Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura

Rafael Serapilha Durelli

Rafael Serapilha Durelli

Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
Maio de 2016

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

Durelli, Rafael Serapilha
D955u Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura / Rafael Serapilha Durelli; orientador Márcio Eduardo Delamaro. - São Carlos - SP, 2016.
239 p.

Tese (Doutorado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2016.

1. ADM. 2. KDM. 3. Refatoração. 4. Engenheiro de Modernização. 5. Desenvolvimento Dirigido por Modelos. 6. Experimento. I. Delamaro, Márcio Eduardo, orient. II. Título.

Rafael Serapilha Durelli

An approach to create, reuse and apply refactoring in the context of architecture driven modernization

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
May 2016

Sou grato a Deus por minha vida, família e amigos.
Em especial, aos pesquisadores do Instituto de Ciências Matemáticas e de Computação (ICMC)
e Departamento de Computação (DC) da Universidade Federal de São Carlos.

AGRADECIMENTOS

Meu primeiro agradecimento é a Deus, por Ele ter me proporcionado tudo para que trilhasse o caminho que trilhei até chegar onde estou agora. Por ter colocado as pessoas certas para me ajudar em minhas dificuldades, por ter me colocado obstáculos nas horas adequadas que me permitiram amadurecer para superar obstáculos maiores no futuro e por ter me iluminado no momento das decisões mais críticas que acabaram direcionando minha vida no futuro.

Ao Prof. Dr. Márcio Eduardo Delamaro, pela dedicação, ética, profissionalismo e amizade, meu eterno agradecimento saiba que o Sr. mudou a minha vida. Agradeço também ao meu amigo, Prof. Dr. Valter Vieira de Camargo. Embora, não tenha sido meu orientador formalmente, eu sou eternamente grato a suas orientações, dicas, postura e paciência - acredite apreendi muito com o Sr.; sempre irei me lembrar das nossas reuniões com carinho. Seguindo a mesma ideia de pessoas a quem devo minha trilha aqui em São Carlos, não poderia esquecer da Prof.ª Dr.ª Rosângela Ap. Dellosso Penteado, muito obrigado por tudo.

Também gostaria de agradecer a minha família em especial a minha mãe e avó. Também vai um agradecimento ao meu irmão por me mostrar o caminho por meio de bons exemplos.

Aos meus amigos dos Laboratório de Engenharia de Software (LabES) e AdvanSE, companheiros de trabalhos e irmãos na amizade que fizeram parte da minha formação e que vão continuar presentes em minha vida com certeza. As pessoas que me receberam de braços abertos em Lille – FR. Em especial, ao Prof. Dr. Nicolas Anquetil e ao grande amigo André Hora.

Aos meus queridos amigos Diego, Amanda, Mateus Ferrer (*White*), Matheus Viana (MC), Simone, Lucas (cabeça), Juliana (juju), Vinícius Pereira, Alinne, FCarlos, Valdemar Neto, Misael e André Oliveira (Deh).

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) (processo N. 2012/05168-4) por apoiar financeiramente este projeto de doutorado e ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) (processo N. 241028/2012-4) pelo financiamento do estágio no exterior.

Deixei para o fim os agradecimentos para minha companheira de toda vida, minha esposa Patrícia ♡. Acho que ela não tem noção do amor que sinto por ela e como sua presença é central em minha vida. Eu agradeço a ela por suas ações ter se tornado a pessoa mais importante da minha vida. Sem ela eu seria incompleto e não teria estrutura para ter trilhado o caminho que segui em minha vida até esse momento. Muito obrigado por todo seu apoio, por todo o seu amor e por todo seu carinho!.

“As invenções são, sobretudo,
o resultado de um trabalho de teimoso.”
(Santos Dumont)

RESUMO

DURELLI, R. S.. **Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura**. 2016. 239 f. Tese (Doutorado em em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

A Modernização Dirigida a Arquitetura (do inglês - *Architecture-Driven Modernization* (ADM)) é uma iniciativa do *Object Management Group* (OMG) no sentido de padronizar os tradicionais processos de reengenharia de software utilizando metamodelos. O metamodelo mais importante da ADM é o Knowledge Discovery Metamodel (KDM), cujo objetivo é representar todos artefatos de um determinado sistema, de forma independente de linguagem e plataforma. Um passo primordial durante processos de modernização de software é a aplicação de refatorações. No entanto, até o presente momento, há carência de abordagens que tratam das questões referentes a refatorações no contexto da ADM, desde a criação até a aplicação das mesmas. Além disso, atualmente, não existe uma forma sistemática e controlada de facilitar o reúso de refatorações que são específicas do KDM. Diante disso, são apresentados uma abordagem para criação e disponibilização de refatorações para o metamodelo KDM e um apoio ferramental que permite aplicá-las em diagramas de classe da UML. A abordagem possui dois passos: (*i*) o primeiro envolve passos que apoiam o engenheiro de modernização durante a criação de refatorações para o KDM; (*ii*) o segundo resume-se na especificação das refatorações por meio da criação de instâncias do metamodelo *Structured Refactoring Metamodel* (SRM) e posterior disponibilização delas em um repositório. O apoio ferramental, denominado KDM-RE, é composto por três *plug-ins* do Eclipse: (*i*) o primeiro consiste em um conjunto de Wizards que apoia o engenheiro de software na aplicação das refatorações em diagramas de classe UML; (*ii*) o segundo consiste em um módulo de propagação de mudanças, permitindo manter modelos internos do KDM sincronizados; (*iii*) o terceiro fornece apoio à importação e reúso de refatorações disponíveis no repositório. Além disso, o terceiro módulo também contém uma linguagem específica de domínio, a qual é utilizada para auxiliar o engenheiro de software a instanciar o metamodelo SRM. Foi realizado um experimento, buscando reproduzir os cenários em que engenheiros de software realizam refatorações em instâncias do metamodelo KDM. Os resultados mostraram que a abordagem, bem como o apoio ferramental podem trazer benefícios para o engenheiro de software durante a atividade de aplicação de refatorações em sistemas, representados pelo metamodelo KDM.

Palavras-chave: ADM, KDM, Refatoração, Engenheiro de Modernização, Desenvolvimento Dirigido por Modelos, Experimento.

ABSTRACT

DURELLI, R. S.. **Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura**. 2016. 239 f. Tese (Doutorado em em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

Architecture Driven Modernization (ADM) is an initiative of the Object Management Group (OMG) whose main purpose is to provide standard metamodels that enable the conduction of modernization activities as reverse engineering and software transformation. In this context, the most important metamodel is the Knowledge Discovery Metamodel (KDM), whose objective is to represent software artifacts in a language- and platform-agnostic fashion. A fundamental step during software modernization is refactoring. However, there is a lack of approaches that address how refactoring can be applied in conjunction with ADM. In addition, few efforts have investigated how refactorings can be reused and systematically applied within the context of KDM representations. We propose an approach for creating and cataloging refactorings tailored to KDM. Our approach is twofold: (*i*) the first part comprises steps that help the software modernization engineer create KDM-compliant refactorings; (*ii*) the second part has to do with turning these refactoring descriptions into Structured Refactoring Metamodel (SRM) and making them available to be reused. To automate these activities, we developed a tool that makes it possible to apply refactorings to Unified Modeling Language (UML) class diagrams. Our tool, named KDM-RE, comprises three Eclipse plug-ins, which can be briefly described as follows: (*i*) a set of Wizards aimed at supporting the software modernization engineer during refactoring activities; (*ii*) a change propagation module that keeps the internal metamodels synchronized; and (*iii*) a plug-in that supports the selection and reuse of the refactorings available in the repository. Moreover, we developed a domain specific language to minimize the effort required to instantiate SRMs. We carried out an evaluation that simulates how software modernization engineers would go about refactoring KDM instances. Our results would seem to suggest that our approach, when automated by our tool, provides several advantages to software modernization engineers refactoring systems represented by KDMs.

Key-words: ADM, KDM, Refactoring, Software Modernization Engineers, Model-Driven Development, Experiment.

LISTA DE ILUSTRAÇÕES

LISTA DE ALGORITMOS

LISTA DE CÓDIGOS-FONTE

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADM ..... *Architecture-Driven Modernization*
ADMPR .. *ADM Pattern Recognition*
ADMRS .. *ADM Refactoring Specification*
ADMTF .. *Architecture-Driven Modernization Task Force*
ADMVS .. *ADM Visualization Specification*
ANA ..... *Average Number of Ancestors*
AOIL ..... ArtofIllusion
ASCII .... *American Standard Code for Information Interchange*
ASTM ... *Abstract Syntax Tree Metamodel*
ATL ..... ATLAS *Transformation Language*
BNF ..... *Backus–Naur Form*
BPMN ... *Business Process Model and Notation*
CA ..... Camada de Abstração
CAM ..... *Cohesion Among Methods in class*
CASE .... *Computer-Aided Software Engineering*
CEE ..... Camada de Elementos Estruturais
CEP ..... Camada de Elementos de Programa
CI ..... Camada de Infraestrutura
CIM ..... *Computation Independent Model*
CIS ..... *Class Interface Size*
COMO ... *Component-Oriented MOdernization*
CR ..... Camada de Refatoração
CRTE .... Camada de Recurso de Tempo de Execução
CRTR .... Camada de Recurso de Transformação e Restrições
DAM ..... *Data Access Metric*
DCC ..... *Direct Class Coupling*
DFS ..... *Depth-First Search*
DSC ..... *Design Size in Classes*
EC ..... *Extract Class*
EI ..... *Extract Interface*
EMF ..... *Eclipse-Modeling Framework*

EMN ..... *Ecole des mines de Nantes*
ETL ..... *Epsilon Transformation Language*
GME ..... *Generic Modeling Environment*
GP ..... GanttProject
IDE ..... *Integrated Development Environment*
INRIA .... *Institut National de Recherche en Informatique et en Automatique*
ISO ..... *International Standards Organization*
JEX ..... Jexel
JFC ..... JFreeChart
JHD ..... JHotDraw
JPA ..... Java Persistence API
JUn ..... JUnit
KDM ..... Knowledge Discovery Metamodel
KDM-RE . *Knowledge Discovery Model-Refactoring Environment*
M2C ..... *Model-To-Code*
M2M ..... *Model-To-Model*
MC ..... *Move Class*
MDA ..... *Model-Driven Architecture*
MDD ..... *Model-Driven Development*
MDSD ... *Model-Driven Software Development*
MF ..... *Move Field*
MFA ..... *Measure of Functional Abstraction*
MM ..... *Move Method*
MOA ..... *Measure Of Aggregation*
MOF ..... *Meta-Object Facility*
MS ..... Mapeamento Sistemático
NOH ..... *Number Of Hierarchies*
NOM ..... *Number of methods*
NOP ..... *Number Of Polymorphic methods*
OAW ..... OpenArchitectureWare
OCL ..... *Object Constratint Language*
OMG ..... *Object Management Group*
PDF ..... *Push Down Field*
PDM ..... *Push Down Method*
PICO ..... *Population*, *Intervention*, *Comparator* e *Outcomes*
PIM ..... *Platform Independent Model*
POO ..... Paradigma Orientado a Objetos

PSM ..... *Platform Specific Model*
PUF ..... *Pull Up Field*
PUM ..... *Pull Up Method*
QMOOD . *Quality Model for Object-Oriented Design*
QPs ..... Questões de Pesquisas
QVT ..... *Query/View/Transformation*
RFP ..... *Request-for-Proposal*
RIA ..... *Rich Internet Application*
RS ..... Revisão Sistemática
SaaS ..... *Software as a Service*
SADT .... *Structured Analysis and Design Technique*
SMM ..... *Structured Metrics metamodel*
SPMS .... *Structured Patterns Metamodel Standard*
SRM ..... *Structured Refactoring Metamodel*
UML ..... *Unified Modeling Language*
XJ ..... Xerces-J
XMI ..... *XML Metadata Interchange*
XML ..... *eXtensible Markup Language*

SUMÁRIO

CAPÍTULO

1

INTRODUÇÃO

1.1 Contextualização

Sistemas legados são aqueles cujos custos de manutenção e evolução encontram-se fora dos níveis aceitáveis para uma organização, mas que ainda são úteis para apoiar seus processos internos. Frequentemente esse tipo de sistema possui uma estrutura inconsistente com a sua documentação, o que dificulta sua manutenção. Porém, substituí-lo completamente pode ser uma tarefa muito cara e propensa a erros (PRESSMAN, 2010). Segundo Pressman (2010), uma empresa de software pode despender de 60% a 70% de todos os seus recursos com manutenção de software.

Uma alternativa à substituição de um sistema legado são os tradicionais processos de reengenharia. Entretanto, em consequência de algumas pesquisas existentes, o *Object Management Group* (OMG) lançou em 2003 o termo Modernização Dirigida a Arquitetura (do inglês - *Architecture-Driven Modernization* (ADM)) cujo objetivo principal é padronizar processos de reengenharia por meio de metamodelos padronizados (ADM, 2012). O Knowledge Discovery Metamodel (KDM) é o principal metamodelo da ADM e possui uma ampla quantidade de metaclasses para representar níveis mais baixos de abstração de um sistema (código-fonte), níveis mais altos (arquitetura, regras de negócio e outros conceitos abstratos do sistema) e níveis técnicos (interface gráficas, arquivos de configuração, bases de dados, etc), permitindo a representação de conceitos de qualquer domínio (KDM, 2015; ISO/IEC19506, 2012).

A ADM almeja que a comunidade comece a desenvolver algoritmos e técnicas que sejam dependentes e específicas apenas do metamodelo KDM, e não de plataformas, metamodelos proprietários ou linguagens de programação específicas. Por exemplo, algoritmos de refatorações (DURELLI *et al.*, 2014a) que sejam específicos para o KDM tem o poder de alterar um sistema independentemente da linguagem de programação que foi usada em seu desenvolvimento, já que as refatorações ocorrem nos modelos. Outro exemplo seria a aplicação de algoritmos de

mineração de interesses transversais, utilizando como base o metamodelo KDM (DURELLI *et al.*, 2013; SANTIBANEZ *et al.*, 2013a; SANTIBANEZ *et al.*, 2015). A ideia principal da existência do KDM é que ele seja o metamodelo padrão de representação de sistemas dentro das ferramentas de modernização baseadas em ADM. Note-se que isso não significa que ele deva ser o único metamodelo interno e reconhecido pela ferramenta. Possivelmente, muitas ferramentas utilizarão o metamodelo UML para a apresentação gráfica do sistema. Entretanto, o uso do KDM permite que algoritmos que atuam sobre esse metamodelo possam ser importados/exportados entre essas ferramentas, propiciando interoperabilidade entre elas e, consequentemente, melhorando a produtividade dos desenvolvedores/mantenedores dessas ferramentas.

Um processo de modernização típico da ADM possui três principais passos: (*i*) engenharia reversa, onde um sistema legado deve ser transformado em uma instância do metamodelo KDM; (*ii*) reestruturação, onde essa instância do KDM é submetida à diversas transformações, ou seja, otimizações e refatorações; e (*iii*) engenharia avante, onde a instância otimizada/refatorada do KDM é transformada novamente em código-fonte. Dessa forma, para apoiar e automatizar esses três passos, é importante que o processo de modernização da ADM possua ferramentas de modernização que atuem em nível de modelo e auxiliem os engenheiros de forma sistemática e guiada para facilitar o processo de modernização.

Considerando o processo de modernização da ADM, uma atividade fundamental durante o passo de reestruturação são as refatorações. Refatorações foram primeiramente propostas por Opdyke (1992) como uma metodologia para reestruturar programas. Seguindo a mesma linha de pensamento, pesquisadores como Fowler (1999) tornaram a refatoração uma disciplina comumente conhecida e aplicada, a qual é um processo disciplinado e utilizado para melhorar a estrutura do software, preservando o comportamento do mesmo (FOWLER, 1999). Com o apoio ferramental adequado, a refatoração pode ser uma maneira eficiente e eficaz para: (*i*) ajudar a melhorar o projeto do software, (*ii*) tornar o software mais fácil de ser entendido, e (*iii*) auxiliar na identificação de erros.

Embora o conceito de refatoração seja bem conhecido e aplicado em código-fonte, sua aplicação em modelos ainda apresenta desafios de pesquisa (GORP *et al.*, 2003). De acordo com Mens e Gorp (2006), refatorações em modelos tendem a ser mais complexas do que refatorações aplicadas em código-fonte, uma vez que, além das refatorações, é necessária também a realização de uma atividade para verificar a consistência do modelo com o código-fonte e manter a sincronização entre eles (KOLAHDOUZ-RAHIMI *et al.*, 2014). Uma das vantagens em se utilizar refatorações em nível de modelos é que os desenvolvedores de software não precisam se preocupar com características específicas de linguagens de programação (Java, C++, C#, etc). Além disso, modelos fornecem uma visão gráfica e abstrata do sistema, assim, o engenheiro de software pode facilmente visualizar e verificar quais refatorações devem ser aplicadas no sistema.

Quando se trata de refatorações em nível de modelo, a maior parte das pesquisas apre-

sentam propostas de aplicação de refatorações em diagramas UML (SALEM; GRANGEL; BOUREY, 2008; GORP *et al.*, 2003; EGYED; LETIER; FINKELSTEIN, 2008; BRIAND *et al.*, 2006; STAROŃ; KUŹNIARZ, 2004; MISBHAUDDIN; ALSHAYEB, 2015). Isso se dá principalmente porque a UML é o padrão geralmente usado para visualizar a estrutura de um sistema. No entanto, utilizar apenas UML como metamodelo base em ferramentas de modernização limita as atividades de reestruturação às visões disponíveis na UML. De acordo com Ulrich e Newcomb (2010), há vários cenários de modernização de sistemas que demandam visões e representações que extrapolam os diagramas da UML e que podem ser representados com o KDM (GORP *et al.*, 2003; KOLAHDOUZ-RAHIMI *et al.*, 2014; MISBHAUDDIN; ALSHAYEB, 2015; DURELLI *et al.*, 2014b). Por exemplo, a UML não contém um conjunto de metaclasses dedicadas e específicas para representar níveis mais baixos de abstração de um sistema, como o código-fonte, nem mesmo metaclasses para representar níveis mais altos, como conceitos abstratos, arquitetura, regras de negócio e interfaces.

Neste contexto, o cenário ideal para concretizar a proposta original da ADM é que ferramentas de modernização apliquem refatorações utilizando diagramas da UML, porém, internamente a ferramenta deve aplicar as refatorações em instância do metamodelo KDM ao invés da UML, assim, diversos artefatos de um sistemas podem continuar consistentes e sincronizados após a aplicação de refatorações em instâncias do KDM. Além disso, algoritmos de refatorações podem ser reutilizados em diversas ferramentas de modernização, uma vez que, o KDM é independente de linguagem e plataforma. Assim, vislumbrou-se uma oportunidade de conduzir essa pesquisa no contexto da ADM.

1.2 Motivações

As motivações que levaram ao desenvolvimento desta tese foram:

1. Carência de ferramentas de modernização que permitem aplicar refatorações em diagramas UML que são representações gráficas de modelos KDM. Isto é, a maioria das ferramentas que permitem refatorações em diagramas UML, mantém internamente apenas o metamodelo UML. Isso faz com que os algoritmos de refatoração não sigam o padrão KDM, impedindo seu reúso em ferramentas de modernização que utilizam esse padrão. Além disso, o metamodelo UML restringe o número de refatorações possíveis em processos de modernização quando comparado com o metamodelo KDM.

2. Escassez de abordagens de criação de refatorações para o KDM. Esta motivação está relacionada com a ausência de diretrizes claras que guiem engenheiros de modernização na criação de refatorações para esse metamodelo. A ausência de diretrizes desse tipo faz com que os engenheiros de modernização adotem procedimentos *ad-hoc* que podem levar a refatorações que não executam corretamente sua função.

3. Ausência de um metamodelo que forneça uma terminologia comum, padronizada e independente de linguagem para a especificação de refatorações. Essa ausência faz com que engenheiros de modernização tenham que criar refatorações usando linguagens de transformação de modelos específicas (QVT, ATL, Kermeta, etc.) e sem seguir nenhum padrão de terminologia, tornando-as bastante específicas e restringindo suas chances de reúso. Uma especificação de refatoração de mais alto nível habilita ferramentas de modernização a gerarem as implementações das refatorações na linguagem que for de interesse. Dessa forma, ferramentas de modernização que reconheçam um metamodelo de refatorações internamente podem gerar as implementações das refatorações a partir da leitura das instâncias desse metamodelo.

1.3 Objetivos

O objetivo desta tese é apresentar uma abordagem para criação e disponibilização de refatorações para o metamodelo KDM, bem como um apoio ferramental que permite aplicá-las em diagramas de classe da UML. Mais especificamente, os seguintes objetivos podem ser especificados:

- Tornar a criação de refatorações para o metamodelo KDM um processo sistemático e guiado, facilitando a tarefa do engenheiro de modernização e procurando garantir que as refatorações desenvolvidas estejam estruturadas corretamente com base nas metaclasses do KDM;

- Potencializar o reúso das refatorações desenvolvidas. Este objetivo é apoiado por um metamodelo para a especificação das refatorações. A ideia é que ferramentas de modernização adotem esse metamodelo como base para suas refatorações, permitindo assim o reúso de instâncias de refatorações entre ferramentas. Esse metamodelo possui um conjunto de metaclasses que define meta-atributos específicos para representar informações (metadados) de refatoração, auxiliando, assim, o compartilhamento das refatorações de forma intuitiva entre os engenheiros.

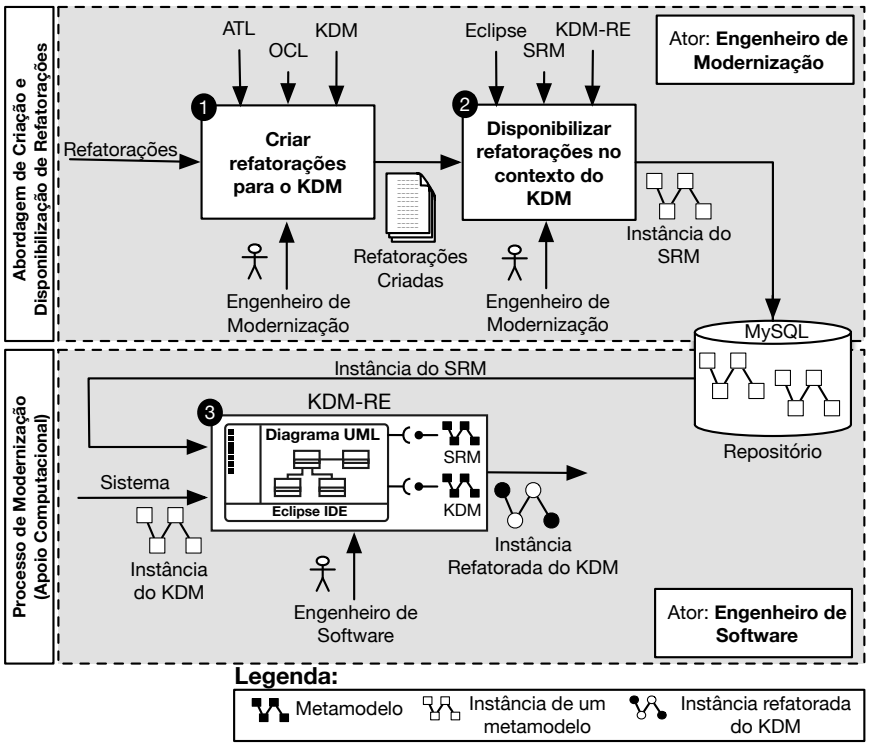
- Viabilizar a aplicação de refatorações em modelos KDM a partir de diagramas UML. Este objetivo é apoiado por um apoio ferramental totalmente integrado ao ambiente de desenvolvimento Eclipse. Esse apoio ferramental permite a aplicação de refatorações graficamente por meio de diagramas de classe da UML, porém, as refatorações são realizadas transparentemente no metamodelo KDM e, posteriormente, replicadas nos diagramas de classe da UML.

1.4 Síntese da Pesquisa Conduzida

Com base nas motivações e nos objetivos supracitados, o trabalho desenvolvido aqui visa auxiliar dois principais atores: (*i*) o engenheiro de modernização e (*ii*) o engenheiro de software. O engenheiro de modernização, no contexto desta tese, é aquele que utiliza a abordagem aqui proposta para criar, disponibilizar e reutilizar refatorações para o KDM. Por sua vez, o engenheiro de software é aquele que é responsável por conduzir um processo de modernização e que utiliza apoios computacionais para aplicar refatorações existentes em diagramas de classe da UML.

Na Figura 1, uma visão geral do trabalho desenvolvido é apresentada. Note que a figura é dividida em dois retângulos tracejados. O primeiro retângulo representa os passos da abordagem e o segundo ilustra a ferramenta de apoio a processos de modernização desenvolvida. A abordagem é composta por dois passos e o processo de modernização é abstraído em apenas um passo. Note também que a figura ilustra o papel do engenheiro de modernização e do engenheiro de software. Os dois passos da abordagem são executados pelo engenheiro de modernização e o processo de modernização é executado pelo engenheiro de software. A seguir, uma descrição resumida é apresentada.

Figura 1 – Visão geral da abordagem proposta.



Fonte: Elaborada pelo autor.

O passo ❶ consiste na criação de refatorações para o metamodelo KDM. Esse passo é apoiado por seis diretrizes, as quais o engenheiro de modernização segue para criar refatorações para o metamodelo KDM. As refatorações são um conjunto de regras ATL, as quais são criadas por meio da combinação de um conjunto de artefatos: (*i*) mapeamento entre KDM e

Paradigma Orientado a Objetos (POO), (*ii*) operações atômicas, (*iii*) *templates*, (*iv*) linguagem de transformação e (*v*) linguagem de restrição.

O passo ❷ consiste na disponibilização das refatorações criadas por meio de um metamodelo aqui definido (*Structured Refactoring Metamodel* - SRM), o qual contém metaclasses que permitem armazenar metadados relacionados com refatorações, tais como: o nome da refatoração, sua motivação, autor, pré- e pós-condições e seu mecanismo. O objetivo desse metamodelo é viabilizar o reúso das refatorações dentro de ferramentas de modernização, propiciando a interoperabilidade entre as mesmas. A instanciação desse metamodelo é apoiada por um apoio computacional criado no contexto desta tese denominado KDM-RE, que fornece uma linguagem específica de domínio para auxiliar o engenheiro de modernização durante a instanciação desse metamodelo. Em seguida, as instâncias desse metamodelo são enviadas pelo KDM-RE para um repositório e podem ser reutilizadas por engenheiros de software.

O processo de modernização (ver Figura 1 ❸) dá enfoque na aplicação de refatorações e a propagação de mudanças por meio do KDM-RE. KDM-RE foi implementado para automatizar a atividade de aplicação e reutilização de refatorações em sistemas representados pelo KDM. As refatorações podem ser aplicadas diretamente em diagramas de classes UML, porém, a refatoração é de fato realizada transparentemente no metamodelo KDM e, posteriormente, replicada nos diagramas de classes UML. Adicionalmente, após a aplicação de refatorações em sistemas representados pelo KDM, é importante manter todos os pacotes/artefatos sincronizados e consistentes. Dessa forma, o KDM-RE também contém um *plug-in* responsável por aplicar regras de propagações, que são realizadas em instância do metamodelo KDM. O intuito desse *plug-in* é manter todos os artefatos sincronizados e consistentes de acordo com a refatoração aplicada.

1.5 Convenções Adotadas nesta Tese

Ao longo desta tese, *Itálico* é utilizado para dar ênfases, introduzir novos termos e para destacar palavras em inglês. `Typewriter` é utilizado para operador Java, operador da DSL, palavras-chaves, nome de metaclasses, meta-associação, meta-atributo, nome de métodos, variáveis e URL que aparecem no texto. Símbolos numéricos (❶, ❷, ❸, ❹, etc) são usados para chamar a atenção do leitor para informações importantes em figuras e códigos.

1.6 Grupo de Pesquisa

Este trabalho é uma contribuição para o grupo de pesquisa do Departamento de Ciência de Computação e Estatística do Instituto de Ciências Matemáticas e de Computação (ICMC) da Universidade de São Paulo (campus São Carlos/SP). Além disso, a presente pesquisa também foi conduzida em parceria com o grupo de pesquisa AdvanSE (*Advanced Research on Software*

Engineering), da Universidade Federal de São Carlos (UFSCar). O grupo possui pesquisas em andamento sobre extensões, refatorações, mineração, métricas e validações de arquitetura utilizando a ADM e o metamodelo KDM, das quais o autor desta tese participa efetivamente.

1.7 Estrutura da Tese

Esta tese está organizada em oito capítulos. No primeiro capítulo, estão apresentados a contextualização, a motivação, a abordagem desenvolvida em resumo, os objetivos, as convenções adotadas e o grupo de pesquisa do trabalho.

No Capítulo 2, estrutura-se uma revisão dos principais conceitos envolvendo MDE e refatoração com o objetivo de facilitar a compreensão com relação à tese. Além disso, é feita uma contextualização sobre a modernização de sistemas com a utilização dos padrões propostos pelo OMG, ADM e KDM, em que seus conceitos e particularidades são observados. Também é apresentada a ferramenta MoDisco. No Capítulo 3, é apresentado um mapeamento sistemático que foi realizado com o objetivo de identificar e entender soluções já desenvolvidas sobre ADM e KDM. Além disso, nesse mapeamento também são demonstradas as principais constatações e questões em aberto que nortearam a tese.

Como descrito na Figura 1, a abordagem aqui proposta contém três passos. O passo ❶ é apresentado no Capítulo 4, onde são destacadas as diretrizes para criar refatorações para o metamodelo KDM. O passo ❷ é salientado no Capítulo 5, o qual apresenta um metamodelo para disponibilizar e promover o reúso de refatorações no contexto da ADM e KDM. O passo ❸ é alicerçado por um apoio computacional, o qual é apresentado no Capítulo 6. Esse apoio computacional é denominado KDM-RE e é composto por três *plug-ins* do Eclipse: (*i*) o primeiro consiste em um conjunto de *Wizards* que apoia o engenheiro de software na aplicação das refatorações em diagramas de classe UML; (*ii*) o segundo consiste em um apoio à importação e reúso de refatorações disponíveis no repositório; (*iii*) o terceiro consiste em um módulo de propagação de mudanças que permite manter modelos internos do KDM sincronizados.

No Capítulo 7, é mostrado o planejamento, execução e análise dos dados de um experimento que visa validar a abordagem desenvolvida nesta tese. E, por fim, no Capítulo 8, são descritas as conclusões do trabalho com as principais contribuições, limitações, lições apreendidas, publicações e trabalhos futuros que poderão ser conduzidos como continuação da presente pesquisa.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

Neste capítulo, são apresentados e discutidos os conceitos fundamentais para o entendimento desta tese. Está organizado da seguinte forma: na Seção 2.2, os conceitos sobre Engenharia Dirigido por Modelos são salientados; na Seção 2.3, são definidos os conceitos sobre refatoração, bem como refatorações para modelos. Na Seção 2.4, é feita uma contextualização sobre a Modernização Dirigida a Arquitetura; na Seção 2.5, o metamodelo KDM é apresentado; na Subseção 2.5.1, é descrito o pacote `Code`; na Subseção 2.5.2, é descrito o pacote `Action`; na Subseção 2.5.3, é apresentado o pacote `Structure`; na Seção 2.6, é mostrada uma ferramenta de apoio ao KDM; na Seção 2.7, as considerações finais deste capítulo são destacadas.

2.2 Engenharia Dirigida por Modelos

De acordo com Booch (2004) modelos são abstrações de sistemas que permitem raciocinar e entender o sistema, ignorando detalhes irrelevantes, enquanto o foco é dado aos detalhes mais relevantes. Segundo Brown e McDermid (2007), Bezivin e Gerard (2002), a utilização de modelos para o desenvolvimento de software não é algo novo. Modelos são usados há algumas décadas para auxiliar a concepção e o projeto de software, sendo utilizados basicamente nas fases iniciais do desenvolvimento. Por exemplo, modelos como os da UML (UML, 2015) não fazem parte do software em si, embora sejam importantes para o entendimento e a construção. Os desenvolvedores os criam, mas os descartam, implementando as funções de forma manual e realizando manutenções somente no código-fonte. Desse modo, o conhecimento acerca da solução fica criptografado no código-fonte e dificilmente é reutilizado. Portanto, há necessidade de criar modelos que representem esse conhecimento e que possam ser úteis tanto para a documentação, quanto para a construção e manutenção de software.

A partir dessa ideia, a *Model-Driven Engineering* (MDE) surge como uma solução complementar aos processos de desenvolvimento tradicionais (LIMA; SOUSA; LOPES, 2007). A MDE é uma abordagem que propõe reduzir a distância semântica entre o problema do domínio e a solução/implementação. Nessa abordagem, o desenvolvimento de software ocorre por meio de modelos que protegem os desenvolvedores das complexidades da implementação e de transformações que originam o código-fonte de maneira automatizada a partir das informações contidas nesses modelos. Na literatura, é possível também identificar MDE como outros acrônimos, por exemplo, *Model-Driven Development* (MDD), *Model-Driven Software Development* (MDSD) ou MD* (KLEPPE; WARMER; BAST, 2003), e todos eles dizem respeito à mesma abordagem. O que é importante ter em mente sobre MDE é que modelos são utilizados no centro do processo de desenvolvimento e manutenção de software. Na MDE, os modelos assumem o papel principal em todo o ciclo de vida de um software e a relevância dos modelos vai além da documentação do software desenvolvido. Na MDE, modelos passam a ser utilizados como artefatos principais; eles podem ser compreendidos por computadores e são fundamentais para o desenvolvimento do software, pois podem ser manipulados, refinados e transformados em uma nova versão, até que a partir deles seja gerado o código-fonte (KLEPPE; WARMER; BAST, 2003; BROWN; MCDERMID, 2007; AMMAR; MAHFOUDHI, 2013).

Na MDE, o enfoque do desenvolvimento é direcionado aos modelos, ou seja, a modelagem deixa de ser meramente uma forma de planejar o código e passa a ser uma forma de construir o software. O nível de abstração da programação torna-se mais elevado, reduzindo a necessidade do desenvolvedor de interagir manualmente com o código-fonte (BRAGANCA; MACHADO, 2007). É importante salientar que a MDE não tem como objetivo substituir o processo tradicional de desenvolvimento de software (KLEPPE; WARMER; BAST, 2003; BROWN; MCDERMID, 2007; BRAGANCA; MACHADO, 2007) e sim contribuir para o seu aprimoramento. O OMG (ADM, 2012) definiu um modelo de arquitetura para o MDE, conhecido como *Model-Driven Architecture* (MDA), o qual tem como objetivo promover o uso de modelos no desenvolvimento de software, para fornecer uma solução ao gerenciamento da complexidade do desenvolvimento, manutenção e evolução de sistemas de software e favorecer a interoperabilidade e portabilidade desses sistemas.

O OMG definiu formalmente a MDA como uma abordagem que é bem definida pela ideia de separar a especificação das operações de um sistema dos detalhes de como tal sistema usa as potencialidades de sua plataforma. Isso possibilita que ferramentas ofereçam a especificação de um sistema de forma independente de plataforma. De acordo com o OMG, os principais objetivos da MDA são a portabilidade, a interoperabilidade e a reusabilidade. Para alcançar esses objetivos a MDA divide o desenvolvimento de software em níveis de abstração (FRANCE; RUMPE, 2007; AMMAR; MAHFOUDHI, 2013):

- Modelo Independente de Computação (do inglês - *Computation Independent Model* (CIM)): descreve o negócio, ou ambiente, no qual o software irá operar. Como a decisão a

respeito da informatização é feita por uma pessoa e não por uma ferramenta de transformação, dificilmente a transformação automatizada desse modelo para o de nível seguinte é implementada;

- Modelo Independente de Plataforma (do inglês - *Platform Independent Model* (PIM)): nível de análise em que ocorre a definição das características do software ou domínio. Esse modelo pode ser transformado para um ou mais modelos do nível seguinte;
- Modelo Específico de Plataforma (do inglês - *Platform Specific Model* (PSM)): nível de projeto que considera as tecnologias de implementação, devendo existir uma instância desse modelo para cada plataforma de implantação do software. A partir desse modelo ocorre a transformação para o código-fonte.

Com base nos princípios mencionados, podem-se citar as seguintes vantagens da MD* (HUTCHINSON *et al.*, 2011; FRANCE; RUMPE, 2007; SCHMIDT, 2006):

1. Maior facilidade na criação dos modelos das aplicações, pois é utilizada uma linguagem específica para o domínio do problema;
2. Maior produtividade e redução do esforço dos desenvolvedores, pois a maior parte do código-fonte das aplicações pode ser gerado a partir dos modelos;
3. Maximização do tempo de vida útil dos modelos e de outros artefatos, pois, como são necessários para a geração de código, eles são menos propensos a serem descartados pelos desenvolvedores nos processos de manutenção;
4. Flexibilidade de desenvolvimento, pois os modelos independentes de plataforma armazenam a lógica do sistema e são menos sensíveis a mudanças;
5. O conhecimento a respeito do software não fica exclusivamente na mente dos desenvolvedores e no código, fazendo com que os processos de desenvolvimento e de manutenção fiquem menos vulneráveis às oscilações de pessoal.

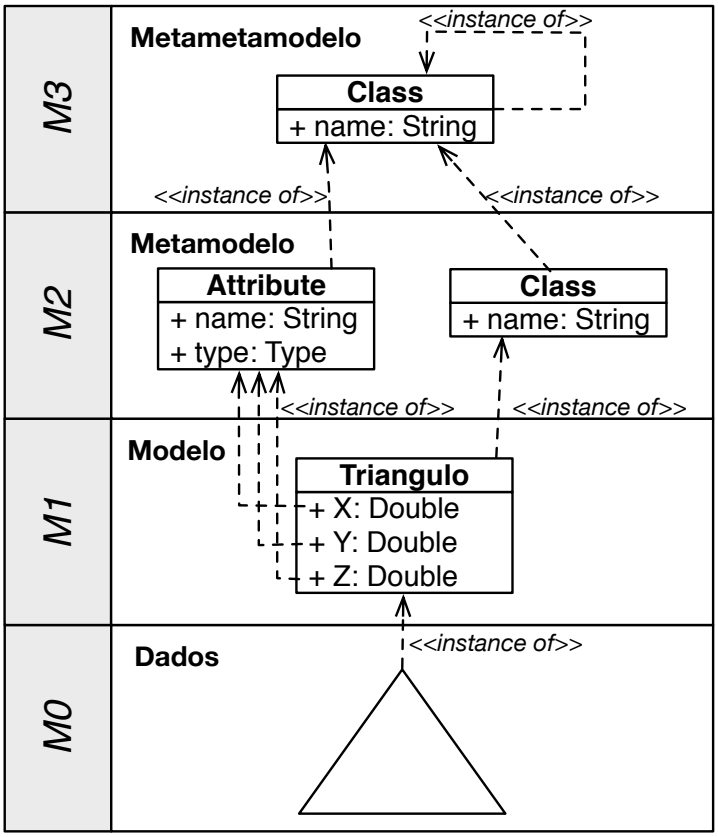
A MD* não pretende substituir os processos tradicionais de desenvolvimento de software, e sim contribuir para seu aprimoramento de uma maneira racional para mover informações de uma fase de desenvolvimento para outra, trazendo respostas rápidas e eficientes para atender as necessidades inesperadas de novos requisitos.

De acordo com Kleppe, Warmer e Bast (2003), um modelo deve ser definido como uma descrição de um (ou de uma parte) sistema expresso em uma linguagem bem definida, isto é, respeitando uma sintaxe e uma semântica. Essa descrição deve ser conveniente para uma interpretação automatizada por computadores. Assim, para criar um modelo, deve-se seguir uma sintaxe precisa e uma semântica bem definida, a fim de regulamentar a criação de elementos

e suas relações. Em MDE, esse formalismo pode ser alcançado utilizando uma linguagem de modelagem. Tais linguagens são especificações que contêm os elementos bases para construir modelos, concebidas dentro de um domínio limitado e com objetivos específicos. Usualmente, uma linguagem de modelagem pode ser gráfica ou textual com notação matemática e deve permitir a definição de modelos sem ambiguidades. O uso de uma linguagem bem definida sintaticamente e semanticamente permite o entendimento e manipulação de modelos por pessoas e computações (HUTCHINSON *et al.*, 2011; FRANCE; RUMPE, 2007; SCHMIDT, 2006).

Uma linguagem de modelagem geralmente está em conformidade com um metamodelo. Um metamodelo é um modelo que define uma linguagem para representar um modelo. A linguagem de modelagem é simplesmente um modelo do metamodelo. Ela define a estrutura, a semântica e as restrições para uma família de modelos (MELLOR *et al.*, 2004). Na Figura 2, são apresentados os quatro níveis de modelos da MDE. É importante notar que as relações entre os níveis descritos na Figura 2 são do tipo "`instance of`" (instância de), definido por Bezivin e Gerard (2002), Brambilla, Cabot e Wimmer (2012). Os quatros níveis são descritos a seguir:

Figura 2 – Arquitetura de Metamodelagem.



Fonte: Elaborada pelo autor.

- metametamodelo (M3): M3 constitui a base da arquitetura de meta-modelagem. A função primordial desse nível é definir linguagens para especificar metamodelos. Um meta-metamodelo define um modelo de mais alto nível de abstração que o metamodelo, e o

primeiro é tipicamente mais compacto que o segundo. *Meta-Object Facility* (MOF) (MOF, 2015) e *Eclipse-Modeling Framework* (EMF)(EMF, 2015) são exemplos de metametamodelos;

- metamodelo (M2): um metamodelo representa uma instância de um metametamodelo. A função principal do nível do metamodelo é definir uma linguagem para especificar modelos. Os metamodelos são tipicamente mais elaborados que os metametamodelos. Por exemplo, a UML (UML, 2015) e o KDM (ISO/IEC19506, 2012) possuem metamodelos que os descrevem estruturalmente;

- modelo (M1): um modelo é uma instância de um metamodelo. A função principal do nível de modelo é definir uma linguagem para descrever um domínio específico;

- dados (M0): os objetos de usuários representam os dados finais. A principal responsabilidade dos objetos de usuários é descrever um domínio específico em uma plataforma final.

Um objetivo claro da MDA é fornecer um *framework* que integra os padrões existentes do OMG. Os principais padrões do OMG utilizados nesta tese são:

- *Meta Object Facility* (MOF): Linguagem abstrata e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Essa especificação contém um conjunto de construtores que é utilizado para a definição de metamodelos. MOF pode ser utilizado para definir outras linguagens;

- *Unified Modeling Language* (UML): Linguagem para especificação, construção, visualização e documentação de artefatos de software. Essa linguagem permite a modelagem de diferentes aspectos ou pontos de vista de um sistema;

- *Knowledge Discovery Metamodel* (KDM): Metamodelo utilizado para representar em nível de modelos artefatos de sistemas de software. Na Seção 2.5, maiores informações sobre esse metamodelo, bem como suas metaclasses são apresentadas;

- *XML Metadata Interchange* (XMI): Padrão do OMG para troca de informação baseado em *eXtensible Markup Language* (XML). Pode ser utilizado para trocar qualquer informação, cujo metamodelo pode ser expresso utilizando MOF. O uso mais comum da XMI é como um formato de intercâmbio de modelos UML, embora também possa ser utilizado para a serialização de modelos de outras linguagens;

- *ATL Transformation Language* (ATL): Implementação da *Query/View/Transformation* (QVT) que é uma especificação híbrida padronizada para transformação de modelos no contexto de metamodelagem MOF. ATL aceita construções declarativas e imperativas (ver Subseção 2.3.2); *Object Constratint Language* (OCL): Linguagem declarativa para

descrever regras que se aplicam aos modelos. A OCL, inicialmente, era apenas uma extensão da UML para especificações formais de modelos. Hoje em dia, a OCL pode ser utilizada para especificar pré- e pós-condições, e é usada em qualquer modelo cujo metamodelo seja MOF.

2.3 Refatorações

Refatoração pode ser entendida como um processo de redistribuição de funcionalidade com o intuito de melhorar um dado sistema. No contexto do paradigma orientado a objetos, essa redistribuição está totalmente ligada com classes, atributos e operações. A refatoração tem como objetivo permitir a redistribuição de classes, atributos e operações na hierarquia de classes para facilitar futuras atividades de desenvolvimento ou de manutenção. A primeira definição de refatoração foi concebida por Opdyke (1992) da seguinte forma: "refatorações são transformações que reestruturam um determinado sistema com o objetivo de melhorar o *design*, a evolução e o reúso de sistemas desenvolvidos no paradigma orientado a objeto".

No contexto do paradigma orientado a objetos, refatoração é uma alternativa do conceito de reestruturação. Em outras palavras, refatoração é um termo aplicado ao paradigma orientado a objetos; para outros paradigmas de programação, esse mesmo processo é descrito como reestruturação (CHIKOFSKY; CROSS J.H., 1990). De acordo com Chikofsky e Cross J.H. (1990), reestruturação "consiste no processo de alterar um software, melhorando a sua estrutura interna, de forma que o comportamento externo do código não seja alterado". Reestruturação e refatoração são técnicas essenciais utilizadas para mitigar problemas relacionados à evolução de software (OPDYKE, 1992). Com o objetivo de melhorar atributos de qualidade dos sistemas, as práticas de refatoração surgiram por meio do emprego de reestruturação sobre unidades de código preservando o seu comportamento (CHIKOFSKY; CROSS J.H., 1990; OPDYKE, 1992).

Quando aplicada durante a fase de manutenção de software, a refatoração ajuda a tornar o código mais legível e também tem como objetivo solucionar problemas de códigos mal escritos (CHIKOFSKY; CROSS J.H., 1990). A refatoração também pode ser usada no contexto da reengenharia, a fim de alterar um sistema específico, visando reconstruí-lo em um novo formato. Nesse contexto, a refatoração é necessária para converter código legado ou deteriorado em um formato mais estruturado ou modular, ou para migrar o código para uma diferente linguagem de programação, ou mesmo um diferente paradigma de linguagem.

Em seu livro, Fowler (1999) apresenta duas definições para refatoração, uma como substantivo e outra como verbo:

- Refatoração: uma mudança que é realizada na estrutura interna de um determinado sistema com o objetivo de deixá-lo mais fácil de ser entendido e de ser modificado, sem alterar o seu comportamento externo;

- Refatorar: reestruturar o software por meio de um conjunto de refatorações sem modificar o seu comportamento externo.

As definições apresentadas por Fowler (1999) enfatizam que o propósito da refatoração é fazer com que o software fique mais fácil de ser entendido (melhorar sua compreensão) e de ser modificado (melhorar sua manutenibilidade). Outra característica de suma importância a ser destacada é que a refatoração, em geral, deve ser um processo para melhorar o *design* do software. De acordo com Wake (2003), refatoração é "uma arte para melhorar cuidadosamente o *design* de códigos existentes". O autor também enfatiza que refatoração deve fornecer maneiras de identificar problemas no código e também deve prover soluções para corrigir tais problemas. Ele caracteriza refatoração como:

- a refatoração não inclui nenhuma mudança no sistema, isto é, refatoração não deve adicionar novas funcionalidades ao sistema;
- a refatoração deve ser utilizada para melhorar o código do sistema;
- nem toda reestruturação pode ser considerada uma refatoração - usualmente refatorações tendem a ser transformações pequenas e seguras.

O primeiro conjunto de refatorações foi proposto por Opdyke (1992), onde o autor definiu 26 refatorações de baixa granularidade. Tais refatorações podem ser resumidas da seguinte forma:

- criar um membro - variável/função/classe: Essas refatorações têm como objetivo criar novas variáveis e/ou funções para uma classe em particular ou criar uma nova classe;
- deletar um membro - variável/função/classe. Essas refatorações têm como objetivo deletar membros que não são utilizados;
- renomear um membro - variável/função/classe. Essas refatorações podem ser utilizadas pare renomear membros e fornecer nomes mais significativos;
- mover um membro - variável/função. Essas refatorações são utilizadas para redistribuir um conjunto de variáveis/funções para sub ou superclasses.

Similarmente, Roberts (1999) definiu um conjunto de refatorações que devem ser aplicadas em classes, métodos e atributos. Porém, o catálogo mais completo e extensivo de refatorações foi definido por Fowler (1999), no qual cada refatoração possui os seguintes tópicos: (*i*) um nome, (*ii*) uma breve descrição, (*iii*) uma motivação para a condução da refatoração, (*iv*) um mecanismo descrevendo como a refatoração deve ser executada e (*v*) um exemplo ilustrando a utilização da refatoração. As refatorações propostas por Fowler (1999) são agrupadas em sete categorias, a saber: (*i*) *Composing Methods*, (*ii*) *Moving Features Between Objects*, (*iii*) *Organizing Data*,

(*iv*) *Simplifying Conditional Expressions*, (*v*) *Make Method Calls Simpler*, (*vi*) *Dealing with Generalization* e (*vii*) *Big Refactorings*.

De acordo com Fowler (1999), existem quatro principais motivações para a aplicação de refatoração:

1. refatorações quando bem conduzidas tendem a melhorar o *design* do software, podendo, assim, auxiliar na prevenção da decadência do software e eliminar código duplicado;
2. refatorações fazem com que o código-fonte fique mais fácil de entender - código bem legível facilita a comunicação e seu propósito;
3. refatoração auxilia na identificação de erros - melhorando a estrutura interna do código-fonte, erros tendem a ser identificados mais facilmente;
4. desenvolvimento mais produtivo - uma boa estrutura interna usualmente facilita o desenvolvimento e melhora a produtividade.

Como já salientado, refatorações devem preservar o comportamento de um determinado software após a aplicação de *n* refatorações. Dessa forma, Mens e Tourwe (2004), Ó Cinnéide (2000) relatam que existem três principais abordagens para auxiliar a preservação (de alguns aspectos) do comportamento do código-fonte. Tais abordagens são: (*i*) abordagem não formal (por exemplo, as refatorações definidas por Fowler (1999)), (*ii*) uma abordagem semiformal (ROBERTS, 1999) e (*iii*) abordagem completamente formal. No entanto, os autores também argumentam que mesmo com a utilização da última abordagem, é impossível garantir totalmente a preservação de comportamento após a aplicação de refatorações (MENS; TOURWE, 2004; Ó CINNÉIDE, 2000).

Código-fonte 1: Simples exemplo do efeito de uma refatoração.

```
1  public class Foo {
2      public void method(){
3          String className = this.getClass().getName();
4          System.out.println(className);
5      }
6  }
```

A ideia de preservação de comportamento no contexto de refatoração foi primeiramente introduzida por Opdyke (1992) da seguinte forma: “Se um programa é chamado duas vezes (antes e depois da refatoração) com o mesmo conjunto de entradas, o resultado deve ser o mesmo”. Essa explicação é plausível e utilizada na literatura (ROBERTS, 1999; FOWLER, 1999), mas, infelizmente, não é suficiente. Por exemplo, considere o seguinte cenário: se uma classe, ou um método ou outra estrutura de código for renomeada utilizando a refatoração `Rename`, será

desejado que todas as declarações e utilizações correspondentes também sejam atualizadas. No entanto, suponha o Código-fonte 1, se a classe `Foo` for renomeada para `Bar`, o comportamento do programa será alterado: o Código-fonte 1 não irá imprimir "Foo" e sim "Bar". Dessa forma, a definição apresentada por Opdyke (1992) não é verdadeira para esse cenário.

Outra abordagem para a definição de comportamento é exigir a preservação sintática e semântica de um sistema após a aplicação de refatorações. Obviamente, uma refatoração por definição não deveria invalidar a sintaxe de um sistema. Usualmente, a sintaxe e a semântica são preservadas por meio de asserções. Asserção é definida por meio de pré- e pós-condições que são executadas antes e após a aplicação de uma refatoração. Pré-condições são asserções que um sistema deve satisfazer para que a refatoração possa ser aplicada de forma segura. Pré-condições podem ser pensadas como condições que caracterizam válidas as transformações. Por exemplo, uma possível pré-condição para a refatoração `Rename Class` seria verificar se o novo nome da classe já existe dentro do pacote em que a classe está definida. Opdyke (1992) foi o primeiro pesquisador a utilizar asserções para garantir que as refatorações aplicadas preservassem a sintaxe e a semântica dos sistemas. É importante observar que as refatorações apresentadas nesta tese para o metamodelo KDM foram propostas e criadas com o intuito de preservar a sintaxe e a semântica do sistema. O metamodelo KDM provê meios de garantir que a estrutura do código-fonte foi preservada utilizando o pacote `Code` e `Action`.

O processo para a aplicação de um refatoração contém três principais passos (WAKE, 2003). O primeiro passo foca a identificação de partes do código que precisam ser refatoradas. O segundo passo baseia-se na escolha da melhor refatoração para solucionar o problema anteriormente identificado. E o terceiro passo resume-se na aplicação da refatoração. Seguindo a mesma ideologia e fundamentação proposta por Fowler (1999) e Opdyke (1992), existe a possibilidade de aplicar refatorações para modelos. Na subseção, a seguir, os conceitos e características relacionados com refatorações para modelos são apresentados.

2.3.1 Transformações e Refatorações de Modelos

Refatorações para modelos são transformação de modelos que têm como principal objetivo melhorar a estrutura do modelo e também preservar suas características internas. É uma área de estudo relativamente nova quando comparada com refatorações tradicionais, ou seja, aquelas aplicadas em código-fonte. De acordo com a literatura, refatorações para modelos é uma área mais desafiadora do que refatorações tradicionais, uma vez que modelos usualmente possuem múltiplas visões que precisam permanecer sincronizadas e consistentes após a aplicação de refatorações. Por exemplo, na literatura é possível identificar trabalhos que apresentam o estado da arte (MENS; TAENTZER; MüLLER, 2008), taxonomias (MOHAMED MOHAMED ROMDHANI, 2010) e desafios (MENS *et al.*, 2003; MENS; TAENTZER; Müller, 2007; STRAETEN; MENS; BAELEN, 2009; MENS; DEURSEN *et al.*, 2003) em relação a refatorações para modelos.

Tais autores afirmam que a transformação em modelo desempenha um papel fundamental em abordagens que utilizam os princípios de MDE, pois permite a manipulação de modelo de forma totalmente automática. Uma transformação consiste na geração automática de um modelo alvo, tendo como base um modelo fonte, sendo que essa transformação é definida por meio de um conjunto de regras de transformações (MENS; GORP, 2006).

Nos trabalhos de Mens e Gorp (2006), Czarnecki e Helsen (2006) e Biehl (2010) os autores buscam identificar e classificar as transformações de modelos. Algumas das classificações apresentadas por tais autores são citadas de forma resumida, a seguir:

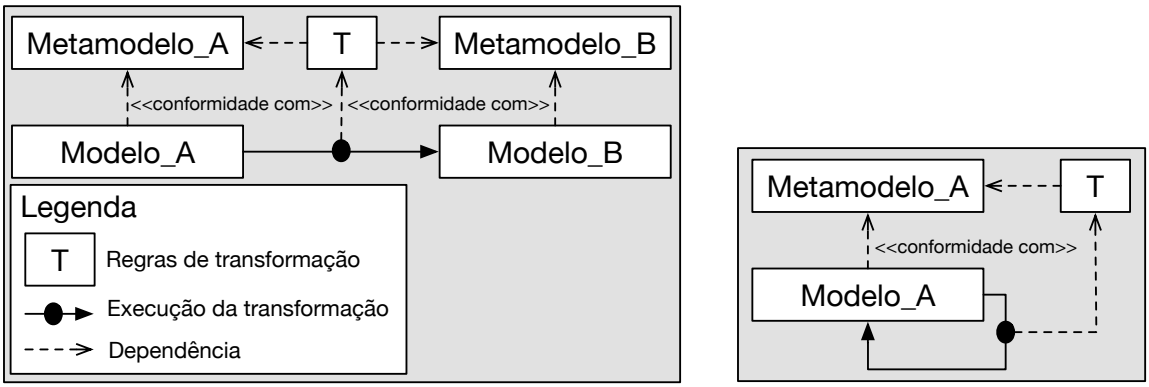
- Vertical ou horizontal: Os modelos fonte e alvo podem estar em um ou mais níveis de abstração. Uma transformação horizontal mantém modelos fonte e alvo no mesmo nível de abstração. Na transformação vertical, existe uma mudança de nível de abstração nos modelos e essa mudança pode ser tanto para aumentar quanto para diminuir o nível de abstração;
- Endógenas ou exógenas: Nas transformações endógenas, os modelos envolvidos são expressos na mesma linguagem de modelagem. Nas transformações exógenas, os modelos que participam da transformação são de linguagens diferentes;
- Bidirecionais: Uma transformação bidirecional pode tanto gerar modelos alvos utilizando como base modelos fontes, quanto gerar modelos fontes utilizando modelos alvos. Em contrapartida, na transformação unidirecional existe apenas um fluxo de execução.

As transformações do tipo classificadas como endógenas ou exógenas (BRAMBILLA; CABOT; WIMMER, 2012) são apresentadas na Figura 3 e, como pode ser observado nela, nas transformações em modelos do tipo endógenas, apenas um modelo e um metamodelo são utilizados; por outro lado, nas transformações do tipo exógenas, os metamodelos alvo e fonte são diferentes. Usualmente, refatorações em modelo são um exemplo de transformações do tipo endógenas, e uma transformação que tem como objetivo transformar uma linguagem para outra linguagem é um exemplo de transformações do tipo exógenas. Transformações em modelos que utilizam apenas um modelo como entrada e geram o mesmo modelo como saída, com algumas modificações, são classificadas como "*in-place*", já as transformações em modelos que utilizam como entrada um modelo e tem como objetivo gerar outro modelo como saída são consideradas "*out-place*".

Transformações endógenas são mais interessantes quando apenas um subconjunto do modelo será afetado pela transformação (BRAMBILLA; CABOT; WIMMER, 2012). Por exemplo, em um editor de modelos, transformações endógenas podem ser utilizadas para definir pequenas mudanças para automatizar tarefas repetitivas durante o desenvolvimento de modelos. Outra aplicação interessante de transformações endógenas é a condução de refatoração em nível de modelos. Um dos principais objetivos desta tese de doutorado são a criação e a utilização de

refatoração em modelos. Assim, maior ênfase será concentrada em transformações de modelos do tipo endógenas no restante desta subseção.

Figura 3 – Diferentes tipos de transformações em modelos.



(a) *exogenous* "*out-place*" (b) *endogenous* "*in-place*"

Fonte: Adaptada de Brambilla, Cabot e Wimmer (2012).

Transformações endógenas geralmente são implementadas utilizando técnicas de "reescrita de grafo", ou como também é conhecida "transformação de grafo" (EHRIG *et al.*, 2006). Na teoria dos grafos, "reescrita de grafo" é um conjunto de regras para reescrever um determinado grafo, por exemplo, dado p : *left-hand side*$(LHS) \rightarrow$ *right-hand side*(RHS), sendo LHS o grafo usado como padrão (no lado esquerdo) e RHS o grafo de substituição (no lado direito da regra). Mais precisamente, o lado esquerdo, LHS, representa todas pré-condições que devem ser satisfeitas antes da execução das regras de transformações. Similarmente, o lado direito, RHS, contém todas pós-condições. As ações que serão executadas pelas regras de transformações são implicitamente definidas tanto no grafo LHS, quanto no grafo RHS. A execução de um conjunto de regra de transformação produz os seguintes efeitos: (*i*) todos elementos que apenas estão contidos no grafo LHS são deletados; (*ii*) todos elementos que apenas estão contidos no grafo RHS são adicionados; (*iii*) todos elementos que estão contidos em ambos os grafos, LHS e RHS, são preservados (EHRIG *et al.*, 2006).

Neste contexto, "reescrita de grafo" é útil para auxiliar na definição de transformações de modelos e metamodelos. Por exemplo, de acordo com Lehnert, Farooq e Riebisch (2012), Fluri e Gall (2006), técnicas de "reescrita de grafo" podem ser aplicadas em qualquer metamodelo e modelos que implementam o padrão MOF, ou seja, KDM, UML, entre outros. Qualquer instância de um metamodelo que implemente o padrão MOF pode ser representada como um grafo da seguinte forma: (*i*) vértices podem ser entendidos como: `EPackage`, `EClass`, `EDataType`, `EEnum`, `EAnotation`, `EOperation`, `EAttribute` e `EEnumLiteral`; (*ii*) arestas podem ser representadas em metamodelo como: `EReference`, `Inheritance`, `EAnnotationLink`. Assim, pode-se definir e realizar evoluções, simulações, refatorações de modelos por meio de técnicas de "reescrita de grafo".

Comumente, transformações em modelos são desenvolvidas utilizando linguagens especializadas, denominadas de linguagens de transformação de modelos. Diversas linguagens de transformação de modelos têm sido propostas atualmente (ALLILAIRE *et al.*, 2006; BIEHL, 2010). Cada linguagem tipicamente fornece um conjunto de características que a torna mais apropriada para o tipo de transformação almejada. No trabalho de Biehl (2010), são citadas várias linguagens de transformação de modelos, o que mostra uma dimensão do número de linguagens para transformação de modelos existentes e disponíveis para o usuário atualmente. Algumas das linguagens citadas são: ATL (ATL, 2015; JOUAULT *et al.*, 2008), *Query/View/Transformation* (QVT) (QVT, 2015), EMF Henshin (HENSHIN, 2015), SmartQVT (SMARTQVT, 2015), ModelMorf (MODELMORF, 2015), Kermeta (KERMETA, 2015), *Epsilon Transformation Language* (ETL) (ETL, 2015), OpenArchitectureWare (OAW) (OPENARCHITECTUREWARE, 2015), VIATRA (VIATRA, 2015), AndroMDA (ANDROMDA, 2015) e Fujaba Transformations (FUJABA, 2015).

Nos trabalhos de Biehl (2010), Mens e Gorp (2006), Allilaire *et al.* (2006), os autores buscam levantar características importantes tanto para classificar as transformações de modelos, quanto as linguagens de transformação de modelos para realização dessas transformações. Similarmente, Huber (2008) busca avaliar diferentes ferramentas e linguagens de transformação de modelos. O estudo conclui que nenhuma ferramenta é melhor do que a outra, mas que uma linguagem pode ser mais adequada para um problema específico do que outras linguagens. Entre as várias características utilizadas pelos autores na classificação das linguagens de transformação, a de maior importância é quanto ao paradigma da linguagem. Segundo Mens e Gorp (2006), a maior distinção entre os mecanismos de transformação de modelos é quanto ao seu paradigma. Os principais paradigmas das linguagens de transformações de modelos são:

- Imperativo: Linguagens imperativas especificam um fluxo de controle sequencial e fornecem meios para descrever a forma como a linguagem de transformação de modelo supostamente deve ser executada (MENS; GORP, 2006). As construções e conceitos de linguagens de transformações imperativas são semelhantes às linguagens de programação de propósito geral, como Java ou C;

- Declarativo: Linguagens declarativas não oferecem um fluxo de controle explícito. Em vez de se concentrar em como a transformação deve ser executada, o foco é sobre o que deve ser mapeado pela transformação (MENS; GORP, 2006). Transformações de modelos declarativas descrevem a relação entre os metamodelos fonte e alvo, e essa relação pode ser interpretada bidirecionalmente. Em geral, tais linguagens são compactas e as descrições de transformações são geralmente curtas e concisas (BIEHL, 2010; MENS; GORP, 2006);

- Híbrido: Linguagens híbridas oferecem tanto as construções de linguagem imperativa, quanto as construções de linguagem declarativa;

- Transformação Direta: Linguagens de programação de uso geral e bibliotecas para ler e gravar os dados dos modelos são utilizadas para implementar as transformações de modelos (HUBER, 2008). A vantagem da transformação direta é que os programadores não precisam aprender uma nova linguagem. Mas, por outro lado, as implementações tendem a se tornar maiores (BIEHL, 2010).

2.3.2 *ATLAS* Transformation Language *(ATL)*

A ATLAS *Transformation Language* (ATL) (ATL, 2015) é uma linguagem de transformação de modelo híbrida, ou seja, a linguagem contém uma mistura de construções declarativas e imperativas. O uso do estilo declarativo é encorajado por vários autores (ALLILAIRE *et al.*, 2006; JOUAULT; KURTEV, 2006; JOUAULT *et al.*, 2008), pois permite uma implementação mais objetiva e mais simples. No entanto, a definição de transformações complexas utilizando apenas construções declarativas pode ser uma tarefa difícil. Nesse caso, os desenvolvedores podem recorrer aos recursos imperativos da linguagem (ALLILAIRE *et al.*, 2006).

A ATL possui uma sintaxe abstrata definida utilizando um metamodelo. Isso significa que cada transformação definida em ATL é de fato um modelo. Uma transformação ATL pode ser decomposta em três partes: um *header*, *helpers* e um conjunto de *rules*. O *header* (cabeçalho) é utilizado para declarar informações gerais, tais como o nome do módulo (nome da transformação que deve coincidir com o nome do arquivo .atl), os metamodelos de entrada e de saída e a importação de bibliotecas necessárias. Os *helpers* são sub-rotinas usadas para evitar a redundância de código. Pode-se imaginar um *helper* como um método igual ao que se tem em linguagens de programação. Já as *rules* (regras) são as principais definições das transformações ATL, porque elas descrevem como os elementos de saída (em conformidade com o metamodelo de saída) são produzidos a partir de elementos de entrada (em conformidade com o metamodelo de entrada). Elas são constituídas por ligações, cada uma expressando um mapeamento entre um elemento de entrada e um elemento de saída (ATL, 2015).

O funcionamento da ATL se dá da seguinte forma. Primeiro o código ATL deve ser compilado e, em seguida, executado pelo mecanismo de transformação ATL. A ATL oferece suporte dedicado para rastreabilidade e a ordem de execução das regras é determinada automaticamente, com exceção das *Lazy Rules*, que precisam ser chamadas explicitamente no código da ATL. Os *helpers* fornecem construções imperativas às transformações. A ATL também possui um módulo denominado ATL *Refining* que suporta transformações do tipo *endogenous* "*in-place*" (ver Figura 3(b)).

A ATL foi escolhida para a implementação deste trabalho considerando vários aspectos. A ATL está integrada na plataforma Eclipse, o que provê uma série de recursos padrões para o desenvolvimento (*syntax highlighting* e *debugger*). A ATL é parte do projeto M2M da ferramenta Eclipse e possui um grupo de discussão ativo, constantemente atualizado. Vários exemplos e

diversos estudos de casos aplicados até mesmo na indústria[1] utilizam a ATL e por se tratar de uma ferramenta de fácil uso, a partir de premissa de conhecimento de linguagem e do metamodelo, traz como vantagem ao processo: baixo custo, por ser uma ferramenta livre, e alta flexibilidade, por facilitar grandes alterações na transformação diretamente usando a interface do editor de regras ATL (SALEM; GRANGEL; BOUREY, 2008). Além disso, a ATL é uma das linguagens de transformações mais madura no contexto da MDE (BRUNELIÈRE *et al.*, 2010).

2.4 Modernização Dirigida a Arquitetura

O crescente interesse na MDE motivou o *Object Management Group* (OMG) a lançar a iniciativa denominada Modernização Dirigida a Arquitetura (do inglês - *Architecture-Driven Modernization* (ADM)), cujo objetivo foi estabelecer metamodelos padronizados para auxiliar todo o processo da reengenharia de software. Tal iniciativa foi motivada devido ao alto número de projetos de reengenharia de software que não obtiveram sucesso (SNEED, 2005; DEMEYER, 2005). Como resultado desse esforço, os conceitos da ADM foram criados, os quais possuem como objetivo revitalizar/modernizar softwares existentes[2] com a utilização de metamodelos padronizados, empregando os princípios da abordagem Arquitetura Dirigida a Modelo (do inglês - *Model-Driven Architecture* (MDA)) (ver Seção 2.2).

Entre os termos mais recentes relacionados à reengenharia de software, a ADM se destaca. De acordo com o OMG (OMG, 2015), o objetivo da ADM não é substituir o processo tradicional da reengenharia de software, pelo contrário, a ADM almeja auxiliar e melhorar o processo de reengenharia de software por meio da utilização dos princípios da MDA. A ADM consiste em uma adaptação do modelo de ferradura tipicamente conhecido em reengenharia de software (i.e, o modelo de ferradura, basicamente contém dois lados, esquerdo, direito e uma "ponte" ligando os dois lados). Na Figura 4, é apresentado o modelo de ferradura adaptado para a ADM e é importante observar que essa figura contém todas as fases e "palavras-chaves" tradicionais que são encontradas na reengenharia de software tradicional e em MDA, tais como: Modelo Específico de Plataforma (do inglês - *Platform-Specific Model* (PSM)), Modelo Independente de Plataforma (do inglês - *Platform-Independent Model* (PIM)) e Modelo Independente de Computação (do inglês - *Computation-Independent Model* (CIM)). As fases tradicionais da reengenharia de software adaptadas para a ADM são:

- **Engenharia Reversa**: essa fase tem como entrada um sistema que será modernizado, e, posteriormente, esse sistema será transformado em um PSM. Além disso, o PSM é utilizado como entrada para a geração do PIM, que, no contexto desta tese, consiste em uma instância do metamodelo denominado KDM (ver Seção 2.5);

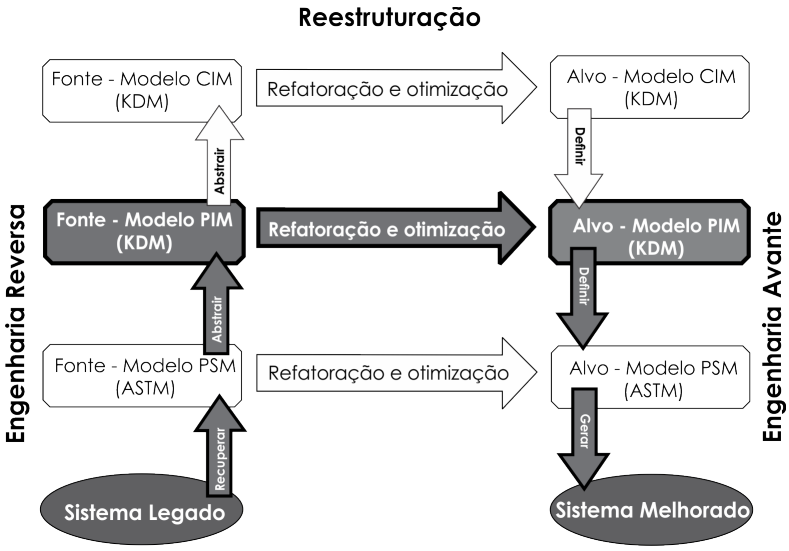
[1] `https://www.eclipse.org/forums/index.php?t=threadfrm`$_id = 241$

[2] No contexto desse documento, softwares existentes e sistemas legados são utilizados de forma intercambiáveis

- **Reestruturação**: nessa fase, um conjunto de reestruturação/refatoração pode ser aplicado sobre uma instância do metamodelo KDM por meio de transformações de modelo para modelo (do inglês - *Model-To-Model* (M2M));

- **Engenharia Avante**: nesta fase um novo código-fonte do sistema modernizado é gerado automaticamente por meio de transformações de modelo para código (do inglês - *Model-To-Code* (M2C)).

