visou validar os resultados obtidos nos capítulos anteriores.

No Capítulo 7 são mostrados os principais trabalhos relacionados que foram encontrados na literatura, assim como uma discussão das semelhanças e diferenças de cada um em relação a esta dissertação.

E, por último, no Capítulo 8 são mostradas as conclusões do trabalho com as principais contribuições, limitações, lições aprendidas e trabalhos futuros.

Capítulo 2

DESENVOLVIMENTO DE SOFTWARE ORIENTADO A ASPECTOS E FRAMEWORKS TRANSVERSAIS

2.1 Considerações Iniciais

Este capítulo apresenta uma abordagem conceitual do desenvolvimento de software orientado a aspectos. Os conceitos da POA são de suma importância no contexto desta dissertação e serão utilizados no decorrer de todos os capítulos. Assim, são descritos neste capítulo os principais conceitos dessa linguagem de programação, tais como aspectos, pontos de junção, conjuntos de junção, adendos e declarações inter-tipo. Outro tópico discutido são os FTs, que são frameworks que tem como objetivo modularizar um sistema em relação a determinado interesse transversal. Neste trabalho, os FTs são utilizados para dar suporte nos capítulos de avaliação, logo, uma análise dos seus principais conceitos é necessária, bem como sua exemplificação de uso.

O capítulo está organizado da seguinte forma: na Seção 2.2 são descritos os principais conceitos da Programação Orientada a Aspectos; na Seção 2.3 são descritos os conceitos de Frameworks Transversais, na Seção 2.4 são mostrados exemplos de instanciação de um FT e na Seção 2.5 são mostradas as considerações finais do capítulo.

2.2 Programação Orientada a Aspectos

O conceito de POA foi introduzido por Kiczales *et al*. (1997) no final dos anos 90 com o intuito de modularizar os chamados interesses transversais (*crosscutting concerns*). A POA é uma tecnologia que permite a separação de interesses transversais introduzindo uma nova unidade de modularização conhecida como aspecto que entrecorta outros módulos. Utilizando POA é possível implementar interesses transversais em aspectos, ao invés de escrevê-los junto com os módulos principais (Laddad, 2003).

Dessa forma, Kiczales *et al*. (1997) definem dois termos importantes que são os componentes e os aspectos. Se uma unidade funcional puder ser encapsulada em um procedimento generalizado, pode-se dizer que ele é um componente ou um interesse base. Interesses base tendem a ser unidades funcionais do sistema, tais como dados de entrada e saída, lógica de negócios e outros.

Já o aspecto não é uma unidade funcional do sistema, e sim, propriedades que afetam o desempenho ou a semântica dos interesses base. O termo “interesse transversal” faz analogia ao fato de que sua implementação com técnicas tradicionais de programação entrecorta transversalmente os módulos do sistema, afetando vários módulos e causando entrelaçamento (*tangling*) e espalhamento (*spreading*) de código de diferentes interesses. O entrelaçamento ocorre quando o código de um determinado interesse encontra-se misturado com o código de outro interesse dentro de um mesmo módulo. O espalhamento ocorre quando o código de um interesse encontra-se em vários módulos do sistema. O entrelaçamento e o espalhamento de código de diferentes interesses causam problemas de manutenção, reúso e evolução (Kiczales *et al*., 1997).

A vantagem de se utilizar a POA é que os interesses transversais são separados em módulos, dessa forma a arquitetura do sistema pode ser mais facilmente projetada, implementada e mantida (Laddad, 2003).

A POA mudou a forma como o software é desenvolvido e baseia-se não apenas em técnicas modulares como as programações procedurais e orientadas a objeto, mas também apoia a modularização de interesses transversais. A POA permite o desenvolvimento de projetos modulares e implementações de interesses

como políticas de sincronização, regras de negócios, gestão de recursos, a aplicação da arquitetura, regras de segurança, interação, camada de persistência, otimizações de desempenho e muitos outros (Kiczales, 2005).

Na POA, os aspectos são programados separadamente das regras de negócio, que por sua vez, são escritas em classes. Para fazer a composição do sistema final é necessário utilizar um mecanismo que una os aspectos e as classes, que na POA, esse mecanismo é conhecido por aspecto *Weaver*. Um aspecto *Weaver* é uma entidade responsável por fazer a composição do sistema final combinando os módulos principais com os módulos transversais, por meio de um processo chamado *weaving* (Laddad, 2003).

Para que se possa programar em orientação a aspectos é necessário ter conhecimento sobre suas abstrações de modularização e mecanismos de composição, que juntos melhoram a separação de interesses transversais. No total são cinco conceitos introduzidos pela POA, são eles: aspectos (*aspects*), pontos de junção (*join points*), conjuntos de junção (*pointcuts*), adendos (*advices*) e declarações inter-tipos (*inter-type declarations*) (Kiselev, 2003).

Como dito anteriormente, um aspecto corresponde a um interesse transversal e constitui uma unidade modular projetada para afetar um conjunto de classes e objetos do sistema. Os pontos de junção são os locais do programa em que os aspectos podem atuar, por meio deles é possível especificar o relacionamento entre aspecto e classe. O AspectJ define onze tipos de pontos de junção que podem ser reconhecidos pelo compilador, que incluem chamadas de métodos, execução de exceções, atribuição de campo e outros, cada um com uma sintaxe diferente (Kiselev, 2003).

Um conjunto de junção é o mecanismo que especifica os pontos de junção em que aspectos e classes se relacionem. O adendo é um construtor semelhante a um método de uma classe, em que é definido o comportamento dinâmico que é executado quando são alcançados um ou mais conjuntos de junção. E as declarações inter-tipos podem ser utilizadas quando se deseja introduzir atributos e métodos em classes-base do sistema (Kiselev, 2003).

Da mesma forma que em uma classe abstrata em Java, a linguagem AspectJ também permite a criação de aspectos abstratos, só que além de métodos abstratos o aspecto também pode ter conjuntos de junção abstratos. Um conjunto de junção

abstrato não tem conhecimento dos pontos de junção que serão afetados, pois esses pontos devem ser informados em um aspecto concreto que especializar o aspecto abstrato. Como um adendo pode ser definido sobre um conjunto de junção abstrato, é possível implementar um comportamento transversal no aspecto abstrato sem saber antes quais são os pontos de junção, e qual é o código-base em que esse comportamento atuará (Elrad *et al*., 2001).

Aspectos abstratos são a base do desenvolvimento orientado ao reúso, e dessa forma, do desenvolvimento de Frameworks Orientados a Aspectos (FOA).

2.3 Frameworks Transversais

No contexto do Desenvolvimento de Software Orientado a Aspectos (DSOA), vários autores tem empregado aspectos abstratos como a principal técnica para o projeto e implementação de frameworks (Rashid and Chitchyan, 2003; Couto *et al*., 2005; Camargo *et al*., 2003; Soares *et al*., 2002; Rausch *et al*., 2003). Entretanto, em consequência da falta de consenso sobre termos usados nessa área, Camargo e Masiero (2005) apresentaram uma terminologia para frameworks que empregam aspectos em sua arquitetura. O termo mais relevante apresentado por Camargo e Masiero são os "Frameworks Transversais", que são um tipo de Framework Orientado a Aspectos que encapsulam comportamento genérico de apenas um único interesse transversal como concorrência, segurança e persistência. Os FTs possuem mecanismos de composição abstratos e variabilidades funcionais, podendo ser classificados como “dependentes de contexto” e “independentes de contexto”. Os FTs independentes de contexto podem ser facilmente acoplados em qualquer código base por não precisarem de detalhes específicos dos pontos de junção. Exemplos de FTs independentes de contexto são os que implementam os interesses de Log e de Rastreamento (Camargo e Masiero, 2008).

Os FTs dependentes de contexto precisam obter um objeto ou valor do código base para que possam funcionar corretamente. Um exemplo de FT que se encaixa nessa categoria é o interesse Controle de Acesso, pois, ele precisa obter os dados de autenticação do usuário para verificar se os usuários podem ou não acessar uma funcionalidade específica do sistema. Esses tipos de FTs são mais complexos e

devem ser projetados com um padrão que forneça várias alternativas de composição (Camargo e Masiero, 2008).

Outra característica dos FTs é que eles só podem ser utilizados se forem acoplados a algum código-base existente, ou seja, ao se reutilizar um FT uma nova aplicação não será gerada. O processo de reúso desse tipo de framework possui duas etapas distintas: a instanciação e a composição (Camargo e Masiero, 2005).

A instanciação é o processo convencional de reúso dos Frameworks Orientados a Objetos (FOOs) tradicionais, que consistem em especializar o código que foi especificamente projetado para esse objetivo. A instanciação é o processo em que são implementados os ganchos do framework, onde é escolhida alguma funcionalidade alternativa ou até mesmo implementar uma nova (Camargo e Masiero, 2005).

A etapa de composição possui duas atividades, a identificação dos pontos de junção e o fornecimento de regras de composição. A atividade de identificação de pontos de junção consiste em identificar no código-base a existência de um ou mais pontos de junção adequados ao acoplamento e deve ser feita com base nas "alternativas de composição" disponíveis no framework. É importante que os FTs sejam projetados com alternativas de composição, pois, as chances de acoplamento em códigos-base são aumentadas, sem falar que pode diminuir a complexidade das regras de composição que precisam ser fornecidas (Camargo e Masiero, 2005).

Já a atividade de fornecimento de regras de composição consistem em fornecer regras que unem as variabilidades que foram implementas e escolhidas do framework com o código-base, e para isso tarefas orientadas a aspectos devem ser realizadas. Na linguagem AspectJ, por exemplo, tarefas orientadas a aspectos podem ser a concretização de um mecanismo de composição abstrato ou a utilização de declarações inter-tipo (Camargo e Masiero, 2005).

Existem casos em que a etapa de composição precisa de informações que são determinadas na etapa de instanciação, por isso é necessário estabelecer a ordem de que primeiro vem a instanciação do framework, para que só então se possa fazer a composição do mesmo com o código desejado. Nos casos em que não ocorrerem essa dependência, as etapas de instanciação e composição poderão ser realizadas em qualquer ordem ou até mesmo em paralelo (Camargo e Masiero, 2005).

2.4 Exemplo de instanciação de um FT

Como visto, o processo de reutilização de um FT envolve diversas etapas. Nesta seção é exemplificado parte do processo de instanciação do FT persistência proposto por Camargo e Masiero (2005).

O processo de reúso do FT de Persistência, que consiste em acoplá-lo a uma determinada aplicação, possui quatro etapas. Três variabilidades precisam ser determinadas: o tipo da consciência, o tipo da conexão com o banco de dados e o próprio banco de dados. É necessário informar também quais são as classes de aplicação persistentes e quais os locais do código-base em que a conexão deve ser aberta e fechada. Nos parágrafos a seguir são mostrados as aspectos "`MyOORelationalMapping`" e "`myConnectionCompositionRules`", responsáveis por armazenar as classes da aplicação que serão afetadas pelo FT e os locais do código-fonte base de abertura e fechamento de conexão, respectivamente.

```
package instantiation;

import persistence.*;
import application.*;

public aspect MyOORelationalMapping extends OORelationalMapping {

    declare parents: CD            implements PersistentRoot;
    declare parents: LabelColor    implements PersistentRoot;
    declare parents: Music         implements PersistentRoot;
    declare parents: Client        implements PersistentRoot;
    declare parents: Purchase      implements PersistentRoot;
}
```

Figura 2.1 - Código-fonte do aspecto *MyOORelationalMapping*.

Na Figura 2.1Figura 2.1 - Código-fonte do aspecto *MyOORelationalMapping*. é representado o aspecto responsável por determinar as classes que devem ser afetadas pelo FT de persistência. Dessa forma, é necessário criar um aspecto que estenda o aspecto `OORelationalMapping` do FT e declarar quais são as classes a serem persistidas. Na Figura 2.1 as classes CD, LabelColor, Music, Client e Purchase são declaradas para serem afetadas pelo FT.

Já na Figura 2.2 é mostrado um trecho de código-fonte do aspecto `myConnectionCompositionRules` que determina quais são os pontos em que deve ser aberta e fechada a conexão com o banco de dados. O conjunto de junção `openConnection()` identifica os pontos em que a conexão deve ser aberta e o conjunto de junção `closeConnection()` identifica os pontos em que a conexão deve ser fechada. No exemplo, a abertura e o fechamento da conexão ocorrem nas execuções (`Execution`) dos métodos `main(..)` das classes especificadas na Figura 2.1. O método `getNameOfConnectionVariabilitiesClass()` informa em uma String qual o nome da classe que implementa a conexão com o banco de dados. No Exemplo, é o caminho completo da classe `MyConnectionVariabilites`.

```
package instantiation;
import persistence.connection.*;
import application.*;

public aspect myConnectionCompositionRules extends ConnectionComposition {

public pointcut openConnection():
      execution (public static void FindSomeCDs.main(..)) ||
      execution (public static void RemovalOfMusic.main(..));

public pointcut closeConnection():
      execution (public static void FindSomeCDs.main(..)) ||
      execution (public static void RemovalOfMusic.main(..));

public String getNameOfConnectionVariabilitiesClass(){
      return "instantiation.myConnectionVariabilities";
      }
}
```

Figura 2.2 – Trecho de código-fonte do aspecto *myConnectionCompositionRules*.

2.5 Considerações Finais

O objetivo deste capítulo foi analisar e caracterizar os principais conceitos que envolvem a programação orientada a aspectos e frameworks transversais. A programação orientada a aspectos é uma tecnologia bastante recomendada para auxiliar nas modularizações de sistemas, uma vez que ela fornece abstrações

capazes de separar interesses transversais dos interesses base. A utilização de aspectos, adendos, pontos e conjuntos de junção e declarações inter-tipo são fundamentais para realizar essa modularização. Foi visto também que existem na literatura aspectos que podem ser reusados por meio de FTs, facilitando assim o processo de modernização de um sistema. Os FTs precisam de informações do código-fonte base para serem reusado e neste capítulo foram demonstradas as principais formas de reusar FTs.

Os conceitos da POA e de FTs fornecem uma base para o desenvolvimento desta dissertação e são amplamente utilizados nos capítulos a seguir.

Capítulo 3

MODERNIZAÇÃO DE SISTEMAS APOIADA PELO METAMODELO KDM

3.1 Considerações Iniciais

Este capítulo apresenta uma abordagem conceitual da modernização de sistemas e os esforços da OMG para a realização de uma padronização em nível de modelo desses sistemas. O escopo do trabalho é delimitado e se discute especificamente sobre *Architecture-Driven Modernization* (ADM) e *Knowledge Discovery Metamodel* (KDM). ADM é o processo de modernização apoiado por modelos da OMG que utiliza os conceitos da modernização dirigida a modelos. Neste capítulo, são abordados os principais pontos dessa modernização, bem como é apresentada uma descrição mais detalhada do seu principal metamodelo, que é o KDM. Esse capítulo fornece a base necessária para realizar as extensões que foram propostas no capítulo inicial, uma vez que a análise desse metamodelo permitiu a criação das duas extensões realizadas no contexto desta dissertação.

O capítulo está organizado da seguinte forma: na Seção 3.2 é feita uma contextualização sobre a Modernização Orientada a Arquitetura e o Metamodelo KDM; na Subseção 3.2.1 é descrito o pacote *kdm* da camada de infraestrutura do KDM; na Subseção 3.2.2 é descrito o pacote *code* da camada de elementos do programa; na Seção 3.3 é mostrada uma ferramenta de apoio ao KDM e na Seção 3.4 são discutidas as considerações finais do capítulo.

3.2 Modernização Orientada a Arquitetura e o Metamodelo KDM

Em 2003 a OMG iniciou esforços no sentido de padronizar o processo de modernização de sistemas legados utilizando modelos por meio da *Architecture-Driven Modernization Task Force* (ADMTF) (OMG, 2013).

O objetivo da ADM é revitalizar aplicações existentes, melhorando ou adicionando funcionalidades; reutilizar os padrões existentes de modelagem da OMG e da iniciativa MDA; e consolidar as melhores práticas que conduzem à modernização bem sucedida. Em outras palavras, a OMG por meio da ADMTF teve a iniciativa de padronizar os processos da reengenharia de software para aumentar o êxito nos projetos dessa natureza (OMG, 2013).

De acordo com a OMG (2013), a ADM não substitui a reengenharia, e sim a melhora, pois, ela resolve os problemas da reengenharia tradicional por meio da utilização dos processos de reengenharia juntamente com os princípios da orientação a modelo.

O fluxo de um processo de modernização apoiado pela ADM possui três fases e é semelhante ao contorno de uma ferradura, são elas: Engenharia reversa, reestruturação e engenharia avante, como pode se visto na Figura 3.1. 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 geração de um modelo PIM (*Platform Independent Model*), que serve como base para a 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 *software*, independentemente da plataforma utilizada anteriormente.

O modelo PSM é um modelo específico de uma plataforma, ou seja, nele existem metadados relacionados a uma plataforma ou linguagem de programação específica. O modelo PIM é representado pelo principal metamodelo da ADM, chamado de KDM (*Knowledge Discovery Metamodel*), que será explicado com mais detalhes posteriormente. O PIM possui um nível de abstração mais alto, pois não contém informação específica de uma plataforma. Por último, têm-se o modelo CIM, também representado por modelos KDM. CIM é o modelo de mais alto nível no

processo de modernização, representando o sistema em uma forma independente de computação.

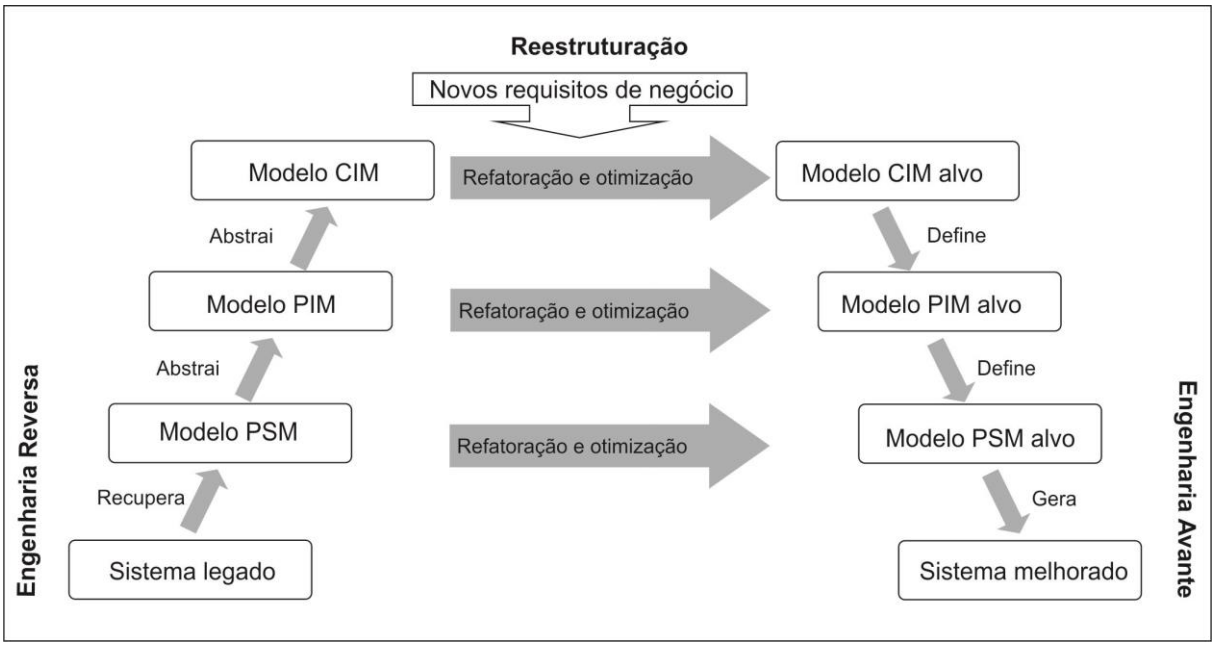


Figura 3.1 - Fluxo do processo de modernização apoiado pela ADM (Pérez-Castillo *et al*., 2011a) (adaptada)

Na fase de reestruturação, refatorações, melhorias e novas regras de negócios podem ser introduzidas no sistema, e, com uma representação independente de plataforma do *software* 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.

Segundo Pérez-Castillo *et al*. (2011a) 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 em que é interessante realizar um processo de modernização somente a partir de modelos PSM ou PIM, dependendo da necessidade.

Para dar suporte ao processo de modernização, foi criado em 2006 um metamodelo que possibilita a comunicação entre diferentes plataformas e linguagens, e foi chamado pela ADMTF de KDM. A ADMTF criada pela OMG criou o *Request-for-Proposal* (RFP) que por sua vez descrevia as principais características do que seria o KDM (OMG, 2013).

O KDM é um metamodelo de representação intermediária comum para sistemas existentes e seus ambientes operacionais. Utilizando essa representação

para sistemas existentes é possível trocar representações do sistema em modelo entre plataformas e linguagens com a finalidade de analisar, padronizar e transformar os sistemas existentes (OMG, 2013).

Fornecendo representações para os sistemas de softwares existentes, o KDM consiste em vários modelos arquiteturais do sistema que são definidos em diferentes camadas de abstrações. Cada modelo é representado por um conjunto de visões arquiteturais, ou seja, modelos KDM representando diferentes perspectivas de conhecimento sobre os artefatos dos sistemas de software existentes. Esses modelos são criados automaticamente, semi-automaticamente ou manualmente por meio da aplicação de várias técnicas de extrações de conhecimento, de análises e de transformações (Normantas *et al*., 2012).

De acordo com Ulrich e Newcomb (2010), o KDM permite a representação do sistema como ele está (*as-is*[2]) e sua arquitetura de dados a partir das informações que podem ser derivadas dos artefatos disponíveis do sistema legado. O desenvolvimento do KDM baseou-se em um número de requisitos chave e suas principais características são:

- Representa os principais artefatos do software usando entidades, relacionamentos e atributos;
- Mapeia artefatos externos com o qual o software interage;
- Possui um núcleo que gera uma representação do sistema independente de plataforma ou linguagem, e pode ser extensível para dar apoio a outras tecnologias;
- Define uma terminologia unificada para a descoberta de conhecimento dos artefatos de software existentes;
- Está representado usando diagramas de classe da UML;
- Utiliza um formato de intercâmbio XML3 Metadata Interchange (XMI) para desenvolvedores importarem e exportarem suas ferramentas específicas;
- Dá apoio à representação de todo tipo de plataforma e de linguagem e descreve tanto as estruturas físicas quanto as lógicas de arquiteturas de software;

[2] *As-is* é uma terminologia utilizada pela OMG para se referir ao sistema legado bruto, ou seja, como ele se encontra antes do processo de modernização.
[3] XML - eXtensible Markup Language, é uma linguagem de marcação recomendada pela W3C para a criação de documentos com dados organizados hierarquicamente, tais como textos, banco de dados ou desenhos vetoriais.

- Facilita o rastreamento de artefatos que se encontram em diferentes níveis de abstração, partindo de sua estrutura lógica até sua representação física.

O metamodelo KDM representa artefatos físicos e lógicos de software dos sistemas legados em diferentes níveis de abstração e constitui doze pacotes organizados em quatro camadas, são elas: infraestrutura (*infrastructure*), elementos do programa (*program elements*), recursos de tempo de execução (*runtime resources*) e abstração (*abstractions*). Na Figura 3.2 está representada a arquitetura do KDM ilustrando a forma como as camadas se relacionam, quais são os pacotes pertencentes a cada camada e a separação dos interesses em KDM. Cada camada baseia-se na camada anterior, dessa forma, elas estão organizadas em pacotes que definem um conjunto de elementos do metamodelo, cujo propósito é representar um interesse específico e independente do conhecimento relacionado a sistemas legados (Pérez-Castillo *et al*., 2011a).

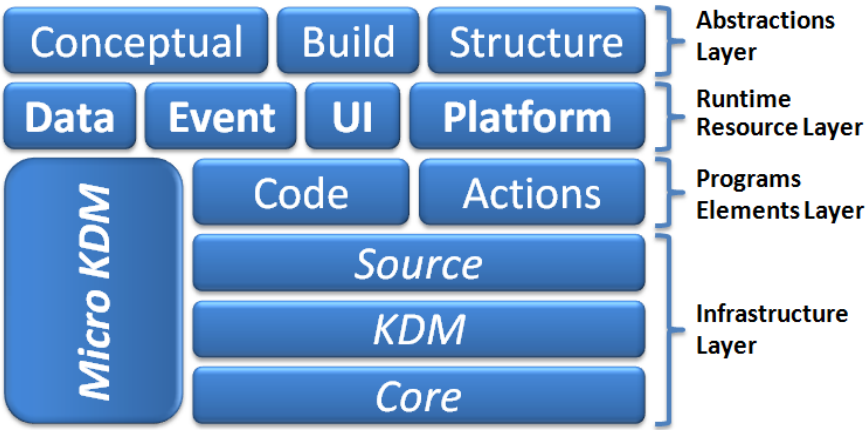


Figura 3.2 - Arquitetura do KDM (Pérez-Castillo et al., 2011a) (Adaptada)

A camada de infraestrutura, que está no nível mais baixo de abstração, define um conjunto de conceitos utilizados ao longo de toda especificação KDM, fornecendo um núcleo comum para todos os outros pacotes. A camada de elementos do programa oferece um amplo conjunto de elementos de metamodelo, a fim de fornecer uma representação intermediária independente de linguagem para várias construções definidas por linguagens de programação comuns. Esta camada representa elementos de programa em nível de implementação e suas associações, dessa forma, a camada de elementos do programa representa a visão lógica de um sistema legado (Pérez-Castillo *et al*., 2011a).

A camada de recursos de tempo de execução permite a representação do alto valor de conhecimento sobre sistemas legados e seu ambiente operacional, ou seja, foca no que não está contido no código-fonte. A camada de recursos de tempo de execução representa os recursos gerenciados pela plataforma de tempo de execução, fornecendo ações de recursos abstratos para gerenciar os demais recursos. Cada pacote desta camada define entidades e *containers* específicos para representar os recursos do sistema legado e também relacionamentos estruturais específicos entre os recursos (Pérez-Castillo *et al*., 2011a).

A camada de abstração define um conjunto de elementos capazes de representar abstrações de domínios e aplicações específicas, bem como artefatos relacionados ao processo de construção do sistema existente. Os elementos do metamodelo da camada de abstração fornecem vários *containers* e grupos para outros elementos de metamodelos (ISO/IEC, 2012).

Por último, o pacote *micro KDM* objetiva refinar a semântica definida na ontologia[4] do KDM. O pacote *micro KDM* é principalmente aplicado nos elementos de ação, encontrados no pacote *action*. Micro KDM é um conjunto de regras de conformidade, orientações adicionais para a construção e interpretação de visões de alta fidelidade KDM, adequados para a realização de análise estática (Pérez-Castillo *et al*., 2011a).

Como foi definido no capítulo 1 desta dissertação, o objetivo mais amplo é representar os conceitos da POA no metamodelo KDM. Baseando-se nos estudos realizados sobre este metamodelo durante o período do mestrado, percebeu-se que os mecanismos de extensão para representar o nível de detalhes do escopo deste trabalho estão localizados em camadas e pacotes específicos no KDM. Para desenvolver uma extensão pesada no KDM que represente os conceitos da POA é necessário estender algumas metaclasses do pacote *code*, localizado na camada de Elementos do Programa. Já no caso da extensão leve, o pacote a ser utilizado é o *kdm* da camada de Infraestrutura. Nas seções seguintes esses pacotes e camadas serão explorados com mais detalhes, no intuito de mostrar suas metaclasses e como eles podem ser utilizados.

[4] Ontologia - Nas Ciências e Tecnologias de Informação, as ontologias são classificações. São usadas como um meio para categorizar ou agrupar as informações em classes.

3.2.1 Pacote *kdm* da Camada de Infraestrutura

O pacote *kdm* define elementos que constituem a infraestrutura para outros pacotes e define os elementos que formam a estrutura de cada representação KDM, ou seja, esse pacote define a organização de todo o metamodelo KDM e especifica as metaclasses que pertencem a um pacote e quais pacotes pertencem a uma camada. Existem nove tipos de modelos definidos pelo pacote *kdm*, sendo, cada um deles, descrito por um ou mais pacotes do KDM. Um "modelo" KDM é uma unidade chave de uma instância do KDM, dessa forma, um "segmento" KDM pode ser formado por um ou mais modelos e representa a unidade mínima a ser criada. Por exemplo, um segmento KDM pode ser formado por um modelo *code*, um modelo *action* e outros mais, sua composição dependerá apenas da ferramenta que fará a recuperação das informações do sistema (Pérez-Castillo *et al*., 2011a).

O pacote *kdm* também define um conjunto de metaclasses que constituem a base para o mecanismo de extensão leve no KDM. Essas metaclasses permitem a representação de conceitos de domínios específicos em tempo de instanciação.