Figura 4 – Modelo de ferradura adaptada para a ADM.



Fonte: Adaptada de ADM (2012).

Durante o processo da ADM, todos os modelos (i.e., PSM, PIM e CIM) podem estabelecer transformações/refatorações entre si, como ilustrado na Figura 4. Geralmente, tais transformações são executadas por meio de linguagens de transformações. Como salientado no Capítulo 2, Seção 2.3.1, usualmente essas transformações são implementadas utilizando diferentes linguagens de transformações, que podem ser declarativas, imperativas ou híbridas.

É importante também destacar que a ADM não tem como intuito apenas seguir todos os princípios da abordagem MDA (ADM, 2012). Um dos principais objetivos da ADM é definir um conjunto de metamodelos padronizados para lidar com diferentes desafios que são encontrados hoje em dia na reengenharia de software. Dessa forma, em Novembro de 2003, a *Architecture-Driven Modernization Task Force* (ADMTF) criou uma *Request-for-Proposal* (RFP), que, por sua vez descrevia um conjunto de metamodelos. Tais metamodelos são: (*i*) Knowledge Discovery Metamodel (KDM), maiores informações sobre esse metamodelo são apresentadas na seção 2.5, (*ii*) *Structured Metrics metamodel* (SMM), que é um metamodelo para representar e definir métricas e resultados de medições, (*iii*) *ADM Pattern Recognition* (ADMPR), que facilita a busca de padrões em um software, (*iv*) *ADM Visualization Specification* (ADMVS), que tem como objetivo representar visualmente metadados de uma aplicação representada em KDM e (*v*)

ADM Refactoring Specification (ADMRS), que almeja definir um metamodelo padronizado para especificar e definir refatorações, utilizando outros metamodelos da ADM, como por exemplo o KDM. O estado atual de cada metamodelo pode ser visto na Tabela 1 (ADM, 2012), a qual mostra que alguns metamodelos ainda encontram-se em fase de desenvolvimento e outros já foram finalizadas e disponíveis pelo OMG. É importante salientar que durante a condução desta pesquisa o metamodelo ADMRS não havia sido definido pelo OMG. Dessa forma, uma proposta inicial de uma metamodelo para especificar e reutilizar refatoração foi desenvolvido nesta tese, ver Capítulo 5.

Tabela 1 – Estado atual dos metamodelos da ADM.

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

É importante destacar que a abordagem proposta nesta tese se concentra no metamodelo KDM. Consequentemente, é de suma importância o entendimento desse metamodelo, por isso, ele é mais detalhado neste capítulo. KDM é um metamodelo que pode ser utilizado para representar todos os artefatos de um determinado software existente, por exemplo, nele há metaclasses específicas para representar desde código-fonte até a arquitetura de um determinado software. O KDM é um metamodelo de representação intermediária comum para sistemas existentes e seus ambientes operacionais. Utilizando esse metamodelo para sistemas existentes, é possível trocar representações do sistema em modelo entre plataformas e linguagens com a finalidade de analisar, padronizar e transformar/refatorar os sistemas existentes (ADM, 2012).

A ideia por trás do KDM é que a comunidade comece a criar analisadores sintáticos (do inglês - *parsers*) para diferentes linguagens de programação, que transformem os códigos-fontes em instâncias do metamodelo KDM. Como resultado, qualquer técnica, ferramenta e abordagem que utilize o KDM como o artefato de entrada pode ser considerada uma técnica, ferramenta e/ou abordagem independente de linguagem e de plataforma. Por exemplo, um catálogo de refatoração para o KDM (DURELLI *et al.*, 2014a; DURELLI *et al.*, 2014c) pode ser usado para refatorar vários sistemas independentemente da linguagem de programação. Maiores informações sobre o KDM, bem como sobre seus pacotes, metaclasses e metarrelacionamentos são apresentados a seguir.

2.5 Knowledge Discovery Metamodel (KDM)

Knowledge Discovery Metamodel (KDM) é um metamodelo que representa artefatos de software, seus elementos, associações e ambientes operacionais. O KDM tem como principal objetivo permitir que os engenheiros de modernização criem ferramentas para auxiliar a modernização de software, as quais sejam independentes de plataforma e linguagem (KDM, 2015; PÉREZ-CASTILLO; GUZMAN; PIATTINI, 2011; PÉREZ-CASTILLO; GUZMÁN; PIATTINI, 2011). Além disso, o KDM facilita e assegura a interoperabilidade e a troca de dados entre diferentes ferramentas.

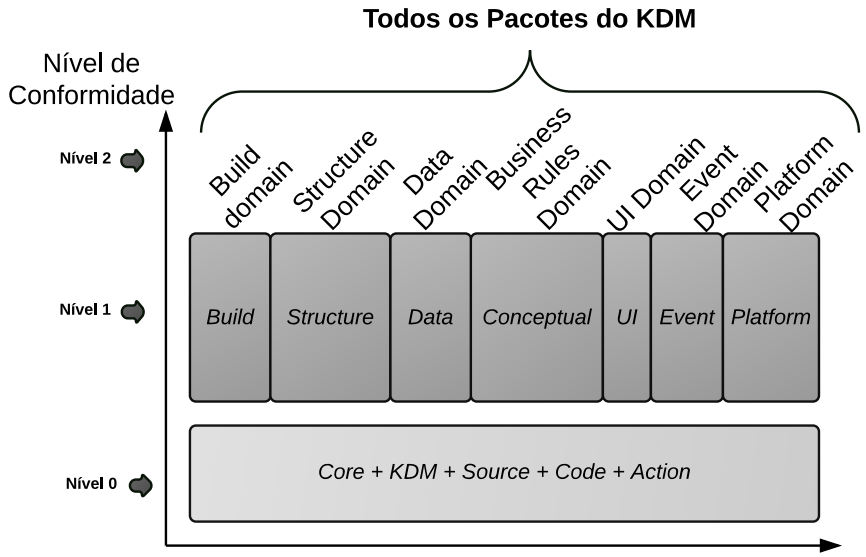
Um problema tradicional facilmente identificado em várias ferramentas que lidam com a reengenharia de software é que tais ferramentas analisam diversos artefatos de um determinado software (por exemplo, código-fonte, banco de dados, *scripts*, etc.) para obter conhecimentos explícitos, com o intuito de realizar transformações/refatorações (ROSENBERG, 1996; CANFORA; PENTA; CERULO, 2011). Como consequência, cada ferramenta gera e analisa tais conhecimentos de forma implícita. Assim, os conhecimentos gerados são restritos a uma específica linguagem de programação, e/ou a uma plataforma. Como resultado, tais restrições podem criar dificuldades com relação à interoperabilidade entre diferentes ferramentas. O KDM fornece uma estrutura que busca facilitar a troca de dados entre diversas ferramentas. Além disso, possui um conjunto de metaclasses e uma estrutura padronizada que fornecem meios para especificar desde artefatos físicos até artefatos lógicos de um determinado sistema de software. Em virtude dessa padronização, todas as técnicas/abordagem/ferramentas que utilizam o KDM como entrada podem ser consideradas independentes de plataforma e linguagem, aumentando, assim, a interoperabilidade e o reúso. Em 2012, o KDM tornou-se *International Standards Organization* (ISO) (ISO/IEC19506, 2012) como uma estrutura que facilita a troca de dados entre as diversas ferramentas. O KDM é definido via *Meta-Object Facility* (MOF) (MOF, 2015) e estabelece o formato de troca de dados via *XML Metadata Interchange* (XMI), o qual é denominado KDM XMI *schema*.

Resumidamente, as principais metas do KDM são (ADM, 2012): (*i*) representa artefatos de um sistema legado como entidades, relacionamentos e atributos; (*ii*) suporta uma variedade de plataformas e linguagens; (*iii*) define uma terminologia unificada para artefatos de sistemas legados; (*iv*) representa estruturas lógicas e físicas de sistemas legados; (*v*) permite a modificação/refatoração de sistemas legados utilizando os princípios da MDA; (*vi*) facilita o rastreamento de mudança entre artefatos; (*vii*) facilita a sincronização de estruturas lógicas e físicas de um determinado sistema legado; (*viii*) contém metaclasses para representar desde código-fonte até metaclasses para representar elementos arquiteturais de um determinado sistema legado.

De acordo com Pérez-Castillo, Guzman e Piattini (2011), o KDM busca cobrir um amplo escopo para abranger um conjunto diversificado de aplicações, plataformas e linguagens de programação, além de almejar fornecer a capacidade de troca de metadados entre diversas ferramentas e, assim, facilitar a cooperação entre fornecedores para integrar e aumentar a

interoperabilidade de diferentes abordagens, técnicas, algoritmos, etc. A fim de alcançar essa interoperabilidade e, especialmente, a integração de informações sobre diferentes facetas de um determinado sistema a partir de múltiplas ferramentas, o KDM define vários níveis de conformidade, aumentando a probabilidade de que duas ou mais ferramentas apoiem o mesmo metamodelo. Além disso, o KDM também é estruturado de forma modular, seguindo o princípio da separação de interesse, com a capacidade de representar partes heterogêneas de um sistema. A separação de interesses no contexto do metamodelo KDM é alcançada por meio de pacotes, como apresentado na Figura 5. Cada pacote está em um nível de conformidade e visa definir um ponto de vista arquitetural do sistema. Em outras palavras, cada pacote do KDM constitui uma determinada ontologia para descrever e representar a grande maioria dos artefatos de sistemas de software existentes. Por exemplo, os pacotes `Code` e `Action` contêm metaclasses que representam o código-fonte de um sistema, tais como variáveis, procedimentos/métodos/funções, chamadas para métodos, etc. Similarmente, o pacote `Structure` contém metaclasses para representar elementos arquiteturais do sistema, tais como, componentes, camadas, subcomponentes, etc. O pacote `Conceptual` possui metaclasses para definir regras de negócio do sistema.

Figura 5 – Pacotes e nível de conformidade do metamodelo KDM.



Fonte: Adaptada de KDM (2015).

Da perspectiva de um engenheiro de modernização, essa separação de interesse do KDM, por meio de pacotes, significa que o engenheiro só precisa se preocupar com os pacotes do KDM que considerar necessários para as suas atividades de modernização, por exemplo, uma determinada abordagem pode necessitar apenas do pacote `Code` e `Action`, enquanto outra abordagem pode utilizar apenas o pacote responsável por definir elementos arquiteturais. Se essas abordagens forem evoluídas ao longo do tempo e necessitarem de outros pacotes do KDM, pacotes podem ser adicionados ao repertório da abordagem/ferramenta, conforme necessário.

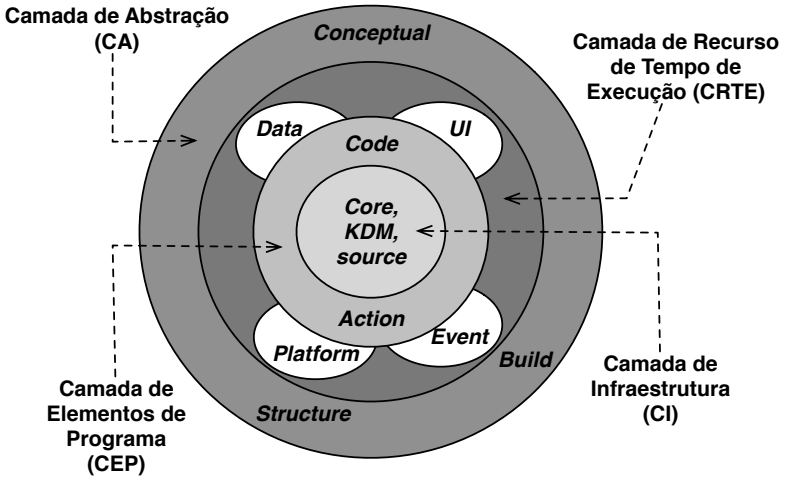
Como observado na Figura 5, o KDM possui três níveis de conformidade, nível 0, nível

1 e nível 2. Cada nível é apresentado a seguir:

- **Nível 0**: nesse nível, são definidos os seguintes pacotes do KDM: (*i*) `Core`, (*ii*) `kdm`, (*iii*) `Source`, (*iv*) `Code` e (*v*) `Action`. Esse nível de conformidade representa um denominador comum que pode servir como uma base para a interoperabilidade entre diferentes categorias de ferramentas que utilizem o metamodelo KDM. Para que uma ferramenta esteja em conformidade com o **Nível 0**, ela deve fornecer completo suporte para todas as metaclasses que foram definidas nos pacotes `Core`, `kdm`, `Source`, `Code` e `Action`;
- **Nível 1**: nesse nível, os pacotes definidos no **Nível 0** são estendidos para representar outros artefatos de um determinado sistema. Além disso, o **Nível 1** define os seguintes pacotes: (*i*) `Build`, (*ii*) `Structure`, (*iii*) `Data`, (*iv*) `Conceptual`, (*v*) `UI`, (*vi*) `Event` e (*vii*) `Platform`. Para que uma ferramenta esteja em conformidade com o **Nível 1** ela deve fornecer suporte para todos os pacotes do **Nível 0** e pelo menos um dos pacotes do **Nível 1**;
- **Nível 2**: esse nível é a união de todos os pacotes definidos no nível anterior. Para que uma ferramenta esteja em conformidade com o **Nível 2**, ela deve fornecer suporte para todos os pacotes do **Nível 1** e pelo menos um do **Nível 2**.

Todos os pacotes do metamodelo KDM apresentados na Figura 5 são organizados em quatro camadas de abstração: (*i*) Camada de Infraestrutura (CI): do inglês *Infrastructure Layer*; (*ii*) Camada de Elementos de Programa (CEP): do inglês *Program Elements Layer*; (*iii*) Camada de Recurso de Tempo de Execução (CRTE): do inglês *Runtime Resource Layer*; (*iv*) Camada de Abstração (CA): do inglês *Abstraction Layer*. Essas quatro camadas estão apresentadas esquematicamente na Figura 6.

Figura 6 – Camadas e pacotes do KDM.



Fonte: Adaptada de KDM (2015).

A camada CI contém três pacotes, e são eles: (*i*) `Core`, (*ii*) `"kdm"` e (*iii*) `Source`. Os dois primeiros pacotes, `Core` e `"kdm"`, representam a infraestrutura básica para outros pacotes do

KDM e definem metaclasses e relacionamentos básicos. O pacote `Source` define o `Inventory Model`, o qual representa artefatos de software e mantém a rastreabilidade entre eles.

A camada CEP possui dois pacotes: (*i*) `Code` e (*ii*) `Action`, os quais coletivamente definem o `Code Model` que contém metaclasses para representar artefatos no âmbito da implementação. O pacote `Code` apresenta um conjunto de metaclasses para representar a estrutura de um determinado programa e seus relacionamentos, já o pacote `Action` possui metaclasses para descrever o comportamento e o fluxo de dados de um programa.

A camada CRTE compreende quatro pacotes: (*i*) `Data`, (*ii*) `Platform`, (*iii*) `Event` e (*iv*) `UI`. Coletivamente, tais pacotes representam a estrutura e o comportamento de recursos de tempo de execução do sistema. Tais pacotes são diretamente instanciados por meio da definição de recursos, abstração do `Code Model`, ou, ainda, são manualmente instanciados pelo engenheiro de modernização. A rastreabilidade entre os elementos abstraídos e os elementos físicos (por exemplo, código-fonte) é mantida pelo meta-atributo, denominado `implementation`. Finalmente, a camada CA engloba três pacotes: (*i*) `Conceptual`, (*ii*) `Structure` e (*iii*) `Build` possuindo metaclasses para representar o maior nível de abstração de um sistema, por exemplo, a estrutura do sistema, regras de negócios, documentações do sistema, etc.

Uma característica importante de ser observada e ressaltada é que todas as camadas do KDM interagem, significando que todas elas são conectadas de alguma forma[3] e, como consequência, se uma mudança/refatoração for realizada em uma camada específica, a mudança/-refatoração deverá ser propagada para outras camadas com o intuito de manter todas as camadas sincronizadas e consistentes preservando, assim, a estrutura sintática e semântica do KDM. Nas próximas seções, são apresentados os principais pacotes do metamodelo KDM que compõem o contexto para o desenvolvimento deste trabalho.

2.5.1 Pacote Code

O pacote `Code` define um conjunto de metaclasses, cujo propósito é representar unidades de programa em nível de implementação e as suas associações. O pacote também inclui metaclasses que representam elementos de programa comuns e suportados por várias linguagens de programação, como: tipos de dados, classes, procedimentos, macros, protótipos e *templates*.

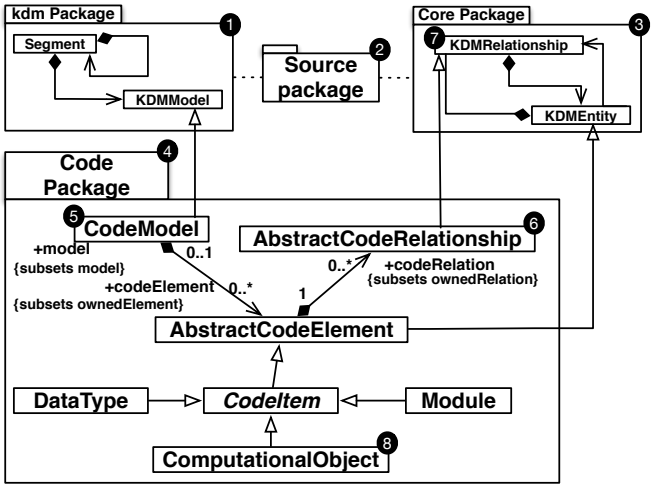
Em uma determinada instância do KDM, cada elemento do pacote `code` representa alguma construção em uma linguagem de programação, determinada pela linguagem de programação utilizada no sistema. Na Figura 7, um trecho do `CodeModel`[4] é retratado.

A metaclasse `CodeModel` representa um contêiner para outras instâncias de elementos

[3] Essas conectividades entre os elementos de cada camada são mantidas por um conjunto de meta-atributo, por exemplo, o meta-atributo *implementation*.

[4] O diagrama de classes do `CodeModel`, mostrado aqui, só representa o conjunto de metaclasses e os seus respectivos relacionamentos lógicos (para informações completas, verifique a especificação do KDM (KDM, 2015).

Figura 7 – Diagrama de classes - `CodeModel`



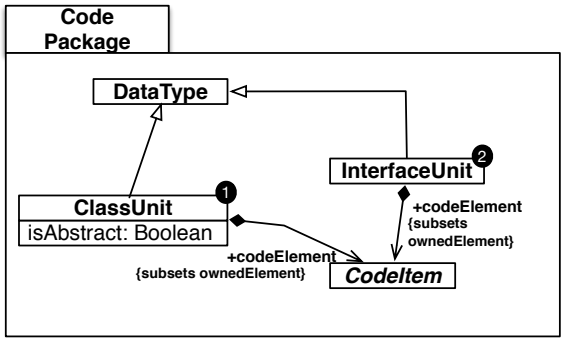
Fonte: Adaptada de KDM (2015).

do tipo `Code`. O pacote `Code` ❹ depende dos outros pacotes `kdm` ❶, `Source` ❷ e `Core` ❸. A metaclasse `CodeModel` ❺ é um modelo que possui coleções de fatos sobre o sistema de software, correspondentes ao domínio `Code` e ela possui uma associação `codeElement:AbstractCodeElement[0..*]`, permitindo a adição de novos elementos de código, por exemplo, métodos, atributos, etc. A metaclasse `AbstractCodeRelationship` ❻ representa qualquer relacionamento determinado por uma linguagem de programação. Por sua vez, a metaclasse `ComputationalObject` representa os elementos determinados pela linguagem de programação, que descreve certos objetos computacionais em tempo de execução, por exemplo, métodos e variáveis.

O pacote `Code` compreende um total de 24 metaclasses, que são um arranjo de abstrações para representar toda a estrutura estática (ou grande maioria) de um determinado código-fonte, dada uma linguagem de programação, seja ela procedural ou orientada a objetos (KDM, 2015). Na Tabela 2, algumas metaclasses são apresentadas. É visto que algumas metaclasses podem ser diretamente elucidadas e mapeadas, como por exemplo, "classes" e "interfaces" construções facilmente encontradas em linguagens orientadas a objetos podem ser facilmente mapeada para as metaclasses denominada `ClassUnit` e `InterfaceUnit`, respectivamente. Um mapeamento mais completo entre elementos estruturais e metaclasses do KDM pode ser identificado em Santos (2014) e no Capítulo 4. Uma representação dessas metaclasses, bem como seus relacionamentos são apresentados em diagrama de classe na Figura 8.

`ClassUnit` e `InterfaceUnit` representam "classes" e "interfaces" que são definidas por usuários de linguagens orientadas a objetos. Essas metaclasses possuem características e relacionamentos similares, como observado na Figura 8 ❶ e ❷. Uma das diferenças que pode ser destacada é que a metaclasse `ClassUnit` contém um meta-atributo `isAbstract : Boolean`, o qual é utilizado para especificar se uma classe é ou não abstrata. `ClassUnit` e `InterfaceUnit` po-

Figura 8 – Diagrama de classes elucidando as metaclasses `ClassUnit` e `InterfaceUnit`



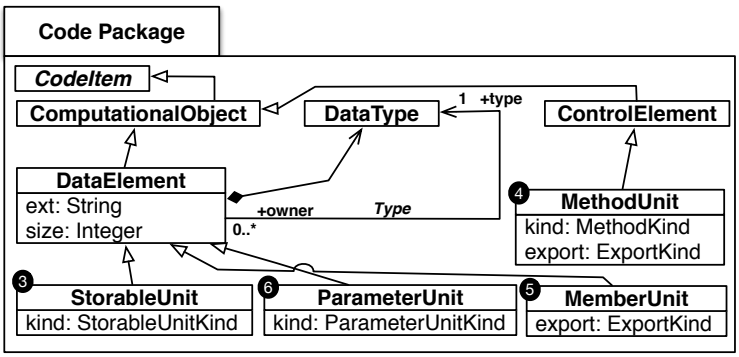
Fonte: Adaptada de KDM (2015).

Tabela 2 – Metaclasses para modelagem de estruturas estáticas do código-fonte.

Elemento do Código-Fonte	metaclasses do KDM
Classe	`ClassUnit`
Interface	`InterfaceUnit`
Método	`MethodUnit`
Atributo	`StorableUnit`
Variável Local	`MemberUnit`
Parâmetro	`ParameterUnit`
Associação	`KDMRelationShip`

dem conter uma coleção de elementos que seja do tipo `CodeItem`, por exemplo, `StorableUnit` ou `MethodUnit`. Além disso, tais metaclasses possuem uma meta-associação denominada `codeElement` : `CodeItem`$[0..*]$, que é utilizada para agrupar todos os membros da classe, por exemplo, construtores, métodos, atributos, etc. Na Figura 9, são apresentados os meta-atributos e of metarrelacionamentos das metaclasses `StorableUnit` ❸, `MethodUnit` ❹, `ParameterUnit` ❻ e `MemberUnit` ❺.

Figura 9 – Diagrama de classes elucidando as metaclasses `StorableUnit`, `MethodUnit`, `ParameterUnit` e `MemberUnit`



Fonte: Adaptada de KDM (2015).

`StorableUnit` representa um atributo em um sistema de software - um objeto computacional para que diferentes valores do mesmo tipo de dados possam ser associados. Ele é usado

para representar as variáveis globais e locais. Ele engloba um meta atributo `String` usado para definir o nome das variáveis. `StorableUnit` também tem a associação `type : DataType[1]`, a qual é herdada da metaclasse `DataElement`, que é utilizado para especificar o tipo da variável (*int, char, boolean, numeric*, etc). Ele também tem uma enumeração `kind : StorableUnit`, que descreve várias propriedades comuns de um `StorableUnit` relacionado com o seu ciclo de vida, por exemplo, sua visibilidade (*private, public, protected*, etc).

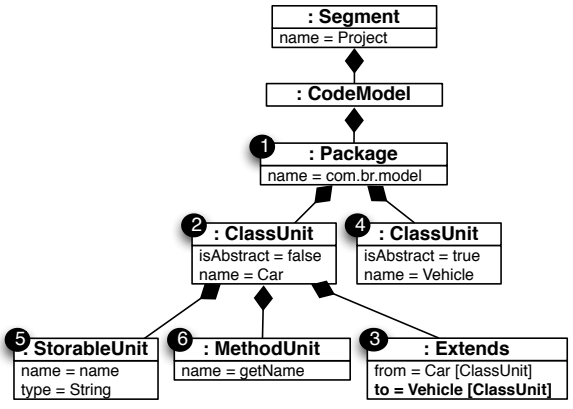
`MethodUnit`, como o próprio nome sugere, representa métodos que são identificados em `ClassUnit` ou `InterfaceUnit`. Também é usado para representar construtores e destrutores. Possui como meta-atributos: (*i*) `name : String`, (*ii*) `kind : MethodKind` e (*iii*) `export : ExportKind`. O primeiro é usado para descrever o nome de um método; o segundo é uma enumeração que define especificações adicionais do tipo de método, ou seja, é possível especificar se a instância do método é um construtor, destrutor, ou um método normal; o último representa a visibilidade do método (*private, public, protected*, etc).

A fim de entender como o KDM é utilizado para representar estruturas em um determinado programa, no Código-fonte 2 é mostrado um exemplo simplificado escrito em Java. O correspondente KDM, embora simplificado, é apresentado na Figura 10. Por questões de simplicidade e para facilitar o entendimento, essa figura ilustra a instância do KDM em forma de um diagrama de objetos; é possível notar que tal diagrama representa o código-fonte como uma árvore, na qual cada nó representa uma metaclasse do KDM. Como pode ser visto na Figura 10, a metaclasse raiz é `Segment`, que é um recipiente para um conjunto significativo de fatos sobre um sistema de software existente. Cada `Segment` pode incluir uma ou mais instâncias de modelos do KDM, como `CodeModel` e `StructureModel`.

Código-fonte 2: Simples código em Java.

```
1  ❶ package model;
2  ❷ public class Car ❸ extends
3  ❹ Vehicle{
4  ➥❺ private String name;
5  ➥❻ public String getName(){
6  ...
7  }
8  }
9
```

Figura 10 – Instância KDM correspondente ao Código-fonte 2.



Fonte: Elaborada pelo autor.

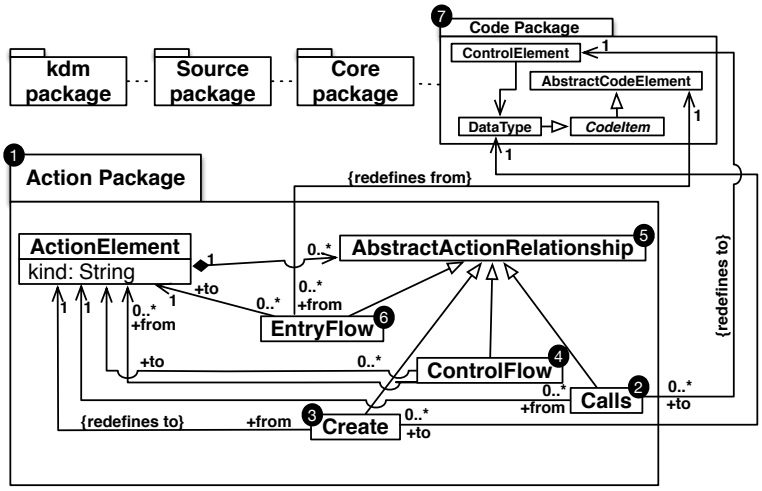
Analisando tanto o Código-fonte 2, quanto a Figura 10, é evidente que cada estrutura estática do código-fonte tem uma metaclasse específica em KDM para representá-la. Por exemplo, a declaração *package model* na Linha 1 do Código-fonte 2 ❶ é representada em KDM pela

metaclasse `package`, como visto na Figura 10 ❶. Posteriormente, como apresentado no Código-fonte 2 ❷, uma classe `Car` é declarada. Essa classe herda características da classe `Vehicle`, no Java isso é feito por meio da palavra-chave `extends` seguida do nome de uma classe, conforme observado no Código-fonte 2 ❸ e ❹. A metaclasse `Extends` representa o conceito de herança em KDM. Como mostrado na Figura 10 ❸, a metaclasse `Extends` possui duas associações, `to` e `from`; a primeira representa a classe pai (*super class*), e a última a classe filha(*sub-class*). Neste contexto, a classe `Car` é a classe filha de `Vehicle`, como mostrado na Figura 10 ❹. Finalmente, o atributo `name` e o método `getName`() (ver Código-fonte 2 ❺ e ❻, respectivamente) são mapeados para os correspondentes elementos do KDM, `StorableUnit` e `MethodUnit` (ver Figura 10 ❺ e ❻). Apenas metaclasses utilizadas para representar estruturas e construções estáticas foram demonstradas. No entanto, o KDM abrange um pacote que permite a representação de construções dinâmicas, em outras palavras, o pacote *Action* contém metaclasses, cujas finalidades são permitir e representar comportamento com relação à execução. Na seção a seguir, mais informações sobre esse pacote são apresentadas.

2.5.2 Pacote Action

O pacote `Action` define um conjunto de metaclasses, com o propósito de representar descrições de comportamento em nível de implementação estabelecida por linguagens de programação, por exemplo, declarações, operadores, condições e as suas associações. A Figura 11 ❶ mostra o pacote `Action` e algumas de suas metaclasses. Nota-se que esse pacote estende o pacote `Code` (ver a Figura 11 ❼).

Figura 11 – Diagrama de classes ilustrando o pacote `Action`.



Fonte: Adaptada de KDM (2015).

O pacote `Action` é composto por 11 diagramas de classes e também depende dos pacotes `Core`, `kdm` e `Source`, e, principalmente, do pacote `Code`. No entanto, o pacote `Action` segue o padrão uniforme para os modelos KDM e estende o KDM com metaclasses específicas

relacionadas com o comportamento do nível de implementação. O pacote `Action` se desvia de um padrão uniforme para os modelos KDM porque ele não define um modelo KDM separado, mas estende o pacote `Code`. Por isso, cada metaclasse do pacote `Action` é uma subclasse de `AbstractCodeElement`, conforme destacado na Figura 11. O pacote `Action` define a maioria das metaclasses que têm como objetivo representar comportamentos para as construções estáticas definidas no pacote `Code`. Assim, ambos os pacotes constituem a CEP, como mostrado na Figura 6.

A metaclasse `AbstractionActionRelationship` apresentada na Figura 11 ❺ é a metaclasse pai usada para representar várias relações que se originam a partir de um `ActionElement`. Além disso, essa metaclasse `AbstractionActionRelationship` possui metaclasses específicas; algumas delas estão representadas na Figura 11, por exemplo, as metaclasses `Calls` ❷, `Creates` ❸, `ControlFlow` ❹ e `EntryFlow` ❻.

O relacionamento `Calls` corresponde a uma chamada para um procedimento, um método estático, um método não-estático de uma instância particular de um objeto, um método virtual, ou um elemento de interface. `Calls` possui duas associações, são elas: `ActionElement`[1] e `ControlElement`[1]. A primeira representa o elemento de ação a partir do qual a relação chamada origina, e a segunda associação representa o elemento alvo.

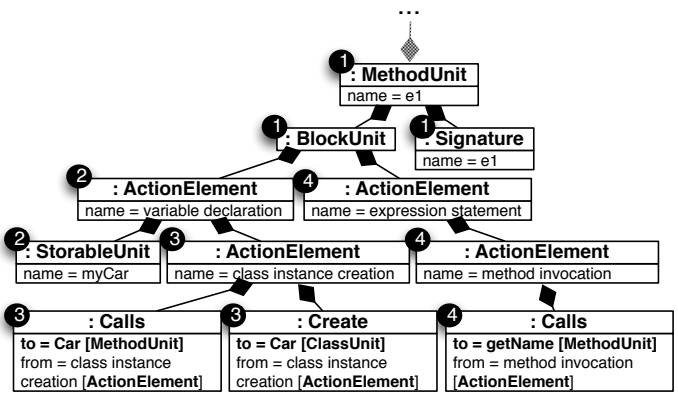
A metaclasse `Creates` representa uma associação entre um elemento de ação que "cria" uma nova instância de um determinado elemento de dados. Por exemplo, em Java, essa metaclasse corresponde à palavra-chave `new`, utilizada para instanciar um novo objeto. `Creates` também possui duas associações: `ActionElement`[1] e `DataType`[1]. Similar a metaclasse `Calls`, a primeira associação representa o elemento que possui o relacionamento e a segunda representa o elemento de dados (objeto), que é instanciado pelo `ActionElement`.

O `ControlFlow` é um elemento de modelagem genérica que representa relação de fluxo de controle entre dois `ActionElements`. Além disso, é uma submetaclasse com elementos de modelagens mais específicas. O `EntryFlow` é um elemento de modelagem que representa um fluxo inicial de controle em um elemento KDM. O relacionamento `EntryFlow` é usado de uma maneira uniforme para descrever os pontos de entrada para outros elementos de código KDM.

Figura 12 – Instância KDM correspondente ao Código-fonte 3.

Código-fonte 3: Método `e1` ilustrando como o Pacote `Action` funciona.

```
1  ...
2  public void e1 ❶(){
3  Car myCar ❷ = new Car() ❸;
4  ➥myCar.getName() ❹;
5  }
6  ...
7
```



Fonte: Elaborada pelo autor.

A fim de compreender como o pacote `Action` é usado no KDM, no Código-fonte 3 é mostrado um simples método implementado em Java. Nesse método é criada uma instância de `Car` e seu método de acesso é invocado. Uma possível correspondente instância do KDM simplificada é apresentada na Figura 12. Nota-se que o diamante em destaque na cor cinza e anexado com três pontos (...) ilustra que outras metaclasses não são mostradas com o intuito de simplificar a figura.

As três primeiras metaclasses mostradas nessa hierarquia são `MethodUnit`, `BlockUnit` e `Signature` conforme destacado na Figura 12 ❶. Essas três metaclasses basicamente representam uma declaração e a assinatura de um determinado método, no caso `e1()`. Mais especificamente, a metaclasse `MethodUnit` é usada para representar o método `e1()` como mostrado tanto no Código-fonte 3 ❶, quanto na Figura 12 ❶. `BlockUnit` representa blocos lógicos e físicos relacionados a `ActionElement`, ou seja, o escopo do método representado por {...}. Por sua vez, `Signature` representa a assinatura do método, isto é, essa metaclasse representa além do nome do método todos os parâmetros, o retorno do método, exceções, etc.

Na Linha 3 do Código-fonte 3 ❷, uma variável chamada `myCar` é declarada. As metaclasses que representam essa declaração podem ser visualizadas na Figura 12 ❷. A metaclasse `ActionElement` representa o significado das operações, por exemplo, uma declaração da variável. A metaclasse `StorableUnit` representa a própria variável `myCar`. Ainda na Linha 3 ❸, a instância da classe `Car` é criada usando a palavra-chave `new`. Nota-se que na Figura 12 ❸ três metaclasses são utilizadas para representar o operador `new`. Primeiramente, a metaclasse `ActionElement` é usada para ilustrar o significado da operação, nesse caso, a instância da classe `Car`. A metaclasse `Calls` é usada para ilustrar a instanciação de um objeto, nesse caso, o objeto `Car`. Adicionalmente, a metaclasse `Calls` posssui duas associações: `to` e `from`, as quais representam a chamada para o construtor de `Car` e representam o alvo `ActionElement`, respectivamente. Em seguida, a metaclasse `Creates` representa a nova instância de `Car`.

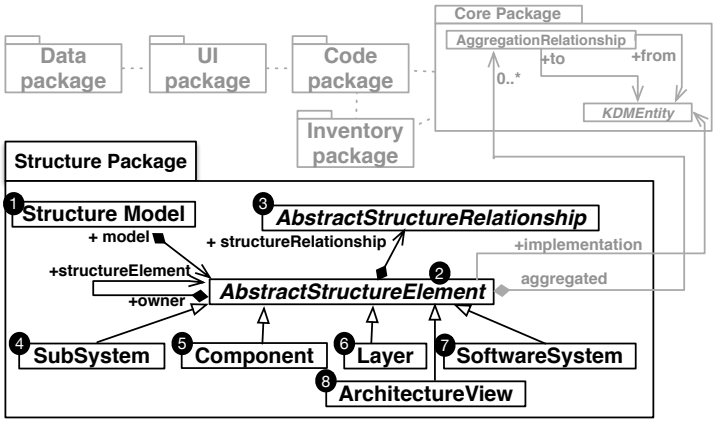
Na linha 4 do Código-fonte 3 ❹, um método acessor é invocado. Como pode ser visto

na Figura 12 ❹, três metaclasses são utilizadas no KDM para representar essa linha. Primeiro é criada uma metaclasse `ActionElement` que representa a declaração em si. Em seguida, outro `ActionElement` é criado para representar a invocação de método. Finalmente, outra metaclasse `Calls` é instanciada para representar a chamada do método `getName()`.

2.5.3 Pacote Structure

O KDM define metaclasses que representam componentes arquiteturais, como subsistemas, camadas, componentes, etc., e ele define também a rastreabilidade desses elementos para outras metaclasses do próprio KDM para o mesmo sistema por meio do pacote `Structure`.

Figura 13 – Diagrama de classes do pacote `Structure`.



Fonte: Adaptada de ADM (2012).

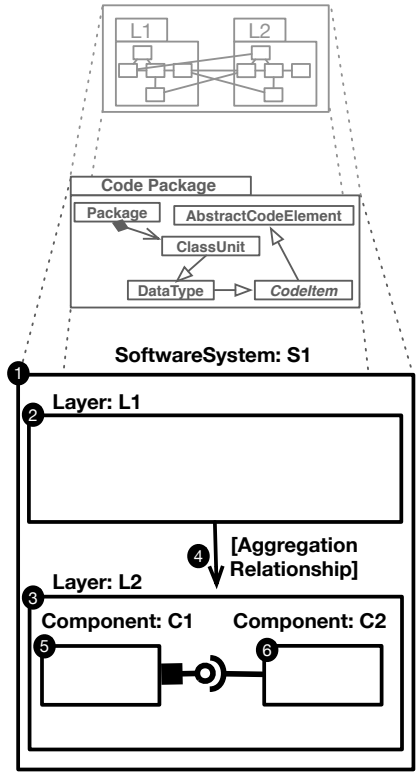
Esse pacote define um ponto de vista arquitetural para um domínio estrutural. As visões de arquitetura com base no ponto de vista definido pelo pacote `Structure` representam a forma como os elementos estruturais do sistema de software estão relacionados com os módulos definidos em código-fonte, que correspondem ao pacote `Code` do KDM. Uma parte simplória do pacote `Structure` é apresentada na Figura 13 como um diagrama de classes.

Usando suas metaclasses é possível relacionar todos os elementos estruturais do sistema, juntamente com os elementos computacionais, isto é, pode-se especificar os elementos estruturais do sistema. Na Figura 13, é mostrado que o pacote `Structure` e suas metaclasses são usados em combinação com os pacotes `Code`, `Data`, `Platform`, `UI` e `Inventory`. O modelo `Structure` possui uma coleção de elementos estruturais, como pode ser visto na Figura 13 ❶, isso é representado por meio de uma associação. Pacotes (do modelo `Code`) são os elementos folha do modelo `Structure`, representando a divisão de um sistema em módulos `Code` discretos, com partes não sobrepostas. A metaclasse `SoftwareSystem` fornece um ponto de encontro para todos os pacotes do sistema direta ou indiretamente por meio de outra associação chamada de `AbstractStructureElement[0..*]`. Os pacotes podem ainda ser agrupados nas metaclasses `SubSystem`,`Layer`,`Component` e `ArchitectureView`.

A metaclasse `AbstractStructureElement` (conforme a Figura 13 ❷) representa uma parte arquitetural relacionada com a organização do sistema de software existente em módulos e possui quatro associações. A primeira associação representa os elementos pertencentes ao modelo e é chamada de `structureElement` : `AbstractStructureElement`$[0..*]$. Em seguida, há uma associação denominada `structureRelationship` : `AbstractStructureRelationship`$[0..*]$; ela é usada para representar todos os relacionamentos em nível arquitetural. A associação `aggregated` : `AggregatedRelationship`$[0..*]$ representa uma relação abstrata entre dois elementos do KDM e dentro dela é possível definir relações concretas. A última associação da metaclasse `AbstractStructureElement` é o `implementation` : `KDMEntity`$[0..*]$. Essa associação é usada para especificar os elementos computacionais (do pacote `Code`, ou seja, `Package`, `ClassUnit`, `InterfaceUnit`, etc.) que representam o elemento estrutural.

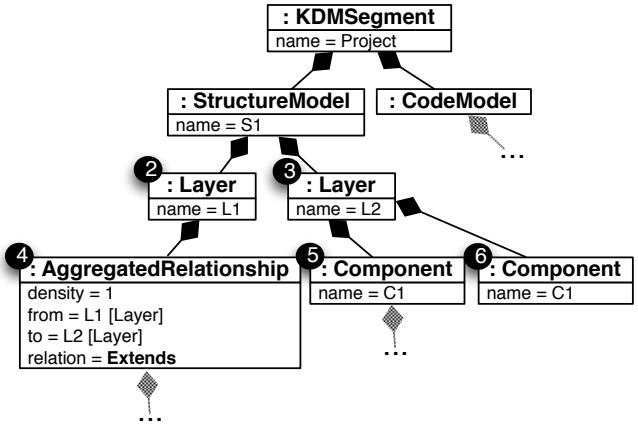
Na Figura 14, é descrita uma possível arquitetura estruturada para ilustrar como o KDM pode ser utilizado para representar elementos arquiteturais. Pode ser observado que essa figura é dividida em três níveis para ilustrar como o pacote `Structure` está relacionado com o pacote `Code`. O nível mais baixo representa o código-fonte e os artefatos físicos. L1 e L2 representam pacotes em código-fonte e cada caixa dentro dos pacotes representa as suas classes e interfaces; também é possível perceber que essas classes e interfaces são relacionadas umas com as outras de alguma maneira. No meio, há metaclasses do pacote `Code`, o que significa que as instâncias dessas metaclasses são usadas para representar os artefatos de baixo nível, ou seja, instâncias de `Package` são usadas para representar L1 e L2 e instâncias de `ClassUnit` e `InterfaceUnit` são usadas para representar as classes e interfaces, respectivamente. Finalmente, no nível superior, a arquitetura é mostrada. Todos os elementos arquiteturais são representados com a seguinte padronização: a metaclasse que representa os elementos arquiteturais, ':' seguido pelo seu nome. A arquitetura demonstrada é dividida da seguinte forma: no ponto mais alto de abstração há um `SoftwareSystem` (`S1`) ❶, que é dividido em duas camadas, `Layer L1` ❷ e `Layer L2` ❸; tais camadas representam elementos arquiteturais correspondentes aos pacotes L1 e L2 e são representadas no nível mais baixo. A `Layer L1` pode acessar os elementos da `Layer L2` e a restrição "pode acessar" é representada pela metaclasse `AggregatedRelationship` ❹. Além disso, a `Layer L2` contém dois componentes, `C1` ❺ e `C2` ❻. Finalmente, o `Component C1` fornece recursos por meio de uma interface para o `Component C2`.

Figura 14 – Exemplo de uma Arquitetura.



Fonte: Elaborada pelo autor.

Figura 15 – Instância KDM correspondente à Figura 14.

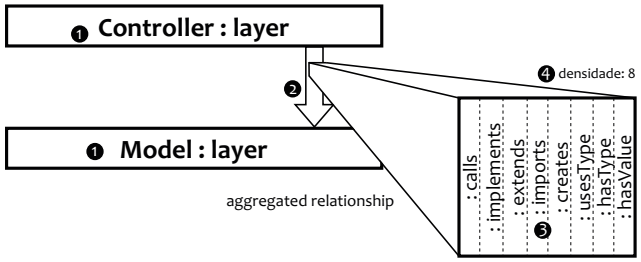


Fonte: Elaborada pelo autor.

A instância correspondente, porém simplificada da Figura 14 é mostrada na Figura 15. Os diamantes destacados em cinza e em anexo com três pontos (...) ilustram que algumas metaclasses não são mostradas de forma a simplificar a figura. Todos os elementos arquiteturais são subclasses de `StructureModel`. As camadas são representadas pela metaclasse `Layer`, como pode ser visto na Figura 15 ❷ e ❸. Da mesma forma, os componentes são representados pela metaclasse `Component`. Além disso, a metaclasse mais importante nessa figura é `AggregatedRelationship`. Ela representa a relação entre a `Layer` `L1` e a `Layer` `L2` e possui meta-atributos que almejam fornecer informações sobre o relacionamento. Por exemplo, o meta-atributo `density` ilustra o número de relações primitivas entre essas camadas. Na Figura 15, o meta atributo `density` possui o valor 1 (um). Outros dois meta-atributos são o `from` e o `to`, que representam os elementos arquiteturais de origem e destino, respectivamente. Eles são usados para especificar que a `Layer` `L1` em `SoftwareSystem` `S1` pode acessar a `Layer` `L2` também em `SoftwareSystem` `S1` de alguma forma. Finalmente, o meta-atributo `relation` representa como a `Layer` `L1` pode acessar a `Layer` `L2` e, nesse contexto, por meio de herança usando a metaclasse `Extends`. Na Figura 16, é mostrado como as relações entre dois elementos arquiteturais são consideradas neste estudo.

Na Figura 16 ❶, os elementos arquiteturais são apresentados, ou seja, duas camadas: `Controller` e `Model`. Como visto anteriormente, um relacionamento em âmbito arquitetural acontece entre dois elementos no nível estrutural (camadas, componentes, subsistemas, etc). Por

Figura 16 – Relacionamento entre dois elementos arquiteturais.



Fonte: Elaborada pelo autor.

exemplo, na Figura 16 ❷, é mostrado um `aggregatedRelationship` contendo todos os possíveis relacionamentos ❸ entre dois elementos (chamadas de métodos, herança, etc.). Finalmente, a densidade ❹ é definida como oito, que representa o número total de relacionamentos possíveis.

2.6 Ferramenta de apoio ao KDM

Um dos trabalhos mais importantes publicados no contexto da ADM é o de Bruneliere *et al.* (2010), Brunelière *et al.* (2014), que propõe uma ferramenta chamada MoDisco, a qual é um *framework* genérico e extensível para a abordagem de Engenharia Reversa dirigida a modelos e foi implementada no *Integrated Development Environment* (IDE) Eclipse como um *plug-in*. Mais especificamente, MoDisco é construído utilizando o *Eclipse Modeling Framework* (EMF). Basicamente, essa ferramenta é capaz de recuperar o código-fonte legado, a base de dados e outros artefatos legados, além de representá-los com o metamodelo KDM. Um dos principais objetivos da ferramenta MoDisco é ser adaptável para diferentes cenários, facilitando a sua utilização por uma base de usuários potencialmente maior (BRUNELIèRE *et al.*, 2014). Inicialmente criado como um modelo experimental de investigação pela Equipe AtlanMod (*Ecole des mines de Nantes* (EMN) & *Institut National de Recherche en Informatique et en Automatique* (INRIA)), o projeto evoluiu para uma solução industrializada graças à colaboração da empresa MIA-Software. Esse trabalho resultou em um conjunto eficiente e utilizável de ferramentas para a descoberta, consulta e manipulação de modelos de software, auxiliando toda a atividade de engenharia reversa.

MoDisco visa representar uma grande variedade de artefatos (por exemplo, código-fonte, banco de dados, arquivos de configuração, documentação, etc.) de um sistema legado. Contudo, algumas das limitações dessa ferramenta são: o suporte de refatoração e a aplicação de refatoração. É visto que, naturalmente, MoDisco não é capaz de aplicar refatorações de forma automática, pois a maioria das refatorações necessita de interação do usuário para fornecer as informações necessárias. No contexto dessa tese foi utilizada a ferramenta MoDisco para recuperar as informações do código-fonte legado escrito em Java. Sem o auxílio dessa ferramenta, todo o sistema legado escrito em Java, deveria ser transformado em uma instância do KDM de forma manual, o que poderia atrasar a presente pesquisa, uma vez que toda a manipulação do

KDM foi possível por causa da existência do MoDisco e do seu suporte em Java para manipular o metamodelo KDM. Por exemplo, não é possível para o MoDisco adivinhar quais refatorações devem ser aplicadas e em quais elementos; tais informações devem ser fornecidas por um usuário.

2.7 Considerações Finais

Neste capítulo, foi apresentada uma revisão dos principais conceitos envolvendo engenharia dirigida por modelos, refatoração, ADM e KDM, que são relevantes para a proposta desta tese.

Foram discutidas e apresentadas todas as etapas que devem ser realizadas para a condução da engenharia dirigida por modelos, ou seja, todos os níveis (CIM, PIM e PSM) foram apresentados. Em seguida, foram evidenciadas a definição e a diferença de metametamodelo, metamodelo, modelo e dados. Posteriormente, transformações em modelos foram descritas e discutidas, salientando as principais classificações relacionadas às transformações encontradas na literatura - vertical ou horizontal; endógenas ou exógenas. Ainda em relação à transformação de modelos, algumas das principais linguagens utilizadas para realizar a transformação em modelos também foram destacadas. Porém, apenas a linguagem ATL foi discutida com mais informações. E os principais conceitos relacionados com refatoração também foram apresentados.

Além disso, este capítulo direcionou-se à análise do panorama atual da literatura que trata sobre a modernização de sistemas legados, levando em consideração a padronização proposta pelo OMG. Assim, foram mostrados os principais conceitos sobre ADM, KDM, bem como seus pacotes e camadas, que são necessários para facilitar o entendimento desta tese e também são fundamentais para o desenvolvimento da proposta aqui desenvolvida.

Observou-se, também, que o metamodelo KDM por intermédio de suas camadas, pacotes e metaclasses permite a criação de um modelo independente de plataforma, representado um sistema em diversas visões. O objetivo do OMG ao criar esse metamodelo é propor uma padronização da reengenharia de software, fornecendo abstrações que ajudem no processo de reengenharia de sistemas. Além disso, pode-se constatar que, diferentemente de metamodelos existentes, como a UML, o KDM tem como intuito agrupar todos os artefatos (visões) do sistema em um único metamodelo. Dessa forma, pode-se argumentar que o metamodelo KDM é considerado como uma família de metamodelos, uma vez que compartilha uma terminologia consistente e homogênea.

Finalmente, também foi evidenciado a ferramenta MoDisco - uma das principais ferramentas que automatiza a instanciação do metamodelo KDM. Essa ferramenta foi utilizada no contexto deste trabalho para dar suporte à recuperação de instâncias do metamodelo KDM a partir de código-fonte escrito em Java. No próximo capítulo, é apresentado um mapeamento sistemático sobre ADM e KDM que foi conduzido para identificar possíveis desertos de evidências.

CAPÍTULO

3

MAPEAMENTO SISTEMÁTICO SOBRE ADM E KDM

3.1 Considerações Iniciais

Quando se conduz uma revisão de literatura sem o pré-estabelecimento de um protocolo de revisão, há um direcionamento por interesses pessoais, o que leva a resultados pouco confiáveis. Nesse contexto, pesquisadores vêm utilizando uma técnica denominada de Mapeamento Sistemático (MS) para auxiliar o pesquisador a conduzir uma revisão bibliográfica de forma totalmente sistemática, com o intuito de evitar que trabalhos importantes fiquem fora de suas pesquisas. Um MS é caracterizado por ser um meio de avaliar e interpretar todas as pesquisas disponíveis, referentes a uma questão de pesquisa, tema, área ou fenômeno de interesse. O MS visa expor uma avaliação justa de um tema de pesquisa, usando uma metodologia confiável e rigorosa (PETERSEN *et al.*, 2008; KITCHENHAM *et al.*, 2010; PETERSEN; VAKKALANKA; KUZNIARZ, 2015).

De acordo com Petersen, Vakkalanka e Kuzniarz (2015), o MS é projetado para dar uma visão geral de uma área de investigação por meio da classificação e contagem de contribuições em relação a um conjunto de categorias de classificação (PETERSEN *et al.*, 2008; KITCHENHAM *et al.*, 2010). Em outras palavras, trata-se de realizar uma vasta pesquisa na literatura, a fim de identificar quais temas já foram abordados, quais temas ainda não foram abordados e quais são novas possíveis pesquisas (KITCHENHAM *et al.*, 2010). Existe também a técnica de Revisão Sistemática (RS), a qual compartilha algumas características com o MS, como no que diz respeito à busca e seleção do estudo. Segundo Petersen, Vakkalanka e Kuzniarz (2015), tais técnicas são diferentes em termos de objetivos e, portanto, usam diferentes abordagens para a análise de dados, pois a RS visa sintetizar evidências, também considerando a força da evidência, e o MS preocupa-se principalmente com a estruturação de uma determinada área de pesquisa (PETERSEN; VAKKALANKA; KUZNIARZ, 2015). Segundo Kitchenham *et al.*

(2010), MS implica na forma mais adequada para se identificar, avaliar e interpretar toda uma área de pesquisa para um tema em particular.

Resume-se, então, que um MS configura um alicerce para novas atividades de pesquisa acerca de um determinado tema. Diante disso, foi realizado um MS sobre ADM e KDM (DURELLI *et al.*, 2014b); a motivação para realizar esse MS é identificar os temas que são mais investigados, bem como os temas que ainda não foram pesquisados no contexto da abordagem ADM e do metamodelo KDM. Embora a ADM seja uma abordagem relativamente nova, o OMG afirma que ela é uma importante abordagem, pois combina dois dos principais campos da Engenharia de Software: MDE (ver Capítulo 2, Seção 2.2) e reengenharia de software. Desde a criação da ADM, muitos esforços têm enfatizado a modernização de sistemas por meio dessa abordagem. Assim, se faz necessário a condução de uma investigação mais sistemática dos temas englobados por tal área de pesquisa. Nota-se que este capítulo é uma extensão do seguinte artigo: *A Mapping Study on Architecture-Driven Modernization* (DURELLI *et al.*, 2014b).

Este capítulo está organizado da seguinte forma: Seção 3.2 descreve a metodologia de como o MS foi conduzido; Subseção 3.2.1 apresenta a estratégia de busca utilizada nesse MS; a Subseção 3.2.2, descreve as fontes de estudos utilizadas no MS, além disso, também apresenta como foi realizada a seleção dos estudos; a Subseção 3.2.3 apresenta o esquema de classificação considerado no MS; na Subseção 3.2.4, a extração e a síntese dos dados são discutidas; na Subseção 3.2.5, um gráfico de bolha resultante do MS é apresentado, dando ênfase nas principais descobertas desse MS. Ainda nessa subseção, as questões de pesquisas são discutidas; a Seção 3.3 apresenta as principais constatações e questões em aberto, relacionadas à ADM e ao KDM; a Seção 3.4 apresenta as ameaças à validade do MS; Seção 3.5 descreve as considerações finais deste capítulo.

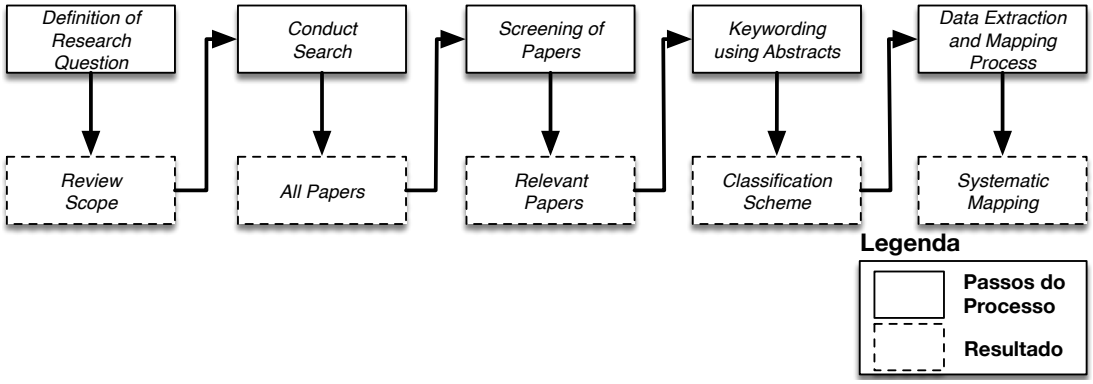
3.2 Metodologia de Pesquisa

Como já salientado nas considerações iniciais, um MS fornece um processo sistemático para identificar relevantes pesquisas com o objetivo de responder específicas questões. Durante a condução desse MS, todos os passos propostos por Petersen, Vakkalanka e Kuzniarz (2015), Petersen *et al.* (2008) foram seguidos. Uma visão geral de todos os passos propostos por tais autores pode ser observada na Figura 17. Nota-se que cada passo produz um resultado intermediário e de acordo com tais autores, cinco passos são essenciais: (*i*) *Definition of Research Questions*, (*ii*) *Conducting Search*, (*iii*) *Screening of Papers*, (*iv*) *Keywording using Abstracts* e (*v*) *Data Extraction and Mapping Process*.

3.2.1 Estratégia de Busca

O protocolo do MS deve ser definido e é usualmente dividido em dois subpassos: (*i*) a definição das questões de pesquisa e (*ii*) a *string* de busca.

Figura 17 – Processo para a condução de um Mapeamento Sistemático.



Fonte: Adaptada de Petersen *et al.* (2008).

Questões de Pesquisas (QPs) devem englobar o propósito do MS, o qual foca a identificação e a caracterização do estado atual da ADM e do metamodelo KDM. Dessa forma, como já citado, a motivação para realizar esse MS é identificar os temas que são mais investigados, bem como os temas que ainda não foram investigados no contexto da abordagem ADM e do metamodelo KDM. A partir desses objetivos, delineiam-se as seguintes questões:

- **QP**$_1$ - Dados os metamodelos da ADM, qual é mais utilizado na literatura? Além disso, dado o metamodelo identificado, qual (is) é (são) o (s) pacote (s) mais e menos utilizado (s)?
- **QP**$_2$ - Que tipos de estudos são publicados no contexto da ADM?
- **QP**$_3$ - Quais são as áreas mais e menos investigadas no contexto da ADM? Adicionalmente, quais são os tipos de contribuições que foram publicados até agora?

Considerando as QPs estabelecidas, definiram-se os atributos e a amplitude do MS com a técnica *Population*, *Intervention*, *Comparator* e *Outcomes* (PICO) (KITCHENHAM *et al.*, 2010), e identificaram-se os termos a serem utilizados na *string* de busca:

- Quanto à população: Em Engenharia de Software e no contexto de MS, população diz respeito a uma área de pesquisa específica. No contexto desse MS, a população são artigos publicados na literatura científica sobre algum processo, técnica ou ferramenta que utilize ADM e seus metamodelos;
- Quanto à intervenção: Em Engenharia de Software, intervenção refere-se à metodologia de software, ferramenta, tecnologia ou procedimento. No contexto, desse MS, a intervenção são abordagens e ferramentas publicadas na literatura científica que utilizam ADM e seus metamodelos;
- Quanto à comparação: A comparação não é aplicada no contexto desse MS;

- Quanto aos resultados esperados: Espera-se como resultado uma visão geral dos estudos que foram publicados para a ADM e seus metamodelos, enfatizando estudos primários que descrevem técnicas, abordagens, processos e ferramentas para auxiliar o engenheiro de modernização durante a condução de modernização de sistemas legados com a utilização da abordagem ADM.

A partir dos termos identificados, define-se a *string* de busca para a recuperação de estudos. Todos os termos devem ser traduzidos de acordo com o idioma dos artigos que se deseja recuperar (no contexto deste MS, inglês) e associados com sinônimos, conforme sugestões de especialistas. Na Figura 18, é exposta a *string* de busca que foi utilizada no presente MS.

Figura 18 – *String* de busca definida.

("*KDM*") **OR** ("*Knowledge Discovery Metamodel*") **OR** ("*Knowledge-Discovery Metamodel*") **OR** ("*Knowledge-Discovery Meta-model*") **OR** ("*Knowledge Discovery Meta-model*") **OR** ("*Architecture Driven Modernization*") **OR** ("*Architecture-Driven Modernization*") **OR** ("*Model Driven Modernization*") **OR** ("*Model-Driven Modernization*") **OR** ("*Model-driven software modernization*") **OR** ("*Abstract Syntax Tree Metamodel*") **OR** ("*ASTM*") **OR** ("*Structured Metrics Metamodel*") **OR** ("*SMM*")

Fonte: Elaborada pelo autor.

3.2.2 Fonte de Estudos e Seleção dos Estudos

As fontes de estudos utilizadas durante o MS foram as bibliotecas digitais da *ACM*, *IEEE XPLORE*, *Scopus*, *Web of Science* e *Engineering Village*. Tais bibliotecas digitais foram selecionadas com base na experiência reportada por Dyba, Dingsoyr e Hanssen (2007). De acordo com esses autores, tais fontes de estudos são suficientes para identificar estudos primários relevantes. Nota-se que os recursos fornecidos por essas bibliotecas digitais, bem como a sintaxe exata da *string* de busca a ser aplicada variam de uma biblioteca para outra, assim, a *string* de busca apresentada na Figura 18 foi utilizada como base para construir uma *string* de busca semanticamente equivalente e sob medida para cada biblioteca digital.

Para determinar quais estudos primários são relevantes para responder às QPs, definiu-se um conjunto de critérios de inclusão.

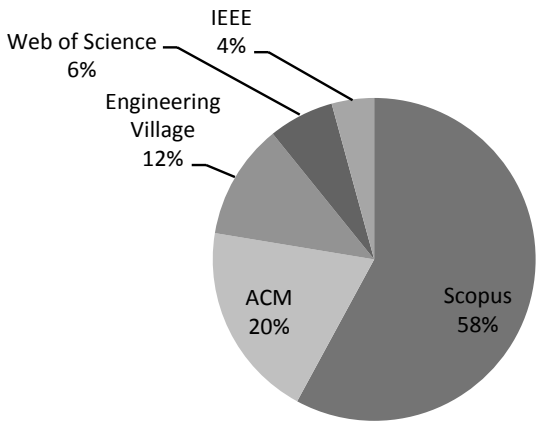
- Critérios de Inclusão:
  - O estudo primário apresenta pelo menos uma abordagem de modernização que utiliza ADM e seus metamodelos;
  - O estudo primário descreve uma avaliação empírica da abordagem que utiliza ADM.

Similarmente, também foram definidos três critérios de exclusão, a saber:

- Critérios de Exclusão:
  - Artigos que mencionam a ADM e seus metamodelos apenas no *abstract*;
  - Artigos introdutórios para livros e *workshops*;
  - O estudo primário é um artigo pequeno (*short paper*), o qual contém até três páginas.

A Scopus foi a biblioteca digital que retornou mais estudos primários, 58% (150) (ver Figura 19). Por outro lado, foram recuperados 20% (51) estudos da ACM, 12% (30) da Engineering Village, 6% (17) da Web of Science e 4% (11) da IEEE.

Figura 19 – Distribuição dos estudos primários de cada biblioteca digital.



Fonte: Elaborada pelo autor.

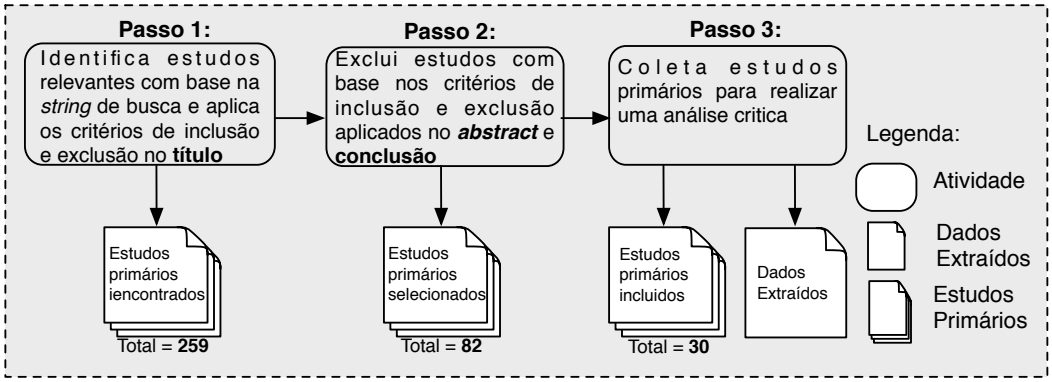
Somando todas as bibliotecas digitais, obtiveram-se 259 estudos primários no primeiro passo, como ilustrado na Figura 20. Após o primeiro passo (ver Figura 20), 82 artigos foram selecionados, sendo possível notar que apenas publicações de conferências e *journals* foram considerados nesse MS. Posteriormente, os critérios de inclusão anteriormente apresentados foram aplicados para os 82 artigos selecionados. Após esse passo, 30 estudos primários foram considerados para serem analisados no MS como mostrado na Figura 20.

3.2.3 Definindo um Esquema de Classificação

O esquema de classificação utilizado no MS foi o esquema proposto por Petersen *et al.* (2008), Petersen, Vakkalanka e Kuzniarz (2015), que classifica cada publicação entre categorias de acordo com três perspectivas: (*i*) **Área de Foco**, (*ii*) **Tipo de Contribuição** e (*iii*) **Tipo de Pesquisa**. O esquema de classificação resultante é descrito a seguir.

- **Área de Foco**: Após ler os estudos primários, foram identificadas cinco principais áreas de foco:

Figura 20 – Todos os passos conduzidos no MS.



Fonte: Elaborada pelo autor.

  - "**Modernização de Software**": está relacionada com estudos primários que descrevem abordagens que empregam ADM para modernizar sistemas legados para outra plataforma ou arquitetura;
  - "**Extração de *Business Knowledge***": descreve estudos primários que apresentam processos, métodos ou abordagens para extrair informações de negócio de sistemas legados;
  - "**Extração de Interesse**": representa estudos primários que descrevem processos, métodos ou abordagens para extrair interesses transversais de sistemas legados;
  - "**Extensão dos metamodelos da ADM**": descreve estudos primários que apresentam abordagens, métodos ou processos para estender um determinado metamodelo da ADM;
  - "**Aplicabilidade**": inclui estudos primários que buscam representar a evidência da utilização da ADM e seus metamodelos na prática, ou seja, artigos que apresentam pesquisas ou relatórios para facilitar o entendimento da ADM e seus metamodelos.

- **Tipo de Contribuição**: Similarmente, também foram identificados cinco tipos de contribuições:

  - "**Ferramentas**": estudos primários que apresentam ferramentas para auxiliar a modernização de sistemas legados utilizando ADM e seus metamodelos;
  - "**Processo**": estudos primários que descrevem processos para auxiliar a modernização de sistemas legados utilizando ADM e seus metamodelos;
  - "**Transformação de Modelos**": estudos primários que descrevem o uso de linguagens de transformações para realizar transformações entre os metamodelos da ADM;
  - "**Metamodelos**": estudos primários que relatam extensão nos metamodelos da ADM para suprir um específico problema, por exemplo, fornecer uma extensão leve para o metamodelo KDM representar o paradigma orientado a aspectos;

  - "**Métricas**": estudos primários que se concentram em propor ou aplicar métricas para medir a eficácia de ADM e seus metamodelos.

- **Tipo de Pesquisa**: reflete a abordagem de pesquisa que foi utilizada no estudo primário, e essa categoria foi criada com base no esquema proposto por Wieringa *et al.* (2005):

  - "**Pesquisa de validação**": tem por objetivo analisar uma proposta de solução que ainda não foi aplicada na prática. A validação é realizada de uma forma sistemática e pode apresentar qualquer um destes tipos: protótipos, análise matemática, etc.;
  - "**Pesquisa de avaliação**": em contraste com a pesquisa de validação, pesquisa de avaliação visa examinar uma solução que já foi praticamente aplicada. Estudos nessa categoria investigam a aplicação na prática da solução proposta e, geralmente, os resultados obtidos utilizando estratégias empíricas (por exemplo, experimentos e estudo de casos);
  - "**Proposta conceitual**": apresenta um arranjo de coisas que já existem, de uma nova maneira. No entanto, isso não resolve precisamente um problema particular. Podem ser incluídos taxonomias, referenciais teóricos, etc.;
  - "**Artigo descrevendo experiência**": artigos que descrevem sobre a experiência pessoal do autor para um ou mais projetos. O autor geralmente apresenta como o projeto foi feito e o que foi realizado;
  - "**Artigo descrevendo opinião**": artigos que descrevem a opinião pessoal do autor sobre a adequação ou inadequação de uma técnica ou ferramenta específica.

3.2.4 Extração e Síntese dos Dados

Os 30 estudos selecionados na etapa anterior foram analisados. Criou-se, então, um formulário para auxiliar a extração de dados, abordando os seguintes aspectos: (*i*) dados relevantes sobre como a ADM e seus metamodelos são utilizados na literatura, (*ii*) a data de quando a extração do dado foi realizada, (*iii*) o título do estudo primário, (*iv*) os autores do estudo primário, (*v*) o veículo de publicação e (*vi*) um resumo destacando as principais contribuições do estudo primário para posteriormente realizar a classificação. Durante o processo de extração, informações sobre cada estudo primário foram independentemente coletadas por todos os pesquisados que participaram do MS. É importante destacar que a primeira execução desse MS foi realizada em novembro de 2013, posteriormente, foi conduzido novamente em agosto de 2015 com o objetivo de atualizá-lo.

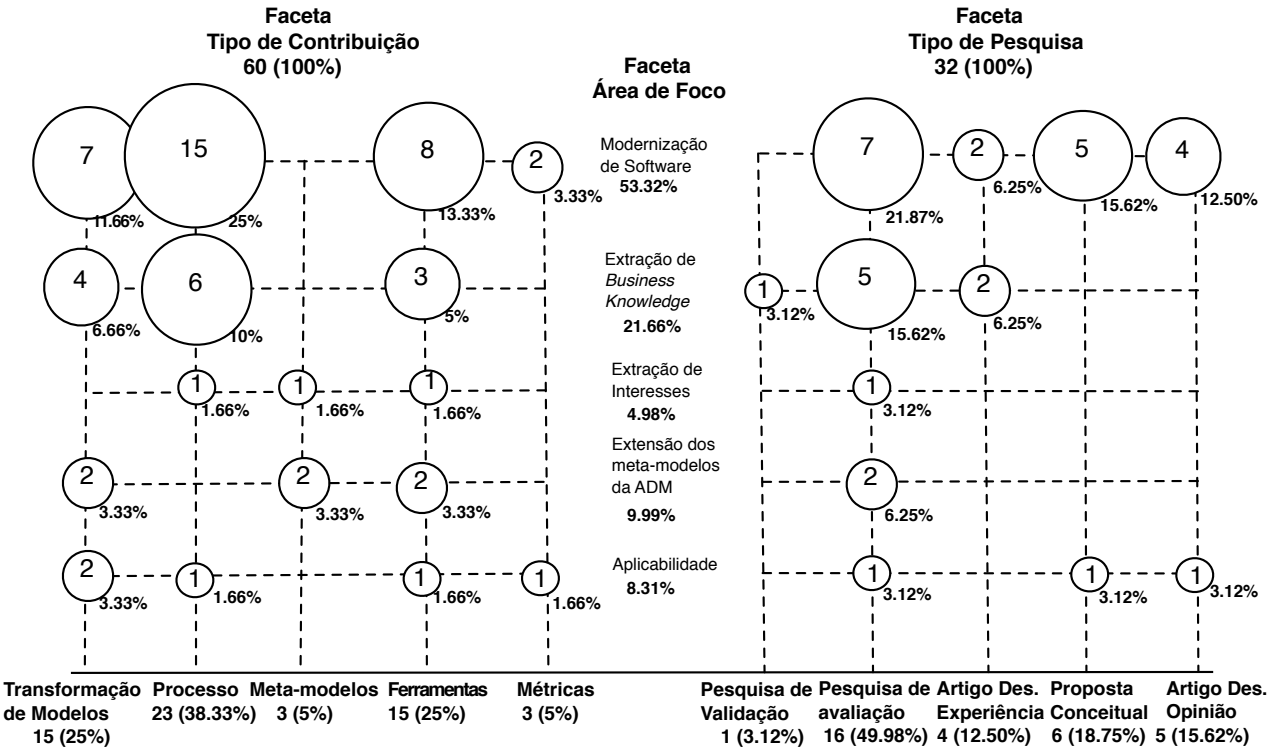
3.2.5 Mapeamento e discussão das QPs

O foco desta seção é apresentar uma visão geral de como a ADM e seus metamodelos são pesquisados e utilizados na literatura, bem como identificar possíveis grupos de evidências

(ou seja, onde pode haver margem para uma literatura mais completa) e deserto de evidências (ou seja, onde melhores ou novas pesquisas são necessárias) de pesquisas. Além de apresentar essa visão geral, a presente seção almeja destacar respostas para as QPs definidas anteriormente.

Em vez de utilizar tabelas de frequência, foi produzido um gráfico de bolha para reportar a frequência e distribuição dos estudos primários selecionados de acordo com suas categorias e data de publicação. Argumenta-se que esse gráfico de bolha representa um mapa geral de como a ADM e seus metamodelos são utilizados na literatura. O mapa resultante é apresentado na Figura 21.

Figura 21 – Visão geral da pesquisa sobre ADM e seus metamodelos.



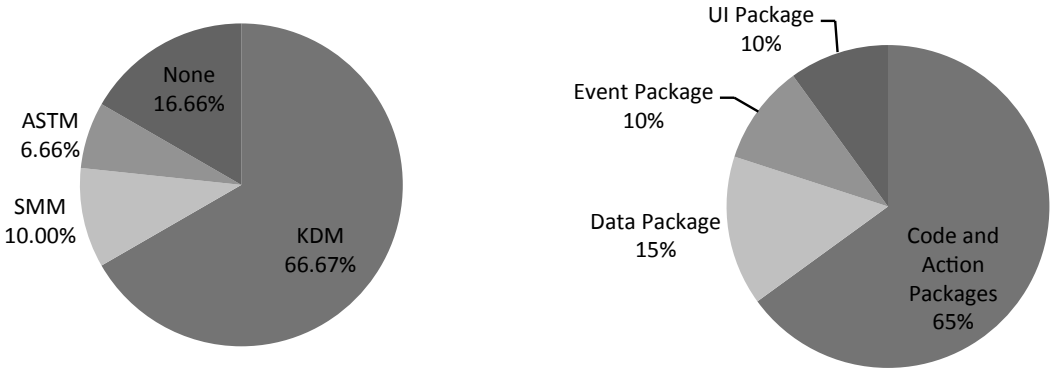
Fonte: Elaborada pelo autor.

Como pode ser observado, esse gráfico de bolha contém dois vértices, X e Y, os quais possuem bolhas em cada categoria. É visto que o tamanho de cada bolha representa o número de estudos primários que foram classificados em uma categoria específica. Esse gráfico é um resumo visual e fornece uma visão panorâmica que permite identificar quais são as categorias que foram salientadas em pesquisas anteriores, além disso, é possível identificar facilmente lacunas e oportunidades para futuras pesquisas. O gráfico de bolha apresentado na Figura 21 contém três facetas: **Tipo de Contribuição**, **Área de Foco** e **Tipo de Pesquisa**. Embora 30 estudos primários tenham sido considerados no presente MS, é importante mencionar que para o gráfico de bolha alguns estudos primários foram agrupados em mais de uma categoria. Por exemplo, para o gráfico de bolha apresentado na Figura 21, a soma dos estudos primários agrupados em cada faceta é maior do que o número de estudos primários selecionados, e isso acontece uma vez

que um determinado estudo primário pode ser classificado em diversas facetas e categorias.

Para responder a primeira parte da $\mathbf{QP}_1$, foram analisados todos os estudos primários, concentrando-se na identificação de qual metamodelo da ADM tem sido mais utilizado na literatura. Na Figura 22, lado esquerdo, são exibidos os metamodelos da ADM utilizados na literatura. Como pode ser observado, o metamodelo KDM é o mais utilizado, tendo uma frequência de 66.67%. Em seguida, o segundo metamodelo mais utilizado é o SMM - 10% dos estudos primários relatam a utilização desse metamodelo. O metamodelo ASTM foi utilizado em apenas 6.66% dos estudos primários. 16.66% dos estudos primários não mencionam explicitamente qual metamodelo foi utilizado durante o processo de modernização conduzido, apenas citam e relatam a utilização da ADM.

Figura 22 – Frequência de utilização dos metamodelos da ADM e frequência de utilização dos seus pacotes.



Fonte: Elaborada pelo autor.

Para responder a segunda parte da $\mathbf{QP}_1$, foram analisados quais pacotes são mais e menos utilizados dentro do metamodelo KDM. Na Figura 22, lado direito, é evidente que os pacotes `Code` e `Action` são os mais utilizados na literatura, contendo uma frequência de 65%. Acredita-se que a razão para essa frequência tão alta seja por dois principais motivos: (*i*) tais pacotes representam o código-fonte de um determinado sistema, além disso, a maioria das abordagens identificadas utiliza como entrada o código-fonte de sistemas que almejam modernizar seguindo a ADM e seus metamodelos; (*ii*) outros pacotes do KDM ainda não possuem ferramentas para realizar a instanciação de forma automática, ou seja, existe uma limitação de *parsers* que análisa outros tipos de artefatos de um determinado sistema para criar uma representação mais fiel do sistema, não apenas do código-fonte. O terceiro pacote mais utilizado é o `Data`, o qual é utilizado para representar dados, tais como banco de dados, registros, etc. Os pacotes `Event` e `UI` foram utilizados 10%. Outros pacotes do metamodelo KDM não foram explicitamente mencionados nos estudos primários identificados.

Observando a faceta **Tipo de Pesquisa**, lado direito da Figura 21, é possível responder a $\mathbf{QP}_2$. A maioria dos estudos primários identificados foi classificado como "**Pesquisa de Avaliação**", aproximadamente 49%. Uma pequena porcentagem de estudos primários foi classificada como "**Pesquisa de Validação**" - apenas 3.12%. 12.50% dos estudos primários identificados

descrevem a experiência dos autores ("**Artigo Des. Experiência**") com a utilização da ADM e de seus metamodelos. 18.75% foram agrupadas em "**Proposta Conceitual**" e "**Artigos Des. Opinião**" teve a frequência de 15.62%.

Ainda em relação à Figura 21 na faceta **Tipo de Contribuição**, lado esquerdo, é visto que a maioria dos estudos primários identificados apresenta "**processos**" para auxiliar os engenheiros de software durante a modernização de sistemas legados. Também foi identificado um total de 15 estudos primários (25%) que apresentam algum tipo de "**transformação de modelos**" utilizando os metamodelos da ADM. Similarmente, 15 estudos primários (25%) demonstram ferramentas para auxiliar o engenheiro de software durante a condução da modernização utilizando ADM e seus metamodelos. Acredita-se que esses dois últimos resultados foram obtidos uma vez que a maioria dos estudos primários identificados descreve processos de modernização, assim, pesquisadores devem criar um conjunto de transformações de modelos e ferramentas para automatizar parcialmente ou totalmente o processo proposto.

Por outro lado, "**metamodelos**" e "**métricas**" são as contribuições com menos estudos primários identificados, 5% cada. Com isso, argumenta-se que os estudos primários que descrevem "**processos**" para ajudar a modernização dos sistemas legados por meio da ADM, os estudos que apresentam "**transformação de modelos**" entre os metamodelos da ADM (KDM, SMM e ASTM) e os artigos que descrevem ferramentas para automatizar parcialmente ou totalmente o processo da ADM podem ser considerados como grupos de evidências. Os "**metamodelos**" (artigos que explicam e/ou apresentam como estender metamodelos da ADM) e "**métricas**" (artigos que descrevem como aplicar métricas nos metamodelos da ADM) podem ser considerados como deserto de evidência, evidenciando que novos estudos primários são necessários.

Considerando o centro da Figura 21, faceta **Área de Foco**, é possível visualizar que a maioria dos estudos primários identificados foi classificado como "**Modernização de Software**", um total de 53.32%. Em seguida, 21.66% dos estudos primários foram categorizados em "**Extração de *Business Knowledge***". "**Extração de Interesses**", "**Extensão dos metamodelos da ADM**" e "**Aplicabilidade**", coletivamente, representam uma porcentagem de aproximadamente 25% dos estudos primários. Como resultado dessa análise, foi possível responder parcialmente a $\mathbf{QP}_3$, ou seja, os principais **Tipo de Contribuição** relacionado à ADM e seus metamodelos disponíveis na literatura e identificados no MS foram destacados. A resposta conclusiva e completa da $\mathbf{QP}_3$ é apresentada a seguir. Para facilitar o entendimento e a organização deste mapeamento cada **Área de Foco** identificado no MS é apresentado em uma subseção.

3.2.5.1 Modernização de Software

Moratalla *et al.* (2012) propõem **GAFEMO**, a qual é uma abordagem para auxiliar a modernização de sistemas legados em sistemas orientados a serviço. Essa abordagem utiliza como entrada o sistema legado e, então, cria uma instância do metamodelo KDM para representar o código-fonte do sistema legado. Posteriormente, os autores definiram um conjunto de

transformações para serem aplicadas nessa instância do KDM almejando criar os serviços.

Mazón e Trujillo (2007) definiram uma abordagem para modernizar *data warehouses* seguindo os conceitos da ADM. Essa abordagem automaticamente executa as seguintes tarefas: (*i*) obtém uma representação lógica das fontes de dados; (*ii*) posteriormente, essa representação lógica é anotada e transformada em instância do KDM; (*iii*) em seguida, a instância do KDM é transformada em um modelo de análise de dados multidimensional. Similarmente, Guzman, Polo e Piattini (2007) definem uma abordagem para realizar a análise de sistemas legados e criar funcionalidades para serem expostas como serviços, usando conceitos de *Web Services* juntamente com a ADM.

Em Frey e Hasselbring (2011), Frey, Hasselbring e Schnoor (2012) os autores desenvolveram uma abordagem seguindo as diretrizes da ADM denominada "CloudMIG". Essa abordagem pretende fornecer software como serviço (do inglês - *Software as a Service* (SaaS)). Do mesmo modo, Guzman (2007) também desenvolveu uma abordagem para utilizar ADM e seus metamodelos, principalmente o KDM, para analisar sistemas legados, descobrir e criar funcionalidades para serem expostas como serviços, utilizando os conceitos de *Web Services*.

Pérez-Castillo *et al.* (2009), Pérez-Castillo, García-Rodríguez e Caballero (2009), Pérez-Castillo, Guzmán e Piattini (2012) apresentam uma abordagem para modernizar sistemas legados juntamente com o banco de dados relacional. Mais especificamente, essa abordagem obtém três principais modelos seguindo a abordagem ADM. Primeiro, tal abordagem recupera uma instância do pacote `Code` do metamodelo KDM. Em seguida, uma instância do pacote `Data` do metamodelo KDM também é recuperada para representar o banco de dados relacional. Essa segunda instância é recuperada com base nas *embedded SQL*, que são encontradas no código-fonte. O objetivo é separar tais SQL para facilitar a modularidade e futuras manutenções. Depois, transformações de modelos são executadas e o sistema legado é modernizado.

Fuentes-Fernandez, Pavon e Garijo (2012) apresentam uma abordagem de modernização denominada XIRUP, interativa e estruturada em quatro fases: (*i*) avaliação preliminar, (*ii*) compreensão, (*iii*) construção e (*iv*) migração. Essa abordagem de modernização é baseada em componentes, com foco no levantamento inicial de informações-chave, e depende de uma abordagem orientada a modelos, com o uso extensivo da experiência dos projetos anteriores.

Mainetti, Paiano e Pandurino (2012) apresentam uma abordagem que permite desenvolvedores automaticamente modernizar a interface gráfica de um determinado sistema legado para *Rich Internet Application* (RIA). Similarmente, Rodríguez-Echeverría *et al.* (2012) também apresentam uma abordagem de modernização que utiliza os metamodelos da ADM para definir um processo sistemático com o objetivo de transformar aplicações web em RIA.

Boussaidi *et al.* (2012) definem uma abordagem que ajuda na construção de distintas visões arquiteturais de sistemas legados. Assim, os autores criaram um conjunto de algoritmos de agrupamento que é conduzido por meio de visões arquiteturais comuns. Essa abordagem faz

utilização do metamodelo KDM. Izquierdo e Molina (2010) utilizam os conceitos da ADM para construir uma ferramenta de modernização, visando gerar relatórios de métricas para avaliar os esforços de migração. Os autores desenvolveram um extrator que gera instâncias do metamodelo KDM a partir de código PS-SQL, ou seja, transformaram PS-SQL para instâncias do metamodelo KDM, e, em seguida, relatórios de métricas foram gerados para o metamodelo KDM.

3.2.5.2 Extração de Business Knowledge

Pérez-Castillo, Guzmán Weber e Places (2011), Pérez-Castillo *et al.* (2011), Pérez-Castillo e Guzman (2013), Pérez-Castillo, Guzmán e Piattini (2010) apresentam uma abordagem para recuperar regras de negócio de um determinado sistema legado, com base na ADM e no KDM. Essa abordagem é baseada em um conjunto de transformações: (*i*) vários PSM são recuperados de acordo com artefatos específicos do sistema legado, (*ii*) posteriormente, tais PSM são transformados em uma instância do metamodelo KDM, (*iii*) em seguida, o metamodelo KDM é transformado em um modelo específico para definir regras de negócio. Além disso, os autores realizaram um conjunto de estudo de caso para verificar a eficiência e a eficácia da abordagem (PÉREZ-CASTILLO *et al.*, 2012).

Pérez-Castillo *et al.* (2012) também fornecem uma técnica semiautomática baseada em análise dinâmica, combinada com análise estática para instrumentar o código-fonte, com o objetivo de descobrir e obter processos de negócios em nível do metamodelo KDM. Pérez-Castillo, Guzmán e Piattini (2010) apresentam e descrevem com detalhes todas as transformações entre o metamodelo KDM e o metamodelo *Business Process Model and Notation* (BPMN).

Normantas e Vasilecas (2012) apresentam uma abordagem que facilita a compreensão de um determinado software, permitindo a rastreabilidade de regras de negócios e cenários de negócios no sistema de software. A abordagem visa extrair conhecimentos específicos de negócios a partir do conhecimento sobre o sistema de software existente, representados no KDM. Fernández-Ropero *et al.* (2012) descrevem um conjunto de regras para transformar o metamodelo *Mining* XML, o qual é utilizado para representar a sequência de atividades de negócios executados para o metamodelo KDM.

3.2.5.3 Extração de Interesses

Santibanez *et al.* (2013a), Santibanez *et al.* (2013b), Santibanez *et al.* (2015) definem uma abordagem denominada CCKDM para auxiliar a identificação de interesses transversais utilizando uma combinação de bibliotecas de interesses com o algoritmo de mineração de dados K-means. A abordagem possui quatro passos e dois deles são opcionais. A entrada ao processo é uma instância do metamodelo KDM e as saídas são a mesma instância do KDM com anotações dos interesses transversais identificados e alguns arquivos de registros (*logs*). A identificação é realizada por meio de uma biblioteca de interesses em conjunto com o repositório de elementos

KDM. A abordagem é explicada de forma que possa ser replicada por outros pesquisadores que tenham interesse em modificá-la e/ou estendê-la.

3.2.5.4 Extensão dos metamodelos da ADM

Baresi e Miraz (2011) definem uma extensão do metamodelo KDM denominada *Component-Oriented MOdernization* (COMO). De acordo com os autores, KDM suporta apenas parcialmente os conceitos de componentes. Dessa forma, a extensão COMO objetiva suprir tal limitação do metamodelo KDM. Assim, novas metaclasses são definidas para detalhar conceitos específicos relacionados com componentes. Similarmente, Pérez-Castillo *et al.* (2012) definem uma extensão para o metamodelo KDM que visa melhorar a representação de arquivos de registros (*logs*). Porém, diferentemente da abordagem COMO, os autores não criam novas metaclasses para o metamodelo KDM, apenas utilizam anotações. Shahshahani (2011) define uma extensão para o metamodelo KDM para representar todos os elementos do paradigma de programação orientada a aspectos, ou seja, novas metaclasses como `Aspect`, `Advice` e `Point-cut` podem ser instanciadas para representar conceitos do paradigma de programação orientada a aspecto por meio desse KDM estendido.

3.2.5.5 Aplicabilidade

Pérez-Castillo, Guzman e Piattini (2011), Pérez-Castillo *et al.* (2012), Pérez-Castillo e Guzman (2013) descrevem como utilizar e aplicar o metamodelo KDM para modernizar sistemas legados. Além disso, os autores também descrevem cada camada do metamodelo KDM e é apresentado um conjunto de exemplos para auxiliar no entendimento de como utilizar os conceitos da ADM e as metaclasses do metamodelo KDM durante as atividades de modernização de sistemas. De acordo com os autores, tais estudos primários podem ser utilizados para auxiliar novos pesquisados a entender e começar a utilizar os conceitos da ADM e do KDM.

3.3 Principais Constatações e Questões em Aberto

As recentes propostas em ADM têm-se concentrado principalmente na elaboração de abordagens para auxiliar a modernização do sistema legado para outra plataforma/arquitetura. No entanto, analisando minuciosamente o problema global da integração da modernização considerando o contexto da ADM, ainda há espaço para melhorias. Por exemplo, hoje em dia, apenas a ferramenta MoDisco[1] cria uma instância do metamodelo KDM de forma automática. Entretanto, essa ferramenta preocupa-se apenas com os pacotes `Code` e `Action`. Dessa maneira, para integrar o metamodelo KDM em um contexto maior, novos *parsers* precisam ser definidos e criados. Embora alguns autores tenham criado algumas iniciativas (IZQUIERDO; MOLINA, 2010; BRUNELIERE *et al.*, 2010), essa área ainda é limitada. Nesse contexto, ainda se faz

[1] https://eclipse.org/MoDisco/

necessário novos esforços para criar *parsers*, para representar todas as camadas do metamodelo KDM de forma automática ou semiautomática.

Foi observado durante o MS que existem algumas principais limitações a serem investidas para facilitar a utilização da ADM e do metamodelo KDM de forma eficaz. Por exemplo, como já salientado no Capítulo 2 Seção 2.3, refatorações são técnicas utilizadas para melhorar a estrutura do software. Atualmente, é evidente que a refatoração é de suma importância para melhorar a qualidade do código-fonte, e, assim, melhorar a sua manutenibilidade. Embora a ADM e, principalmente, o KDM tenham sido criados para auxiliar todo o processo da modernização de sistemas, até esse momento existe uma ausência de abordagens e ferramentas que auxiliem os engenheiros de modernização a criar e aplicar refatorações de forma consistente para o metamodelo KDM. Dessa forma, usualmente, os engenheiros de modernização precisam desenvolver suas próprias ferramentas para refatorar diversos sistemas representados em nível do metamodelo KDM. Tais soluções geralmente são proprietárias e, consequentemente por meio delas fica difícil reutilizar e prover interoperabilidade entre ferramentas.

Além disso, é sabido que a atividade de refatoração é pertinente a qualquer processo de modernização. Dessa forma, quando um sistema é representado utilizando diferentes visões conceituais para representar níveis de abstração do sistema (por exemplo, visão arquitetural, visão de código-fonte, visão do banco de dados, etc.), um acidente comum que surge durante atividades de refatorações é a dessincronização das instâncias do metamodelo, resultando em visões inconsistentes após a aplicação de uma refatoração. Assim, no contexto do metamodelo KDM existe uma carência de abordagens e apoios ferramentais que auxiliam a sincronizar tais mudanças após a aplicação de um conjunto de refatorações no KDM. Pesquisas recentes sugerem que a aplicação de técnicas de propagação de mudanças pode auxiliar na identificação e atualização de todas as instâncias/visões do KDM, permitindo manter todas as visões/instâncias do metamodelo KDM sincronizadas.

Na literatura, é possível identificar um conjunto de refatorações já validadas e que são geralmente aplicadas em código-fonte, por exemplo, *Extract Class*, *Move Method*, *Move Attribute*, etc. Essas são apenas alguns exemplos de refatorações úteis que não são facilmente reutilizadas na prática durante a condução de modernização de um determinado sistema. Essa limitação pode ser atribuída devido à ausência de um meio padronizado de disponibilizar refatorações. Embora a ADM forneça um conjunto de metamodelos para auxiliar o engenheiro de modernização a conduzir MDE, até esse momento a ADM não provê instruções para auxiliar o engenheiro a promover o reúso de refatorações juntamente com os seus metamodelos padronizados (por exemplo, KDM) durante o processo de modernização. Essa limitação faz com que o engenheiro de modernização crie suas próprias soluções/refatorações, resultando em um possível atraso no processo de modernização. Contudo, as soluções/refatorações definidas não são facilmente reutilizadas, pois são proprietárias e dependente de linguagem. Uma abordagem promissora é lidar com a refatoração de forma independente da linguagem, aumentando as possibilidades

de reutilização de refatorações. Dessa forma, existe uma necessidade de criar um metamodelo para auxiliar o engenheiro de modernização a promover o reúso de metadados relacionados às refatorações. Adicionalmente, esse metamodelo deve ser coerente com as terminologias da ADM e, principalmente, deve trabalhar de forma uniforme com o metamodelo KDM para garantir a independência de linguagem e plataforma proporcionada pelo KDM.

3.4 Ameaças à Validade

Nesta seção as quatro ameaças à validade do MS aqui estruturado são destacadas:

- **Seleção dos estudos primários**: Com o objetivo de garantir um processo de seleção imparcial, definiram-se questões de pesquisa e critérios de inclusão e exclusão. No entanto, não é possível descartar ameaças de uma perspectiva de avaliação da qualidade, uma vez que os estudos foram selecionados sem atribuir qualquer pontuação. Além disso, tentou-se ser o mais abrangente possível, de modo que nenhum limite foi definido em relação à data de publicação dos estudos primários. Nota-se que não foram definidas muitas restrições, pois almeja-se obter uma visão ampla da área de pesquisa;

- **Estudo primário não identificado**: Foi conduzido o MS em várias bibliotecas digitais (*ACM*, *IEEE XPLORE*, *Scopus*, *Web of Science* e *Engeneering Village*), todavia, é possível que alguns estudos primários não tenham sidos identificados durante a condução do MS. Para mitigar essa ameaça, foram selecionadas as bibliotecas digitais recomendas por Kitchenham *et al.* (2009), Dyba, Kitchenham e Jorgensen (2005);

- **Confiabilidade dos colaboradores**: Os revisores desse MS são pesquisadores da área de reutilização de software e não há conhecimento de qualquer viés que possa ter sido introduzido durante as análises;

- **Extração dos dados**: Outra ameaça com relação ao MS refere-se a como os dados foram extraídos das bibliotecas digitais. Nem toda informação estava clara o suficiente para responder às perguntas, assim, alguns dados tiveram de ser interpretados. A fim de garantir a validade, foram analisadas várias fontes de dados. No caso de desacordo entre dois colaboradores, um terceiro colaborador, definido como árbitro, era consultado para garantir um acordo comum.

3.5 Considerações Finais

Pesquisas na área da ADM podem levar a avanços na modernização de sistemas, resultando em sistemas que são mais sustentáveis, extensíveis e reutilizáveis. Para obter uma visão geral da pesquisa atual nessa área, foi realizado e apresentado neste capítulo um MS. O MS foi

conduzido em várias bibliotecas digitais, como *ACM*, *IEEE XPLORE*, *Scopus*, *Web of Science* e *Engeneering Village*. Posteriormente, identificaram-se 30 estudos primários, os quais foram utilizados para extrair informações para responder três QPs. Scopus foi a biblioteca digital que retornou mais estudos primários, 58% (150), foram recuperados 20% (51) estudos da ACM, 12% (30) da Engineering Village, 6% (17) da Web of Science e 4% (11) da IEEE.

O metamodelo KDM é o é mais utilizado, tendo uma frequência de 66.67%. Em seguida, o metamodelo que é mais utilizado é o SMM - 10% dos estudos primários relatam a utilização desse metamodelo. Já o metamodelo ASTM foi utilizado em apenas 6.66% dos estudos primários. 16.66% dos estudos primários não mencionam explicitamente qual metamodelo foi utilizado durante o processo de modernização conduzido, apenas citam e relatam a abordagem ADM. Também foi identificado que os pacotes `Code` e `Action` são os mais utilizados na literatura, contendo uma frequência de 65%. O terceiro pacote mais utilizado é o `Data`, o qual é usado para representar dados e persistência. Os pacotes `Event` e `UI` foram utilizados 10%. Outros pacotes do metamodelo KDM não foram explicitamente mencionados nos estudos primários.

A maioria dos estudos primários identificados foram classificados como "**Pesquisa de Avaliação**", aproximadamente 49%. Uma pequena porcentagem de estudos primários foi classificada como "**Pesquisa de Validação**", apenas 3.12%. 12.50% dos estudos primários identificados descrevem a experiência dos autores ("**Artigo Des. Experiência**") com a utilização da ADM e seus metamodelos. 18.75% foram agrupados em "**Proposta Conceitual**" e "**Artigos Des. Opinião**" teve a frequência de 15.62%. A maioria dos estudos primários identificados apresenta "**processos**" para auxiliar os engenheiros de software durante a modernização de sistemas legados. Também foram identificados 15 estudos primários (25%) que apresentam algum tipo de "**transformação de modelos**" utilizando os metamodelos da ADM. Similarmente, 15 estudos primários (25%) apresentam ferramentas para auxiliar o engenheiro de software durante a condução da modernização utilizando ADM e seus metamodelos. "**Metamodelos**" e "**métricas**" são as contribuições com menos estudos primários identificados, 5% cada. Observando a faceta **Área de Foco**, é possível visualizar que a maioria dos estudos primários identificados foram classificados como "**Modernização de Software**", um total de 53.32%. Em seguida, 21.66% dos estudos primários foram categorizados em "**Extração de *Business Knowledge***". "**Extração de Interesses**", "**Extensão dos metamodelos da ADM**" e "**Aplicabilidade**" coletivamente representam uma porcentagem de aproximadamente 25% dos estudos primários.

Outra contribuição deste capítulo é o mapa definido na Figura 21. Ao observar esse mapa, tem-se uma visão global da literatura em relação à ADM, identificando quais categorias foram enfatizadas nas últimas pesquisas, além de lacunas e possibilidades para futuras pesquisas. Adicionalmente, esse mapa também fornece esclarecimentos adicionais sobre as frequências de publicação ao longo do tempo.

Até esse momento, existe uma ausência de abordagens e apoios computacionais que auxiliem os engenheiros de modernização a aplicar refatorações de forma consistente para o meta-

modelo KDM. Diante disso, usualmente, os engenheiros de modernização precisam desenvolver suas próprias ferramentas para refatorar diversos sistemas. Tais soluções geralmente tendem a ser proprietárias e, como consequência, torna-se difícil a reutilização e a interoperabilidade entre ferramentas.

Com o intuito de mitigar essa ausência, nos próximos capítulos desta tese, uma abordagem, um metamodelo e um apoio computacional para auxiliar o engenheiro de modernização e o engenheiro de software durante a aplicação, reúso e compartilhamento de refatorações para o KDM são apresentados. Mais especificamente, no Capítulo 4 é destacada uma abordagem para criar refatorações para o metamodelo KDM, ou seja, evidenciando como refatorações podem ser criadas para o KDM (DURELLI *et al.*, 2014a; DURELLI *et al.*, 2014c).

No Capítulo 5, é apresentado um metamodelo para disponibilizar e promover o reúso de refatorações no contexto da ADM e do KDM. Um apoio computacional, denominado KDM-RE, é apresentado no Capítulo 6 e é composto por três *plug-ins* do Eclipse: (*i*) o primeiro consiste em um conjunto de *Wizards* que apoia o engenheiro de software na aplicação das refatorações em diagramas de classe UML; (*ii*) o segundo consiste em um apoio à importação e reúso de refatorações disponíveis no repositório; (*iii*) o terceiro consiste em um módulo de propagação de mudanças que permite manter modelos internos do KDM sincronizados;

CAPÍTULO

4

UMA ABORDAGEM PARA CRIAR REFATORAÇÕES PARA O KDM

4.1 Considerações Iniciais

O OMG fornece conceitos gerais para a condução de modernização dirigida a modelos por meio da ADM, porém, o grupo OMG não fornece instruções sobre como criar ou aplicar refatorações em instâncias de seus metamodelos. Assim, existe uma carência de soluções padronizadas e apoios ferramentais durante atividades de refatorações no contexto da ADM. A ausência de instruções claras que guiem o engenheiro de modernização na criação de refatoração para o metamodelo KDM faz com que os engenheiros de modernização sigam procedimentos *ad-hoc* que podem levar a refatorações de baixa qualidade.

Diante desse contexto, neste capítulo, é apresentada uma abordagem para auxiliar o engenheiro de modernização a criar refatorações para o KDM, a qual conta com um conjunto de passos para auxiliar a criação de refatorações para serem aplicadas em instâncias desse metamodelo. Para ilustrar a utilização dos passos apresentados nesse capítulo, algumas refatorações propostas por Fowler (1999) são criadas para o metamodelo KDM. Utilizando os passos aqui definidos, engenheiros de modernização podem criar refatorações para o metamodelo KDM e facilitar a condução da modernização de um sistema representado como uma instância desse metamodelo (DURELLI *et al.*, 2014a; DURELLI *et al.*, 2014c). É importante observar que, neste capítulo, apenas a abordagem para criar refatorações para o metamodelo KDM é apresentada - a preservação de comportamento, pren servação de consistência e sincronização do metamodelo KDM é tratado no módulo de sincronização do apoio computacional KDM-RE, o qual é apresentado no Capítulo 6 Seção 6.5. Dessa forma, observando a Figura 1 ❶, nota-se que a abordagem apresentada neste capítulo corresponde a um conjunto de passos para auxiliar o engenheiro de modernização durante a criação de refatorações para o KDM.

As demais seções deste capítulo estão organizadas da seguinte forma: Na Seção 4.2, é

apresentada a abordagem de criação de refatorações para o metamodelo KDM. Essa abordagem possui seis passos: (*i*) identificar elementos estruturais, (*ii*) identificar operações; (*iii*) implementar refatoração; (*iv*) definir restrições e (*v*) especificar refatoração. Na Seção 4.3, são apresentados exemplos de uso da abordagem; na Seção 4.4, alguns trabalhos relacionados são descritos; e na Seção 4.5, as considerações finais deste capítulo são destacadas. Nota-se que esse capítulo é uma extensão do seguinte artigo: *Towards a Refactoring Catalogue for Knowledge Discovery Metamodel* (DURELLI *et al.*, 2014a).

4.2 Abordagem para Criar Refatorações para o Metamodelo KDM

Nesta seção é apresentada uma abordagem para guiar o engenheiro de modernização durante a criação de refatorações para o metamodelo KDM. Os passos apresentados neste capítulo utilizam um conjunto de artefatos essenciais para a criação e documentação de refatorações para o contexto da ADM e do KDM. Esses artefatos são descritos a seguir:

- Mapeamento entre POO e KDM: Esse artefato mapeia os conceitos do POO para o metamodelo KDM (ver Subseção 4.2.1). Refatorações são aplicadas em elementos de código, tais como: classes, interfaces, atributos, métodos, etc. Assim, esse artefato é de suma importância para guiar os engenheiros de modernização a identificar quais são as metaclasses em KDM que representam os elementos que serão refatorados;

- Operações atômicas/compostas: Esse artefato é utilizado para identificar as operações que compõem uma determinada refatoração (ver Subseção 4.2.2); As refatorações podem ser agrupadas com relação à granularidade: (*i*) operações atômicas e (*ii*) operações compostas;

- Linguagem de Transformação de Modelo: Aqui *templates* são utilizados como artefatos para guiar a criação de refatorações para o KDM. Refatorações são transformações aplicadas em determinados elementos; no contexto desta tese, em instâncias das metaclasses do KDM. Assim, os elementos estruturais juntamente com a linguagem de transformação de modelo formam o sistema de transformação (ver Subseção 4.2.3);

- Linguagem de Restrições: Similarmente, também são utilizados *templates* para guiar a criação de restrições para evitar erros comuns de sintaxe e semântica ao aplicar uma refatoração. Instâncias do metamodelo KDM são entidades não executáveis, assim, se faz necessária a utilização de linguagens de restrições para garantir a correta execução da refatoração antes e depois de sua aplicação (ver Subseção 4.2.5);

- Documentação: Artefato utilizado para documentar as refatorações. As refatorações podem ser informalmente e formalmente apresentadas e documentadas (ver Subseção 4.2.6);

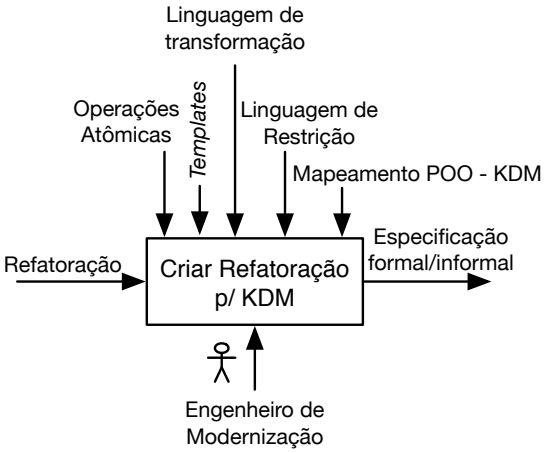
No contexto desta tese, cada refatoração criada para o metamodelo KDM consiste em uma transformação que pode ser executada sobre instâncias desse metamodelo. Além disso, as refatorações também possuem pré- e pós-condições que devem ser satisfeitas antes e depois de sua execução. Por exemplo, a refatoração `Remove ClassUnit` tem como pré-condição a existência de uma instância da metaclasse `ClassUnit`.

É importante destacar que, apenas a semântica e a sintaxe são garantidas e preservadas no contexto desta tese após a aplicação de refatorações no KDM. Garantir que a saída/resultado de um sistema representado por meio do KDM não mude após a aplicação de refatorações não é o objetivo desta tese, uma vez que a instância do KDM não pode ser executada. Por exemplo, no que diz respeito à sintaxe, a abordagem restringe que as metaclasses corretas do KDM sejam utilizadas durante a refatoração, por outro lado, a semântica está relacionada com o fato de não existir duas instâncias iguais de uma mesma metaclasse em um mesmo escopo. Dessa forma, a Definição 1 formaliza refatorações no contexto desta tese:

Definição 1. *Uma refatoração é uma tripla ordenada* $Ref = (pre, T, pos)$ *em que* ***pre*** *é uma asserção (ou seja, pré-condição), que deve ser verdade em uma determinada instância* ***R*** *do metamodelo KDM,* ***T*** *consiste na transformação do modelo* ***R*** *e* ***pos*** *é a pós-condição a ser aplicada após* ***T*** *ser executada.*

Na Figura 23, é exibida uma macrovisão dos elementos/artefatos que são necessários para a criação de refatorações para o KDM, utilizando a notação *Structured Analysis and Design Technique* (SADT) (MARCA; MCGOWAN, 1987). Como observado, um conjunto de elementos são utilizados como base para a criação de refatorações para o KDM: operações atômicas, *templates*, linguagens de transformações, linguagens de restrições e mapeamento POO - KDM.

Figura 23 – Macrovisão para a criação de refatorações para o KDM.

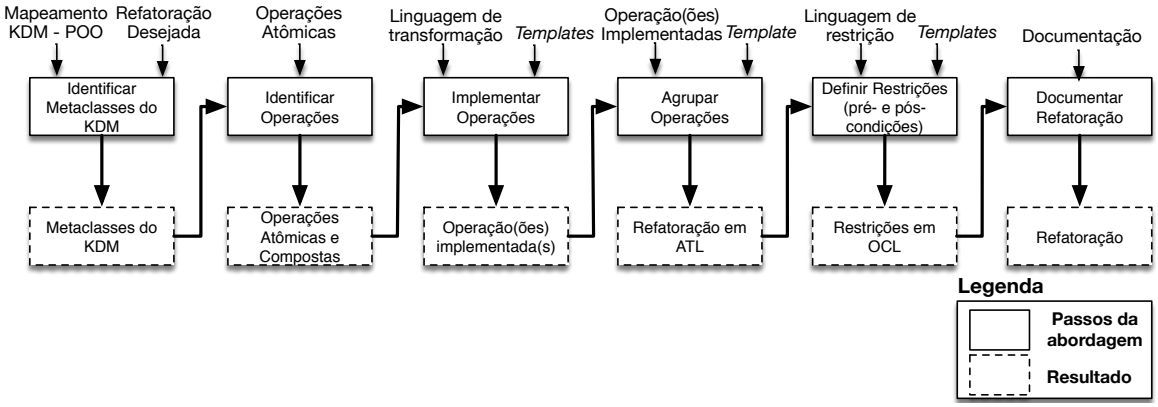


Fonte: Elaborada pelo autor.

Na Figura 24, uma microvisão da Figura 23 é apresentada. É visto que a criação de refatorações para o metamodelo KDM possui seis principais passos que o engenheiro de modernização

deve seguir. Nas subseções a seguir, esses passos são apresentados.

Figura 24 – Passos para criar refatorações para o KDM.



Fonte: Elaborada pelo autor.

4.2.1 Identificar Metaclasses do KDM

Neste passo, o objetivo é identificar as metaclasses do KDM envolvidas na refatoração que será implementada. Os artefatos de entrada para a correta condução deste passo são: (*i*) O Mapeamento KDM - POO e (*ii*) A refatoração desejada. Após a conclusão deste passo, o artefato gerado é uma lista contendo todas as metaclasses que serão utilizadas na criação da refatoração para o KDM.

De acordo com Zhang, Lin e Gray (2005), Boger, Sturm e Fragemann (2003), um dos maiores desafios quando necessita-se adaptar refatorações para um determinado metamodelo é saber quais são as metaclasses corretas que representam determinadas construções/declarações de um determinado paradigma de programação. É importante identificar se a metaclasse escolhida para fazer a refatoração representa realmente o elemento e o conceito do POO. Dessa forma, antes de realizar a criação de qualquer refatoração para o metamodelo KDM, deve-se, primeiramente, identificar as metaclasses do KDM que têm características similares aos conceitos do POO.

O artefato Mapeamento KDM - POO é apresentado na Tabela 3, onde é possível visualizar as metaclasses do metamodelo KDM que possuem características similares aos conceitos/elementos do POO. Utilizando este artefato, engenheiros de modernização podem identificar as metaclasses necessárias para criar as refatorações para o metamodelo KDM. Assim, qualquer exemplo de refatoração definida para o POO pode ser criada para o metamodelo KDM. Como observado, a Tabela 3 contém três colunas: “Elemento do Código-fonte”, “Pacote::metaclasse do KDM” e “Descrição”. A primeira coluna informa a construção da linguagem de programação (*statement*)e/ou o conceito do POO (pacote, classe, interface, etc.) envolvido na refatoração que se deseja implementar. Em seguida, a coluna “Pacote::metaclasse do KDM” apresenta a metaclasse responsável por mapear a construção e/ou o conceito do POO e segue o formato

"*Package*::*Meta-class*". A última coluna, "Descrição", possui informações sobre a metaclasse, tais como seu propósito, seus meta-atributos e suas meta-associações.

Tabela 3 – Mapeamento entre POO e metaclasses do metamodelo KDM.

Início da Tabela		
Construções de Linguagens OO	Pacote::metaclasse do KDM	Descrição
Pacote	code::Package	A metaclasse `Package` é um contêiner para elementos de programa, como classes e interfaces. Essa metaclasse contém um meta-atributo, `name`, que especifica o nome do pacote. Além disso, tal metaclasse possui uma principal meta-associação `codeElement:AbstractCodeElement[0..*]`, na qual pacotes, classes e interfaces podem ser incluídos.
Classe	code::ClassUnit	A metaclasse `ClassUnit` possui dois principais meta-atributos, `name:String` e `isAbstract:boolean`. O primeiro é utilizado para especificar o nome da classe, o segundo é utilizado para informar se a classe é ou não abstrata. Além disso, essa metaclasse possui três meta-associações: `attribute:Attribute[0..*]`, `codeRelation:KDMRelationship[0..*]` e `codeElement:AbstractCodeElement[0..*]`. `attribute` é utilizado para especificar a visibilidade da classe, ou seja, *public*, *private*, ou *protected*. `codeRelation`; agrupa todos os relacionamentos que uma determinada classe possui, por exemplo, heranças e associações. `codeElement` agrupa qualquer metaclasse cujo tipo é uma concretização de `AbstractCodeElement`, como: `StorableUnit`, `MethodUnit`, `MemberUnit`, etc.
Interface	code::InterfaceUnit	A metaclasse `InterfaceUnit` possui características similares as da metaclasse `ClassUnit`, porém, não tem o meta-atributo `isAbstract`, uma vez que todas as interfaces são abstratas por padrão.

Continuação da Tabela 3

Construções de Linguagens OO	Pacote::metaclasse do KDM	Descrição
Atributo	code::StorableUnit	A metaclasse `StorableUnit` possui dois principais meta-atributos: `name:String` e `kind:StorableKind`. Similarmente, essa metaclasse possui dois principais metarrelacionamentos: `attribute:Attribute[..*]` e `type:DataType[1]`. `name` é utilizado para especificar o nome do atributo. `kind` é uma enumeração utilizada para especificar propriedades do atributo, ou seja, informar se ele é local, global, estático, etc. `attribute` é utilizado para definir o escopo do atributo, *public*, *private*, ou *protected*. `type` é utilizado para definir o tipo do atributo.
Método	code::MethodUnit	A metaclasse `MethodUnit` possui dois principais meta-atributos: `name:String` e `kind:MethodKind`. `name` é utilizado para especificar o nome do método. `kind` é uma enumeração utilizada para especificar propriedades do método, ou seja, informar se o método é *construtor*, *destructor*, *virtual*, *abstract*, etc. Similarmente, essa metaclasse possui dois principais metarrelacionamentos: `attribute:Attribute[..*]`, `codeElement:AbstractCodeElement[0..*]`. `attribute` é utilizado para definir o escopo do método - informar ele é *public*, *private*, ou *protected*. `codeElement` é utilizado para agrupar declarações internas do método, ou seja, assinatura do método, bloco do método, etc.
Assinatura do Método	code::Signature	A metaclasse `Signature` possui um principal meta-atributo, `name:String`, o qual é utilizado para especificar o nome do método. Além disso, essa metaclasse contém um principal metarrelacionamento denominado `parameterUnit:ParameterUnit[..*]`, que é utilizado para especificar os parâmetros que o método possui.

Continuação da Tabela 3		
Construções de Linguagens OO	Pacote::metaclasse do KDM	Descrição
Bloco do Método	action::BlockUnit	A metaclasse `BlockUnit` representa blocos lógicos e físicos relacionados, por exemplo, com blocos de instruções *if*, *for*, *while*, etc. Possui um metarrelacionamento denominado `codeElement:AbstractCodeElement[0..*]` que é utilizado para agrupar quaisquer instruções lógicas ou físicas.
Parâmetro	code::ParameterUnit	A metaclasse `ParameterUnit` pode representar o nome, o tipo e a posição dos parâmetros em uma assinatura de método, além de permitir o tipo de parâmetro (valor ou referência). Essa metaclasse contém dois principais meta-atributos: `name:String` e `kind:ParameterKind`. O primeiro meta-atributo representa o nome do parâmetro e o segundo meta-atributo é uma enumeração para especificar o tipo de parâmetro (valor ou referência). Além disso, `ParameterUnit` possui o meta-relacionamento `type:DataType[1]` para especificar o tipo do parâmetro; esse tipo pode ser tipos primitivos ou outros tipos.
Associação	code::HasType	A metaclasse `HasType` representa uma relação semântica entre um elemento de dados e seu tipo. Essa meta-calsse possui dois principais metarrelacionamentos: `from:CodeItem` e `to:DataType`.
Herança `extends`	code::Extends	A metaclasse `Extends` representa relação semântica de herança entre duas `ClassUnits` ou duas `InterfaceUnits`. Essa relação semântica é representada por dois metarrelacionamentos: `from:DataType` e `to:DataType`.
Herança `implements`	code::Implements	A metaclasse `Implements` representa relação semântica de herança entre uma `ClassUnit` e uma `InterfaceUnit`. Similarmente a metaclasse `Extends`, a relação semântica é representada por dois metarrelacionamentos: `from:DataType` e `to:DataType`.

Continuação da Tabela 3		
Construções de Linguagens OO	Pacote::metaclasse do KDM	Descrição
if, *for*, *while*, etc	action::ActionElement	A metaclasse `ActionElement` representa instruções e declarações de uma determinada linguagem de programação, ou seja, pode ser utilizada para representar ramificações, iterações, etc. `ActionElement` possui um meta-atributo principal denominado `kind:String`, que representa qual tipo de instrução que a metaclasse está representando. Essa metaclasse possui dois metarrelacionamentos: `codeElement:AbstractCodeElement[0..*]` e `actionRelation:ActionRelationship[0..*]`.
Fim da Tabela 3		

Como apresentado na Tabela 3, algumas metaclasses podem ser diretamente mapeadas com elementos do POO, tais como: classes (`ClassUnit`), interfaces (`InterfaceUnit`), atributos (`StorableUnit`), métodos (`MethodUnit`), etc. Entretanto, como o KDM é um metamodelo independente de plataforma para representar de forma genérica todas as abstrações e paradigmas de programação, algumas construções de programação não possuem uma metaclasse particular. Por exemplo, iterações e ramificações em KDM são representadas utilizando a mesma metaclasse `ActionElement`. Esse mapeamento ocorre pois o KDM define uma metaclasse genérica para especificar ramificações e iterações para preservar a independência de plataforma; seria inviável para o metamodelo definir metaclasses específicas para representar ramificações e iterações, uma vez que cada linguagem de programação possui uma determinada particularidade. Para que o mapeamento fique o mais genérico e independente de plataforma possível, a metaclasse `ActionElement` utiliza o meta-atributo `kind`, que possui os seguintes valores: *variable declaration*, *if*, *for*, *while*, etc.

Com a utilização da Tabela 3, o engenheiro de modernização pode identificar as metaclasses do KDM que serão utilizadas para implementar a refatoração. Por exemplo, suponha que o engenheiro de modernização almeje criar a refatoração `Rename Method`. Dessa forma, utilizando a Tabela 3 pode-se observar qual é a metaclasse em KDM que representa métodos (métodos em KDM são representados por instâncias da metaclasse `MethodUnit`), assim, a refatoração torna-se `Rename MethodUnit`. Suponha também que a refatoração `Extract Class` também será criada para o KDM. De acordo com Fowler (1999), uma nova classe deve ser criada e em seguida deve-se mover atributos e métodos para essa nova classe - note que as construções de linguagens OO para a refatoração `Extract Class` são: classe, atributo e método. Assim, por meio do artefato Mapeamento KDM - POO descrito na Tabela 3 é possível identificar que

as metaclasses `ClassUnit`, `StorableUnit` e `MethodUnit` deverão ser utilizadas para criar a refatoração `Extract ClassUnit`. Adicionalmente, por meio da Tabela 3, é possível identificar todos os relacionamentos que as as metaclasses possuem, os quais podem ser úteis durante a implementação da refatoração.

4.2.2 Identificar Operações

Neste passo, o engenheiro de modernização deve identificar quais operações são utilizadas na refatoração que será criada para o KDM. Os artefatos de entrada para a correta execução deste passo são um conjunto de operações atômicas. Como saída deste passo, uma lista é criada, a qual contém um conjunto de operações que serão utilizadas para implementar a refatoração.

As refatorações podem ser agrupadas no nível de sua granularidade. As granularidades podem ser definidas em dois níveis de operações: (*i*) operações atômicas e (*ii*) operações compostas. As granularidades definidas como operações atômicas podem ser especificadas por meio de operações primitivas, que são executadas na instância do metamodelo KDM. Tais operações primitivas são descritas no artefato Operações Atômicas, o qual é apresentado na Tabela 4. As operações atômicas e seus efeitos são ilustradas na Figura 25[1].

Tabela 4 – Operações Atômicas utilizadas para criar refatorações para o KDM.

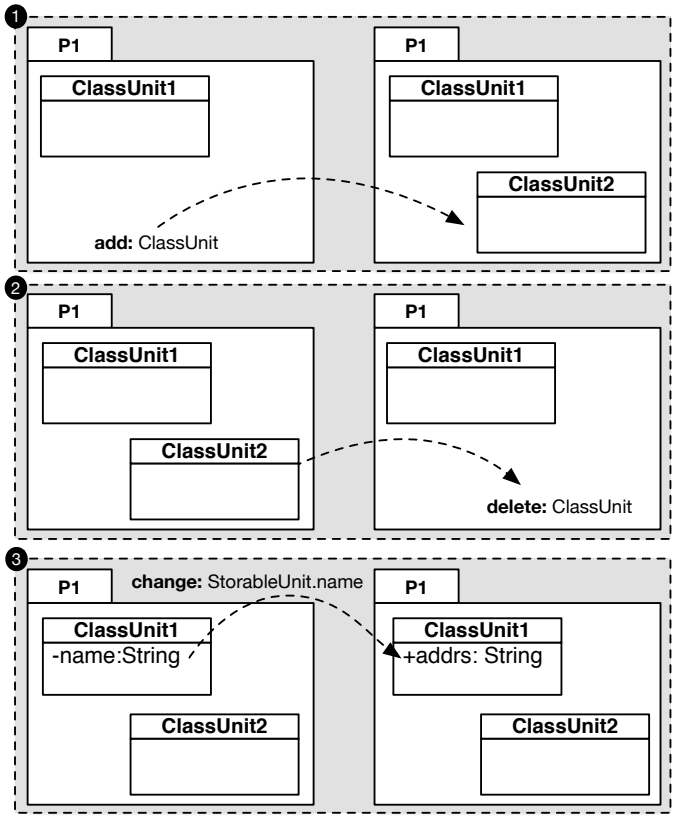
Operações Atômicas	Descrição
`add`	Qualquer operação que adicione uma instância de uma metaclasse do metamodelo KDM (ver Tabela 3). Por exemplo, na Figura 25 ❶ a operação `add` está adicionando uma instância de `ClassUnit` no pacote "P1";
`delete`	Qualquer operação que remove uma instância de uma metaclasse do metamodelo KDM (ver Tabela 3). Por exemplo, na Figura 25 ❷ a operação `delete` remove a instância da metaclasse `ClassUnit` do pacote "P1";
`change`	Qualquer operação que altere um valor de um meta-atributo de uma metaclasse do metamodelo KDM (ver Tabela 3); Por exemplo, na Figura 25 ❸ a operação `change` está alterando o meta-atributo `name` da metaclasse `StorableUnit`; `name` para `addrs`. Além disso, a visibilidade da metaclasse `StorableUnit` também é alterada de `private` para `public`, no diagrama "-" representa a visibilidade `private` e "+" representa `public`.

Utilizando o artefato apresentado na Tabela 4, engenheiros de modernização podem criar refatorações atômicas, ou seja, adicionar, deletar e alterar qualquer instância de metaclasse e meta-atributos do KDM.

As refatorações compostas consistem em uma combinação de operações atômicas. Por exemplo, considerando a refatoração `Move ClassUnit`, ela compõe-se de duas operações atô-

[1] Para exemplificar o funcionamento das operações atômicas no KDM, diagramas de classes da UML foram utilizados.

Figura 25 – Operações atômicas para o KDM.

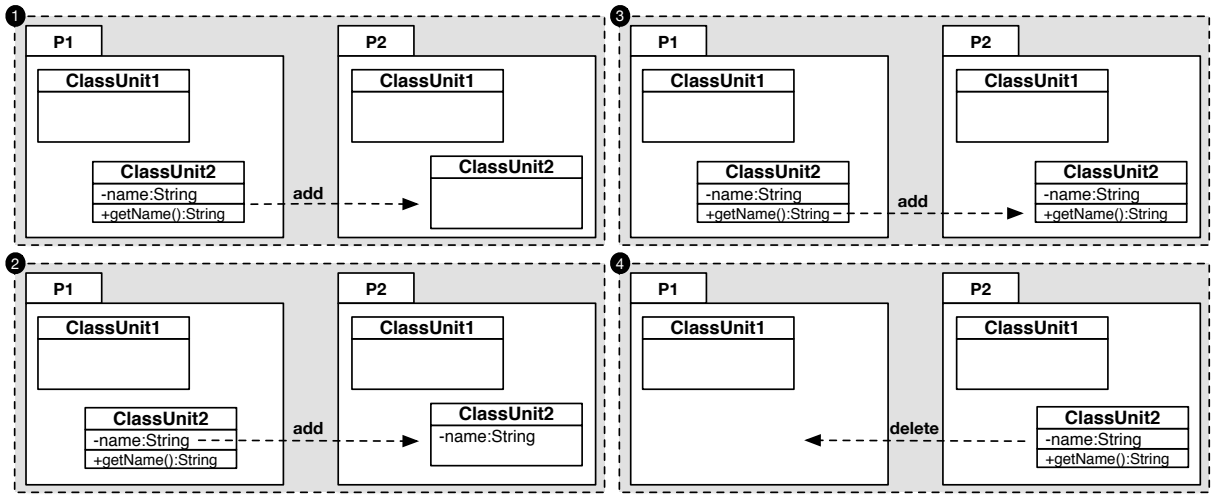


Fonte: Elaborada pelo autor.

micas: `add` e `delete`. Assim, primeiro, uma nova instância da metaclasse `ClassUnit` deve ser adicionada no pacote onde se almeja mover essa instância (ver Figura 26 ❶). Porém, usualmente, a classe a ser movida contém atributos e/ou métodos (`StorableUnits` e/ou `MethodUnits`). Dessa forma, o engenheiro de modernização pode utilizar o Algoritmo 1 para verificar quais operações atômicas precisam ser implementadas. Esse algoritmo verifica se as instâncias das metaclasses `Package`, `InterfaceUnit` e `ClassUnit` possuem elementos internos (hierarquia); no caso da metaclasse `Package`, pode-se haver pacotes internos e/ou classes; na instância da metaclasse `ClassUnit`, pode-se conter `ClassUnits`, `StorableUnits` e/ou `MethodUnits`; e na instância da metaclasse `InterfaceUnit`, pode-se haver também `StorableUnits` e/ou `MethodUnits`.

Por exemplo, ainda considerando os exemplos ilustrados na Figura 26, é possível notar que a instância da metaclasse `ClassUnit` a ser movida possui atributos e/ou métodos (`StorableUnits` e/ou `MethodUnits`), então, tais atributos e/ou métodos, também devem ser movidos (`add` e `delete`) (ver Algoritmo 1). Na Figura 26 ❷, uma instância da metaclasse `StorableUnit` denominada `name`, é criada e adicionada na instância da metaclasse `ClassUnit`, a qual foi criada na Figura 26 ❶. Similarmente, uma instância da metaclasse `MethodUnit` denominada `getName()`, também é criada e adicionada na instância da metaclasse `ClassUnit` (ver Figura 26 ❸). Caso a instância da `ClassUnit` que se almeja mover possua *n* elementos, então,

Figura 26 – Operações compostas para o KDM.



Fonte: Elaborada pelo autor.

Algoritmo 1: Algoritmo para criar refatorações compostas.

Input: (M, N) onde *M* é uma instância da metaclasse do KDM a ser movida, *N* é uma instância da metaclasse do KDM movida
Output: Uma lista de operações primitivas (add, delete)

```
1  begin
2      if M instanceOf Package then
3          foreach M.codeElement() do
4              listPrimitives.put(add)
5          end
6      end
7      else if M instanceOf InterfaceUnit then
8          foreach M.codeElement() do
9              listPrimitives.put(add)
10         end
11     end
12     else if M instanceOf ClassUnit then
13         foreach M.codeElement() do
14             listPrimitives.put(add)
15         end
16     end
17     listPrimitives.put(delete)
18     return listPrimitives
19 end
```

esse processo deve-se repetir *n* vezes (ver Algoritmo 1). Após adicionar todos os atributos e/ou métodos na instância da `ClassUnit`, a instância da metaclasse `ClassUnit` contida no pacote "P1" pode ser removida e, assim, seus atributos e/ou métodos também são deletados.

Por meio da junção dessas operações atômicas, é possível criar refatorações já existentes

na literatura para o KDM. Por exemplo, podem-se focar as refatorações propostas por Fowler (1999), as quais podem ser facilmente criadas para o KDM. Na Tabela 5, algumas refatorações propostas por Fowler (1999) são apresentadas ressaltando a classificação da refatoração e qual(is) a(s) operação(ões) deve(m) ser utilizada(s) para criar a refatoração. A última coluna da Tabela 5 apenas ilustra as operações atômicas que serão utilizadas e não a quantidade de operações atômicas para efetivamente criar a refatoração. Nota-se que as refatorações apresentadas nessa tabela seguem a mesma convenção de nomenclatura definida por Fowler (1999). Porém, os nomes de algumas refatorações foram alterados para indicar o metamodelo KDM como seu novo domínio de aplicação, por exemplo, `MoveMethod` torna-se `MoveMethodUnit` e `MoveAttribute` torna-se `MoveStorableUnit`, etc.

Tabela 5 – Refatorações criadas para o metamodelo KDM.

Refatoração	Classificação	Tipo de Operação
Add Package	Atômica	`add`
Add ClassUnit	Atômica	`add`
Add StorableUnit	Atômica	`add`
Add MethodUnit	Atômica	`add`
Delete Package	Atômica	`delete`
Delete ClassUnit	Atômica	`delete`
Delete StorableUnit	Atômica	`delete`
Delete MethodUnit	Atômica	`delete`
Rename Package	Atômica	`change`
Rename ClassUnit	Atômica	`change`
Rename StorableUnit	Atômica	`change`
Rename MethodUnit	Atômica	`change`
Move StorableUnit	Composta	`add` | `delete`
Move MethodUnit	Composta	`add` | `delete`
Extract ClassUnit	Composta	`add` | `delete` | `change`
Inline ClassUnit	Composta	`change` | `delete`
Flatten Hierarchy	Composta	`add` | `delete` | `change`
Push Down MethodUnit	Composta	`add` | `delete`
Push Down StorableUnit	Composta	`add` | `delete`
Pull Up MethodUnit	Composta	`add` | `delete`
Pull Up StorableUnit	Composta	`add` | `delete`
Extract SubClass	Composta	`add` | `change`
Encapsulate StorableUnit	Composta	`add` | `change`

Após o engenheiro de modernização escolher qual refatoração irá criar para o metamodelo KDM e identificar qual(is) é(são) a(s) operação(ões) que compõe(m) a refatoração escolhida, o próximo passo é a implementação da refatoração utilizando técnicas e linguagem de transformação em modelo.

4.2.3 Implementar Operações

Neste passo, o engenheiro de modernização deve implementar as operações atômicas. Os artefatos de entrada para a correta condução deste passo são: (*i*) linguagem de transformação e (*ii*) um conjunto de *templates*. Após a condução deste passo, o artefato gerado é uma lista de operações atômicas implementadas em ATL.

Como apresentado na Seção 4.2.2, as refatorações no contexto desta tese são agrupadas em dois níveis de operações: (*i*) operações atômicas e (*ii*) operações compostas. As granularidades definidas como operações atômicas são: `add`, `delete` e `change`. Essas operações atômicas podem ser facilmente implementadas em ATL. Dessa forma, um *template* para cada operação atômica é disponibilizado como artefato para auxiliar o engenheiro de modernização a criar refatorações para o KDM. O *template* para a operação atômica `add` é apresentado no Código-fonte 4. As partes fixas dos *templates* são formadas por texto ATL e as partes variantes são formadas por três tipos de instruções: (*i*) argumentos, **ArgX**[2], demarcados pelos símbolos "**<#**" e "**#>**" e que devem ser substituídos por *strings* válidas na linguagem ATL; (*ii*) argumentos, **ArgX**[2], demarcados pelos símbolos "**<%**" e "**%>**" e que devem ser substituídos por metaclasses do metamodelo KDM, por exemplo, `ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc; (*iii*) argumentos, **ArgX**[2], demarcados pelos símbolos "**<@**" e "**@>**", que devem ser substituídos de acordo com o domínio do sistema que será aplicada a refatoração, por exemplo, nomes de pacotes, classes, interfaces, atributos, métodos, etc.

Código-fonte 4: *Template* ATL para realizar a operação atômica `add`.

```
1  rule create<#Arg1#>{
2    from
3      source : MM!<%Arg2%> (source.name = <@Arg3@>)
4    to
5      target: MM!<%Arg2%> (
6        codeElement ← source.codeElement→including(newElement)
7      ),
8      newElement: MM!<%Arg4%> (
9        name ← <@Arg5@>
10     )
11 }
```

Além do *template* apresentado no Código-fonte 4, outro artefato também deve ser utilizado para auxiliar o engenheiro de modernização a substituir as partes variáveis do *template* de forma correta. Esse artefato é apresentado na Tabela 6, o qual é utilizado como um guia para conduzir o engenheiro de modernização a especificar corretamente as partes variantes do *template*, ou seja, os argumentos.

[2] **ArgX**, onde **X** representa um número sequencial dos argumentos.

Tabela 6 – Guia para auxiliar a substituição dos argumentos do *template* apresentado no Código-fonte 4.

Argumentos	Valores
<#Arg1#>	Nome da regra. Pode-se utilizar qualquer *string* válida na linguagem ATL.
<%Arg2%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclasse que conterá a nova instância da metaclasse a ser criada.
<@Arg3@>	Nome da instância do elemento estrutural especificado no **<%Arg2%>**. Deve-se especificar o nome da instância da metaclasse que conterá a nova instância da metaclasse a ser criada. Esse argumento depende do domínio do sistema que será aplicada a refatoração e é identificada no meta-atributo `name` das metaclasses do KDM.
<%Arg4%>	Nome de uma metaclasse que será instanciada. Deve-se especificar o nome da metaclasse que será criada.
<@Arg5@>	Nome da nova instância que será criada. Deve-se especificar o nome da instância da metaclasse que será criada. Esse argumento também depende do domínio do sistema que será aplicado à refatoração.

Dado o *template* apresentado no Código-fonte 4, bem como as Tabelas 3 e 6, a combinação desses três artefatos auxiliam/guiam o engenheiro de modernização a criar a operação atômica `add` para um determinado elemento estrutural do KDM, ou seja, `ClassUnit`, `InterfaceUnit`, `Package`, `StorableUnit`, `MethodUnit`, etc. Por exemplo, no Código-fonte 5 é apresentada uma simples ATL que foi criada utilizando esses três artefatos. O argumento **<#Arg1#>** foi substituído pela *string* `Class`, respectivamente, ver linha 1. Na linha 3, **<%Arg2%>** foi substituído pela metaclasse `Package` e o argumento **<@Arg3@>** foi substituído pela *String* `"com.br.teste"`, a qual representa o nome da instância da metaclasse `Package` onde uma nova instância da metaclasse `ClassUnit` será adicionada. Finalmente, os argumentos **<%Arg4%>** e **<@Arg5@>** foram substituídos pela metaclasse `ClassUnit` e pela *String* `"novaClassUnit"`, ver linhas 8 e 9. A execução desse código cria uma nova instância da metaclasse `ClassUnit` denominada "novaClassUnit" no pacote "com.br.teste".

Código-fonte 5: ATL para realizar a operação atômica `add ClassUnit`.

```
1  rule createClass{
2    from
3      source : MM!Package (source.name = "com.br.teste")
4    to
5      target: MM!Package (
6        codeElement ← source.codeElement→including(newElement)
7      ),
8      newElement: MM!ClassUnit (
9        name ← "novaClassUnit"
10     )
11 }
```

Similarmente, o engenheiro de modernização pode implementar a operação atômica `delete` facilmente seguindo um *template*. O *template* responsável pela operação atômica `delete` é apresentado no Código-fonte 6[3]. Esse *template* também possui partes fixas e partes variantes. As partes fixas são textos em ATL e as partes variantes são formadas por argumentos demarcados pelas instruções "**<#**" e "**#>**", "**<%**" e "**%>**" e "**<@**" e "**@>**".

O guia para auxiliar o engenheiro de modernização a especificar corretamente as partes variantes do *template* `delete` é apresentado na Tabela 7. Por meio de três artefatos: (*i*) o *template* representado no Código-fonte 6, (*ii*) o mapeamento entre OO e KDM apresentado na Tabela 3, (*iii*) e o guia apresentado na Tabela 7, o engenheiro de modernização pode criar a operação atômica `delete`. Por exemplo, pode-se considerar Código-fonte 7, onde almeja-se remover uma instância da metaclasse `ClassUnit`.

Código-fonte 6: *Template* ATL para realizar a operação atômica `delete`.

```
1 rule delete<#Arg1#> {
2   from
3       source : MM!<%Arg2%> (source.name = <@Arg3@> and source.
      refImmediateComposite().name = <@Arg4@>
4   to
5       drop
6 }
```

Tabela 7 – Guia para auxiliar a substituição dos argumentos do *template* apresentado no Código-fonte 6.

Argumentos	Valores
<#Arg1#>	Nome da regra. Pode-se utilizar qualquer *string* válida na linguagem ATL.
<%Arg2%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclasse que representa a instância da metaclasse a ser deletada.
<@Arg3@>	Nome da instância do elemento estrutural especificado no **<%Arg2%>**. Deve-se especificar o nome da instância da metaclasse que representa a instância da metaclasse a ser deletada. Esse argumento depende do domínio do sistema que será aplicada à refatoração e é identificada no meta-atributo `name` das metaclasses do KDM.
<@Arg4@>	Nome da instância do elemento estrutural que contém o elemento especificado no **<%Arg2%>**. Deve-se especificar o nome da instância da metaclasse que possui a instância da metaclasse a ser deletada. Esse argumento depende do domínio do sistema que será aplicada à refatoração e é identificada no meta-atributo `name` das metaclasses do KDM.

Código-fonte 7: ATL para realizar a operação atômica `delete ClassUnit`.

[3] A palavra-chave `drop` é utilizada na linha 5 do Código-fonte 6 para especificar que uma determinada instância será removida.

```
1 rule deleteClass {
2   from
3       source : MM!ClassUnit (source.name = "ClassToRemove" and source.
      refImmediateComposite().name = "PackageThatContainsTheClassToRemove")
4   to
5       drop
6 }
```

O argumento **<#Arg1#>** foi substituído pela *String* `Class` (ver linha 1). Depois, na linha 3 o argumento **<%Arg2%>** foi substituído pela metaclasse `ClassUnit` e o argumento **<@Arg3@>** foi substituído pela *String* `"ClassToRemove"`. Essa *String* representa o nome da instância da `ClassUnit` que se almeja deletar. O argumento **<@Arg4@>** foi substituído pela *String*`"PackageThatContainsTheClassToRemove"`, a qual representa o pacote que contém a classe que será deletada.

Finalmente, a operação atômica `change` também pode ser implementada por meio de *template*. O Código-fonte 8 representa o *template* da operação atômica `change`. Esse *template* também possui partes fixas e instruções demarcadas por "**<#**" e "**#>**", "**<%**" e "**%>**" e "**<@**" e "**@>**" que representam as partes variantes.