O mecanismo de extensão leve é uma forma padronizada de adicionar novos elementos em uma instância do KDM. O novo elemento criado por meio da extensão leve é chamado de estereótipo e é formado por um elemento existente no metamodelo original que possui o seu significado estendido e com atributos, também possivelmente estendidos. Este mecanismo é definido como parte do KDM para que se possa introduzir novas metaclasses estendidas. O exato significado e intenção da metaelemento estendido está fora do escopo do KDM e deve ser informado pelos implementadores para os usuários que forem utilizar as representações estendidas.

O mecanismo de extensão leve do KDM fornece os seguintes recursos:

1. Definir um estereótipo (*stereotype*):
    - A definição de um estereótipo inclui o nome da classe do elemento base permitido, podendo ser um elemento concreto, genérico ou abstrato.

2. Definir marcações (*tags*) associadas a estereótipos. Marcações são atributos adicionais aos elementos estendidos. As definições de marcações (*Tag definition*) incluem o nome do atributo estendido e o nome do tipo do elemento (representado por uma string). Valores de atributos estendidos podem ser strings ou referências a algum elemento modelado. Cada estereótipo define o seu próprio

conjunto de marcações, dessa forma, definições de marcações estão ligadas a seu estereótipo correspondente.

3. Organizar estereótipos em famílias de estereótipos. Famílias de estereótipos estão contidas dentro da estrutura de elementos de um modelo KDM, ou seja, estão dentro de modelos e segmentos KDM.

4. Usar elementos estendidos em instâncias do KDM por meio do uso de um ou mais estereótipos:

- Valores de marcações (*tag values*) concretas podem ser adicionadas em um elemento se o estereótipo definir essa marcação, ou seja, uma marcação só pode ser usada se o seu estereótipo estiver sendo usado.
- Cada valor de marcação está associado à sua definição de marcação.
- O tipo completo do novo elemento está definido como a união de todos os estereótipos adicionados no elemento, assim, o novo elemento possui características do seu elemento base mais os valores adquiridos com o uso do estereótipo.

Instâncias puras do KDM devem usar apenas metaclasses concretas que possuem sua semântica especificada no KDM, sem o uso de estereótipos. Instâncias do KDM que usam estereótipos são chamadas de instâncias estendidas do KDM. Instâncias estendidas do KDM podem adicionar estereótipos em instâncias de elementos do KDM. O mecanismo de extensão leve do KDM não permite multiplicidade de marcações, restrições em marcações, nem relacionamento entre marcações e estereótipos. Dessa forma, o responsável pela criação da extensão deverá escolher a metaclasse mais específica para definir seu estereótipo para que a semântica entre o elemento e estereótipo faça sentido.

O metamodelo KDM fornece um conjunto de metaclasses no pacote *KDM* que permitem a criação de famílias de estereótipos; essas metaclasses são mostradas na Figura 3.3. O elemento *ExtensionFamily* atua como um *container* para um conjunto de estereótipos relacionados e suas *TagDefinitions* correspondentes. O elemento *Stereotype* fornece uma maneira de marcar elementos de um modelo para que eles se comportem como se fossem instâncias de um novo modelo virtual.

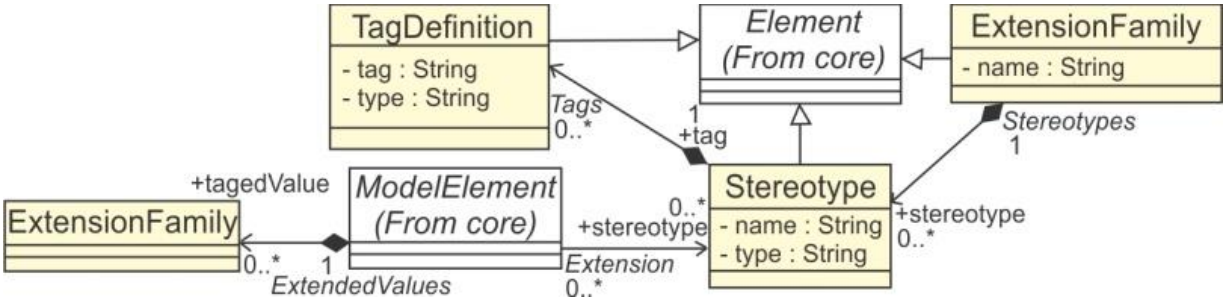


Figura 3.3 - Diagrama de classes de extensões no KDM (ISO/IEC, 2012) (Adaptado)

Como já mencionado, cada *Stereotype* pode possuir um conjunto de *TagDefinitions* e cada um fornece o nome de uma marcação e o nome de um tipo em KDM do valor correspondente. Os elementos *ExtensionFamily*, *Stereotype* e *TagDefinition* são os elementos principais para representar extensões leves no KDM.

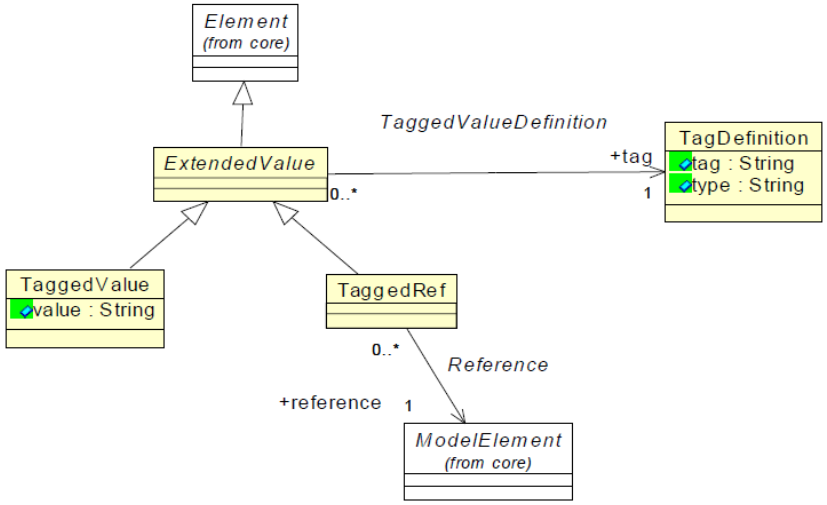


Figura 3.4 - Diagrama de classes de valores estendidos (ISO/IEC, 2012).

Outro diagrama importante a ser mostrado que faz parte do mecanismo de extensão leve do KDM é o diagrama de classes de valores estendidos (*ExtendedValues*) (Figura 3.4). Esse diagrama define construções adicionais no metamodelo que representam valores estendidos que podem ser adicionados aos elementos estendidos do modelo KDM. A chave do mecanismo de extensão leve é o estereótipo. Os estereótipos permitem especificar um significado adicional em uma instância do metamodelo. Enquanto o significado de um estereótipo não é definido dentro do KDM, estereótipos permitem a diferenciação de elementos dentro de instâncias do KDM e permitem adicionar atributos a esses elementos estendidos.

Na Figura 3.4 o elemento representado pela classe *ExtendedValue* é abstrato e é superclasse de outras duas classes concretas, que são: *TaggedValue* e *TaggedRef*. A classe *ExtendedValue* define propriedades em comum para essas

duas classes e representa o valor de um atributo. A definição de um atributo é fornecido por uma *TagDefinition* que faz parte de um estereótipo. Como visto, um estereótipo permite a criação de novos elementos e permite também a inserção de novos atributos. Esses novos atributos são instanciados todas as vezes que um novo elemento estendido for criado. Cada instância do elemento *ExtendedValue* está associada a uma *TagDefinition* correspondente.

Os elementos *TaggedValue* e *TaggedRef* permitem que informações sejam inseridas em um modelo KDM.

Uma *TaggedValue* insere informação em um elemento do KDM que possui um estereótipo vinculado. Para fornecer um sentido completo a esse elemento é necessário preencher as propriedades *Tag* e *Value*. O valor de *Tag* é uma lista das *TagDefinitions* que o estereótipo possui e o valor de *Value* é uma string que representa um valor do tipo que foi definido na *TagDefinition*. Por exemplo, se for criada a *TagDefinition* chamada *IsAbstract* e seu tipo for *boolean*, na *TaggedValue* a propriedade *Tag* será preenchida com *IsAbstract* e a propriedade *Value* será preenchida com "*true*" ou "*false*". Lembrando que todos os valores passados na extensão leve são strings, dessa forma, não existe checagem de tipo.

Já o elemento *TaggedRef* insere informação no formato de uma referência a outro elemento existente no modelo. Da mesma forma que uma TaggedValue, uma *TaggedRef* deve estar em conformidade a uma *TagDefinition* correspondente e representa atributos complexos, que são associações para outros elementos do KDM. A *TaggedRef* pode ser o nome de qualquer elemento do metamodelo KDM, por exemplo, "*AbstractCodeElement*," "*ControlElement*," ou "*ClassUnit*". As propriedades de uma TaggedRef são *Tag* e *Reference*. O valor de *Tag* é uma lista das *TagDefinition* que estão ligadas ao estereótipo que foi aplicado e o valor de *Reference* é uma lista de todos os elementos contidos no modelo em que deverá ser escolhido apenas um para que a referência seja feita.

A interpretação da semântica dos elementos *TaggedValue* e *TaggedRef* está fora do escopo do KDM, assim, ela deve ser determinada pelo usuário ou ferramenta que suporta a extensão.

3.2.2 Pacote *code* da Camada de Elementos do Programa

O pacote *code* define um conjunto de elementos cujo propósito é representar elementos do programa em nível de implementação e suas associações. Ele é determinado por uma ou mais linguagens de programação

Esse pacote possui metaclasses que representam elementos de várias linguagens de programação tais como programação orientada a objetos, procedural e outras. Com uma regra geral, em uma instância do KDM, cada instância de um elemento do pacote *code* representa alguma construção em determinada linguagem de programação.

O pacote *code* possui 90 metaelementos que estão organizados em 24 diagramas de classes, que são: *CodeModel*, *CodeInheritances*, *Modules*, *ControlElements*, *DataElements*, *Values*, *PrimitiveTypes*, *EnumeratedTypes*, *CompositeTypes*, *DerivedTypes*, *Signature*, *DefinedTypes*, *ClassTypes*, *Templates*, *TemplateRelations*, *ClassRelations*, *TypeRelations*, *InterfaceRelations*, *PreprocessorDirectives*, *PreprocessorRelations*, *Comment*, *Visibility*, *VisibilityRelations* e *ExtendedCodeElements*.

Para atender a um dos objetivos desta dissertação, uma extensão pesada do metamodelo KDM teve que ser realizada no pacote *code*, pois, os elementos da POA que se deseja representar estão em nível de implementação. Para identificar quais dos elementos deste pacote deveriam ser estendidos foi escolhido um perfil na UML para AspectJ e, em seguida, foi feito um mapeamento entre os elementos desse perfil e os elementos presentes no pacote *code*. Mais detalhes sobre esses assuntos são discutidos no Capítulo 5.

Visto que esse pacote é bastante amplo e possui muitas metaclasses, nos parágrafos seguintes serão discutidas apenas as metaclasses que foram utilizadas nesta pesquisa, que são: *Datatype*, *ClassUnit, MemberUnit*, *MethodUnit*, *StorableUnit*, *ControlElement* e *Package*.

A metaclasse *Datatype* está contida no diagrama de classes *CodeModel*. Esse diagrama é um modelo KDM específico que possui coleções de informações sobre um sistema existente de tal forma que essas informações representam o domínio do código-fonte de um sistema. O *CodeModel* é o único modelo da camada de elementos do programa do KDM, sendo assim, os demais diagramas do pacote *code* contem metaclasses mais específicas de cada metaclasse do *CodeModel*. Em

outras palavras, o *CodeModel* é uma abstração de todas as metaclasses do pacote *code*, em que apenas as principais metaclasses são representadas.

Na Figura 3.5 é possível ver as metaclasses do *CodeModel*. Esse diagrama define classes determinadas pelo modelo padrão do KDM, que são CodeModel, *AbstractCodeElement* e *AbstractCodeRelationship*. *CodeModel* representa um modelo para os elementos do pacote *code*, *AbstractCodeElement* representa uma classe abstrata controladora para todas as entidades KDM que podem ser usadas no modelo code e AbstractCodeRelationship é a classe que representa uma classe abstrata controladora de todos os relacionamentos do KDM no pacote *code*. O diagrama *CodeModel* também define várias classes abstratas chave, que determinam a taxonomia do KDM para elementos do programa, tais como *CodeItem*, *ComputationalObject*, *Datatype* e *Module*.

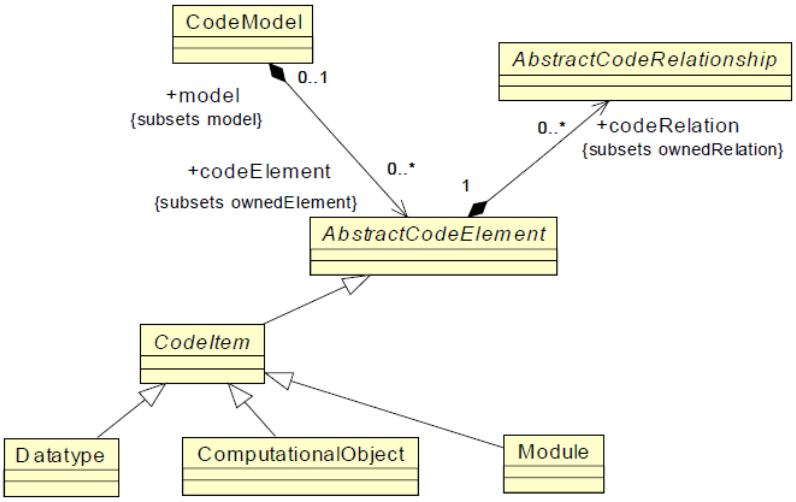


Figura 3.5 - Diagrama de classes *CodeModel* (ISO/IEC, 2012).

A metaclasse *Datatype* representa elementos que podem ser nomeados e são determinados pela linguagem de programação que descreve tipos de dados. Datatype é um elemento genérico com semântica especificada previamente que pode ser usado como um ponto de extensão para definir novos elementos que representam tipos de dados nomeados que não se encaixam em categorias semânticas de subclasses concretas do elemento *Datatype*.

A metaclasse *ClassUnit* faz parte do diagrama de classes *ClassTypes*. Esse diagrama define elementos que representam tipos de dados compostos comumente fornecidos por várias linguagens de programação. Essa metaclasse representa as classes de linguagens orientadas a objetos. Uma classe é um elemento do tipo *Datatype* que representa uma classe, ou seja, uma coleção ordenada de elementos

nomeados, que por sua vez pode ser outro elemento do tipo CodeItem, tal como um *MemberUnit* ou um *MethodUnit*. Em outras palavras, uma instância do elemento *ClassUnit* pode conter outros elementos do mesmo tipo ou outros elementos que representam construções em uma linguagem OO, como métodos e variáveis.

A metaclasse *MemberUnit* é uma subclasse concreta do elemento *DataElement* que representa um membro de um tipo de classe. Instâncias da classe *MemberUnit* são pontos de extremidade das relações de dados do KDM que descrevem o acesso a classes. *MemberUnit* é similar a um *ItemUnit*, sua única diferença é que um *ItemUnit* geralmente representa uma parte de certo objeto computacional, enquanto o objeto computacional correspondente a um *MemberUnit* é geralmente determinado por uma instância de uma classe.

A metaclasse *MethodUnit* representa funções de um membro dentro de uma instância do elemento *ClassUnit*, incluindo operadores definidos pelo usuário, construtores e destrutores. Uma instância do elemento *MethodUnit* é capaz de representar um método na programação orientada a objetos.

Já o elemento *StorableUnit* representa uma variável do sistema, ou seja, um objeto computacional em que os diferentes valores do mesmo tipo de dados pode ser associado mais de uma vez. *StorableUnit* pode representar tanto uma variável global quanto uma variável local.

ControlElement é um elemento genérico que pode ser usado para representar estruturas de controle. Esse elemento representa itens que podem ser nomeados do sistema que descrevem um comportamento que pode ser feito por demanda e por meio de mecanismos de invocação, tais como mecanismo de chamada de retorno.

A metaclasse *Package* é um subtipo da metaclasse *Module*. A metaclasse *Module* é um elemento de modelagem genérico do KDM que representa todo um módulo de software ou um componente. Um *Package* é um *conteiner* lógico para elementos do programa, tais como classes e interfaces, na Programação Orientada a Objetos (POO).

3.3 Ferramenta de apoio ao KDM

Um dos trabalhos mais importantes publicados no contexto da ADM é o de Bruneliere *et al*. (2010) que propõem uma ferramenta chamada de MoDisco. MoDisco é um framework genérico e extensível para a abordagem de Engenharia Reversa dirigida a modelos e foi implementado no Eclipse. Basicamente essa ferramenta é capaz de recuperar código-fonte legado, base de dados e outros artefatos legados e representá-los com o metamodelo KDM. Com o auxílio dessa ferramenta também é possível aplicar refatorações nos modelos recuperados e transformá-los em modelos modernizados.

Contudo, uma das limitações dessa ferramenta é a inserção de domínios específicos, tais como a POA. No contexto deste trabalho foi utilizado a ferramenta MoDisco para recuperar as informações do código-fonte legado escrito em Java, maiores detalhes sobre a utilização do MoDisco são descritas nos capítulos que seguem. Sem o auxílio dessa ferramenta toda a parte que utiliza Java deveria ser transformada em modelo manualmente, o que poderia atrasar esta pesquisa, uma vez que toda manipulação do KDM foi possível por causa da existência do MoDisco e do seu suporte em Java para manipular o kdm.

3.4 Considerações Finais

O foco deste capítulo direciona-se à análise do panorama atual da literatura que trata sobre a modernização de sistemas legados levando em consideração a padronização proposta pela OMG. Foram mostrados os principais conceitos da ADM, KDM e seus pacotes e camadas, necessários ao entendimento desta dissertação e fundamentais ao desenvolvimento da proposta aqui desenvolvida.

Observou-se também que o metamodelo KDM, por intermédio de suas camadas e pacotes, permite a criação de modelos que representem um sistema legado em diversas visões. O objetivo da OMG ao criar este metamodelo propõe uma padronização da reengenharia de software, fornecendo modelos que ajudem no processo de reengenharia de sistemas.

Neste capítulo também foi comentado sobre a ferramenta MoDisco, uma das principais ferramentas que trabalha com ADM e KDM. Essa ferramenta foi utilizada no contexto deste trabalho para dar suporte à recuperação de modelo KDM a partir de código-fonte legado escrito em Java.

Mais adiante, no Capítulo 4, são apresentados os mecanismos de extensão leve e pesada no metamodelo KDM. Caracterizando os principais artefatos necessários para a realização de ambas as extensões e mostrando como realizá-las.

Capítulo 4

EXTENSÕES DO METAMODELO KDM PARA POA

4.1 Considerações iniciais

O propósito deste capítulo é apresentar as duas extensões realizadas no KDM (pesada e leve) e como usá-las. Foram vistos nos capítulos anteriores os conceitos da programação orientada a aspectos e do metamodelo KDM. Uma parte de suma importância do Capítulo 3 é a seção que trata sobre a modernização orientada a aspectos. Essa abordagem leva em consideração a utilização dos conceitos da POA para modernizar sistemas legados, contudo, para poder utilizar esses conceitos é preciso representar os conceitos da POA no metamodelo KDM. Este capítulo foi desenvolvido com base nos conceitos apresentados nos capítulos 2 e 3, uma vez que é nele que são reportadas as extensões no KDM (leve e pesada) para a POA.

O capítulo está organizado da seguinte forma: na Seção 4.2 são apresentadas as extensões orientadas a aspectos no metamodelo KDM; na Seção 4.3 é descrita extensão orientada a aspectos pesada do KDM; na Subseção 4.3.1 são mostrados detalhes de implementação da extensão pesada; na Seção 4.4 é descrita a extensão orientada a aspectos leve do KDM; na Subseção 4.4.1 é mostrado como é feita a extensão leve graficamente do KDM e na Seção 4.5 são discutidas as considerações finais do capítulo.

4.2 Extensões Orientadas a Aspectos do Metamodelo KDM

A primeira atividade antes de iniciar a criação de uma extensão para o metamodelo KDM (ou de qualquer outro metamodelo) é definir os elementos que deverão ser representados e as relações entre eles. Está fora do escopo desta dissertação se aprofundar em todas as metaclasses do metamodelo KDM, dessa forma, os elementos analisados se restringiram aos elementos capazes de representar código-fonte nos paradigmas OO e OA. Nesse sentido, foi realizada uma revisão da literatura para identificar metamodelos OA e perfis UML que pudessem ser considerados bons candidatos. O trabalho apresentado por Evermann (2007) foi o escolhido como base para nossas extensões porque ele apresentava a quantidade de informações necessárias para incluir todos os elementos da POA que podem ser representados pela linguagem AspectJ. Os elementos que se buscava representar eram Aspectos, Adendos, Declarações Intertipo, Diferentes tipos de Pontos e Conjuntos de junção. Outro ponto importante para a escolha do perfil do Evermann é que se trata de um perfil bastante aceito e utilizado no meio acadêmico. Nos próximos parágrafos encontram-se descrições curtas dos outros trabalhos investigados e das razões pelas quais eles não foram considerados.

O trabalho de Grundy e Patel (2001) é baseado em definir novas metaclasses na UML, ao invés de definir estereótipos para classes existentes. Nesse trabalho os autores estenderam a ferramenta chamada *JComposer* para adicionar suas novas metaclasses, contudo, a abordagem é focada em sistemas baseados em componentes, o tornando bastante específico. O grande problema dessa abordagem é a necessidade de constantes manutenções na ferramenta todas as vezes que novas modificações forem feitas.

O trabalho de Kandé *et al*. (2002) foi um dos primeiros nessa área, porém, a notação proposta pelos autores foi baseada em versões anteriores da UML 2.0 e uma vez que nessa versão a UML não possui metaclasses suficientes para representar a ideia de *weaving* da POA, os interesses transversais ficavam espalhados pelo modelo.

O problema encontrado no trabalho de Pawlak *et al*. (2002) é que esse perfil utiliza palavras-chave e definição de regras para especificar Adendos e *PointCuts*, o que o torna um perfil passível de erros. Outro problema é que esse trabalho não

cobre todos os conceitos da POA, dessa forma, muitas representações seriam perdidas.

A proposta apresentada em Basch e Sanchez (2003), assim como Pawlak *et al*. (2002), usa especificação textual de *joinpoints*, quando deveria ser baseado nos metaelementos do modelo. É proposta também a inclusão de dois elementos no metamodelo da UML, um para representar *joinpoints* e outro para representar os aspectos em si para melhorar a interação entre os aspectos e os outros componentes do sistema. Contudo, este trabalho apresenta uma abordagem mais conceitual e não abrange os principais conceitos da orientação a aspectos.

Uma proposta inicial para modelar aspectos usando UML 2.0 é apresentada por Barra *et al*. (2004), contudo, a extensão do perfil não é completa, uma vez que os autores afirmam que, na época, não existiam muitos padrões para se desenvolver softwares orientados a aspectos. A falta desses padrões prejudicava especialmente a etapa de modelagem do sistema, dessa forma, eles afirmam que uma vez essa padronização seja feita, a representação da POA na UML seria mais facilmente realizada.

Na extensão proposta por Fuentes e Sánchez (2006), os aspectos são especificações textuais aplicadas em classes estereotipadas. No trabalho são descritos os passos de como modelar aspectos usando sua abordagem, mas ele descreve como representar basicamente *pointcuts*, declarações inter-tipo e os aspectos. A representação de *pointcuts* nessa abordagem se dá por meio de um estereótipo chamado <<*hook*>> que é aplicado em instâncias de metaclasses comportamentais, assim elas podem ser interceptadas por aspectos. A abordagem apresentada atribui muitas restrições para utilizar o mesmo estereótipo em diferentes situações, tornando seu reúso complexo e inviável, uma vez que ele não possui um nível de detalhes suficientes para representar um código-fonte em uma linguagem atual.

O trabalho de Evermann (2007) foi o trabalho mais completo encontrado na literatura. Nele é possível encontrar todos os principais conceitos da POA (Aspecto, Adendo, Ponto de Junção e Declarações Inter-tipo) com um nível de detalhes o suficiente para representar um código-fonte em AspectJ.

O trabalho de Amroune *et al*. (2012) apresenta um perfil UML para modelar aspectos em Sistemas de Informação Cooperativo (*An Aspect approach to develop Cooperative Information Systems* - AspeCiS). Nesse trabalho é mostrado um perfil

capaz de representar apenas aspectos, pontos de junção e adendos. Além de não representar declarações inter-tipos, não fica claro como é a declaração de um tipo de ponto de junção ou quais são os tipos possíveis de serem aplicados, uma vez que essa informação não está presente em seu perfil.

Como já comentado, mediante a análise destes trabalhos, o perfil do Evermann foi considerado o mais adequado, pois, ele engloba o nível de detalhes que atende aos objetivos propostos nesta pesquisa (Evermann, 2007). Dessa forma, ambas extensões desenvolvidas no contexto deste projeto foram criadas com base nesse perfil.

Na Figura 4.1 estão sendo mostradas ambas as extensões. Essa figura foi adaptada do artigo de Evermann (2007), que é usada para representar as metaclasses que constituem o seu perfil UML para AspectJ. Como ambas as extensões realizadas neste trabalho tem como base as metaclasses e relacionamentos criados por Evermann, optou-se por tomar este perfil como base para a construção da abordagem aqui proposta.

Cada classe/elemento possui quatro linhas no primeiro compartimento. A primeira palavra representa o nome das metaclasses que foram criadas na extensão pesada. Por exemplo, *AspectUnit* é uma metaclasse que existe na extensão pesada. A segunda palavra, dentro dos colchetes, é a metaclasse do KDM que foi escolhida para que o atual elemento a estenda, servindo como uma metaclasse base para a criação de uma nova metaclasse. Por exemplo, a nova metaclasse *AspectUnit* que pertence à extensão pesada, estende a metaclasse *ClassUnit* do KDM (Santos *et al*., 2014a).

Na extensão leve não são criadas novas metaclasses, mas sim estereótipos. Assim, logo abaixo dos primeiros dois elementos já descritos existe um <<stereotype>>. Portanto, todos os estereótipos mostrados nessa figura existem na extensão leve aqui criada. Estereótipos são aplicados em elementos existentes, então, o nome do elemento em que o estereótipo pode ser aplicado é a metaclasse mostrada na segunda linha. Por exemplo, o estereótipo <<aspect>> pode ser aplicado apenas em instâncias de *ClassUnit*. A última linha contém o nome da metaclasse UML usada na versão original do perfil do Evermann. Vale lembrar que os estereótipos são aplicados em instâncias de metaclasses do KDM, dessa forma, o primeiro par de colchetes ([ ]) abaixo do nome de cada elemento mostra o nome da metaclasse KDM em que esse estereótipo pode ser aplicado.

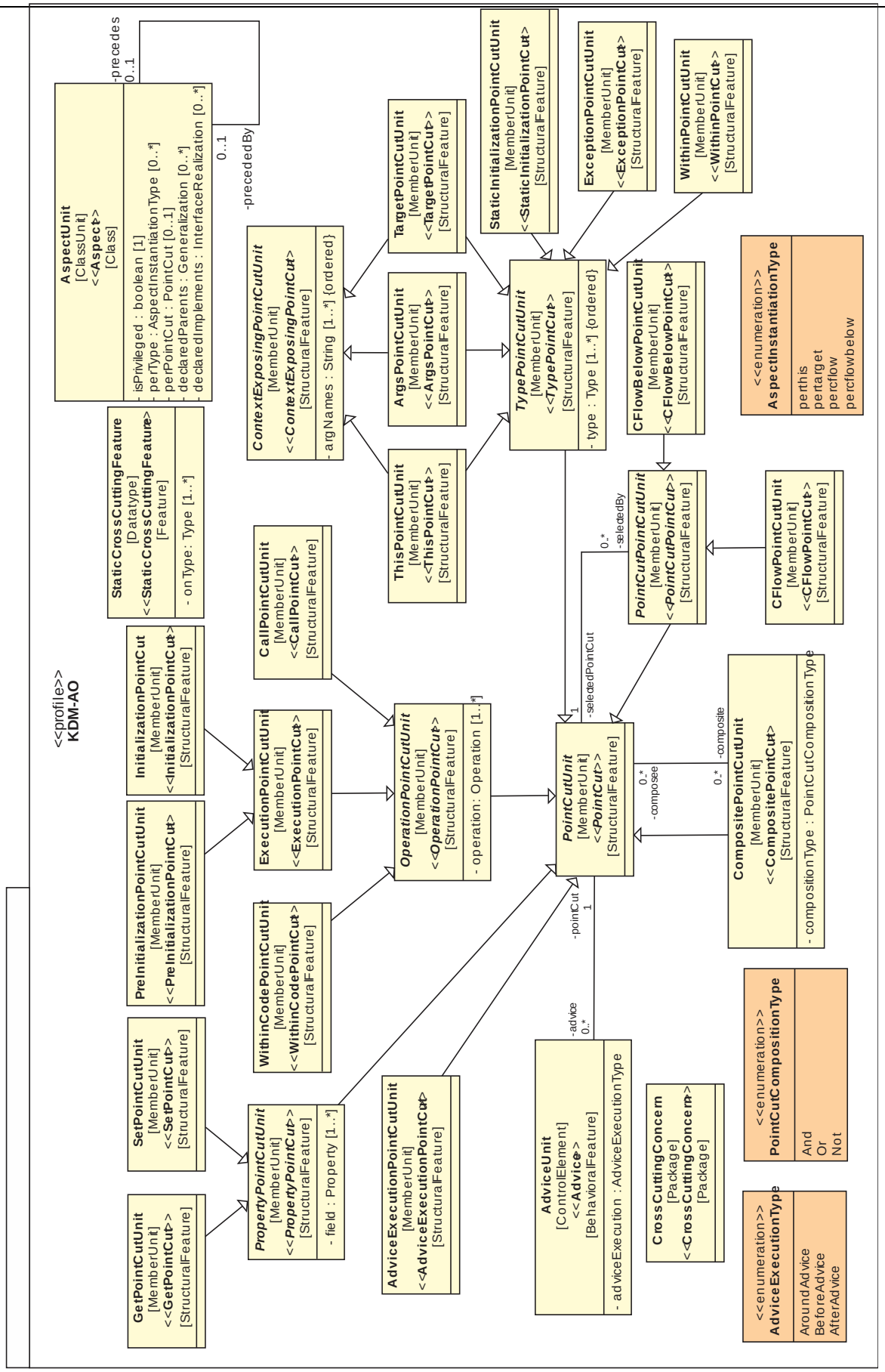


Figura 4.1 - KDM-AO e Perfil do Evermann (2007) (Adaptado).