Código-fonte 8: *Template* para realizar a operação atômica `change`.

```
1 rule change<#Arg1#> {
2   from
3     source : MM!<%Arg2%> (source.name=<@Arg3@>)
4   to
5     target : MM!<%Arg2%> (
6       <%Arg4%> ← <@Arg5@>
7     )
8 }
```

Na Tabela 8, o guia para auxiliar o engenheiro de modernização a especificar corretamente as partes variantes do *template* `change` é apresentado. A junção dos três artefatos: (*i*) o *template* apresentado no Código-fonte 8, (*ii*) o mapemanto entre OO e KDM (ver Tabela 3) e (*iii*) o guia apresentado na Tabela 8 auxilia o engenheiro de modernização a criar a operação atômica `change`. Por exemplo, pode-se considerar o Código-fonte 9 onde esses três artefatos foram utilizados pelo engenheiro de modernização para implementar a operação atômica `change`. Nesse código-fonte, uma determinada instância da metaclasse `Package` é alterada, ou seja, seu meta-atributo `name` é renomeado. Assim, o argumento **<#Arg1#>** foi substituído pela *string* `Package` (linha 1). Na linha 3 **<%Arg2%>** foi substituído pela metaclasse `Package` e o argumento **<@Arg3@>** foi substituído pela *String* `"PackageToRename"`, a qual representa o nome da instância da metaclasse `Package` onde uma nova instância da metaclasse `ClassUnit` será

Tabela 8 – Guia para auxiliar a substituição dos argumentos do *template* apresentado no Código-fonte 8.

Argumentos	Valores
<#Arg1#>	Nome da regra. Pode-se utilizar qualquer *string* válida na linguagem ATL.
<%Arg2%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclasse que conterá a instância da metaclasse a ser alterada.
<@Arg3@>	Nome da instância do elemento estrutural especificado no **<%Arg2%>**. Deve-se especificar o nome da instância da metaclasse a ser alterada. Esse argumento depende do domínio do sistema que será aplicada à refatoração e é identificada no meta-atributo `name` das metaclasses do KDM.
<%Arg4%>	Nome do(s) meta-atributo(s) da metaclasse que será alterada. Deve-se especificar o(s) nome(s) do(s) meta-atributo(s) que será(ão) alterado(s).
<@Arg5@>	Novo(s) valor(es) para ser setado(s) no(s) meta-atributo(s) especificado(s) no **<%Arg4%>**. Esse argumento também depende do domínio do sistema que será aplicada à refatoração.

alterada. O argumento **<%Arg4%>** foi substituído pelo meta-atributo `name` e o **<@Arg5@>** foi substituído por uma *String* que representa o novo nome da instância da metaclasse `Package` (ver linha 6).

Código-fonte 9: ATL para realizar a operação atômica `change ClassUnit`.

```
1 rule changePackage {
2   from
3     source : MM!Package (source.name="PackageToRename")
4   to
5     target : MM!Package (
6       name ← "newName"
7     )
8 }
```

4.2.4 Agrupar Operações

Neste passo, o engenheiro de modernização deve agrupar as operações atômicas criadas no passo anterior para formar a refatoração para o KDM. Os artefatos de entrada para a correta execução deste passo são: (*i*) as operações atômicas implementadas e (*ii*) um *template* para guiar o engenheiro a agrupar as operações atômicas e formar a refatoração. Após a conclusão deste passo, o artefato gerado é uma refatoração escrita em ATL para o KDM.

Como apresentado na Tabela 5, com a combinação dessas operações atômicas os engenheiros de modernização podem criar refatorações mais significativas. Por exemplo, o Código-fonte 5 e o Código-fonte 7 ilustram as operações atômicas `add` e `delete` para instâncias da

metaclasse `ClassUnit`, respectivamente, assim, a combinação dessas duas operações atômica resulta na refatoração `move`.

Para guiar o engenheiro de modernização a agrupar as operações atômicas e criar a refatoração, um *template* deve ser utilizado. Esse *template* é apresentado no Código-fonte 10, o qual contém partes fixas e instrução demarcada por "**<#**" e "**#>**" que que deve ser substituída por *strings* válidas na linguagem ATL. Além disso, esse *template* também contém trechos em pseudocódigo (**para cada**, **se**, **senão se**, etc) para auxiliar e guiar o engenheiro de modernização a agrupar corretamente as operações atômicas implementadas no passo anterior. Assim, para cada operação atômica implementada, deve-se verificar se ela é do tipo `add`, `delete` e/ou `change`. Esse processo deve ser repetido até não existir mais operações atômicas para serem copiadas ao *template* apresentado no Código-fonte 10. O argumento **<#Arg1#>** pode ser substituido por qualquer *string* válida na linguagem ATL.

Código-fonte 10: *Template* ATL para agrupar as operações atômicas.

```
1  module <#Arg1#>;
2  create OUT : MM refining IN : MM;
3  [para cada operações atômicas faça]
4        [se operação atômica == add então]
5              adiciona o template preenchido do Código-fonte 4
6        [senão se operação atômica == delete então]
7              adiciona o template preenchido do Código-fonte 6
8        [senão se operação atômica == change então]
9              adiciona o template preenchido do Código-fonte 8
10       [/fim se]
11 [/fim para]
```

Após a criação de uma refatoração, o engenheiro de modernização deve especificar as restrições da refatoração criada. Tais restrições são especificadas utilizando linguagens de restrições, por exemplo, OCL. Na próxima seção, maiores informações são apresentadas.

4.2.5 Definir Restrições (Pré- e Pós-condições)

Neste passo, o engenheiro de modernização deve implementar as restrições (pré- e pós-condições). Os artefatos de entrada para a correta condução deste passo são: (*i*) linguagem de restrição e (*ii*) um conjunto de *templates*. Após a condução deste passo, o artefato gerado é uma lista de restrições implementadas em OCL.

Após o engenheiro de modernização criar uma determinada refatoração, o próximo passo consiste na criação de restrições (pré- e pós-condições) para a refatoração. Usualmente, antes e após a aplicação de uma determinada refatoração, algumas restrições precisam ser satisfeitas. Tais restrições geralmente são úteis para verificar se os parâmetros necessários para executar a refatoração foram corretamente informados, bem como verificar se a refatoração foi aplicada de

forma correta. No contexto de modelos, tais restrições são especificadas utilizando linguagens como OCL e são caracterizadas como pré- e pós-condições. Além disso, essas restrições são importantes para assegurar que a refatoração será aplicada de forma correta e ainda irá preservar a semântica da instância do metamodelo. Por exemplo, verificar se uma determinada instância de uma metaclasse realmente existe antes de aplicar a operação atômica `delete`, ou verificar se uma determinada instância já existe antes de realizar a operação atômica `add`. Diante disso, cada refatoração (operação atômica) definida nesta tese está associada com uma pré- e pós-condição definida em OCL. OCL foi escolhida pois é uma linguagem padronizada pelo OMG e também possui suporte no ambiente de desenvolvimento Eclipse.

Para auxiliar o engenheiro de modernização durante a criação dessas restrições (pré- e pós-condições) *templates* também foram definidos. Os *templates* aqui apresentados estão associados a uma determinada operação atômica. Por exemplo, para cada operação atômica (`add`, `delete` e `change`) dois *templates* foram definidos para auxiliar a criação das restrições, ou seja, um *template* para auxiliar a criação da pré-condição e outro para auxiliar a criação da pós-condição.

Os *templates* para a operação atômica `add` são apresentados nos Códigos-fontes 11 e 12, onde o primeiro código ilustra o *template* para criar a pré-condição e o segundo representa o *template* para especificar a pós-condição da operação atômica `add`. As partes fixas dos *templates* são formadas por texto em OCL e as partes variantes são formadas por três tipos de instruções: (*i*) argumentos, **ArgX**[4], demarcados pelos símbolos "**<#**" e "**#>**" e que devem ser substituídos por *strings* válidas na linguagem OCL; e (*ii*) argumentos, **ArgX**[4], demarcados pelos símbolos "**<%**" e "**%>**" e que devem ser substituídos por elementos/metaclasses do metamodelo KDM, por exemplo, `ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc; e (*iii*) argumentos, **ArgX**[2], demarcados pelos símbolos "**<@**" e "**@>**" que devem ser substituídos de acordo com o domínio do sistema que será aplicado à refatoração, por exemplo, nomes de pacotes, classes, interfaces, atributos, métodos, etc.

Código-fonte 11: *Template* OCL para realizar a pré-condição da operação atômica `add`.

```
1 context <%Arg1%>::<#Arg2#>(newName: String)
2 pre : <%Arg3%>.allInstances->select(e : <%Arg3%> | a.name = <@Arg4@>)
      and not <%Arg1%>.refImmediateComposite().codeElement->exist (e :
      <%Arg1%> | e.name = newName)
```

O *template* apresentado no Código-fonte 11 tem como função verificar se uma determinada instância de uma metaclasse a ser criada pela operação atômica `add` ainda não existe na instância do KDM. Isso é importante para garantir que não existam duas instâncias iguais no KDM. Caso positivo a operação atômica pode ser executada. A pós-condição apresentada no

[4] **ArgX**, onde **X** representa um número sequencial dos argumentos.

Código-fonte 12, por outro lado verifica se a instância de uma metaclasse realmente foi criada pela execução da operação atômica `add`.

Código-fonte 12: *Template* OCL para realizar a pós-condição da operação atômica `add`.

```
1 context <%Arg1%>::<#Arg2#>(newName: String)
2 post : <%Arg3%>.allInstances->select(e : <%Arg3%> | a.name = <@Arg4@>)
      and <%Arg1%>.refImmediateComposite().codeElement→exist (e :
     <%Arg1%> | e.name = newName)
```

Tabela 9 – Guia para auxiliar a substituição dos argumentos dos *templates* apresentados nos Códigos-fontes 11 e 12.

Argumentos	Valores
<%Arg1%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclassse que foi criada pela operação atômica `add`.
<#Arg2#>	Nome da restrição. Pode-se utilizar qualquer *string* válida em OCL.
<%Arg3%>	Nome de uma metaclasse que contém o elemento especificado no **<%Arg1%>**. Por exemplo, se o **<%Arg1%>** for uma `ClassUnit` e/ou `InterfaceUnit` deve-se especificar a metaclasse `Package`.
<@Arg4@>	Nome da instância do elemento estrutural que contém o elemento especificado no **<%Arg3%>**. Esse argumento depende do domínio do sistema que será aplicado à refatoração.

Além dos *templates*, outro artefato também deve ser utilizado para auxiliar o engenheiro de modernização a criar as restrições (pré- e pós-condições). Esse artefato é apresentado na Tabela 9, a qual é utilizada como guia para conduzir o engenheiro de modernização a especificar corretamente as partes variantes dos *templates*. Por exemplo, pode-se considerar que o engenheiro de modernização deseja criar a operação atômica `add ClassUnit`. Desse modo, utilizando os *templates* apresentados nos Códigos-fontes 11 e 12, bem como a Tabela 9 é possível criar as restrições dessa operação atômica como apresentado no Código-fonte 13. O argumento **<%Arg1%>** foi substituído pela metaclasse `ClassUnit` e o argumento **<#Arg2#>** foi alterado por uma *String* válida em OCL. O argumento **<%Arg3%>** foi substituído pela metaclasse `Package` e o argumento **<@Arg4@>** foi substituído por uma *String* que representa o pacote ("com.br.util") onde a instância da metaclasse `ClassUnit` será adicionada.

Código-fonte 13: Asserções em OCL para realizar a operação atômica `add`.

```
1 context ClassUnit::preCond(newName: String)
2 pre : Package.allInstances->select(e : Package | a.name = "com.br.util") and
     not ClassUnit.refImmediateComposite().codeElement→exist (e :
     ClassUnit | e.name = newName)
3 context ClassUnit::postCond(newName: String)
4 post : Package.allInstances->select(e : Package | a.name = "com.br.util") and
      ClassUnit.refImmediateComposite().codeElement→exist (e : ClassUnit
     | e.name = newName)
```

Da mesma forma o engenheiro de modernização pode implementar as restrições para a operação atômica `delete` seguindo *templates*. Os *templates* para auxiliar o engenheiro de modernização a implementar as restrições para a operação atômica `delete` são apresentados nos Códigos-fontes 14 e 15. Similarmente, esses *templates* também possuem partes fixas e instruções demarcadas por **<%** e **%>**, **<#** e **#>** e **<$@** e **$@>** que representam as partes variantes.

Código-fonte 14: *Template* OCL para realizar a pré-condição da operação atômica `delete`.

```
1 context <%Arg1%>::<#Arg2#>(Ename: String)
2 pre : <%Arg3%>.allInstances->select(e : <%Arg3%> | a.name = <@Arg4@>)
      and <%Arg1%>.refImmediateComposite().codeElement→exist (e :
      <%Arg1%> | e.name = Ename)
```

Código-fonte 15: *Template* OCL para realizar a pós-condição da operação atômica `delete`.

```
1 context <%Arg1%>::<#Arg2#>(Ename: String)
2 post : <%Arg3%>.allInstances->select(e : <%Arg3%> | a.name = <@Arg4@>)
       and not <%Arg1%>.refImmediateComposite().codeElement→exist (e :
      <%Arg1%> | e.name = Ename)
```

O *template* apresentado no Código-fonte 14 busca verificar se uma determinada instância de uma metaclasse a ser deletada pela operação atômica `delete` existe na instância do KDM. Caso positivo a operação atômica pode ser então executada. Por outro lado, a pós-condição apresentada no Código-fonte 15, verifica se a instância da metaclasse realmente foi deletada após a execução da operação atômica `delete`.

Tabela 10 – Guia para auxiliar a substituição dos argumentos dos *templates* apresentados nos Códigos-fontes 14 e 15.

Argumentos	Valores
<%Arg1%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclassse que será deletada pela operação atômica `delete`.
<#Arg2#>	Nome da restrição. Pode-se utilizar qualquer *string* válida em OCL.
<%Arg3%>	Nome de uma metaclasse que contém o elemento especificado no **<%Arg1%>**. Por exemplo, se o **<%Arg1%>** for uma `ClassUnit` e/ou `InterfaceUnit` deve-se especificar a metaclasse `Package`.
<@Arg4@>	Nome da instância do elemento estrutural que contém o elemento especificado no **<%Arg3%>**. Esse argumento depende do domínio do sistema que será aplicado à refatoração.

O artefato utilizado para auxiliar e conduzir o engenheiro de modernização especificar corretamente as partes variantes dos *templates* apresentados nos Códigos-fontes 14 e 15 é apresentado na Tabela 10.

Código-fonte 16: Asserções em OCL para realizar a operação atômica `delete`.

```
1 context StorableUnit::preCond(Ename: String)
2 pre : ClassUnit.allInstances->select(e : ClassUnit | a.name = "Foo") and
      StorableUnit.refImmediateComposite().codeElement→exist (e :
      StorableUnit | e.name = Ename)
3 context StorableUnit::postCond(Ename: String)
4 post : ClassUnit.allInstances->select(e : ClassUnit | a.name = "Foo") and
       not StorableUnit.refImmediateComposite().codeElement→exist (e :
      StorableUnit | e.name = Ename)
```

No Código-fonte 16, é apresentado as asserções da operação atômica `delete` quando almeja-se deletar uma instância da metaclasse `StorableUnit`. Note que o argumento **<%Arg1%>** foi substituído pela metaclasse `StorableUnit` e **<#Arg2#>** foi alterado por uma *String* válida em OCL. **<%Arg3%>** foi substituído pela metaclasse `ClassUnit` e o argumento **<@Arg4@>** foi substituído por uma *String* que representa a classe ("Foo") e que contém a instância da metaclasse `StorableUnit` que será deletada.

Código-fonte 17: *Template* OCL para realizar a pré-condição da operação atômica `change`.

```
1 context <%Arg1%>::<#Arg2#>(el: String | Integer | Real | Boolean)
2 pre : <%Arg1%>.<%Arg3%> <> el
```

Finalmente, as asserções para a operação atômica `change` também podem ser implementadas seguindo os *templates* apresentados nos Códigos-fontes 17 e 18. O *template* apresentado no Código-fonte 17 visa verificar se um determinado meta-atributo de uma metaclasse é diferente do valor que almeja-se alterar pela operação atômica `change`. A pós-condição apresentada no Código-fonte 18 busca verificar se o meta-atributo realmente foi alterado pela execução da operação atômica `change`.

Código-fonte 18: *Template* OCL para realizar a pré-condição da operação atômica `change`.

```
1 context <%Arg1%>::<#Arg2#>(el: String | Integer | Real | Boolean)
2 post : <%Arg1%>.<%Arg3%> = el
```

O guia utilizado pelo engenheiro de modernização para especificar corretamente as partes variantes dos *templates* apresentados nos Códigos-fontes 17 e 18 é apresentado na Tabela 11.

No Código-fonte 19, são apresentadas as asserções da operação atômica `change` quando almeja-se alterar o meta-atributo `name` de uma instância da metaclasse `Package`. O argumento **<%Arg1%>** foi substituído pela metaclasse `Package` e o argumento **<#Arg2#>** foi alterado por uma *String* válida em OCL. O argumento **<%Arg3%>** foi substituído pelo meta-atributo `name` da metaclasse `Package`.

Tabela 11 – Guia para auxiliar a substituição dos argumentos dos *templates* apresentados nos Códigos-fontes 17 e 18.

Argumentos	Valores
<%Arg1%>	Nome de uma metaclasse (`ClassUnit`, `InterfaceUnit`, `StorableUnit`, `MethodUnit`, etc). Deve-se especificar o nome da metaclassse que almeja-se alterar um meta-atributo por meio da operação atômica `change`.
<#Arg2#>	Nome da restrição. Pode-se utilizar qualquer *string* válida em OCL.
<%Arg3%>	Nome do meta-atributo a ser alterado. Deve-se especificar o meta-atributo da metaclasse especificado no **<%Arg1%>** que almeja-se alterar.

Código-fonte 19: Asserções em OCL para realizar a operação atômica `change`.

```
1 context Package::preCond(el: String | Integer | Real | Boolean)
2 pre : Package.name <> el
3 context Package::postCond(el: String | Integer | Real | Boolean)
4 post : Package.name = el
```

4.2.6 Documentar Refatoração

O último passo da abordagem é a documentação da refatoração. Esse último passo é opcional e fica a critério do engenheiro de modernização documentar ou não a refatoração criada. Caso o engenheiro de modernização opte por disponibilizar uma documentação pode então seguir as especificações definidas nesse passo.

Dessa forma, após implementar a refatoração, bem como suas pré- e pós-condições para o metamodelo KDM, é importante documentar a refatoração criada. Com o intuito de facilitar a visualização, formalização e entendimento das refatorações criadas para o metamodelo KDM, os engenheiros de modernização podem especificar as refatorações criadas seguindo duas especificações (STARON; KUŹNIARZ, 2004): (*i*) especificação informal e (*ii*) especificação formal.

Na especificação informal, a lógica da refatoração é expressada e frequentemente definida por meio de linguagem natural. A especificação formal é responsável por representar as pré- e pós-condições, bem como a transformação/refatoração em OCL e ATL, respectivamente. Ambas especificações são úteis para o engenheiro de modernização, pois a especificação informal é utilizada para facilitar a compreensão e o propósito da refatoração e a especificação formal facilita a implementação da refatoração. Além disso, a especificação formal é de extrema importância para facilitar a automação das refatorações. As refatorações criadas para o metamodelo KDM devem ser especificadas utilizando os seguintes *templates*:

1. Especificação Informal:

    a) Nome: o nome da refatoração;

b) Definição: uma lista contendo os parâmetros utilizados na refatoração - após a definição dos parâmetros, eles são utilizados e referenciados dentro das especificações formal e informal por meio de {...};

c) Objetivo: o objetivo da refatoração;

d) Descrição (opcional): uma pequena explicação sobre a refatoração;

e) Pré-condição: uma lista de asserções que tem que ser verdade antes de realizar a refatoração;

f) Pós-condição: uma lista de asserções que tem que ser verdade após a realização da refatoração;

g) Mecanismo: um mecanismo de transformação descrevendo todos os passos da refatoração;

h) Algoritmo: um algoritmo que descreve a refatoração.

2. Especificação Formal:

a) Pré-condição: a pré-condição expressa em OCL;

b) Algoritmo: a refatoração (*model transformation*) definida em ATL;

c) Pós-condição: a pós-condição expressa em OCL.

4.3 Exemplo de uso da abordagem de criação das refatorações

Para facilitar o entendimento de como as refatorações são criadas para o metamodelo KDM, nesta seção são apresentadas algumas refatorações que foram criadas para o KDM, seguindo os passos descritos anteriormente. As refatorações que foram criadas para o metamodelo KDM podem ser visualizadas na Tabela 5. Porém, neste capítulo, apenas algumas refatorações são detalhadas. As refatorações apresentadas na Tabela 5 foram implementadas em um ambiente computacional denominado KDM-RE, o qual é apresentado no Capítulo 6. Nas próximas seções, as refatorações *Push Down Attribute* e *Extract Class* são apresentadas e detalhadas.

4.3.1 Criar Refatoração Push Down Attribute

Nesta seção, a refatoração *Push Down Attribute* é criada seguindo os passos descritos neste capítulo. Essa refatoração é utilizada para remover a generalização de atributos, ou seja, um atributo é apenas utilizado em algumas subclasses, assim, não é interessante manter o atributo na superclasse, uma vez que cada subclasse pode definir um comportamento para o atributo (FOWLER, 1999).

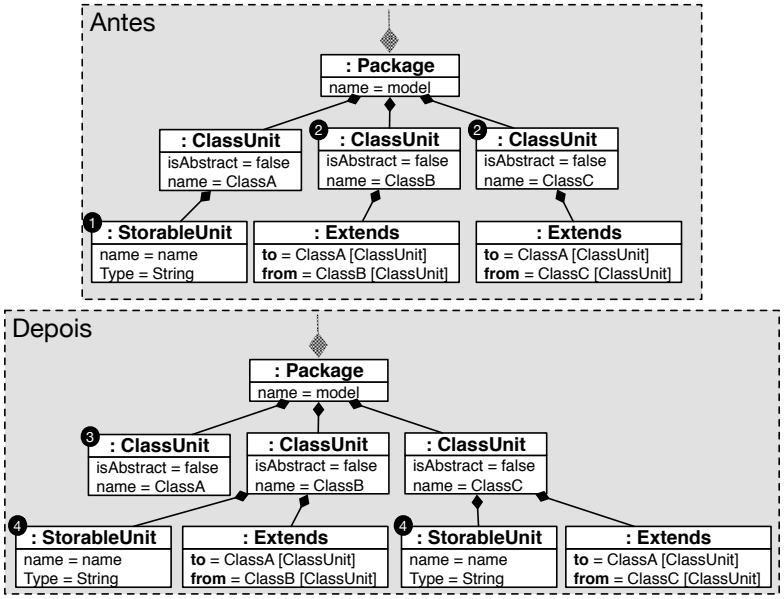
4.3.1.1 Identificar Elementos Estruturais

O primeiro passo da abordagem consiste na identificação do elemento estrutural a ser refatorado. Pelo nome da refatoração é possível identificar que o elemento estrutural a ser refatorado é um atributo (do inglês - *attribute*). Dessa forma, utilizando o artefato de mapeamento entre KDM e POO (ver Tabela 3) é possível identificar que um atributo em KDM é representado pela metaclasse `StorableUnit`.

4.3.1.2 Identificar Operações

No segundo passo da abordagem, o engenheiro de modernização deve identificar as operações atômicas que compõem a refatoração a ser criada. Na Figura 27, duas instâncias simplificadas do KDM são ilustradas, a parte superior ilustra a instância antes da realização da refatoração *Push Down Attribute* e a parte inferior representa o resultado da refatoração. Como observado, o primeiro passo da refatoração *Push Down Attribute* é selecionar um específico `StorableUnit` (ver Figura 27 ❶). Em seguida, deve-se selecionar as subclasses que realmente utilizam esse `StorableUnit` para move-lo (ver Figura 27 ❷). Posteriormente, o {`StorableUnit`Selecionado} é movido para as sub-`ClassUnit`Selecionadas, como ilustrado na parte inferior da Figura 27 ❹. É visto que a instância da `ClassUnit` em que havia o {`StorableUnit`Selecionado} não possui mais sua instância (ver Figura 27 ❸). Observando a Figura 27 é possível concluir que a refatoração *Push Down Attribute* pode ser criada por meio da combinação das seguintes operações atômicas: (*i*) `add` e (*ii*) `delete`.

Figura 27 – Instância simplificada do KDM antes e depois da refatoração *Push Down Attribute*.



Fonte: Elaborada pelo autor.

4.3.1.3 Implementar Operações

Após identificar as operações que compõem a refatoração a ser criada, o próximo passo consiste na implementação das operações. O terceiro passo da abordagem é apoiado por um conjunto de *templates* que auxilia o engenheiro de modernização a implementar as operações. As operações atômicas identificadas para implementar a refatoração *Push Down Attribute* foram implementadas em ATL utilizando os *templates* apresentados nos Códigos-fontes 4 e 6. A combinação desses *templates* resulta nos Códigos-fontes 20 e 21.

Código-fonte 20: ATL representando a operação atômica `add`.

```
1 rule createStorableUnit {
2   from
3     source : MM!ClassUnit (source.name = "{sub-ClassUnitSelecionada}"
4   to
5     target : MM!ClassUnit (
6       codeElement←source.codeElement→including(newElement)
7     ),
8     newElement: MM!StorableUnit (
9       name←{StorableUnitSelecionado.name}
10    )
11 }
```

Código-fonte 21: ATL representando a operação atômica `delete`.

```
1 rule deleteStorableUnit {
2   from
3     source : MM!StorableUnit (source.name = {StorableUnitSelecionado.name} and
      source.refImmediateComposite().name = {ClassUnit.name} )
4   to
5     drop
6 }
```

4.3.1.4 Agrupar Operações

Após implementar as operações atômicas seguindo os *templates* apresentados neste capítulo, o próximo passo consiste em agrupar tais operações. Dessa forma, um *template* para guiar o agrupamento das operações atômicas também foi definido neste capítulo (ver Código-fonte 10). O agrupamento das operações atômicas apresentadas nos Códigos-fontes 20 e 21 resulta na refatoração apresentada no Código-fonte 22.

Código-fonte 22: ATL da refatoração *Push Down Attribute*.

```
1 module pushDownStorableUnit;
```

```
2 create OUT : MM refining IN : MM;
3 rule createStorableUnit {
4   from
5     source : MM!ClassUnit (source.name = "{sub-ClassUnitSelecionada}"
6   to
7     target : MM!ClassUnit (
8       codeElement←source.codeElement→including(newElement)
9     ),
10    newElement: MM!StorableUnit (
11      name←{StorableUnitSelecionado.name}
12    )
13 }
14 rule deleteStorableUnit {
15   from
16    source : MM!StorableUnit (source.name = {StorableUnitSelecionado.name} and
       source.refImmediateComposite().name = {ClassUnit.name} )
17   to
18    drop
19 }
```

4.3.1.5 Definir Restrições

Após identificar as operações que compõem a refatoração e implementar a refatoração utilizando os *templates* definidos na abordagem, o próximo passo consiste na definição de asserções (pré- e pós-condições) da refatoração. Para cada operação atômica identificada no segundo passo da abordagem, *templates* são disponibilizados para auxiliar o engenheiro de modernização a criar tais asserções. No Código-fonte 23, é apresentada a pré-condição criada para a refatoração *Push Down Attribute*, a qual foi criada usando uma combinação dos *templates* apresentados nos Códigos-fontes 11 e 14.

Código-fonte 23: Pré-condição da refatoração *Push Down Attribute*.

```
1 context StorableUnit::preCond(newName: String)
2 pre : ClassUnit.allInstances->select(e : ClassUnit | a.name = "{sub-
    ClassUnitSelecionada}" and not StorableUnit.refImmediateComposite().
    codeElement→exist (e : StorableUnit | e.name = newName)
3 and ClassUnit.allInstances->select(e : ClassUnit | a.name =
    {ClassUnit.name}) and StorableUnit.refImmediateComposite().codeElement
    →exist (e : StorableUnit | e.name = newName)
```

Similarmente, no Código-fonte 24, é apresentada a pós-condição criada para a refatoração *Push Down Attribute*.

Código-fonte 24: Pós-condição da refatoração *Push Down Attribute*.

```
1 context StorableUnit::preCond(newName: String)
2 post : ClassUnit.allInstances->select(e : ClassUnit | a.name = "{sub-
    ClassUnitSelecionada}" and StorableUnit.refImmediateComposite().
    codeElement→exist (e : StorableUnit | e.name = newName)
3 and ClassUnit.allInstances->select(e : ClassUnit | a.name =
    {ClassUnit.name}) and not StorableUnit.refImmediateComposite().
    codeElement→exist (e : StorableUnit | e.name = newName)
```

4.3.1.6 Documentar Refatoração

Para os exemplos apresentados neste capítulo, escolheu-se especificar as refatorações criadas seguindo o *template* apresentado na Seção 4.2.6.

1. Especificação Informal:
   a) Nome: *Push Down Attribute*;
   b) Definição:
      - `StorableUnit`Selecionado - um atributo que será movido para subclasses;
      - `ClassUnit` - uma classe na qual o {`StorableUnit`Selecionado} é definido;
      - sub-`ClassUnit`Selecionadas - subclasses de {`ClassUnit`}.
   c) Objetivo: Mover um {`StorableUnit`Selecionado} para as {sub-`ClassUnit`Selecionadas}.
   d) Descrição (opcional): {`StorableUnit`Selecionado} é utilizada em apenas algumas subclasses.
   e) Pré-condição:
      - {`StorableUnit`Selecionado} não existe nas sub-`ClassUnit`Selecionadas;
      - {`StorableUnit`Selecionado} existe na `ClassUnit`.
   f) Pós-condição:
      - {`StorableUnit`Selecionado} existe nas sub-`ClassUnit`Selecionadas;
      - {`StorableUnit`Selecionado} não existe na `ClassUnit`.
   g) Mecanismo: move um atributo de uma classe para todas suas subclasses;
   h) Algoritmo:
      - para cada sub-`ClassUnit`Selecionadas que realmente usa o {`StorableUnit`Selecionado} - sub-`ClassUnit`Selecionadas.add({`StorableUnit`Selecionado})
      - {`ClassUnit`}.delete({`StorableUnit`Selecionado}).
2. Especificação Formal:
   a) Pré-condição: a pré-condição da refatoração *Push Down Attribute* é apresentada no Código-fonte 23;

b) Algoritmo: a ATL responsável por realizar a refatoração *Push Down Attribute* é apresentada no Código-fonte 22

c) Pós-condição: a pós-condição da refatoração *Push Down Attribute* é apresentada no Código-fonte 24;

4.3.2 Criar Refatoração Extract Class

A refatoração *Extract Class* deve ser utilizada quando uma determinada classe está fazendo o trabalho que deveria ser realizado por duas classes (FOWLER, 1999). Dessa forma, nesta seção, a refatoração *Extract Class* é criada seguindo todos os passos da abordagem.

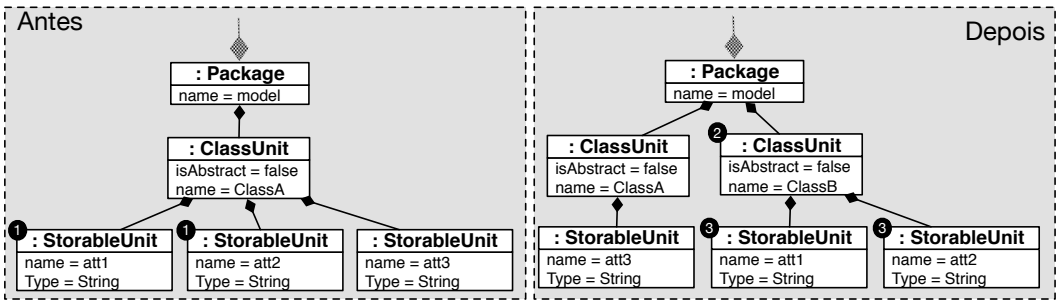
4.3.2.1 Identificar Elementos Estruturais

A refatoração *Extract Class* basicamente consiste na criação de uma nova classe, e em sequência deve-se mover todos os atributos relevantes de uma classe para essa nova classe. Dessa forma, as construções da linguagem OO utilizadas na refatoração são: (*i*) classes e (*ii*) atributos. Utilizando o artefato apresentado na Tabela 3, é possível identificar que as metaclasses em KDM que representam essas construções em OO são `ClassUnit` e `StorableUnit`, respectivamente.

4.3.2.2 Identificar Operações

No segundo passo da abordagem deve-se identificar as operações atômicas que compõem a refatoração a ser criada. Na Figura 28, são apresentadas duas instâncias simplificada do KDM, uma representa a instância antes da refatoração e a outra representa a instância após a aplicação da refatoração.

Figura 28 – Instância simplificada do KDM antes e depois da refatoração *Extract Class*.



Fonte: Elaborada pelo autor.

O primeiro passo da refatoração *Extract ClassUnit* é selecionar um conjunto de `StorableUnits` que será adicionado em uma nova instância de metaclasse `ClassUnit`, no exemplo ilustrado na Figura 28 ❶, duas instâncias de `StorableUnits` são selecionadas, `att1` e `att2`. Em seguida, uma instância da metaclasse `ClassUnit` é criada e adicionada ao mesmo `Package` que a instância da metaclasse `ClassUnit` e que contém os {`StorableUnit`Selecionados}, como ilustrado

na Figura 28 ❸. Em seguida, todos {StorableUnitSelecionados} são adicionados nessa nova instância, como representado na Figura 28 ❷. Além disso, os {StorableUnitSelecionados} devem ser deletados da {ClassUnitSelecionada}. Observando a Figura 28 é possível identificar que é a refatoração *Extract Class* pode ser criada por meio da combinação das seguintes operações atômicas: (*i*) `add` e (*ii*) `delete`.

4.3.2.3 Implementar Operações

No passo anterior identificou-se que as operações atômicas `add` e `delete` quando combinadas podem compor e criar a refatoração *Extract Class*. Dessa forma, essas operações atômicas foram implementadas em ATL utilizando os *templates* apresentados nos Códigos-fontes 4 e 6. A combinação desses *templates* resultou nos Códigos-fontes 25, 26 e 27.

Código-fonte 25: ATL representando a operação atômica `add ClassUnit` da refatoração *Extract ClassUnit*.

```
1 rule createClassUnit {
2   from
3     source : MM!Package (source.name = "{Package}")
4   to
5     target: MM!Package (
6       codeElement←source.codeElement→including(newElement)
7     ),
8     newElement: MM!ClassUnit (
9       name←"{newName}"
10    )
11 }
```

Código-fonte 26: ATL representando a operação atômica `add StorableUnit` da refatoração *Extract ClassUnit*.

```
1 rule createStorableUnit {
2   from
3     source : MM!ClassUnit (source.name = "{newName}")
4   to
5     target: MM!ClassUnit (
6       codeElement←source.codeElement→including(newElement)
7     ),
8     newElement: MM!StorableUnit (
9       name←"storableUnitName"
10    )
11 }
```

Código-fonte 27: ATL representando a operação atômica `delete StorableUnit` da refatoração *Extract ClassUnit*.

```
1 rule deleteStorableUnit {
2   from
3     source : MM!StorableUnit (source.name = "{StorableUnit}Selecionado" and
      source.refImmediateComposite().name = "{ClassUnit}Selecionado")
4   to
5     drop
6 }
```

4.3.2.4 Agrupar Operações

Após implementar as operações atômicas seguindo os *templates* apresentados neste capítulo, o próximo passo consiste em agrupar tais operações. Dessa forma, um *template* para guiar o agrupamento das operações atômicas também foi definido neste capítulo (ver Código-fonte 10). O agrupamento das operações atômicas apresentadas nos Códigos-fontes 25, 26 e 27 resulta na refatoração apresentada no Código-fonte 28.

Código-fonte 28: ATL da refatoração *Extract ClassUnit*.

```
1 module extractClassUnit;
2 create OUT : MM refining IN : MM;
3 rule createClassUnit {
4   from
5     source : MM!Package (source.name = "{Package}")
6   to
7     target: MM!Package (
8       codeElement←source.codeElement→including(newElement)
9     ),
10    newElement: MM!ClassUnit (
11      name←"{newName}"
12    )
13 }
14 rule createStorableUnit {
15   from
16     source : MM!ClassUnit (source.name = "{newName}")
17   to
18     target: MM!ClassUnit (
19       codeElement←source.codeElement→including(newElement)
20     ),
21     newElement: MM!StorableUnit (
22       name←"storableUnitName"
23     )
24 }
25 rule deleteStorableUnit {
```

```
26    from
27      source : MM!StorableUnit (source.name = "{StorableUnit}Selecionado" and
        source.refImmediateComposite().name = "{ClassUnit}Selecionado")
28    to
29      drop
30  }}
```

4.3.2.5 Definir Restrições

Após identificar as operações e implementar a refatoração, o próximo passo consiste na definição das pré- e pós-condições da refatoração. Os *templates* apresentados neste capítulo foram utilizados para criar as asserções da refatoração *Extract Class*. O Código-fonte 29 apresenta a pré-condição criada para a refatoração. Essa pré-condição foi criada utilizando os *templates* apresentados nos Códigos-fontes 11 e 14. Similarmente, o Código-fonte 30 apresenta a pós-condição criada para a refatoração *Extract ClassUnit*.

Código-fonte 29: Pré-condição da refatoração *Extract Class*.

```
1 context StorableUnit::preCond(newName: String)
2 pre : ClassUnit.allInstances->select(e : ClassUnit | a.name =
      {ClassUnitSelecionada}) and StorableUnit.refImmediateComposite().
      codeElement→exist (e : StorableUnit | e.name = {StorableUnitSelecionado})
3 and Package.allInstances->select(e : Package | a.name = "{Package}" and not
       ClassUnit.refImmediateComposite().codeElement→exist (e : ClassUnit
      | e.name = newName)
4 and ClassUnit.allInstances->select(e : ClassUnit | a.name = newName and
      not StorableUnit.refImmediateComposite().codeElement→exist (e :
      StorableUnit | e.name = {StorableUnitSelecionado}.name)
```

Código-fonte 30: Pós-condição da refatoração *Extract ClassUnit*.

```
1 context StorableUnit::preCond(newName: String)
2 post : ClassUnit.allInstances->select(e : ClassUnit | a.name =
      {ClassUnitSelecionada}) and not StorableUnit.refImmediateComposite().
      codeElement→exist (e : StorableUnit | e.name = {StorableUnitSelecionado}.
      name)
3 and Package.allInstances->select(e : Package | a.name = "{Package}" and
      ClassUnit.refImmediateComposite().codeElement→exist (e : ClassUnit |
       e.name = newName)
4 and ClassUnit.allInstances->select(e : ClassUnit | a.name = newName and
      StorableUnit.refImmediateComposite().codeElement→exist (e :
      StorableUnit | e.name = {StorableUnitSelecionado}.name)
```

4.3.2.6 Documentar Refatoração

O último passo da abordagem é a especificação da refatoração criada. Esse passo é opcional e fica a critério do engenheiro de modernização realizar esse passo. Para a refatoração *Extract ClassUnit* escolheu-se especificar a refatoração seguindo o *template* apresentado na Seção 4.2.6.

1. Especificação Informal:
   a) Nome: *Extract ClassUnit*;
   b) Definição:
      - `Package` - uma instância da metaclasse `Package` para adicionar a nova `ClassUnit`;
      - `ClassUnit`Selecionada - a classe que contém os atributos e métodos que devem ser movido para a nova classe;
      - novoNome - um novo nome para a nova classe a ser criada;
      - `StorableUnit`Selecionado - atributo selecionado para ser movido para a nova classe.
   c) Objetivo: Criar uma nova `ClassUnit` e mover o `StorableUnit`Selecionado para essa nova instância.
   d) Descrição (opcional): `ClassUnit`Selecionada está realizando o trabalho que deveria ser realizado por duas classes.
   e) Pré-condição:
      - {`StorableUnit`Selecionado} existe na {`ClassUnit`Selecionada};
      - nova `ClassUnit` não existe no `Package`;
      - {`StorableUnit`Selecionado} não existe na nova `ClassUnit`.
   f) Pós-condição:
      - {`StorableUnit`Selecionado} não existe na {`ClassUnit`Selecionada};
      - nova `ClassUnit` existe no `Package`;
      - {`StorableUnit`Selecionado} existe na nova `ClassUnit`.
   g) Mecanismo: Deve-se criar uma nova classe e mover os atributos selecionados;
   h) Algoritmo:
      - addNewClassUnit({novoNome});
      - adiciona essa nova instância dentro de um `Package`;
      - para cada {`StorableUnit`Selecionado} - add({newClassUnit}, {`StorableUnit`-Selecionado});
      - para cada {`StorableUnit`Selecionado} - delete({`ClassUnit`Selecionada}, {`Storable`-`Unit`Selecionado});

2. Especificação Formal:

   a) Pré-condição: a pré-condição da refatoração *Extract Class* é apresentada no Código-fonte 29;

   b) Algoritmo: a ATL responsável por realizar a refatoração *Extract Class* é apresentada no Código-fonte 28;

   c) Pós-condição: a pós-condição da refatoração *Extract Class* é apresentada no Código-fonte 30;

4.4 Trabalhos Relacionados

Nesta seção, são descritos os principais trabalhos encontrados na literatura que remetem às abordagens para a criação de refatorações em nível de modelos. De acordo com o mapeamento sistemático (DURELLI *et al.*, 2014b), poucos trabalhos propuseram abordagens para criar refatorações em nível de modelo, entre eles, alguns não mostraram dados suficientes que pudessem ser utilizados nesta tese. Entretanto, foram identificados alguns trabalhos que deram suporte e guiaram o desenvolvimento da abordagem descrita neste capítulo, assim, nesta seção, são mostradas as principais semelhanças e diferenças encontradas entre eles.

Sen *et al.* (2012) definem uma abordagem para especificar refatorações de forma genérica. Os autores apresentam um metametamodelo denominado *GenericMT*, o qual contém elementos estruturais do POO (classes, métodos, atributos, parâmetros, etc.). Por meio desse metametamodelo, refatorações genéricas podem ser criadas. Para ativar as refatorações de forma genérica para um metamodelo específico, uma adaptação é necessária. Após a criação dessa adaptação, todas as refatorações criadas podem ser aplicadas em instâncias do metamodelo adaptado. A aplicação das refatorações é realizada utilizando os conceitos de entrelaçamento do paradigma orientado a aspectos. Uma limitação dessa abordagem é a estrutura do *GenericMT*, pois apenas elementos estruturais do POO podem ser refatorados. Refatorações que necessitam de outros elementos estruturais para realizar a sua operação não podem ser criadas por meio dessa abordagem. Por outro lado, a abordagem apresentada neste capítulo permite que o engenheiro de modernização crie refatorações para diversos elementos estruturais, uma vez que o metamodelo KDM contém metaclasses para representar diversos artefatos de um sistema. Além disso, a abordagem definida por esses autores não utiliza metamodelos padronizados, tais como: UML e KDM. Os autores não deixam claro se as refatorações são apoiadas por diretrizes/passos. A abordagem apresentada aqui define um conjunto de seis passos para auxiliar o engenheiro de modernização a criar refatorações para o KDM, assim, *templates* foram definidos. Da mesma forma que a abordagem destacada neste capítulo, Sen *et al.* (2012) também utilizam restrições (pré- e pós-condições) nas refatorações criadas.

Uma abordagem muito similar é proposta por Tichelaar *et al.* (2000). Os autores introduzem uma abordagem que utiliza o metamodelo FAMIX, permitindo a criação de refatorações

para elementos estruturais do POO (classes, objetos, métodos, etc.) e todas as refatorações criadas utilizam a ferramenta MOOSE (DUCASSE *et al.*, 2005). De acordo com os autores, tal abordagem é independente de linguagem, visto que MOOSE aceita como entrada as linguagens Java, C++ e Smalltalk. Porém, os autores não utilizam um metamodelo padronizado pelo OMG, FAMIX é um metamodelo proprietário, o que pode dificultar a interoperabilidade entre outros apoios ferramentais. Da mesma forma que a abordagem descrita aqui, os autores também utilizam operações atômicas (`add`, `delete` e `change`) para compor as refatorações criadas para o metamodelo FAMIX. Restrições também são suportadas na abordagem proposta por Tichelaar *et al.* (2000).

Seguindo a mesma linha de pensamento, outra abordagem foi proposta por Zhang, Lin e Gray (2005), a qual salienta que refatorações genéricas também são criadas por meio do *Generic Modeling Environment* (GME) (LEDECZI *et al.*, 2001). Essa abordagem utiliza instâncias do metamodelo UML e todas as refatorações genéricas são criadas para serem aplicadas em UML. Os autores utilizam o metamodelo UML, o qual é também padronizado pelo OMG. Similarmente a abordagem descrita neste capítulo, os autores também utilizam operações atômicas (`add`, `delete` e `change`) para compor as refatorações criadas para o metamodelo UML. Restrições também são definidas em OCL na abordagem proposta por Zhang, Lin e Gray (2005).

Brosch *et al.* (2009) definem uma abordagem que permite a especificação de refatorações em instâncias de qualquer metamodelo. Modificações realizadas nos metamodelos são armazenadas, abstraídas e, então, propagadas para manter o metamodelo sincronizado e consistente. As principais semelhanças com a abordagem descrita neste capítulo são: (*i*) operações atômicas são utilizadas para criar a base para as refatorações e (*ii*) tais operações são agrupadas para criar a refatoração. As principais diferenças são: (*i*) os autores não definem passos para a criação das refatorações, (*ii*) não utilizam um metamodelo padronizado pelo OMG como base, porém, os autores afirmam que a abordagem pode ser aplicada a qualquer metamodelo, (*iii*) *templates* não são utilizados para auxiliar a criação das refatorações e (*iv*) não é evidente se linguagens de restrições são utilizadas nas refatorações.

Sun, White e Gray (2009) apresentam uma abordagem similar, porém, diferentemente de Brosch *et al.* (2009), modificações realizadas são armazenadas em uma instância concreta de um metamodelo, e, em seguida, a abordagem identifica padrões de transformações, os quais podem ser replicados em outras instâncias do metamodelo para manter sua consistência. As principais diferenças com a abordagem descrita neste capítulo são: (*i*) os autores não especificam passos para conduzir o engenheiro de modernização durante a criação de refatorações, (*ii*) da mesma forma que Brosch *et al.* (2009), Sun, White e Gray (2009) não utilizam um metamodelo padronizado pelo OMG como base, porém, os autores afirmam que a abordagem pode ser aplicada a qualquer metamodelo, (*iii*) *templates* também não são explicitamente citados na abordagem e (*iv*) não é evidente se linguagens de restrições são utilizadas na abordagem. As principais semelhanças identificadas são: (*i*) operações atômicas são utilizadas para criar as

refatorações e (*ii*) tais operações são agrupadas para criar a refatoração.

Langer, Wimmer e Kappel (2010) também apresentam uma abordagem para realizar transformações em modelos, a qual é denominada REMA e busca a derivação incremental de regras de transformação. Essas regras podem ser usadas para aplicar refatorações automaticamente em qualquer modelo. As principais semelhanças identificadas com a abordagem descrita neste capítulo são: (*i*) operações atômicas são utilizadas, (*ii*) operações compostas são criadas por meio da combinação de um conjunto de operações atômicas e, assim, as refatorações são criadas e (*iii*) restrições também são utilizadas na abordagem proposta por Langer, Wimmer e Kappel (2010). As diferenças explicitamente identificadas com a abordagem apresentada neste capítulo são: (*i*) nenhuma diretriz é apresentada para guiar a criação das refatorações, (*ii*) os autores não utilizam metamodelo padronizados e (*iii*) *templates* também não são citados na abordagem.

Similar à abordagem apresentada neste capítulo, Astels (2002) define uma abordagem para aplicar refatorações em diagrama de classe da UML e suas refatorações são criadas por meio da combinação de operações atômicas (`add`, `delete`, etc.). Além disso, o autor também utiliza linguagem de transformação e OCL para definir as refatorações e suas restrições, respectivamente. Porém, nenhuma diretriz é apresentada na abordagem. Além disso, *templates* também não são utilizados na abordagem.

Reimann (2015) define uma abordagem para criar e aplicar refatorações em nível de modelo de forma genérica. De acordo com o autor, sua abordagem pode ser aplicada em qualquer metametamodelo, metamodelo e modelo. Para isso, ele definiu um conjunto de passos e criou uma DSL, em que o usuário pode especificar a refatoração de forma independente e genérica. As principais semelhanças identificadas com a abordagem descrita neste capítulo são: (*i*) os autores apresentam um conjunto de diretrizes para auxiliar o engenheiro a criar as refatorações, (*ii*) operações atômicas também são utilizadas, (*iii*) a composição dessas operações resulta na refatoração e (*iv*) restrições são implementadas utilizando OCL. As diferenças são: (*i*) da mesma forma que Brosch *et al.* (2009) e Sun, White e Gray (2009), Reimann (2015) também não utiliza um metamodelo padronizado pelo OMG como base, porém, o autor afirma que a abordagem pode ser aplicada a qualquer metamodelo e (*ii*) o autor não menciona a utilização de *templates*.

Misbhauddin (2012) descreve uma abordagem para aplicar refatorações em diversos diagramas da UML. Quando uma refatoração é aplicada, por exemplo, em um diagrama de classe, o efeito da refatoração é replicado em outros diagramas, tais como: diagrama de sequência, diagrama de caso de uso, etc. As principais semelhanças identificadas com a abordagem descrita neste capítulo são: (*i*) o autor apresenta um conjunto de passos para auxiliar a criação de refatorações para um metamodelo padronização do OMG, nesse contexto, UML, (*ii*) operações atômicas são utilizadas para criar as refatorações e (*iii*) o autor utiliza linguagem de restrição escrita em OCL. As principais diferenças entre a abordagem proposta neste capítulo e a abordagem proposta por Misbhauddin (2012) são: (*i*) o autor não evidencia a utilização de *templates* para criar as refatorações e (*ii*) também não é evidente se as operações atômicas são agrupadas para

criar refatorações mais complexas.

Na Tabela 12 é apresentada uma comparação entre a abordagem apresentada neste capítulo e os trabalhos relacionados quanto à forma como criam refatorações para modelos. Dessa forma, alguns critérios foram definidos durante a comparação, tais critérios são: (*i*) a abordagem fornece diretrizes/passos; (*ii*) a abordagem emprega o metamodelo KDM; (*iii*) a abordagem disponibiliza *templates*; (*iv*) a abordagem considera operações atômicas; (*v*) considera a composição de refatorações; e (*vi*) a abordagem considera restrições. As colunas da tabela foram abreviadas, assim, "ATR" representa "Autor(es)", "DRZ" representa "Diretrizes", "TPT" representa "*Templates*", "OA" representa "Operações Atômicas", "CR" representa "Composição de Refatoração" e "RTÇ" representa "Restrições". O símbolo "✓" representa que a abordagem define o critério e o símbolo "✗" representa que a abordagem não define o critério.

Tabela 12 – Comparação entre a abordagem definida neste capítulo e os trabalhos relacionados.

ATR	DRZ	KDM	TPT	OA	CR	RTÇ
Durelli R. S. Capítulo 4	✓	✓	✓	✓	✓	✓
Sen *et al.* (2012)	✗	✗	✗	✓	✗	✓
Tichelaar *et al.* (2000), Ducasse *et al.* (2005)	✗	✗	✗	✓	✓	✓
Zhang, Lin e Gray (2005)	✗	✗	✗	✓	✓	✓
Brosch *et al.* (2009)	✗	✗	✗	✓	✓	✗
Sun, White e Gray (2009)	✗	✗	✗	✓	✓	✗
Langer, Wimmer e Kappel (2010)	✗	✗	✗	✓	✓	✓
Astels (2002)	✗	✗	✗	✓	✓	✓
Reimann (2015)	✓	✗	✗	✓	✓	✓
Misbhauddin (2012)	✓	✗	✗	✓	✗	✓

4.5 Considerações Finais

Neste capítulo, foram apresentados seis passos para auxiliar o engenheiro de modernização a criar refatorações para o metamodelo KDM. Com o intuito de exemplificar esses seis passos, foram escolhidas algumas refatorações propostas por Fowler (1999) para serem criadas para o metamodelo KDM.

Antes de criar qualquer refatoração para o metamodelo KDM, o primeiro passo diz que é necessário identificar as metaclasses do KDM que têm características similares aos conceitos do POO, bem como instruções comumente utilizadas em todas as linguagens de programação, tais como, ramificações, iterações, etc. Dessa maneira, esse capítulo também apresentou um mapeamento entre os conceitos do POO e o metamodelo KDM. Acredita-se que engenheiros de modernização pouco familiarizados com o metamodelo KDM podem gastar menos tempo durante a criação de novas refatorações com a utilização desse mapeamento, assim, qualquer exemplo de refatoração pode ser facilmente adaptado para o metamodelo KDM seguindo tal mapeamento, bem como os passos aqui apresentados.

Após identificar todos os elementos estruturais e identificar o mapeamento entre POO e o metamodelo KDM, o próximo passo é identificar quais operações compõem a refatoração que se almeja criar para o KDM. No contexto desta tese, todas as refatorações podem ser agrupadas em nível de granularidade. As granularidades podem ser definidas em dois níveis de operações: (*i*) operação atômica e (*ii*) operações compostas. As granularidades definidas como operações atômicas podem ser especificadas por meio de operações primitivas (`addd`, `delete` e `change`). As refatorações de granularidade compostas são uma combinação de operações atômicas (ver Tabela 5). Em seguida, no próximo passo, o engenheiro de modernização deve implementar a refatoração por meio da linguagem de transformação ATL. Dessa forma, *templates* foram criados para auxiliar os engenheiros de modernização a implementar refatorações utilizando a linguagem ATL. Posteriormente, o engenheiro de modernização deve implementar as restrições (pré- e pós-condições) da refatoração. Similarmente, *templates* também foram definidos para auxiliar os engenheiros de modernização a criar as restrições para uma refatoração em nível do KDM.

Após criar uma determinada refatoração, bem como suas pré- e pós-condições para o metamodelo KDM, o engenheiro de modernização pode especificar e documentar a refatoração. Esse passo é opcional e fica a critério do engenheiro de modernização realizá-lo. Caso o engenheiro almeje documentar a refatoração duas especificações podem ser utilizadas: (*i*) especificação informal e (*ii*) especificação formal. Na primeira, a ideologia básica é permitir que o engenheiro de modernização expresse o propósito da refatoração por meio de linguagem natural. Na segunda, o engenheiro de modernização representa as pré- e pós-condições, bem como a transformação/refatoração propriamente dita em linguagens executáveis - OCL e ATL foram utilizadas, respectivamente. Ambas as especificações são úteis para o engenheiro de modernização, pois a especificação informal é utilizada para facilitar a compreensão e o propósito da refatoração e a especificação formal facilita a visualização da implementação da refatoração e suas asserções. Além disso, a especificação formal é de extrema importância para facilitar a automação das refatorações.

Seguindo os passos apresentados neste capítulo, o engenheiro de modernização pode criar refatorações para o metamodelo KDM. No entanto, as especificações formais e informais resultantes não são suficientes para promover o reúso de refatorações no contexto do metamodelo KDM. Diante disso, no Capítulo 5 é apresentado um metamodelo para auxiliar o engenheiro de modernização a promover o reúso de refatorações no contexto do metamodelo KDM. Com a utilização desse metamodelo, informações (metadados) sobre refatorações podem ser reutilizadas de forma independente de linguagem e plataforma. No Capítulo 6, é apresentado um apoio computacional denominado KDM-RE. Esse apoio computacional é composto por três *plug-ins* do Eclipse: (*i*) o primeiro compreende um conjunto de *Wizards* que apoia o engenheiro de software na aplicação das refatorações em diagramas de classe UML; (*ii*) o segundo consiste em um apoio à importação e ao reúso de refatorações disponíveis no repositório; (*iii*) o terceiro constitui um módulo de propagação de mudanças que permite manter modelos internos sincronizados do KDM;

CAPÍTULO

5

SRM: UM METAMODELO PARA PROMOVER O REÚSO DAS REFATORAÇÕES PARA O KDM

5.1 Considerações Iniciais

Como ressaltado no Capítulo 2, a ADM fornece um conjunto de metamodelos para padronizar processos de modernização de software. Embora seja a intenção do OMG, até o presente momento, a ADM não publicou um metamodelo padrão para especificação de refatorações, que é uma atividade fundamental em modernizações. A ideia do OMG é que ferramentas de modernização sejam projetadas e construídas de forma a reconhecer internamente um metamodelo desse tipo. Como resultado, refatorações que são instâncias desse metamodelo, poderão ser reusadas entre essas ferramentas. Por exemplo, considerando que algum engenheiro de modernização especificou a refatoração *Extract Class* com o metamodelo supracitado, a instância desse metamodelo poderia ser importada por ferramentas de modernização para executar essa refatoração em seus projetos. Assim, o engenheiro de modernização poderia utilizar os metadados contidos nessa instância e aplicar a refatoração em qualquer sistema representado pelo KDM. Além disso, o engenheiro de modernização poderia instanciar o metamodelo de refatoração, criar um catálogo de refatorações e disponibilizá-lo para que outros possam reutilizar tais refatorações, tornando possível compartilhar e reutilizar refatorações no contexto da ADM.

Com o intuito de suprir tal limitação, no presente capítulo, é apresentado um metamodelo para a especificação de refatorações para o metamodelo KDM. Observando a Figura 1 ❷, nota-se que o metamodelo apresentado neste capítulo corresponde a uma terminologia comum para a especificação de refatorações. Quando ferramentas de modernização adotam um metamodelo como este, refatorações podem ser reutilizadas de forma independente de linguagem e plataforma. Além do metamodelo criado, esse capítulo também apresenta a metodologia empregada para a

sua criação.

O metamodelo aqui proposto tem as seguintes características: (*i*) permite a interoperabilidade entre ferramentas de modernização, desde que elas adotem o metamodelo supracitado e (*ii*) viabiliza a especificação de refatorações em um formato padronizado por parte dos engenheiros de modernização. Nota-se que o metamodelo aqui proposto segue a mesma proposta de outros metamodelos definidos na ADM e está totalmente integrado com o metamodelo KDM. Em outras palavras, uma instância do metamodelo de refatoração contém metadados que representam uma refatoração escrita para ser executada em uma instância do metamodelo KDM.

As demais seções deste capítulo estão organizadas da seguinte forma: na Seção 5.2, são destacadas as principais motivações para criar um metamodelo para prover o reúso de refatorações; na Seção 5.3, o Metamodelo de Refatorações Estruturadas é apresentado, destacando os principais passos para a engenharia do metamodelo SRM; na Seção 5.3.3, é destacado como o metamodelo SRM foi implementado. Na Seção 5.4 é apresentada a gramática de uma DSL criada com o propósito de auxiliar a instanciação do metamodelo SRM; na Seção 5.5, alguns trabalhos relacionados ao metamodelo proposto são apresentados; e na Seção 5.6, são feitas as considerações finais a respeito do metamodelo de refatoração proposto neste capítulo.

5.2 Motivação para a criação de um metamodelo de refatoração

Durante o mapeamento sistemático conduzido (ver Capítulo 3) (DURELLI *et al.*, 2014b), observou-se na literatura a carência de estudos que apresentam soluções para especificar e promover o reúso de refatorações no contexto da ADM e do metamodelo KDM. Sem a adequada representação de refatorações para o KDM, a especificação de uma refatoração pode se tornar uma atividade propensa a erros e difícil de ser reutilizada. Como visto no capítulo da fundamentação teórica, existem diversos cenários de modernização, alguns mais simples e outros mais complexos. A grande maioria desses cenários geralmente necessita e é precedida por refatorações de granularidade fina. Dessa forma, pode-se dizer que refatorações são atividades menores dentro de processos de modernização. Um metamodelo separado para refatorações flexibiliza a especificação de refatorações e permite que elas sejam reusadas dentro de transformações maiores.

O cenário em que se vislumbra o uso do metamodelo aqui proposto é o seguinte: *um engenheiro de modernização por meio de uma ferramenta de modernização que adota o metamodelo aqui proposto especifica uma determinada refatoração e, em seguida, compartilha a instância dessa refatoração em um repositório. Dessa forma, outros engenheiros que também utilizam ferramentas de modernização e o metamodelo supracitado podem, então, navegar nesse repositório e identificar, não apenas uma, mas um conjunto de refatorações. O engenheiro, então, escolhe uma determinada refatoração no repositório, faz o download e a reutiliza em sua*

ferramenta de modernização, podendo aplicar a refatoração em seu sistema representado em nível de uma instância do metamodelo KDM.

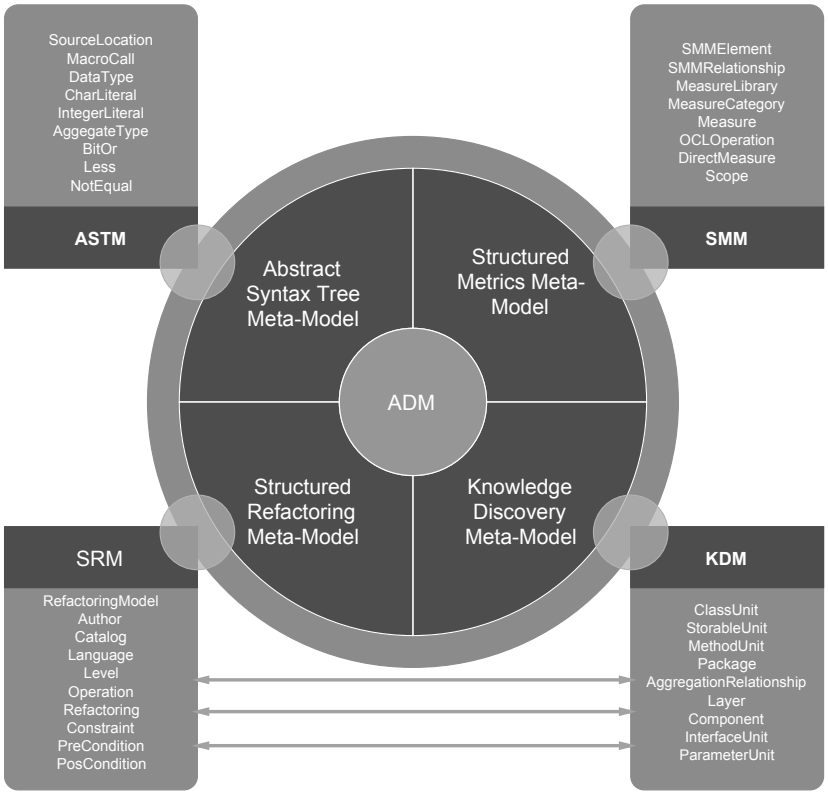
Uma das soluções para a concretização desse cenário é a criação de um metamodelo para persistir metadados referentes às refatorações. Esse metamodelo deve possuir metaclasses e metarrelacionamentos que permitam representar e especificar metadados de refatorações, por exemplo, o nome da refatoração, sua motivação, os passos para sua realização, e até mesmo o mecanismo e suas pré- e pós-condições. Devido à carência de um metamodelo que possua tais particularidades, este capítulo tem como principal objetivo definir e apresentar um metamodelo que permita a representação de metadados relacionadas com refatorações, porém, que ainda respeite e siga as características dos metamodelos definidos na abordagem ADM, por exemplo, o metamodelo aqui definido precisa ser independente de linguagem e plataforma. Assim, o metamodelo aqui apresentado utiliza as metaclasses do metamodelo KDM. Uma instância do metamodelo de refatoração deve possuir instâncias de metaclasses específicas para definir metadados sobre refatorações (autor, nome, motivação, descrição, etc) e também instâncias de metaclasses do KDM que representam os elementos estruturais (`ClassUnit`, `StorableUnit`, `MethodUnit`, etc.), onde a refatoração/transformação será aplicada. O metamodelo de refatoração faz com que a operação/mecanismo de uma refatoração torne-se independente de plataforma e linguagem de programação. Assim, o metamodelo pode ser facilmente utilizado e aplicado em ferramentas existentes que utilizam como base o KDM, aumentando a interoperabilidade de futuras ferramentas que utilizem esse metamodelo de refatoração. Levando em consideração as motivações destacadas, na Seção 5.3 é apresentado o metamodelo de refatorações estruturadas (do inglês - *Structured Refactoring Metamodel* – SRM). O SRM define uma especificação de refatorações e tem como princípio ser independente de linguagem de programação para fornecer uma plataforma comum pela qual o arquiteto, o pesquisador e o modernizador possam expressar refatorações sem se preocuparem com a plataforma ou linguagem de programação. O SRM objetiva definir uma terminologia comum para a especificação de refatorações para facilitar o reúso e a interoperabilidade entre ferramentas. Esse objetivo é apoiado por um conjunto de metaclasses que definem meta-atributos específicos para representar informações (metadados) de uma refatoração, auxiliando o compartilhamento das refatorações de forma intuitiva para os modernizadores.

5.3 Metamodelo de Refatorações Estruturadas

Nesta seção é apresentado o metamodelo denominado Metamodelo de Refatorações Estruturadas (do inglês - *Structured Refactoring Metamodel* (SRM)). Na Figura 29, o SRM é apresentado no sentido de mostrar onde ele se enquadra no contexto da ADM e sua interação com os outros metamodelos. A esfera externa é dividida em quatro facetas e cada uma possui o nome de um metamodelo. Além disso, cada faceta é acoplada por um retângulo contendo o nome de um metamodelo e um conjunto de metaclasses. Vê-se nessa figura que o SRM está

inserido no contexto da ADM para preencher a definição de um metamodelo de refatorações.

Figura 29 – Integração do SRM com outros metamodelos da ADM.

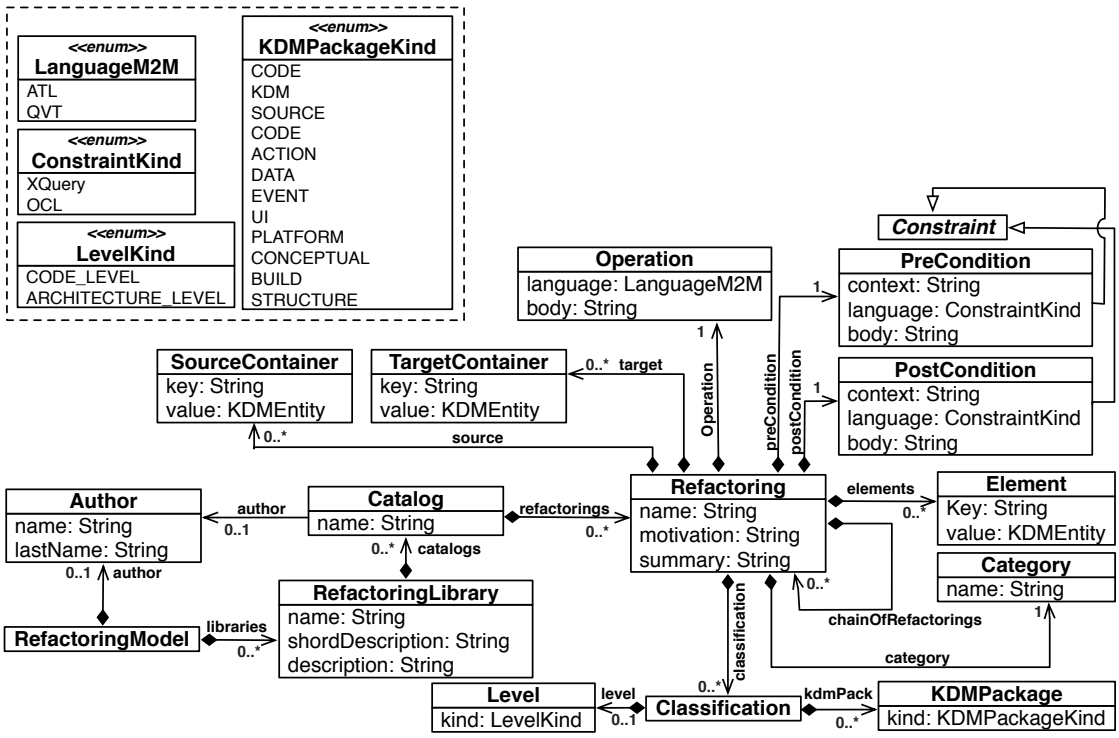


Fonte: Elaborada pelo autor.

O SRM segue as padronizações propostas pela ADM. Assim, deve-se ressaltar que o SRM é utilizado para especificar a representação de refatorações sem se preocupar com os elementos que serão refatorados (classes, métodos, atributos, etc.). O SRM assume que tais elementos devem ser representados utilizando outro metamodelo proposto pela ADM, nesse contexto, o KDM. Em outras palavras, o engenheiro de modernização deve referenciar os elementos que serão refatorados utilizando metaclasses do KDM, assim, o engenheiro deve conhecer detalhes e nomenclaturas sobre o metamodelo KDM e não apenas as metaclasses do SRM. Como visualizado na Figura 29, o SRM interage com o metamodelo KDM para utilizar instâncias de metaclasses desse metamodelo, representando os elementos que serão refatorados.

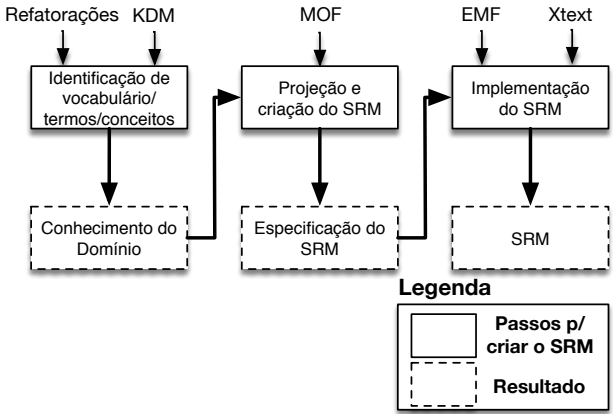
O metamodelo SRM é apresentado na Figura 30. Esse metamodelo contém três pacotes, 16 metaclasses e quatro enumerações. A Figura 31 mostra a metodologia que foi seguida para a criação do metamodelo SRM. Nesse sentido, três passos principais foram seguidos durante a metodologia empregada para a criação do metamodelo SRM: (*i*) Identificação de vocabulário/termos/conceitos, (*ii*) Projeção e criação do SRM e (*iii*) Implementação do SRM. Cada passo da metodologia utilizada para criar o SRM é apresentado com mais detalhes a seguir.

Figura 30 – Visão completa (sem os pacotes) do metamodelo SRM.



Fonte: Elaborada pelo autor.

Figura 31 – Metodologia empregada na criação do metamodelo SRM.



Fonte: Elaborada pelo autor.

5.3.1 Identificação de vocabulário/termos/conceitos

Neste passo, focou-se na identificação do vocabulário, termos e conceitos comuns que são utilizados dentro da comunidade de refatoração. Os artefatos de entrada que foram utilizados neste passo foram um conjunto de refatorações, bem como o metamodelo KDM. Durante a criação de metamodelos, é de suma importância entender o domínio que eles representam. Metamodelos definem abstrações (termos), notações e relacionamentos para representar um determinado domínio. Portanto, nesse passo, tanto os vocabulários, termos e conceitos definidos

por Opdyke (1992) e Fowler (1999) foram analisados para a identificação de abstrações para facilitar a criação do metamodelo SRM. Durante a análise, pode-se observar e identificar alguns termos comumente utilizados durante a definição de uma refatoração. Por exemplo, todas as refatorações descritas e definidas por Opdyke (1992) e Fowler (1999) seguem os seguintes termos:

- Refatoração: o nome da refatoração;
- Autor: autor da refatoração;
- Catálogo: o catálogo ao qual a refatoração pertence;
- Biblioteca de refatoração: onde um conjunto de catálogos pode ser incluído;
- Descrição: informando uma típica situação onde a refatoração deveria ser aplicada;
- Motivação: informando a motivação para a realização da refatoração;
- Operação: descrevendo os passos que devem ser realizados para executar a refatoração;
- Parâmetros: informações necessárias para executar a operação da refatoração;
- Restrições: asserções utilizadas para garantir a semântica e a sintaxe após a aplicação da refatoração:
  - Pré-condição: asserção que deve ser verdadeira antes de executar a operação da refatoração;
  - Pós-condição: asserção que deve ser verdadeira após executar a operação da refatoração.

Após a condução deste passo, uma lista dos termos utilizados no domínio de refatorações foi criada. Utilizando esses termos foi possível projetar e criar o metamodelo SRM como apresentado a seguir.

5.3.2 Projeção e criação do SRM

A partir dos termos identificados anteriormente, o SRM foi especificado. Por meio dos termos, foi possível realizar a especificação do SRM, o qual pode ser definido como uma quádrupla, como observado na Definição 2:

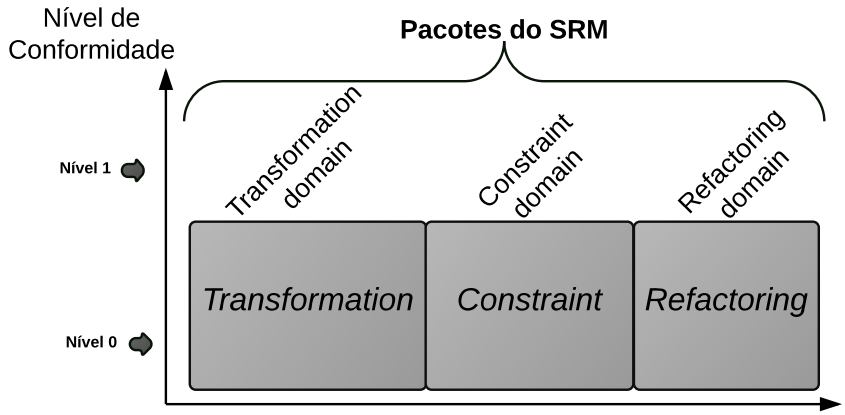
Definição 2. *O SRM é uma quádrupla* $SRM = (SRM_{mC}, SRM_{mA}, SRM_e, SRM_{mR})$*, onde* SRM_{mC} *representa um conjunto de metaclasses,* SRM_{mA} *representa um conjunto de meta-atributos,* SRM_e *representa um conjunto de enumerações e* SRM_{mR} *representa associações.*

Formalmente, pode-se definir o metamodelo como:

- Todas as metaclasses $mC \in SRM_{mC}$ têm um nome que representa o seu significado;
- Todos os meta-atributos $mA \in SRM_{mA}$ contêm um nome, um tipo e uma cardinalidade. Além disso, cada mA está associado a uma metaclasse;
- Todas as enumerações $e \in SRM_{mA}$ contêm um nome e um valor;
- Todas as meta-associações $mR \in SRM_{mR}$ são um conjunto R = E_1, E_2 , onde E_1 e E_2 são associações de R. Posteriormente, cada R contém um nome. Ambos E_1 e E_1 possuem uma cardinalidade e são associados a uma metaclasse de SRM_{mC}.

O SRM possui dois níveis de conformidade e ele é estruturado de forma modular, seguindo o princípio da separação de interesse, com a capacidade de representar diferentes partes de uma refatoração. Essa separação de interesse é alcançada por meio de pacotes, como ilustrado na Figura 32. O primeiro nível de conformidade representa que o SRM é separado em três pacotes: (*i*) *Transformation*, (*ii*) *Constraint* e (*iii*) *Refactoring*. Cada pacote constitui de um conjunto de metaclasses para descrever e representar transformações, restrições e refatorações, respectivamente. O SRM possui dois níveis de conformidade, nível 0 e nível 1, descritos a seguir:

Figura 32 – Pacotes e níveis de conformidade do metamodelo SRM.



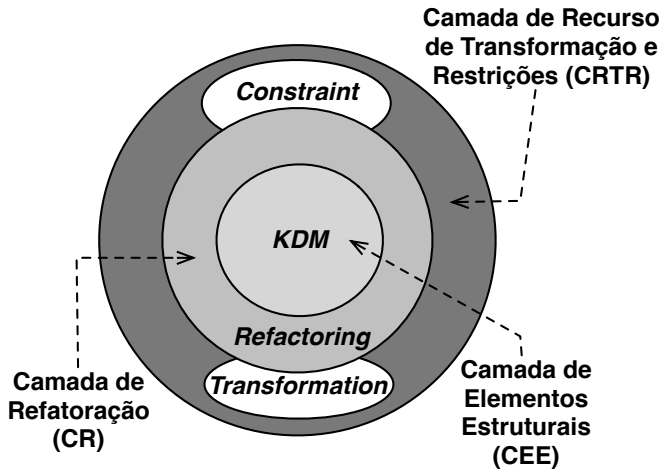
Fonte: Elaborada pelo autor.

- **Nível 0**: esse nível define os seguintes pacotes: (*i*) `Transformation`, (*ii*) `Constraint` e (*iii*) `Refactoring`. Para que uma ferramenta esteja em conformidade com o **Nível 0**, ela deve fornecer completo suporte para todas as metaclasses que foram definidas no SRM;
- **Nível 1**: esse nível é a união de todos os pacotes definidos no nível anterior. Para que uma ferramenta esteja em conformidade com o **Nível 1**, ela deve fornecer suporte para todos os pacotes dos **Nível 0**.

Todos os pacotes do metamodelo SRM apresentados na Figura 32 foram projetados e organizados esquematicamente em três camadas de abstração: (*i*) Camada de Elementos

Estruturais (CEE), (*ii*) Camada de Refatoração (CR) e (*iii*) Camada de Recurso de Transformação e Restrições (CRTR). As três camadas estão apresentadas na Figura 33.

Figura 33 – Camadas e pacotes do SRM.



Fonte: Elaborada pelo autor.

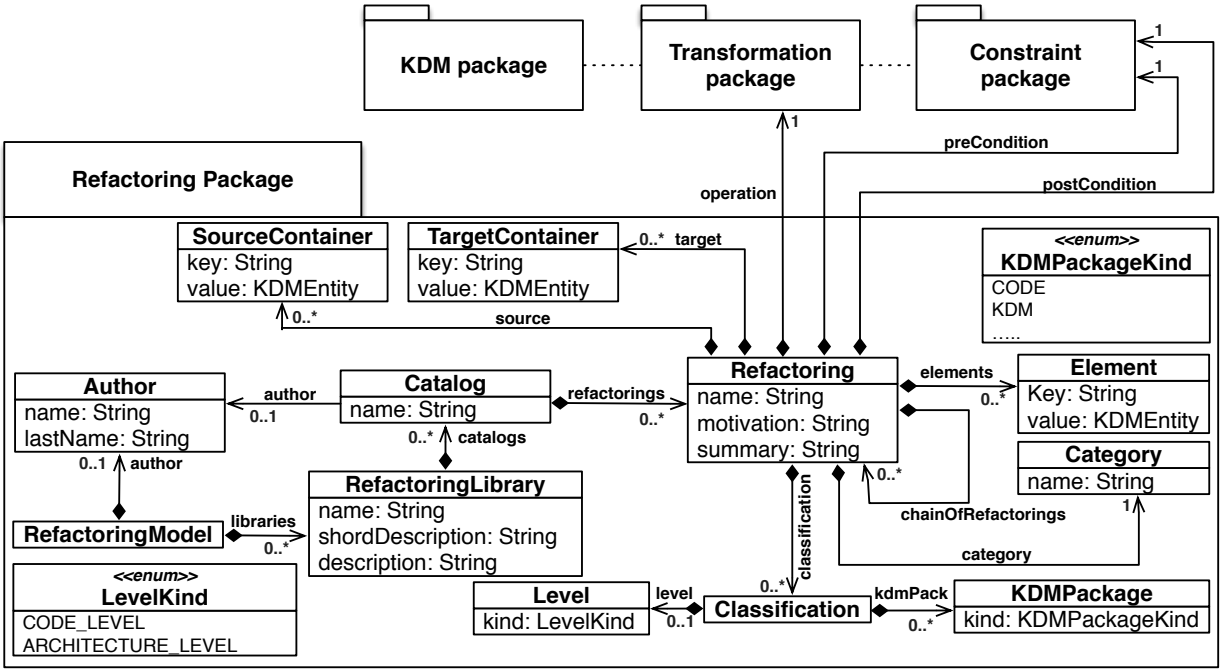
A camada CEE utiliza as metaclasses do metamodelo KDM para representar os elementos estruturais que serão refatorados. A camada CR contém o pacote `Refactoring`. Esse pacote define metaclasses para representar metadados sobre refatorações, tais como autor da refatoração, nome da refatoração, parâmetro, catálogo, etc. A camada CRTR contém dois pacotes: (*i*) `Transformation` e `Constraint`. Coletivamente, esses pacotes representam a operação/mecanismos e as restrições/asserções de uma determinada refatoração. A seguir, os pacotes do SRM, bem como suas metaclasses são apresentados.

5.3.2.1 Pacote `Refactoring`

O pacote `Refactoring` define um conjunto de metaclasses cujo propósito é representar metadados de refatoração em alto nível. O pacote inclui metaclasses para definir a qual catálogo uma determinada refatoração pertence, definir o nome da refatoração, sua motivação, autor, parâmetros, classificação e sua biblioteca de refatoração, etc. Na Figura 34, o pacote `Refactoring` e seus elementos (nove metaclasses e uma enumeração) são apresentados. Como ilustrado nessa figura, o pacote `Refactoring` utiliza outras metaclasses que são definidas em outros pacotes: (*i*) pacotes do KDM, (*ii*) pacote `Transformation` e (*iii*) pacote `Constraint`. Por exemplo, a operação de uma determinada refatoração é definida no pacote `Transformation`. Similarmente, as asserções de uma determinada refatoração são especificadas no pacote `Constraint`. Abaixo, as descrições sobre as metaclasses do pacote `Refactoring` são apresentadas.

- `RefactoringModel` representa a metaclasse raiz do metamodelo. Essa metaclasse representa uma coleção de metadados que corresponde a uma ou mais refatorações definidas para o metamodelo KDM.

Figura 34 – Pacote `Refactoring` do SR.



Fonte: Elaborada pelo autor.

- **Meta-associações**
  - `author:Author[0..1]`: representa o autor de uma refatoração;
  - `libraries:RefactoringLibrary[0..*]`: representa um conjunto de bibliotecas de refatorações que uma instância da metaclasse `RefactoringModel` possui.

- `Author` representa o autor de uma refatoração. Essa metaclasse contém dois meta-atributos:
  - **Meta-atributos**
    - `name:String`: utilizado para definir o nome do autor;
    - `lastName:String`: utilizado para definir o sobrenome do autor.

- `RefactoringLibrary` utilizado para descrever uma biblioteca de refatorações.
  - **Meta-atributos**
    - `name:String`: utilizado para descrever o nome da biblioteca de refatoração;
    - `shortDescription:String`: representa uma breve descrição sobre a biblioteca de refatoração;
    - `description:String`: representa uma completa descrição sobre a biblioteca de refatoração.

  - **Meta-associação**
    - `catalogs:Catalog[0..*]`: um conjunto de catálogos que engloba refatorações.

- `Catalog` metaclasse utilizada para representar um catálogo de refatorações.
  - **Meta-atributos**
    - `name:String`: representa o nome do catálogo.
  - **Meta-associações**
    - `author:Author[0..1]`: representa o autor do catálogo;
    - `refactorings:Refactoring[0..*]`: conjunto de todas as refatorações que um catálogo possui.

- `Refactoring` representa uma das principais metaclasses do SRM.
  - **Meta-atributos**
    - `name:String`: utilizado para identificar a refatoração e ajudar a construir um vocabulário comum para os desenvolvedores de software;
    - `motivation:String`: descreve o motivo pelo qual a refatoração deve ser realizada e lista também as circunstâncias nas quais a refatoração deve ser utilizada;
    - `summary:String`: informa quando e onde uma determinada refatoração deve ser utilizada. Também é útil para auxiliar o engenheiro de software a identificar uma refatoração relevante em uma determinada situação.
  - **Meta-associações**
    - `operation:Operation[1]`: representa a ação que será executada, ou seja, representa o mecanismo da refatoração;
    - `preCondition:PreCondition[1]`: representa uma pré-condição que deve ser satisfeita antes da execução da operação/mecanismo da refatoração;
    - `postCondition:PostCondition[1]`: representa uma pós-condição que tem como intuito verificar a corretude da refatoração;
    - `elements:Element[0..*]`: um conjunto de elementos utilizados para realizar a refatoração. Tais elementos são metaclasses do KDM;
    - `chainOfRefactoring:Refactoring[0..*]`: um conjunto de refatorações que, quando combinadas, descrevem uma refatoração mais complexa;
    - `source:SourceContainer[0..*]`: define as metaclasses fontes do KDM que serão utilizadas durante a refatoração.

    * `target:TargetContainer[0..*]`: define as metaclasses alvos do KDM que serão utilizadas durante a refatoração.
    * `category:Category`: `String` para definir a categoria da refatoração.
    * `classification:Classification[0..*]`: define a classificação de uma refatoração.

- `Element` define um conjunto de elementos necessários para executar a refatoração. Essa metaclasse utiliza uma estrutura similar à da tabela *hash* para definir os elementos.

  – **Meta-atributos**

    * `key:String`: representa o nome do elemento;
    * `value:KDMEntity`: representa o tipo do elemento. Esse tipo deve uma metaclasse do metamodelo KDM.

- `SourceContainer` define um conjunto de metaclasses fontes do KDM que serão utilizadas durante a refatoração. Essa metaclasse utiliza uma estrutura similar a da tabela *hash* para definir os elementos.

  – **Meta-atributos**

    * `key:String`: representa o nome da metaclasse fonte;
    * `value:KDMEntity`: representa o tipo da metaclasse fonte. Esse tipo deve ser uma metaclasse do metamodelo KDM.

- `TargetContainer` define um conjunto de metaclasses alvos do KDM que serão utilizadas durante a refatoração. Essa metaclasse utiliza uma estrutura similar a da tabela *hash* para definir os elementos.

  – **Meta-atributos**

    * `key:String`: representa o nome da metaclasse alvo;
    * `value:KDMEntity`: representa o tipo da metaclasse alvo. Esse tipo deve ser uma metaclasse do metamodelo KDM.

- `Category` define um valor para informar qual a categoria da refatoração.

  – **Meta-atributos**

    * `name:String`: representa a categoria da refatoração.

- `Classification` define a classificação da refatoração.

  – **Meta-associações**

    * `level`: representa se a refatoração é *fine* ou *macro grained refactoring*;

    * `kdmPack`: define qual pacote do KDM é necessário para executar a refatoração.

- `Level` utilizado para definir se a refatoração é de granularidade baixa ou alta.

  – **Meta-atributos**

    * `kind:LevelKind`: especifica o nível da refatoração.

- `KDMPackage` define qual pacote do KDM é necessário para executar a refatoração.

  – **Meta-atributos**

    * `kind:KDMPackageKind`: representa qual pacote do KDM é necessário para executar a refatoração.

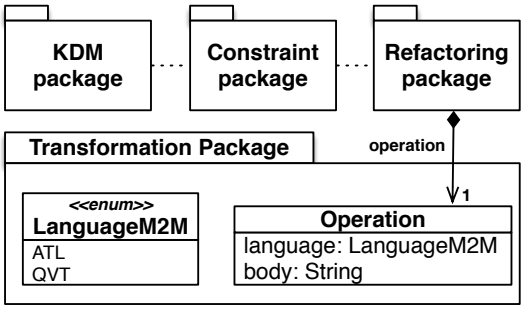
5.3.2.2 Pacote `Transformation`

Conceitualmente, refatorações são definidas por meio de um conjunto de passos que devem ser seguidos para realizar uma determina mudança (FOWLER, 1999; DEMEYER; BOIS; VERELST, 2004). Por outro lado, programaticamente, as refatorações são definidas como "programas" parametrizados que executam um conjunto de transformações seguindo uma ordem lógica. Usualmente, no contexto de modelos, tais transformações são conhecidas como endógenas e são implementadas utilizando técnicas de reescrita de grafo, ou como também é conhecida transformação de grafo (ver Capítulo 2 Seção 2.3.1). Dessa forma, pode-se caracterizar que técnica de reescrita de grafo é útil para auxiliar na elaboração de transformações de forma independente para *n* metamodelos, ou seja, a técnica de reescrita de grafo pode ser aplicada em qualquer metamodelo que implemente o padrão MOF, como por exemplo o metamodelo KDM. Comumente, transformações em modelos são definidas utilizando linguagens específicas de transformações de modelos. Diversas linguagens de transformação de modelos são usadas atualmente (BIEHL, 2010; ALLILAIRE *et al.*, 2006), entre elas pode-se citar ATL e QVT.

Neste contexto, o pacote `Transformation` contém apenas uma metaclasse e uma enumeração para definir o mecanismo da refatoração. O propósito dessa metaclasse é representar metadados de ATL e/ou QVT. Com o uso da ATL e/ou QVT, pode-se automatizar os mecanismos e todos os passos que uma determinada refatoração deve realizar. Dessa forma, como o propósito desse pacote é prover o reúso dos mecanismos e todos os passos de uma refatoração no contexto do metamodelo SRM, tanto ATL, quanto QVT foram consideradas boas candidatas para especificar programaticamente os metadados sobre o mecanismo das refatorações. Portanto, o SRM permite persistir metadados relacionados à ATL e QVT.

Na Figura 35, é mostrado esquematicamente o pacote `Transformation`, sua metaclasse e uma enumeração. Como ilustrado nessa figura, o pacote `Transformation` utiliza outros pacotes: (*i*) pacotes do KDM, (*ii*) pacote `Refactoring` e (*iii*) pacote `Constraint`. A seguir, uma descrição de cada elemento desse pacote é apresentado.

Figura 35 – Pacote `Transformation` do SRM.



Fonte: Elaborada pelo autor.

- `Operation` também representa uma das principais metaclasses do SRM. Essa metaclasse possui metadados do código responsável por realizar a transformação/refatoração.

  - **Meta-atributos**

    * `language:LanguageM2M`: especifica a linguagem que será usada para escrever o código responsável por realizar a transformação/refatoração. Valores válidos são: "ATL" e "QVT";
    * `body:String`: especifica a transformação/refatoração com base na linguagem selecionada, "ATL" ou "QVT".

- `LanguageM2M`: representa quais os tipos de linguagens de transformação de modelos que podem ser utilizados para implementar a refatoração. Os valores possíveis são: "ATL" ou "QVT";

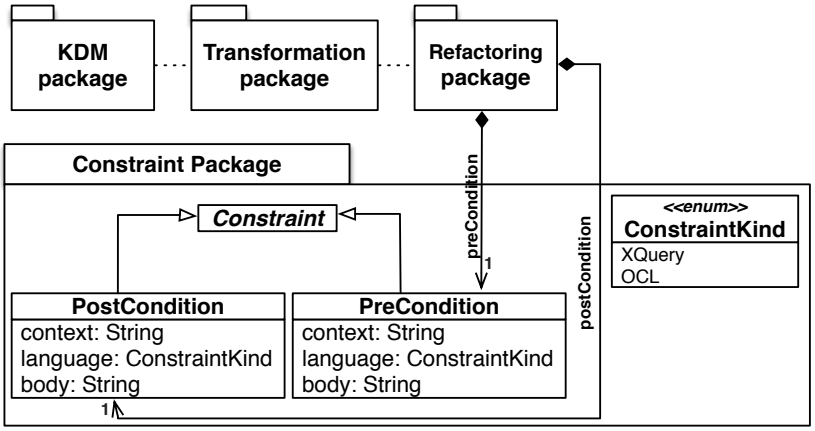
5.3.2.3 Pacote `Constraint`

É importante que o metamodelo SRM permita definir asserções e não apenas o mecanismo da refatoração, assim, engenheiros podem utilizar uma determinada instância do metamodelo SRM e verificar quais são as asserções que eles devem respeitar para executar a refatoração de forma correta. Tais asserções no contexto de refatorações são conhecidas como pré- e pós-condições (OPDYKE, 1992; ROBERTS, 1999). Asserções são úteis para garantir que a sintaxe e semântica do modelo sejam preservadas após a aplicação das refatorações, assegurando assim uma possível preservação de comportamento.

Neste contexto, o pacote `Constraint` do SRM contém três metaclasses e uma enumeração para definir as asserções (pré- e pós-condições) de uma determinada refatoração. O propósito desse pacote é representar metadados sobre asserções utilizando as linguagens OCL e/ou XQuery. Utilizando essas linguagens, é possível verificar, por exemplo, se todos os parâmetros obrigatórios para executar uma refatoração foram especificados corretamente. Assegurar que a refatoração será aplicada de forma correta é de suma importância para preservar a sintaxe e a semântica da instância do metamodelo, por exemplo, preservar comportamentos. Na Figura 36,

é mostrado o pacote `Constraint`, suas metaclasses e uma enumeração. A seguir, os elementos desse pacote são descritos:

Figura 36 – Pacote `Constraint` do SRM.



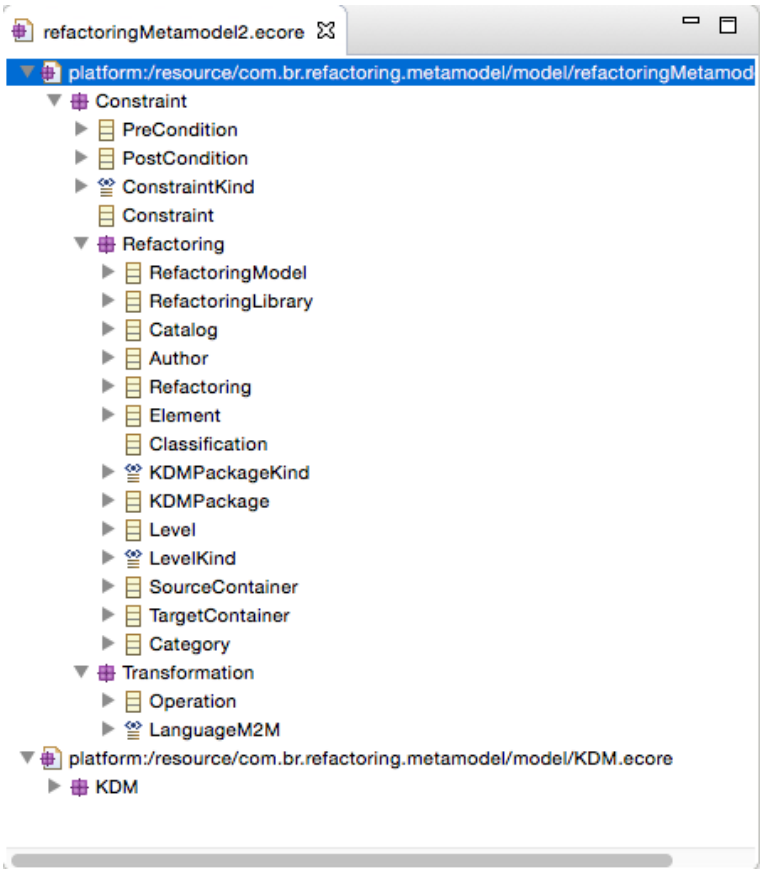
Fonte: Elaborada pelo autor.

- `Constraint` representa a metaclasse raiz de asserções. Em outras palavras, essa metaclasse indica de forma genérica, que todas as refatorações possuem asserções/restrições e ela é abstrata e não é instanciada.
- `PreCondition` define uma pré-condição para ser executada antes de operação/refatoração. Essa metaclasse possui três meta-atributos:
  - **Meta-atributos**
    - `context:String`: especifica o classificador para qual a pré-condição será definida;
    - `language:ConstraintKind`: especifica a linguagem que usada para escrever a pré-condição. Valor válido é: “OCL” ou “XQuery”;
    - `body:String`: especifica a OCL ou “XQuery” que representa a pré-condição.
- `PostCondition` define uma pós-condição para ser executada após a operação/refatoração. Essa metaclasse possui três meta-atributos:
  - **Meta-atributos**
    - `context:String`: especifica o classificador para qual a pós-condição será definido;
    - `language:ConstraintKind`: especifica a linguagem que será usada para escrever a pós-condição. Valor válido é: “OCL” ou “XQuery”;
    - `body:String`: especifica a OCL ou “XQuery” que representa a pós-condição.
- `ConstraintKind`: representa quais tipos de linguagens de restrições de modelos podem ser utilizados para implementar a asserção. Os valores possíveis são: “OCL” ou “XQuery”.

5.3.3 Implementação do SRM

Neste passo, o metamodelo SRM foi implementado. A junção dos pacotes apresentados nas seções anteriores e suas metaclasses forma o metamodelo SRM. Na Figura 30, é apresentado esquematicamente o metamodelo SRM. Como pode ser observado, o SRM contém 16 metaclasses e quatro enumerações.

Figura 37 – Visão de árvore do SRM implementado em EMF.



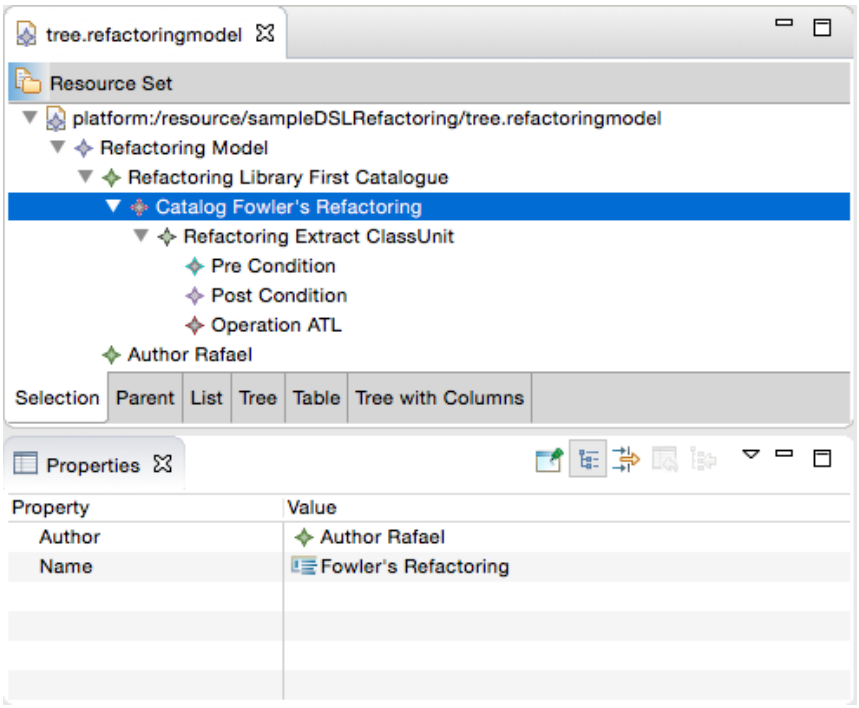
Fonte: Elaborada pelo autor.

Consistente com outros metamodelos definidos pela OMG, o SRM é definido utilizando a padronização de modelagem MOF e o *framework* EMF. Um dos benefícios de utilizar MOF é que ele permite que o metamodelo seja serializado e deserializado sem perder nenhum tipo de informação, ou seja, metamodelos instanciados são demonstrados utilizando uma representação textual padronizada (XMI). Além disso, o SRM é compatível com repositórios MOF para armazenamento e recuperação em várias ferramentas, aumentando a interoperabilidade de futuras ferramentas que utilizem esse metamodelo. Na Figura 37, é possível ver o SRM implementado em EMF. Todos os pacotes do metamodelo SRM contém internamente um conjunto de metaclasses e cada metaclasse contém seus meta-atributos e seus tipos.

5.4 Gramática da DSL utilizada para instanciar o SRM

Dado que o SRM foi implementado usando EMF existem duas opções para instanciá-lo: (*i*) utilizando uma visão de árvore do metamodelo e (*ii*) utilizando a API fornecida pelo EMF. A primeira opção, utiliza uma visão de árvore da instância do metamodelo SRM, como apresentado na Figura 38. Quando o engenheiro de modernização clicar com o botão direito do mouse, metaclasses do metamodelo são apresentadas para que ele escolha qual metaclasse especificar e instanciar. Porém, é visto que o engenheiro de modernização deve conhecer a estrutura correta do metamodelo SRM para utilizar a visão de árvore.

Figura 38 – Visão de árvore de uma instância do metamodelo SRM.



Fonte: Elaborada pelo autor.

A segunda opção é a utilização da API fornecida pelo EMF. Nela, a linguagem Java é utilizada juntamente com a API EMF. Para utilizar essa opção, os engenheiros de modernização necessitam ter conhecimento avançado de Java, isto é, a instanciação de uma refatoração utilizando a segunda opção é bastante verbosa, complexa e propensa a erros, pois exige conhecimentos avançados de refatoração e habilidades de programação em relação à API Ecore/EMF.

No Código-fonte 31, é apresentado um trecho onde é feita a instanciação em memória de algumas metaclasses definidas no metamodelo SRM, utilizando a segunda opção. Nesse código-fonte a instanciação das metaclasses do metamodelo SRM é um processo verboso e propenso a erros. Para instanciar o metamodelo SRM, o engenheiro deve saber utilizar a API do *framework* EMF. Metaclasses são instanciadas em EMF utilizando o conceito de fábrica (em inglês - *Factory*). Cada fábrica de uma metaclasse possui um método `createX`, onde X representa o nome da metaclasse do metamodelo.

Para criar uma instância válida do metamodelo SRM, o engenheiro de modernização deve seguir um conjunto de passos. Por exemplo, como pode ser observado na linha 1 do Código-fonte 31, uma instância da metaclasse `Author` é criada por meio da interface `RefactoringModelFactory` e do método `createAuthor()`. Em seguida, todos os meta-atributos da metaclasse `Author` devem ser especificados como apresentado nas linhas 2 e 3 do Código-fonte 31. Nas linhas 4-14, outras metaclasses do metamodelo SRM são instanciadas.

Código-fonte 31: Instanciação do metamodelo SRM.

```
1  Author author = RefactoringModelFactory.eINSTANCE.createAuthor();
2  author.setName("Rafael");
3  author.setLastName("Durelli");
4  RefactoringModel rM = RefactoringModelFactory.eINSTANCE.
       createRefactoringModel();
5  rM.setAuthor(author);
6  RefactoringLibrary library = RefactoringModelFactory.eINSTANCE.
       createRefactoringLibrary();
7  lib.setName("Fowler's refactorings");
8  lib.setDescription("Contains some Fowler's refactorings such as
       ExtractClass, RenameElements, PushMethod, PushAttribute, etc.");
9  lib.setShortDescription("Fine grained Refactorings");
10 rM.getLibraries().add(lib);
11 Catalog cat = RefactoringModelFactory.eINSTANCE.createCatalog();
12 cat.setAuthor(author);
13 cat.setName("Fowler's catalog");
14 libr.getCatalogs().add(cat);
15 ...
```

Como observado, o processo de instanciação do metamodelo SRM não é um processo trivial em ambas as opções. Além disso, engenheiros de modernização devem estar familiarizados como as particularidades das refatorações (por exemplo, qual(is) é (são) o(s) pré-requisito(s) para a execução de uma refatoração) e como/onde utilizar e programar tais refatorações. Com o intuito de facilitar e diminuir a quantidade de código-fonte, esforços obrigatórios e competências necessárias para instanciar refatorações utilizando o metamodelo SRM, foi criado uma DSL para auxiliar a instanciação de refatorações. A seguir, as gramáticas definidas que regem a DSL para auxiliar a instanciação do metamodelo SRM são apresentadas.

Código-fonte 32: Gramática da DSL - parte 1

```
1 ❶ grammar refactoring.xtext.SRM with org.eclipse.xtext.common.Terminals
2 ❷ import platform:/resource/refactoring/model/SRM.ecore
3 ❷ import platform:/resource/OMG/KDM/KDM.ecore
4 ❸ import http://www.eclipse.org/emf/2002/Ecore as ecore
5 RefactoringModel:
6     ❹ 'refactoringModel' name = ID '{'
7     ❺ author = Author
8     ❻ libraries += RefactoringLibrary*;
```

```
9     '}'
```

A DSL criada para auxiliar a instanciação do SRM foi desenvolvida utilizando Xtext[1]. Xtext é um *framework* do Eclipse[2] que tem como principal objetivo automatizar e agilizar o processo de desenvolvimento de DSLs. Em Xtext, a gramática segue uma notação similar ao *Backus–Naur Form* (BNF), chamada de regras do *parser*. Tais regras representam a sintaxe concreta da DSL. Note que para facilitar o entendimento da DSL, trechos dela são mostrados em listagens de códigos separados, bem como símbolos para explanar o propósito de uma determinada linha da gramática. No Código-fonte 32, é ilustrado o primeiro trecho da gramática concreta da DSL desenvolvida. A gramática começa com a definição do nome da DSL (SRM) (ver Código-fonte 32 ❶), em sequência são definidos os metamodelos que devem ser importados para serem utilizados durante a criação da DSL: o metamodelo SRM❷, o metamodelo KDM❷ e o Ecore❸.

Em seguida, é criado a primeira regra, a qual começa com a definição da metaclasse `RefactoringModel`. O corpo da regra começa logo após os ":". Primeiramente, para o entendimento da regra, é importante destacar que literais de *string* (em Xtext os literais podem ser expressos com aspas simples ou duplas) definem palavras-chave da DSL. Como pode ser observado no Código-fonte 32, é esperada a palavra-chave `refactoringModel`❹, seguido por um `ID` e "{". A gramática que rege o objeto `ID` é definida como uma sequência ilimitada de maiúsculas e minúsculas, números e o carácter de sublinhado embora possa, não começa por um dígito. A gramática que representa o nó `ID`❶ pode ser visualizada no Código-fonte 33.

Código-fonte 33: Gramática da DSL - parte 2

```
1     ❶ terminal ID: ('a'..'z' | 'A'..'Z'|'_')('a'..'z' | 'A'..'Z'|'_'|'0'..'9')*;
```

Ainda no Código-fonte 32, a expressão `author=Author`❺ especifica que pode-se instanciar uma instância da metaclasse `Author`. A expressão `(libraries += RefactoringLibrary)*`❻ descrita no Código-fonte 32 especifica que é possível instanciar várias instâncias da metaclasse `RefactoringLibrary`. O operador estrela, "*", ilustra que o número de elementos (nesse caso `RefactoringLibrary`) é arbitrário; em particular, ele pode ser qualquer número >= 0. Operador += por sua vez representa que a propriedade `libraries` será uma lista do tipo `RefactoringLibrary`.

Código-fonte 34: Gramática da DSL - parte 3

```
1 Author:
2     ❶ 'author' '{'
3     ❷ 'name' ':' name = ID
4         ➥❸ 'lastName' ':' lastName = ID;
5 '}'
6 RefactoringLibrary:
7     ❹ 'refactoringLibraries' '{'
```

[1] https://www.eclipse.org/Xtext/
[2] https://www.eclipse.org

```
8       ❺ 'name' ':' name = ID
9           ➥ 'shortDescription' ':' shortDescription = STRING
10          ➥ 'description' ':' description = STRING
11          ➥ ❻ catalogs += Catalog*
12  '}'
```

A definição das regras que regem as metaclasses `Author` e `RefactoringLibrary` são apresentadas no Código-fonte 34. A regra para a definição de `Author` começa com a definição da palavra-chave `author`, seguida por um "{"❶. Em seguida, a palavra-chave `name` é esperada, seguida por ":" ❷. Posteriormente, a palavra-chave `lastName` também é esperada, seguida por ":" ❸. Na linha 6 do Código-fonte 34, começa a definição da regra da metaclasse `RefactoringLibrary`. A regra para a definição de `RefactoringLibrary` começa com a definição da palavra-chave `refactoringLibraries`, seguida por um "{"❹. Depois, deve-se especificar a palavra-chave `name` e `:`. Posteriormente, as palavras-chave `shortDescription` e `description` são especificadas nas linhas 9 e 10, respectivamente. A expressão descrita na linha 11 representa que pode haver qualquer número de instâncias da metaclasse `Catalog`.

Código-fonte 35: Gramática da DSL - parte 4

```
1  Catalog:
2      ❶'catalog' '{'
3          ➥❷'name' ':' name=ID
4          ➥❸'author' ':' author=[Author]
5          ➥❹refactorings += Refactoring*
6      '}'
7  Refactoring:
8      'refactoring' '{'
9          ➥'name' ':' name = ID
10         ➥'motivation' ':' motivation = STRING
11         ➥'summary' ':' summary = STRING
12         ➥❺'source' ':' '{'source += [SourceContainer] (','(source += [
       SourceContainer]))*'}'
13         ➥❺'target' ':' '{'target += [TargetContainer] (','(target += [
       TargetContainer]))*'}'
14         ➥❺ category = Category?
15         ➥❺'elements' ':' '{'elements += [Element] (','(elements += [Element]))*'}'
16         ➥❻ operation = Operation?
17         ➥❼ preCondition = PreCondition?
18         ➥❼ postCondition = PostCondition?
19         ➥❽('containedRefactoring' ':' chainOfRefactoring+=Refactoring)*
20     '}'
```

O Código-fonte 35 representa as sintaxes concretas das metaclasses `Catalog` e `Refactoring`. A sintaxe concreta da metaclasse `Catalog` começa com a palavra-chave `catalog`, seguida por um "{"❶. Depois, o nome do catálogo de refatoração deve ser especificado por meio da palavra-chave `name` ❷. Posteriormente, especifica-se uma instância da metaclasse `Author`, informando quem é o autor desse catálogo de refatoração (ver Código-fonte 35❸). Na linha 5 do Código-fonte 35, deve-se informar várias instâncias da metaclasse `Refactoring`; essa sintaxe representa as refatorações que compõem esse catálogo de refatoração. Nas linhas 7-18, a definição

da sintaxe concreta para especificar uma refatoração por meio do metamodelo SRM é apresentada. Inicialmente, uma refatoração deve possuir um nome, conforme ilustrado na linha 9 do Código-fonte 35, e posteriormente, a motivação, bem como o resumo da refatoração também devem ser especificados, conforme apresentado nas linhas 10 e 11 do Código-fonte 35. As linhas 12, 13, 14 e 15 informam que as metaclasses dos tipos `SourceContainer`❺, `TargetContainer`❺, `Category`❺ e `Element`❺ devem ser instanciadas. Da mesma forma, as linhas 16, 17 e 18 informam que metaclasses dos tipos `Operation`❻, `PreCondition`❼ e `PostCondition`❼ devem ser instanciadas, respectivamente. A linha 19 representa a sintaxe da DSL para especificar um conjunto de refatorações que, quando combinadas, podem realizar refatorações complexas.

Código-fonte 36: Gramática da DSL - parte 5

```
1  SourceContainer:
2      ❶ key=STRING value=KDMEntity
3  TargetContainer:
4      ❷ key=STRING value=KDMEntity
5  Category:
6      ❸'category' '{'
7          ➥'name' ':' name=STRING
8      '}'
9  Element:
10     ❹ key=STRING value=KDMEntity
```

No Código-fonte 36, as sintaxes concretas das metaclasses `SourceContainer`, `TargetContainer`, `Category` e `Element` são definidas. A sintaxe concreta da metaclasse `SourceContainer` inicia com a palavra-chave `source`, seguida por um "{"❶. Depois, deve-se especificar a palavra-chave `key`, seguida por ":". Posteriormente, a palavra-chave `value` também é esperada seguida por ":". Essa palavra-chave, `value`, armazena um determinado elemento fonte da metaclasse do metamodelo KDM. Da mesma forma, a sintaxe concreta da metaclasse `TargetContainer` inicia-se com a palavra-chave `target`, seguida por um "{"❷. Em seguida, também deve-se especificar a palavra-chave `key`, seguida por ":". A palavra-chave `value` seguida por ":" é esperada. Novamente, essa palavra-chave armazena um determinado elemento alvo que é uma metaclasse do metamodelo KDM. A sintaxe concreta da metaclasse `Category` inicia-se com a palavra-chave `category`, seguida por ":" ❸. Deve-se, então, especificar a palavra-chave `name` seguida por ":". Nas linhas 9 e 10 do Código-fonte 36, a sintaxe concreta da metaclasse `Element` é definida.

Código-fonte 37: Gramática da DSL - parte 6

```
1  Operation:
2      ❶'operation' '{'
3          ➥'language' ':' language=LanguageM2M
4          'body' ':' '{'
5              ➥❸body = STRING
6          '}'
7      '}'
8  PreCondition:
9      ❹'preCondition' '{'
```

```
10          ➥'context' ':' context=STRING
11          ➥'language' ':' language=ConstraintKind
12          'body' ':' '{'
13              ➥body=STRING
14          '}'
15      '}'
16  PostCondition:
17      ❺'postCondition' '{'
18          ➥'context' ':' context=STRING
19          ➥'language' ':' language=ConstraintKind
20          'body' ':' '{'
21              ➥body=STRING
22          '}'
23      '}'
24  enum LanguageM2M:
25      ATL | QVT
26  enum ConstraintKind:
27      XQuery | OCL
```

No Código-fonte 37, as sintaxes concretas das metaclasses `Operation`, `PreCondition` e `PostCondition` são definidas. A sintaxe concreta da metaclasse `Operation` inicia com a palavra-chave `operation`, seguida por um "{"❶. Depois, deve-se especificar em qual linguagem a operação/refatoração será escrita. `LanguageM2M` é uma enumeração (*enum*) que está representado na linha 24 do Código-fonte 37. A linha 5 representa a sintaxe concreta para especificar o "corpo" da operação/refatoração propriamente dita ❸. Na linha 8, a sintaxe concreta da metaclasse `PreCondition` é definida e inicia-se com a palavra-chave `preCondition`, seguida por um "{"❶, `context`, `language` e `body`. A sintaxe concreta da metaclasse `PostCondition` é definida nas linhas 16 até 23. Pode-se notar que a sintaxe é similar à sintaxe definida na metaclasse `PreCondition`. A partir das gramáticas da DSL apresentadas, XText gera um editor textual no ambiente de desenvolvimento Eclipse IDE. Esse editor provê *highlighting* de sintaxe, *code completion* e *code navigation*. O editor resultante pode ser observado no Capítulo 6 na Seção 6.4, onde é apresentada a ferramenta KDM-RE.

5.4.1 Exemplo de uso da DSL

A partir das gramáticas da DSL apresentadas o engenheiro de modernização pode instanciar o metamodelo SRM. A DSL contém 23 palavras-chave, as quais são ilustradas na Tabela 13. A DSL também contém as palavras-chaves `ATL`, `QVT`, `OCL` e `XQuery` que são enumerações literais para representar a linguagem de transformação e a linguagem de restrição da instância de uma refatoração, respectivamente. A DSL também contém alguns caracteres *American Standard Code for Information Interchange* (ASCII), tais como: `:`, `"`, `{` e `}`.

Nesta seção, exemplos graduais são apresentados para facilitar o entendimento de como utilizar a DSL. Dessa forma, para ilustrar o uso da DSL nessa seção, suponha que o engenheiro de modernização almeje prover o reúso da refatoração `Extract ClassUnit` por meio da DSL. Dessa forma, as regras que regem as gramáticas devem ser seguidas. De acordo com a gramática

Tabela 13 – Palavras-chave da DSL.

Palavras-chave da DSL		
`refactoringModel`	`author`	`name`
`lastName`	`refactoringLibraries`	`shortDescription`
`description`	`catalog`	`refactoring`
`motivation`	`summary`	`operation`
`language`	`body`	`preCondition`
`context`	`postCondition`	`key`
`value`	`source`	`target`
`category`	`elements`	—

definida no Código-fonte 32, primeiramente, o engenheiro de modernização deve instanciar a metaclasse `RefactoringModel`, isto é feito por meio da palavra-chave `refactoringModel`, seguida por { e } como apresentado no Código-fonte 38.

Código-fonte 38: Exemplo de uso da DSL - parte 1.

```
1 refactoringModel {
2
3 }
```

De acordo com a gramática (ver Código-fonte 32), o próximo passo consiste na especificação do autor da refatoração. Isto é realizado por meio da palavra-chave `author`, seguida por { e }. Dentro do escopo da palavra-chave `author`, deve-se especificar o nome e o sobrenome do autor, como apresentado no Código-fonte 39.

Código-fonte 39: Exemplo de uso da DSL - parte 2.

```
1 refactoringModel {
2     author {
3         name : Rafael
4         lastName : Durelli
5     }
6 }
```

Posteriormente, deve-se instanciar a metaclasse `RefactoringLibrary` e seus meta-atributos. De acordo com a gramática da DSL, isso é feito por meio da palavra-chave `refactoring-Libraries` seguida por { e } (ver Código-fonte 40 linha 6).

Código-fonte 40: Exemplo de uso da DSL - parte 3.

```
1 refactoringModel {
2     author {
3         name : Rafael
4         lastName : Durelli
```

```
5       }
6       refactoringLibraries {
7           name : FineGrainedRefactoring
8           shortDescription : "contains a set of refactorings"
9           description : "refactorings"
10      }
11  }
```

Em seguida, deve-se instanciar a metaclasse `Catalog`. De acordo com a gramática apresentada no Código-fonte 35, isso é feito por meio da palavra-chave `catalog` seguida por { e } (ver Código-fonte 41, linhas 10-12).

Código-fonte 41: Exemplo de uso da DSL - parte 4.

```
1  refactoringModel {
2      author {
3          name : Rafael
4          lastName : Durelli
5      }
6      refactoringLibraries {
7          name : FineGrainedRefactoring
8          shortDescription : "contains a set of refactorings"
9          description : "refactorings"
10         catalog {
11             name : Fowler Catalog
12             author : Rafael
13         }
14     }
15 }
```

Cada catálogo de refatoração contém um conjunto de refatorações. Dessa forma, o próximo passo é realizar a instanciação da metaclasse `Refactoring` e seus meta-atributos. Isto pode ser realizado por meio da palavra-chave `refactoring` seguida por { e } como apresentado na gramática da DSL (ver Código-fonte 35).

Código-fonte 42: Exemplo de uso da DSL - parte 5.

```
1  refactoringModel {
2      author {
3          name : Rafael
4          lastName : Durelli
5      }
6      refactoringLibraries {
7          name : FineGrainedRefactoring
8          shortDescription : "contains a set of refactorings"
9          description : "refactorings"
```

```
10          catalog {
11              name : Fowler Catalog
12              author : Rafael
13              refactoring {
14                  name : Extract ClassUnit
15                  motivation : "You have one ClassUnit doing work that
    should be done by two."
16                  summary : "Create a new ClassUnit and move the relevant
    StorableUnit from the old ClassUnit into the new ClassUnit."
17              }
18          }
19      }
20  }
```

Como observado no Código-fonte 42, a metaclasse `Refactoring` é instanciada nas linhas 13-16 e seus meta-atributos também devem ser informados, ou seja, `name`, `motivation` e `summary`. Após instanciar a metaclasse `Refactoring`, o próximo passo consiste na instanciação das metaclasses `SourceContainer`, `TargetContainer`, `Category` e `Element` como apresentado no Código-fonte 43.

Código-fonte 43: Exemplo de uso da DSL - parte 5.

```
1  refactoringModel {
2      author {
3          name : Rafael
4          lastName : Durelli
5      }
6      refactoringLibraries {
7          name : FineGrainedRefactoring
8          shortDescription : "contains a set of refactorings"
9          description : "refactorings"
10         catalog {
11             name : Fowler Catalog
12             author : Rafael
13             refactoring {
14                 name : Extract ClassUnit
15                 motivation : "You have one ClassUnit doing work that
    should be done by two."
16                 summary : "Create a new ClassUnit and move the relevant
    StorableUnit from the old ClassUnit into the new ClassUnit."
17                 source : {
18                     Package, ClassUnit
19                 }
20                 target : {
21                     Package, ClassUnit
22                 }
```

```
23                  category : {
24                      name: "Moving Features Between Objects"
25                  }
26                  elements : {
27                      ClassUnit, StorableUnit
28                  }
29              }
30          }
31      }
32  }
```

Como observado no Código-fonte 42, a metaclasse `Refactoring` é instanciada nas linhas 13-29 e seus meta-atributos também devem ser informados, ou seja, `name`, `motivation` e `summary`. Após instanciar a metaclasse `Refactoring`, o próximo passo consiste na instanciação da metaclasse `Operation` como apresentado no Código-fonte 44 linhas 29-33. Note que, para facilitar o entendimento da DSL o código em ATL (ver linhas 31-33) que representa a operação da refatoração foi removido.

Código-fonte 44: Exemplo de uso da DSL - parte 6.

```
1  refactoringModel {
2      author {
3          name : Rafael
4          lastName : Durelli
5      }
6      refactoringLibraries {
7          name : FineGrainedRefactoring
8          shortDescription : "contains a set of refactorings"
9          description : "refactorings"
10         catalog {
11             name : Fowler Catalog
12             author : Rafael
13             refactoring {
14                 name : Extract ClassUnit
15                 motivation : "You have one ClassUnit doing work that
       should be done by two."
16                 summary : "Create a new ClassUnit and move the relevant
       StorableUnit from the old ClassUnit into the new ClassUnit."
17                 source : {
18                     Package, ClassUnit
19                 }
20                 target : {
21                     Package, ClassUnit
22                 }
23                 category : {
24                     name: "Moving Features Between Objects"
```

```
25                  }
26                  elements : {
27                      ClassUnit, StorableUnit
28                  }
29                  operation {
30                      language : ATL
31                      body : {
32                          TO BE DEFINED
33                      }
34                  }
35              }
36          }
37      }
38  }
```

Código-fonte 45: Exemplo de uso da DSL - parte 7.

```
1  refactoringModel {
2      author {
3          name : Rafael
4          lastName : Durelli
5      }
6      refactoringLibraries {
7          name : FineGrainedRefactoring
8          shortDescription : "contains a set of refactorings"
9          description : "refactorings"
10         catalog {
11             name : Fowler Catalog
12             author : Rafael
13             refactoring {
14                 name : Extract ClassUnit
15                 motivation : "You have one ClassUnit doing work that
       should be done by two."
16                 summary : "Create a new ClassUnit and move the relevant
       StorableUnit from the old ClassUnit into the new ClassUnit."
17                 source : {
18                     Package, ClassUnit
19                 }
20                 target : {
21                     Package, ClassUnit
22                 }
23                 category : {
24                     name: "Moving Features Between Objects"
25                 }
26                 elements : {
27                     ClassUnit, StorableUnit
28                 }
```

```
29                  operation {
30                      language : ATL
31                      body : {
32                          TO BE DEFINED
33                      }
34                  }
35                  preCondition {
36                      context : ClassUnit
37                      language : OCL
38                      body : {
39                          "TO BE DEFINED"
40                      }
41                  }
42                  postCondition {
43                      context : ClassUnit
44                      language : OCL
45                      body : {
46                          "TO BE DEFINED"
47                      }
48                  }
49          }
50      }
51  }
```

Como apresentado na gramática da DSL (ver Código-fonte 35), após instanciar a metaclasse `Operation` deve-se instanciar as asserções, ou seja, instanciar as metaclasses `PreCondition` e `PostCondition`, respectivamente. No Código-fonte 45, as asserções da refatoração são instanciadas por meio da DSL. Nas linhas 35-41 a metaclasse `PreCondition` é instanciada com os seus respectivos metadados e a metaclasse `PostCondition` é instanciada nas linhas 42-48. Note que, para facilitar o entendimento da sintaxe da DSL os códigos em OCL que representam os metadados do meta-atributo `body` foram removidos.

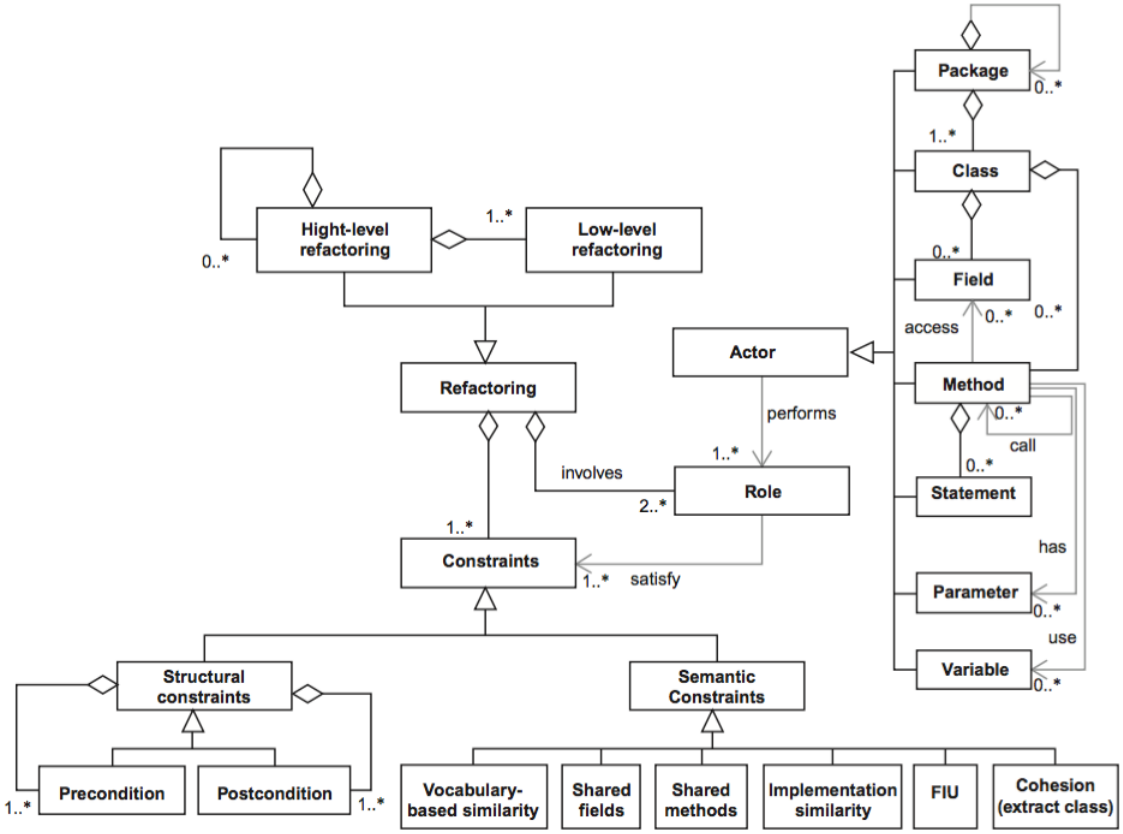
5.5 Trabalhos Relacionados

Poucos trabalhos relacionados foram identificados na literatura sobre metamodelos para especificar e promover o reúso de refatorações. Porém, alguns trabalhos foram identificados e nortearam a criação do SRM. Assim, nesta seção, são mostradas as principais semelhanças e diferenças encontradas entre eles.

Ouni, Kessentini e Sahraoui (2014) definiram um metamodelo, o qual é apresentado na Figura 39. Note que esse metamodelo é semelhante ao metamodelo aqui proposto. A metaclasse `Refactoring` representa a principal entidade do metamodelo. Refatorações são classificadas em duas principais metaclasses: (*i*) `Low-Level Refactoring` e (*ii*) `High-Level Refactoring`.

A primeira metaclasse representa operações atômicas (`add`, `delete` e `change`). Tais operações quando combinadas são representadas pela metaclasse `High-Level Refactoring` e representam refatorações mais complexas, tais como: *Move Method*, *Extract Class*, etc. Similarmente, o metamodelo SRM aqui apresentado também permite especificar operações atômicas e refatorações complexas. A representação de refatorações complexas é feita no SRM por meio da associação denominada `chainOfRefactoring` da metaclasse `Refactoring` - por meio dessa associação, os engenheiros de modernização podem compor um conjunto de refatorações complexas.

Figura 39 – RefMetamodel.



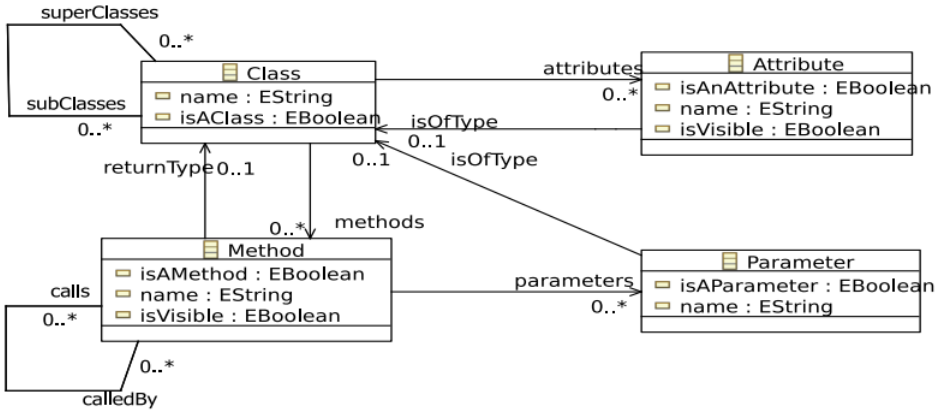
Fonte: Ouni, Kessentini e Sahraoui (2014).

De acordo com Ouni, Kessentini e Sahraoui (2014), para aplicar uma refatoração primeiramente é necessário definir qual elemento estrutural é envolvido/impactado pela refatoração. O elemento estrutural é definido pela metaclasse `Actor` e suas especificações: `Package`, `Class`, `Field`, `Method`, etc. No SRM, o engenheiro de modernização também deve definir quais são os elementos estruturais que são envolvidos na operação da refatoração. Porém, tais elementos estruturais são representados por metaclasses do metamodelo KDM - fazendo com que as refatorações especificadas no SRM sejam independentes de linguagem e plataforma.

Como observado na Figura 39, é possível definir dois tipos de restrições: (*i*) `Structural Constraints` e (*ii*) `Semantic Constraints`. Por meio da metaclasse `Structural Constraints` é possível especificar as pré- e pós-condições de uma refatoração. Restrições de semântica

são especificadas pelas seguintes metaclasses: (*i*) `Vocabulary-Based Similarity`, `Shared Fields`, `Shared Methods`, etc. Diferentemente, o SRM contém apenas três metaclasses para representar as restrições da refatoração (`Constraint`, `PreCondition` e `PostCondition`). Porém, o SRM permite que o engenheiro de modernização especifique restrições em OCL ou XQuery. Além disso, a principal diferença entre o SRM e o metamodelo apresentado na Figura 39 é que o SRM não tem como objetivo apenas permitir a especificação e documentação de refatorações. O SRM permite compartilhar refatorações, prover o reúso e aumentar a interoperabilidade entre diversas ferramentas, assim, qualquer apoio computacional que utilize o metamodelo KDM poderá utilizar as refatorações definidas e especificadas pelo SRM.

Figura 40 – *Generic Metamodel* - GenericMT.



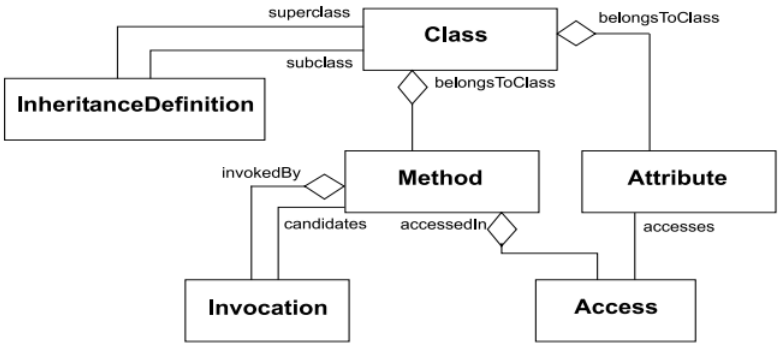
Fonte: Sen *et al.* (2012).

Sen *et al.* (2012) definem uma abordagem para especificar refatorações de forma genérica. Os autores apresentam um metamodelo denominado *GenericMT* (ver Figura 40), o qual contém algumas construções do POO (classes, métodos, atributos, parâmetros, etc.). Por meio desse metamodelo, refatorações genéricas podem ser criadas. A principal semelhança entre o SRM é o GenericMT, é que ambos objetivam ser independentes de plataformas. As principais diferenças são: (*i*) GenericMT não possui metaclasses para representar metadados de refatoração, (*ii*) GenericMT não possui metaclasses para representar restrições (pré- e pós-condições), (*iii*) GenericMT não foi definido para permitir a interoperabilidade entre diversas ferramentas e (*iv*) GenericMT não utiliza nenhum metamodelo padronizado (UML/KDM) para especificar os elementos que serão refatorados.

Um metamodelo similar ao GenericMT é definido por Tichelaar *et al.* (2000). Os autores introduzem um metamodelo denominado FAMIX (ver Figura 41), o qual permite a criação de refatorações para elementos estruturais do POO (classes, objetos, métodos, etc.) e todas as refatorações criadas utilizam a ferramenta MOOSE (DUCASSE *et al.*, 2005). O metamodelo FAMIX é apresentado na Figura 41. A principal semelhança entre o SRM e o FAMIX é que ambos objetivam ser independente de plataformas e as principais diferenças entre esses metamodelos são: (*i*) FAMIX não contém metaclasses para representar informações sobre refatoração, (*ii*)

FAMIX também não possui metaclasses para representar metadados sobre as restrições (pré- e pós-condições), (*iii*) FAMIX não foi definido para permitir a interoperabilidade entre diversas ferramentas, FAMIX é utilizado apenas na ferramenta MOOSE (DUCASSE *et al.*, 2005) e (*iv*) FAMIX também não utiliza nenhum metamodelo padronizado (UML/KDM) para especificar os elementos que serão refatorados.

Figura 41 – Metamodelo FAMIX.



Fonte: Tichelaar *et al.* (2000).

Tabela 14 – Comparação entre o metamodelo SRM e os metamodelos relacionados.

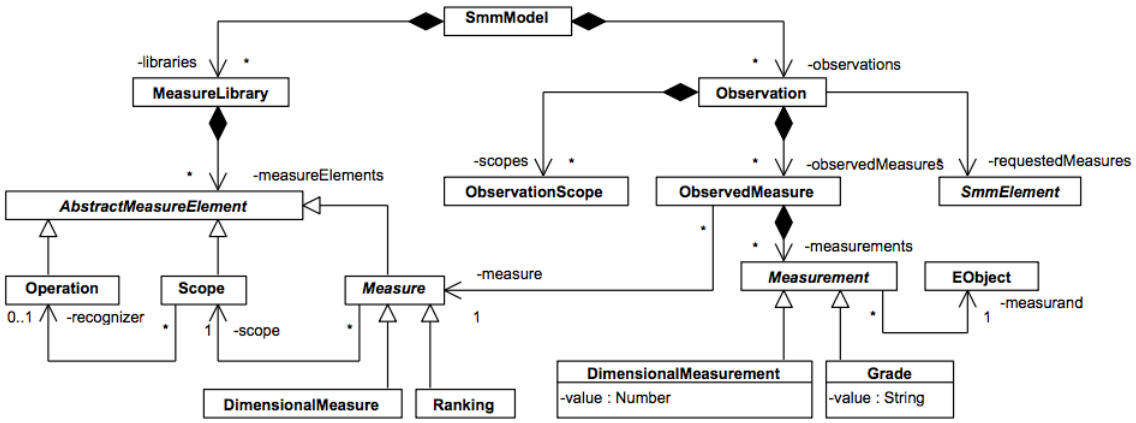
Metamodelo	PMPRR	PMPRRe	FC	IP	I
Structured Refactoring Metamodel - SRM	✓	✓	++	✓	✓
RefMetamodel (OUNI; KESSENTINI; SAHRAOUI, 2014)	✓	✓	+	✗	✗
GenericMT (SEN *et al.*, 2012)	✗	✗	+++	✓	✗
FAMIX (TICHELAAR *et al.*, 2000)	✗	✗	+++	✓	✗

Na Tabela 14 é apresentada uma comparação entre o metamodelo SRM e os metamodelos identificados quanto à forma como representam as refatorações. Dessa forma, alguns critérios foram definidos durante a comparação, tais critérios são: (*i*) o metamodelo fornece metaclasses para representar refatorações, (*ii*) o metamodelo fornece metaclasses para representar restrições (pré- e pós-condições), (*iii*) qual a facilidade de compreensão do metamodelo, (*iv*) o metamodelo é independente de plataforma e (*v*) o metamodelo provê a interoperabilidade. A Tabela 14 contém seis colunas e algumas foram abreviadas: (*i*) "PMPRR" significa "**P**ossui **M**etaclasses **P**ara **R**epresentar **R**efatorações", (*ii*) "PMPRRe" significa "**P**ossui **M**etaclasses **P**ara **R**epresentar **Re**strições", (*iii*) "FC" representa qual a "**F**acilidade de **C**ompreensão do metamodelo", (*iv*) "IP" representa se o "metamodelo é **I**ndependente de **P**lataforma" e (*v*) "I" informa se o "metamodelo provê a **I**nteroperabilidade entre diversas ferramentas". O símbolo "✓" representa que o metamodelo define o critério e o símbolo "✗" representa que o metamodelo não define o critério. O símbolo "+" representa o grau de facilidade de compreensão do metamodelo, sendo que "+++" representa maior facilidade de compreensão e "+" representa menor facilidade de compreensão.

Não foram encontrados outros metamodelos similares ao SRM. No entanto, na literatura é possível identificar o metamodelo SMM, o qual é uma padronização da ADM e é utilizado para

especificar métricas de software (ADM, 2016a). A ideia é similar ao metamodelo SRM, por isso o metamodelo SMM foi considerado um trabalho relacionado. O SMM é um metamodelo que proporciona um formato padrão tanto para definir métricas quanto para representar os resultados das medições. Um trecho do SMM é apresentado na Figura 42 e suas principais metaclasses do SMM são: `Measures` (métricas) e `Measurements` (medições). A metaclasse `Measures` descreve uma maneira para calcular propriedades de um sistema. A metaclasse `Measurements` representa medições, mais especificamente resultados da aplicação de uma métrica em um modelo alvo a ser medido. Existem algumas vantagens no uso do SMM. Além de ser um metamodelo para métricas e medições proposto pela ADM, ele se faz vantajoso pelo fato de ser um padrão, logo, métricas definidas em conformidade com o metamodelo SMM podem ser testadas, reusadas e compartilhadas da mesma forma que o SRM pode promover o compartilhamento e reúso de refatorações. Além disso, o uso de um padrão facilita o desenvolvimento de ferramentas compatíveis capazes de automatizar o processo de medição (ADM, 2016a).

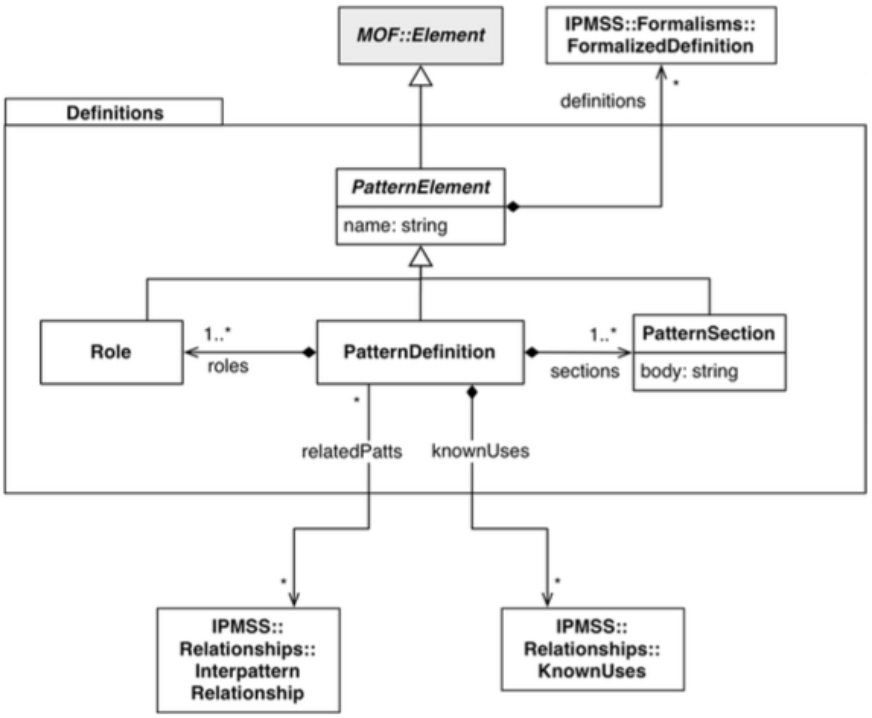
Figura 42 – *Structured Metrics Metamodel* - SMM.



Fonte: ADM (2016a).