Um dos maiores desafios quando necessita-se estender metamodelos é saber se a metaclasse escolhida como base para a criação de um novo elemento é a mais apropriada. Como os elementos do perfil do Evermann já foram previamente mapeados para as metaclasses da UML (estendendo eles por meio de estereótipos), a principal tarefa foi identificar as metaclasses do KDM que tivessem características similares às metaclasses da UML usadas por Evermann. Dessa forma, foi necessário fazer um mapeamento entre ambos os metamodelos (UML e KDM), que pode ser visto na Tabela 4.1.

Essa tabela de mapeamento identifica metaclasses do KDM que possuem características similares às metaclasses da UML. Algumas metaclasses podem ser diretamente mapeadas, tal como *Class* da UML, que pode ser facilmente mapeada para a metaclasse *ClassUnit* do KDM. Ambas possuem o mesmo objetivo e características; representando classes em um contexto orientado a objetos. Entretanto, como o KDM consegue representar níveis mais altos e mais baixos de abstração do que a UML, algumas metaclasses da UML não possuem apenas um único candidato equivalente no metamodelo KDM e outras não possuem nenhum equivalente. A metaclasse Property da UML, por exemplo, possui três metaclasses possíveis no KDM: *StorableUnit*, *ItemUnit* or *MemberUnit*. *StorableUnit* representa variáveis de tipo primitivo, *ItemUnit* representa registros e *MemberUnit* representa associações com outras classes. Essa lacuna ocorre porque o pacote *Code* do KDM está em um nível de abstração mais baixo do que a UML. Também existem metaclasses no KDM que não possuem metaclasse correspondente na UML, em consequência do baixo nível de abstração. Por exemplo, a metaclasse *CodeAssembly* que é uma metaclasse que representa um *conteiner* lógico para elementos de um programa na linguagem de máquina que foram construídos em um sistema operacional ou hardware específico.

Tabela 4.1 - 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 *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; público, 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 *containers* 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 4.1 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. Para atender aos objetivos deste trabalho essa tabela de mapeamento mostra apenas os principais elementos que foram usados nas extensões KDM Orientado a Aspectos (KDM *Aspect Oriented* ou KDM-AO), uma vez que seria inviável mapear todas as noventa metaclasses do pacote *Code*. Contudo, todas as classes do perfil do Evermann foram mapeadas e estão presentes na Tabela 4.1.

4.3 Extensão Orientada a Aspectos Pesada do KDM

Baseado no mapeamento realizado, o desenvolvimento do KDM-AO na versão pesada 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 (2007), 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, na extensão pesada deverá estender a metaclasse *ClassUnit*.

Como pode ser visto na Figura 4.1, 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 pesada. Como já apresentado anteriormente, a extensão pesada 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 (Extensão pesada) e os elementos do perfil do Evermann.

No perfíl do Everman, o element *CrossCuttingConcern* estende a metaclasse *Package* da UML e tem como objetivo representar a existência de um interesse transversal como persistência, segurança e concorrência. No KDM-AO esse elemento estendo a metaclasse *Package*. Essa metaclasse KDM representa um pacote no qual é possível inserir aspectos, classes e outros elementos da programação OA e OO.

AspectUnit é o elemento da extensão para a representação de aspectos, que foi estendido da metaclasse *ClassUnit* (Figura 4.2). 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, pois podem ser vistos na Figura 4.1.

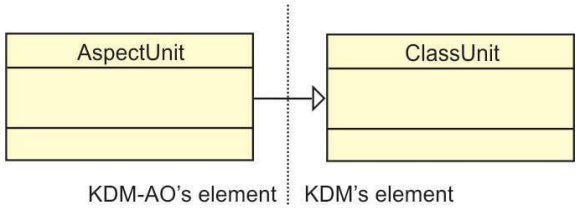


Figura 4.2 - AspectUnit

O elemento para representar adendos é *AdviceUnit* (Figura 4.3), que estende a metaclasse *ControlElement*. Sabendo que adendo é um elemento que especifica comportamento, é possível considerá-lo como um método. Entretanto, adendos não possuem nem especificadores de acesso (público, privado e protegido) nem tipos (construtor, destrutor, etc.). Por esses motivos foi decidido não fazer a metaclasse *AdviceUnit* estender o comportamento de *MethodUnit*.

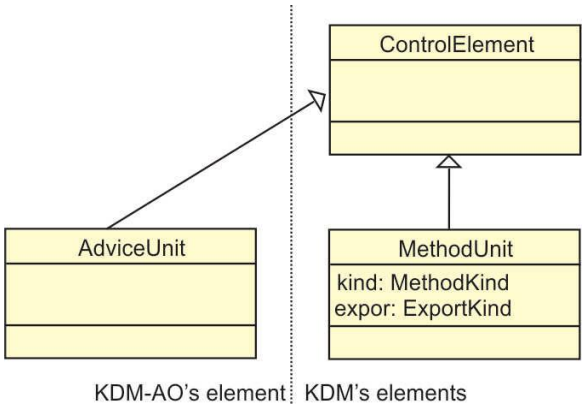


Figura 4.3 – AdviceUnit.

PointCutUnit é o elemento para representar pontos de junção e conjuntos de junção. De acordo com o perfil do Evermann, *PointCut* é um elemento estrutural e estende a metaclasse *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 fazer referências a membros de outras classes dentro de uma determinada classe. As relações entre essas metaclasses podem ser vistas na Figura 4.4.

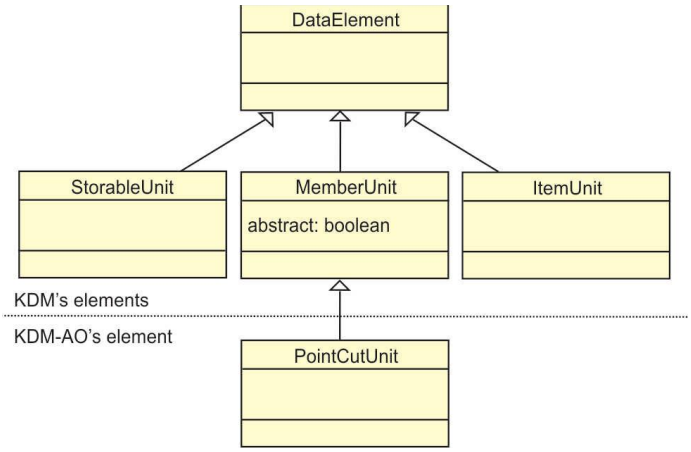


Figura 4.4 – PointCutUnit

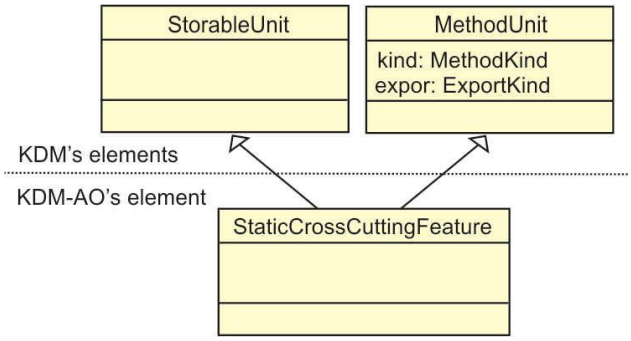


Figura 4.5 - StaticCrossCuttingFeature.

StaticCrosscuttingFeature é o elemento para representar declarações inter-tipo. Na extensão pesada 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 em uma determinada classe (ver Figura 4.5).

O elemento *CrossCuttingConcern* foi estendido do elemento *Package* do pacote *code* do KDM. Contudo, nenhum atributo ou relacionamento adicional foi inserido no novo elemento, uma vez que sua criação objetiva apenas separar os interesses em um modelo KDM, sem adicionar comportamento.

4.3.1 Detalhes de implementação

Para a criação do KDM-AO versão pesada, foram utilizados o Eclipse IDE (*Integrated Development Environment*) e o Framework Eclipse Modeling Framework (EMF), que permitiu a visualização e edição do metamodelo KDM original, em formato .ECORE, disponibilizado pela 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 4.1, 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 meio de elementos *EAttribute* e os relacionamentos entre as classes do perfil são especificadas por elementos *EReference*. A Figura 4.6 mostra a metaclasse *AspectUnit* representada no metamodelo KDM. É possível ver na parte A da Figura 4.6 os atributos da metaclasse (*isPrivileged*, *perType*, *perPointCut*, *declaredParents* e *declaredImplements*) e os relacionamentos (*precedes* e *precededBy*). Na parte B da Figura 4.6 é mostrada a metaclasse já introduzida no metamodelo KDM, assim como todos os seus atributos.

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 4.7 são mostradas as propriedades pertencentes à metaclasse *AspectUnit*. Após a criação de todas as novas metaclasses no metamodelo KDM, o *plug-in* KDM-AO

versão pesada foi gerado, permitindo a criação de instâncias orientadas a aspectos do metamodelo KDM.

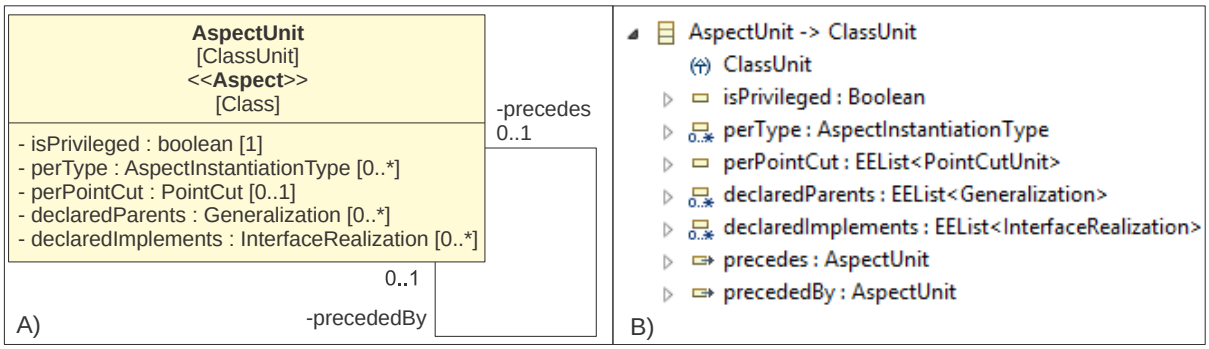


Figura 4.6 - KDM-AO (Extensão pesada) no EMF.

Property	Value
Abstract	false
Default Value	
ESuper Types	ClassUnit -> Datatype
Instance Type Name	
Interface	false
Name	AspectUnit

Figura 4.7 - Propriedade do elemento AspectUnit.

Na Figura 4.8 está representado um exemplo de instanciação do mecanismo de extensão pesada KDM-AO. Na Linha 1 é criada uma instância da metaclasse *AspectUnit* , na linha 2 é passada a informação do nome da instância e na linha 3 é informada a propriedade *IsPrivileged*, de acordo com o que foi explicado na Figura 4.6.

```
1..AspectUnit myAspect = CodeFactory.eINSTANCE.createAspectUnit();
2..myAspect.setName("connectionComposition");
3..myAspect.setIsPrivileged(true);
```

Figura 4.8 - Exemplo de instância da extensão pesada programaticamente.

No quesito reusabilidade o *plug-in* KDM-AO (Pesada) só pode ser reusado na ferramenta Eclipse, uma vez que ele foi desenvolvido nessa IDE. Contudo, as metaclasses, propriedade e relacionamentos referentes a POA que foram inseridos no metamodelo KDM original, formando assim o KDM-AO pesada, pode ser reutilizado em qualquer ferramenta que consiga interpretar o código-fonte em XMI das extensões .ECORE. Fazendo a leitura dessas informações no XMI das novas metaclasses e criando uma ferramenta/*plug-in* com essas novas informações, acredita-se que seria possível criar instâncias do KDM-AO independente da linguagem de programação (C++, AspectS, etc.), já que as metaclasses criadas são

extensões de classes existentes, que por sua vez são independentes de plataforma e linguagem.

4.4 Extensão Orientada a Aspectos Leve do KDM

Como visto no Capítulo 3, o metamodelo KDM fornece um conjunto de metaclasses no pacote *KDM* que permitem a criação de famílias de estereótipos. Os elementos *ExtensionFamily*, *Stereotype* e *TagDefinition* são os elementos principais para representar extensões leves no KDM. Vale ressaltar que neste trabalho a extensão leve foi feita programaticamente. Assim, para realizar essa extensão o usuário tem que criar uma nova classe Java e criar instâncias dos elementos *ExtensionFamily*, *Stereotype* e *TagDefinition* com os valores que representam os novos elementos.

Na Figura 4.9 é representado um trecho da extensão leve programaticamente em KDM. Na Figura 4.9 é possível ver a criação de três instâncias do pacote *kdm*, *ExtensionFamily*, *Stereotype* e *TagDefinition*. Na linha 1 é mostrada a criação de uma instância do elemento *ExtensionFamily* nomeada de *AspectConcepts*. Esse elemento serve para encapsular todos os estereótipos criados para o perfil KDM-AO leve. Na segunda linha é mostrada uma instância do elemento *Stereotype* e nela é possível ver a criação do estereótipo *AspectUnit*. Uma vez criado um estereótipo é necessário especificar a ExtensionFamily que ele pertence. O trecho de código-fonte presente na linha 3 adiciona o estereótipo criado na linha 2 dentro do elemento *ExtensionFamily* criado na linha 1.

```
1..ExtensionFamily AspectConcepts = KdmFactory.eINSTANCE.createExtensionFamily();
2..Stereotype AspectUnit = KdmFactory.eINSTANCE.createStereotype();
3..AspectConcepts.getStereotype().add(AspectUnit);
4..AspectUnit.setName("AspectUnit");
5..AspectUnit.setType("ClasUnit");
6..TagDefinition IsPrivileged = KdmFactory.eINSTANCE.createTagDefinition();
7..AspectUnit.getTag().add(IsPrivileged);
8..IsPrivileged.setTag("isPrivileged");
9..IsPrivileged.setType("boolean");
[...]
```

Figura 4.9 - Trecho da extensão leve criada programaticamente.

Nas linhas 4 e 5 são informados os valores *Name* e *Type* do estereótipo que são do tipo String. Vale lembrar que nessa linha o valor que *setName* recebe é *AspectUnit* e representa o nome do estereótipo, diferentemente do que ocorre na linha 2 que o nome *AspectUnit* representa a instância do elemento *Stereotype* propriamente dito.

Na linha 6 a *TagDefinition IsPrivileged* é criada e na linha 7 essa tag é vinculada ao *Stereotype AspectUnit*. E nas linhas 8 e 9 as propriedades *Tag* e *Type* do elemento *TagDefinition* são definidas. Mais uma vez, os valores passados como valores desse elemento são Strings, como é definido pelas regras do metamodelo KDM.

Para que a extensão fosse concluída, todos os estereótipos, relacionamentos e atributos mostrados na Figura 4.1 tiveram que ser adicionados programaticamente e devidamente vinculados, ou seja, os estereótipos vinculados a um *ExtensionFamily* e os relacionamentos e atributos criados por meio de *TagDefinitions* foram vinculados a seus respectivos estereótipos. Uma vez que todos os elementos foram criados programaticamente foi possível reusá-lo por meio de uma classe com todos os *Stereotypes* e *TagDefinitions* programados.

```
1..ClassUnit MyAspect = CodeFactory.eINSTANCE.createClassUnit();
2..MyAspect.setName("connectionComposition");
3..MyAspect.getStereotype().add(Profile.aspectUnit);

4..TaggedValue myIsPrivileged = KdmFactory.eINSTANCE.createTaggedValue();
5..myIsProvileged.setTag(Profiles.isPrivileged);
6..myIsProvileged.setValue("True");

7..MyAspect.getTaggedValue().add(myIsPrivileged);
```

Figura 4.10 - Exemplo de instância da extensão leve programaticamente.

Na Figura 4.10 é mostrado um exemplo de instanciação da extensão leve programaticamente. O código-fonte representa a criação de um aspecto com o preenchimento de duas propriedades, *name* e *isPrivileged*, assim como foi mostrado na Figura 4.8. Na linha 1 é feita a criação de uma *ClassUnit*, na linha seguinte é passado seu nome e na linha 3 é aplicado o estereótipo *aspectUnit* na instância de *ClassUnit*. Na linha 4 é criada uma instância de *TaggedValue*, na linha 5 é aplicada a *TagDefinition isPrivileged* no referido elemento e na linha 6 é passado o valor que a *TagDefinition* deve receber. O trecho de código-fonte presente na linha 7 é

responsável por vincular a *TaggedValue* criada (linhas de 4 a 6) à instância de *ClassUnit* (linhas de 1 a 3).

É importante ressaltar que os engenheiros de refatorações podem reutilizar o KDM-AO (leve) escrito em Java para escrever refatorações em modelos KDM. Outro ponto é que esse perfil pode ser reescrito em qualquer outra linguagem que tenha uma ferramenta capaz de interpretar o metamodelo KDM, atualmente disponibilizado nos formatos .CMOF, .ECORE e .XSD. pela OMG (2014). O mesmo pode ser aplicado à extensão pesada, uma vez que o metamodelo KDM, assim como os outros metamodelos da ADM, prima pela independência de linguagem e plataforma.

4.4.1 Extensão leve graficamente

Assim como na extensão pesada, também é possível criar a extensão leve graficamente. Para criar uma extensão leve no KDM graficamente de acordo com um perfil é necessário utilizar os *plug-ins* EMF e KDM – *Software Development Kit* (KDM-SDK).

Na Figura 4.11 é possível ver um exemplo gráfico de uma extensão leve no KDM. Na parte A da Figura é possível ver uma representação gráfica do mesmo trecho mostrado na Figura 4.9. Com o apoio do *plug-in* KDM-SDK as instâncias dos elementos do KDM são criados e os valores das propriedades podem ser armazenados em um modelo XMI. As partes B e C da Figura mostram as propriedades do estereótipo *AspectUnit* e da *TagDefinition isPrivileged*, respectivamente. Vale lembrar que todos esses valores também são Strings, assim como os valores passados na extensão leve feita programaticamente.

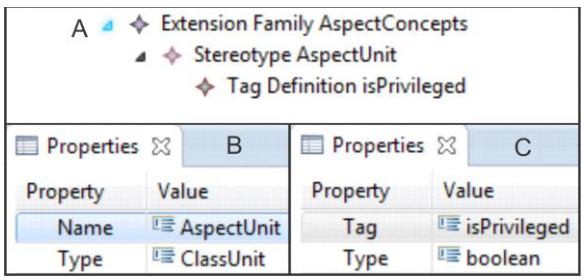


Figura 4.11 – Trecho da extensão leve feita graficamente.

A principal vantagem de se realizar uma extensão leve graficamente é que fica mais fácil visualizar e entender o funcionamento desse mecanismo. No contexto desta dissertação as versões gráficas e programáticas foram utilizadas em

momentos diferentes. As versões gráficas das extensões leve e pesada foram utilizadas para desenvolver o estudo de caso e as versões programáticas foram utilizadas para a realização do experimento real. Dessa forma, foi possível utilizar as duas versões de cada extensão, cada uma com um propósito diferente, como é mostrado nos capítulos seguintes.

A usabilidade tanto da extensão leve quanto da pesada são mostradas em detalhes no Capítulo 5. Nesse capítulo também é mostrado um estudo de caso que usa ambas as extensões para mostrar a representação dos principais conceitos da POA.

4.5 Considerações Finais

Neste capítulo foram apresentadas as duas extensões desenvolvidas neste trabalho. As extensões leve e pesada para POA no KDM foram descritas com mais detalhes, bem como os principais artefatos utilizados para auxiliar no processo de extensão que são: o diagrama KDM-AO baseado no perfil do Evermann e a tabela de mapeamento da UML para KDM. Notou-se que a extensão pesada no KDM é mais trabalhosa, uma vez que é necessário modificar o metamodelo original do KDM e criar uma ferramenta/*plug-in* para que ele possa ser reusado. Já o mecanismo de extensão leve é menos exigente, uma vez que seu processo de criação exige um esforço menor e seu reúso é mais facilmente adaptado em ferramentas existentes. Em linhas gerais, a principal vantagem de se utilizar a extensão pesada é devido a garantia de qualidade dos modelos produzidos, pois, existe uma checagem de tipos muito presente nesse mecanismo. Já a principal vantagem de se utilizar o mecanismo de extensão leve é a rapidez e praticidade de adicionar novos comportamentos em instâncias do KDM, pois, seu processo de criação é menos trabalhoso se comparado ao mecanismo de extensão pesada.

No Capítulo 5 é mostrado o primeiro processo de avaliação das extensões criadas que consta em mostrar um estudo de caso em que uma modernização orientada em framework transversal foi feita tanto para a extensão pesada quanto para a leve. A segunda parte desta avaliação leva em consideração critérios de

comparação que serão aplicados em ambas as extensões para mostrar qual das abordagens é a mais adequada para o contexto deste trabalho.

Capítulo 5

EXEMPLO DE USO E COMPARAÇÃO ENTRE AS EXTENSÕES

5.1 Considerações Iniciais

Este capítulo tem o propósito de mostrar a usabilidade das duas extensões apresentadas no Capítulo 4, bem como fazer uma análise comparativa em que as vantagens e desvantagens de cada abordagem são levantadas. A usabilidade das extensões é testada por meio de um estudo de caso para ambas as extensões, que visa comprovar a representatividade dos conceitos da POA em modelo KDM. Com o objetivo de verificar qual das extensões é mais indicada para o propósito deste trabalho e para auxiliar engenheiros de modernização no processo de escolha do mecanismo de extensão, foi elaborado um conjunto de critérios de avaliação. Esses critérios levam em consideração características como intuitividade de uso, produtividade, impacto ferramental e outros.

O capítulo está organizado da seguinte forma: na Seção 5.2 é apresentado o estudo de caso; na Seção 5.3 são mostradas especificações de baixo nível das extensões; na Seção 5.4 é mostrada a comparação qualitativa preliminar das extensões e na Seção 5.5 são discutidas as considerações finais do capítulo.

5.2 Estudo de caso

Para avaliar as extensões leve e pesada do KDM (KDM-AO) aqui feitas foram realizadas modernizações orientadas a frameworks transversais em um sistema de gerenciamento de uma loja de CD/DVD que está representado na Figura 5.1 (Camargo e Masiero 2005).

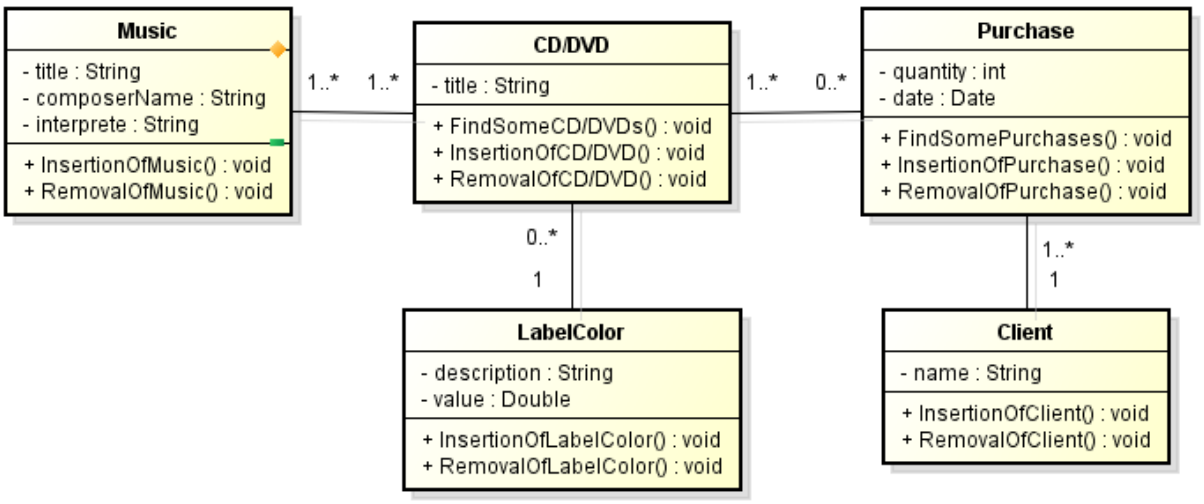


Figura 5.1 - Diagrama de classes do sistema de Loja de CD/DVD.

Na Figura 5.1 é possível ver os relacionamentos entre as classes do sistema. De acordo com o diagrama, o cliente (Classe *Client*) pode efetuar a compra (Classe *Purchase*) de um ou mais CDs ou DVDs. Todo CD/DVD possui uma etiqueta de cor (Classe *LabelColor*) que determina seu preço e essas etiquetas devem receber uma descrição e um valor, de tal forma que um CD só possua uma única etiqueta de cor. É possível ver também que uma música (Classe *Music*) pode ser atribuída a um ou mais de um CD ou DVD e os mesmos podem ser compostos por uma ou mais músicas.

É importante ressaltar que as modernizações aqui realizadas tiveram o objetivo de avaliar as extensões criadas no sentido de representação, ou seja, foi avaliado se os elementos criados na extensão pesada e os estereótipos da extensão leve eram suficientemente corretos para representar os conceitos da POA em um código-fonte escrito em AspectJ. Assim, não foi realizado um processo de modernização completo no sistema de loja de CD/DVD e sim apenas uma representação de alguns aspectos que fazem parte de todo processo de modernização.

O cenário de modernização que é referido aqui considera a existência de i) uma instância do KDM representando um sistema legado (aqui chamado de “KDM legado” ou “modelo base”) que precisa ser modernizado; ii) uma ou mais instâncias de FTs representados em KDM em um repositório; e iii) uma ou mais instâncias do KDM representando os elementos de instanciação, ou seja, classes e aspectos concretos criados pelo engenheiro de aplicação para acoplar os FTs ao modelo base (neste caso o KDM legado).

Neste 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 com aspectos. A ideia foi reusar um FT de persistência previamente desenvolvido para auxiliar este processo de modernização. Esta atividade tem como objetivo validar as extensões leve e pesada na representação dos conceitos orientados a aspectos.

É importante ressaltar que o foco desta avaliação é mostrar que é possível representar conceitos da OA por meio da extensão do KDM. Está fora do escopo o processo de mineração do KDM legado para buscar interesses transversais, removê-los ou fornecer uma ferramenta que facilite o acoplamento dos FTs. Assim, o primeiro passo foi obter uma instância do KDM que representasse o sistema da loja de CD/DVD. Esse processo foi realizado com o auxílio da ferramenta MoDisco (Bruneliere *et al*., 2010), que dentre outras funcionalidades é um *parser* que transforma automaticamente código-fonte Java em instâncias em XMI do KDM. O segundo passo foi focado em obter uma instância em KDM do FT de persistência, tanto na versão leve quanto na versão pesada. Para fazer esse processo na versão pesada da extensão foi usado um *plug-in* desenvolvido no contexto deste trabalho, chamado KDM-AO *plug-in*. Esse *plug-in* foi utilizado em classes e aspectos para que fossem convertidos manualmente em instâncias do KDM que representam o FT de persistência. Este *plug-in* permite que a extensão orientada a aspectos seja reusada, uma vez que ele encapsula todos os conceitos que foram adicionados no metamodelo KDM referente a POA. Já na versão leve da extensão foi utilizado o *plug-in* KDM-SDK fornecido pela própria OMG e que permite a criação e edição de modelos XMI em conformidade aos padrões da ADM. A versão leve possui outra abordagem de criação de suas instâncias que é apoiada por uma classe em Java que possui todos os estereótipos e tags necessárias para representar os conceitos da POA, também desenvolvido no contexto deste trabalho.