Outro metamodelo relacionado ao SRM é o metamodelo *Structured Patterns Metamodel Standard* (SPMS) (ADM, 2016b). Esse metamodelo é utilizado para representar padrões de software, tais como: *anti-patterns*, *design patterns*, *architectural patterns*, *build patterns* e etc. SPMS fornece um conjunto de metaclasses para instanciar diversos padrões de software. Uma visão geral do metamodelo SPMS é apresentado na Figura 43. A metaclasse `PatternElement` possui meta-atributos para representar o padrão que se almeja instanciar. `Role` é uma metaclasse para representar elementos conceituais que um determinado padrão necessita. A metaclasse `PatternDefinition` é uma submetaclasse de `PatternElement` e possui duas composições: (*i*) `sections` do tipo `PaternsSection` e (*ii*) knownUses do tipo `KnownUses`. Além disso, a metaclasse `PatternDefinition` tem uma dependência denominada `relatedPatts` do tipo `InterPatternRelationship` que representa os padrões internos que um padrão pode possuir.

Figura 43 – *Structured Patterns Metamodel Standard* - SPMS.



Fonte: ADM (2016b).

5.6 Considerações Finais

Refatorações são de suma importância durante a manutenção, produção e análise de software. Inúmeras comunidades têm surgido na literatura para criar e definir vários tipos de refatorações, incluindo refatorações de baixa granularidade (FOWLER, 1999; DEMEYER; BOIS; VERELST, 2004; DEMEYER, 2005), refatorações arquiteturais, refatorações para o paradigma orientado a aspectos, etc.

Diante disso, há uma grande necessidade da definição de um padrão para auxiliar e promover o compartilhamento dessas refatorações, tanto dentro, como entre essas comunidades. Como uma iniciativa para suprir tal limitação, neste capítulo foi apresentado o metamodelo SRM para auxiliar o engenheiro a promover o reúso de refatorações. O SRM está inserido no contexto da ADM para preencher a definição de um metamodelo de refatorações. Assim, com a utilização desse metamodelo, metadados sobre refatorações podem ser reutilizados de forma independente de linguagem e plataforma.

O SRM contém 16 metaclasses e quatro enumerações e foi implementado utilizando o EMF. Para a representação dos elementos estruturais que são utilizados durante uma refatoração, o metamodelo KDM foi utilizado. O mecanismo/operação da refatoração é representados utilizando a linguagem de transformação ATL e/ou QVT. As restrições que devem ser satisfeitas antes e após a condução da refatoração são representadas por meio de OCL e/ou XQuery. A fim de utilizar plenamente as vantagens dos metamodelo SRM, os engenheiros de modernização

precisam ter um bom conhecimento de linguagem de programação avançada. Na verdade, o engenheiro de modernização deve estar familiarizado como as semânticas das refatorações (por exemplo, qual(is) é (são) o(s) pré-requisito(s) para a execução de uma refatoração) e como/onde utilizar e programar tais refatorações. Além disso, a instanciação de uma refatoração utilizando o SRM é bastante verbosa, complexa e propensa a erros, uma vez que exige o conhecimento avançado de refatoração e habilidades de programação em relação à API Ecore, uma vez que o SRM foi desenvolvido utilizando o EMF. Nesse contexto, foi criado e apresentado, neste capítulo, um conjunto de gramáticas que, quando compostas, derivam uma DSL, a qual é utilizada para facilitar a utilização e instanciação do metamodelo SRM. Uma visão geral da DSL e como utilizá-la também foi apresentada neste capítulo. O editor resultante pode ser observado no Capítulo 6.

CAPÍTULO

6

A FERRAMENTA KDM-RE

6.1 Considerações Iniciais

Este capítulo apresenta uma ferramenta denominada *Knowledge Discovery Model-Refactoring Environment* (KDM-RE). A ferramenta KDM-RE é um protótipo para auxiliar o processo de modernização e ela utiliza internamente os seguintes metamodelos: (*i*) UML, (*ii*) KDM e (*iii*) SRM. Como pré-requisito para utilizar a ferramenta KDM-RE e iniciar o processo de modernização ilustrado na Figura 1 ❸, o engenheiro de software deve primeiramente converter o sistema a ser modernizado para uma instância do metamodelo KDM. Em seguida, a instância do KDM deve ser convertida para uma instância do metamodelo UML. Após realizar essas conversões, a instância do metamodelo UML pode então ser visualizada por meio de diagrama de classes da UML, o qual é utilizado como interface gráfica durante o processo de modernização. Dessa forma, o sistema a ser modernizado pode ser visualizado graficamente facilitando a identificação de problemas estruturais. Durante o processo de modernização, o engenheiro de software pode interagir com o diagrama de classes e escolher um conjunto de refatorações a ser aplicadas. Assim que o engenheiro de software escolhe uma determinada refatoração, um *Wizard* é executado, onde o engenheiro deve fornecer informações para a correta execução da refatoração. As informações providas nesse *Wizard* são enviadas para uma linguagem de transformação que é responsável por realizar a refatoração na instância do metamodelo KDM. Além disso, KDM-RE também utiliza internamente o metamodelo SRM (ver Capítulo 5), o qual permite a potencialização do reúso de refatorações em nível de modelo. Assim, refatorações podem ser reutilizadas e compartilhadas por meio da KDM-RE.

A ferramenta KDM-RE foi implementada como um conjunto de *plug-ins* para o Ambiente de Desenvolvimento Eclipse IDE[1]. Cada *plug-in* representa um módulo da KDM-RE responsável por demonstrar a viabilidade dos conceitos descritos nos Capítulos 4 e 5. Por exemplo, KDM-RE dá apoio a refatorações criadas por meio da abordagem mostrada no Capítulo 4. Além disso, o

[1] `https://eclipse.org/home/index.php`

formato exigido pela ferramenta para disponibilização das refatorações é o metamodelo mostrado no Capítulo 5. Dessa forma, observando a Figura 1 ❸, nota-se que a ferramenta mostrada neste capítulo é a infraestrutura base para que toda a abordagem opere adequadamente. Nota-se que este Capítulo é uma extensão do seguinte artigo: *KDM-RE: A Model-Driven Refactoring Tool for KDM* (DURELLI *et al.*, 2014c).

As seções deste capítulo estão organizadas da seguinte forma: na Seção 6.2, o desenvolvimento da KDM-RE é mostrado; na Seção 6.3, é apresentado o módulo responsável por aplicar refatorações no metamodelo KDM; na Seção 6.4, é apresentado o módulo responsável por instanciar o metamodelo SRM por meio de uma DSL, e também é apresentado como reutilizar programaticamente as refatorações persistidas no metamodelo SRM; na Seção 6.5, é apresentado o módulo responsável por manter a instância do metamodelo KDM sincronizada e consistente após a aplicação de refatorações. Algumas ferramentas relacionadas são comentadas na Seção 6.6. As considerações finais desse capítulo são apresentadas na Seção 6.7.

6.2 Construção da KDM-RE

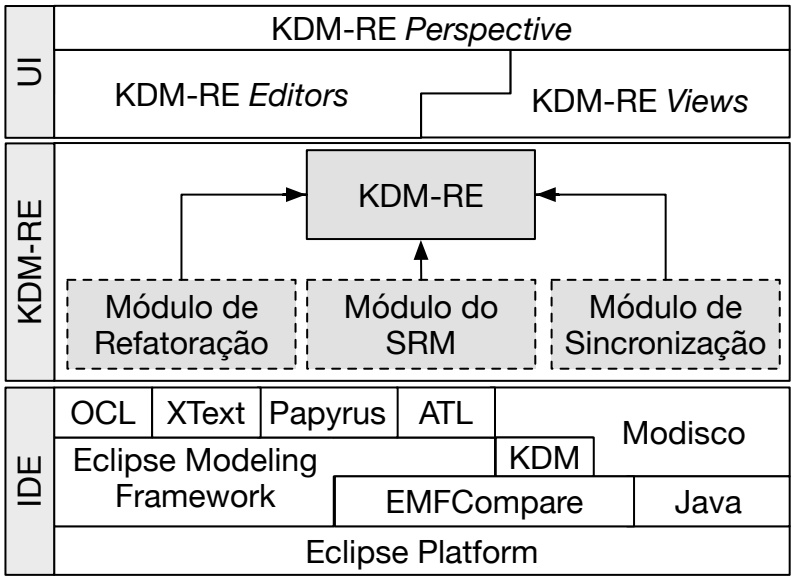
A KDM-RE foi construída de modo a ser utilizada em conjunto com os demais recursos oferecidos pelo ambiente de desenvolvimento Eclipse IDE. Os *plug-ins* da KDM-RE são organizados em módulos de acordo com a sua funcionalidade. Os três principais módulos são:

- **Módulo de Refatoração**, que é responsável pela aplicação de refatorações de forma transparente em instâncias do metamodelo KDM;
- **Módulo do SRM**, que é responsável pela instanciação e reúso do metamodelo SRM;
- **Módulo de Sincronização**, que é responsável por manter consistente e propagar mudanças após a aplicação de refatorações em instâncias do metamodelo KDM.

A Figura 44 apresenta uma visão lógica da arquitetura da ferramenta KDM-RE. Como observado, a KDM-RE contém três camadas: (*i*) IDE, (*ii*) KDM-RE e (*iii*) UI. A primeira camada da KDM-RE agrupa os recursos do Eclipse IDE para a criação dos três principais módulos da ferramenta KDM-RE. EMF, ATL, OCL, MoDisco, Papyrus, XText, EMFCompare e KDM são utilizadas durante a criação do KDM-RE.

Na segunda camada, os módulos da KDM-RE são definidos. O módulo de refatoração utiliza EMF, ATL, OCL e MoDisco para criar um *plug-in* onde o engenheiro de software pode aplicar refatorações em instâncias do metamodelo KDM. O módulo de sincronização utiliza o *framework* EMFCompare e ATL. Por sua vez, o módulo do SRM utiliza EMF e XText. XText é utilizado no módulo do SRM para definir uma DSL para auxiliar a instanciação do metamodelo SRM. A última camada, UI, é responsável pela interface gráfica da ferramenta KDM-RE. Essa camada possui *Wizards*, editores e visões para a aplicação, definição e reutilização

Figura 44 – Arquitetura da Ferramenta KDM-RE.



Fonte: Elaborada pelo autor.

das refatorações no contexto do KDM. Nas seções seguintes, os três principais módulos da ferramenta KDM-RE são apresentados e discutidos.

6.3 Módulo de Refatoração

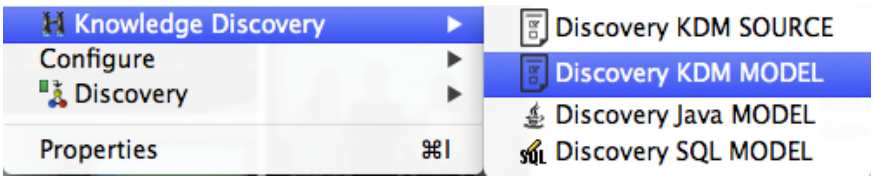
Nesta seção, o módulo de refatoração é apresentado e ele foi implementado para prover suporte às refatorações criadas por meio das diretrizes apresentadas no Capítulo 4. Na literatura, é possível identificar um conjunto de técnicas e linguagens específicas para auxiliar a condução e especificação de transformação de modelos (BIEHL, 2010; MENS; GORP, 2006; ALLILAIRE *et al.*, 2006). Nesta tese, a linguagem de transformação ATL (ATL, 2015; JOUAULT *et al.*, 2008) foi escolhida para definir e aplicar refatorações no metamodelo KDM. Similarmente, as pre- e pós-condições foram implementadas em OCL. KDM-RE programaticamente executa as refatorações implementadas em ATL por meio da ATL EMF *Transformation Virtual Machine*. As pre- e pós-condições são executadas na KDM-RE por meio da API Desden OCL[2]. Essa API facilita a aplicação de OCL em qualquer metamodelo definido em EMF. Como consequência, KDM-RE é capaz de suportar a detecção de violações semânticas estáticas em instâncias do metamodelo KDM.

Como já salientado no Capítulo 4, as refatorações foram definidas para serem executadas no contexto de instâncias do metamodelo KDM. Dessa forma, é necessário, primeiramente, transformar um determinado sistema em instância do metamodelo KDM. Para tal, foi integrada a ferramenta MoDisco na KDM-RE como apresentado na Figura 45. Primeiramente, o engenheiro de software deve clicar com o botão direito em um projeto Java e escolher a opção `Knowledge`

[2] `http://www.dresden-ocl.org/`

`Discovery` e, depois, escolher o menu `Discovery KDM MODEL`. MoDisco irá, então, transformar o código escrito na linguagem de programação Java para uma instância do metamodelo KDM.

Figura 45 – KDM *Discovery*.



Fonte: Elaborada pelo autor.

Após a criação da instância do metamodelo KDM, o engenheiro de software pode aplicar as refatorações. A ferramenta KDM-RE permite que as refatorações sejam aplicadas por meio de duas interfaces gráficas. A primeira é uma extensão do editor gráfico da ferramenta MoDisco (BRUNELIèRE *et al.*, 2014), que representa a instância do metamodelo em formato de árvore. A segunda interface gráfica é uma extensão do editor gráfico da ferramenta Papyrus[3] que é uma ferramenta *Computer-Aided Software Engineering* (CASE). Por meio da segunda interface, o engenheiro de software pode aplicar refatorações utilizando um diagrama de classe UML. Nota-se que nessa segunda interface, de forma transparente, as refatorações são de fato aplicadas em instâncias do KDM, e o diagrama de classe UML é utilizado apenas como uma ponte entre as informações (nome de classe, atributos, métodos, etc.) e as refatorações.

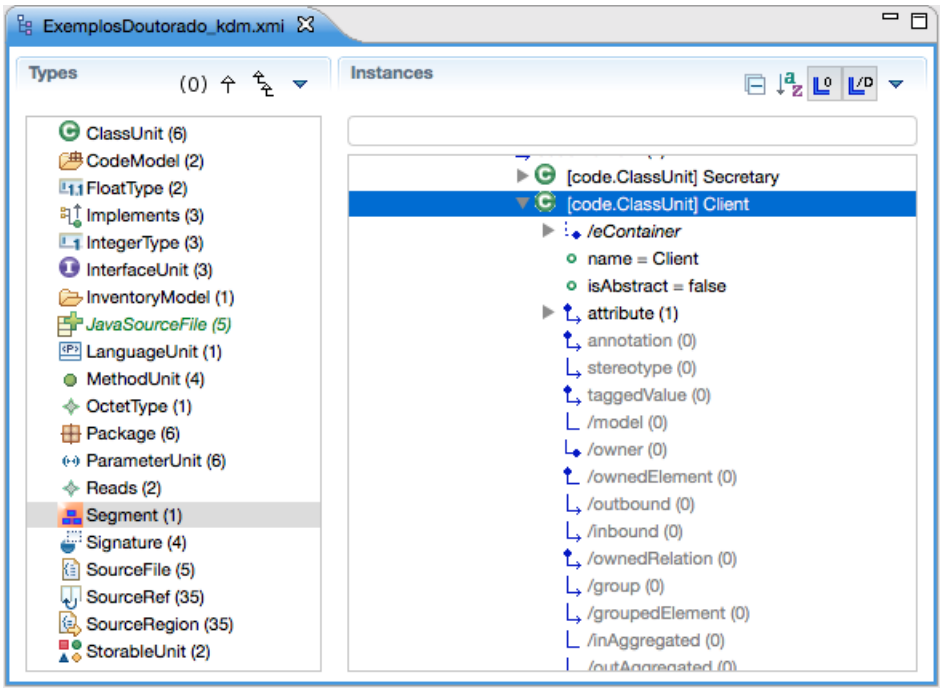
A primeira interface gráfica fornece uma visão de árvore da instância do metamodelo KDM (*model browser*) como apresentado na Figura 46. No lado esquerdo dessa figura, algumas das metaclasses (`ClassUnit`, `MethodUnit` e `StorableUnit`, etc.) instanciadas do metamodelo KDM são apresentadas. O lado direito é onde as refatorações são aplicadas pelo engenheiro de software. Para aplicar as refatorações, o engenheiro de software deve clicar com o botão direito em cima de uma determinada instância de metaclasse, por exemplo, `ClassUnit`, `MethodUnit`, `StorableUnit`, etc., assim, o menu `Refactoring KDM` irá aparecer como ilustrado na Figura 47. Além disso, utilizando-se esse menu, o engenheiro de software pode interagir com a instância do metamodelo KDM e escolher qual refatoração deve ser executada. Após o engenheiro de software clicar no menu `Refactoring KDM` e escolher uma determinada refatoração, o processo será iniciado.

Para cada refatoração, um determinado *RefactoringWizard* é executado. Esse *Wizard* irá guiar o engenheiro de software durante a aplicação da refatoração. Em cada refatoração, a instância do metamodelo KDM deve ser analisada para identificar e obter os metadados das metaclasses que serão afetadas pela refatoração. Esses metadados são utilizados tanto na ATL (refatoração), quanto na OCL (pré- e pós-condições).

Por exemplo, suponha que o engenheiro de software identificou que uma instância da metaclasse `ClassUnit` está realizando o trabalho que deveria ser feito por duas instâncias (ver

[3] `https://eclipse.org/papyrus/`

Figura 46 – MoDisco *Model Browser*.



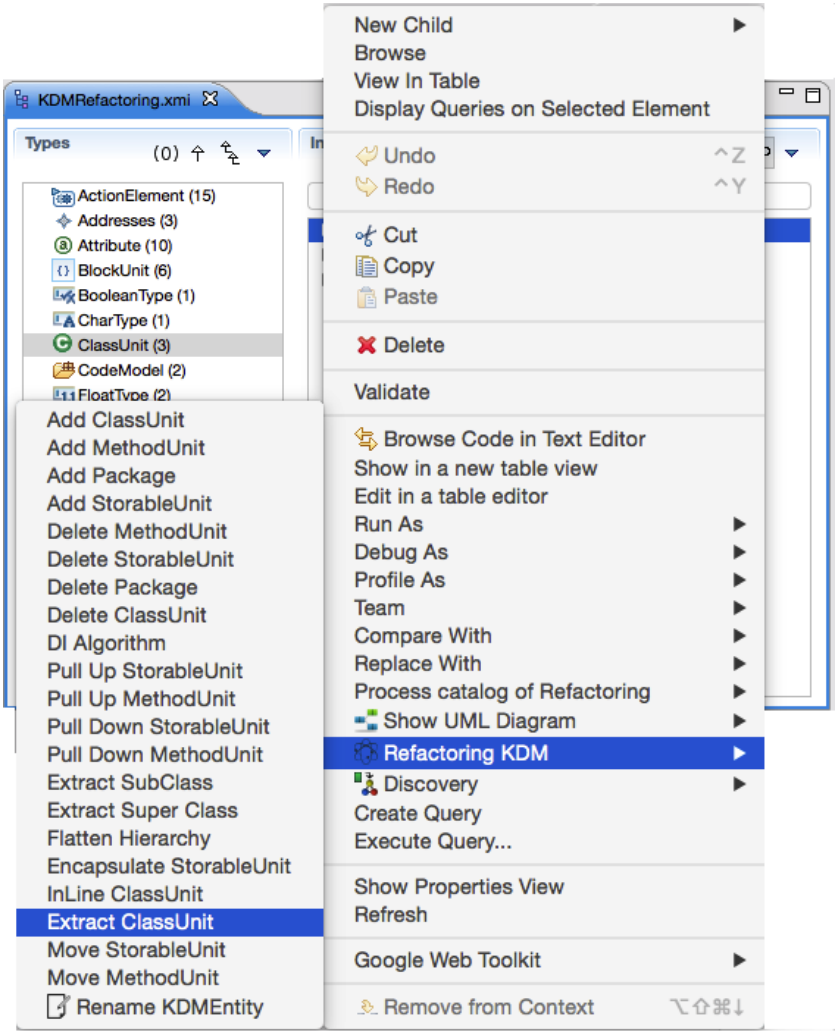
Fonte: Elaborada pelo autor.

Figura 48). Assim, o engenheiro de software deve aplicar a refatoração `Extract ClassUnit`. O primeiro passo é selecionar a instância da metaclasse `ClassUnit` para realizar a refatoração. Em seguida, deve-se selecionar a opção `Extract ClassUnit` no menu `Refactoring KDM`. Automaticamente, KDM-RE irá criar um *Wizard* para auxiliar o engenheiro de software, como ilustrado na Figura 48. Utilizando esse *Wizard*, o engenheiro de software define o nome da nova instância da metaclasse `ClassUnit`. Além disso, esse *Wizard* permite visualizar todas as instâncias das metaclasses `StorableUnit` e `MethodUnit` que serão extraídas para a nova instância a ser criada. O *Wizard* também permite especificar se instâncias da metaclasse `MethodUnit` devem ser criadas para representar os métodos assessores (*getters* e *setters*).

Uma característica importante da KDM-RE é a opção de visualizar previamente o resultado da refatoração. Assim, caso o engenheiro de software almejar visualizar o efeito da refatoração antes de efetivamente realizá-la, ele poderá selecionar o botão `Preview`. Após clicar no botão `Preview`, uma visão de comparação será criada como apresentado na Figura 49. Essa visão de comparação contém duas principais partes. A parte superior representa quais instâncias foram deletadas, movidas e adicionadas de forma textual. Na parte inferior, é possível visualizar graficamente a diferença entre as duas instâncias do metamodelo KDM, ou seja, a instância não refatorada (original) e a instância refatorada. O lado direito representa a instância do metamodelo KDM após a aplicação da refatoração `Extract ClassUnit` e o lado esquerdo representa a instância do metamodelo KDM antes da aplicação da refatoração.

Ao focar a Figura 49, vê-se a instância direita do metamodelo KDM (instância refatorada)

Figura 47 – KDM-RE *Refactoring Browser*.



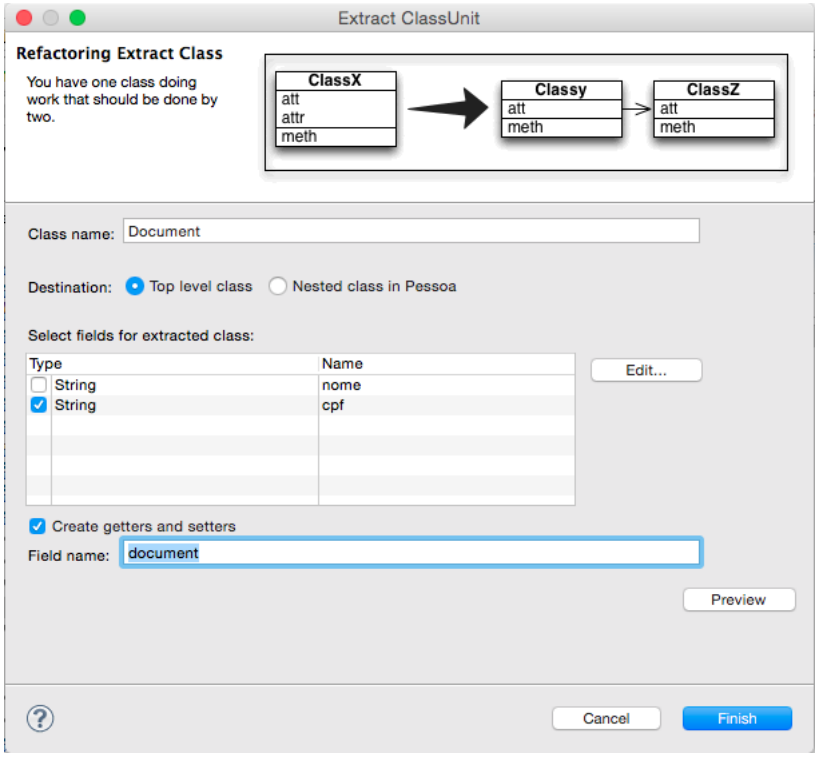
Fonte: Elaborada pelo autor.

uma nova instância da metaclasse `ClassUnit` chamada `Document` foi criada e uma instância da metaclasse `StorableUnit` (“CPF”) foi movida para a nova `ClassUnit Document`. Instâncias da metaclasse `MethodUnit` também foram criadas para representar os métodos assessores. A instância da metaclasse denominada `Pessoa` agora possui uma instância da metaclasse `StorableUnit`, denominada `document` que representa um link entre as duas instâncias de `ClassUnit`.

Após especificar todas as entradas necessárias e visualizar o efeito que a refatoração irá resultar na instância do metamodelo KDM, o engenheiro de software deve clicar no botão *Finish*. A KDM-RE executa a API Dresden OCL para verificar as pré-condições. Caso as pré-condições forem satisfeitas, a refatoração `Extract ClassUnit` é executada efetivamente. A execução da refatoração é totalmente realizada com base na ATL. As entradas informadas pelo engenheiro de software no *Wizard* são enviadas para ATLs pré-definidas e, em seguida, são executadas programaticamente pela KDM-RE por meio da ATL EMF *Transformation Virtual Machine*[4].

[4] `https://wiki.eclipse.org/ATL/EMFTVM`

Figura 48 – `Extract ClassUnit` *Wizard*.



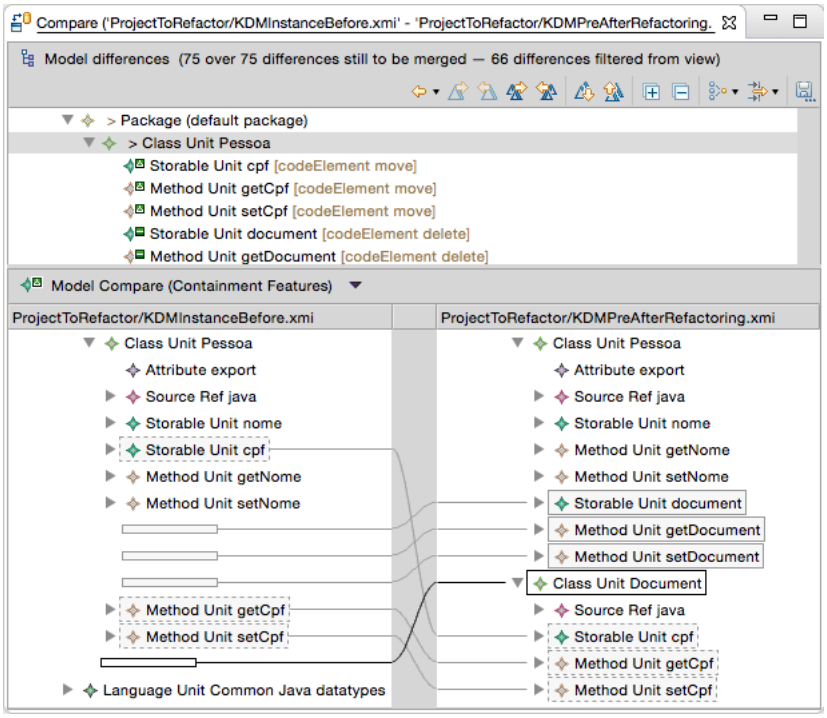
Fonte: Elaborada pelo autor.

Posteriormente, a KDM-RE executa a API Dresden OCL para verificar as pós-condições.

Embora a primeira interface gráfica seja útil para aplicar refatorações diretamente em instâncias do metamodelo KDM, ela não é intuitiva. Dessa forma, a segunda interface gráfica almeja deixar as refatorações mais fáceis e intuitivas de serem aplicadas. Por exemplo, embora Fowler (1999) tenha criado um catálogo de refatorações para ser utilizado em código-fonte, mais de 60% das refatorações (44 de 72) são ilustradas utilizando modelos, mais especificamente diagramas de classes da UML. Além disso, Zhang, Lin e Gray (2005), Boger, Sturm e Fragemann (2003) afirmam que algumas refatorações (por exemplo, `Extract Method`) são mais naturais quando executadas diretamente no código-fonte e outras refatorações, como: `Rename Class`, `Pull Up Method`, `Push Down Method`, etc., podem ser aplicadas tanto em código-fonte, quanto em modelos; já as refatorações que lidam com herança, tais como `Extract Class`, `Extract Interface`, `Replace Inheritance with Delegation`, são mais intuitivas quando aplicadas diretamente em nível de modelo, tais como o diagrama de classe da UML.

Diante disso, a KDM-RE foi integrada com a ferramenta CASE Papyrus para utilizar o diagrama de classe da UML. Para utilizar essa segunda interface, primeiramente, é necessário transformar a instância do metamodelo KDM para uma instância da UML. Assim, as refatorações definidas na KDM-RE podem ser aplicadas diretamente em diagramas da UML; por exemplo, pode-se aplicar refatorações por meio do diagrama de classe. É importante observar que, embora as refatorações sejam aplicadas graficamente por meio do diagrama de classe da UML, todas

Figura 49 – Prévia do resultado da refatoração `Extract ClassUnit`.



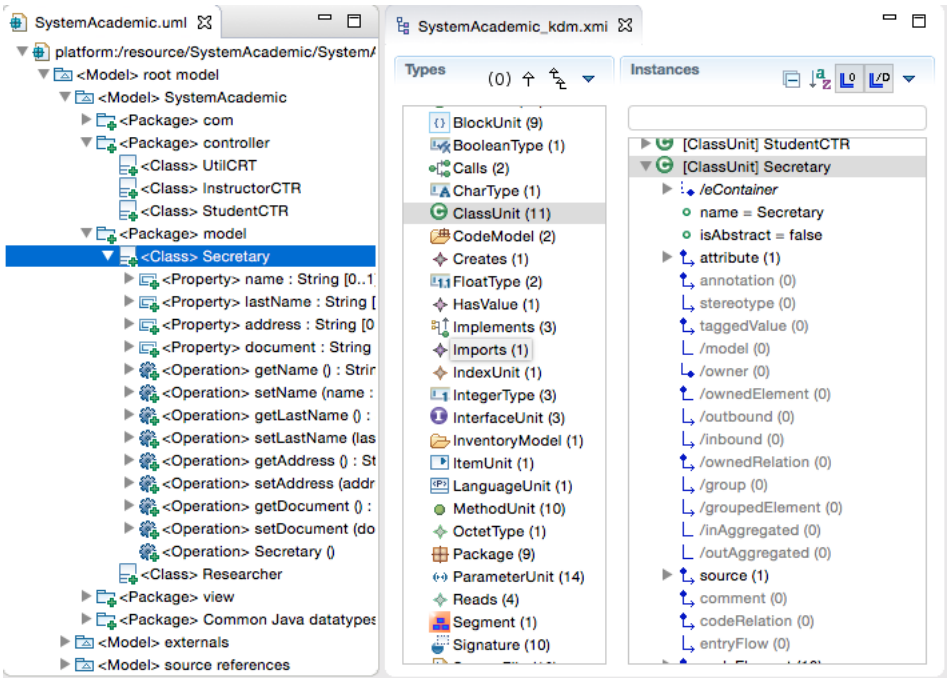
Fonte: Elaborada pelo autor.

as refatorações (transformações) são aplicadas transparentemente na instância do metamodelo KDM e não na instância da UML - apenas informações são extraídas do diagrama de classe da UML e são enviados como entrada para as refatorações pré-definidas em ATL.

Os passos para utilizar a segunda interface são quase os mesmos da primeira interface. Porém, um passo a mais se faz necessário para utilizar a segunda interface. Deve-se gerar uma instância do metamodelo UML tendo como base uma instância do metamodelo KDM. A geração da instância do metamodelo UML é totalmente apoiada por um *plug-in* denominado `DiscoverUmlModelFromKdmModel` do MoDisco. Esse *plug-in* utiliza regras ATL para criar uma instância do metamodelo UML, tendo como base outra instância do metamodelo KDM. Por exemplo, na Figura 50, são apresentadas duas instâncias dos metamodelos UML (lado esquerdo) e KDM (lado direito) após a transformação, respectivamente.

Após a criação da instância do metamodelo UML, o próximo passo é utilizar a ferramenta CASE Papyrus para exibir a nova instância do metamodelo UML por meio do diagrama de classe, como apresentado na Figura 51. Por meio desse diagrama, a KDM-RE permite que o engenheiro de software realize refatorações. Por exemplo, na Figura 52 é ilustrado que, primeiramente, o engenheiro de software deve clicar com o botão direito em cima do elemento que almeja refatorar (nesse caso a classe `Secretary`), escolher a opção `Refactoring Model` e, depois, decidir qual refatoração aplicar (nesse exemplo: `Extract ClassUnit`). Em seguida, um *RefactoringWizard* similar ao apresentado na Figura 48 é executado e ele também irá guiar o engenheiro de software durante a aplicação da refatoração. Da mesma forma como na primeira

Figura 50 – Instância UML gerada a partir do KDM.



Fonte: Elaborada pelo autor.

interface, na segunda interface o engenheiro de software pode solicitar também a realização de uma prévia da refatoração. O resultado dessa solicitação será uma interface similar à apresentada na Figura 49.

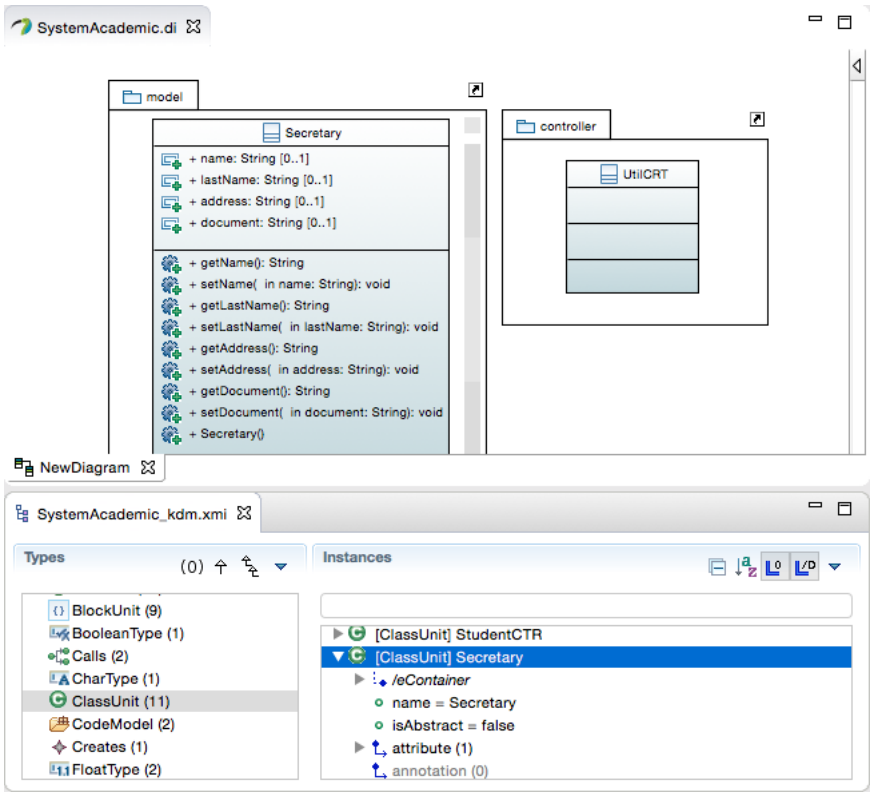
As informações fornecidas pelo engenheiro de software no *Wizard* são enviadas para ATLs pré-definidas e, em seguida, são executadas programaticamente pela KDM-RE por meio da ATL EMF *Transformation Virtual Machine*. O resultado da refatoração altera a instância do metamodelo KDM e é replicado automaticamente no diagrama de classe da UML; portanto, o engenheiro de software pode visualizar graficamente no diagrama de classe UML o resultado da refatoração.

6.4 Módulo do SRM

Nessa seção, é apresentado um módulo desenvolvido para fornecer suporte ao metamodelo SRM apresentado no Capítulo 5 (Figura 30). Esse módulo implementa o metamodelo SRM utilizando EMF por meio do meta-metamodelo Ecore. Nesse módulo, também foi definida uma linguagem específica de domínio (DSL) para facilitar a instanciação do metamodelo SRM. A gramática dessa DSL é apresentada no Capítulo 5, mais detalhadamente nos Códigos-fontes 32, 33, 34, 35 e 37.

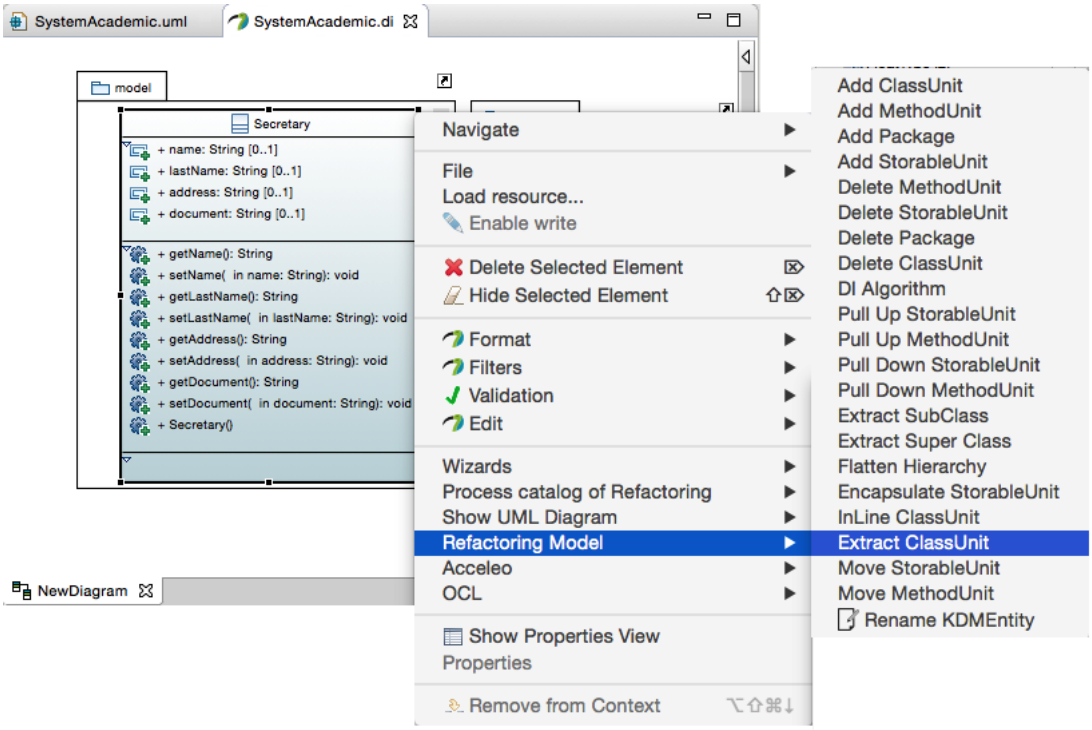
Esse módulo fornece uma forma de compartilhar e reutilizar refatorações por meio de instâncias do metamodelo SRM. Para permitir o reúso e o compartilhamento de refatorações,

Figura 51 – Diagrama de Classe da UML gerada a partir do KDM.



Fonte: Elaborada pelo autor.

Figura 52 – Refatorações por meio do Diagrama de Classe.

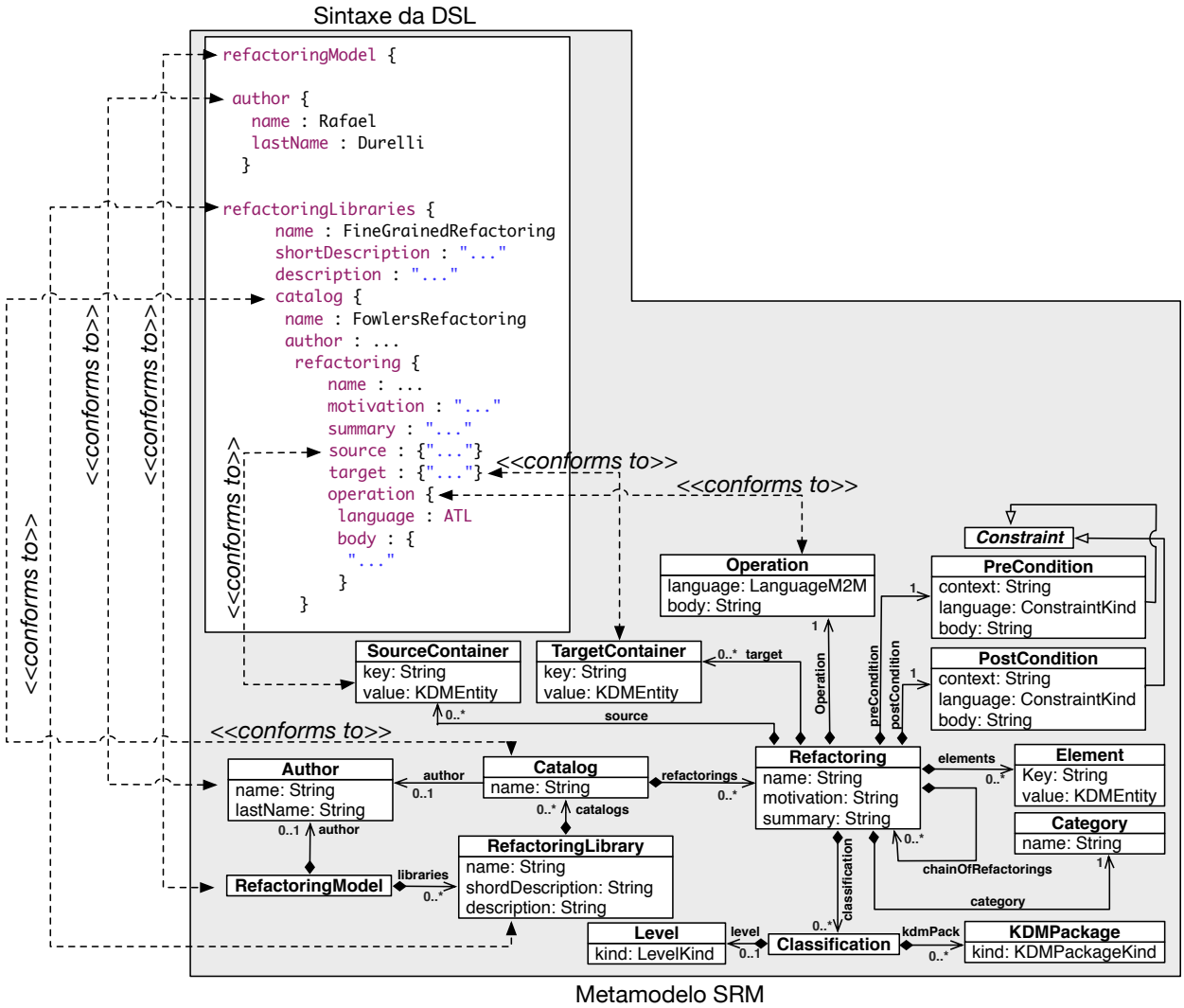


Fonte: Elaborada pelo autor.

um repositório remoto foi criado. Esse repositório remoto é dedicado para executar solicitações RESTful. Instâncias do metamodelo SRM são enviadas e recebidas por meio da API RESTful. Isso é possível, pois as instâncias do SRM são arquivos persistidos no formato XMI. Java Persistence API (JPA) e o banco de dados MySQL foram utilizados para realizar as persistências das instâncias do metamodelo SRM.

Por meio das gramáticas apresentadas nos Códigos-fontes 32, 33, 34, 35, 36 e 37, Xtext gera um editor textual para o ambiente de desenvolvimento Eclipse IDE. Esse editor textual provê *highlighting* de sintaxe, *autocomplete* de código e navegação de código. Na parte superior, à esquerda da Figura 53, é possível visualizar trechos da DSL resultante. Além disso, essa figura ilustra o relacionamento entre a DSL criada e as metaclasses do metamodelo SRM, ou seja, representa que a sintaxe da DSL está em conformidade com as metaclasses do SRM.

Figura 53 – DSL para auxiliar a instanciação do SRM.

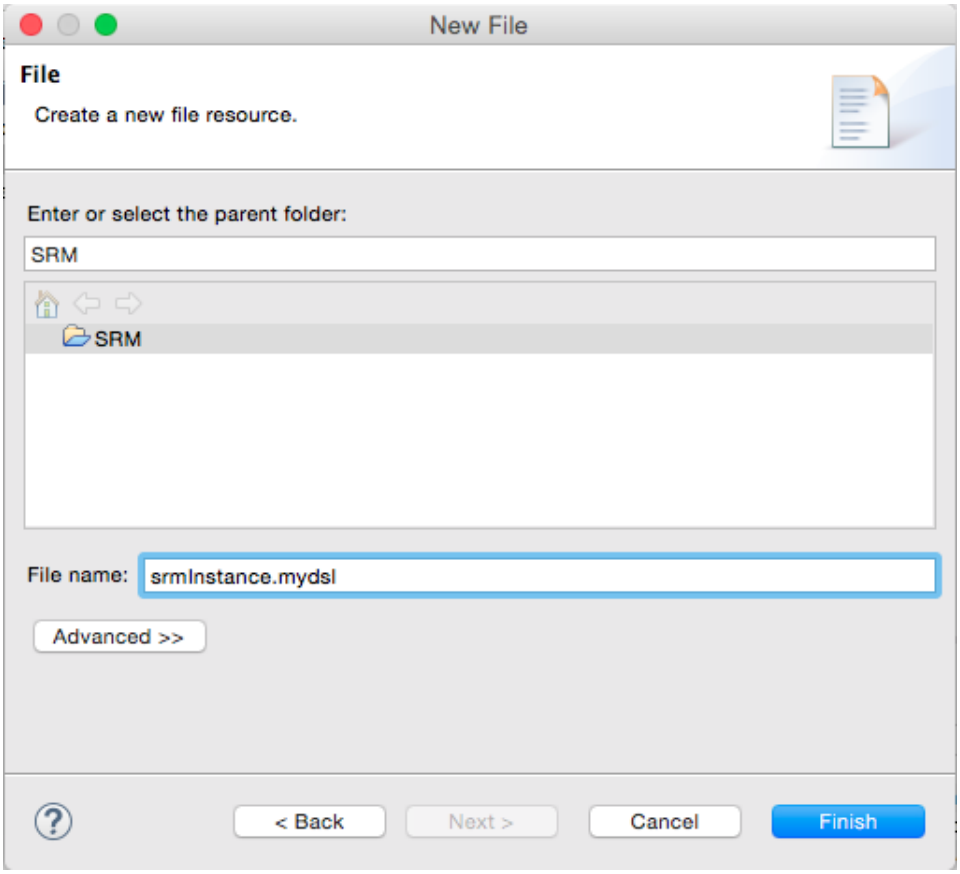


Fonte: Elaborada pelo autor.

Cada trecho de código da DSL representa uma instância de uma metaclasse do meta-

modelo SRM, ou seja, cada declaração da DSL está em conformidade com uma metaclasse do SRM. Por exemplo, a palavra-chave `refactoringModel` `author` entre "{" e "}" representa a instanciação da metaclasse `Author` do SRM que possui dois meta-atributos: `name` e `lastName`. A DSL criada para auxiliar a instanciação do SRM foi desenvolvida utilizando Xtext[5]. Xtext é um *framework* do Eclipse[6] que facilita a definição de gramática[7] com a utilização de um metamodelo que foi definido utilizando EMF. Xtext tem como principal objetivo automatizar e agilizar o processo de desenvolvimento de DSLs. Além disso, a sintaxe da DSL segue as terminologias e conceitos definidos no metamodelo SRM para facilitar a utilização da DSL e o entendimento do metamodelo SRM.

Figura 54 – Instanciando a DSL.



Fonte: Elaborada pelo autor.

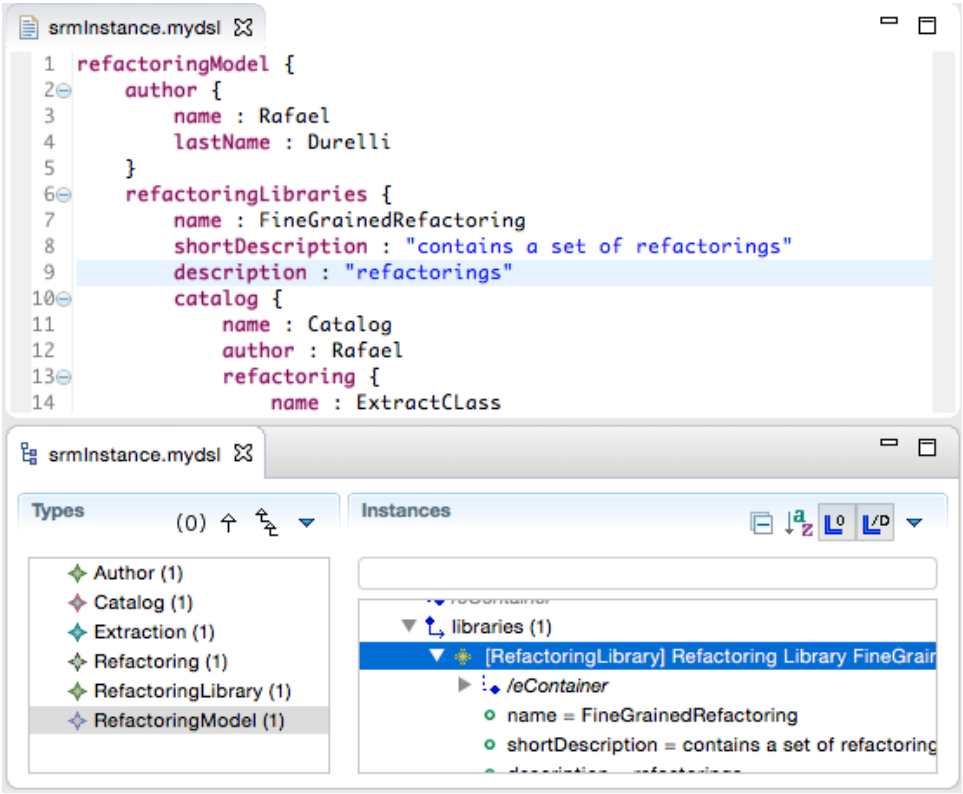
Para utilizar a DSL, primeiramente, o engenheiro de modernização precisa criar um arquivo com a extensão ".mydsl". A KDM-RE fornece um *wizard* para criar esse arquivo, como apresentado na Figura 54. Por meio desse *wizard*, o engenheiro de modernização deve especificar um nome para o arquivo e também deve especificar a extensão do arquivo como ".mydsl". É

5 `https://www.eclipse.org/Xtext/`

6 `https://www.eclipse.org`

7 Gramáticas representam a definição formal de uma sintaxe textual concreta. Consistem em um conjunto de regras de produção para definir como o *textual input* (ou seja, sentenças) é representado. Basicamente, as regras de produção podem ser representadas utilizando *Backus–Naur Form* (BNF), por exemplo, $S ::= P1 \ldots Pn$, essa gramática define um símbolo S por um conjunto de expressões $P1 \ldots Pn$.

Figura 55 – Editor Textual para instanciar o metamodelo SRM.

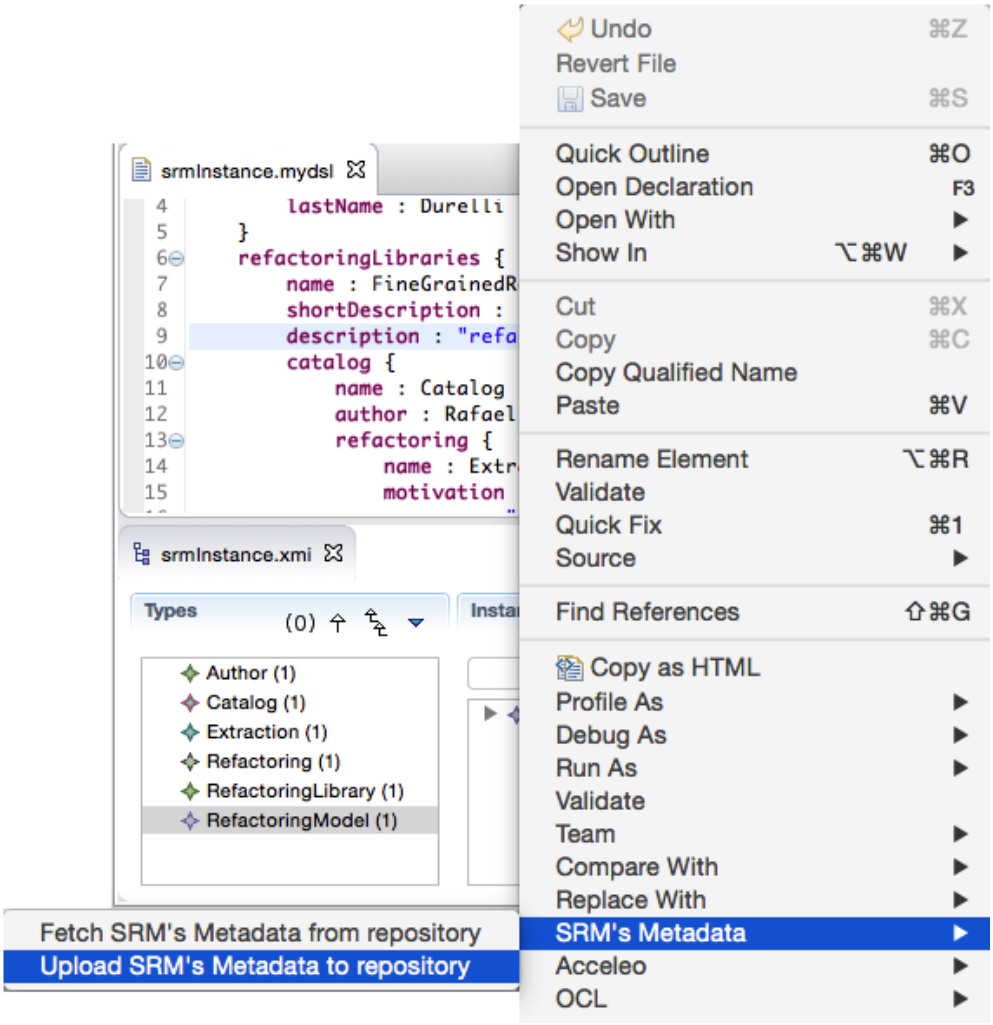


Fonte: Elaborada pelo autor.

importante que o engenheiro de modernização especifique a extensão correta do arquivo, ou seja, ".mydsl", a qual é utilizada para o módulo do SRM identificar que o arquivo criado é, na verdade, a DSL definida na gramática apresentada nos Códigos-fontes 32- 37. No exemplo apresentado na Figura 54, o arquivo foi definido como `srmInstance.mydsl`. Dessa forma, a KDM-RE automaticamente irá criar um editor para auxiliar o engenheiro de modernização a especificar uma refatoração utilizando as sintaxes da DSL. Após a criação do arquivo `srmInstance.mydsl`, o engenheiro de modernização deve começar a especificar uma determinada refatoração, como apresentado na Figura 55.

Adicionalmente, a KDM-RE fornece uma maneira de armazenar e compartilhar instâncias do metamodelo SRM. O principal objetivo é fazer com que refatorações definidas utilizando o metamodelo SRM sejam reutilizadas em projetos que utilizam o metamodelo KDM. Assim, após definir uma instância do metamodelo SRM utilizando a DSL, por meio do editor apresentado na Figura 55, o engenheiro de modernização deve enviar os metadados de uma determinada instância do SRM para um repositório remoto. Esse processo é realizado por meio de um menu denominado `Upload SRM's Metadata to repository`, e, para interagir com esse menu, o engenheiro deve clicar com o botão direito no editor textual da DSL e escolher `SRM's Metadata`, como apresentado na Figura 56. Transparentemente, a KDM-RE irá converter a sintaxe e a semântica da DSL em um arquivo XMI, como ilustrado no Código-fonte 46. Note que nesse

Figura 56 – Menu para enviar instâncias do metamodelo SRM.



Fonte: Elaborada pelo autor.

código-fonte os códigos em ATL e OCL (que representam a refatoração e as pré- e pós-condições) foram omitidos para facilitar o entendimento do arquivo XMI.

Cada marcação apresentada nesse XMI representa uma metaclasse do metamodelo SRM. Por exemplo, "**<refactoringModel>**", "**<libraries>**", "**<catalogs>**" e "**<refactorings>**" estão em conformidade com as metaclasses `RefactoringModel`, `RefactoringLibrary`, `Catalog` e `Refactoring` do SRM, respectivamente. Adicionalmente, cada marcação contém atributos que representam os meta-atributos de cada metaclasse do SRM.

Código-fonte 46: Arquivo XMI representando a instância do SRM.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <refactoringModel:RefactoringModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/
    XMI" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:
    refactoringModel="http://refactoringModel/1.0">
3   <libraries name="FineGrainedRefactoring" shortDescription="contains a set of
    refactorings" description="refactorings">
4     <catalogs name="Catalog" author="//@author">
5       <refactorings name="ExtractCLass" motivation="Motivation" summary="Summary">
```

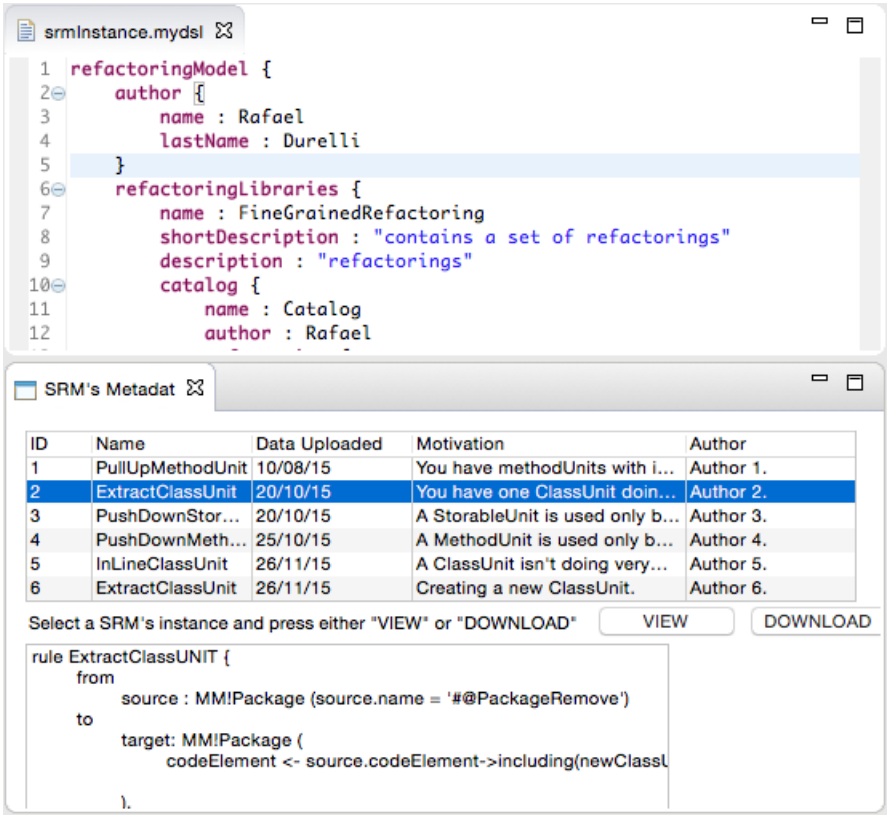
```
6          <source key="Class" value=//@ClassUnit/>
7          <target key="Class" value=//@ClassUnit/>
8          <elements key="Class" value=//@ClassUnit key "Attribute" value=//
       @StorableUnit/>
9          <preCondition context="TO BE DEFINED" language="OCL" body="TO BE DEFINED"/>
10         <postCondition context="TO BE DEFINED" language="OCL" body="TO BE DEFINED"/>
11         <operation body="TO BE DEFINED"/>
12       </refactorings>
13     </catalogs>
14   </libraries>
15   <author name="Rafael" lastName="Durelli"/>
16 </refactoringModel:RefactoringModel>
```

Posteriormente, o arquivo XMI é lido, enviado e armazenado em um repositório remoto para ser manipulado. A KDM-RE também fornece uma maneira de visualizar todas as instâncias do SRM disponíveis no repositório remoto. Essa opção é realizada por meio do menu `SRM's Metadata` e, em seguida, `Fetch SRM's Metadata from repository` (ver Figura 56). Após clicar no menu `Fetch SRM's Metadata from repository`, a Figura 57 é apresentada, a qual fornece a visualização de todas as instâncias do metamodelo SRM disponíveis para serem reutilizadas. Nessa figura pode ser observado que existem seis instâncias do metamodelo SRM: `PullUpMethodUnit`, `ExtractClassUnit`, `PushDownStorableUnit`, `PushDownMethodUnit`, `InLineClassUnit` e outra `ExtractClassUnit`. KDM-RE permite que o engenheiro de software visualize a refatoração para cada instância do metamodelo SRM. Por exemplo, se o engenheiro almejar visualizar a refatoração escrita em ATL, ele deverá, então, selecionar uma determinada instância do SRM e clicar no botão `VIEW`. Assim, a refatoração escrita em ATL será apresentada em uma área de texto, como ilustrado na parte inferior da Figura 57. Após escolher uma determinada instância do metamodelo SRM, o botão `DOWNLOAD` deve ser clicado para realizar a transferência da instância do metamodelo SRM e reutilizá-la em seu projeto.

6.5 Módulo de Sincronização

Usualmente, durante o desenvolvimento e modernização de software seguindo as diretrizes e passos da abordagem MDE, o software geralmente é modelado e representado utilizando diferentes instâncias de metamodelos para representar as visões e todos os artefatos de um sistema. Em outras palavras, geralmente existem metamodelos para abstrair e representar todos os artefatos do sistema, tais como: metamodelos para o código-fonte, metamodelos para representar e abstrair o banco de dados, metamodelos para representar e abstrair a arquitetura do sistema, etc. Como apresentado no Capítulo 2, o metamodelo KDM é capaz de agrupar todos esses artefatos em um único metamodelo, sendo assim, possível de representar diferentes visões/artefatos e seus relacionamentos de um determinado sistema em uma única instância do metamodelo KDM. Porém, conforme o engenheiro de software aplica um conjunto de refatorações em uma determinada instância do metamodelo KDM mudanças são realizadas. Geralmente, tais mudanças podem necessitar que subsequentes alterações sejam realizadas para que outras visões/artefatos

Figura 57 – Visão das instâncias do metamodelo SRM disponíveis no repositório.



Fonte: Elaborada pelo autor.

do metamodelo KDM fiquem consistentes e sincronizados.

Uma premissa fundamental é manter todas as visões/artefatos do metamodelo KDM sincronizadas durante todo o processo de modernização do software. Dessa forma, quando as visões/artefatos representadas em nível de modelos forem alteradas, é de extrema importância realizar um conjunto de propagação de mudança por todas as visões/artefatos para mantê-las atualizadas e sincronizadas, espelhando, assim, a alteração em todas as visões/artefatos do software. Usualmente, como apresentado nos Capítulo 2, Seção 2.3 e Capítulo 4, essas alterações podem ser realizadas por meio de refatorações, as quais são atividades centrais durante o processo de modernização. Porém, quando um software é representado utilizando diferentes abstrações e visões em nível de modelos, um acidente comum que pode ocorrer durante a atividade de refatoração é a dessincronização dessas visões, fazendo com que as visões/artefatos que representam o sistema fiquem inconsistentes após a atividade de refatoração. Uma forma de resolver esse problema é aplicar técnicas de propagação de mudança, cujo objetivo é identificar e atualizar todas as instâncias dependentes dos elementos que foram refatorados. Diante desse contexto, a ferramenta KDM-RE possui um módulo de sincronização para realizar a propagação de mudança e preservação de comportamento após a aplicação de refatorações em instâncias do metamodelo KDM. Utilizando esse módulo, engenheiros de software podem se concentrar apenas na aplicação das refatorações ou reutilizá-las por meio do metamodelo SRM (ver Capítulo 5),

sem se preocuparem com a propagação de mudanças para outras visões/artefatos do metamodelo KDM.

É importante destacar que o fluxo desse módulo de sincronização inicia-se considerando que o engenheiro de software almeja aplicar um conjunto de refatorações em um sistema que já esteja representado por meio de uma instância do metamodelo KDM. Essa instância deve ser a mais completa possível, ou seja, representa todas as visões/artefatos do sistema, desde o código-fonte até os elementos arquiteturais do sistema[8]. Após o engenheiro de software aplicar uma determinada refatoração, o módulo de sincronização, o qual contém três principais passos, efetivamente será iniciado. De forma resumida, os três passos do módulo da seguinte são descritos a seguir.

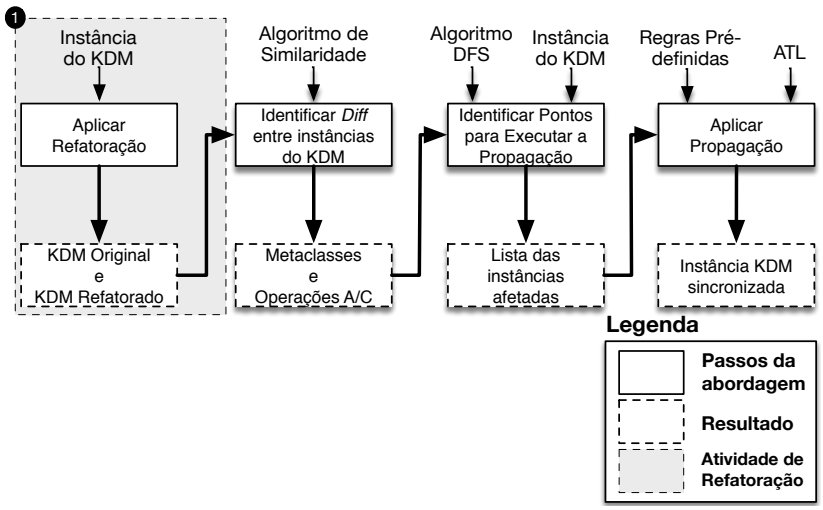
O primeiro passo realiza uma comparação (do inglês - *diff*) entre a instância refatorada do metamodelo KDM com a instância do metamodelo KDM original, ou seja, a instância do metamodelo KDM antes do engenheiro de software aplicar a refatoração. Como resultado, esse passo cria uma lista que contém todas as instâncias das metaclasses do KDM que sofreram uma modificação durante a refatoração, quando comparado com a instância do KDM original. Em seguida, o segundo passo utiliza como entrada a lista gerada para ser utilizada como parâmetro para um algoritmo de mineração e identificação de dependências. Esse algoritmo visa identificar todas as instâncias das metaclasses do KDM que possuem dependência com as metaclasses refatoradas. Como resultado, esse algoritmo também cria uma lista, a qual é utilizada no terceiro passo. O terceiro passo utiliza a lista criada pelo algoritmo para realizar um conjunto de transformações em nível de modelo. Tais transformações foram pré-definidas e representam as propagações de mudanças por todas as visões do KDM. É importante destacar que o módulo de sincronização da KDM-RE foi implementado com a preocupação de ser uma forma genérica e desacoplada, assim, pode ser aplicado em um grande conjunto de refatorações, fazendo com que o engenheiro de software não tenha que se preocupar com a propagação de mudança para outras visões/artefatos do KDM.

Como já mencionado, um problema crítico durante a modernização de software diz respeito à propagação de mudança, por exemplo, dado um conjunto de refatorações que são aplicadas durante a modernização de software, é importante identificar quais são as mudanças que precisam ser realizadas para manter a consistência e sincronia de todos os artefatos do sistema. Dessa forma, propagação de mudança é uma técnica de extrema importância durante a elaboração de processo de modernização de software. O engenheiro de software tem que ter a certeza de que a refatoração foi corretamente propagada e que o software não possui nenhuma inconsistência. Embora muitas abordagens de propagações de mudanças possam ser identificadas na literatura, a propagação de mudanças ainda é um desafio durante a manutenção e modernização de software (MENS, 2008). Além disso, a maioria das abordagens de propagação de

[8] Na verdade, é importante que mais de uma visão/artefato seja representada utilizando o metamodelo KDM, seja código-fonte, banco de dados, elementos estruturais, etc.

mudanças existentes tem como principal artefato o código-fonte (RAJLICH, 2009; DEURSEN; VISSER; WARMER, 2007). Similarmente, também é possível identificar algumas abordagens que dão suporte para a propagação de mudança para o metamodelo UML (EGYED; LETIER; FINKELSTEIN, 2008; LIU; EASTERBROOK; MYLOPOULOS, 2002; BRIAND *et al.*, 2006). Porém, até o momento nenhuma iniciativa foi criada para o metamodelo KDM. Para suprir essa limitação, a KDM-RE possui um módulo de sincronização, o qual visa propagar mudanças por todas as visões/artefatos do metamodelo KDM para mantê-lo atualizado e sincronizado após a aplicação de uma refatoração. A intenção é criar um apoio computacional que mantenha uma determinada instância do metamodelo KDM consistente e sincronizada entre todas as visões/artefatos do metamodelo KDM após a aplicação de uma determinada refatoração.

Figura 58 – Visão Geral do Módulo de Sincronização.



Fonte: Elaborada pelo autor.

Na Figura 58, é apresentada uma visão geral do módulo de sincronização, o qual contempla três principais passos. Antes dos passos do módulo se iniciarem, uma atividade de refatoração deve ser realizada. Essa atividade está representada na caixa cinza da Figura 58 ❶ e é apoiada pelo módulo de refatoração apresentado na Seção 6.3. A atividade de refatoração está fora do escopo do módulo de sincronização, assim, é de responsabilidade do engenheiro de software aplicar e/ou reutilizar refatoração para o metamodelo KDM e executa-lá em uma instância do metamodelo KDM. A única restrição do módulo de sincronização da KDM-RE é que duas versões da instância do metamodelo KDM sejam utilizadas como entrada - uma versão que represente a instância do metamodelo KDM antes da aplicação das refatorações (“instância original”) e outra versão que represente uma instância do metamodelo KDM após a aplicação de *n* refatorações (“instância refatorada”). Note que uma “instância original” do metamodelo KDM é aquela que ainda não foi refatorada, também denominada de “KDM esquerdo”, e, similarmente, a “instância refatorada” do metamodelo KDM pode ser entendida como “KDM direito”.

Após a aplicação de um conjunto de refatorações, o primeiro passo do módulo de

sincronização é iniciado. Nesse passo, uma comparação (*diff*) entre a instância original e a instância refatorada é realizada. Como resultado desse passo uma lista é criada, contendo todas as instâncias das metaclasses do KDM que sofreram alguma modificação durante a refatoração quando comparada com a instância do KDM original. Além disso, essa lista também especifica qual(is) foi(ram) a(s) modificação(ões) realizada(s). Por exemplo, se na versão refatorada ("KDM direito") uma nova instância da metaclasse `ClassUnit` foi adicionada, a lista irá conter duas importantes informações: (*i*) a instância da metaclasse `ClassUnit` e (*ii*) qual operação foi realizada nesse exemplo `add ClassUnit`. Essas duas informações são importantes para identificar o que foi alterado e qual operação foi realizada (nesse caso, uma `ClassUnit` foi adicionada); assim, é possível identificar quais propagações devem ser realizadas nas outras visões/pacotes do metamodelo KDM.

Em seguida, o segundo passo do módulo de sincronização é iniciado, identificando todas as metaclasses que precisam ser sincronizadas/atualizadas após a aplicação da refatoração. Utiliza-se, aqui, o algoritmo de busca em profundidade (do inglês - *Depth-First Search* (DFS)). O algoritmo DFS foi alterado para utilizar os seguintes parâmetros como entrada: (*i*) a lista criada no primeiro passo e (*ii*) a instância refatorada do KDM ("KDM direito"). Utilizando a instância refatorada, o algoritmo DFS identifica e cria uma lista que contém todas as metaclasses que possuem dependência com as metaclasses que efetivamente foram refatoradas.

Posteriormente, o último passo pode ser iniciado para realizar a propagação de mudanças na instância do KDM. Como entrada, esse passo utiliza todas as metaclasses que possuem dependência com as metaclasses que foram refatoradas (lista criada no passo anterior). As propagações de mudanças são um conjunto de regras pré-definidas e realizadas de acordo com a instância alterada (`Package`, `ClassUnit`, `MethodUnit`, `StorableUnit`, etc.) e sua operação atômica (`add`, `delete` e `change`). Após o término desse último passo, todas as visões/artefatos da instância do KDM estão sincronizadas e consistentes.

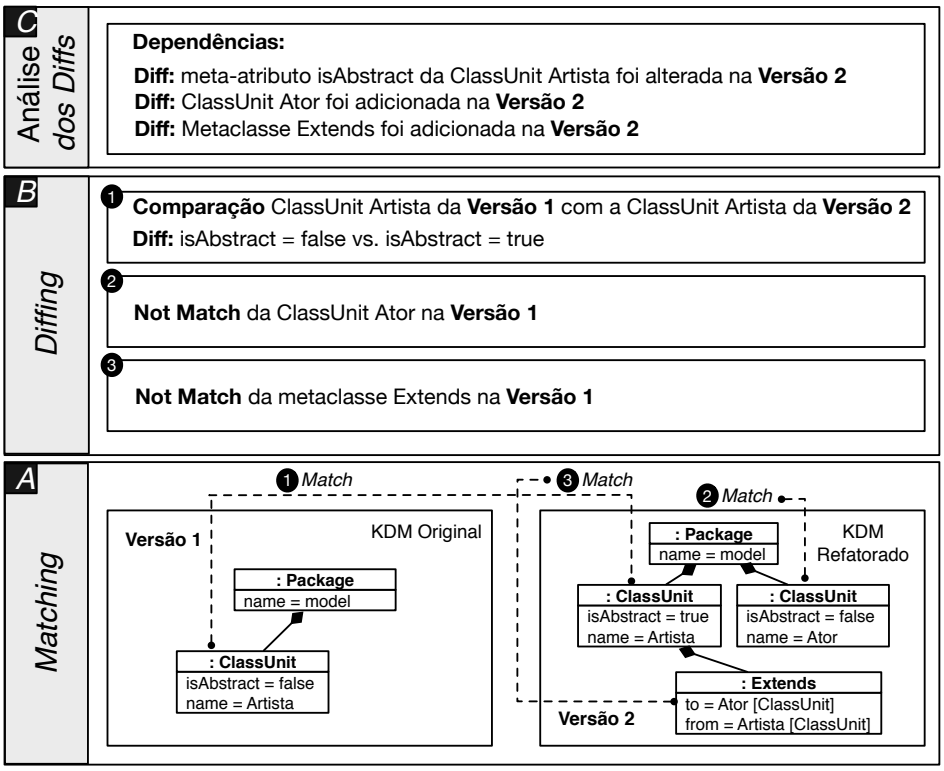
Para auxiliar a elaboração do primeiro passo do módulo de sincronização, o *framework* EMFCompare[9] foi estendido para comparar instâncias do metamodelo KDM. O segundo passo é apoiado por um motor de busca, cuja parte central é o algoritmo DFS juntamente com um conjunto de expressões definidas em linguagem de buscas que são executadas em uma instância do metamodelo KDM para obter todos os pacotes do KDM. O último passo do módulo de sincronização é apoiado por um motor de propagação, o qual utiliza um conjunto de regras pré-definidas e implementadas em ATL para executar as propagações. Todas as propagações foram definidas com base nas instâncias das metaclasses alteradas juntamente com as operações atômicas (`add`, `delete` e `change`), apresentadas no Capítulo 4. Maiores detalhes sobre cada passo do módulo de sincronização são apresentados a seguir.

[9] https://www.eclipse.org/emf/compare/

6.5.1 Identificar Diff entre Instâncias do Metamodelo KDM

Nessa seção, o primeiro passo do módulo de sincronização é apresentado e discutido. Como já salientado, o primeiro passo é apoiado pelo *framework* EMFCompare, o qual foi escolhido, pois pode ser facilmente adaptado e estendido, além de implementar um algoritmo de similaridade de instâncias de metamodelo eficiente. A fim de entender melhor como o primeiro passo do módulo de sincronização funciona, deve-se considerar os seguintes três subpassos: (*i*) *Matching*, (*ii*) *Diffing* e (*iii*) Análise dos *Diffs*, como apresentado na Figura 59.

Figura 59 – Visão Geral da Execução do Módulo de Sincronização.



Fonte: Elaborada pelo autor.

Com pode ser observado na Figura 59, o primeiro subpasso, *matching*, necessita de duas instâncias do metamodelo KDM - uma instância original ("KDM esquerdo"), denominada **versão 1** na Figura 59, e uma instância refatorada ("KDM direito"), **versão 2** na Figura 59. Dadas essas duas instâncias do KDM, os correspondentes elementos nas duas versões do metamodelo KDM são identificados. Em uma instância do KDM, cada elemento possui um identificar único, não volátil e persistente. Portanto, os elementos correspondentes são identificados por meio desses identificadores únicos, como XMI IDs. Por exemplo, ainda analisando a Figura 59, pode-se observar que a instância da metaclasse `ClassUnit` "Artista", apresentada na **versão 1**, corresponde à instância da metaclasse `ClassUnit` "Artista" na **versão 2**. Para as instâncias das metaclasses `ClassUnit` "Ator" e `Extends` na **versão 2**, nenhum elemento correspondente foi identificado na **versão 1**. É visto que para cada elemento correspondente identificado, ou não identificado, um elemento *match* é criado e será utilizado no subpasso seguinte.

No segundo subpasso, *Diffing*, todos os correspondentes elementos identificados são examinados para identificar diferenças em seus meta-atributos. Para cada diferença identificada, um objeto *diff* é criado, o qual descreve com precisão cada diferença identificada entre os correspondentes elementos. Por exemplo, ainda considerando a Figura 59, quando as instâncias da metaclasse `ClassUnit` "Artista" da **versão 1** e **versão 2** são examinadas, é possível observar que o meta-atributo `isAbstract` possui o valor *false* na **versão 1**, e que na **versão 2** o mesmo meta-atributo apresenta o valor *true*, representando a operação `change`. Instâncias de metaclasses que não contêm elementos correspondentes em ambas as versões são consideradas adicionadas ou deletadas (`add` e `delete`). A operação é identificada dependendo da direção; por exemplo, se uma instância de uma metaclasse apenas existe do lado direito ("KDM direito"), essa instância foi adicionada, por outro lado, se uma instância apenas existe do lado esquerdo (KDM esquerdo), essa instância foi deletada. Na Figura 59, é possível identificar que duas instâncias foram adicionadas - uma instância da metaclasse `ClassUnit` denominada Ator e uma instância da metaclasse `Extends`.

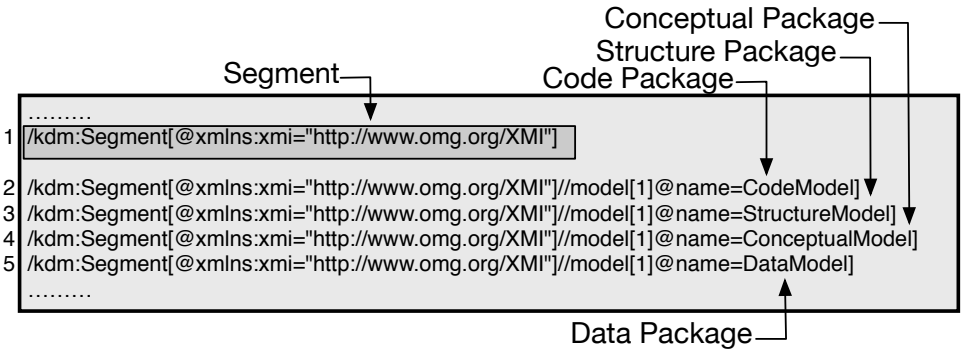
Subsequentemente, o terceiro subpasso, Análise dos *Diffs*, é executado, no qual todos objetos *diffs* criados anteriormente são examinados para criar uma lista de dependência contendo as instâncias das metaclasses alteradas e quais operações foram realizadas. No exemplo apresentado na Figura 59, a lista criada possui três dependências. A primeira dependência informa que o meta-atributo `isAbstract` da metaclasse `ClassUnit` Artista da **versão 1** foi alterado (`change`) de *false* para *true* na **versão 2**. A segunda dependência, ilustra que uma instância da metaclasse `ClassUnit` Ator foi adicionada (`add`) na **versão 2**, e a terceira dependência representa que uma instância da metaclasse `Extends` foi adicionada na **versão 2**.

6.5.2 Identificar Pontos para Executar a Propagação

Nessa seção, o segundo passo do módulo de sincronização é apresentado. Esse passo resume-se basicamente na adaptação do algoritmo DFS para identificar todas as metaclasses que precisam ser sincronizadas/atualizadas após a aplicação da refatoração. Esse algoritmo utiliza como entrada a lista criada no passo anterior. Como as instâncias do metamodelo KDM são persistidas utilizando a padronização XMI, o algoritmo precisa de uma forma para buscar as dependências nesse XMI. Assim, esse passo utiliza expressões em XPath (KAY, 2011) que são executadas na instância do metamodelo KDM para obter todos os pacotes do KDM. Por exemplo, na Figura 60, são apresentadas algumas expressões definidas em XPath, utilizadas antes da aplicação do algoritmo DFS. A primeira expressão retorna a metaclasse `Segment` que é o elemento inicial de qualquer instância do metamodelo KDM. As outras expressões ilustradas nas linhas 2-5 representam os outros pacotes do metamodelo KDM. Os elementos retornados nas expressões XPath são também utilizados como entrada para o algoritmo DFS.

O Algoritmo 2 ilustra como o DFS identifica todas as metaclasses que precisam ser sincronizadas/atualizadas após a aplicação da refatoração. A Figura 61 apresenta como é o

Figura 60 – Expressões definidas em XPath para obter os pacotes do KDM.



Fonte: Elaborada pelo autor.

funcionamento do algoritmo DFS. Cada nó representa uma instância de uma metaclasse do metamodelo KDM e os vértices representam os relacionamentos entre as instâncias das metaclasses, por exemplo, o nó `A` representa uma instância da metaclasse `Segment` e os nós `K`, `H`, `E` e `B` representam instâncias das metaclasses `CodeModel`, `StructureModel`, `ConceptualModel` e `DataModel`, respectivamente. Mais especificamente, o algoritmo funciona da seguinte forma: primeiro é necessário escolher um ponto inicial de partida, no caso do módulo de sincronização o ponto inicial é a instância da metaclasse `Segment`, metaclasse raiz de qualquer instância do metamodelo KDM. Depois, a instância da metaclasse `Segment` deve ser visitada, adicionada em uma pilha e marcada como visitada. Posteriormente, o algoritmo visita outra instância de outra metaclasse que ainda não foi visitada e verifica se ela possui uma associação do tipo `implementation`. Caso afirmativo, o algoritmo deve verificar se essa associação possui referência para algum elemento identificado na lista gerada no primeiro passo; se a associação apresentar um elemento, ele deverá ser adicionado em outra pilha e marcado como visitado. Todo esse processo continua até que o algoritmo alcance a última metaclasse instanciada no KDM.

Em seguida, o algoritmo ainda verifica se a instância da metaclasse `Segment` possui alguma instância adjacente que ainda não foi marcada como visitada. Caso o algoritmo identifique uma instância de metaclasse adjacente ainda não visitada todo o processo é iniciado novamente, sempre verificando a associação `implementation`. Quando o algoritmo finalmente alcançar a última instância da metaclasse, ou seja, todas as instâncias de metaclasse do KDM foram visitadas e verificadas corretamente, o algoritmo criará uma lista contendo todas as instâncias das metaclasses afetadas na refatoração.

6.5.3 Aplicar Propagação

Nesta seção, o terceiro passo do módulo de sincronização é apresentado. Esse passo objetiva realizar as mudanças e propagações necessárias para manter uma determinada instância do metamodelo KDM sincronizada e consistente. A sincronização é importante para o metamodelo KDM, uma vez que ele possui metaclasses que contêm conexões diretas com outras

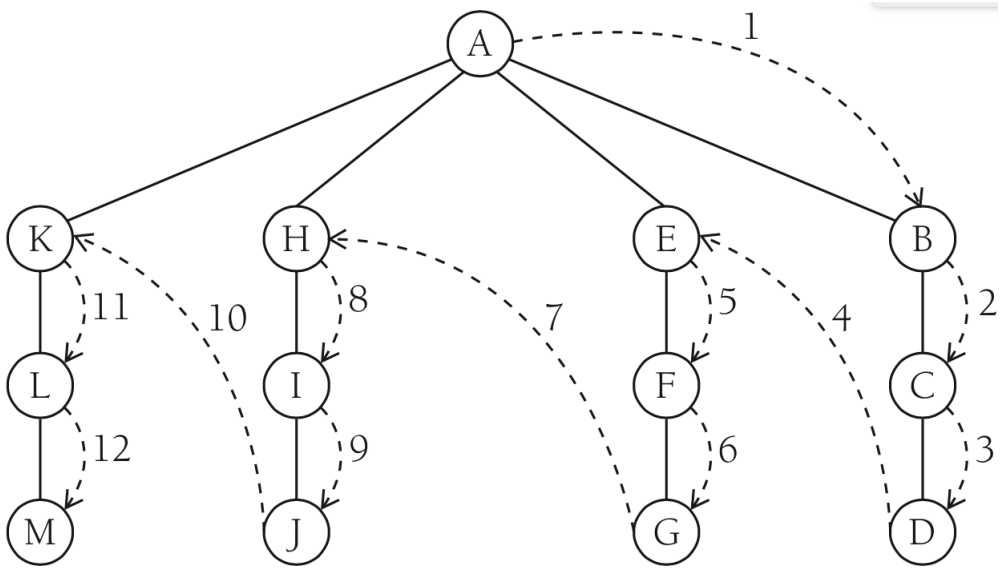
Algoritmo 2: Algoritmo DFS.

Input: DFS (G, u) onde *G* é uma instância do KDM, *u* é a metaclasse inicial obtida pela expressão XPath, ou seja, `Segment`

Output: Uma coleção de metaclasses, as quais precisam ser sincronizadas