Nas seções seguintes são mostrados os principais trechos de código-fonte do FT com uma ampla variação no uso dos elementos que foram inseridos no metamodelo KDM relacionado ao perfil do Evermann (2007) para ambas as extensões.

Para representar a aplicação de loja de CD/DVD usando as extensões em KDM foram criados quatro aspectos e dois pontos de junção em ambas as extensões.

Para mostrar a usabilidade das extensões aqui feitas serão mostrados dois aspectos, um proveniente do FT de persistência e o outro proveniente do modelo de instanciação. O nome do primeiro aspecto é *ConnectionComposition*. Seu propósito é fornecer um comportamento base para a abertura e fechamento de conexões com o banco de dados. Durante a instanciação, engenheiros de aplicação precisam fornecer implementações completas para os pointcuts abstratos *openConnection*() e *closeConnection*(). Esse aspecto possui no seu corpo um atributo, dois pointcuts abstratos, um método concreto e outro abstrato e dois adendos. Na Figura 5.2 são mostradas as partes A e B, na parte A está representada uma instância da extensão leve e a parte B representa uma instância da extensão pesada. Cada linha contém um tipo de elemento e seus valores para cada tipo de extensão. Os parágrafos seguintes descrevem apenas os elementos da POA, os elementos da orientação a objetos estão fora do escopo deste trabalho.

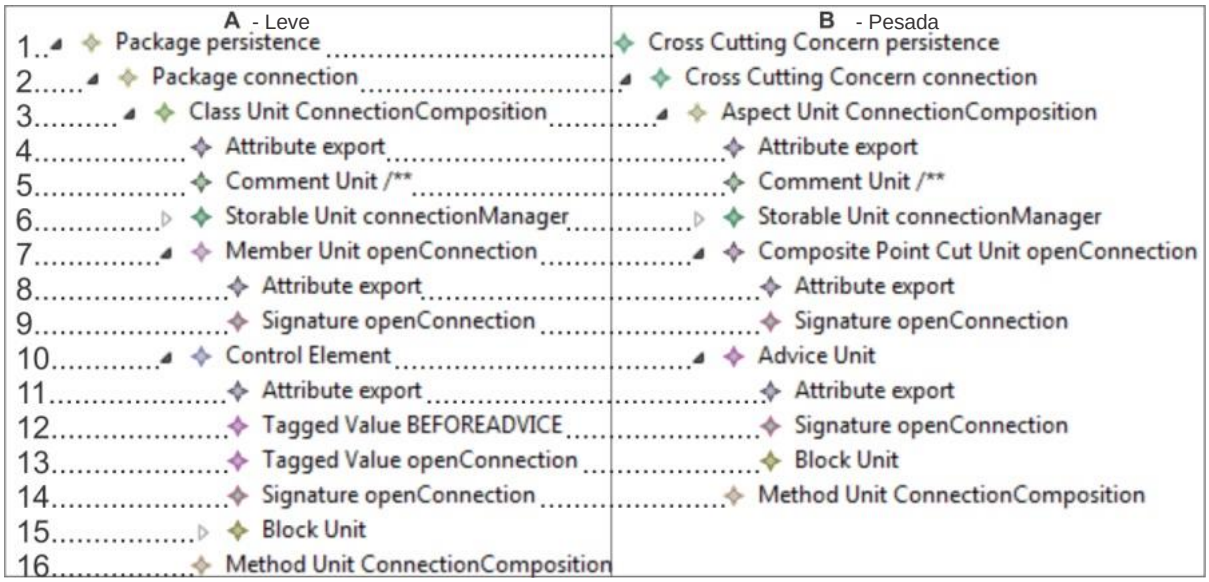


Figura 5.2- Trecho do aspecto *ConnectionComposition*.aj em KDM-AO leve e pesada.

Na linha 1 é possível visualizar um *Package* (Parte A) e *CrossCuttingConcern* (Parte B) cujo valor é persistence, ou seja, na leve isso é uma instância da

metaclasse *package* e na pesada é uma instância da metaclasse *CrossCuttingConcern*. A principal diferença entre eles é que além de serem provenientes de metaclasses diferentes, a leve é estereotipado com *CrossCuttingConcern*. Na linha 3 é mostrado o nome de um aspecto, na leve o estereótipo *AspectUnit* é aplicado em uma instância da metaclasse *ClassUnit* e a pesada não possui estereótipo, ou invés disso, uma instância da metaclasse *AspectUnit* é criada para representar o elemento aspecto. Para declarar um conjunto de junção ou ponto de junção na extensão leve é necessário aplicar um estereótipo no elemento *MemberUnit*, e isso é mostrado na linha 7 parte A. *CompositePointCutUnit* (Linha 7 parte B) é usado para representar *PointCuts* concretos ou abstratos de um aspecto.

O elemento em KDM que pode representar um adendo na extensão leve é *ControlElement* quando o estereótipo *AdviceUnit* é aplicado (linha 10). Na pesada *AdviceUnit* (linha 10) representa o adendo que foi declarado em um aspecto. É essencial preencher o valor para a propriedade *Advice Execution*, pois essa propriedade declara o tipo de adendo que o elemento representa (*After*, *before* ou *Around*). Esse tipo de declaração é muito diferente em ambas as extensões, por exemplo, enquanto que na pesada essa informação é informada nas propriedades do elemento *AdviceUnit*, na leve é necessário criar o elemento *TaggedValue* e aplicar a *TagDefinition* nele. Nas linhas 12 e 13 (Parte A) existem duas *TaggedValues*, a primeira armazena a *TagDefinition adviceExecution* e a segunda armazena o nome do *PointCut* que esse *AdviceUnit* (*ControlElement*) pertence.

O segundo aspecto que foi escolhido é do modelo de instanciação e seu nome é *myConnectionCompositionRules*. Esse aspecto armazena o nome do método no código-fonte base que será afetado pelo FT de persistência e é composto por dois pointcuts e um método que armazena o nome das classes que implementa o interesse de persistência. Na Figura 5.3 está mostrado um trecho do aspecto *myConnectionCompositionRules* na extensão pesada. Essa figura também mostra um *ExecutionPointCutUnit*. O *ExecutionPointCutUnit* intercepta a execução do método main na classe *FindSomeCDs* e está ligado ao *CompositePointCut openConnection*. Na linha 1 está representado o código-fonte para criar uma instância de *ExecutionPointCutUnit* na extensão pesada. Nas linhas 2 e 3 estão representadas declarações de algumas propriedades do elemento

ExecutionPointCutUnit, na linha 2 a propriedade *operation* é informada e na linha 3 é informado o *CompositePointCutUnit* que o *ExecutionPointCutUnit* pertence.

```
1..ExecutionPointCutUnit myExecution = codeFactory.eINSTANCE.createExecutionPointCutUnit();
2..myExecution.getOperation().add(FindSomeCDs.main);
3..myExecution.getComposite().add(openConnection); [...]
```

Figura 5.3 - Trecho do aspecto *myConnectionCompositionRules* na extensão pesada.

Na Figura 5.4 é mostrada uma instância do KDM na extensão leve que representa o mesmo trecho de código-fonte mostrado na Figura 5.3, mas nessa figura são usadas apenas metaclasses do metamodelo original do KDM.

```
1..MemberUnit myExecution = CodeFactory.eINSTANCE.createMemberUnit();
2..myExecution.getStereotype().add(Profile.executionPointCutUnit);
3..TaggedValue operation = KdmFactory.eINSTANCE.createTaggedValue();
4..operation.setTag(Profile.operation);
5..operation.setValue("FindSomeCDs.main");
6..TaggedValue composite = KdmFactory.eINSTANCE.createTaggedValue();
7..composite.setTag(Profiles.composite);
8..composite.setValue("openConnection");
9..myExecution.getTaggedValue().add(operation);
10..myExecution.getTaggedValue().add(composite); [...]
```

Figura 5.4 - Trecho do aspecto myConnectionCompositionRules na extensão leve.

Para criar um *ExecutionPointCutUnit* é necessário aplicar o estereótipo *executionPointCutUnit* no elemento *MemberUnit*, isto está representado nas linhas 1 e 2. Existem duas principais propriedades em um *ExecutionPointCutUnit* que é o *operation* e o *composite*, mas essas propriedades não existem no elemento *MemberUnit*. Para declarar propriedades adicionais em um elemento é necessário criar instâncias do elemento *TaggedValue* e aplicar uma *TagDefinition*. Logo após a criação da *TaggedValue* é necessário informar o valor da propriedade e seu valor é sempre do tipo string. A propriedade *operation* está declarada nas linhas 3, 4 e 5 e a propriedade está declarada nas linhas 6, 7 e 8. Uma vez criadas as *TaggedValues* e informadas as propriedades é necessário vincular a *TaggedValue* a seu respectivo elemento, o código-fonte que implementa essas informações são declaradas nas linhas 9 e 10.

O estudo de caso prova que ambas as extensões são capazes de representar todos os detalhes inerentes a este tipo de framework, bem como todos os conceitos da orientação a aspectos. Além disso, os resultados mostram que é possível modernizar sistemas legados para POA usando as extensões do KDM.

5.3 Especificações de Baixo Nível

Para representar um ponto de junção genérico na extensão pesada, é necessário apenas criar uma instância do elemento *OperationPointCutUnit* (Figura 5.3) e informar os parâmetros que entrecortam o sistema base. Mas, se um ponto de junção mais específico tiver que ser representado, é possível criar uma instância de um ponto de junção mais específico. Por exemplo, *GetPointCutUnit* e *SetPointCutUnit* são tipos especiais de *PointCutUnit* que representam acesso a um campo.

Outro exemplo é o fluxo de controle de um conjunto de junção. Um ponto de junção de controle de fluxo sempre especifica outro ponto de junção como um argumento. Existem dois tipos de pontos de junção de fluxo de controle, e na extensão pesada eles são representados por *CFlowPointCutUnit* e *CFlowBelowPointCut*. O primeiro ponto de junção captura todos os *OperationPointCutUnit* no fluxo de controle de um *PointCutUnit* específico, incluindo o *OperationPointCutUnit* combinado ao próprio *PointCutUnit*. O segundo ponto de junção exclui o *OperationPointCutUnit* no *PointCutUnit* especificado (Laddad, 2003).

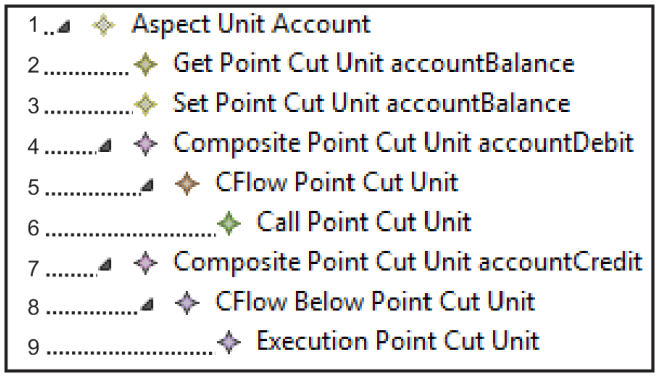


Figura 5.5 - Exemplo de especificações de baixo nível no *plug-in* KDM-AO versão pesada.

Na Figura 5.5 é mostrado como os *PointCutUnits* mencionados podem ser representados no *plug-in* KDM-AO na versão pesada da extensão. Nas Linhas 2 e 3 estão representados o *GetPointCutUnit* e *SetPointCutUnit*, representando que o campo *accountBalance* será entrecortado quando houver uma leitura ou escrita. O elemento *CompositePointCutUnit* (Linhas 4 e 7) encapsula elementos do tipo *PointCutUnit*, permitindo ao engenheiro de aplicação especificar os pontos do sistema base que serão afetados pelos *PointCutUnits*. Na Lina 6 é mostrado um

CallPointCutUnit que é modificado por um *CFlowPointCutUnit* (Linha 5) e na Linha 9 é mostrado um *ExecutionPointCutUnit* que é modificado por um *CFlowBelowPointCutUnit* (Linha 8).

Existem outras representações possíveis de pontos de junção apoiados pela extensão pesada aqui apresentada, assim, o nível de detalhes de uma instância de KDM-AO versão pesada dependerá principalmente do engenheiro de aplicação e do *parser* que criará as instâncias.

Na extensão leve, as especificações de baixo nível seguem as mesmas regras que as demais especificações neste mecanismo. É necessário criar uma instância da metaclasse do KDM original que representa o elemento a ser criado, por exemplo, para representar um *GetPointCutUnit* é necessário criar uma instância da metaclasse *MemberUnit* e depois aplicar o estereótipo *GetPointCutUnit*.

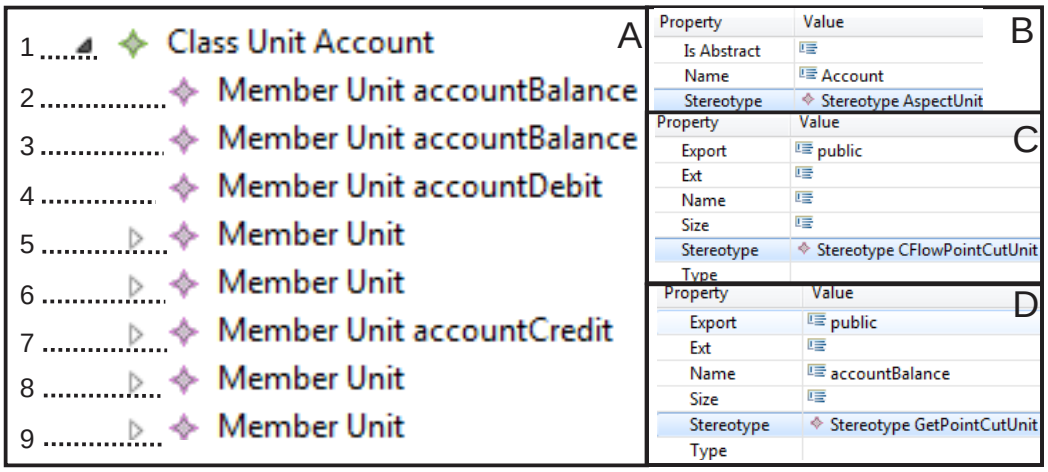


Figura 5.6 - Exemplo de especificações de baixo nível no *plug-in* KDM-AO versão leve.

Na Figura 5.6 parte A está representado um trecho do modelo em XMI que mostra exatamente o mesmo modelo apresentado na Figura 5.5. Na Figura é possível ver uma instância do elemento *ClassUnit* e nas linhas seguintes, instâncias do elemento *MemberUnit*. Contudo, não é possível identificar os estereótipos que foram aplicados nesses elementos, consequentemente, não é possível identificar qual elemento é um *CFlowBelowPointCutUnit* ou qual é um *ExecutionPointCutUnit*. Nesse caso é necessário verificar as propriedades de cada elemento e ver qual estereótipo foi aplicado, como mostrado na Figura 5.6 parte B, C e D.

No caso da Figura 5.6 parte B é possível ver que o estereótipo *AspectUnit* foi aplicado no elemento *ClassUnit Account*, o estereótipo *CFlowPointCutUnit* foi

aplicado no elemento *MemberUnit* da linha 5 e o estereótipo *GetPointCutUnit* foi aplicado na instância do elemento *MemberUnit* da linha 2.

Percebeu-se que por meio do mecanismo de extensão leve gráfico, o analista de sistemas pode facilmente se perder no modelo, uma vez que é requerida uma alta concentração para que o modelo seja instanciado corretamente.

5.4 Uma comparação qualitativa preliminar

Nesta seção é apresentada uma comparação preliminar entre as duas extensões aqui desenvolvidas. Esta comparação tem como objetivo fornecer uma base para os engenheiros de modernização em escolher a extensão que mais se adequa às suas necessidades. A comparação foi feita baseada em alguns critérios escolhidos e aplicados no estudo de caso e experimento realizados no escopo deste trabalho, tomando como base o trabalho de Magableh *et al*. (2012). Foi feita também uma listagem que contém as principais vantagens e desvantagens de cada extensão criada, e por meio dela, os engenheiros podem identificar qual extensão é melhor para o seu contexto de uso.

Para comparar os dois mecanismos de extensão foram definidos alguns critérios de avaliação. Esses critérios foram criados para comparar extensões, e objetiva caracterizar se um mecanismo é melhor que o outro.

O primeiro critério é **intuitividade na criação das instâncias**. Este critério tenta comparar a intuitividade de instanciar as extensões. Nesse sentido, intuitividade está relacionado a quão simples é para usar os elementos da extensão. Por exemplo, para criar um aspecto na extensão pesada é necessário apenas uma instância da metaclasse *AspectUnit*. Já na versão leve da extensão do KDM-AO, é necessária a criação de uma instância da metaclasse *ClassUnit* e aplicar o estereótipo *AspectUnit* nela. Neste caso é muito mais intuitivo usar a versão pesada do que a leve. Note que esse critério impacta tanto a criação de novas instâncias das extensões quanto a manutenibilidade dessas instâncias. Como a pesada é mais intuitiva, ela acaba sendo mais simples de criar e também de modificar (adicionar, remover) instâncias existentes da extensão. Isto pode ser visto se for feita uma comparação a Figura 5.3 e Figura 5.4.

O segundo critério é a **corretude de criação das instâncias**. Este critério está relacionado probabilidade de criação de instâncias com erro. A versão pesada da extensão é melhor do que a versão leve, pois, os valores passados na extensão leve são apenas tipos String, dessa forma, se o programador inserir alguma informação errônea, este modelo não será validado. Por outro lado, extensões pesadas são fortemente tipadas, assim, se uma propriedade recebe um tipo booleano, os únicos valores possíveis a serem informados são “verdadeiro” ou “falso”, qualquer outro valor será considerado como um erro de tipo e o modelo não será compilado.

O terceiro critério é o **impacto ferramental**, que está relacionado ao quão fácil de reusar a extensão é em diferentes ferramentas. A versão leve é mais fácil de reusar, pois, ferramentas podem ser previamente projetadas para considerar o reúso de perfis. Neste caso, a tarefa seria apenas importar o perfil para a ferramenta e usá-lo. No entanto, extensões pesadas são geralmente soluções proprietárias. Na maioria das vezes essas soluções não são projetadas para serem reusadas em outros contextos, ou seja, seu reúso não é tão simples quanto a versão leve. Embora não seja impossível criar uma extensão pesada reusável, é mais difícil quando comparada com a versão leve. Por exemplo, se uma ferramenta foi projetada para minerar instâncias de KDM, talvez essa ferramenta possa não funcionar corretamente se um KDM modificado for passado para ela. O impacto pode ser minimizado quando a extensão pesada adiciona apenas novas metaclasses no metamodelo original do KDM, ao invés de modificar as metaclasses existentes.

O quarto critério é a **produtividade em criação das extensões**. Este critério caracterizar o esforço na criação de extensões leves e pesadas. Claramente, a criação das extensões leves é bem mais fácil do que as extensões pesadas. Com a criação de uma extensão leve não é necessário se preocupar com os tipos, pois, basicamente todo valor que for informado será do tipo *String*. A criação de uma extensão pesada consome muito mais tempo, pois, além de criar uma metaclasse para cada novo conceito a ser adicionado no modelo, todos os relacionamento e atributos viram propriedades dentro de uma metaclasse e assim eles devem, obrigatoriamente, possui um tipo primitivo ou não. Um ponto importante para ser destacado aqui é que a produtividade na criação das extensões não é tão importante quanto a manutenibilidade delas. Isso acontece porque, em geral, as extensões serão criadas uma única vez e usada várias vezes, mas, já que as extensões leves

são mais facilmente construídas, consequentemente elas são mais facilmente mantidas. Portanto, se a criação leva dois meses ou duas horas isso não é importante. Vale lembrar que, embora a criação de extensões pesadas seja mais difícil, sua intuitividade no uso é melhor, assim, menos erros são esperados em suas instâncias.

O quinto critério é sobre a **produtividade para manutenção das extensões**. Este critério tenta caracterizar o esforço de adicionar, remover e modificar elementos de instâncias existentes. Quanto à manutenção das extensões, a leve se mostrou mais rápida e fácil que a pesada. Isso se deve ao fato de que se uma modificação seja necessária na extensão leve, deve-se apenas aplicar as atualizações e a extensão estará pronta para uso novamente. Por outro lado, depois de aplicar todas as mudanças na extensão pesada é necessário recompilar e atualizar o *plug-in* ou ferramenta.

O sexto critério é a **complexidade do XMI**. A ideia aqui é identificar qual XMI é mais complexo. Um XMI complexo é mais difícil de processar e vai requerer um esforço muito maior na escrita do algoritmo que o manipulará. Mais uma vez, extensões pesadas apresentam um XMI mais claro e intuitivo. Nessa extensão o algoritmo precisará apenas buscar pelo termo correto, por exemplo, *AspectUnit*. No entanto, na versão leve, a identificação de um elemento estendido requer a identificação da metaclasse que representa o elemento, a identificação do estereótipo e suas marcações (*tags*).

Na Tabela 5.1 é possível ver uma comparação entre as extensões leve e pesada. Existe uma seta para cima (🡅) quando a extensão é melhor e uma seta para baixo (🡇) quando a extensão é pior. Por meio dessa tabela é possível perceber que em relação aos critérios 1, 2 e 6 a extensão pesada se sobressaiu em relação a leve, por outro lado, em relação aos critérios 3, 4 e 5 a extensão leve teve um resultado melhor do que a pesada.

Em termos de reusabilidade de extensão, a versão leve é a melhor alternativa, uma vez que ferramentas podem ser mais facilmente preparadas para trabalhar com perfis. Por exemplo, considere a existência de uma ferramenta que aplica refatorações no modelo KDM. Esta ferramenta pode ser facilmente estendida para considerar estereótipos. No entanto, não é possível fazer com que a extensão

pesada seja reusada, o impacto na ferramenta vai requerer um esforço bem maior em relação à extensão leve.

Tabela 5.1 - Comparação entre os mecanismos de extensão.

Critério	Leve	Pesada
1. Intuitividade na criação das instâncias	↓	↑
2. Corretude de criação das instâncias	↓	↑
3. Impacto ferramental	↑	↓
4. Produtividade em criação das extensões	↑	↓
5. Produtividade para manutenção das extensões	↑	↓
6. Complexidade do XMI	↓	↑

De acordo com os resultados aqui obtidos, se o engenheiro de software pretende fazer com que sua extensão fique disponível para reúso, então o mecanismo de extensão leve é a melhor alternativa. Isso ocorre porque é possível disponibilizar apenas os estereótipos e tags criadas. Se ferramentas existentes já foram projetadas para aceitar perfis KDM, o seu reúso é bem simples.

Se o engenheiro de software pretende usar a extensão em uma organização ou ferramenta proprietária, o mecanismo de extensão pesada pode ser a melhor solução em termos de produtividade e qualidade das instâncias. Isso é importante, pois, a qualidade da instância modernizada terá um impacto direto no código-fonte gerado.

Baseando-se na experiência em realizar extensões leve e pesada, adquirida por meio deste trabalho, considera-se o mecanismo de extensão pesada a melhor abordagem para apoiar modernizações orientadas a aspectos, pois, esse tipo de modernização envolve muitos conceitos e a extensão pesada garante uma maior produtividade e qualidade nos modelos produzidos O conjunto de critérios apresentados objetivam apoiar engenheiros de software em escolher um dos tipos de extensão. A decisão final de qual mecanismo escolher vai depender de diversos detalhes, do contexto e dos cenários no qual a extensão será usada.

5.5 Considerações Finais

Neste capítulo foi apresentado um estudo de caso que modernizou em partes um sistema de gerenciamento de loja de CD/DVD para a POA por meio da modularização do interesse transversal de persistência. Esse estudo de caso serviu para mostrar que os conceitos da POA podem ser representados em ambas as extensões. Neste capítulo foram mostrados também os critérios utilizados para comparar as duas abordagens. Os resultados mostram que a extensão pesada é o mecanismo mais apropriado para lidar com conceitos de domínio específico com uma grande quantidade de conceitos, pois, esse mecanismo aumenta a produtividade e garante uma maior qualidade dos modelos. Contudo, é importante lembrar que a escolha do mecanismo dependerá fortemente do propósito na qual ela será desenvolvida.

No Capítulo 6 é mostrado o segundo processo de avaliação das extensões que consta em descrever a organização e os resultados de um experimento realizado no contexto da escrita de refatorações em um modelo KDM. Os sujeitos deste experimento são os alunos de pós-graduação em Ciência da Computação da Universidade Federal de São Carlos (UFSCar).

Capítulo 6

EXPERIMENTO

6.1 Considerações Iniciais

O propósito deste capítulo é apresentar uma análise experimental das extensões desenvolvidas no Capítulo 4 e utilizadas no Capítulo 5 objetivando comprovar sua usabilidade por meio de outros engenheiros de software. O principal objetivo deste capítulo é identificar, por meio da análise dos dados obtidos no experimento, qual das duas extensões é mais adequada para representar os conceitos da programação orientada a aspectos no metamodelo KDM. Para garantir que o procedimento aqui realizado fosse cientificamente aceito, foi seguido o procedimento descrito por Wohlin *et al.* (2000). Dessa forma, garante-se que o experimento aqui desenvolvido e aplicado possa ser replicado em outros contextos, tal como em um ambiente empresarial.

O capítulo está organizado da seguinte forma: na Seção 6.2 é feita uma contextualização sobre o desenvolvimento da experimentação; na Seção 6.3 é descrita a definição do experimento; na Subseção 6.4 é descrito o planejamento do experimento; na Seção 6.5 é mostrada a execução do experimento, nas Subseções 6.5.1, 6.5.2, 6.5.3 e 6.5.4 são mostradas a preparação, execução, validação dos dados em relação ao tempo e a validação dos dados em relação aos erros do experimento, respectivamente; na Seção 6.6 é discutida a análise das hipóteses, na Seção 6.7 são caracterizadas as ameaças à validade do experimento e na Seção 6.8 são discutidas as considerações finais do capítulo.

6.2 Desenvolvimento da experimentação

Com o objetivo inicial de validar a usabilidade das extensões leve e pesada propostas neste trabalho, foi planejado um experimento que levasse em consideração a prática de um grupo de alunos de pós-graduação da UFSCar.

A experimentação seguiu as fases experimentais, conforme proposto por Wohlin *et al.* (2000), e foi conduzida no primeiro semestre de 2014. O experimento procurou averiguar se a produtividade de engenheiros de modernização que criam refatorações é diferente ao usar extensões leves e pesadas do metamodelo KDM. Isto é, se o uso desses dois tipos de extensão influencia o tempo de implementação e também a quantidade de erros cometidos.

O cenário que este experimento reproduziu foi o de engenheiros de modernização que escrevem refatorações orientadas a aspectos. Em geral, esse profissional atua no desenvolvimento de ambientes de programação, conhecidos como IDEs, com é o Eclipse. A atividade simulou a implementação de refatorações que tem como entrada um determinado modelo KDM Orientado a Objetos (KDM-OO) e como saída outro KDM contendo aspectos de persistência usando a versão estendida do KDM. Assim, a atividade consistia em criar aspectos, conjuntos de junção, pontos de junção, adendos e declarações inter-tipo usando a versão estendida do KDM. Assume-se que quando mais fácil for usar o KDM estendido, melhor será a produtividade e a qualidade do modelo gerado.

Mais especificamente tem-se como objetivo investigar a seguinte questão de pesquisa (**QP**):

QP: Qual mecanismo de extensão (leve ou pesada) é mais intuitivo em relação a seu uso, consome menos esforço do modernizador de software para representar conceitos da POA e induz a menos erros quando da criação de refatorações orientadas a aspectos usando o KDM?

6.3 Definição do Experimento

O objetivo do estudo foi:

- **Analisar** a eficiência das duas extensões criadas (pesada e leve) para representar e instanciar os conceitos da orientação a aspectos em nível de modelo do KDM, levando em consideração a produtividade e quantidade de erros dos engenheiros de modernização ao utilizarem esses mecanismos;
- **Com o propósito** de avaliação;
- **Com respeito à** eficiência em termos de tempo gasto (produtividade) e número de erros (corretude);
- **Do ponto de vista de** modernizadores de software;
- **No contexto de** estudantes de pós-gradução em Ciência da Computação da UFSCar.

6.4 Planejamento do Experimento

O planejamento do experimento envolveu as seguintes etapas:

A. Seleção do contexto: O experimento ocorreu em ambiente universitário, sendo realizado com estudantes de pós-graduação de Ciência da Computação da UFSCar, no âmbito da disciplina de Engenharia de Software.

B. Formulação das Hipóteses:

- **Hipótese nula A (H0A)**: O tempo de desenvolvimento usando a extensão pesada é igual ou maior do que usando a extensão leve do KDM.
- **Hipótese alternativa A (H1A)**: O tempo de desenvolvimento usando a extensão pesada é menor do que usando a extensão leve do KDM.
- **Hipótese nula B (H0B)**: A quantidade de erros gerados no desenvolvimento usando a extensão pesada é igual ou maior do que usando a extensão leve do KDM.
- **Hipótese alternativa B (H1B)**: A quantidade de erros gerados no desenvolvimento usando a extensão pesada é menor do que usando a extensão leve do KDM.
- **Hipótese nula C (H0C)**: O tempo na manutenção usando a extensão pesada é igual ou maior do que usando a extensão leve do KDM.

- **Hipótese alternativa C (H1C)**: O tempo de manutenção usando a extensão pesada é menor do que usando a extensão leve do KDM.
- **Hipótese nula D (H0D)**: A quantidade de erros gerados na manutenção usando a extensão pesada é igual ou maior do que usando uma extensão leve do KDM.
- **Hipótese alternativa D (H1D)**: A quantidade de erros gerados na manutenção usando uma extensão pesada é menor do que usando uma extensão leve do KDM.

Seleção das Variáveis: As variáveis dependentes selecionadas para o experimento foram o tempo de se instanciar as extensões e o número de erros gerados durante a instanciação dessas mesmas extensões. As variáveis independentes foram:

1. **Aplicação:** Cada sujeito deveria desempenhar o papel de um engenheiro de modernização escrevendo refatorações para conversão de um KDM-OO em um KDM-OA. Eles também deveriam desempenhar o papel de engenheiros de modernização que precisam alterar (dar manutenção) nos códigos desenvolvidos*;*
2. **Ambiente de desenvolvimento:** Eclipse 4.3.1
3. **Tecnologias de desenvolvimento:** Java, KDM, MoDisco e EMF
4. **Seleção dos Participantes:** A seleção dos participantes foi feita através de **amostragem não probabilística por conveniência** (Wohlin *et al.*, 2000), selecionando os indivíduos disponíveis mais próximos para participarem do experimento. Participaram do estudo 14 alunos de mestrado e doutorado (sujeitos) do curso de Ciência da Computação da UFSCar, matriculados na disciplina Engenharia de Software.

C. Projeto do Experimento: O projeto do experimento descreve como os testes experimentais são organizados e executados. O estudo foi planejado **em blocos** (WOHLIN *et al.*, 2000), a fim de assegurar que o efeito do fator nível de experiência dos participantes não interferisse nos resultados dos tratamentos do fator processo de utilização das extensões. Dessa forma, os alunos foram divididos em 2 grupos (blocos) homogêneos, de modo que cada grupo possuísse, tanto quanto possível, médias semelhantes de nível de experiência. A experiência dos participantes foi avaliada por um Formulário de caracterização dos participantes **-** documento utilizado para capturar a experiência profissional e experiência nos assuntos relacionados ao estudo (Java, KDM, MoDisco e EMF). Esse formulário foi

entregue a cada participante uma semana antes da execução do experimento, de forma que fosse possível planejar o estudo antecipadamente.

D. **Tipo do Projeto:** O tipo do projeto do experimento foi de **dois fatores** (*processo de modernização e manutenção*) com **dois tratamentos** (*extensões leve e pesada*) **completamente randomizados** (WOHLIN *et al.*, 2000). Nesse tipo de projeto um mesmo objeto a ser manipulado durante o estudo é utilizado em todos os tratamentos e os participantes, divididos em dois grupos, foram aleatoriamente atribuídos a cada tratamento. A Tabela 6.1 ilustra a distribuição dos grupos em relação ao tratamento e às fases do experimento. Cada grupo foi submetido aos dois tratamentos, mas em fases diferentes. O experimento seguiu o formato em que os participantes são alocados em grupos homogêneos para que o nível de experiência deles não impactasse nos resultados. Um Formulário de Caracterização de Participante (Apêndice A) foi elaborado e distribuído entre os participantes para determinar o nível de experiência de cada um.

G. Instrumentação: Os materiais necessários para apoiar os sujeitos no experimento foram previamente planejados, compreendendo a definição dos objetos que seriam manipulados, as diretrizes para orientar os sujeitos na execução dos processos, bem como os instrumentos para a coleta de dados e medição.

Tabela 6.1 - Distribuição dos grupos em relação aos tratamentos.

	Grupo 1	Grupo 2
Fase 1	Leve	Pesada
Fase 2	Pesada	Leve

6.5 Execução do Experimento

A execução do experimento foi realizada em três passos: (i) **preparação,** (ii) **execução** e (iii) **validação dos dados.** Tais passos são explicados a seguir:

6.5.1 Preparação

Neste passo os materiais definidos para a instrumentação do experimento foram efetivamente elaborados, a saber:

- **Objetos:** Dois projetos em Java no eclipse foram elaborados, um para cada abordagem. No projeto da extensão pesada, chamado de *Heavyweight_experiment* havia duas classes em java. A primeira classe se chamava *Heavyweight_maintenance_experiment* e continha um código-fonte que representava um trecho de uma aplicação, para que o sujeito realizasse as atividades de manutenção. Já a segunda classe, de nome *Heavyweight_development_experiment*, também continha um pequeno trecho de código-fonte, contudo, ele servia apenas para usar como referência no modelo que seria desenvolvido. Por exemplo, esse trecho representava partes de uma aplicação, como um método, que deveria ser referenciado em um ponto de junção que o sujeito teria que criar nas atividades de desenvolvimento.

  A segunda classe se chamava *Lightweight_maintenance_experiment* e também continha um código-fonte que representava um trecho de uma aplicação, para que o sujeito realizasse as atividades de manutenção. A segunda classe, de nome *Lightweight_development_experiment*, estava em branco, uma vez que as referências a outras classes, métodos e atributos são feitos por meio de Strings, ou seja, não havia uma verificação de conformidade.
- **Diretrizes:** Os seguintes documentos foram produzidos para serem utilizados no experimento: (i) Formulário de caracterização dos participantes, para obtenção da experiência profissional e nos assuntos relacionados diretamente ao estudo (Apêndice A); (ii) Formulário de consentimento, para aprovação e ciência dos participantes dos objetivos do estudo e das condições de participação (Apêndice A); (iii) Descrição das atividades do experimento real (Apêndice B), com as instruções da sua execução; (iv) manual de utilização das extensões na versão leve e outro na versão pesada (Apêndice C); (v) tabela de mapeamento dos elementos de AspectJ para as extensões leve e pesada (Apêndice C) e (vi) diagrama de classes das extensões (Apêndice C), para que o sujeito soubesse quais atributos e relacionamentos pertencem a certo elemento, esse artefato é complementar ao artefato (v).
- **Instrumentos para Coleta de Dados**: Medições em experimento são conduzidas por meio dos dados que são coletados. Em experimentos

envolvendo atividades executadas por seres humanos, os dados geralmente são coletados manualmente por meio de formulários ou entrevista. Assim, para a coleta de dados desse experimento, foi elaborado o ***Formulário de coleta de dados*** (Apêndice B), no qual os sujeitos deveriam preencher todas as informações requeridas durante a execução do experimento. No mesmo formulário de coleta de dados, foi colocado um espaço para a *avaliação qualitativa*, no qual cada participante individualmente, após a execução do experimento, deveria reportar sua percepção sobre dificuldades, facilidade e sugestões durante a utilização das extensões. Esse formulário foi elaborado no mesmo arquivo que as Descrições das atividades do experimento real, de tal forma que o sujeito tivesse todas as informações e pudesse marcar seus tempos no mesmo documento.

Foram realizados treinamentos com todos os participantes do experimento ao longo de três semanas que antecederam sua realização especificamente nos dias 13, 20 e 27 de junho de 2014. No primeiro dia foi dado um treinamento que preparou os sujeitos para a realização do experimento real, esse treinamento contou com uma contextualização de POA, FTs, como criar instâncias do KDM usando as extensões leve e pesada e exercícios. No segundo dia foi feito um piloto com os sujeitos. Esse piloto objetivou simular as atividades que seriam realizadas no dia do experimento real. Essa atividade serviu para otimizar alguns detalhes como o tempo necessário para a realização de cada atividade e a forma como as atividades seriam distribuídas. Vale ressaltar que cada uma das etapas que foram comentadas ocorreu no horário da disciplina de Engenharia de Software 2014.1 com 4 horas de aula cada.

O treinamento contou com duas horas de explicação sobre os tópicos abordados e as outras duas horas foram passadas atividades de treinamento para os alunos, de tal forma que eles se habituassem à desenvolver as instâncias do KDM-AO (Leve e Pesada). No dia do treinamento foram entregues aos sujeitos os formulários de caracterização e consentimento, para que essas informações fossem utilizadas no dia do piloto.

No segundo dia, foram tiradas as dúvidas dos sujeitos e aplicado o piloto. Esse piloto foi organizado no formato de grupos, em que os 14 sujeitos foram

divididos em dois grupos iguais e cada grupo usaria as duas abordagens, mas em fases diferentes, como está representada na Tabela 6.1.

Na Tabela 6.1 é possível ver que os sujeitos do grupo 1 utilizaram a extensão leve na fase 1 e a pesada na fase 2. Já o grupo 2 utilizou a extensão pesada na fase 1 e a extensão leve na fase 2. Todos usaram a mesma aplicação hipotética, isto é, tiveram que criar aspectos com nomes genéricos e sem contexto. Isso foi diferente do experimento real, em que os aspectos que foram criados eram referentes ao interesse de persistência.

A justificativa dessa organização de grupos e fases se dá pelo fato de que se objetivava investigar o fator aprendizado, assim, poderíamos analisar se a utilização de uma abordagem na fase 1 implicaria em um tempo menor na fase 2. Essa distribuição também permite averiguar se o fator "pessoas" teve influência no resultado. É importante lembrar que a Tabela 6.1 foi usada tanto no piloto quanto no experimento real.

Algo importante de ser ressaltado aqui é quanto ao exemplo usado nas atividades. Optou-se por usar o mesmo exemplo a ser instanciado em ambas as fases, denominado no piloto de Aplicação 1 (App1). Essa aplicação consiste em um conjunto de aspectos com conjuntos de junção, adendos e declarações inter-tipo. Julgou-se que o exemplo usado é suficiente e bastante representativo do domínio. Isto é, não haveria diferença na criação de aspectos, conjuntos de junção e adendos com outros nomes. O que realmente impacta o tempo de desenvolvimento e a manutenção é a quantidade de elementos e a complexidade deles. No experimento real a decisão foi a mesma. Optou-se por usar um conjunto de aspectos do interesse de persistência para ambas as fases e ambos os grupos.

Ainda no piloto, foram entregues aos sujeitos uma tabela para coleta de dados e um formulário de análise qualitativa. A tabela de coleta de dados continha campos em que os sujeitos marcariam o tempo de início e término de cada atividade e outros campos que marcariam o tempo de parada e retorno. Esses campos registrariam o tempo que os sujeitos gastariam consultando os demais artefatos entregues nessa etapa (Apêndice C). Contudo, percebeu-se que 90% dos sujeitos não preencheram essas informações, pois afirmaram que muitas vezes as consultas eram muito rápidas e eles acabavam se esquecendo de registrar os tempos. Mediante essas informações, foi decidido desconsiderar esses campos nos artefatos

utilizados no experimento real. As atividades realizadas nessa etapa pelos sujeitos podem ser analisadas na Tabela 6.2.

Nela estão descritas as atividades atribuídas aos sujeitos no piloto, essas atividades foram passadas por meio de uma apresentação e antes de serem realizadas elas foram explicadas detalhadamente, para que os sujeitos não tivessem dúvidas do que deveria ser feito quando o tempo começasse a contar. O tempo previsto para cada uma dessas atividades foi baseado no tempo em que o autor deste trabalho utilizou para realizar as mesmas atividades. Lembrando que o tempo total de 60 minutos foi previsto para realizar apenas uma das fases, independente da abordagem utilizada. Uma vez que a fase 1 foi concluída, as atividades foram repetidas e os sujeitos que estavam utilizando a abordagem leve passou a usar a pesada e vice-versa.

Tabela 6.2 - Descrição das atividades realizadas no piloto pelo sujeitos.

Atividades	Descrição (Extensão leve e pesada)	Tempo previsto (Minutos)
Atividade 1	Criar três instâncias do elemento *CrosscuttingConcerns*.	6
Atividade 2	Criar três instâncias do elemento *Aspect* e fazer o vinculo com os elementos criados na atividade anterior.	6
Atividade 3	Criar duas instâncias do elemento *PointCut* com um *Joinpoint* cada.	12
Atividade 4	Criar duas instâncias do elemento *PointCut* com dois *Joinpoints* cada.	12
Atividade 5	Criar três instâncias do elemento *Advice* e vincular a outros três *PointCuts* criados nas atividades anteriores	6
Atividade 6	Criar cinco instâncias do elemento *Inter-Type Declaration*.	9
Atividade 7	Adicionar uma propriedade ao elemento Aspect e tranformar um pointcut com dois joinpoints em um pointcut com um joinpoint.	9
Tempo total		**60**

O treinamento e o piloto foram conduzidos de forma que, ao final da fase de treinos, cada participante estivesse suficientemente apto para utilizar qualquer uma das abordagens.

O dia do experimento real seguiu basicamente as mesmas etapas que o piloto, as principais diferenças foram: (i) as atividades, bem como suas descrições, foram entregues aos sujeitos, como mostrado no Apêndice B e (ii) todas as oito atividades foram explicadas antes de ser dado início ao experimento real, essa decisão foi tomada para otimizar o tempo dos participantes e assim evitar uma maior quantidade de interrupções. Foi explicado aos sujeitos que depois que o tempo fosse iniciado eles só poderiam tirar dúvidas com o material entregue e quaisquer outras dificuldades encontradas deveriam ser marcadas no formulário de avaliação

qualitativa (Apêndice B). As atividades realizadas nessa etapa estão descritas na Tabela 6.3.

As atividades do experimento real diferiram das que foram aplicadas no piloto principalmente porque foram otimizadas em relação ao tempo previsto e pela quantidade de instâncias a serem criadas em cada atividade. Essas atividades foram elaboradas visando reproduzir um código-fonte mais real, ou seja, que representasse a criação de aspectos de persistência. Enquanto que no piloto o código-fonte passado possuía nomes como *Aspect1*, *PointCut2* ou *Advice3*, no experimento esses elementos possuíam nomes como *SecurityComposition*, *saveOrUpdate* e *affectedClasses*, como mostrado no Apêndice B. Dessa forma era possível simular refatorações reais em um processo de modernização orientado a aspectos.

Tabela 6.3 - Descrição das atividades realizadas no experimento real pelo sujeitos.

Atividades	Descrição (Extensão leve e pesada)
	Atividades de Desenvolvimento (escrita de novas refatorações)
Atividade 1	Criar três instâncias do elemento *CrosscuttingConcerns*
Atividade 2	Criar três instâncias do elemento *Aspect* e fazer o vinculo com os elementos criados na atividade anterior.
Atividade 3	Criar três instâncias do elemento *PointCut* com um *joinpoint* cada
Atividade 4	Criar duas instâncias do elemento *PointCut* com dois *joinpoints* cada
Atividade 5	Criar três instâncias do elemento *Advice* e vincular a outros três PointCuts específicos, criados nas atividades anteriores
Atividade 6	Criar cinco instâncias do elemento *Inter-Type Declaration*.
	Atividades de Manutenção (alteração de refatorações existentes)
Atividade 7	Adicionar três propriedade ao elemento *Aspect*.
Atividade 8	Tranformar um *PointCut* com um *Joinpoint* em um *PointCut* com dois *Joinpoint*.
Tempo total	

Outra diferença fundamental no experimento real foi a categorização das atividades em "desenvolvimento" e "manutenção". Optou-se por fazer essa diferenciação para investigar não só a produtividade na escrita de novas refatorações como também na manutenção de refatorações existentes. Dessa forma, as atividades de 1 até 6 referem-se à desenvolvimento, pois visam apenas à criação de novos elementos, enquanto que as atividades 7 e 8 visam modificação de elementos existentes.

Para melhor ilustrar e fornecer uma base para entender melhor a seção de análise dos dados, é mostrado na Figura 6.1 e Figura 6.2 parte da resolução da atividade 4 da Tabela 6.3.

Na Figura 6.1 é mostrado um trecho de código-fonte referente à atividade 4 do experimento feito pelo sujeito 9. Esse trecho corresponde à criação de um conjunto de junção com dois pontos de junção, um do tipo *Execution* e outro do tipo *Call*. Nas linhas de 2 a 4 é criado o elemento *CompositePointCutUnit* e são informadas suas propriedades. Nas linhas de 5 a 7 é criado o ponto de junção *ExecutionPointCutUnit*, é passada sua operação e esse elemento é vinculado ao conjunto de junção criado na linha 2. Já nas linhas de 8 a 10, é criado o ponto de junção *CallPointCutUnit*, é passada sua operação e também é vinculado ao conjunto de junção da linha 2.

```
1..//Activity 4
2..CompositePointCutUnit pointcut4 =
CodeFactory.eINSTANCE.createCompositePointCutUnit();
3..pointcut4.setName("dirtyObjects");
4..pointcut4.setCompositionType(PointCutCompositionType.OR);
5..ExecutionPointCutUnit pointcutEx1 =
CodeFactory.eINSTANCE.createExecutionPointCutUnit();
6..pointcutEx1.getOperation().add(save);
7..pointcutEx1.getComposite().add(pointcut4);
8..CallPointCutUnit pointcutCall1 = CodeFactory.eINSTANCE.createCallPointCutUnit();
9..pointcutCall1.getOperation().add(delete);
10.pointcutCall1.getComposite().add(pointcut4);
```

Figura 6.1 - Trecho de código-fonte da atividade 4 usando a extensão pesada.

Já na Figura 6.2 é mostrada a representação da mesma atividade 4, que foi mostrado na Figura 6.1, só que utilizando o mecanismo de instanciação leve do KDM-AO. É possível notar que o mesmo código-fonte que foi representado na Figura 6.1 com 10 linhas, precisou de 18 linhas a mais para ser implementado. Ambos códigos-fonte representam a mesma funcionalidade, no entanto, na extensão leve é necessário escrever mais, já que os elementos da programação a aspectos não existe no metamodelo e com isso é necessário criar elementos e aplicar estereótipos. Outro ponto-chave que faz com que a codificação leve se torne mais trabalhosa é que todas as propriedades de elementos estereotipados só podem ser informados por meio da criação de uma instância do elemento *TaggedValue*, assim, toda propriedade precisa de pelo menos quatro linhas de código-fonte.

```
1..//Activity 4
2...MemberUnit pointcut4 = CodeFactory.eINSTANCE.createMemberUnit();
3...pointcut4.getStereotype().add(Profiles.compositePointCutUnit);
4...pointcut4.setName("dirtyObjects");
5.............TaggedValue tagPointcut4 = KdmFactory.eINSTANCE.createTaggedValue();
6.............tagPointcut4.setTag(Profiles.compositionType);
7.............tagPointcut4.setValue("OR");
8.............pointcut4.getTaggedValue().add(tagPointcut4);

9......MemberUnit execution1 = CodeFactory.eINSTANCE.createMemberUnit();
10.....execution1.getStereotype().add(Profiles.executionPointCutUnit);
11............TaggedValue tagExecution1 = KdmFactory.eINSTANCE.createTaggedValue();
12............tagExecution1.setTag(Profiles.operation);
13............tagExecution1.setValue("save()");
14....................TaggedValue tag1 = KdmFactory.eINSTANCE.createTaggedValue();
15....................tag1.setTag(Profiles.composite);
16....................tag1.setValue("pointcut4");
17............execution1.getTaggedValue().add(tagExecution1);
18............execution1.getTaggedValue().add(tag1);

19.....MemberUnit call1 = CodeFactory.eINSTANCE.createMemberUnit();
20.....call1.getStereotype().add(Profiles.callPointCutUnit);
21............TaggedValue tagCall1 = KdmFactory.eINSTANCE.createTaggedValue();
22............tagCall1.setTag(Profiles.operation);
23............tagCall1.setValue("delete()");
24....................TaggedValue tag2 = KdmFactory.eINSTANCE.createTaggedValue();
25....................tag2.setTag(Profiles.composite);
26....................tag2.setValue("pointcut4");
27............call1.getTaggedValue().add(tagCall1);
28............call1.getTaggedValue().add(tag2);
```

Figura 6.2 - Trecho de código-fonte da atividade 4 usando a extensão leve.

6.5.2 Execução

Primeiramente, os participantes foram posicionados nos grupos com base na sua pontuação do **Formulário de Caracterização de Participante**. Cada grupo ficou com sete sujeitos. Na Tabela 6.4 está representada a pontuação dos sujeitos em relação às questões do formulário e o total de sujeitos para cada categoria de pontuação. Ambos os grupos ficaram com a mesma quantidade de sujeitos na mesma categoria, ou seja, cada grupo ficou com um sujeito com pontuação muito baixa (0 a 10), um sujeito com pontuação baixa (11 a 20), três sujeitos com a pontuação média (21 a 30), um sujeito com a pontuação alta (31 a 40) e um sujeito com a pontuação muito alta (41 a 50). Após tomarem conhecimento de qual grupo pertenciam, os participantes receberam o material para executarem as atividades.

Cada sujeito recebeu um manual de utilização das extensões de acordo com seu grupo e sua fase e uma lista com oito atividades, sendo que das oito, seis atividades eram de desenvolvimento e duas eram atividades de manutenção, como mostrado na Tabela 6.3. As atividades eram as mesmas tanto para a extensão leve quanto para a pesada.

As atividades de desenvolvimento consistiam em escrever código para instanciação de aspectos, conjuntos de junção, pontos de junção, adendos, pacotes de interesses transversais e declarações inter-tipo, como mostrados nas atividades de 1 a 6 na Tabela 6.3 Já as atividades de manutenção consistiam em modificar o código escrito na atividade anterior acrescentando ou modificando propriedades dos elementos existentes. Cada atividade teve um tempo específico, que foi definido com a ajuda das atividades realizadas durante o piloto. O tempo previsto para cada atividade pode ser visto na Tabela 6.3Tabela 6.3 - Descrição das atividades realizadas no experimento real pelo sujeitos. no Apêndice B.

Tabela 6.4 – Distribuição dos sujeitos em relação à pontuação.

	Pontuação				
	Muito baixa	**Baixa**	**Média**	**Alta**	**Muito Alta**
	0 a 10	**11 a 20**	**21 a 30**	**31 a 40**	**41 a 50**
Sujeito1			24		
Sujeito2			21		
Sujeito3			27		
Sujeito4					41
Sujeito5		19			
Sujeito6			21		
Sujeito7	9				
Sujeito8			24		
Sujeito9			29		
Sujeito10				32	
Sujeito11					42
Sujeito12		20			
Sujeito13	10				
Sujeito14				35	
Total	**2**	**2**	**6**	**2**	**2**

Para prosseguir à próxima etapa, validação de dados, foi necessário analisar os dados para identificar os dados que apresentam um grande afastamento das demais séries de dados, aqui chamados de *outliers* de acordo com Wohlin *et al.* (2000). A existência de *outliers*, geralmente, traz prejuízos a interpretação dos resultados dos testes estatísticos aplicados às amostras, assim, foi necessário identificar possíveis *outliers* para que eles pudessem ser removidos.

Na Figura 6.3 e Figura 6.4 são mostrados dois gráficos de Box-plot que leva em consideração a média e o desvio padrão dos dados. Esses gráficos são indicados para identificar valores muito afastados dos demais dados classificados.

Na Figura 6.3 são levados em consideração os tempos de todos os sujeitos de ambos os grupos que utilizaram a extensão leve nas atividades de desenvolvimento. Pode-se notar que nessa figura um dos sujeitos está marcado fora dos quartis, assim é possível classifica-lo como *outlier*. Para identificar o sujeito que apresentou esse dado fora do padrão, recorreu-se aos dados brutos e foi identificado que o sujeito 11 utilizou 7 minutos para realizar uma atividade que a média foi 3.5, aproximadamente.

Já na Figura 6.4, que leva em consideração os tempos de todos os sujeitos que utilizaram a extensão leve nas atividades de manutenção, não houve indícios de *outliers*, uma vez que todos os sujeitos apresentaram valores próximos.

Ainda levando em consideração o tempo das atividades, foram analisados também os dados dos sujeitos que utilizaram a extensão pesada, nas duas fases e de ambos os grupos. Na Figura 6.5 foi gerado o gráfico de Box-Plot para essas condições e três sujeitos apresentaram dados discrepantes, nas atividades 3, 5 e 6. Para identificar esses sujeitos foram analisados os dados brutos para identificar esses valores. No caso, o sujeito 13 é o *outlier* das atividades 3 e 6 e o sujeito 3 é o *outlier* da atividade 5.

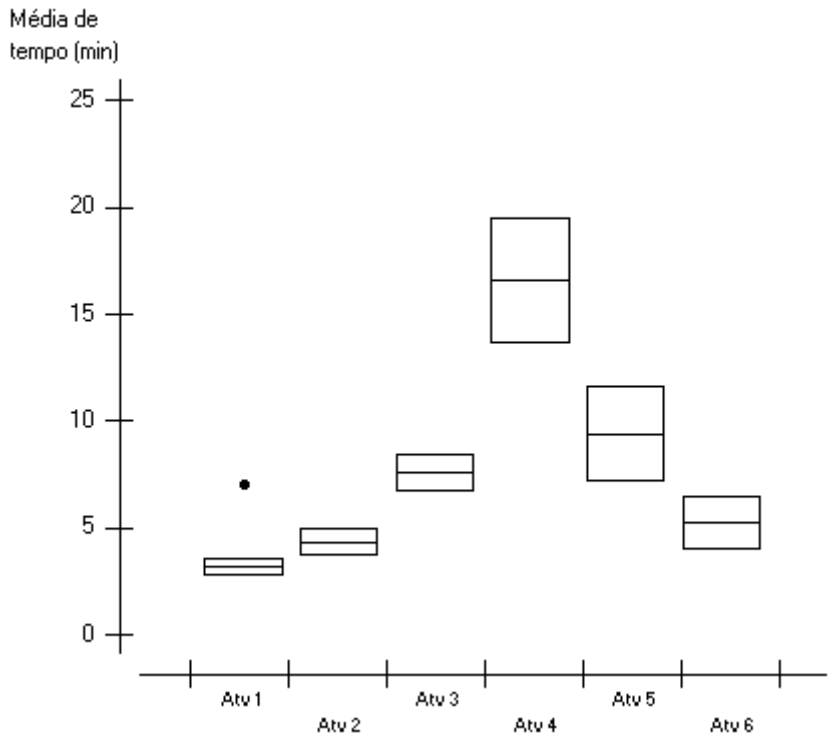


Figura 6.3 – Identificação de *Outliers* extensão Leve em relação ao tempo (Atividades de Desenvolvimento).

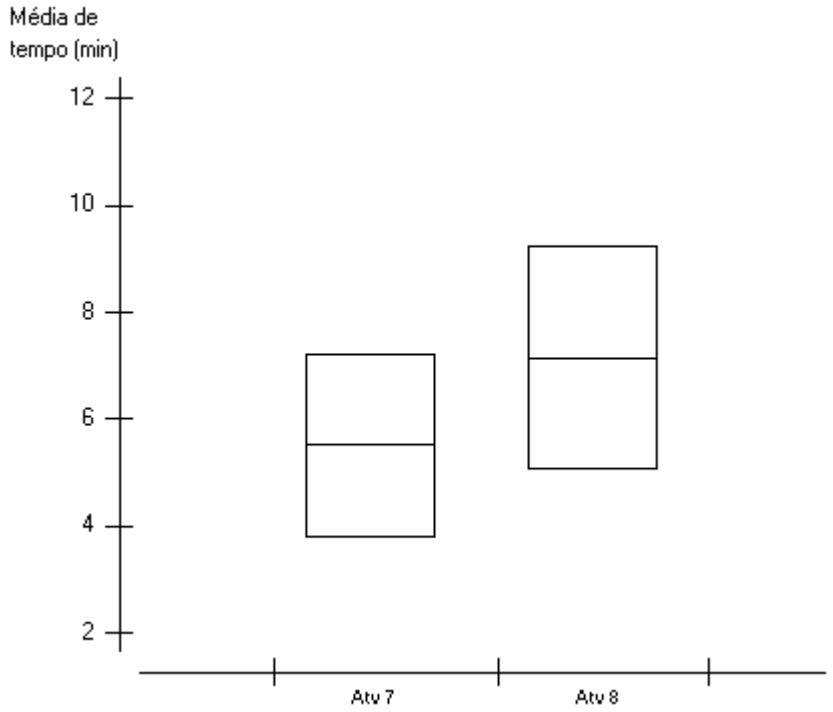


Figura 6.4 - Identificação de *Outliers* extensão Leve em relação ao tempo (Atividades de Manutenção).

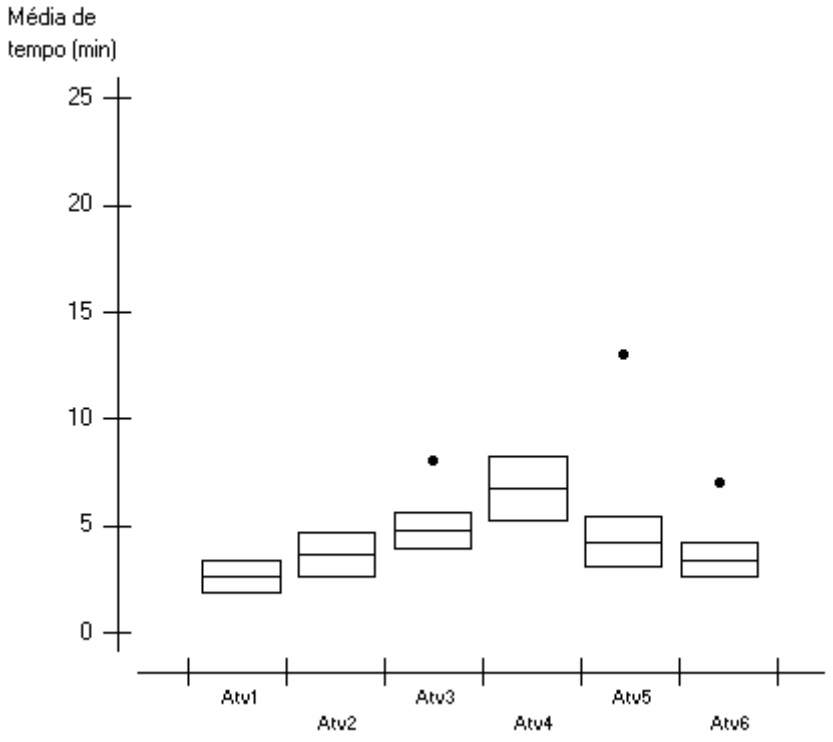


Figura 6.5 - Identificação de *Outliers* extensão Pesada em relação ao tempo (Atividades de Desenvolvimento).

Na Figura 6.6 está representado o gráfico referente aos sujeitos que utilizaram a extensão pesada na atividade de manutenção. É possível notar que apenas na atividade 8 foi registrado um *outlier*, que corresponde ao sujeito 3.

Mediante a análise dos gráficos de tempo para as extensões leve e pesada, conclui-se que foram identificados cinco *outliers* no total e os dados produzidos por esses sujeitos serão retirados da análise, mas, apenas nas atividades em que eles foram identificados. Esse processo de análise é discutido na seção de validação dos dados.

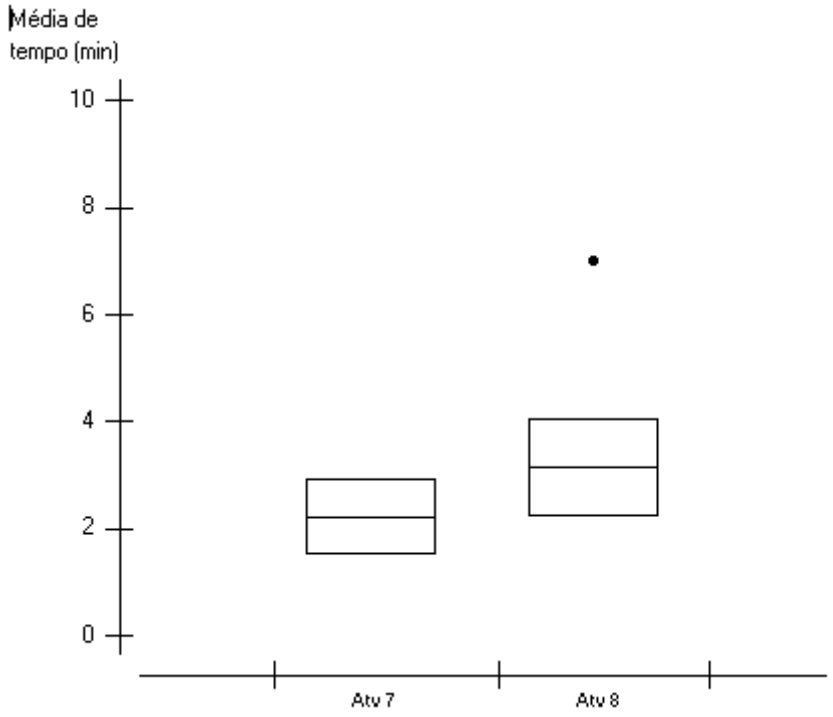


Figura 6.6 - Identificação de *Outliers* extensão Pesada em relação ao tempo (Atividade de Manutenção).

Outros dados que também passaram pelo processo de identificação de *outliers* é a quantidade de erro cometida pelos sujeitos em cada atividade. A identificação dos *outliers* em relação à quantidade de erros foi feita por meio da análise dos dados brutos, uma vez que a leitura do gráfico de Box-Plot com essa amostra de dados não gerou um gráfico muito legível.

Com base no treinamento ministrado e na aplicação do piloto, esperou-se que os sujeitos produzissem a menor quantidade de erros possível, uma vez que as atividades propostas no experimento real continha, basicamente, as mesmas atividades com um nível de detalhes maior e mais fiel a códigos-fonte reais. Baseando-se por essas duas etapas, foi estipulada uma média de até três erros para cada atividade, em ambas as extensões. Para valores acima dessa faixa seria aplicado outro tratamento, antes de classifica-lo como *outlier*. Caso um sujeito apresente um número de erros maior do que 3 em uma atividade, seria analisado se

algum outro sujeito apresentou a mesma quantidade ou um valor próximo, de até uma unidade.

Na Tabela 6.5 está representada a quantidade de erros de todos os sujeitos na extensão leve. A tabela está dividida em duas partes, Atividades de Desenvolvimento e Atividades de manutenção. Referente a parte das atividades de desenvolvimento, o sujeito 3 foi identificado como sendo um *outlier* na atividade 6 (Atv 6), uma vez que ele cometeu 6 erros e os demais sujeitos não cometeram nenhum. Na atividade 4 (Atv 4) existem dois sujeitos que poderiam ser identificados como outliers, sujeitos 3 e 12. Contudo, eles não foram descartados da análise final já que o mesmo valor se repetiu mais de uma vez e essa quantidade de erros é "aceitável" na utilização da extensão leve, já que ela demanda mais esforço e concentração em suas instanciações.

Tabela 6.5 - Identificação de *Outliers* pela quantidade de erros na extensão leve.

	Extensão Leve							
	Atividades de Desenvolvimento						Atividades de Manutenção	
	Atv 1	Atv 2	Atv 3	Atv 4	Atv 5	Atv 6	Atv 7	Atv 8
Sujeito 1	0	0	0	0	0	0	0	0
Sujeito 2	0	0	0	0	0	0	2	0
Sujeito 3	0	0	0	5	2	**6**	3	**5**
Sujeito 4	0	0	0	0	0	0	0	0
Sujeito 5	0	0	0	0	0	0	2	0
Sujeito 6	0	0	0	0	0	0	0	0
Sujeito 7	0	0	0	0	0	0	0	0
Sujeito 8	0	0	0	0	0	0	0	0
Sujeito 9	0	0	0	0	0	0	0	0
Sujeito 10	0	0	0	0	0	0	0	1
Sujeito 11	0	0	0	0	0	0	2	0
Sujeito 12	0	0	0	5	3	0	1	2
Sujeito 13	0	0	0	0	0	0	0	0
Sujeito 14	0	0	0	2	0	0	1	0

Tabela 6.6 - Identificação de *Outliers* pela quantidade de erros na extensão pesada.

	Extensão Pesada							
	Atividades de Desenvolvimento						Atividades de Manutenção	
	Atv 1	Atv 2	Atv 3	Atv 4	Atv 5	Atv 6	Atv 7	Atv 8
Sujeito 1	0	0	0	0	0	0	0	0
Sujeito 2	0	0	0	0	0	0	0	0
Sujeito 3	0	3	0	3	0	**6**	0	3
Sujeito 4	0	0	0	0	0	0	0	0
Sujeito 5	0	0	2	0	0	0	0	1
Sujeito 6	0	0	0	0	0	0	0	0
Sujeito 7	0	0	0	0	0	0	0	0
Sujeito 8	0	0	0	0	0	0	0	0
Sujeito 9	0	0	0	0	0	0	0	0
Sujeito 10	0	0	0	0	0	0	0	0
Sujeito 11	0	0	0	0	0	0	0	0
Sujeito 12	0	0	0	0	0	0	0	1
Sujeito 13	0	0	0	0	0	0	0	0
Sujeito 14	0	0	0	0	0	0	0	0

Analisando a segunda parte da Tabela 6.5, percebeu-se que nas atividades de manutenção, 6 dos 14 sujeitos cometeram erros e desses 6 apenas um foi identificado como *outlier* (sujeito 3), por ter apresentado cinco erros na atividade 8.

Na Tabela 6.6 é feita a análise dos erros cometidos pelos sujeitos que utilizaram a extensão pesada, de ambas as fases, nas atividades de desenvolvimento e manutenção. Nas atividades de desenvolvimento (1 a 6), o sujeito 3 foi identificado como *outlier* por ter apresentado 6 erros na atividade 6, quando nenhum outro sujeito apresentou erro nas mesmas condições. Já nas atividades de manutenção, nenhum sujeito foi identificado como *outlier*, uma vez que a quantidade de erro apresentada estava dentro do esperado.

6.5.3 Validação dos dados em relação ao tempo

Depois de aplicado o experimento, os dados coletados foram classificados em uma tabela para serem analisados. Como dito anteriormente, cada atividade possuía um tempo previsto para sua realização, dessa forma, todos os sujeitos preencheram o formulário de coleta de dados com os tempos iniciais e finais de cada uma delas. Esses tempos que foram marcados foram tabelados obedecendo a seguinte fórmula: Tt = Tf – Ti. Em que Tt é o tempo total da atividade, Tf é o tempo final e o Ti é o tempo inicial.

Para iniciar a análise, foi selecionada a amostra de dados referentes às atividades de desenvolvimento de ambos os grupos na extensão leve. Na Tabela 6.7 estão representadas as médias de tempo das atividades de desenvolvimento. Por meio dessa tabela é possível tirar algumas conclusões como, o grupo que utilizou a extensão leve na primeira fase, demorou mais tempo para realizar as mesmas atividades que o grupo 2, que utilizou a extensão leve na segunda fase. Essa diferença pode ser explicada devido ao ganho de conhecimento de uma fase para a outra.

Apenas na atividade 1, o grupo da fase 2 reportou uma média maior do que a fase 1. Analisando os dados brutos, dois dos sujeitos apresentaram o tempo de 4 minutos, esses foram os dados que modificaram a média. Existem algumas variáveis que podemos levar em consideração, que são: O sujeito pode ter demorado um pouco mais para fechar o tempo da atividade fazendo alguma verificação e/ou pode ter levado um tempo a mais para organizar as ideias e começar a implementação.

De qualquer forma, a diferença não foi considerada grande o suficiente para invalidar as conclusões aqui apresentadas.

Como as atividades eram as mesmas, o grupo que desenvolveu com a extensão pesada na primeira fase já sabia o que desenvolver e já tinha uma base/aquecimento para desenvolver novamente as atividades da segunda fase com a abordagem leve.

Outro ponto é que o grupo 2 utilizando a extensão leve na fase 2 levou mais tempo em média para realizar as mesmas atividades que o grupo 1 utilizando a extensão pesada na fase 2. O grupo 1 que desenvolveu a leve na fase 1 levou em média 19,63% a mais de tempo do que o grupo 2, que utilizou a leve na fase 2.

Tabela 6.7 - Média de tempo das atividades de desenvolvimento (Leve).

	Média de tempo das atividades de desenvolvimento (Leve)						Somatória das atividades	Média Geral
	Atv 1	Atv 2	Atv 3	Atv 4	Atv 5	Atv 6		
Grupo1 Fase1	3	4,28	8,14	18,71	10,57	5,57	50,27	46,14
Grupo2 Fase2	3,33	4,28	7	14,42	8,14	4,85	42,02	

Na Tabela 6.8 é apresentada a média de tempo das atividades de desenvolvimento na extensão pesada. Notou-se que o grupo que utilizou a extensão pesada na fase 2 foi mais rápido 5,84% do que o grupo que utilizou a mesma extensão na fase 1. A mesma explicação que foi dada para a Tabela 6.7 também serve a essa análise, ou seja, acredita-se que há um ganho de conhecimento de uma fase para a outra, independente do mecanismo de extensão utilizado.

Tabela 6.8 - Média de tempo das atividades de desenvolvimento (Pesada).

	Média de tempo das atividades de desenvolvimento (Pesada)						Somatória das atividades	Média Geral
	Atv 1	Atv 2	Atv 3	Atv 4	Atv 5	Atv 6		
Grupo1 Fase 2	2,28	3,57	4,57	6,57	4,66	3	24,65	25,37
Grupo2 Fase 1	2,85	3,71	5	6,85	3,85	3,83	26,09	

Contudo, foi visto que no caso das atividades de desenvolvimento da leve a diferença foi de 19,63% e para a pesada foi de 5,84%. A justificativa para essa disparidade de valores se dá principalmente pela complexidade do código-fonte necessário para criar instâncias do KDM-AO leve. Dessa forma é possível concluir que independente da disparidade entre esses valores, a extensão pesada permite a criação de código-fonte mais rápida, em relação ao mecanismo de extensão leve.

As próximas tabelas a serem analisadas são referentes às atividades de manutenção em ambas as extensões. Na Tabela 6.9 são mostradas as médias de tempo das atividades 7 e 8, percebe-se que a fase 1 levou cerca de 45% a mais de

tempo para ser produzida do que a fase 2. Na Tabela 6.10, referente às atividades de manutenção com a extensão pesada, é possível analisar a diferença entre as duas fases em relação ao tempo. A diferença entre essas fases é de apenas 9,14%, ou seja, na fase 2 essas atividades foram realizadas mais rapidamente. Mais uma vez o ganho de conhecimento de uma fase para a outra pode ser usado como justificativa para essa diferença de valores.

Tabela 6.9 - Média de tempo das atividades de manutenção (Leve).

	Média de tempo das atividades de manutenção (Leve)		Somatória das atividades	Média Geral
	Atv 7	Atv 8		
Grupo1 Fase 1	6,43	8,57	15	12,64
Grupo2 Fase 2	4,57	5,71	10,28	

Tabela 6.10 - Média de tempo das atividades de manutenção (Pesada).

	Média de tempo das atividades de manutenção (Pesada)		Somatória das atividades	Média Geral
	Atv 7	Atv 8		
Grupo1 Fase 2	2,28	3,33	5,61	5,38
Grupo2 Fase 1	2,14	3	5,14	

Ao serem analisados os dados das tabelas, percebe-se que em geral a extensão pesada é mais rápida do que a extensão leve, isso ocorre devido à grande quantidade de código-fonte que precisa ser escrito para representar um trecho usando a extensão leve.

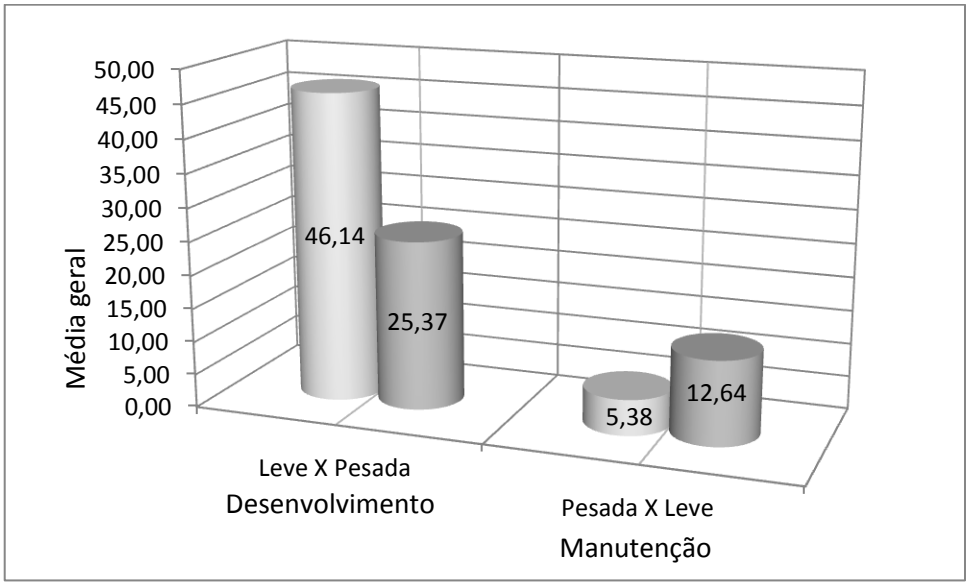


Figura 6.7 - Comparação das médias de tempo geral entre as duas extensões nas atividades de desenvolvimento e manutenção.

Na Figura 6.7 é apresentada outra perspectiva de comparação em que são confrontadas todas as médias de tempo das duas abordagens nas atividades de desenvolvimento e manutenção. Por meio desse gráfico é possível reafirmar que a

abordagem pesada permite a criação de modelos mais rapidamente, ou seja, é a abordagem mais produtiva. De acordo com as médias gerais apresentadas nas Tabelas de Tabela 6.7 à Tabela 6.10, de maneira geral, o mecanismo de extensão pesada agiliza em 81,86% a implementação de novas refatorações (Desenvolvimento) em modelos KDM-AO. Enquanto que a mesma extensão (pesada) agiliza a manutenção de refatorações de modelos em KDM-AO em 134,94%.

6.5.4 Validação dos dados em relação aos erros

Outro dado tabelado foi o número de erros de cada sujeito em cada atividade. No final do experimento foi solicitado aos sujeitos que enviassem seus projetos para uma posterior correção. A correção feita levou em consideração erros no código-fonte tais como: erros de sintaxe, erros de tipo, valores passados erroneamente, trechos de código-fonte incompletos e código-fonte sem vínculo correto.

Os erros de sintaxe são erros de digitação ou outro erro no código que bloqueiam a execução do programa. Os erros de tipo foram considerados como os valores passados em campos que exigiram um valor booleano e foi passado um inteiro, por exemplo. Os valores passados erroneamente são os nomes de variáveis ou nome de elementos que deveriam ser preenchidos com certo valor e foram preenchidos com outro, incluindo palavras em caixa alta ou baixa. Os trechos de código-fonte que faltavam algum comando foram considerados como trechos incompletos e os trechos que estavam vinculados com elementos errados ou sem vínculo algum foram considerados como código-fonte sem vínculo correto.

Diferentemente do que foi feito na análise dos dados em relação ao tempo das atividades, a análise dos dados de erros não levou em consideração a média de erros uma vez que os gráficos e tabelas gerados por meio da média não mostrou uma representatividade satisfatória dos dados. Assim, a análise desta seção foi feita por meio da somatória de erros dos sujeitos em relação às atividades de desenvolvimento e manutenção nos dois mecanismos de extensão.

Para a confecção dos gráficos e tabelas aqui apresentados foram desconsiderados os dados de todas as atividades dos *outliers* identificados na Seção 6.5.2. Lembrando que o sujeito 3 foi identificado como *outlier* nas atividades de desenvolvimento leve e pesada e nas atividades de manutenção da extensão

leve, logo, todos os seus dados foram desconsiderados da somatória de erros dos demais sujeitos.

Na Figura 6.8 está representada a somatória dos erros identificados nas atividades de desenvolvimento nas extensões leve e pesada. Pecebeu-se que na extensão leve houve ocorrência de erros nas atividades 4 e 5, enquanto os sujeitos que utilizaram a extensão pesada só produziram erros na atividade 3. Julga-se que as atividades iniciais (de 1 a 2) são atividades mais básicas e servem como aquecimento para que os sujeitos “subam o nível” de dificuldade gradualmente. As atividades mais complexas estão concentradas nas atividades de 3 a 6, uma vez que elas exigem mais concentração e esforço para serem desenvolvidas.

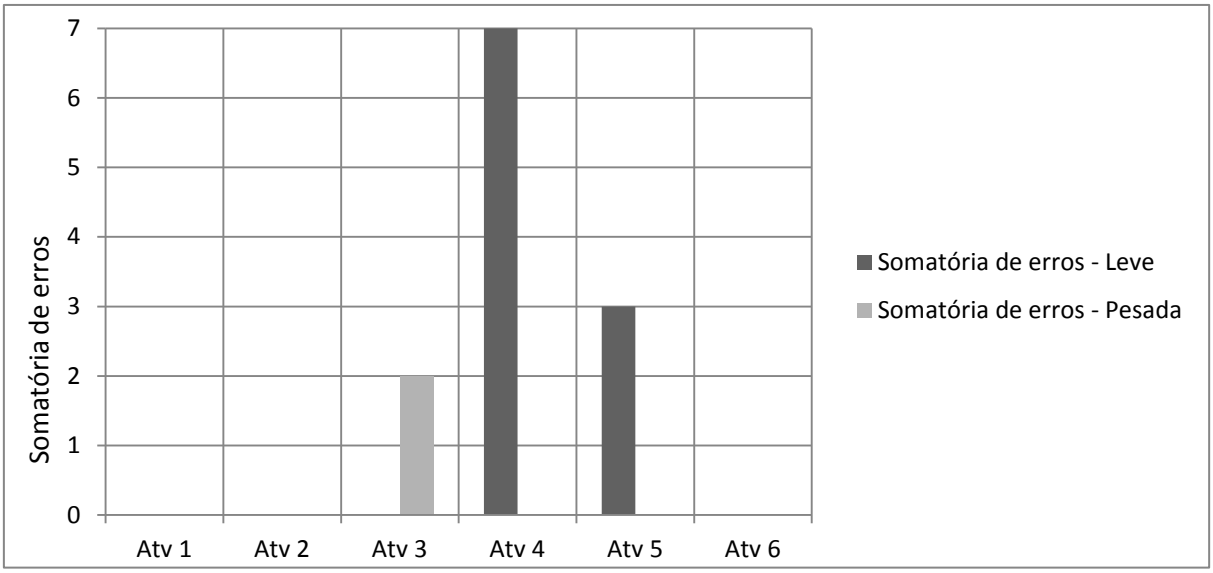


Figura 6.8 – Somatória de erros das atividades de desenvolvimento (Leve x Pesada).

Baseando-se na Figura 6.8 é possível afirmar que na extensão leve foram identificados 400% a mais de erros do que na extensão pesada, ou seja, cerca de 4 vezes a mais de erros.

Na Figura 6.9 estão representadas as somatórias de erros para as atividades de manutenção nas duas extensões. Na extensão pesada não foram identificados erros para atividade 7, apenas a extensão leve apresentou erros, no total de 8 erros. Foram identificados erros na atividade 8 para as duas extensões, contudo, foram identificados cerca de 150% a mais de erros na extensão leve. No geral, a extensão pesada apresentou 450% a menos de erros em relação à extensão leve.

Diante da análise dos gráficos de somatória de erros é possível concluir que a incidência de erros é bem maior nas atividades de desenvolvimento (400%) e manutenção (450%) quando se utiliza a extensão leve, uma vez que esse

mecanismo permite uma maior inserção de código-fonte errado, já que quase não existe uma verificação de tipos.

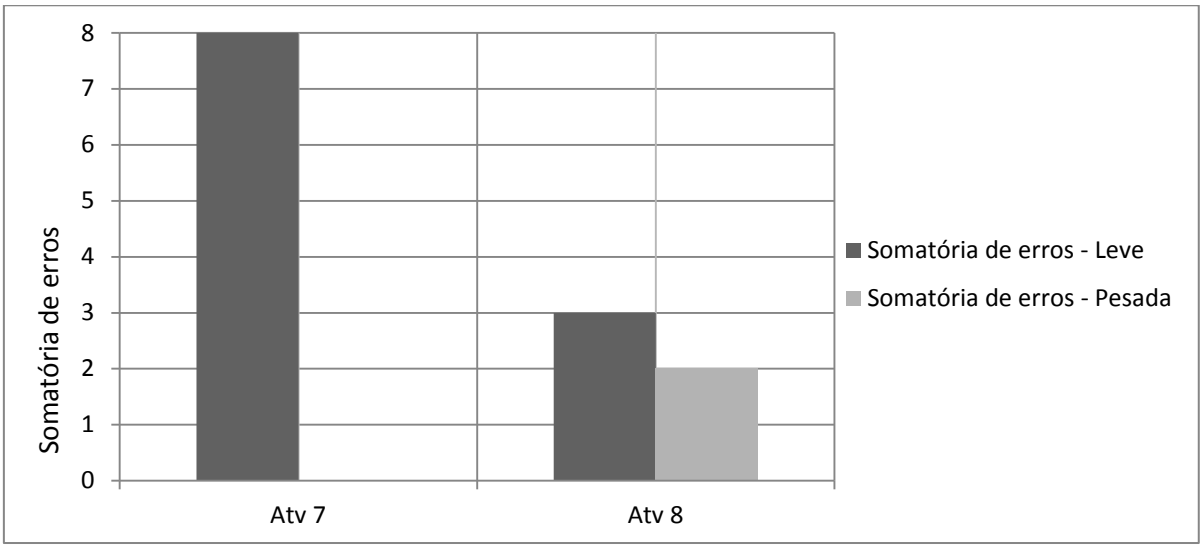


Figura 6.9 - Somatória de erros das atividades de manutenção (Leve x Pesada).

6.6 Análise das Hipóteses

Analisando as hipóteses aqui formuladas, pode-se concluir que:

A Hipótese nula **A (H0A)** foi descartada, uma vez que foram encontradas diferenças entre o tempo de desenvolvimento dos engenheiros de software nas extensões e o tempo da extensão leve é maior do que a pesada. Dessa forma, a Hipótese alternativa A **(H1A)** foi aceita, já que os sujeitos que usaram a extensão pesada, independente da fase, foram mais produtivos (81,86%), como foi provado e discutido na Tabela 6.7, Tabela 6.8 e Figura 6.7.

Da mesma forma que H0A, a Hipótese nula B **(H0B)** foi desconsiderada, já que a extensão leve apresentou uma maior ocorrência de erros nas atividades de desenvolvimento do que a extensão pesada. Sendo assim, a hipótese válida é a **H1B**, que afirma que os sujeitos que usaram a extensão pesada nas atividades de desenvolvimento produziram um modelo em KDM-AO mais correto, com uma ocorrência de erros de 400% a menos em relação à extensão leve.

A Hipótese nula **C (H0C)** também foi descartada, uma vez que foram encontradas diferenças entre o tempo de manutenção dos engenheiros de software nas extensões e o tempo da extensão leve é maior do que a pesada. Dessa forma, a

Hipótese alternativa C **(H1C)** foi validada, já que os sujeitos que usaram a extensão pesada foram mais rápidos (134,94%) nas atividades de manutenção, como foi provado e discutido na Tabela 6.9, Tabela 6.10 e Figura 6.7.

A Hipótese nula D **(H0D)** foi desconsiderada, já que a extensão leve apresentou uma maior ocorrência de erros nas atividades de manutenção do que a extensão pesada. Sendo assim, a hipótese válida é a alternativa **H1D**, que afirma que os sujeitos que usaram a extensão pesada nas atividades de manutenção produziram um modelo em KDM-AO mais correto, com uma ocorrência de erros de 450% a menos em relação à extensão leve.

6.7 Ameaças à Validade do Experimento

Existem alguns pontos que podem ser discutidos em relação a ameaças à validade desse experimento.

Os diferentes níveis de conhecimento dos participantes poderiam afetar os dados coletados. Pensando nisso, foi aplicado um formulário de caracterização dos participantes, com o objetivo de avaliar o nível de cada sujeito e assim, poder dividi-los em dois grupos homogêneos. Outra ação que foi tomada para minimizar essa ameaça foi a realização do treinamento que teve como principal objetivo nivelar o conhecimento entre os sujeitos e fornecer a base necessária para a utilização das duas abordagens.

Os participantes poderiam ter aprendido com o piloto ou com treinamentos anteriores e aplicado seus conhecimento ou código-fonte no experimento real. Em relação ao primeiro ponto, era esperado que com o piloto os sujeitos tivessem uma ideia de como o experimento real seria, uma vez que objetivava-se medir o desempenho deles levando em consideração o tempo e a quantidade de erros. Em relação ao segundo ponto, foram entregues classes em Java, uma que representava as atividades de desenvolvimento e outra para as atividades de manutenção, em um projeto específico, dessa forma foi dificultada a reutilização de código-fonte, uma vez que o ambiente era configurado previamente, sem mais classes ou projetos abertos.

Pode ser argumentado que as abordagens foram testadas apenas no meio acadêmico, uma vez que os participantes do experimento são alunos do curso de

pós-graduação. Todavia, o experimento real foi aplicado de tal forma que ele poderia ser replicado em ambiente empresarial.

6.8 Considerações Finais

Neste capítulo foi apresentada a análise experimental desta pesquisa. Foram mostrados os passos seguidos e o planejamento que foi feito para a realização do experimento. É importante lembrar que os procedimentos adotados aqui seguiram a metodologia de Wohlin *et al.* (2000), dessa forma, é possível replicar os procedimentos aqui realizados. Todos os artefatos produzidos neste experimento estão disponíveis para consulta no link http://tinyurl.com/p7fzwgp.

Por fim, foi apresentada uma análise dos dados coletados em relação ao tempo levado para a realização das atividades propostas e a quantidade de erros produzidos pelos sujeitos do experimento. Por meio deste capítulo, provou-se que a extensão pesada é mais produtiva em relação a tempo, linhas de código-fonte e erros, no contexto desta pesquisa.

No Capítulo 7 são apresentados os trabalhos relacionados levando em consideração as semelhanças e diferenças em relação ao trabalho desenvolvido nesta dissertação.

Capítulo 7

TRABALHOS RELACIONADOS

7.1 Considerações Iniciais

Neste capítulo são descritos os principais trabalhos relacionados que foram encontrados na literatura. De acordo com o mapeamento sistemático realizado por Durelli *et al*. (2014a) poucos trabalhos propuseram extensões no metamodelo KDM, dentre eles, alguns não mostraram dados o suficiente que pudessem ser utilizados nesta dissertação. Entretanto, foram identificados alguns trabalhos que deram suporte ao desenvolvimento deste trabalho, assim, neste capítulo são mostradas as principais semelhanças e diferenças encontradas.

O capítulo está organizado da seguinte forma: na Seção 7.2 é mostrado parte do mapeamento sistemático, na Seção 7.3 são mostrados os trabalhos relacionados que deram apoio a esta dissertação e na Seção 7.4 são discutidas as considerações finais do capítulo.

7.2 Mapeamento Sistemático

Com o objetivo de conhecer o estado atual das pesquisas em relação à modernização de sistemas legados utilizando ADM/KDM, estabeleceu-se a seguinte pergunta: Como a ADM e seus metamodelos estão dando suporte ao processo de modernização de sistemas legados?

Foi realizada uma busca nas principais bases de dados para computação: Scopus, ACM, IEEE, Spring, Scienci Direct, Engeneering Village e Web of Science. A busca foi realizada nos idiomas português e inglês, a string de busca foi definida na IEEE e após ser refinada foi traduzida para as outras bases de dados.

A String de busca foi a seguinte “("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") OR ("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")”.

Essa string foi definida sem nenhum tipo de restrições, tais como: ano de publicação, nome do autor, área de conhecimento, dentre outros. Optou-se pela construção de uma string genérica para se ter uma noção mais abrangente do que se está sendo pesquisando na área de modernização de sistemas no contexto da ADM. A string contém termos que não são especificamente da ADM e seus metamodelos, pois, alguns trabalhos que mencionavam strings como “*Model Driven Modernization*” continham informações sobre ADM.

Foram encontrados 354 artigos que passaram por duas etapas de seleção. Na primeira seleção dos artigos foram lidos os títulos, abstracts e palavras-chaves. Dos 354 artigos obtidos, 62 foram duplicados, 134 foram rejeitados na primeira seleção e 153 passaram para a fase de extração de dados. Os artigos rejeitados não atendiam a nenhum critério de inclusão (Tabela 7.1), logo não descreviam ou mencionavam os metamodelos do KDM ou ADM, nem discutiam sobre o processo de modernização de sistemas legados, dessa forma, foram descartados por não atenderem a nenhum dos critérios de inclusão, mostrados na primeira coluna da Tabela 7.1 (Durelli *et al*., 2014a).

Na fase seguinte, chamada de extração de dados, foram aceitos 146 artigos, dos quais 2 foram rejeitados por não descreverem nenhum metamodelo, 31 artigos foram identificados como duplicados e 113 foram classificados para etapa da leitura

do artigo completo. Nessa etapa, dos 113 artigos, 28 artigos não foram encontrados para a leitura completa gratuita.

Tabela 7.1 - Critérios da pesquisa

Critérios de Inclusão (I) e Exclusão (E) dos artigos	
(I) Mostrar de alguma forma a modernização de sistemas legados;	(E) Não ter acesso ao artigo complete;
(I) Falar sobre a modernização com ADM, mencionando pelo menos um de seus metamodelos;	(E) Se vários artigos descreverem um estudo idêntico, apenas o mais recente será selecionado;
(I) O estudo primário relatar de alguma forma ADM.	(E) Não estar em inglês ou português;
----------	(E) Não mostrar nenhum metamodelo.

Dos artigos lidos, 53 mencionaram diretamente o uso do ADM na composição do trabalho, 12 dos artigos apenas citaram o ADM como trabalhos relacionados e 12 artigos tratavam sobre modernização de sistemas, mas não mencionaram a ADM (Durelli *et al*., 2014a).

7.3 Trabalhos Relacionados

O mapeamento sistemático que foi realizado no contexto da ADM, objetivou identificar os principais tópicos que estavam sendo pesquisados sobre esse assunto. Percebeu-se por meio deste mapeamento que grande parte dos trabalhos estavam realizando modernizações de sistemas legados por meio de mudanças de linguagens de programação e modernizações de banco de dados legado. Os resultados deste mapeamento sistemático foram publicados no formato de uma artigo científico por Durelli *et al*. (2014a). Foram encontrados também alguns trabalhos que realizaram extensões no metamodelo KDM, contudo o mais relevante é o de Baresi e Miraz (2011).

Baresi e Miraz (2011) em seu artigo propõem uma extensão do KDM para dar suporte à modernização orientada a componentes (*Component-Oriented MOdernization* - COMO). COMO é um metamodelo que combina o KDM com os conceitos tradicionais da arquitetura de software, enriquecendo o KDM com os elementos de modelagem necessários para processar totalmente a arquitetura do sistema e explora essa arquitetura para correlacionar, com base em modelos, as

mudanças que poderão ser feitas no código fonte. Esse metamodelo permite acoplar componentes de softwares aos metamodelos em KDM, fazendo com que partes do sistema possam ser substituídas ou adicionadas ao novo código-fonte.

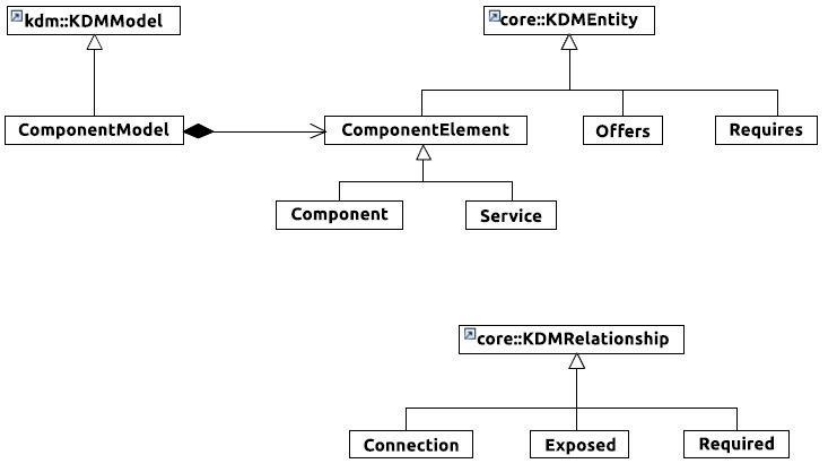


Figura 7.1 - Principais elementos do metamodelo COMO.

O metamodelo COMO está representado na Figura 7.1. As super classes *kdm::KDMModel*, *core::KDMEntity* e *core::KDMRelationship* fornecem a base para a criação deste metamodelo e é o elo que liga os metamodelos KDM e COMO. A classe *ComponentModel* armazena um modelo COMO e gera uma visão da arquitetura do sistema, dessa forma, todos os seus *ComponentElements* podem ser *Components* ou *Services* (softwares componentes ou serviços). *Components* geram os componentes de software e o *Services* expressam a ideia de funcionalidade que pode ser oferecida (*Offers*) ou requerida (*Requires*) por um componente de software. Existem também novos tipos de relacionamentos que permitem um componente oferecer (*Offers*) ou requisitar (*Requires*) algum serviço para poder interligar componentes (*Connection*), informando se eles estão disponíveis (*Exposed*) ou foram requisitados (*Required*) (Baresi e Miraz, 2011).

Diferente do que foi feito nesta pesquisa, Baresi e Miraz (2011) não utilizaram um perfil existente para realizar uma extensão do metamodelo KDM, ao invés disso, eles combinaram outro metamodelo ao KDM.

Fora do contexto do mapeamento sistemático, foi realizada uma busca informal pela internet (Google Acadêmico) com o objetivo de encontrar trabalhos que ainda não tivessem sido publicados e que fossem mais relacionados ao tema proposto nesta dissertação. O único trabalho encontrado foi o de Mirshams (2011).

Mirshams (2011) em sua dissertação de mestrado propõe uma abordagem para a conversão de código-fonte orientado a aspectos em modelos KDM. Sua abordagem consiste em quatro passos e estão representados na Figura 7.2. O primeiro consiste na separação entre os arquivos que representam aspectos e os arquivos que representam as classes orientadas a objetos (*dividing into classes/interfaces & aspects*). O segundo consiste na execução do compilador abc para a extração dos interesses transversais e gerar um arquivo *abc* chamado de *AspectInfo* (*Using abc to extract AspectJ elements*).

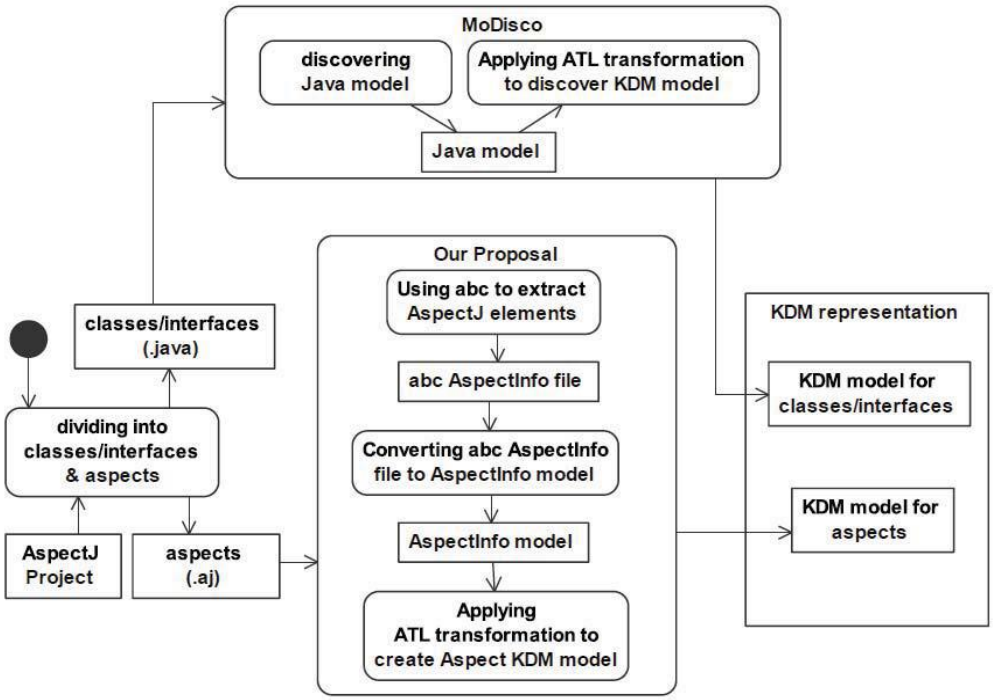


Figura 7.2 - Abordagem para a conversão de aspectos em modelos KDM (Shahshahani, 2011).

O terceiro passo é logo após a execução do compilador abc que por sua vez extrai as informações dos arquivos de aspectos (arquivo *AspectInfo*) e os transformam em um modelo de acordo com as especificações de um metamodelo denominado *AspectInfo*, que foi feito utilizando o *Eclipse Modeling Framework* (EMF) (*Converting abc AspectInfo file to AspectInfo model*). No quarto e último passo são aplicadas as transformações ATL[5] no modelo *AspectInfo* para gerar uma representação KDM chamado *Aspect KDM model* (*Applying ATL transformation to create Aspect KDM model*). Para que isso fosse possível, foi necessário realizar uma

[5] A ATLAS Transformation Language (ATL) é uma linguagem de transformação desenvolvida como parte da plataforma ATLAS Model Management Architecture (AMMA). O metamodelo ATL é compatível com XML Metadata Interchange (XMI).

extensão do KDM para representar as abstrações orientadas a aspectos (Mirshams, 2011).

Existem três principais diferenças entre o trabalho de Mirshams (2011) e o aqui apresentado. A primeira é que enquanto Mirshams baseou sua extensão em um perfil UML para aspectos mais genérico, as extensão aqui criadas se basearam em um perfil bem conhecido para a programação orientada a aspectos. O perfil do Evermann (2007) engloba todos os conceitos OA presentes na linguagem de programação AspectJ e em outras linguagens menos conhecidas, como AspectC++ e AspectS.

A segunda diferença é o nível de abstração das extensões leve e pesada que aqui foram desenvolvidas. O modelo de aspectos usado por Mirshams contém muito menos elementos se comparado ao modelo proposto por Evermann. Isso significa dizer que ambas as extensões aqui realizadas são capazes de representar tanto níveis altos de abstrações (usando as metaclasses mais genéricas) quanto níveis mais baixos de abstrações (usando metaclasses mais específicas) de um sistema escrito no paradigma orientado a aspectos.

A Terceira diferença é que o trabalho de Mirshams é limitado a interesses transversais dinâmicos, uma vez que não existem elementos para representar declarações inter-tipo. Entretanto, apesar dessas diferenças, a principal similaridade entre os trabalhos é que foram utilizadas as mesmas metaclasses do KDM original para estender as novas metaclasses, mesmo o processo de seleção das metaclasses sendo diferentes em ambas as abordagens.

Outro trabalho relacionado não leva em consideração extensões no metamodelo KDM e sim realiza uma comparação entre extensões leve e pesada no metamodelo UML. Magableh *et al*. (2012) reuniu quatorze artigos que realizaram extensões leves e pesadas no metamodelo da UML e aplicou alguns critérios de comparação. Da mesma forma como foi feito aqui, Magableh *et al*. (2012) lista alguns critérios para avaliar os trabalhos escolhidos. Os critérios utilizados por Magableh *et al*. (2012) foram utilizados como inspiração para a elaboração dos critérios que foram apresentados no Capítulo 5 desta dissertação. Contudo, nem todos os critérios foram utilizados, uma vez que alguns deles eram muito específicos para UML ou consideravam se a ferramenta criada para dar suporte à extensão era completa o suficiente, o que não se aplicava aqui, uma vez que nenhuma ferramenta para tratamento das extensões foi desenvolvida ainda. Devido a essas condições,

foi necessário criar um novo conjunto de critérios para apoiar as discussões desta pesquisa.

7.4 Considerações Finais

Este capítulo mostrou os principais trabalhos relacionados que foram encontrados na literatura. Conclui-se que nenhum dos trabalhos discutidos apresentaram uma abordagem semelhante a que foi discutida nesta dissertação, assim, cada trabalho relacionado se referem a praticamente um capítulo da dissertação. O trabalho de Mirshams (2011) e o de Baresi e Miraz (2011) apresentam extensões no metamodelo KDM, sendo que o de Mirshams está diretamente relacionado a esta pesquisa, uma vez que nele é realizada uma extensão pesada para POA no metamodelo KDM. O trabalho de Magableh *et al*. (2012) realiza uma análise comparativa entre as extensões leve e pesada realizadas no metamodelo da UML. Uma vez que essa avaliação foi realizada em um metamodelo diferente, seu trabalho foi usado apenas como referência nesta dissertação.

No Capítulo 8 são apresentadas as considerações finais do trabalho que envolve as principais contribuições e limitações, lições aprendidas e trabalhos futuros.

Capítulo 8

CONCLUSÃO

8.1 Contribuições e Limitações

Os pontos mais importantes desta dissertação estão centrados em como desenvolver extensões no metamodelo KDM, listar as vantagens e desvantagens de usar cada um dos mecanismos de extensão e a análise experimental aqui discutida. No Capítulo 5 foram mostradas algumas diretrizes que podem ajudar engenheiros de software a desenvolverem extensões no metamodelo KDM. No Capítulo 6 foi apresentando um estudo de caso para as duas extensões que foram desenvolvidas para um mesmo sistema e, em seguida, foram mostrados um conjunto de critérios que podem ajudar a decidir qual mecanismo de extensão um engenheiro de software deve usar, e sem dúvidas, dependendo de sua necessidade e contexto. Já no Capítulo 7 é apresentada uma análise experimental das duas extensões que mostram a usabilidade delas em relação a um grupo de sujeitos.

Assim como na UML, também é possível realizar tanto extensões leves quanto pesadas no KDM por meio dos mecanismos de extensões. Extensões pesadas são baseadas em um metamodelo KDM modificado, incluindo novas metaclasses ou mudando as metaclasses existentes. Por outro lado, extensões leves (perfis) são baseados em um conjunto de estereótipos e Definição de Tags (*TagDefinitions*), que são basicamente “anotações” no modelo. Perfis são capazes de impor restrições nas metaclasses existentes, mas, respeitam o metamodelo, sem modificar a semântica original dos elementos. Uma das maiores vantagens em

utilizar perfis é que eles podem ser manipulados facilmente por ferramentas já existentes.

No geral, a desvantagem de realizar extensões pesadas é que ferramentas existentes não são compatíveis com o novo metamodelo. Entretanto, a única forma de garantir uma maior corretude em nível de modelo é usando extensões pesadas. Isso ocorre por que é possível relacionar os elementos do metamodelo pelos seus tipos e não apenas por seus nomes, o que geralmente ocorre em extensões leves. Usando extensões leves a corretude dos modelos deve ser feita por ferramentas.

Por meio do estudo de caso apresentado, é relativamente evidente que as extensões podem representar todos os elementos da POA. Contudo, como não foi realizado um estudo de caso amplamente completo para medir o quão confiável são as extensões para representar os conceitos da orientação a aspectos em outras linguagens de programação, tal como AspectC++, argumenta-se que isso é uma limitação das extensões apresentadas no Capítulo 5. Contudo, para reduzir essa limitação os elementos das linguagens de programação AspectC++ e AspectS foram analisados. Consequentemente, conclui-se que existem elementos o suficiente nas extensões que podem ser usados para representar código-fonte em ambas as linguagens orientadas a aspectos.

A principal contribuição do estudo de caso é provar que os conceitos da POA podem ser representados tanto com a extensão leve quanto a pesada. Dessa forma, conclui-se que mesmo se esse tipo de modernização for feita em outro sistema usando outro FT com um interesse transversal diferente, os mesmo resultados serão obtidos. Para provar essa teoria, como trabalho futuro, pode-se realizar novas modernizações orientadas a aspectos em outros sistemas com FTs diferentes.

8.2 Lições aprendidas

As maiores dificuldades desta pesquisa foram encontrar trabalhos que dessem apoio às ideias aqui discutidas, como por exemplo, diretrizes de como realizar extensões no metamodelo KDM, critérios de comparação entre as duas extensões e trabalhos que utilizaram POA no metamodelo KDM.

Ao conduzir esta pesquisa, notou-se que o poder da modernização dirigida a modelos é fortemente influenciada pela capacidade de representar conceitos específicos de uma forma apropriada. Notou-se também que o KDM é um metamodelo poderoso que pode ser adaptado a quase todos os tipos de contexto, assim, dependerá apenas do engenheiro de software criar solução que melhor servir a seu propósito.

8.3 Trabalhos Futuros

Ao conduzir o estudo de caso usando a programação orientada a aspectos por meio de frameworks transversais, apresentado no Capítulo 5, notou-se que as extensões aqui apresentadas seriam mais úteis e mais expressivas se os metaelementos que representam as características de FTs também fossem inclusos no metamodelo. Por exemplo: *hot spots*, *frozen spots*, métodos gancho e outras características de FTs. Dada esta lacuna pretende-se implementar essas modificações em um trabalho futuro (Camargo e Masiero 2008).

Como nesta pesquisa está sendo proposta uma comparação qualitativa preliminar, pode ser que o conjunto de critérios aqui apresentados possam não ser suficientes para comparar as extensões. Visando solucionar esse problema pretende-se pesquisar na literatura por mais critérios que possam ser aplicados no contexto desta dissertação para garantir uma comparação mais justa.

A análise experimental aqui realizada levou em consideração o desenvolvimento e manutenção de modelos em KDM. Essas atividades foram realizadas em uma IDE (Eclipse) sem o auxílio de uma ferramenta que abstraísse e simplificasse o processo de codificação. Dessa forma, vê-se a oportunidade de desenvolver como trabalho futuro uma ferramenta que permita a instanciação de elementos do KDM-AO (Leve ou Pesada) e assim facilitar a criação de refatorações no metamodelo KDM. Uma vez desenvolvida essa ferramenta seria possível também desenvolver outro experimento que levasse em consideração sua usabilidade no contexto de empresas reais. Esse experimento seria interessante para avaliar até que ponto o processo de modernização proposto pela ADM é viável para empresas e sistemas reais.

Outro trabalho futuro, pretende-se conduzir outros estudos de casos usando as linguagens de programação AspectC++ e AspectS para testar as extensões do KDM-AO aqui propostas e assim avaliar a questão da independência de plataforma.

8.4 Publicações

Durante o desenvolvimento deste projeto os resultados obtidos foram divulgados nos seguintes trabalhos:

- SANTOS, B. M. *et al.* KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel. In: 28º Simpósio Brasileiro de Engenharia de Software (SBES), 2014a, Maceió, Brazil: IEEE. p. 1-10.

- SANTOS, B. M. *et al*. Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization. In: 11th Workshop on Software Modularity (WMod), 2014b, Maceió, Brazil. p. 1-12.

- DURELLI, R. S. *et al*. A Mapping Study on Architecture-Driven Modernization. In: 15th IEEE International Conference on Information Reuse and Integration, 2014, San Francisco, CA, USA. p 1-8.

REFERÊNCIAS

AMROUNE, M. *et al*. An UML profile to model Aspects in AspeCiS approach, In: IEEE Second International Workshop on Advanced Information Systems for Enterprises, 2012, Constantine, Algeria. p. 1-6.

BARESI, L. e MIRAZ, M. A Component-oriented Metamodel for the Modernization of Software Applications. In: 16th IEEE International Conference on Engineering of Complex Computer Systems, 2011, Las Vegas, USA. p. 1-9.

BARRA, E., GENOVA, G. e LLORENS, J. An approach to aspect modelling with UML 2.0, In: Proceedings of the 5th International Workshop on Aspect-Oriented Modelling Workshop, 2004, Lisbon, Portugal. p. 1-7.

BASCH, M. e SANCHEZ, A. Incorporating aspects into the UML, In: International Conference on Aspect-Oriented Software Development, 2003, Boston, Massachusetts. p. 1-5.

BRUNELIERE, H. *et al.* MoDisco: A generic and extensible framework for model driven reverse engineering. In: IEEE/ACM international conference on Automated software engineering, 2010, ACM New York, NY, USA. p. 173-174.

CAMARGO, V. V., *et al*. Projeto Orientado a Aspectos do Padrão Camada de Persistência, In: 17º Simpósio Brasileiro de Engenharia de Software (SBES), 2003, Manaus, Brazil, pp. 114-129.

CAMARGO, V. V. de; MASIERO, P. C. Frameworks Orientados a Aspectos, In: XIX Simpósio Brasileiro de Engenharia de Software, 2005, Uberlândia, Brazil. p. 200-216.

CAMARGO, V. V. e MASIERO, P. C. An Approach to Design Crosscutting Framework Families, In: 7th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2008), 2008, Brussels, Belgium. p. 1-6.

COUTO, C.F.M., *et al*. Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência, In: 2º Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), 2005, Uberlândia, Brasil. p. 1-8.

DURELLI, R. S. *et al*. A Mapping Study on Architecture-Driven Modernization. In: 15th IEEE International Conference on Information Reuse and Integration, 2014a, San Francisco, CA, USA. p 1-8.

DURELLI, R. S. *et al*. Towards a Refactoring Catalogue for Knowledge Discovery Metamodel. In: 15th IEEE International Conference on Information Reuse and Integration, 2014b, San Francisco, CA, USA. p 1-8.

ELRAD, T., FILMAN R. e BADER A. Aspect-Oriented Programming, In: Communications of the ACM, 2001, New York, USA. p. 29-32.

EVERMANN, J. An overview and an empirical evaluation of UML: an UML profile for aspect-oriented frameworks, In: Workshop on Aspect-Oriented Modeling, Vancouver, British Columbia, Canada. p. 1-7.

FUENTES, L. e SANCHEZ, P. Elaborating UML 2.0 profiles for AO design, In: Proceedings of the 8th International Workshop on Aspect-Oriented Modeling, 2006, Bonn, Germany. p. 1-7.

GRUNDY, J. e PATEL, R. Developing software components with the UML, Enterprise Java Beans and aspects, In: Proceedings of 13th Australian Software Engineering Conference (ASWEC 2001), 2001, Canberra, Australia. p. 1-10.

ISO/IEC. ISO/IEC DIS 19506. Information technology - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) http://www.iso.org/iso/catalogue_detail.1128htm?csnumber=32625, ISO/IEC. p. 331. 2012.

KANDÉ, M., KIENZLE, J. e STROHMEIER, A. From AOP to UML - a bottom-up approach, In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, 2002, Enschede, The Netherlands. p. 1-6.

KICZALES, G. e MEZINI, M. Aspect-Oriented Programming and Modular Reasoning. In: Proceedings 27th International Conference on Software Engineering (ICSE 2005), 2005, St. Louis, Missouri, USA. p. 1-10.

KICZALES, G. *et al*. Aspect-Oriented Programming. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP), 1997, Jyväskylä, Finland. p. 220-242.

KISELEV, I. Aspect-Oriented Programming with AspectJ. 1st ed. Indiana: SAMS, 2003

LADDAD, R. AspectJ in Action: Practical Aspect-Oriented Programming. 1st ed. Greenwich: Manning Publications. 2003.

MAGABLEH, A., SHUKUR, Z. e ALI, N. Heavy-Weight and Light-weight UML Modelling Extensions of Aspect-Orientation in the Early Stage of Software Development. Journal of Applied Sciences, 2012, DOI: 10.3923/jas.2012.2195.2201.

MIRSHAMS, P. S. Extending the Knowledge Discovery Metamodel to Support Aspect-Oriented Programming, p. 79. Dissertation (Master in Applied Science in Software Engineering) – Computer Science Department and Software Engineering, University of Montreal, Quebec, Canada, 2011, unpublished.

NORMANTAS, K., *et al*. An Overview of the Knowledge Discovery Meta-Model, In: Proceedings of the 13th International Conference on Computer Systems and Technologies (CompSysTech'12), 2012, Ruse, Bulgaria. p. 52-57.

OBJECT MANAGEMENT GROUP. (2014) OMG Specifications, April 2014. Documents omg/ http://www.omg.org/spec/.

PAWLAK, R., *et al*. A UML notation for aspect-oriented software design, In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development, 2002, Enschede, The Netherlands. p. 1-7.

PÉREZ-CASTILLO, R., GUZMÁN, I. G. e PIATTINI, M. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Journal of Computer Standards & Interfaces, 2011, DOI: 10.1016/j.csi.2011.02.007.

RASHID, A., e CHITCHYAN, R. Persistence as an Aspect, In: 2nd International Conference on Aspect-Oriented Software Development (AOSD), 2003, Boston, USA. p. 120-129.

RAUSCH, A., RUMPE, B. e HOOGENDOORN, L. Aspect-Oriented Framework Modeling, In: 4th AOSD Modeling with UML Workshop, 2003, San Francisco, USA. p. 1-7.

SADOVYKH, A., *et al*. Architecture Driven Modernization in Practice – Study Results, In: 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2009), 2009, Potsdam, Germany. p. 50-57.

SANTIBÁÑEZ, D. S. M. *et al*. A Combined Approach for Concern Identification in KDM models, In: 7th Latin American Workshop on Aspect-Oriented Software Development (LA-WASP), 2013, Brasília, Brasil. p. 1-6.

SANTOS, B. M. *et al*. Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization. In: 11th Workshop on Software Modularity (WMod), 2014b, Maceió, Brazil. p. 1-12.

SANTOS, B. M. *et al.* KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel. In: 28º Simpósio Brasileiro de Engenharia de Software (SBES), 2014a, Maceió, Brazil: IEEE. p. 1-10.

SOARES, S., LAUREANO, E. e BORBA, P. Implementing Distribution and Persistence Aspects with AspectJ, In: 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002, Seattle, USA. p. 174-190.

ULRICH, W. e NEWCOMB, P. Information Systems Transformation: Architecture-Driven Modernization Case Studies. 1st ed. Burlington, USA: Elsevier. 2010.

VISAGGIO, G. Ageing of a data-intensive legacy system: symptoms and remedies. Journal of Software Maintenance, 2001, DOI: 10.1002/smr.234.

WOHLIN, C., *et al.* Experimentation in software engineering: an introduction. 1st ed. Norwell, MA, USA: Kluwer Academic Publishers. 2000.

Apêndice A

FORMULÁRIOS DE CARACTERIZAÇÃO DOS PARTICIPANTES E DE CONSENTIMENTO

Form to characterize subjects

Please read it and answer it as honest as possible

This form aims to characterise your academic, personal and professional experience regarding to Computer Science. Please, try to answer ALL the questions as faithfully as possible. Your name and other informations related to this form will not be disclosed under any circumstances.

1. Subject

1. Name: _____
2. Age:_____

2. Academic Education

- ( ) Master
- ( ) PhD

Year of entry:_____ Month/Year of graduation (or forecast finish):___/_____

3. Professional Experience

• Check the box that best reflects your current level of experience with the technologies listed below, based on the 5-point scale:

• 0 = no experience
• 1 = studied either at class or in book
• 2 = practiced in projects at class
• 3 = used in personal projects
• 4 = use in most projects I conducted

Tecnologies	Points				
Java Programming Language	0	1	2	3	4
IDE Eclipse	0	1	2	3	4
Eclipse Modeling Framework	0	1	2	3	4
Lightweight extension mechanism	0	1	2	3	4
Heavyweight extension mechanism	0	1	2	3	4
Knowledge Discovery Metamodel (KDM)	0	1	2	3	4
Object-oriented programming	0	1	2	3	4
MoDisco	0	1	2	3	4

4. What is your ability to …

a. … develop software using Aspect-Oriented Programming

( ) specialist ( ) advanced ( ) medium ( ) basic ( ) none

b. … devise software using concepts of Model-Driven Development

( ) specialist ( ) advanced ( ) medium ( ) basic ( ) none

5. Check the option below that best reflects your current level of experience with related items in the table, considering the 4-point scale:

• 0 = I'm not familiar with this subject. I never have done this.
• 1 = I have already either read or study about it. I know the concepts and/or techniques.
• 2 = I use this sometimes in personal projects or in the industry, but I'm not consider myself an specialist in the subject
• 3 = I'm very familiar with this subject. I would feel comfortable doing this.

Tecnologies	Points			
Experience with the instantiation of KDM	0	1	2	3
Experience in programmatically instantiate the KDM	0	1	2	3
Ability with java programming	0	1	2	3

Consent Form

1. Experiment

This experiment aims to examine whether the productivity of the application engineer is different when instantiating heavyweight/lightweight extension based on the KDM metamodel. Also, we want to investigate which of the extensions leads the modernization engineers to make more mistakes.

2. Age

I declare to be greater than eighteen (18) years of age and agree to participate in the experiment conducted at the Federal University of São Carlos (UFSCar).

3. Procedure

This experiment will take place in a single session, which will include the instantiation of heavyweight/lightweight extension based on the KDM metamodel. The instantiation of the extensions will be made following a few guidelines as determined by the experimenter. I understand that, once the experiment has been completed, the work that I developed as well as the data collected will be studied in order to analyze the implementation of procedures and techniques proposed.

4. Confidentiality

I understand that all information collected in this experiment is confidential, and my name or any other means of identification will not be disclosed. Likewise, I agree not to communicate my findings to other participants until you have finished the experiment, as well as maintaining confidentiality of documents submitted and techniques that are part of the experiment.

5. Benefits and freedom to dropout

I understand that the benefits I receive by taking part of this experiment is limited to the learning of the material that is distributed. I understand that I am free to make inquiries at any time. I also understand that I am free to request that any information related to my person is not included in the experiment, or communicate my dropping out of this experiment. I understand that I participate freely with the only purpose of contributing to the advancement and development of techniques and processes for Software Engineering.

6. The Researchers Responsible

Bruno Marinho	Programa de Pós-Graduação em Ciência da Computação – PPG-CC/DC/UFSCar
Raphael Honda	Programa de Pós-Graduação em Ciência da Computação – PPG-CC/DC/UFSCar
Rafael S. Durelli	ICMC - Instituto de Ciências Matemáticas e de Computação/USP
Valter V. de Camargo	Programa de Pós-Graduação em Ciência da Computação – PPG-CC/DC/UFSCar

7. Professor

Prof. Dr. Valter Vieira de Camargo

By completing and signing this form, I give full notice and consent to the terms stated above.

Name (printscript): _____

Apêndice B

DESCRIÇÃO DAS ATIVIDADES DO EXPERIMENTO REAL E FORMULÁRIO DE COLETA DE DADOS

Lightweight and Heavyweight extensions: Creating KDM-AO instances Experiment

Name:_____

Group:_____Extension:_____Phase: _____

Activity 1

-Create three CrosscuttingConcerns, following the source code below.

```
package security;
package logging;
package persistence;
```

Time to perform this activity: 6 minutes

Start time: ___h____min

End time: ___h____min

Activity 1 – Observations (Only after you have finished this activity):

_____

_____

_____

Activity 2

-Create the three Aspects presented below and link them to their respective crosscutting concern created earlier.

```
public aspect ObjectsController {}
public aspect SecurityComposition{}
public aspect PartialAwareness {}
```

Time to perform this activity: 6 minutes

Start time: ___h____min

End time: ___h____min

Activity 2 – Observations (Only after you have finished this activity):

Activity 3

-Create these PointCuts, following the source code below.

```
public pointcut affectedClasses(): execution (move());
public pointcut saveOrUpdate(): withincode (getAccount());
public pointcut excludedJoinPoints(): call (setName());
```

Time to perform this activity: 12 minutes

Start time: ___h____min

End time: ___h____min

Activity 3 – Observations (Only after you have finished this activity):

Activity 4

-Create two PointCuts, following the source code below.

```
public pointcut dirtyObjects():
        execution (save()) || call (delete());
public pointcut cleanObjects():
        call (update()) && execution(copy());
```

Time to perform this activity: 25 minutes

Start time: ___h____min

End time: ___h____min

Activity 4 – Observations (Only after you have finished this activity):

Activity 5

Create three Advices, following the source code below.

```
after(): affectedClasses(){ //source code }
before(): saveOrUpdate(){ //source code }
around(): excludedJoinPoints(){ //source code }
```

Time to perform this activity: 9 minutes

Start time: ___h____min

End time: ___h____min

Time to perform this activity: 9 minutes

Activity 5 – Observations (Only after you have finished this activity):

Activity 6

Create the following Inter-Type Declarations, following the source code below.

```
public boolean PersistenceRoot. tableName;
public int PersistenceRoot. keyName;
public String PersistenceRoot. numberOfAttributes;
public int PersistenceRoot. setTableName() {}
public void PersistenceRoot. setColValues() {}
```

Time to perform this activity: 9 minutes

Start time: ___h____min

End time: ___h____min

Activity 6 – Observations (Only after you have finished this activity):

Activity 7

Now you have to maintain the source code.

Please, open the java class maintenance_experiment to perform this activity.

-Add the following properties to Aspect1:

- "isAbstract = false";
- "PrecededBy = Aspect1";
- "Precedes = Aspect1".

Time to perform this activity: 9 minutes

Start time: ___h____min

End time: ___h____min

Activity 7 – Observations (Only after you have finished this activity):

Activity 8

Continuing in java class maintenance_experiment, perform perform the following maintenance.

-Transform the following pointcut as follows.

As is

```
public pointcut pointcut4():  execution (method1());
```

To be

```
public pointcut pointcut4():
        execution (method1()) || call (method2());
```

Time to perform this activity: 9 minutes

Start time: ___h____min

End time: ___h____min

Activity 8 – Observations (Only after you have finished this activity):

Apêndice C

MANUAL DE UTILIZAÇÃO DAS EXTENSÕES, TABELA DE MAPEAMENTO DOS ELEMENTOS E DIAGRAMA DE CLASSES DAS EXTENSÕES

Heavyweight Guide

Creating Heavyweight instances

1) How to create CrossCuttingConcern

To create a package like this "`Package Persistence;`" you have to create this following source code:

```
CrossCuttingConcern PersistenceConcern =
CodeFactory.eINSTANCE.createCrossCuttingConcern();
PersistenceConcern.setName("Persistence");
```

2) How to create Aspect

To create an aspect like this "`public aspect Logging{}`" you have to create this following source code:

```
AspectUnit Logging = CodeFactory.eINSTANCE.createAspectUnit();
Logging.setName("Logging");
```

If your aspect is inserted in a package you have to bound it the corresponding package, like this:

```
PersistenceConcern.getCodeElement().add(Logging);
```

If you want to add some of its properties, you just have choose the property and set the value, as you can see in the source code below:

```
Logging.setIsPrivileged(true);
```

3) How to create Pointcut with one Joinpoint

To create a pointcut like this "`public pointcut figure(): call (move());`"you have to create this following source code:

```
CallPointCutUnit myPointcut =
CodeFactory.eINSTANCE.createCallPointCutUnit();
myPointcut.setName("figure");
myPointcut.getOperation().add(move());
```

4) How to create Pointcut with more than one Joinpoint

To create a pointcut like this:

"`public pointcut move(): execution (getID()) || call (getName());`" You have to create this following source code:

```
CompositePointCutUnit myComposite =
CodeFactory.eINSTANCE.createCompositePointCutUnit();
myComposite.setName("move");
myComposite.setCompositeType(PointCutCompositionType.OR);
```

First you have to create the "*CompositePointCutUnit*" and then you have to set the property "*name*" and "*compositionType*" . Notice that you have to set the name ONLY in the "*CompositePointCutUnit*" element.

The next step is to create the joinpoints and for this you have to create the respective element that you want to represent. In this example we have two joinpoints, so the following source code represents these two joinpoints.

```
ExecutionPointCutUnit myExecution =
CodeFactory.eINSTANCE.createExecutionPointCutUnit();
myExecution.getOperation().add(getID);
myExecution.getComposite().add(myComposite);
CallPointCutUnit myCall =
CodeFactory.eINSTANCE.createCallPointCutUnit();
myCall.getOperation().add(getName);
myCall.getComposite().add(myComposite);
```

To set correctly a joinpoint you have to set the properties "*Operation*" and "*Composite*", the "*Operation"* is the method affected by the joinpoint and the "*Composite"* is the "*CompositePointCutUnit"* that the joinpoints belongs.

5) How to create Advice

To create an advice like this: "`before() : move() {// Some JAVA souce code}`" You have to create this following source code:

```
AdviceUnit myAdvice =
CodeFactory.eINSTANCE.createAdviceUnit();
myAdvice.setPointCut(move);
myAdvice.setAdviceExecution(AdviceExecutionType.BEFOREADVICE);
```

The main properties that you have to set while creating an Advice are "*pointCut"* (which specifies the pointcut that it belongs) and the "*adviceExecutionType"* (that specifies the behavior of the advice).

6) How to create Inter-type declaration

To create an advice like this:

"`public PersistentRoot.name;`

`public void PersistentRoot.move();`" You have to create this following source code:

```
StaticCrossCuttingFeature myFeature =
CodeFactory.eINSTANCE.createStaticCrossCuttingFeature();
myFeature.getOnType().add(PersistentRoot);
StorableUnit myStorable =
CodeFactory.eINSTANCE.createStorableUnit();
myStorable.setName("name");
myStorable.getCodeElement().add(myFeature);
MethodUnit myMethod =
CodeFactory.eINSTANCE.createMethodUnit();
myMethod.setName("move()");
myMethod.getCodeElement().add(myFeature);
```

Notice that you have to create a "*StaticCrossCuttingFeature"* element and set the "*OnType*" property. After creating the "*StorableUnit"* and the "*MethodUnit"* you have to bound them into the "*StaticCrossCuttingFeature"*.

Lightweight Guide

Creating Lightweight instances

1) How to create CrossCuttingConcern

To create a package like this "`Package Persistence;`" you have to create this following source code:

```
Package myCrosscuttingConcern =
CodeFactory.eINSTANCE.createPackage();
myCrosscuttingConcern.setName("Persistence");
myCrosscuttingConcern.getStereotype().add(Profiles.crossCutting
Concern);
```

2) How to create Aspect

To create an aspect like this "`public aspect connectionComposition{}`" you have to create this following source code:

```
ClassUnit MyAspect = CodeFactory.eINSTANCE.createClassUnit();
MyAspect.setName("connectionComposition");
MyAspect.getStereotype().add(Profiles.aspectUnit);
```

If your aspect is inserted in a package you have to bound it the corresponding package, like this:

```
myCrosscuttingConcern.getCodeElement().add(MyAspect);
```

If you want to add some of its properties, you have to create a "*TaggedValue"* as you can see in the source code below:

```
TaggedValue myIsProvileged =
KdmFactory.eINSTANCE.createTaggedValue();
myIsProvileged.setTag(Profiles.isPrivileged);
myIsProvileged.setValue("True");
MyAspect.getTaggedValue().add(myIsProvileged);
```

Notice that when you create a "*TaggedValue"* you have to bound it to the corresponding element. In this case you have to bound the "*isPrivileged"* Tag to the "*ClassUnit myAspect"*.

3) How to create Pointcut with one Joinpoint

To create a pointcut like this "**public pointcut openConnection(): execution (main());**"you have to create this following source code:

```
MemberUnit myMember = CodeFactory.eINSTANCE.createMemberUnit();
myMember.getStereotype().add(Profiles.executionPointCutUnit);
myMember.setName("openConnection");
TaggedValue myOperation = KdmFactory.eINSTANCE.createTaggedValue();
myOperation.setTag(Profiles.operation);
myOperation.setValue("main()");
myMember.getTaggedValue().add(myOperation);
```

Notice that you have to create first an instance of *MemberUnit* element, and then apply the stereotype. In this kind of pointcut with just one jointpoint you have to set the name in the memberUnit that represents your joinpoint.

The second observation is that to set the "*operation"* of the pointcut you have to create a "*TaggedValue"*, apply the "*TagDefinition operation"* and then set the value. In order to complete the creation of this instance you have to bound the "*TaggedValue"* to the "*MemberUnit"* created, see the last source code line.

4) How to create Pointcut with more than one Joinpoint

To create a pointcut like this:

"**public pointcut openConnection(): execution (main()) || call (main());**" You have to create this following source code:

```
MemberUnit myCompositionPointcut =
CodeFactory.eINSTANCE.createMemberUnit();
myCompositionPointcut.getStereotype().
                        add(Profiles.compositePointCutUnit);
myCompositionPointcut.setName("openConnection");
TaggedValue myComposition =
KdmFactory.eINSTANCE.createTaggedValue();
myComposition.setTag(Profiles.compositionType);
myComposition.setValue("OR");

myCompositionPointcut.getTaggedValue().add(myComposition);
```

First you have to create a "*MemberUnit"* and then apply the "*compositePointCutUnit"* Stereotype. You have to set the property "*compositionType"* too, so you just have to create a "*TaggedValue*" and then you have to bound this "*TaggedValue"* to the "*MemberUnit"* stereotyped with "*CompositePointCutUnit"*.

Remember to bound the "*TaggedValue"* to the respective "*MemberUnit"*.
Notice that you have to set the name ONLY in the "*MemberUnit"* stereotyped with "*CompositePointCutUnit"*.

The next step is to create the joinpoints and for this you have to create a "*MemberUnit"* for each joinpoint specified on the pointcut. In this example we have two joinpoints, so the following source code represents these two joinpoints.

```
MemberUnit myExecutionPointcut = CodeFactory.eINSTANCE.createMemberUnit();
myExecutionPointcut.getStereotype().add(Profiles.executionPointCutUnit);
     TaggedValue myExecutionOperation = KdmFactory.eINSTANCE.createTaggedValue();
     myExecutionOperation.setTag(Profiles.operation);
     myExecutionOperation.setValue("main()");
          TaggedValue myTagComposite1 = KdmFactory.eINSTANCE.createTaggedValue();
          myTagComposite1.setTag(Profiles.composite);
          myTagComposite1.setValue("myCompositePointCut");
myExecutionPointcut.getTaggedValue().add(myExecutionOperation);
myExecutionPointcut.getTaggedValue().add(myTagComposite1);

MemberUnit myCallPointcut = CodeFactory.eINSTANCE.createMemberUnit();
myCallPointcut.getStereotype().add(Profiles.callPointCutUnit);
    TaggedValue myCallOperation = KdmFactory.eINSTANCE.createTaggedValue();
    myCallOperation.setTag(Profiles.operation);
    myCallOperation.setValue("main()");
          TaggedValue myTagComposite2 =
KdmFactory.eINSTANCE.createTaggedValue();
          myTagComposite2.setTag(Profiles.composite);
          myTagComposite2.setValue("myCompositePointCut");
myCallPointcut.getTaggedValue().add(myCallOperation);
myCallPointcut.getTaggedValue().add(myTagComposite2);
```

The first "*MemberUnit"* is stereotyped with "*callPointCutUnit"* and it has two "*TaggedValues"* bounded that represents the properties "*Operation*" and "*Composite*". The second "*MemberUnit"* is stereotyped with "*callPointCutUnit"* and it has two "*TaggedValues"* bounded that represents the properties "*Operation*" and "*Composite*", like in the first one.

5) How to create Advice

To create an advice like this: "`around() : move() {// Some JAVA souce code}`" You have to create this following source code:

```
ControlElement myAdvice =
CodeFactory.eINSTANCE.createControlElement();
myAdvice.getStereotype().add(Profiles.adviceUnit);
```

```
TaggedValue myAdviceofPointCut =
KdmFactory.eINSTANCE.createTaggedValue();
myAdviceofPointCut.setTag(Profiles.pointCut);
myAdviceofPointCut.setValue("move()");
TaggedValue myAdviceExecution =
KdmFactory.eINSTANCE.createTaggedValue();
myAdviceExecution.setTag(Profiles.adviceExecutionType);
myAdviceExecution.setValue("AROUNDADVICE");
myAdvice.getTaggedValue().add(myAdviceofPointCut);
myAdvice.getTaggedValue().add(myAdviceExecution);
```

The main properties that you have to set while creating an Advice are "*pointCut"* (which specifies the pointcut that it belongs) and the "*adviceExecutionType"* (that specifies the behavior of the advice).

You can set this properties by creating "*TaggedValues"* and applying the "*TagDefinitions" "pointCut"* and "*adviceExecutionType"* and then bound them into the "*ControlElement"* that represents the Advice.

6) How to create Inter-type declaration

To create an advice like this:

"`public PersistentRoot.name;`

`public void PersistentRoot.move();`" You have to create this following source code:

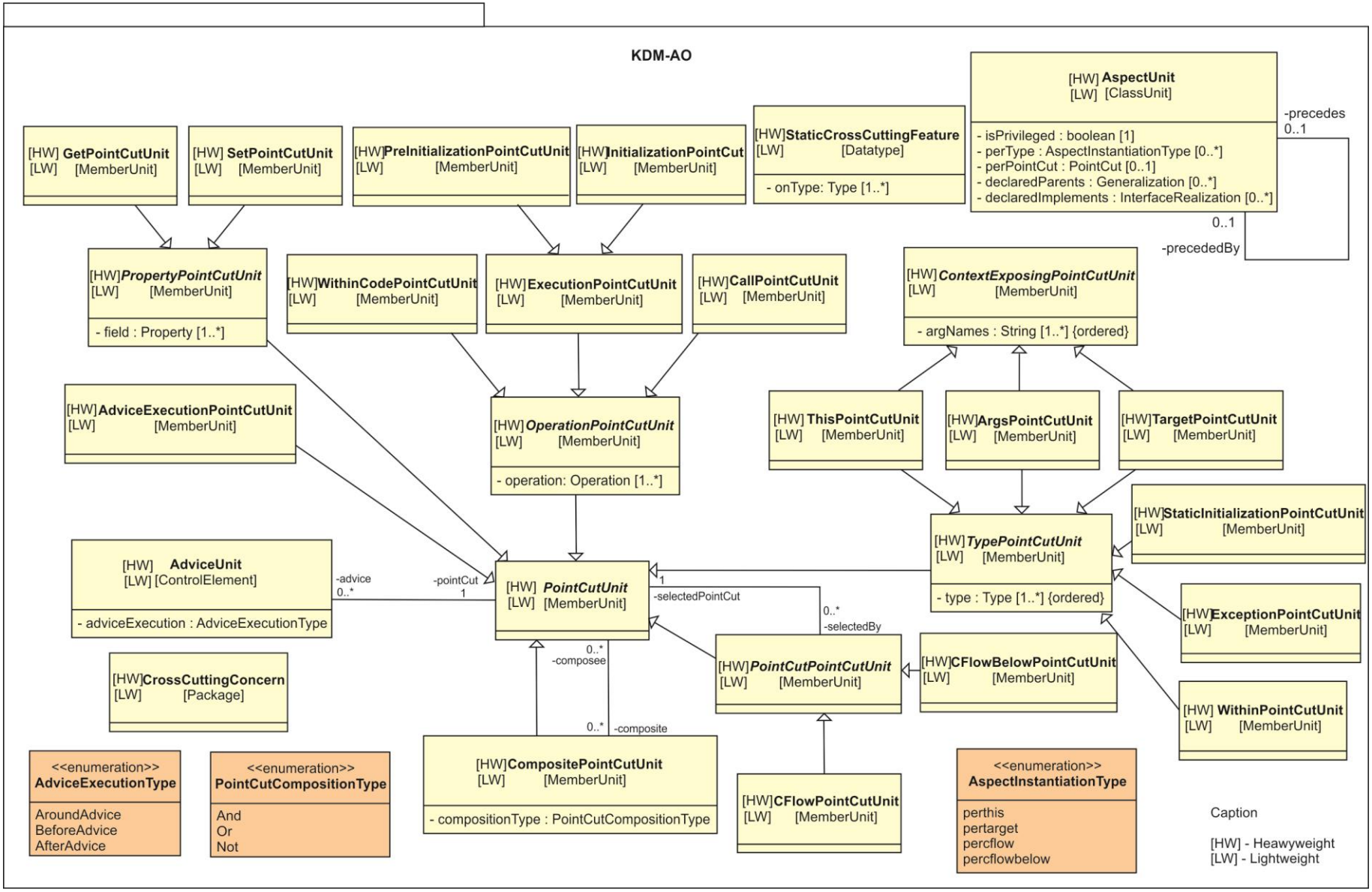
```
Datatype myFeature = CodeFactory.eINSTANCE.createDatatype();
myFeature.getStereotype().add(Profiles.staticCrossCuttingFeature);
TaggedValue myOnType = KdmFactory.eINSTANCE.createTaggedValue();
myOnType.setTag(Profiles.onType);
myOnType.setValue("PersistenceRoot");
myFeature.getTaggedValue().add(myOnType);
StorableUnit myStorable = CodeFactory.eINSTANCE.createStorableUnit();
myStorable.setName("name");
myStorable.getCodeElement().add(myFeature);
MethodUnit myMethod = CodeFactory.eINSTANCE.createMethodUnit();
myMethod.setName("move()");
myMethod.getCodeElement().add(myFeature);
```

Notice that you have to create a "*Datatype"* element and apply the stereotype "staticCrossCuttingFeature". You can set the "*OnType*" property by creating a "*TaggedValue"*. Don't forget to bound the TaggedValue to the Datatype element.

The attribute and the method have to be bounded to the "*Datatype"* too.

AspectJ Profile Elements	KDM-AO elements (Heawyweight)
Aspect	AspectUnit
PointCut	PointCutUnit
CompositePointCut	CompositePointCutUnit
OperationPointCut	OperationPointCutUnit
WithinCodePointCut	WithinCodePointCutUnit
ExecutionPointCut	ExecutionPointCutUnit
CallPointCut	CallPointCutUnit
PreInitializationPointCut	PreInitializationPointCutUnit
InitializationPointCut	InitializationPointCutUnit
PropertyPointCut	PropertyPointCutUnit
GetPointCut	GetPointCutUnit
SetPointCut	SetPointCutUnit
AdviceExecutionPointCut	AdviceExecutionPointCutUnit
PointCutPointCut	PointCutPointCutUnit
CFlowPointCut	CFlowPointCutUnit
CFlowBelowPointCut	CFlowBelowPointCutUnit
TypePointCut	TypePointCutUnit
WithinPointCut	WithinPointCutUnit
ExceptionPointCut	ExceptionPointCutUnit
StaticInitializationPointCut	StaticInitializationPointCutUnit
TargetPointCut	TargetPointCutUnit
ArgsPointCut	ArgsPointCutUnit
ThisPointCut	ThisPointCutUnit
ContextExposingPointCuit	ContextExposingPointCuitUnit
Package	CrossCuttingConcern
AdviceExecutionPointCut	setAdviceExecution
PointCutCompositionType	setPointCutCompositionType
AspectInstantiationType	setAspectInstantiationType
Intertype Declaration	StaticCrossCuttingFeature
Advice	AdviceUnit

AspectJ Profile Elements	KDM Elements	Stereotype (Lightweigth)
Aspect	ClassUnit	aspectUnit
PointCut	MemberUnit	pointCutUnit
CompositePointCut		compositePointCutUnit
OperationPointCut		operationPointCutUnit
WithinCodePointCut		withinCodePointCutUnit
ExecutionPointCut		executionPointCutUnit
CallPointCut		callPointCutUnit
PreInitializationPointCut		preInitializationPointCutUnit
InitializationPointCut		initializationPointCutUnit
PropertyPointCut		propertyPointCutUnit
GetPointCut		getPointCutUnit
SetPointCut		setPointCutUnit
AdviceExecutionPointCut		adviceExecutionPointCutUnit
PointCutPointCut		pointCutPointCutUnit
CFlowPointCut		cFlowPointCutUnit
CFlowBelowPointCut		cFlowBelowPointCutUnit
TypePointCut		typePointCutUnit
WithinPointCut		withinPointCutUnit
ExceptionPointCut		exceptionPointCutUnit
StaticInitializationPointCut		staticInitializationPointCutUnit
TargetPointCut		targetPointCutUnit
ArgsPointCut		argsPointCutUnit
ThisPointCut		thisPointCutUnit
ContextExposingPointCuit		contextExposingPointCuitUnit
CrossCuttingConcern	Package	crossCuttingConcern
AdviceExecutionType	TaggedValue	setTag(Profiles.adviceExecutionType)
PointCutCompositionType	TaggedValue	setTag(Profiles.pointCutCompositionType)
AspectInstantiationType	TaggedValue	setTag(Profiles.aspectInstantiationType)
Intertype Declaration	Datatype	staticCrossCuttingFeature
Advice	ControlElement	adviceUnit


KDM-AO
[HW] GetPointCutUnit
[LW] [MemberUnit]
[HW] SetPointCutUnit
[LW] [MemberUnit]
[HW]PreInitializationPointCutUnit
[LW] [MemberUnit]
[HW]InitializationPointCut
[LW] [MemberUnit]
[HW]StaticCrossCuttingFeature
[LW] [Datatype]
- onType: Type [1..*]
[HW] AspectUnit
[LW] [ClassUnit]
- isPrivileged : boolean [1]
- perType : AspectInstantiationType [0..*]
- perPointCut : PointCut [0..1]
- declaredParents : Generalization [0..*]
- declaredImplements : InterfaceRealization [0..*]
-precedes
0..1
0..1
-precededBy
[HW]PropertyPointCutUnit
[LW] [MemberUnit]
- field : Property [1..*]
[HW]WithinCodePointCutUnit
[LW] [MemberUnit]
[HW]ExecutionPointCutUnit
[LW] [MemberUnit]
[HW]CallPointCutUnit
[LW] [MemberUnit]
[HW]ContextExposingPointCutUnit
[LW] [MemberUnit]
- argNames : String [1..*] {ordered}
[HW]AdviceExecutionPointCutUnit
[LW] [MemberUnit]
[HW]OperationPointCutUnit
[LW] [MemberUnit]
- operation: Operation [1..*]
[HW] ThisPointCutUnit
[LW] [MemberUnit]
[HW]ArgsPointCutUnit
[LW] [MemberUnit]
[HW]TargetPointCutUnit
[LW] [MemberUnit]
[HW]StaticInitializationPointCutUnit
[LW] [MemberUnit]
[HW] AdviceUnit
[LW] [ControlElement]
- adviceExecution : AdviceExecutionType
-advice
0..*
-pointCut
1
[HW] PointCutUnit
[LW] [MemberUnit]
1
-selectedPointCut
0..*
-selectedBy
[HW]TypePointCutUnit
[LW] [MemberUnit]
- type : Type [1..*] {ordered}
[HW]ExceptionPointCutUnit
[LW] [MemberUnit]
[HW]CrossCuttingConcern
[LW] [Package]
0..*
-composee
0..*
-composite
[HW]PointCutPointCutUnit
[LW] [MemberUnit]
[HW]CFlowBelowPointCutUnit
[LW] [MemberUnit]
[HW] WithinPointCutUnit
[LW] [MemberUnit]
<<enumeration>>
AdviceExecutionType
AroundAdvice
BeforeAdvice
AfterAdvice
<<enumeration>>
PointCutCompositionType
And
Or
Not
[HW]CompositePointCutUnit
[LW] [MemberUnit]
- compositionType : PointCutCompositionType
[HW]CFlowPointCutUnit
[LW] [MemberUnit]
<<enumeration>>
AspectInstantiationType
perthis
pertarget
percflow
percflowbelow
Caption
[HW] - Heawyweight
[LW] - Lightweight