```
1  begin
2      foreach outgoing edge e = (u, v) of u do
3          if vertex v as has not been visited then
4              if vertex v contain implementation = true then
5                  foreach implementations element do
6                      verify all elements in implementation
7                  end
8                  Mark vertex v as visited (via edge e). Recursively call DFS (G, v).
9              end
10         end
11     end
12 end
```

Figura 61 – Funcionamento do Algoritmo DFS.



Fonte: Elaborada pelo autor.

metaclasses de outras visões/artefatos do KDM. Assim, manter a instância do metamodelo KDM sincronizado e consistente após a aplicação de uma refatoração é importante.

No contexto desta tese, como apresentado no Capítulo 4, as refatorações que são criadas para o metamodelo KDM são de baixa granularidade e aplicadas diretamente na camada `Code` do metamodelo KDM. Contudo, uma determinada refatoração pode demandar outras modificações que deveriam ser realizadas em outras camadas/visões do metamodelo KDM para mantê-lo consistente e sincronizado. Por exemplo, na refatoração *Rename Package*, o nome de um determinado pacote é alterado de PacoteX para PacoteY se uma instância da metaclasse `Layer`[10]

[10] Metaclasse definida no pacote `Structure` do metamodelo KDM para representar camadas em âmbito arquite-

for utilizada para representar o pacote em âmbito arquitetural, essa mesma instância da metaclasse `Layer` também deve ser renomeada.

Esse passo utiliza um conjunto de regras pré-definidas que são iniciadas de acordo a(s) refatoração(ões) aplicada(s) na instância do metamodelo KDM. Mais detalhadamente, todas as propagações especificadas nesse passo são pré-definidas para serem disparadas após a aplicação de específicas refatorações. Todas as propagações são definidas com base nas mudanças realizadas em uma determinada instância de metaclasses do metamodelo KDM. Além disso, todas as regras pré-definidas foram implementadas em ATL. No Apêndic A nas Tabelas 24, 25, 26 e 27 todas as regras de programação pré-definidas são explicadas.

6.5.4 Exemplo de execução do Módulo de Sincronização

Como apresentado no Capítulo 4, a refatoração `Extract ClassUnit` pode ser criada por um conjunto de duas operações atômicas: `add` e `delete`. Dessa forma, o módulo de sincronização identifica que a refatoração `Extract ClassUnit` executou um conjunto de operações atômica (`add` e `delete`) e cria uma lista para ser utilizada no passo seguinte. No contexto da refatoração `Extract ClassUnit`, apresentada na Figura 48, as seguintes operações foram realizadas:

- `add` uma instância de `ClassUnit` denominada `Document`;
- `delete` uma instância de `StorableUnit` denominada CPF da `ClassUnit Cliente`;
- `add` uma instância de `StorableUnit` na `ClassUnit Document`;
- `add` uma instância de `StorableUnit` do tipo `Document` na `ClassUnit Cliente`;
- `add` duas instâncias de `MethodUnit` para representar os métodos assessores na `ClassUnit Cliente`.

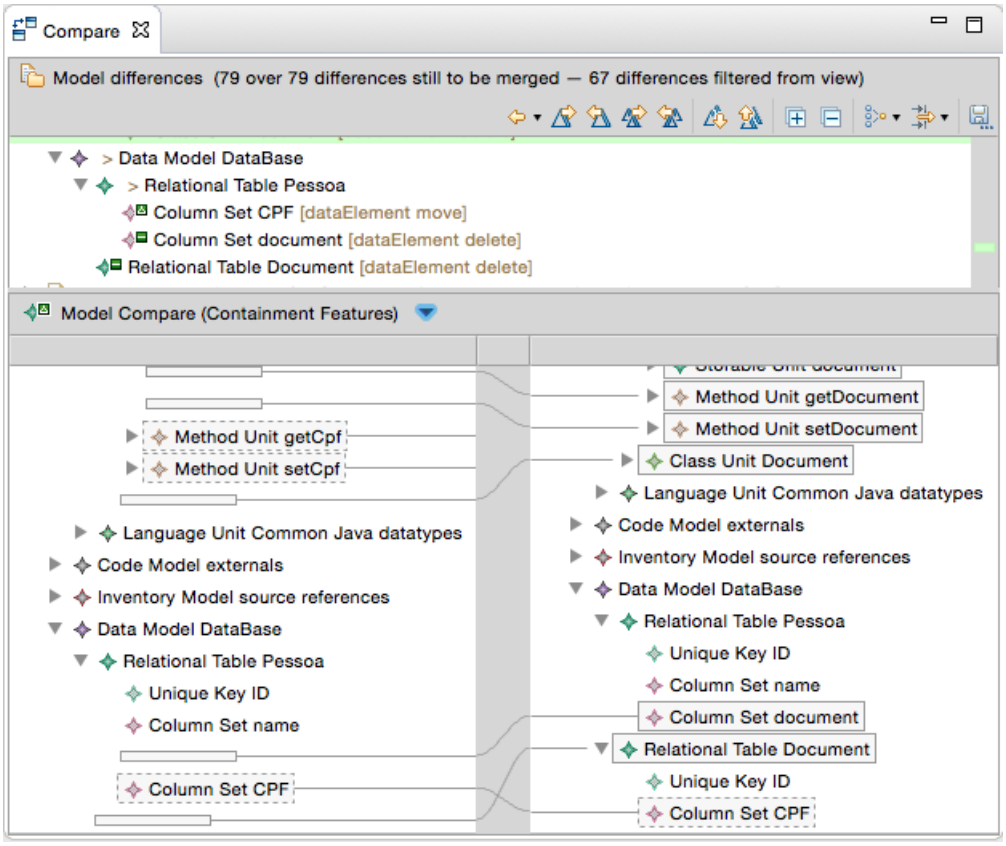
Depois, o módulo de sincronização executa o algoritmo DFS para identificar todas as metaclasses que precisam ser sincronizadas/atualizadas após a aplicação da refatoração. Esse algoritmo utiliza como entrada a lista criada no passo anterior. Expressões definidas em XPath são utilizadas para navegar em todas as visões da instância do metamodelo KDM. Ao término da execução desse algoritmo, ele irá criar uma lista que contém todas as instâncias das metaclasses afetadas na refatoração.

O terceiro passo do módulo de sincronização objetiva realizar as mudanças e propagações necessárias para manter uma determinada instância do metamodelo KDM sincronizada e consistente. Utilizando a lista de operações realizadas na refatoração e a lista que possui os elementos afetados na refatoração, o módulo de programação utiliza um conjunto de regras

tural.

pré-definidas para realizar a propagação. Todas as propagações estão apresentadas nas Tabelas 24, 25, 26 e 27 do Apêndice A. Para a refatoração `Extract ClassUnit` utilizada como exemplo, as propagações que serão realizadas são:

Figura 62 – Instância antes e após a execução do Módulo de Sincronização.



Fonte: Elaborada pelo autor.

- `add` uma instância de `RelationalTable` com o meta-atributo `name` denominado `Document`;
  - `add` uma instância da metaclasse `UniqueKey` na instância `RelationalTable` criada;
- `delete` uma instância de `ColumnSet` denominada CPF da `RelationalTable Pessoa`;
- `add` uma instância de `ColumnSet` com o meta-atributo `name` denominado CPF na instância da metaclasse `RelationalTable Document`;
- `add` uma instância de `ColumnSet` com o meta-atributo `name` denominado `document` na `RelationalTable Pessoa`.

Adicionalmente, a KDM-RE fornece uma forma de visualizar graficamente se as propagações foram realmente executadas. Por exemplo, a Figura 62 representa duas instâncias do metamodelo KDM. A instância apresentada do lado esquerdo é denominada **versão 1** e equivale à instância antes (original) de aplicar o módulo de sincronização. A instância do lado direito é

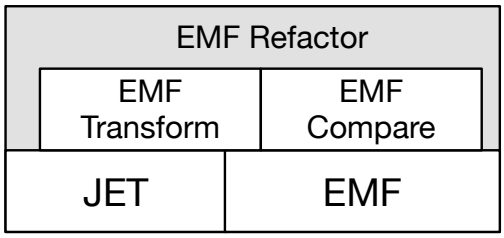
intitulada de **versão 2** e representa uma instância do metamodelo KDM sincronizada e propagada de acordo com os passos e regras definidas pela abordagem KDM-SInc.

6.6 Trabalhos Relacionados

Nesta seção, são descritos os principais trabalhos relacionados com ferramentas que aplicam refatorações em modelos encontradas na literatura. Foram identificadas algumas ferramentas que nortearam o desenvolvimento da KDM-RE, assim, nesta seção, são mostradas as principais semelhanças e diferenças encontradas.

Arendt e Taentzer (2013) em seu artigo propõem uma ferramenta denominada EMF Refactor para aplicar refatorações em modelos EMF. Na Figura 63, é apresentada a arquitetura da ferramenta EMF Refactor. Como pode ser observado, a EMF Refactor também foi desenvolvida para ser utilizada no ambiente de desenvolvimento Eclipse. EMF Refactor também utiliza o *framework* EMF Compare para mostrar os efeitos da refatoração. De acordo com Arendt e Taentzer (2013), a EMF Refactor permite realizar a identificação de *bad smells* e, em seguida, o usuário pode aplicar refatorações. Da mesma forma que a KDM-RE, um conjunto de refatorações propostas por Fowler (1999) foi implementado na EMF Refactor. Por exemplo, na Figura 64 é possível visualizar quais refatorações o usuário pode aplicar por meio da EMF Refactor.

Figura 63 – Arquitetura da EMF Refactor.



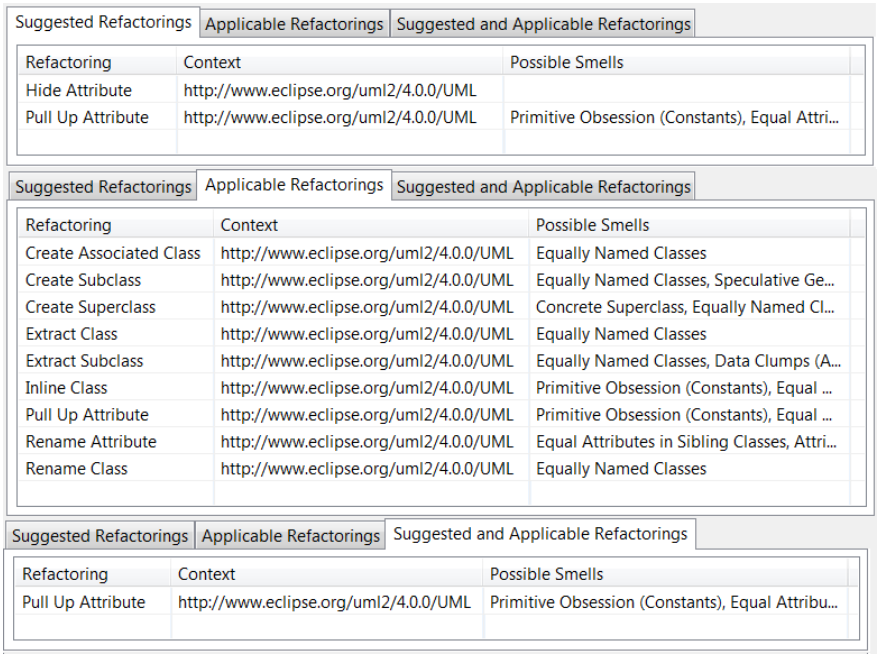
Fonte: Adaptada de Hoste (2013).

Existem duas principais diferenças entre a KDM-RE e a ferramenta EMF Refactor. A primeira é que a ferramenta EMF Refactor não se preocupa em sincronizar as instâncias de um determino metamodelo após a aplicação das refatorações; diferentemente da KDM-Re, a qual define um módulo de sincronização exclusivo para essa característica. A segunda diferença está relacionada com a identificação de qual(is) refatoração(ões) aplicar. Por meio da EMF Refactor, é possível identificar quais refatorações precisam ser realizadas, ou seja, a ferramenta permite identificar quais são os *bad smells*; KDM-RE no seu estado atual não permite a identificação de *bad smells*.

Outra ferramenta relacionada à KDM-RE é a Refactory[11]. A arquitetura da ferramenta Refactory é apresentada na Figura 65. Da mesma forma que a KDM-RE, Refactory também

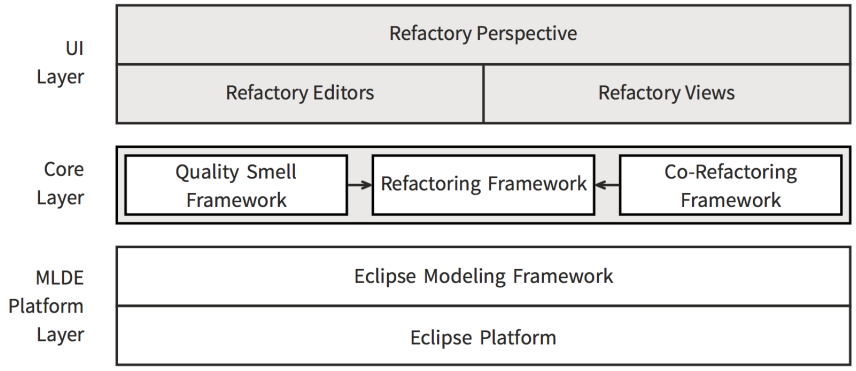
[11] `http://www.modelrefactoring.org/index.php/Refactoring`

Figura 64 – *Wizard* para aplicar refatoração do EMF Refactor.



Fonte: Hoste (2013).

Figura 65 – Arquitetura da ferramenta Refactory.



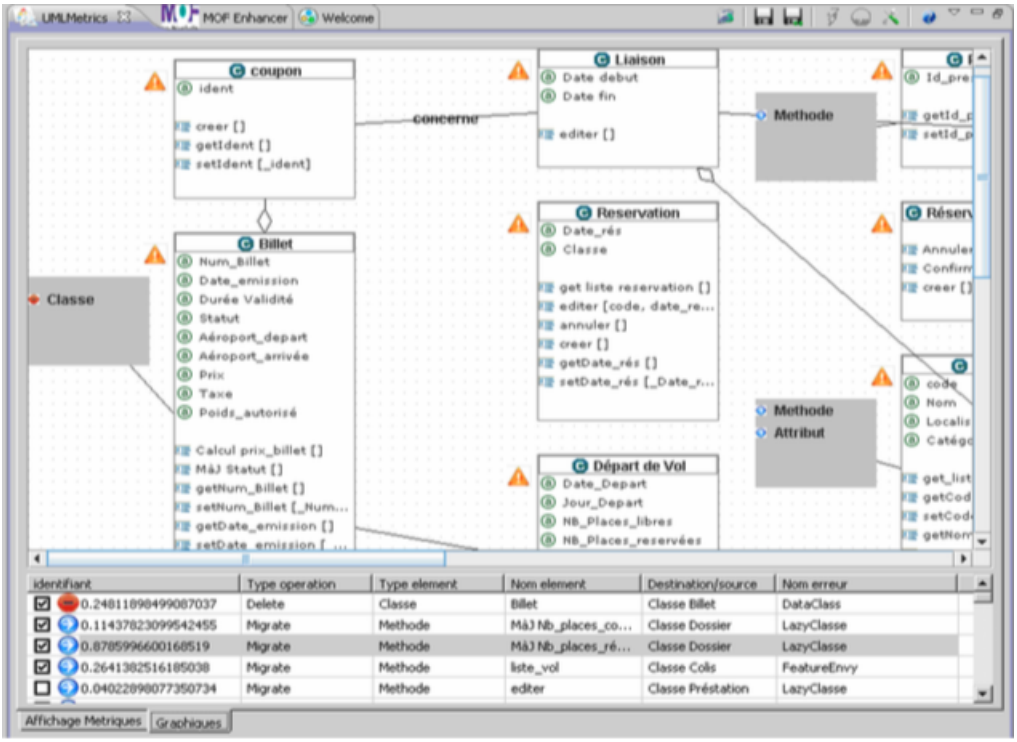
Fonte: Reimann (2015).

contém três camadas: (*i*) *Platform Layer*, (*ii*) *Core Layer* e (*iii*) *UI layer*. A camada *Core* contém três módulos: (*i*) *Quality Smell Framework*, (*ii*) *Refactoring Framework* e (*iii*) *Co-Refactoring Framework*. Note que, a Refactory também é baseada no EMF e pode refatorar qualquer metamodelo definido pelo EMF. Além disso, de acordo com Reimann, Seifert e Aßmann (2010), Refactory foi criada para permitir refatorações genéricas. Os autores afirmam que, realizar uma refatoração, por exemplo, *extract method* em Java é o mesmo mecanismo para um sistema representado por instância da UML. As refatorações são todas implementadas em QVT e as asserções são implementadas em OCL. As principais semelhanças com a KDM-RE são: (*i*) Refactory também permite aplicar refatorações graficamente por meio de diagramas da UML, (*ii*) refatorações são executadas utilizando QVT, uma linguagem de transformação similar à ATL, (*iii*) restrições (pré- e pós-condições) são implementadas em OCL e (*iv*) Refactory

também não identifica *bad-smells*. As principais diferenças são: (*i*) Refactory não se preocupa em sincronizar/propagar o metamodelo após a aplicação de refatorações e (*ii*) Refactory não utiliza o metamodelo KDM para aplicar as refatorações.

Mohamed, Romdhani e Ghedira (2011) propõem uma abordagem e uma ferramenta denominada M-Refactor. M-Refactor possui duas principais funcionalidades: (*i*) identificar *bad-smells* em instâncias do metamodelo UML e (*ii*) aplicar refatorações em instâncias do metamodelo UML. A identificação dos *bad-smells* é realizada em diagramas de classe e diagramas de sequência da UML. Os *bad-smells* identificados são representados graficamente como apresentado na Figura 66. Após a identificação e visualização dos *bad-smells*, o usuário pode aplicar as refatorações. As principais semelhanças entre KDM-RE e M-Refactor são: (*i*) ambas utilizam diagramas da UML para aplicar as refatorações, (*ii*) M-Refactor também utiliza linguagens de transformações para executar as refatorações e (*iii*) linguagens de restrições (pré- e pós-condições) também são especificadas na M-Refactor. As principais diferenças entre M-Refactor e KDM-RE são: (*i*) M-Refactor pode ser utilizada para identificar *bad-smells* em instâncias do metamodelo UML, (*ii*) M-Refactor não implementa nenhum mecanismo para propagar/sincronizar a instância do metamodelo UML após a aplicação de um conjunto de refatorações e (*iii*) M-Refactor não utiliza o metamodelo KDM como base para aplicar as refatorações.

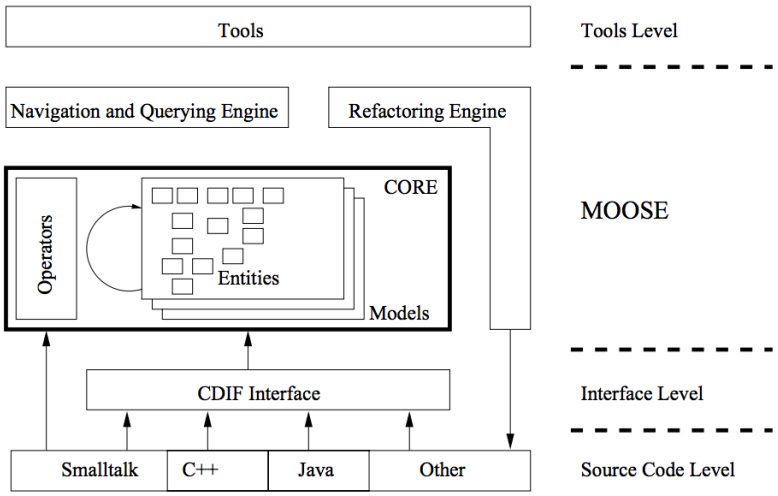
Figura 66 – Visão geral da ferramenta M-Refactor.



Fonte: Mohamed, Romdhani e Ghedira (2011).

Outra ferramenta relacionada é a MOOSE (TICHELAAR *et al.*, 2000; DUCASSE; LANZA; TICHELAAR, 2000). A arquitetura da ferramenta MOOSE é apresentada na Figura 67. Como pode ser observado, MOOSE aceita como entrada diversos tipos de linguagem de progra-

Figura 67 – Arquitetura da Ferramenta MOOSE.

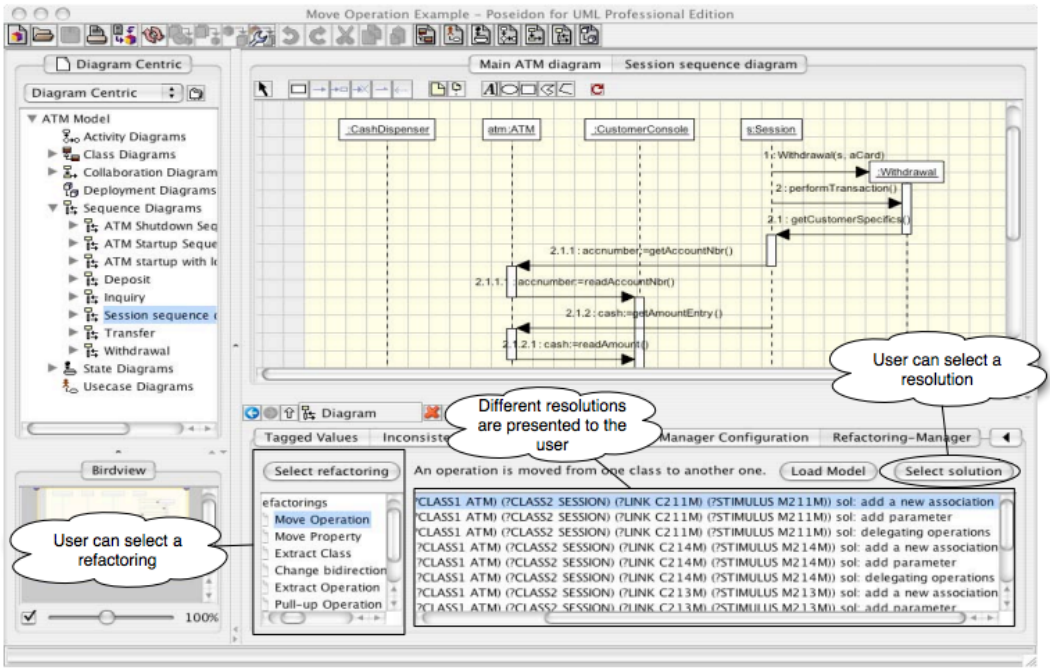


Fonte: Ducasse, Lanza e Tichelaar (2000).

mação (Smalltalk, Java, C++, etc) e, em seguida, MOOSE transforma tais linguagens em um metamodelo denominado FAMIX (TICHELAAR *et al.*, 2000). Após realizar essa transformação, refatorações podem ser aplicadas. De acordo com os autores, MOOSE permite a aplicação de refatorações em nível de modelo de forma independente utilizando o metamodelo FAMIX (TICHELAAR *et al.*, 2000). MOOSE utiliza a ferramenta Refactoring Browser (ROBERTS; BRANT; JOHNSON, 1997) e, assim, permite que as refatorações aplicadas em instâncias do metamodelo FAMIX sejam automaticamente replicadas no código-fonte Smalltalk, porém, essa função apenas funciona quando a linguagem de entrada é a Smalltalk. Da mesma forma que a KDM-RE, MOOSE, permite a aplicação de refatorações de forma independente de linguagem, além disso, KDM-RE e MOOSE permitem a aplicação de refatorações por meio de diagramas. As principais diferenças são: (*i*) MOOSE permite identificar *bad-smells* utilizando o metamodelo FAMIX, (*ii*) MOOSE não implementa nenhum mecanismo de propagação/sincronização após a aplicação um conjunto de refatorações, isso se deve ao fato do metamodelo FAMIX ser sucinto e representar apenas construções do POO, (*iii*) linguagem de transformação também não é utiliza na MOOSE, (*iv*) restrições (pré- e pós-condições) também não são apoiadas na MOOSE e (*v*) MOOSE utiliza o metamodelo FAMIX, um metamodelo proprietário.

RACOoN é um *plug-in* desenvolvido por Straeten e D'Hondt (2006), ver Figura 68. Utilizando esse *plug-in*, regras de transformação em modelo podem ser implementadas, carregadas e em seguida executadas em diagramas da UML. RACOoN é uma ferramenta totalmente manual e permite que o usuário definida qual refatoração almeja aplicar, sendo de inteira responsabilidade do usuário definir a refatoração corretamente. Inconsistências encontradas durante a aplicação de múltiplas refatorações são apresentadas para o usuário juntamente com um conjunto de soluções. As principais semelhanças entre KDM-RE e RACOoN são: (*i*) RACOoN também não identifica *bad-smells*, (*ii*) as refatorações podem ser aplicadas graficamente por meio de diagramas da UML, (*iii*) linguagem de transformação também é utilizada para implementar as refatorações

Figura 68 – Visão geral da Ferramenta RACOoN.



Fonte: Straeten e D'Hondt (2006).

e (*iv*) linguagem de restrição é utilizada para implementar as pré- e pós-condições. As principais diferenças são: (*i*) nenhum mecanismo de propagação/sincronização é implementado na RACOoN e (*ii*) RACOoN utiliza o metamodelo UML, não KDM.

Boger, Sturm e Fragemann (2003) definiram uma extensão da ferramenta Refactoring Browser (ROBERTS; BRANT; JOHNSON, 1997) e criaram a primeira ferramenta para aplicar refatorações em modelos UML, a qual é denominada Refactoring Browser for UML. Algumas semelhanças com a KDM-RE podem ser destacadas: (*i*) Refactoring Browser for UML também não identifica *bad-smells*, (*ii*) as refatorações podem ser aplicadas graficamente por meio de diagramas da UML e (*iii*) a linguagem de restrição OCL também é utilizada na Refactoring Browser for UML. As principais diferenças são: (*i*) nenhum mecanismo de sincronização/propagação é implementado na Refactoring Browser for UML, (*ii*) linguagem de transformação (ATL e/ou QVT) não é utilizada na Refactoring Browser for UML e (*iii*) apenas KDM-RE utiliza o metamodelo KDM.

VisTra (ŠTOLC; POLÁŠEK, 2010) é uma ferramenta visual desenvolvida também como um *plug-in* para o ambiente de desenvolvimento Eclipse, a qual aplica refatorações em diagramas de classes da UML. Essa ferramenta permite que o usuário defina regras de transformação graficamente e, em seguida, a ferramenta gera automaticamente restrições em OCL e regras de transformações em QVT. As principais semelhanças entre KDM-RE e VisTra podem ser resumidas como: (*i*) VisTra também permite aplicar refatorações por meio de diagramas da UML, (*ii*) a linguagem de transformação QVT, linguagem similar à ATL, é utilizada para implementar as refatorações e (*iii*) a linguagem de restrição OCL também é utilizada para definir as pre- e

pós-condições. As diferenças entre VisTra e KDM-RE são: (*i*) VisTra pode ser utilizada para identificar *bad-smells*, (*ii*) nenhum mecanismo de propagação/sincronização é implementado na VisTra e (*iii*) VisTra não utiliza o metamodelo KDM.

NEPTUNE (MILLAN *et al.*, 2009) é uma ferramenta que permite verificar e transformar instâncias do metamodelo UML. Essa ferramenta utiliza uma extensão da OCL denominada pOCL para automatizar a detecção de *bad-smells*. Em seguida, NEPTUNE sugere um conjunto de refatorações para reparar os *bad-smells* identificados. As principais semelhanças entre KDM-RE e NEPTUNE podem ser resumidas como: (*i*) NEPTUNE também permite aplicar refatorações por meio de diagramas da UML, (*ii*) a linguagem de transformação QVT, linguagem similar à ATL, é utilizada para implementar as refatorações e (*iii*) a linguagem de restrição OCL também é utilizada para definir as pre- e pós-condições. As diferenças entre NEPTUNE e KDM-RE são: (*i*) NEPTUNE pode ser utilizada para identificar *bad-smells*, (*ii*) nenhum mecanismo de propagação/sincronização é implementado na NEPTUNE e (*iii*) NEPTUNE não utiliza o metamodelo KDM.

Tabela 15 – Comparação entre a KDM-RE e as ferramentas relacionadas.

Ferramenta	B-S	AG	S/P	LT	LR	KDM
KDM-RE	✗	✓	✓	✓	✓	✓
EMF Refactor (ARENDT; TAENTZER, 2013)	✓	✓	✗	✗	✓	✗
Refactory (REIMANN; SEIFERT; ASSMANN, 2010)	✗	✓	✗	✓	✓	✗
M-Refactor (MOHAMED; ROMDHANI; GHEDIRA, 2011)	✓	✓	✗	✓	✓	✗
MOOSE (DUCASSE; LANZA; TICHELAAR, 2000)	✗	✓	✗	✗	✗	✗
RACOoN (STRAETEN; D'HONDT, 2006)	✗	✓	✗	✓	✓	✗
Refactoring Browser for UML (BOGER; STURM; FRAGEMANN, 2003)	✗	✓	✗	✗	✓	✗
VisTra (ŠTOLC; POLÁŠEK, 2010)	✓	✓	✗	✓	✓	✗
NEPTUNE (MILLAN *et al.*, 2009)	✓	✓	✗	✓	✓	✗

Na Tabela 15 é apresentada uma comparação entre a KDM-RE e as ferramentas relacionadas identificadas. Alguns critérios foram definidos durante a comparação, tais critérios são: (*i*) a ferramenta permite identificar *bad-smells*, (*ii*) a ferramenta permite aplicar refatorações graficamente, ou seja, por meio de diagramas, (*iii*) a ferramenta permite a sincronização/propagação da instância do metamodelo após aplicar um conjunto de refatorações, (*iv*) a ferramenta utiliza linguagens de transformação para executar a refatoração, (*v*) a ferramenta utiliza linguagens de restrições para verificar pré- e pós-condições. A Tabela 15 contém sete colunas e algumas foram abreviadas: (*i*) "B-S" significa "***B**ad-**S**mells*", (*ii*) "AG" significa "**A**plicação das refatorações **G**raficamente", (*iii*) "S/P" representa se a ferramenta realiza a "**S**incronização/**P**ropagação" após aplicar um conjunto de refatorações, (*iv*) "LT" representa se a ferramenta utiliza "**L**inguagens de **T**ransformações" para aplicar a refatoração, (*v*) "LR" informa se a ferramenta utiliza "**L**inguagens

de **R**estrições" para verificar as pré- e pós-condições e (*vi*) "KDM" representa se a ferramenta aplica as refatorações no metamodelo KDM. O símbolo "✓" representa que o metamodelo define o critério e o símbolo "✗" representa que o metamodelo não define o critério.

Embora todas as ferramentas identificadas apliquem refatorações em nível de modelo, nenhuma realiza as refatorações utilizando o metamodelo KDM. Além disso, nenhuma ferramenta identificada fornece uma forma de reutilizar as refatorações. Dessa forma, acredita-se que a KDM-RE é uma contribuição para a área de pesquisa de refatoração em nível de modelo. Além disso, somente a KDM-RE utiliza o metamodelo KDM como base para as refatorações, o que significa que a KDM-RE é independente de plataforma e de linguagem de programação. Embora todas as ferramentas identificadas utilizem o EMF e/ou UML, nenhuma utiliza o metamodelo KDM. Uma das vantagens de utilizar o KDM é que ele possui diversas metaclasses para representar níveis diferentes de um determinado sistema.

6.7 Considerações Finais

O presente capítulo foca a ferramenta KDM-RE, a qual foi implementada de modo a ser utilizada em conjunto com os demais recursos oferecidos pelo ambiente de desenvolvimento Eclipse IDE. Dessa forma, a KDM-RE foi desenvolvida utilizando os conceitos de *plug-ins*. Três *plug-ins* foram criados e organizados em módulos de acordo com a sua predominante funcionalidade. Os três principais módulos da KDM-RE são: (*i*) módulo de refatoração, (*ii*) módulo do SRM e (*iii*) módulo de sincronização.

O primeiro módulo é responsável por criar uma infraestrutura que auxilia o engenheiro de software a aplicar refatorações em nível de modelos no contexto do metamodelo KDM. O engenheiro de software pode aplicar as refatorações no metamodelo KDM por meio de duas interfaces; a primeira denominada *model browser* permite que o engenheiro tenha uma visão de árvore da instância do metamodelo KDM e aplique refatorações. A segunda interface permite que o engenheiro de software aplique refatorações diretamente em diagramas de classe da UML. Embora o engenheiro de software utilize diagrama de classe da UML na segunda interface, as refatorações são aplicadas transparentemente na instância do metamodelo KDM; o diagrama de classe da UML é utilizado apenas para extrair metadados (nome da classe, nome do atributo, tipo do atributo, etc.), que são enviados como entrada para a refatoração pré-definida em ATL.

O segundo módulo é responsável por disponibilizar uma DSL para instanciar o metamodelo SRM apresentado no Capítulo 5. Além disso, o módulo do SRM fornece uma forma de reutilizar e compartilhar refatorações por meio de instâncias do metamodelo SRM. Esse segundo módulo converte a sintaxe e a semântica da DSL em um arquivo XMI. Cada marcação desse arquivo XMI representa uma instância da metaclasse do metamodelo SRM. Para permitir o reúso e o compartilhamento de refatorações de forma consistente e abrangente, um repositório remoto foi criado, o qual é dedicado para executar solicitações RESTful. Instâncias do metamodelo

SRM são enviadas e recebidas por meio da API RESTful. JPA e o banco de dados MySQL foram utilizados para realizar as persistências das instâncias do metamodelo SRM.

Finalmente, o terceiro módulo é responsável por implementar uma forma de manter a instância do metamodelo KDM sincronizada e consistente. Esse módulo visa propagar e sincronizar uma determinada instância do metamodelo KDM após a aplicação de refatorações. Essas propagações são realizadas com base em regras pré-definidas e apresentadas no Apêndice A. Como o metamodelo KDM possui um conjunto de pacotes para representar diferentes artefatos, é importante manter a instância e todos os outros artefatos sincronizados após a aplicação de refatorações. No Capítulo 7, uma avaliação é executada para avaliar a abordagem apresentada nesta tese.

CAPÍTULO

7

AVALIAÇÃO

7.1 Considerações Iniciais

Nos Capítulo 4 e 5, são apresentadas soluções para auxiliar o engenheiro de modernização a criar e especificar refatorações no contexto da abordagem ADM e do metamodelo KDM. Além disso, no Capítulo 6, a ferramenta KDM-RE é apresentada. KDM-RE automatiza todo o processo de aplicação, reutilização e propagação de mudanças no contexto do metamodelo KDM, deixando somente sob responsabilidade do engenheiro de software a identificação de onde aplicar refatorações. Desse modo, com o uso dessa ferramenta, o engenheiro de software tem um ambiente de desenvolvimento integrado ao Eclipse, onde refatorações podem ser aplicadas e reutilizadas sem se preocupar com a propagação de mudanças para outras visões/artefatos representadas em uma determinada instância do metamodelo KDM.

Um experimento foi realizado com o propósito de avaliar se as refatorações criadas para o metamodelo KDM contribuem para melhorar a qualidade de um sistema. Esse experimento foi planejado e executado seguindo a abordagem definida por Wohlin *et al.* (2012), que é composta por três principais fases: (*i*) definição e planejamento: que consistem em especificar o contexto, formular as hipóteses, fornecer as definições operacionais das variáveis e detalhar os participantes (quando aplicável); (*ii*) operação: envolvendo detalhes relacionados à preparação e execução do experimento; e (*iii*) análise dos dados: que consiste em examinar as informações coletadas durante o experimento de acordo com o tipo dessas informações.

Na Seção 7.2, é descrito o experimento que verifica se as refatorações criadas para o metamodelo KDM realmente melhoram a qualidade de um sistema. Por fim, na Seção 7.3, são comentadas as considerações finais.

7.2 Experimento: Refatorações no contexto do metamodelo KDM

Nesta seção, é apresentado um experimento conduzido com o objetivo de analisar as qualidades das refatorações criadas para o metamodelo KDM. Especificamente, a seguinte questão de pesquisa é investida nesse experimento:

Questão de Pesquisa (*QP*): Como as refatorações criadas para o metamodelo KDM podem ser úteis para engenheiros de software no cenário do mundo real? O benefício esperado de refatoração não está limitada apenas à correção de *bad-smells*, mas também as refatorações podem e devem ser utilizadas para melhorar os atributos de qualidade dos programas, ou seja, melhorar a "reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia".

Dessa forma, para avaliar a *QP*, sete sistemas foram escolhidos para aplicar um conjunto de refatorações. Esses sete sistemas são bem conhecidos na literatura e estão implementados na linguagem de programação Java: Xerces-J[1], Jexel[2], JFreeChart[3], Junit[4], GanttProject[5], ArtofIllusion[6] e JHotDraw[7]. Xerces-J é uma família de software para analisar (*parsing*) XML; Jexel é API para escrever expressões regulares em Java; JFreeChart é uma biblioteca Java utilizada para gerar gráficos; JUnit é uma biblioteca utilizada para gerar teste de unidades; GanttProject é um sistema para gerenciamento de projetos; ArtofIllusion é uma API para modelagem e renderização em 3D; JHotDraw é uma ferramenta para auxiliar a criação de desenhos. Todos esses sistemas foram transformados em instâncias do metamodelo KDM, utilizando a ferramenta MoDisco[8].

Tabela 16 – Sistemas utilizados no Experimento.

Sistemas	Siglas	Versão	KLOC	Classes
Xerces-J	XJ	2.7.0	240	991
Jexel	JEX	1.3	50.4	75
JFreeChart	JFC	1.0.9	170	521
JUnit	JUn	4.0	17.48	225
GanttProject	GP	1.10.2	41	245
ArtofIllusion	AOIL	2.8.1	87	459
JHotDraw	JHD	7.0.6	16	468

Na Tabela 16, é possível visualizar algumas informações sobre os sete sistemas. Como apresentado no Capítulo 6, a ferramenta KDM-RE apenas auxilia o engenheiro de software a aplicar refatorações em instâncias do metamodelo KDM. É de responsabilidade do engenheiro

[1] http://xerces.apache.org/xerces-j/
[2] https://jexel.googlecode.com
[3] http://www.jfree.org/jfreechart/
[4] http://junit.org/
[5] http://www.ganttproject.biz/
[6] www.artofillusion.org
[7] jhotdraw.org
[8] https://eclipse.org/MoDisco/

de software identificar onde aplicar a refatoração, uma vez que a ferramenta KDM-RE ainda não identifica *bad-smells* (FOWLER, 1999). Nesse contexto, esses sete sistemas foram aplicados nas ferramentas Jdeodorant[9] e inFusion[10], as quais identificam um conjunto de *bad-smells*.

Os *bad-smells* identificados nesses sistemas são: *Blob*/*God Class* (uma classe que faz coisa demais no sistema), *Data Class* (uma classe demonstra apenas atributos, porém, não os utiliza para realizar nenhum tipo de processamento), *Spaghetti Code* (código com uma estrutura de controle complexo e emaranhado) e *Functional Decomposition* (quando uma classe desempenha uma única função, ao invés de ser um encapsulamento de dados e funcionalidade) (FOWLER, 1999). Está fora do escopo deste capítulo se aprofundar nesses *bad-smells*.

O principal objetivo desse experimento é verificar se, após a aplicação de um conjunto de refatorações com base nos *bad-smells* identificados pelas ferramentas Jdeorante e inFusion, as refatorações criadas para o metamodelo KDM melhoraram os sistemas em termos de atributos de qualidade. Os atributos de qualidade foram avaliados utilizando o modelo *Quality Model for Object-Oriented Design* (QMOOD) (BANSIYA; DAVIS, 2002). O QMOOD é um modelo de qualidade para POO que estabelece um modelo hierárquico definido e empiricamente validado para avaliar atributos de qualidade de POO. O uso do modelo QMOOD nesse experimento ocorreu, pois: (*i*) o QMOOD é bastante utilizado na literatura (O'KEEFFE; CINNÉIDE, ; SENG; STAMMEL; BURKHART, 2006; JENSEN; CHENG, 2010) para avaliar o efeito de refatoração e (*ii*) o QMOOD define seis atributos de qualidade ("facilidade de compreensão" (*understandability*), "eficácia" (*effectiveness*), "extensibilidade" (*extendibility*), "reusabilidade" (*reusability*), "funcionalidade" (*functionality*) e "flexibilidade" (*flexibility*)), que são calculados utilizando 11 métricas (BANSIYA; DAVIS, 2002) (ver Tabela 17 e Tabela 18).

Nesse experimento apenas quatro atributos de qualidade foram considerados: "reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia". De acordo com Bansiya e Davis (2002) esses atributos podem ser definidos como:

- Reusabilidade: características de POO que permitem um projeto ser reaplicado para um novo problema sem esforço significativo;
- Flexibilidade: a capacidade de um projeto ser adaptado para disponibilizar novos recursos;
- Facilidade de Compreensão: propriedades do projeto que permitem ser facilmente aprendido e compreendido. Está relacionado com a complexidade da estrutura do projeto;
- Eficácia: capacidade do projeto obter a funcionalidade e o comportamento desejado, utilizando conceitos e técnicas de POO.

O atributo de qualidade "funcionalidade" não foi levado em consideração nesse experimento pois assume-se que, por definição, refatorações não devem alterar o comportamento

[9] `https://marketplace.eclipse.org/content/jdeodorant`
[10] `https://www.intooitus.com/products/infusion`

observável, ou seja, a funcionalidade do sistema[11]. "Extensibilidade" também não foi levada em consideração no experimento devido à subjetividade associada a esse atributo de qualidade (BANSIYA; DAVIS, 2002).

A Tabela 17 apresenta as propriedades de POO e as métricas que foram utilizadas para mensurar as instâncias do metamodelo KDM antes e após a aplicação das refatorações. A Tabela 18 apresenta os atributos de qualidade e as propriedades definidas por Bansiya e Davis (2002). Nesse experimento, o ganho de qualidade é calculado pela comparação de cada atributo de qualidade antes e após a aplicação de um conjunto de refatorações. Com isso, o ganho total da qualidade (G_q) para cada atributo de qualidade QMOOD pode ser estimado de acordo com a Definição 3.

Definição 3. *$G_{q_i} = q'_i - q_i$ onde q'_i e q_i representam os valores do atributo de qualidade i antes e após a aplicação das refatorações, respectivamente.*

Tabela 17 – Métricas utilizadas no modelo QMOOD (BANSIYA; DAVIS, 2002).

Propriedades de Projeto	Métrica	Descrição
Tamanho do Projeto	*Design Size in Classes* (DSC)	Número de classes no projeto
Hierarquias	*Number Of Hierarchies* (NOH)	Número de Hierarquias
Abstração	*Average Number of Ancestors* (ANA) ANA	Média de Ancestrais
Encapsulamento	*Data Access Metric* (DAM)	Métrica de acesso a dados
Acoplamento	*Direct Class Coupling* (DCC)	Acoplamento direto entre classes
Coesão	*Cohesion Among Methods in class* (CAM)	Coesão entre os métodos da classe
Composição	*Measure Of Aggregation* (MOA)	Medida de agregação
Herança	*Measure of Functional Abstraction* (MFA)	Abstração funcional
Polimorfismo	*Number Of Polymorphic methods* (NOP)	Número de métodos polimórficos
Troca de Mensagens	*Class Interface Size* (CIS)	Tamanho da interface de uma classe
Complexidade	*Number of methods* (NOM)	Quantidade de métodos

7.2.1 Definição e Planejamento do Experimento

Esse experimento foi planejado utilizando o modelo GQM (*Goal, Question, Metric*) (WOHLIN *et al.*, 2012), que é dividido em cinco partes:

[11] No contexto desta tese, as refatorações criadas preservam a sintaxe e a semântica de instâncias do KDM.

Tabela 18 – Relacionamento de atributos de qualidade Vs. Propriedade no QMOOD (BANSIYA; DAVIS, 2002).

Atributo de Qualidade	Fórmula com base nas propriedades de projeto
Reusabilidade	-0,25 x DCC + 0,25 x CAM + 0,5 x CIS + 0,5 x DSC
Flexibilidade	0.25 x DAM - 0.25 x DCC + 0.5 x MOA +0.5 x NOP
Facilidade de Compreensão	-0.33 x ANA + 0.33 x DAM - 0.33 x DCC + 0.33 x CAM -0.33 x NOP + 0.33 x NOM - 0.33 x DSC
Eficácia	0.2 x ANA + 0.2 x DAM + 0.2 x MOA + 0.2 x MFA + 0.2 x NOP

- **Objeto de estudo**: o objeto de estudo desse experimento são as refatorações criadas para o metamodelo KDM;
- **Propósito**: o propósito desse experimento é avaliar as refatorações criadas para o metamodelo KDM;
- **Perspectiva**: esse experimento é executado sob a perspectiva de um engenheiro de software;
- **Qualidade de foco**: os atributos de qualidade QMOOD são os efeitos primários investigados nesse experimento. Mais especificamente, os atributos "reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia" após a aplicação de um conjunto de refatoração;
- **Contexto**: esse experimento foi conduzido utilizando o ambiente de desenvolvimento Eclipse 4.3.2 em uma máquina 2.5 GHz Intel Core i5 com 8GB de memória física, executando o sistema operacional Mac OS X 10.9.2.

O experimento pode ser resumido utilizando o seguinte *template* (WOHLIN *et al.*, 2012): **Analisar** as refatorações criadas para o metamodelo KDM; **com o propósito de** avaliar as refatorações criadas; **com respeito à** "reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia"; **do ponto de vista dos** pesquisadores; **no contexto de** distintos sistemas.

7.2.1.1 Formulação das Hipóteses

A *QP* foi formalizada em hipóteses. Para cada atributo de qualidade ("reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia"), hipóteses foram criadas como apresentado a seguir:

- "Reusabilidade":
  - **Hipótese Nula** $(H1_0)$: Não há diferença entre instâncias do metamodelo KDM antes e após a aplicação de um conjunto de refatorações em relação à reusabilidade. Isso pode ser formalizado como:

  - $H1_0$: $\mu_{reusabilidade_{antes}} = \mu_{reusabilidade_{depois}}$

- **Hipótese Alternativa** ($H1_1$): Há diferença entre as instâncias do metamodelo KDM refatorada e a instância do metamodelo KDM não refatorada em relação à reusabilidade. Isso pode ser formalizado como:

  - $H1_1$: $\mu_{reusabilidade_{antes}} \neq \mu_{reusabilidade_{depois}}$;

- "Flexibilidade":

  - **Hipótese Nula** ($H2_0$): Não há diferença entre instâncias do metamodelo KDM antes e após a aplicação de um conjunto de refatorações em relação à flexibilidade. Isso pode ser formalizado como:

    - $H2_0$: $\alpha_{flexibilidade_{antes}} = \alpha_{flexibilidade_{depois}}$

  - **Hipótese Alternativa** ($H2_1$): Há diferença entre as instâncias do metamodelo KDM refatorada e a instância do metamodelo KDM não refatorada em relação à flexibilidade. Isso pode ser formalizado como:

    - $H2_1$: $\alpha_{flexibilidade_{antes}} \neq \alpha_{flexibilidade_{depois}}$;

- "Facilidade de Compreensão":

  - **Hipótese Nula** ($H3_0$): Não há diferença entre instâncias do metamodelo KDM antes e após a aplicação de um conjunto de refatorações em relação à facilidade de compreensão. Isso pode ser formalizado como:

    - $H3_0$: $\beta_{compreensao_{antes}} = \beta_{compreensao_{depois}}$

  - **Hipótese Alternativa** ($H3_1$): Há diferença entre as instâncias do metamodelo KDM refatorada e a instância do metamodelo KDM não refatorada em relação à facilidade de compreensão. Isso pode ser formalizado como:

    - $H3_1$: $\beta_{compreensao_{antes}} \neq \beta_{compreensao_{depois}}$;

- "Eficácia":

  - **Hipótese Nula** ($H4_0$): Não há diferença entre instâncias do metamodelo KDM antes e após a aplicação de um conjunto de refatorações em relação à eficácia. Isso pode ser formalizado como:

    - $H4_0$: $\gamma_{eficacia_{antes}} = \gamma_{eficacia_{depois}}$

  - **Hipótese Alternativa** ($H4_1$): Há diferença entre as instâncias do metamodelo KDM refatorada e a instância do metamodelo KDM não refatorada em relação à eficacia. Isso pode ser formalizado como:

    - $H4_1$: $\gamma_{eficacia_{antes}} \neq \gamma_{eficacia_{depois}}$;

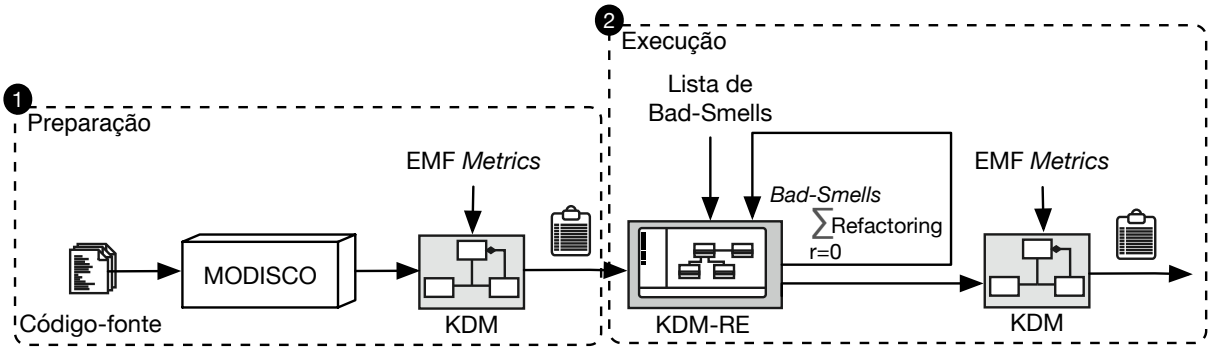
7.2.1.2 Variáveis

O experimento teve as seguintes variáveis independentes: (*i*) a ferramenta KDM-RE, (*ii*) EMF *Metrics*, Jdeodorant[12] e inFusion[13], (*iii*) o ambiente de desenvolvimento Eclipse versão 4.3.2, (*iv*) a linguagem de programação Java e (*v*) os sete sistemas utilizados (ver Tabela 16). As variáveis dependentes são as seguintes: (*i*) reusabilidade, (*ii*) flexibilidade, (*iii*) facilidade de compreensão e (*iv*) eficácia.

7.2.2 Operação do Experimento

Depois de definir e planejar o experimento, sua fase de operação foi realizada em duas etapas: Preparação e Execução. Na Figura 69, é apresentado o esquema seguido durante a preparação (❶) e execução (❷) desse experimento.

Figura 69 – Preparação e Execução ilustradas do Experimento.



Fonte: Elaborada pelo autor.

7.2.2.1 Preparação

Alguns dias antes da realização do experimento, os códigos-fontes dos sete sistemas apresentados na Tabela 16 foram baixados. Em seguida, as ferramentas Jdeodorant e inFusion foram executadas para cada sistema com o intuito de identificar os *bad-smells*. Assim, uma lista com todos os *bad-smells* identificados para cada sistema foi obtida.

Posteriormente, todos os sete sistemas foram transformados em instâncias do metamodelo KDM. Essa transformação foi realizada por meio da ferramenta MoDisco[14] (ver Figura 69 ❶). Em seguida, para cada instância do metamodelo KDM gerada (que representa os sete sistemas), a ferramenta EMF *Metrics* (ARENDT; TAENTZER, 2012; THORSTEN; PAWEL; GABRIELE, 2010) foi executada. Essa ferramenta foi escolhida para mensurar os atributos de qualidade QMOOD (reusabilidade, flexibilidade, facilidade de compreensão e eficácia). Segundo Arendt e Taentzer (2012), Thorsten, Pawel e Gabriele (2010) EMF *Metrics* é um Eclipse *plug-in* que

[12] https://marketplace.eclipse.org/content/jdeodorant
[13] `https://www.intooitus.com/products/infusion`
[14] `https://eclipse.org/MoDisco/`

provê a especificação e calcula métricas para qualquer instância de metamodelo que utilize como base o metamodelo EMF, nesse caso, o metamodelo KDM. Mais especificamente, EMF *Metrics* calcula as métricas apresentadas na Tabela 17 e, em seguida, as fórmulas apresentadas na Tabela 18 foram mensuradas. Como resultado da aplicação do EMF *Metrics* e da Tabela 18, uma lista especificando os atributos de qualidade QMOOD ("reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia") foi obtida para os sete sistemas.

7.2.2.2 Execução

Após tomar conhecimento dos *bad-smells* de cada sistema, refatorações foram escolhidas e depois executadas por meio da ferramenta KDM-RE. Cada refatoração foi aplicada para tentar resolver e remover os *bad-smells* identificados e melhorar, assim, os atributos de qualidade QMOOD. As refatorações escolhidas para remover os *bad-smells* e melhorar os atributos de qualidade foram criteriosamente identificadas no catálogo de refatoração proposto por Fowler (1999), que descreve qual(is) refatoração(ões) aplicar para resolver um determinado *bad-smell*.

É importante salientar que as refatorações não foram aplicadas no diagrama de classe, ou seja, não foram aplicadas de forma totalmente manual e por intervenção de *Wizards*, como apresentado no Capítulo 6. As refatorações não foram executadas manualmente pelo engenheiro de software, uma vez que muitas refatorações foram aplicadas, o que iria demandar muito esforço e tempo para a condução do experimento. Dessa forma, foi criado um arquivo (*script*) com as refatorações, os parâmetros e os caminhos para cada instância do KDM, que representava os sete sistemas utilizados no experimento. Esse arquivo foi criado tendo como base os *bad-smell* identificados, assim, soube-se previamente qual refatoração aplicar, bem como seus parâmetros – portanto, a ferramenta KDM-RE executou as refatorações de forma semiautomática, diminuindo o tempo para a condução do experimento. Para cada refatoração, um limite de tempo de 5 minutos foi definido, e esse limite de tempo é importante para verificar se a ferramenta KDM-RE não entrou em *loop* infinito. As refatorações que foram aplicadas nos sete sistemas estão apresentadas na Tabela 19, bem como suas respectivas siglas.

Tabela 19 – Refatorações aplicadas no experimento.

Refatorações	Siglas	Sistemas	Siglas
Move Method	MM	Xerces-J	XJ
Move Field	MF	Jexel	JEX
Extract Class	EC	JFreeChart	JFC
Extract Interface	EI	JUnit	JUn
Move Class	MC	GanttProject	GP
Pull Up Field	PUF	ArtofIllusion	AOIL
Pull Up Method	PUM	JHotDraw	JHD
Push Down Field	PDF	—	—
Push Down Method	PDM	—	—

Na Tabela 20 é possível observar os dados relacionados com a quantidade de refatorações

aplicada em cada sistema. Nota-se que a refatoração `Move Method` foi a refatoração mais executada, aproximadamente 26.35%. Em seguida, a refatoração `Move Field` foi a segunda refatoração mais aplicada, aproximadamente 13.42%. A maioria das refatorações aplicadas está relacionada com mover (`Move Field` e `Move Method`) e extrair elementos (`Extract Class`); somando as refatorações `Move Method`, `Move Field` e `Extract Class` têm-se 51.06% das refatorações aplicadas, ou seja, mais da metade das refatorações. Esses dados estão diretamente relacionados aos *bad-smells* utilizados nesse experimento: *blob*, *data class* e *spaghetti code*. Por exemplo, para remover o *bad-smell* do tipo *blob* é necessário mover elementos de uma classe para outra classe, a fim de reduzir o número de funcionalidades e adicionar comportamento em outras classes. Similarmente, para remover o *bad-smell* do tipo *data class*, deve-se aplicar a refatoração `Move Method`. Ainda observando a Tabela 20, é possível notar que as refatorações menos aplicadas foram: `Push Up Method`, `Push Up Field` e `Move Class`, respectivamente.

Tabela 20 – Quantidade de refatorações aplicadas no experimento.

	Sistemas								
Refatorações	XJ	JEX	JFC	JES	GP	AOIL	JHD	Média	Porcentagem
MM	40	30	45	35	43	45	23	37.29	26.34%
MF	25	15	20	16	21	23	13	19.00	13.42%
EC	15	20	20	19	12	15	11	16.00	11.30%
EI	13	10	15	16	8	12	10	12.00	8.48%
MC	9	11	7	13	10	10	18	11.14	7.87%
PUF	12	14	5	8	12	11	9	10.14	7.16%
PUM	10	9	8	6	8	14	13	9.71	6.86%
PDF	17	13	15	10	9	16	8	12.57	8.88%
PDM	16	12	14	11	10	18	15	13.71	9.69%
TOTAL	157	134	149	134	133	164	120	—	100.00%

Após a aplicação das refatorações nos sete sistemas, a ferramenta EMF *Metrics* foi executada novamente para mensurar os atributos de qualidade QMOOD. A análise dos dados é apresentada na próxima seção.

7.2.3 Análise dos Dados do Experimento

Na Tabela 21, é possível observar os dados relacionados aos atributos (“reusabilidade”, “flexibilidade”, “facilidade de compreensão” e “eficácia”) de qualidade avaliados no experimento antes e após a aplicação das refatorações apresentadas na Tabela 20. A Tabela 21 possui sete colunas; a coluna denominada “Antes” representa as métricas mensuradas dos sistemas antes de aplicar as refatorações. A coluna “Depois” representa as métricas coletadas dos sistemas após a aplicação das refatorações e a coluna “Diferença” representa a discrepância das colunas “Antes” e “Depois”, ou seja, a diferença entre as métricas é calculada da seguinte forma: “Depois”-“Antes”. Os mesmos dados são plotados nos gráficos de barra mostrados na Figura 70.

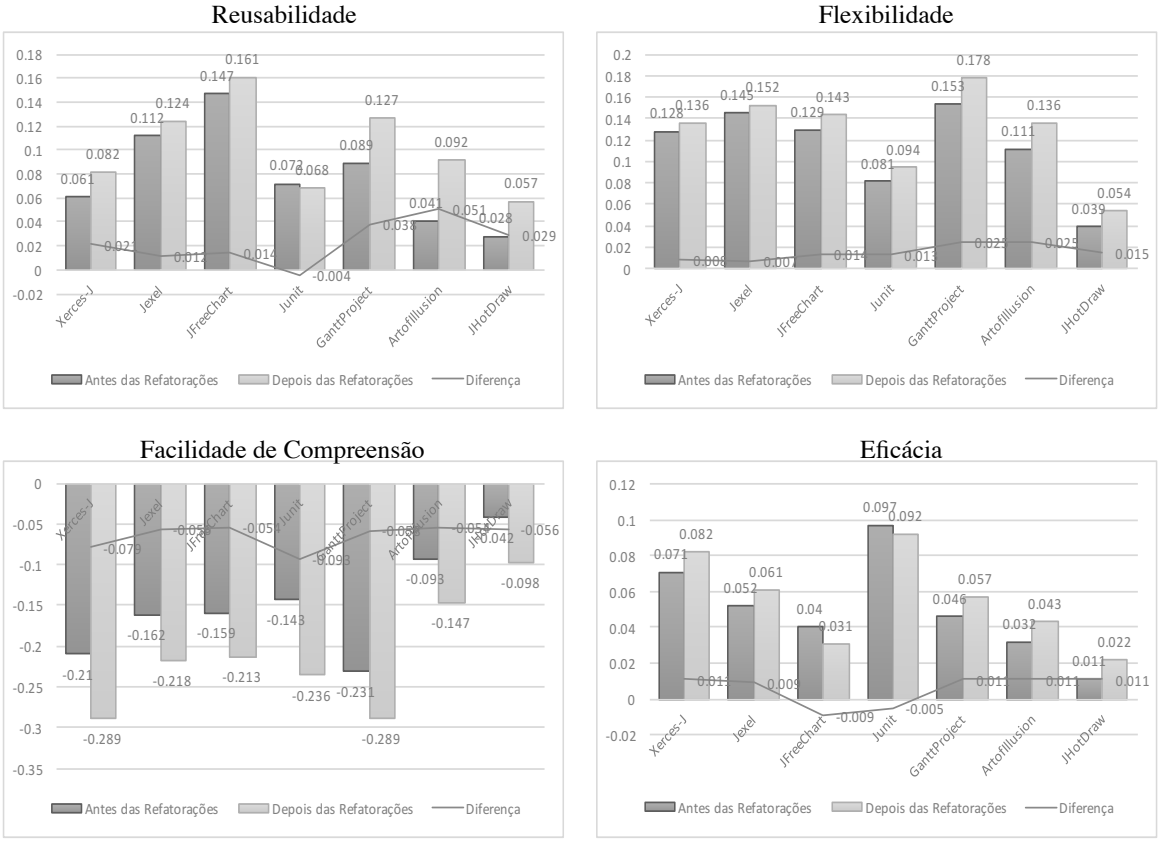
Tabela 21 – Dados coletados do experimento.

	Reusabilidade			Flexibilidade		
Sistemas	Antes	Depois	Diferença	Antes	Depois	Diferença
Xerces-J	0.061	0.082	0.021	0.128	0.136	0.008
Jexel	0.112	0.124	0.012	0.145	0.152	0.007
JFreeChart	0.147	0.161	0.014	0.129	0.143	0.014
JUnit	0.072	0.068	-0.004	0.081	0.094	0.013
GanttProject	0.089	0.127	0.038	0.153	0.178	0.025
ArtofIllusion	0.041	0.092	0.051	0.111	0.136	0.025
JHotDraw	0.028	0.057	0.029	0.039	0.054	0.015
Média	0.078	0.094	0.023	0.112	0.127	0.015
Porcentagem	43.62%	56.38%	12.77%	46.81%	53.19%	6.37%
	Facilidade de Compreensão			Eficácia		
Sistemas	Antes	Depois	Diferença	Antes	Depois	Diferença
Xerces-J	-0.21	-0.289	-0.079	0.071	0.082	0.011
Jexel	-0.162	-0.218	-0.056	0.052	0.061	0.009
JFreeChart	-0.159	-0.213	-0.054	0.04	0.031	-0.009
JUnit	-0.143	-0.236	-0.093	0.097	0.092	-0.005
GanttProject	-0.231	-0.289	-0.058	0.046	0.057	0.011
ArtofIllusion	-0.093	-0.147	-0.054	0.032	0.043	0.012
JHotDraw	-0.042	-0.098	-0.054	0.011	0.022	0.011
Média	-0.148	-0.212	-0.064	0.049	0.055	0.005
Porcentagem	41.11%	58.89%	17.79%	47.35%	52.65%	5.29%

Nota-se que todos os atributos de qualidade foram melhorados após a aplicação das refatorações: (*i*) "reusabilidade" (antes = 43.62%, depois = 56.38% e diferença = 12.77%), "flexibilidade" (antes = 46.81%, depois = 53.19% e diferença = 6.37%), "facilidade de compreensão" (antes = 41.11%, depois = 58.89% e diferença = 17.79%) e "eficácia" (antes = 47.35%, depois = 52.65% e diferença = 5.29%). É importante observar que o atributo de qualidade "facilidade de compreensão" foi o que alcançou o maior ganho, aproximadamente 18%. Por outro lado, o atributo de qualidade "eficácia" teve o menor ganho, alcançando 5.29%. Isso deve-se principalmente às refatorações aplicadas. Por exemplo, a maioria das refatorações aplicadas (`Move Method`, `Move Field` e `Extract Class` - ver Tabela 20) aumenta o acoplamento (DCC), coesão (CAM) e tamanho do projeto (DSC), que são métricas utilizadas para calcular o atributo de qualidade Facilidade de Compreensão.

Além disso, pode-se também observar na Figura 70 que o sistema JHotDraw produziu o menor aumento para os quatro atributos de qualidade. Acredita-se que a principal razão disso é que JHotDraw é conhecido na literatura por seguir as melhores práticas de projeto e implementação (KESSENTINI; VAUCHER; SAHRAOUI, 2010), assim, poucas refatorações precisaram ser aplicadas nesse sistema.

Figura 70 – Gráficos resultantes antes e após as refatorações.



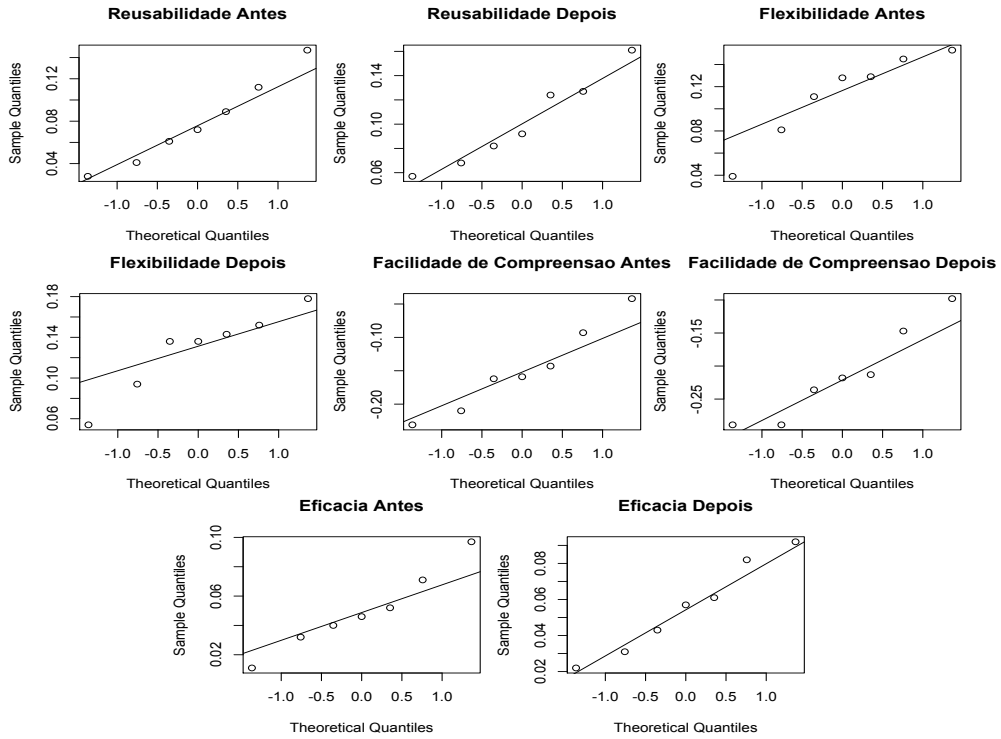
Fonte: Elaborada pelo autor.

7.2.3.1 Teste das Hipóteses

Nesta seção, são apresentados os resultados dos testes estatísticos aplicados sobre os dados coletados no experimento. Antes de aplicar qualquer teste estatístico, é importante verificar se um conjunto de dados segue, ou não, uma distribuição normal. Dessa forma, para os conjuntos de dados (antes e depois das refatorações) dos atributos de qualidade "reusabilidade", "flexibilidade", "facilidade de compreensão" e "eficácia" apresentados na Tabela 21, aplicou-se o teste **Shapiro-Wilk** para verificar se os dados seguem uma distribuição normal. Além disso, para cada conjunto de dados também foram criados gráficos de probabilidade denominados *Q-Q Plot*, ver Figura 71. Como pode ser observado na Tabela 21, o p-valor resultante para cada atributo de qualidade é: reusabilidade (antes = 0.8988, depois = 0.7245), flexibilidade (antes = 0.3298, depois = 0.416), facilidade de compreensão (antes = 0.8137, depois = 0.4716) e eficácia (antes = 0.9355, depois = 0.8362). É visto que todos os p-valor dos testes foram superior a 0.05, então, pode-se afirmar, com nível de confiança de 95%, que os dados seguem uma distribuição normal. Isso também pode ser verificado nos gráficos mostrados na Figura 71. Nota-se que em tais gráficos os pontos estão formados pelo quantis amostrais e o pontos alinham-se nas respectivas retas, com isso, pode-se afirmar que os dados estão normalizados.

Como os dados estão normalizados (ver Tabela 22), o *Paried T-Test* foi aplicado sobre

Figura 71 – Gráficos resultantes do teste de normalidade.



Fonte: Elaborada pelo autor.

Tabela 22 – *Shapiro-Wilk* aplicado para verificar se os dados seguem uma distribuição normal.

Reusabilidade			
Antes		Depois	
w=0.97	p-valor=0.8988	w=0.9494	p-valor=0.7245
Flexibilidade			
Antes		Depois	
w=0.8998	p-valor=0.3298	w=0.9129	p-valor=0.416
Facilidade de Compreensão			
Antes		Depois	
w=0.9594	p-valor=0.8137	w=0.9203	p-valor=0.4716
Eficácia			
Antes		Depois	
w=0.9756	p-valor=0.9355	w=0.9621	p-valor=0.8362

os dados para verificar as hipóteses da *QP* do experimento (Seção 7.2). Como mostrado na Tabela 23, todos atributos de qualidade foram p-valor < 0.05; então, com nível de confiança de 95%, existem evidências de diferença entre os atributos de qualificação antes e após a aplicação das refatorações. Portanto, podem-se refutar as hipóteses nulas $H1_0$, $H2_0$ e $H3_0$ e as hipóteses alternativas foram aceitas. O atributo de qualidade "eficácia" tem uma relação com o atributo de qualidade "facilidade de compreensão", pois quando um é melhorado o outro tende a piorar. Dessa maneira, para o atributo de qualidade "eficácia" não se pode refutar a hipótese nula, porque

Tabela 23 – *Paried T-Test* aplicado sobre os dados para verificar as hipóteses da *QP*.

Atributo de Qualidade	T	P-valor	Comentário
Reusabilidade	-3.3498	0.01542	Refuta-se $(H1_0)$ com significância de 5%
Flexibilidade	-5.5602	0.001433	Refuta-se $(H2_0)$ com significância de 5%
Facilidade de Compreensão	11.0194	3.322e-05	Refuta-se $(H3_0)$ com significância de 5%
Eficácia	-1.6951	0.141	Não se refuta $(H4_0)$, pois 0.141 > 0.05

os dados não se demostraram estatisticamente significantes para refutá-la, ou seja, não houve evidência suficientemente forte para provar que a hipótese nula era falsa.

7.2.4 Ameaças à Validade do Experimento

São apresentados, aqui, os itens que podem afetar os valores e a conclusão do experimento. O experimento é afetado pelas ameaças à validade listadas abaixo.

Validade Externa: refere-se à generalidade do experimento. O experimento, foi conduzido em sete sistemas diferentes de código aberto amplamente utilizados, pertencentes a diferentes domínios e com tamanhos diferentes. No entanto, não é possível afirmar que os resultados podem ser generalizados para todas as aplicações Java instanciadas para o metamodelo KDM, bem como para outras linguagens de programação, e outros engenheiros de software. Outra ameaça pode ser o número limitado de sujeitos/sistemas estudados, o que ameaça externamente a generalização dos resultados do presente trabalho. Replicações futuras desse estudo são necessárias para confirmar o resultado da abordagem.

Validade por Construção: está preocupada com a relação entre a teoria e o que é observado. Como a ferramenta KDM-RE no seu estado atual não identifica *bad-smells*, a escolha das refatorações a serem aplicadas foi totalmente dependente das ferramentas Jdeodorant e inFusion. Portanto, não se pode eliminar ameaças à validade relacionadas aos problemas durante a identificação de *bad smells*.

Validade de Conclusão: está relacionada com a precisão das métricas empregadas durante o experimento. Para mitigar essa ameaça, utilizaram-se métricas já consolidadas na literatura, tais como QMOOD.

7.3 Considerações Finais

Neste capítulo foi apresentado o experimento realizado para avaliar as refatorações criadas para o metamodelo KDM. Esse experimento foi previamente planejado com base no objetivo pretendido, nos recursos a serem utilizados e nas variáveis. Além disso, testes estatísticos foram aplicados para obter maior confiabilidade nos resultados obtidos.

Sete sistemas foram utilizados no experimento, os quais foram transformados em instâncias KDM por meio da ferramenta MoDisco. Em seguida, a ferramenta EMF *Metrics* foi

usada para mensurar os atributos de qualidade QMOOD. Como já salientado, a ferramenta KDM-RE no seu estado atual não identifica *bad-smells*. Dessa forma, as ferramentas Jdeodorant e inFusion foram aplicadas nos sete sistemas para identificar um conjunto de *bad-smells*. Após tomar conhecimento dos *bad-smells*, refatorações foram escolhidas e depois executadas por meio da ferramenta KDM-RE. Note que, as refatorações não foram aplicadas no diagrama de classe, ou seja, não foram aplicadas de forma totalmente manual e por intervenção de *Wizards*, como apresentado no Capítulo 6. Dessa forma, foi criado um arquivo (*script*) com as refatorações, os parâmetros e os caminhos para cada instância do KDM, que representava os sete sistemas utilizados no experimento.

A refatoração `Move Method` foi a refatoração mais executada, aproximadamente 26.35%. Em seguida, a refatoração `Move Field` foi a segunda refatoração mais aplicada, aproximadamente 13.42%. A maioria das refatorações aplicadas está relacionada com mover (`Move Field` e `Move Method`) e extrair elementos (`Extract Class`); somando as refatorações `Move Method`, `Move Field` e `Extract Class` têm-se 51.06% das refatorações aplicadas, ou seja, mais da metade das refatorações. Após a aplicação das refatorações nos sete sistemas, a ferramenta EMF *Metrics* foi executada novamente para mensurar os atributos de qualidade QMOOD.

Diante disso, foi visto que nos sete sistemas todos os atributos de qualidade foram melhorados após a aplicação das refatorações: (*i*) "reusabilidade" (antes = 43.62%, depois = 56.38% e diferença = 12.77%), "flexibilidade" (antes = 46.81%, depois = 53.19% e diferença = 6.37%), "facilidade de compreensão" (antes = 41.11%, depois = 58.89% e diferença = 17.79%) e "eficácia" (antes = 47.35%, depois = 52.65% e diferença = 5.29%). É importante observar que o atributo de qualidade "facilidade de compreensão" foi o que alcançou o maior ganho, aproximadamente 18%. Por outro lado, o atributo de qualidade "eficácia" teve o menor ganho, alcançando 5.29%. Isso deve-se principalmente às refatorações aplicadas. Por exemplo, a maioria das refatorações aplicadas (`Move Method`, `Move Field` e `Extract Class`) aumenta o acoplamento (DCC), coesão (CAM) e tamanho do projeto (DSC), que são métricas utilizadas para calcular o atributo de qualidade Facilidade de Compreensão.

CAPÍTULO

8

CONCLUSÕES

8.1 Considerações Finais do Projeto de Doutorado

Neste projeto de doutorado, almejou-se buscar soluções para facilitar a aplicação e o reúso de refatorações para o metamodelo KDM. Para isso, foi definida uma abordagem com o foco na criação e disponibilização de refatorações para o KDM, bem como um apoio ferramental que permite aplicá-las em diagramas de classe da UML. A abordagem possui dois principais passos: (*i*) o primeiro envolve diretrizes que apoiam o engenheiro de modernização durante a implementação de refatorações para o KDM; (*ii*) o segundo consiste na especificação das refatorações por meio da criação de instâncias do metamodelo SRM e posterior disponibilização delas em um repositório.

Além disso, um apoio computacional, denominado KDM-RE, também foi desenvolvido para auxiliar o engenheiro de software durante o processo de modernização. Esse apoio computacional é um *plug-in* do Ambiente de Desenvolvimento Eclipse, o qual foi dividido em três principais módulos: (*i*) o primeiro engloba um conjunto de *Wizards* que apoia o engenheiro de software na aplicação das refatorações em diagramas de classe UML; (*ii*) o segundo consiste em um apoio à importação e reúso de refatorações disponíveis no repositório; (*iii*) o terceiro compreende um módulo de propagação de mudanças que permite manter modelos internos do KDM sincronizados.

Como parte da pesquisa, um experimento foi conduzido com o objetivo de testar e avaliar as vantagens de utilizar a abordagem aqui apresentada. Os resultados mostram que a abordagem pode trazer benefícios para o engenheiro de software durante a atividade de aplicação de refatorações em sistemas, representados pelo metamodelo KDM.

Nas demais seções deste capítulo, são apresentados os seguintes tópicos: na Seção 8.2, as contribuições desta tese são descritas; na Seção 8.3, as limitações são destacadas; na Seção 8.4, os possíveis trabalhos futuros são salientados; e na Seção 8.5, as publicações resultantes durante

o período de doutorado são destacadas.

8.2 Contribuições desta Tese

A principal contribuição desta tese é fornecer para a ADM soluções para a criação, aplicação e reúso de refatorações para o metamodelo KDM. As contribuições mais específicas são os passos propostos para criar refatorações e restrições para o metamodelo KDM, o metamodelo SRM que fornece uma terminologia comum, padronizada e independente de linguagem para a especificação de refatorações e o apoio computacional que permite aplicar refatorações em diagramas UML que são representações gráficas de modelos KDM.

A abordagem para criar refatorações torna a criação de refatorações para o metamodelo KDM um processo sistemático e guiado, facilitando a tarefa do engenheiro de modernização e procurando garantir que as refatorações desenvolvidas estejam estruturadas corretamente com base nas metaclasses do KDM. Acredita-se que a abordagem apresentada tem potencial para conduzir engenheiros de modernização durante a criação de refatorações para o metamodelo KDM. Embora nenhuma avaliação tenha sido realizada neste sentido, a dificuldade na escrita de refatorações (ATL) e restrições (OCL) são evidentes, uma vez que engenheiros não estão familiarizados com linguagens de transformação de modelo e linguagens de restrição (SENDALL; KOZACZYNSKI, 2003), assim, o fornecimento de passos sistemáticos e *templates* auxiliam e guiam engenheiros de modernização durante a criação de refatorações e restrições para o KDM.

Sobre o metamodelo SRM, é possível concluir que ele fornece uma terminologia comum, padronizada e independente de linguagem para a especificação de refatorações. A ideia é que ferramentas de modernização adotem esse metamodelo como base para suas refatorações, permitindo assim o reúso de instâncias de refatorações entre ferramentas. Esse metamodelo possui um conjunto de metaclasses que define meta-atributos específicos para representar informações (metadados) de refatoração, auxiliando, assim, o compartilhamento das refatorações de forma intuitiva entre os engenheiros. Dessa forma, o SRM permite a interoperabilidade entre ferramentas de modernização, desde que elas adotem o metamodelo supracitado, e viabiliza a especificação de refatorações em um formato padronizado por parte dos engenheiros de modernização. O SRM segue a mesma proposta de outros metamodelos definidos na ADM e está totalmente integrado com o metamodelo KDM. Uma instância do metamodelo de refatoração contém metadados que representam uma refatoração escrita para ser executada em uma instância do metamodelo KDM. O algoritmo de refatoração, bem como as restrições, podem ser especificadas por meio do SRM. Além disso, se ferramentas de modernização utilizarem internamente o SRM elas podem possuir *templates* que são capazes de gerar automaticamente algoritmos de refatoração para o KDM.

Outra contribuição são os produtos gerados pela pesquisa realizada. A ferramenta KDM-RE foi implementada como um conjunto de *plug-ins* para o Ambiente de Desenvolvimento Eclipse IDE. Cada *plug-in* representa um módulo da KDM-RE: (*i*) módulo de refatoração, (*ii*)

módulo do SRM e (*iii*) módulo de sincronização. O primeiro módulo permite a aplicação de refatorações graficamente por meio de diagramas de classe da UML, porém, as refatorações são realizadas transparentemente no metamodelo KDM e, posteriormente, replicadas nos diagramas de classe da UML. O segundo módulo contém uma DSL que foi desenvolvida para facilitar a instanciação do metamodelo SRM. Optou-se por desenvolver uma DSL, pois o processo de instanciação do SRM não é trivial, engenheiros de modernização devem estar familiarizados como as particularidades das refatorações (por exemplo, qual(is) é (são) o(s) pré-requisito(s) para a execução de uma refatoração) e como/onde utilizar e programar tais refatorações. No entanto, com o uso da DSL, engenheiros de modernização são forçados a seguir a semântica correta para a instanciação do metamodelo SRM. Adicionalmente, o segundo módulo da KDM-RE também permite que instâncias do metamodelo SRM sejam enviadas para um repositório, fazendo com que as refatorações definidas por engenheiros de modernização possam ser reutilizadas. O terceiro módulo é responsável por manter consistente e propagar mudanças após a aplicação de refatorações em instâncias do metamodelo KDM. Regras pré-definidas em ATL foram criadas e são disparadas após a aplicação de refatorações em instâncias do metamodelo KDM. Pretende-se disponibilizar a KDM-RE em um repositório de projetos de software, como por exemplo o Sourceforge e GitHub.

Com relação à arquitetura da KDM-RE, optou-se por permitir que o engenheiro de software aplique refatorações nos diagramas UML, mas ao invés dessas refatorações serem executadas na instância UML correspondente, são diretamente executadas na instância do KDM que está sendo exibido graficamente. Pode-se imaginar que uma estratégia diferente seja aplicar as refatorações na instância UML e depois transformá-la para uma instância do KDM. Entretanto, a estratégia adotada foi escolhida por causa da semelhança das metaclasses da UML e do KDM, permitindo que um mapeamento implícito fosse mantido durante as transformações. A geração da instância UML refatorada é então feita com o apoio de um *plug-in* existente fornecido pelo MoDisco (BRUNELIERE *et al.*, 2010).

Durante a condução desta pesquisa também foi possível concluir que a UML mostra-se limitada mediante a tarefa de representar graficamente o KDM. Idealmente, deveriam existir visualizações gráficas mais adequadas que permitissem uma análise mais clara e detalhada de todas as visões do KDM. Porém, quando engenheiros de software estiverem focados em refatorações de baixo nível, não há problema em usar os diagramas UML.

Outra contribuição desta tese é o experimento conduzido, no qual sete sistemas foram escolhidos para aplicar um conjunto de refatorações utilizando o apoio computacional KDM-RE. O principal objetivo desse experimento foi verificar se após a aplicação de um conjunto de refatorações com base em *bad-smells* já identificados, as refatorações criadas para o metamodelo KDM melhoram os sistemas em termos de atributos de qualidade. Quatro atributos de qualidade foram considerados no experimento: “reusabilidade”, “flexibilidade”, “facilidade de compreensão” e “eficácia”. Os resultados mostraram que para todos os atributos de qualidade, exceto “eficácia”,

uma melhora foi obtida após a aplicação de um conjunto de refatorações.

Com base em todo o trabalho realizado foi possível perceber que o ideal do OMG com a ADM é possível de ser atingido mas ainda há vários desafios a serem superados. A existência de ferramentas de modernização que adotem o KDM e os demais metamodelos da ADM como metamodelos base depende bastante de evidências sobre a qualidade do mesmo. Além disso, organizações que já possuem suas próprias ferramentas com modelos proprietários precisam ter evidências muito claras dos benefícios de trocar pelo KDM. Um dos fatores primordiais para que organizações passem a utilizar o KDM em suas ferramentas é a quantidade de recursos disponíveis para esse metamodelo. Quanto maior for a quantidade de repositórios de recursos para o KDM, maior será a motivação das organizações em adotarem esse metamodelo. Porém, acredita-se, que as contribuições enfatizadas nesta tese contribuem para o avanço do ideal do OMG durante o processo de modernização, principalmente, no contexto da ADM e do KDM. Em especial, a abordagem aqui apresentada contribui com as perspectivas do engenheiro de modernização e o apoio computacional auxilia o engenheiro de software. O engenheiro de modernização é capaz de utilizar a abordagem e o metamodelo SRM para criar e disponibilizar refatorações no contexto do metamodelo KDM e, possivelmente, aumentar o nível de reusabilidade e interoperabilidade de refatorações em nível de modelo. O engenheiro de software é beneficiado porque pode utilizar as facilidades providas pelo apoio computacional KDM-RE e, com isso, aplicar refatorações graficamente por meio de diagramas UML e reutilizar refatorações para o KDM.

8.3 Limitações

A abordagem e o apoio computacional desenvolvidos nesta tese possuem as seguintes limitações:

- A abordagem de criação de refatorações para o KDM descreve como criar refatorações para o metamodelo KDM. Contudo, a forma como o mecanismo, bem como as pré- e pós-condições dependem de *templates* e devem ser criados totalmente manualmente. Neste projeto de doutorado, foram realizados exemplos práticos com a utilização de ATL e OCL para implementar o mecanismo e as asserções (pré- e pós-condições), respectivamente;

- O metamodelo SRM viabiliza o reúso das refatorações dentro de ferramentas de modernização, propiciando a interoperabilidade entre as mesmas. No entanto, como o SRM é baseado no KDM, podem-se apenas especificar refatorações para os elementos estruturais representados pelas metaclasses contidas no KDM;

- O apoio computacional KDM-RE contém um módulo que provê uma DSL para auxiliar a instanciação do metamodelo SRM. Essa DSL aumenta o nível de abstração do processo de instanciação de refatorações, escondendo determinadas complexidades. O uso dessa DSL guia o engenheiro de modernização durante a especificação de uma determinada

refatoração, norteando quais metaclasses devem ser especificadas. No entanto, a semântica de uma determinada refatoração utilizando essa DSL continua sendo responsabilidade do engenheiro de modernização;

- O apoio computacional KDM-RE foi desenvolvido para ser utilizado no ambiente de desenvolvimento Eclipse, portanto, o conhecimento sobre esse ambiente é um pré-requisito necessário para o manuseio desse apoio computacional.

8.4 Sugestões de Trabalhos Futuros

Refatoração, especialmente refatoração em nível de modelo, é uma disciplina relativamente nova e uma área de pesquisa ativa de acordo com o OMG (OMG, 2015; ADM, 2012). Esta tese considera a aplicação de refatorações no contexto da ADM e do metamodelo KDM de forma integrada, subjacente e automatizada. Assim, o trabalho desenvolvido nesta tese deve passar por vários ajustes com base na experiência substancial de aplicações práticas para obter relevância na indústria. Com base no MS conduzido e apresentado no Capítulo 3 e no trabalho apresentado nesta tese, há muitas pesquisas que podem ser realizadas no futuro pelo autor e por outros pesquisadores do grupo de pesquisa em engenharia de software do ICMC.

Uma dessas pesquisas seria realizar mais estudos de caso com outros programas, visando comprovar se os mesmos resultados, ou resultados semelhantes, serão obtidos. Além disso, os passos para criar refatorações para o metamodelo KDM, os quais foram definidos no Capítulo 4, necessitam de estudos de caso que envolvam a criação de um conjunto de refatorações. No contexto do metamodelo SRM, também seria importante aplicar estudos de caso envolvendo uma população de novas refatorações para verificar se o mesmo pode ser utilizado para prover a especificação e a reutilização de refatorações de forma eficiente. Também seria interessante como trabalho futuro a construção de uma ferramenta de apoio à abordagem apresentada no Capítulo 4. Essa ferramenta poderia auxiliar no acompanhamento da abordagem, na documentação e criação de refatorações para o metamodelo KDM de forma totalmente automática.

Outro trabalho futuro seria implementar algum mecanismo de suporte à evolução do metamodelo SRM. Isso se mostra necessário quando novas refatorações muito específicas precisam ser criadas. Também pode-se implementar uma forma de identificar automaticamente *bad-smells* utilizando o metamodelo KDM como base.

A ferramenta KDM-RE utiliza apenas o diagrama de classe da UML como interface (*front-end*) para a aplicação de refatorações. Outros diagramas da UML são raramente utilizados na literatura durante atividades de refatorações. O uso combinado de múltiplos diagramas da UML durante a atividade de refatoração poderiam auxiliar o engenheiro de software a entender melhor a estrutura do sistema, bem como compreender também seu comportamento. Por exemplo, além do diagrama de classe da UML, diagramas de caso de uso e sequencia poderiam ser utilizados durante a atividade de refatoração como interface.

8.5 Publicações Resultantes

Durante este projeto de doutorado foi possível publicar um conjunto de artigos científicos. Tais artigos são resultados de trabalhos totalmente relacionados a esta pesquisa, bem como publicações realizadas em parcerias com outros pesquisadores. Acredita-se que estas também são importantes como parte da experiência de uma pesquisa em nível de doutorado. Dessa forma, nesta seção os artigos científicos são separados na seguinte ordem: (*i*) publicações totalmente relacionadas a este projeto, bem como ADM e KDM e (*ii*) publicações realizadas em parcerias.

8.5.1 Publicações totalmente relacionado a este projeto, bem como ADM e KDM.

Nesta seção, as publicações diretamente relacionadas a este projeto são listadas. Além dessas publicações, aqui também são destacadas as publicações realizadas com parceiros durante o doutorado e que são totalmente relacionados com ADM e KDM. Tais publicações foram importante para auxiliar o doutorando a entender e aprimorar o seu conhecimento sobre ADM e KDM e realizar sua pesquisa.

- **Trabalho completo publicado em revista**

  1. SANTIBÁÑEZ, DANIEL S. M. ; **DURELLI, RAFAEL** ; DE CAMARGO, VALTER . "*A Combined Approach for Concern Identification in KDM models*". **Journal of the Brazilian Computer Society**, 2015.
     - Nível de contribuição: Alto - Auxiliou-se na elaboração do apoio ferramental, bem como na escrita e na estrutura do artigo;

- **8 Trabalhos completos publicados em anais de congressos/workshops**

  1. **DURELLI, R. S.**; SANTIBANEZ, D. S. M. ; ANQUETIL, N. ; DELAMARO, M. E. ; CAMARGO, V. V. "*A Systematic Review on Mining Techniques for Crosscutting Concerns*". **The ACM Symposium on Applied Computing (SAC)**, 2013, Coimbra.
     - Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;
  2. SANTIBÁÑEZ, DANIEL S. M. ; **DURELLI, RAFAEL S.**; CAMARGO, V. V. "*CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process*". **XXVII Simpósio Brasileiro de Engenharia de Software - XXVII Sessão de Ferramenta**, 2013, Brasília.
     - Nível de contribuição: Alto - Auxiliou-se na elaboração do apoio ferramental, bem como na escrita e na estrutura do artigo;

3. SANTIBANEZ, D. S. M. ; **DURELLI, RAFAEL S.**; CAMARGO, V. V. "*A Combined Approach for Concern Identification in KDM models*". **Latin American Workshop on Aspect-Oriented Software Development (LA-WASP)**, 2013, Brasília. Congresso Brasileiro de Software: Teoria e Prática (CBSoft), 2013.

   – Nível de contribuição: Alto - Auxiliou-se na elaboração do apoio ferramental, bem como na escrita e na estrutura do artigo;

4. **DURELLI, R. S.** ; SANTIBANEZ, D. S. M. ; DELAMARO, MÁRCIO E. ; CAMARGO, V. V. "*Towards a Refactoring Catalogue for Knowledge Discovery Metamodel*". **IEEE International Conference on Information Reuse and Integration**, 2014, São Francisco.

   – Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;

5. **DURELLI, R. S.** ; SANTIBANEZ, D. S. M. ; MARINHO, B. S. ; HONDA, R. R. ; DELAMARO, M. E. ; ANQUETIL, N. ; CAMARGO, V. V. "*A Mapping Study on Architecture-Driven Modernization*". **IEEE International Conference on Information Reuse and Integration**, 2014, São Francisco.

   – Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;

6. MARINHO, B. S. ; CAMARGO, V. V. ; HONDA, R. R. ; **DURELLI, R. S.** . "*KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel*". **28th Brazilian Symposium on Software Engineering (SBES)**, 2014, Maceió.

   – Nível de contribuição: Alto - Auxiliou-se na elaboração da extensão, bem como na escrita e na estrutura do artigo;

7. MARINHO, B. S. ; **DURELLI, RAFAEL S.** ; HONDA, R. R. ; CAMARGO, V. V. . "*Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization*". **11th Workshop on Software Modularity (WMod) – Brazilian Conference on Software: theory and practice**, 2014, Maceió.

   – Nível de contribuição: Alto - Auxiliou-se na elaboração da extensão, bem como na escrita e na estrutura do artigo;

8. **DURELLI, RAFAEL S.** ; MARINHO, B. S. ; HONDA, R. R. ; DELAMARO, MÁRCIO E. ; CAMARGO, V. V. . "*KDM-RE: A Model-Driven Refactoring Tool for KDM*". **II Workshop on Software Visualization, Evolution and Maintenance – Brazilian Conference on Software: theory and practice**, 2014, Maceió.

   – Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;

8.5.2 Publicações realizadas em parcerias

Nesta seção, os trabalhos realizados em parcerias durante o doutorado são apresentados. Embora tais artigos não estejam diretamente relacionados com o tema ADM e KDM, todos foram de alguma forma importantes para a formação do doutorando, bem como para a criação de colaboração.

- **2 trabalhos completos publicados como capítulo de livro**

  1. Viana, Matheus ; Penteado, Rosângela ; Prado, Antônio do ; **Durelli, Rafael** . "*Developing Frameworks from Extended Feature Models*". Advances in Intelligent Systems and Computing. 1ed.: Springer International Publishing, 2014, v. 263, p. 263-284.
     - Nível de contribuição: Médio - Auxiliou-se na escrita e na estrutura do artigo;
  2. Júnior, Paulo Afonso Parreira ; Penteado, Rosângela Dellosso ; Viana, Matheus Carvalho ; **Durelli, Rafael Serapilha** ; DE CAMARGO, VALTER VIEIRA ; Costa, Heitor Augustus Xavier . "*Reengineering of Object-Oriented Software into Aspect-Oriented Ones Supported by Class Models*". Lecture Notes in Business Information Processing. 1ed.: Springer International Publishing, 2014, v. 190, p. 296-313.
     - Nível de contribuição: Médio - Auxiliou-se na escrita e na estrutura do artigo;

- **Trabalho completo publicado em revista**

  1. GOTTARDI, THIAGO ; **DURELLI, RAFAEL** ; LÓPEZ, ÓSCAR ; DE CAMARGO, VALTER . "*Model-based reuse for crosscutting frameworks: assessing reuse and maintenance effort*". **Journal of Software Engineering Research and Development**, 2013.
     - Nível de contribuição: Alto - Auxiliou-se na elaboração do apoio ferramental, bem como na escrita e na estrutura do artigo;

- **10 trabalhos completos publicados em anais de congressos/*workshops***

  1. **DURELLI, R. S.**; DURELLI, V. H. S. . ""*A Systematic Mapping Study on Formal Methods Applied to Crosscutting Concerns Mining*"". **IX Experimental Software Engineering Latin American Workshop (ESELAW)**, 2012, Buenos Aires.
     - Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;
  2. **DURELLI, R. S.**; DURELLI, V. H. S. . ""*F2MoC: A Preliminary Product Line DSL for Mobile Robots*"". **Simpósio Brasileiro de Sistemas de Informação (SBSI)**, 2012, São Paulo.
     - Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;

3. Gottardi; **DURELLI, R. S.** ; PASTOR, O. L. ; CAMARGO, V. V. . "*Model-Based Reuse for Crosscutting Frameworks: Assessing Reuse and Maintainability Effort*". **Simpósio Brasileiro de Engenharia de Software (SBES)**, 2012, Natal.

   – Nível de contribuição: Alto - Auxiliou-se na elaboração do apoio ferramental, bem como na escrita e na estrutura do artigo;

4. **DURELLI, R. S.**; Gottardi ; CAMARGO, V. V. . "*CrossFIRE: An Infrastructure for Storing Crosscutting Framework Families and Supporting their Model-Based Reuse*". **XXVI Simpósio Brasileiro de Engenharia de Software - XXVI Sessão de Ferramenta**, 2012, Natal.

   – Nível de contribuição: Alto - O doutorando é o principal investigador e conduziu a elaboração do artigo com ajuda de colaboradores;

5. PARREIRA JUNIOR, P. A.; VIANA, M. C. ; **DURELLI, R. S.** ; CAMARGO, V. V. ; COSTA, H. A. X. ; PENTEADO, R. A. D. "*Concern-Based Refactorings Supported by Class Models to Reengineer Object-Oriented Software into Aspect-Oriented Ones*". **International Conference on Enterprise Information Systems (ICEIS)**, 2013, ANGERS-França.

   – Nível de contribuição: Baixo - Auxiliou-se na estrutura do artigo;

6. VIANA, M. C. ; **DURELLI, R. S.** ; PENTEADO, R. A. D. ; PRADO, A. F. "*F3: From features to frameworks*". **International Conference on Enterprise Information Systems (ICEIS)**, 2013, ANGERS-França.

   – Nível de contribuição: Médio - Auxiliou-se na escrita e na estrutura do artigo;

7. VIANA, M. C. ; PENTEADO, R. A. D. ; PRADO, A. F. ; **DURELLI, R. S**. . "*An Approach to Develop Frameworks from Feature Models*". **International Conference on Information Reuse and Integration**, 2013, São Francisco.

   – Nível de contribuição: Médio - Auxiliou-se na escrita e na estrutura do artigo;

8. VIANA, M. C. ; PENTEADO, R. A. D. ; PRADO, A. F. ; **DURELLI, RAFAEL S**. "*F3T: From Features to Frameworks Tool*". **XXVII Simpósio Brasileiro de Engenharia de Software (SBES)**, 2013, Brasília.

   – Nível de contribuição: Médio - Auxiliou-se na escrita e na estrutura do artigo;

9. PINTO, Victor Hugo S. C. ; **DURELLI, R. S.** ; OLIVEIRA, A. L. ; CAMARGO, V. V. "*Evaluating the Effort for Modularizing Multiple-Domain Frameworks towards Framework Product Lines with Aspect-Oriented Programming and Model-Driven Development*". **International Conference on Enterprise Information Systems (ICEIS)**, 2014, Lisboa.

   – Nível de contribuição: Baixo - Auxiliou-se na elaboração do experimento;

10. DIAS, D. R. C. ; **DURELLI, R. S.** ; BREGA, J. R. F. ; GNECCO, B. B ; TREVELIN, L. C. ; GUIMARAES, M. P. "*Data Network in Development of 3D Collaborative Virtual Environments: A Systematic Review*". **The 14th International Conference on Computational Science and Applications (ICCSA)**, 2014, Guimarães.
    - Nível de contribuição: Baixo - Auxiliou-se na elaboração do protocolo da revisão sistemática;

REFERÊNCIAS

ADM. **Architecture-driven modernization**. 2012. Disponível em: <http://www.omgwiki.org/admtf/doku.php?id=start>. Acesso em: 23/03/2015. Citado 7 vezes nas páginas 31, 40, 53, 54, 55, 65 e 213.

____. **Structured Metrics Metamodel - SMM**. 2016. Disponível em: <http://www.omg.org/spec/SMM/>. Acesso em: 23/01/2016. Citado na página 157.

____. **Structured Patterns Metamodel Standard - SPMS**. 2016. Disponível em: <http://www.omg.org/spec/SPMS/>. Acesso em: 23/01/2016. Citado 2 vezes nas páginas 157 e 158.

ALLILAIRE, F.; BéZIVIN, J.; JOUAULT, F.; KURTEV, I. Atl - eclipse support for model transformation. In: **IN: PROC. OF THE ECLIPSE TECHNOLOGY EXCHANGE WORKSHOP (ETX) AT ECOOP**. [S.l.: s.n.], 2006. Citado 4 vezes nas páginas 50, 51, 138 e 163.

AMMAR, L. B.; MAHFOUDHI, A. Usability driven model transformation. In: **Human System Interaction (HSI), 2013 The 6th International Conference on**. [S.l.: s.n.], 2013. p. 110–116. Citado na página 40.

ANDROMDA. **AndroMDA**. 2015. Disponível em: <http://www.andromda.org/>. Acesso em: 01/02/2015. Citado na página 50.

ARENDT, T.; TAENTZER, G. Integration of smells and refactorings within the eclipse modeling framework. In: **Proceedings of the Fifth Workshop on Refactoring Tools**. [S.l.: s.n.], 2012. p. 8–15. Citado na página 201.

____. A tool environment for quality assurance based on the eclipse modeling framework. **Automated Software Engineering**, Springer, v. 20, n. 2, p. 141–184, 2013. Citado 2 vezes nas páginas 186 e 191.

ASTELS, D. Refactoring with uml. In: **Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering**. [S.l.: s.n.], 2002. p. 67–70. Citado 2 vezes nas páginas 124 e 125.

ATL, E. **ATL - a model transformation technology**. 2015. Disponível em: <https://eclipse.org/atl/>. Acesso em: 20/06/2015. Citado 3 vezes nas páginas 50, 51 e 163.

BANSIYA, J.; DAVIS, C. A hierarchical model for object-oriented design quality assessment. **Software Engineering, IEEE Transactions on**, p. 4–17, 2002. Citado 4 vezes nas páginas 21, 197, 198 e 199.

BARESI, L.; MIRAZ, M. A component-oriented metamodel for the modernization of software applications. In: **Engineering of Complex Computer Systems, 16th IEEE Inter. Conf. on**. [S.l.: s.n.], 2011. Citado na página 83.

BEZIVIN, J.; GERARD, S. A preliminary identification of mda components. In: . [S.l.: s.n.], 2002. Citado 2 vezes nas páginas 39 e 42.

BIEHL, M. Literature Study on Model Transformations. 2010. Citado 5 vezes nas páginas 48, 50, 51, 138 e 163.

BOGER, M.; STURM, T.; FRAGEMANN, P. Refactoring browser for uml. In: **Objects, Components, Architectures, Services, and Applications for a Networked World**. [S.l.]: Springer Berlin Heidelberg, 2003, (Lecture Notes in Computer Science). p. 366–377. Citado 4 vezes nas páginas 92, 167, 190 e 191.

BOOCH, G. **Object-Oriented Analysis and Design with Applications (3rd Edition)**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN 020189551X. Citado na página 39.

BOUSSAIDI, G.; BELLE, A.; VAUCHER, S.; MILI, H. Reconstructing architectural views from legacy systems. In: **Reverse Engineering, 19th Working. Conference on**. [S.l.: s.n.], 2012. Citado na página 81.

BRAGANCA, A.; MACHADO, R. J. Model driven development of software product lines. **2012 Eighth International Conference on the Quality of Information and Communications Technology**, IEEE Computer Society, Los Alamitos, CA, USA, p. 199–203, 2007. Citado na página 40.

BRAMBILLA, M.; CABOT, J.; WIMMER, M. **Model-Driven Software Engineering in Practice**. 1st. ed. [S.l.]: Morgan & Claypool Publishers, 2012. Citado 3 vezes nas páginas 42, 48 e 49.

BRIAND, L. C.; LABICHE, Y.; O'SULLIVAN, L.; SóWKA, M. M. Automated impact analysis of uml models. **Journal of System Software.**, Elsevier Science Inc., 2006. Citado 2 vezes nas páginas 33 e 178.

BROSCH, P.; SEIDL, M.; WIELAND, K.; WIMMER, M.; LANGER, P. The operation recorder: specifying model refactorings by-example. In: ACM. **Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications**. [S.l.], 2009. p. 791–792. Citado 3 vezes nas páginas 123, 124 e 125.

BROWN, A.; MCDERMID, J. The art and science of software architecture. In: **Software Architecture**. [S.l.]: Springer Berlin Heidelberg, 2007. (Lecture Notes in Computer Science), p. 237–256. Citado 2 vezes nas páginas 39 e 40.

BRUNELIÈRE, H.; CABOT, J.; JOUAULT, F.; TISI, M.; BÉZIVIN, J. Industrialization of Research Tools: the ATL Case. In: **Third International Workshop on Academic Software Development Tools and Techniques - WASDeTT-3 (co-located with the 25th IEEE/ACM International Conference on Automated Software Engineering - ASE'2010)**. [S.l.: s.n.], 2010. Citado na página 52.

BRUNELIERE, H.; CABOT, J.; JOUAULT, F.; MADIOT, F. Modisco: A generic and extensible framework for model driven reverse engineering. In: **Proceedings of the IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2010. (ASE '10), p. 173–174. Citado 3 vezes nas páginas 68, 83 e 211.

BRUNELIèRE, H.; CABOT, J.; DUPé, G.; MADIOT, F. Modisco: A model driven reverse engineering framework. **Information and Software Technology**, p. 1012 – 1032, 2014. Citado 2 vezes nas páginas 68 e 164.

CANFORA, G.; PENTA, M. D.; CERULO, L. Achievements and challenges in software reverse engineering. **Commun. ACM**, ACM, New York, NY, USA, v. 54, p. 142–151, abr. 2011. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/1924421.1924451>. Citado na página 55.

CHIKOFSKY, E.; CROSS J.H., I. Reverse engineering and design recovery: a taxonomy. **Software, IEEE**, p. 13–17, 1990. Citado na página 44.

CZARNECKI, K.; HELSEN, S. Feature-based survey of model transformation approaches. **IBM Syst. J.**, IBM Corp., Riverton, NJ, USA, v. 45, n. 3, 2006. Citado na página 48.

DEMEYER, S. Refactor Conditionals into Polymorphism: What's the Performance Cost of Introducing Virtual Calls? **Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on**, 2005. Citado 2 vezes nas páginas 52 e 158.

DEMEYER, S.; BOIS, B. D.; VERELST, J. Refactoring - Improving Coupling and Cohesion of Existing Code. **Reverse Engineering, 2004. Proceedings. 11th Working Conference on**, 2004. Citado 2 vezes nas páginas 138 e 158.

DEURSEN, A. V.; VISSER, E.; WARMER, J. Model-driven software evolution: A research agenda. 2007. Citado na página 178.

DUCASSE, S.; GIRBA, T.; LANZA, M.; DEMEYER, S. Moose: a collaborative and extensible reengineering environment. **Tools for Software Maintenance and Reengineering, RCOST/-Software Technology Series**, Citeseer, v. 71, 2005. Citado 4 vezes nas páginas 123, 125, 155 e 156.

DUCASSE, S.; LANZA, M.; TICHELAAR, S. Moose: an extensible language-independent environment for reengineering object-oriented systems. In: **Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)**. [S.l.: s.n.], 2000. v. 4. Citado 3 vezes nas páginas 188, 189 e 191.

DURELLI, R.; SANTIBANEZ, D.; DELAMARO, M.; CAMARGO, V. de. Towards a refactoring catalogue for knowledge discovery metamodel. In: **Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on**. [S.l.: s.n.], 2014. p. 569–576. Citado 5 vezes nas páginas 31, 54, 87, 89 e 90.

DURELLI, R.; SANTIBANEZ, D.; MARINHO, B.; HONDA, R.; DELAMARO, M.; ANQUETIL, N.; CAMARGO, V. de. A mapping study on architecture-driven modernization. In: **Information Reuse and Integration (IRI), 2014 IEEE 15th International Conference on**. [S.l.: s.n.], 2014. p. 577–584. Citado 4 vezes nas páginas 33, 72, 122 e 128.

DURELLI, R.; SANTOS, B.; HONDA, R.; DELAMARO, M. E.; CAMARGO, V. de. Kdm-re: A model-driven refactoring tool for kdm. In: **Workshop on Software Visualization, Maintenance, and Evolution (VEM), 2014**. [S.l.: s.n.], 2014. p. 1–8. Citado 4 vezes nas páginas 54, 87, 89 e 162.

DURELLI, R. S.; nEZ, D. S. M. S.; ANQUETIL, N.; DELAMARO, M. E.; CAMARGO, V. V. de. A systematic review on mining techniques for crosscutting concerns. In: **Proceedings of the 28th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2013. p. 1080–1087. Citado na página 32.

DYBA, T.; DINGSOYR, T.; HANSSEN, G. Applying systematic reviews to diverse study types: An experience report. In: **Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on**. [S.l.: s.n.], 2007. p. 225–234. Citado na página 74.

DYBA, T.; KITCHENHAM, B. A.; JORGENSEN, M. Evidence-based software engineering for practitioners. **IEEE Softw.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 22, n. 1, p. 58–65, 2005. Citado na página 85.

EGYED, A.; LETIER, E.; FINKELSTEIN, A. Generating and evaluating choices for fixing inconsistencies in uml design models. In: **23rd IEEE/ACM International Conference on Automated Software Engineering**. [S.l.]: IEEE Computer Society, 2008. p. 99–108. Citado 2 vezes nas páginas 33 e 178.

EHRIG, H.; EHRIG, K.; PRANGE, U.; TAENTZER, G. **Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. Citado na página 49.

EMF. **Eclipse Modeling Framework (EMF)**. 2015. Disponível em: <https://eclipse.org/modeling/emf/>. Acesso em: 15/03/2015. Citado na página 43.

ETL. **Epsilon Transformation Language (ETL)**. 2015. Disponível em: <https://www.eclipse.org/epsilon/doc/etl/>. Acesso em: 01/04/2015. Citado na página 50.

FERNáNDEZ-ROPERO, M.; PÉREZ-CASTILLO, R.; WEBER, B.; PIATTINI, M. Empirical assessment of business model transformations based on model simulation. In: **International Conference on Theory and Practice of Model Transformations**. [S.l.: s.n.], 2012. Citado na página 82.

FLURI, B.; GALL, H. Classifying change types for qualifying change couplings. In: **Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on**. [S.l.: s.n.], 2006. p. 35–45. Citado na página 49.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. Boston, MA, USA: Addison-Wesley, 1999. Citado 18 vezes nas páginas 32, 44, 45, 46, 47, 89, 96, 100, 112, 117, 125, 132, 138, 158, 167, 186, 197 e 202.

FRANCE, R.; RUMPE, B. Model-driven development of complex software: A research roadmap. In: **Future of Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2007. p. 37–54. Citado 3 vezes nas páginas 40, 41 e 42.

FREY, S.; HASSELBRING, W. An extensible architecture for detecting violations of a cloud environment's constraints during legacy software system migration. In: **Software Maintenance and Reengineering, 15th European Conf. on**. [S.l.: s.n.], 2011. Citado na página 81.

FREY, S.; HASSELBRING, W.; SCHNOOR, B. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. **Journal of Software: Evolution and Process**, 2012. Citado na página 81.

FUENTES-FERNANDEZ, R.; PAVON, J.; GARIJO, F. A model-driven process for the modernization of component-based systems. **Science of Computer Programming**, 2012. Citado na página 81.

FUJABA. **Fujaba**. 2015. Disponível em: <http://www.fujaba.de/>. Acesso em: 12/02/2015. Citado na página 50.

GORP, P. V.; STENTEN, H.; MENS, T.; DEMEYER, S. Towards automating source-consistent uml refactorings. p. 144–158, 2003. Citado 2 vezes nas páginas 32 e 33.

GUZMAN, I. G.-R. d.; POLO, M.; PIATTINI, M. An adm approach to reengineer relational databases towards web services. In: **14th Work. Conference on Reverse Engineering**. [S.l.: s.n.], 2007. Citado na página 81.

GUZMAN, I. Garcia-Rodriguez de. Pressweb: A process to reengineer legacy systems towards web services. In: **Reverse Engineering, 14th Working Conf. on**. [S.l.: s.n.], 2007. Citado na página 81.

HENSHIN, E. **EMF Henshin**. 2015. Disponível em: <https://www.eclipse.org/henshin/>. Acesso em: 23/03/2015. Citado na página 50.

HOSTE, M. Supporting model-driven refactoring. 2013. Citado 2 vezes nas páginas 186 e 187.

HUBER, P. **The Model Transformation Language Jungle - An Evaluation and Extension of Existing Approaches**. Dissertação (Mestrado) — Technische Universität Wien, 2008. Citado 2 vezes nas páginas 50 e 51.

HUTCHINSON, J.; WHITTLE, J.; ROUNCEFIELD, M.; KRISTOFFERSEN, S. Empirical assessment of mde in industry. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, USA: ACM, 2011. p. 471–480. Citado 2 vezes nas páginas 41 e 42.

ISO/IEC19506. **ISO/IEC19506**. 2012. Disponível em: <http://www.iso.org/iso/catalogue_detail.htm?csnumber=32625>. Acesso em: 09/06/2015. Citado 3 vezes nas páginas 31, 43 e 55.

IZQUIERDO, J.; MOLINA, J. An architecture-driven modernization tool for calculating metrics. **Software, IEEE**, 2010. Citado 2 vezes nas páginas 82 e 83.

JENSEN, A. C.; CHENG, B. H. On the use of genetic programming for automated refactoring and the introduction of design patterns. In: **Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation**. [S.l.]: ACM, 2010. p. 1341–1348. Citado na página 197.

JOUAULT, F.; ALLILAIRE, F.; BÉZIVIN, J.; KURTEV, I. Atl: A model transformation tool. **Sci. Comput. Program.**, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, p. 31–39, 2008. Citado 3 vezes nas páginas 50, 51 e 163.

JOUAULT, F.; KURTEV, I. Transforming models with atl. In: **Proceedings of the 2005 International Conference on Satellite Events at the MoDELS**. Berlin, Heidelberg: Springer-Verlag, 2006. (MoDELS'05). Citado na página 51.

KAY, M. **XSLT 2.0 and XPath 2.0 Programmer's Reference**. [S.l.]: John Wiley & Sons, 2011. Citado na página 181.

KDM. **Knowledge Discovery Meta-model (KDM)**. 2015. Disponível em: <http://www.omg.org/technology/kdm/>. Acesso em: 23/09/2015. Citado 8 vezes nas páginas 31, 55, 56, 57, 58, 59, 60 e 62.

KERMETA. **kermeta**. 2015. Disponível em: <http://www.kermeta.org/>. Acesso em: 01/05/2015. Citado na página 50.

KESSENTINI, M.; VAUCHER, S.; SAHRAOUI, H. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: **Proceedings of the IEEE/ACM International Conference on Automated Software Engineering**. [S.l.: s.n.], 2010. p. 113–122. Citado na página 204.

KITCHENHAM, B.; BRERETON, O. P.; BUDGEN, D.; TURNER, M.; BAILEY, J.; LINKMAN, S. Systematic literature reviews in software engineering - a systematic literature review. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, USA, v. 51, p. 7–15, January 2009. Citado na página 85.

KITCHENHAM, B.; PRETORIUS, R.; BUDGEN, D.; BRERETON, O. P.; TURNER, M.; NIAZI, M.; LINKMAN, S. Systematic literature reviews in software engineering - a tertiary study. **Inf. Softw. Technol.**, Butterworth-Heinemann, Newton, MA, USA, p. 792–805, 2010. Citado 3 vezes nas páginas 71, 72 e 73.

KLEPPE, A. G.; WARMER, J.; BAST, W. **MDA Explained: The Model Driven Architecture: Practice and Promise**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. Citado 2 vezes nas páginas 40 e 41.

KOLAHDOUZ-RAHIMI, S.; LANO, K.; PILLAY, S.; TROYA, J.; GORP, P. V. Evaluation of model transformation approaches for model refactoring. **Science of Computer Programming**, p. 5 – 40, 2014. Citado 2 vezes nas páginas 32 e 33.

LANGER, P.; WIMMER, M.; KAPPEL, G. Model-to-model transformations by demonstration. In: **Theory and Practice of Model Transformations**. [S.l.]: Springer, 2010. p. 153–167. Citado 2 vezes nas páginas 124 e 125.

LEDECZI, A.; MAROTI, M.; BAKAY, A.; KARSAI, G.; GARRETT, J.; THOMASON, C.; NORDSTROM, G.; SPRINKLE, J.; VOLGYESI, P. The generic modeling environment. In: **Workshop on Intelligent Signal Processing, Budapest, Hungary**. [S.l.: s.n.], 2001. v. 17, p. 1. Citado na página 123.

LEHNERT, S.; FAROOQ, Q.; RIEBISCH, M. A taxonomy of change types and its application in software evolution. In: **Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on**. [S.l.: s.n.], 2012. p. 98–107. Citado na página 49.

LIMA, B.; SOUSA, J. de; LOPES, D. Using mda to support hypermedia document sharing. In: **Software Engineering Advances, 2007. ICSEA 2007. International Conference on**. [S.l.: s.n.], 2007. p. 65–65. Citado na página 40.

LIU, W.; EASTERBROOK, S.; MYLOPOULOS, J. Rule-based detection of inconsistency in uml models. In: **Workshop on Consistency Problems in UML-Based Software Development**. [S.l.: s.n.], 2002. p. 106–123. Citado na página 178.

MAINETTI, L.; PAIANO, R.; PANDURINO, A. Migros: a model-driven transformation approach of the user experience of legacy applications. In: **12th International Conference on Web Engineering**. [S.l.: s.n.], 2012. Citado na página 81.

MARCA, D. A.; MCGOWAN, C. L. **SADT: Structured Analysis and Design Technique**. New York, NY, USA: McGraw-Hill, Inc., 1987. ISBN 0-07-040235-3. Citado na página 91.

MAZóN, J.-N.; TRUJILLO, J. A model driven modernization approach for automatically deriving multidimensional models in data warehouses. In: **26th International Conference on Conceptual modeling**. [S.l.: s.n.], 2007. Citado na página 81.

MELLOR, S. J.; KENDALL, S.; UHL, A.; WEISE, D. **MDA Distilled**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004. Citado na página 42.

MENS, T. Introduction and roadmap: History and challenges of software evolution. Springer Berlin Heidelberg, p. 1–11, 2008. Citado na página 177.

MENS, T.; DEMEYER, S.; BOIS, B. D.; STENTEN, H.; GORP, P. V. Refactoring: Current research and future trends. **Electronic Notes in Theoretical Computer Science**, p. 483–499, 2003. Citado na página 47.

MENS, T.; DEURSEN, A. V. *et al.* Refactoring: Emerging trends and open problems. In: **Proceedings First International Workshop on REFactoring: Achievements, Challenges, Effects (REFACE). University of Waterloo**. [S.l.: s.n.], 2003. Citado na página 47.

MENS, T.; GORP, P. V. A taxonomy of model transformation. **Electron. Notes Theor. Comput. Sci.**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 152, p. 125–142, 2006. Citado 4 vezes nas páginas 32, 48, 50 e 163.

MENS, T.; TAENTZER, G.; Müller, D. Challenges in model refactoring. In: **Proc. 1st Workshop on Refactoring Tools**. [S.l.: s.n.], 2007. Citado na página 47.

MENS, T.; TAENTZER, G.; MüLLER, D. Model-driven software refactoring. **Model-driven software development: integrating quality assurance.**, v. 20, n. 1, 2008. Citado na página 47.

MENS, T.; TOURWE, T. A Survey of Software Refactoring. **Transactions on Software Engineering, IEEE**, v. 30, 2004. Citado na página 46.

MILLAN, T.; SABATIER, L.; THI, T.-T. L.; BAZEX, P.; PERCEBOIS, C. An ocl extension for checking and transforming uml models. In: **International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'09), Cambridge, United Kingdom**. [S.l.: s.n.], 2009. Citado na página 191.

MISBHAUDDIN, M. Towards an integrated metamodel based approach to software refactoring. King Fahd University of Petroleum and Minerals (Saudi Arabia), 2012. Citado 2 vezes nas páginas 124 e 125.

MISBHAUDDIN, M.; ALSHAYEB, M. Uml model refactoring: a systematic literature review. **Empirical Software Engineering**, Springer US, v. 20, n. 1, p. 206–251, 2015. Citado na página 33.

MODELMORF. **ModelMorf**. 2015. Disponível em: <https://code.google.com/p/qvtr2/>. Acesso em: 03/03/2015. Citado na página 50.

MOF. **OMG's MetaObject Facility**. 2015. Disponível em: <http://www.omg.org/mof/>. Acesso em: 09/06/2015. Citado 2 vezes nas páginas 43 e 55.

MOHAMED, M.; ROMDHANI, M.; GHEDIRA, K. M-refactor: A new approach and tool for model refactoring. **ARPN Journal of Systems and Software (July 2011)**, 2011. Citado 2 vezes nas páginas 188 e 191.

MOHAMED MOHAMED ROMDHANI, K. G. M. Classification of model refactoring approaches. **JOURNAL OF OBJECT TECHNOLOGY**, p. 143–158, 2010. Citado na página 47.

MORATALLA, J.; CASTRO, V. de; SANZ, M.; MARCOS, E. A gap-analysis-based framework for evolution and modernization: Modernization of domain management at red.es. In: **SRII Global Conf.** [S.l.: s.n.], 2012. Citado na página 80.

NORMANTAS, K.; VASILECAS, O. Extracting business rules from existing enterprise software system. In: **Information and Software Technologies**. [S.l.]: Springer Berlin Heidelberg, 2012. (Communications in Computer and Information Science). Citado na página 82.

Ó CINNÉIDE, M. **Automated Application of Design Patterns: A Refactoring Approach**. Tese (Doutorado) — Trinity College, University of Dublin, October 2000. Citado na página 46.

O'KEEFFE, M.; CINNÉIDE, M. í. Search-based refactoring for software maintenance. **Journal of System and Software**, p. 502–516. Citado na página 197.

OMG. **Object Management Group (OMG)**. 2015. Disponível em: <http://www.omg.org>. Acesso em: 23/03/2015. Citado 2 vezes nas páginas 52 e 213.

OPDYKE, W. F. **REFACTORING OBJECT-ORIENTED FRAMEWORKS**. Tese (Doutorado) — University of Wisconsin, 1992. Citado 7 vezes nas páginas 32, 44, 45, 46, 47, 132 e 139.

OPENARCHITECTUREWARE. **OpenArchitectureWare (OAW)**. 2015. Disponível em: <https://www.eclipse.org/gmt/oaw/>. Acesso em: 01/02/2015. Citado na página 50.

OUNI, A.; KESSENTINI, M.; SAHRAOUI, H. A. Multiobjective optimization for software refactoring and evolution. **Advances in Computers**, Academic Press, v. 94, p. 103–167, 2014. Citado 3 vezes nas páginas 153, 154 e 156.

PÉREZ-CASTILLO, R.; CRUZ-LEMUS, J. A.; GUZMAN, I. G.-R. de; PIATTINI, M. A family of case studies on business process mining using marble. **Systems and Software**, 2012. Citado na página 82.

PÉREZ-CASTILLO, R.; FERNANDEZ-ROPERO, M.; GUZMAN, I. Garcia-Rodriguez de; PIATTINI, M. Marble. a business process archeology tool. In: **27th IEEE Inter. Conf. on Software Maintenance**. [S.l.: s.n.], 2011. Citado na página 82.

PÉREZ-CASTILLO, R.; GARCÍA-RODRÍGUEZ, I.; CABALLERO, I. Preciso: A reengineering process and a tool for database modernisation through web services. In: **ACM SAC**. [S.l.: s.n.], 2009. Citado na página 81.

PÉREZ-CASTILLO, R.; GUZMáN, I. G.-R. D.; PIATTINI, M. Implementing business process recovery patterns through qvt transformations. In: **Inter. Conf. on Theory and practice of model transformations**. [S.l.]: Springer-Verlag, 2010. p. 168–183. Citado na página 82.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G.-R. de; PIATTINI, M. **Architecture Driven Modernization**. San Francisco, CA, USA: Information Science Reference, 2011. ISBN 0123749131, 9780123749130. Citado na página 55.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. R. de; PIATTINI, M. Database schema elicitation to modernize relational databases. In: **International Conference on Enterprise Information Systems (ICEIS)**. [S.l.: s.n.], 2012. Citado na página 81.

PÉREZ-CASTILLO, R.; GUZMáN, I. G.-R. de; PIATTINI, M.; WEBER, B. Integrating event logs into kdm repositories. In: **27th Annual ACM Symposium on Applied Computing**. [S.l.: s.n.], 2012. Citado 2 vezes nas páginas 82 e 83.

PÉREZ-CASTILLO, R.; GUZMAN, I. Garcia-Rodriguez de. Mimos, system model-driven migration project. In: **17th European Conf. on Software Maintenance and Reengineering**. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 82 e 83.

PÉREZ-CASTILLO, R.; GUZMAN, I. Garcia-Rodriguez de; PIATTINI, M. Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. **Computer Standards & Interfaces**, v. 33, n. 6, p. 519–532, 2011. Citado 2 vezes nas páginas 55 e 83.

PÉREZ-CASTILLO, R.; GUZMAN, I. Garcia Rodriguez de; PIATTINI, M.; PIATTINI, M. On the use of adm to contextualize data on legacy source code for software modernization. In: **16th Work. Conf. on Reverse Engineering**. [S.l.: s.n.], 2009. Citado na página 81.

PÉREZ-CASTILLO, R.; GUZMáN WEBER, B. de; PLACES, A. S. An empirical comparison of static and dynamic business process mining. In: **ACM Symposium on Applied Computing**. [S.l.: s.n.], 2011. Citado na página 82.

PETERSEN, K.; FELDT, R.; MUJTABA, S.; MATTSSON, M. Systematic mapping studies in software engineering. In: **Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering**. [S.l.]: British Computer Society, 2008. p. 68–77. Citado 4 vezes nas páginas 71, 72, 73 e 75.

PETERSEN, K.; VAKKALANKA, S.; KUZNIARZ, L. Guidelines for conducting systematic mapping studies in software engineering: An update. **Information and Software Technology**, p. 1 – 18, 2015. Citado 3 vezes nas páginas 71, 72 e 75.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 7. ed. New York, NY, USA: McGraw-Hill, Inc., 2010. Citado na página 31.

QVT. **Query/View/Transformation**. 2015. Disponível em: <http://www.omg.org/spec/QVT/1.1/>. Acesso em: 23/03/2015. Citado na página 50.

RAJLICH, V. A methodology for incremental changes. Springer Berlin Heidelberg, p. 1–11, 2009. Citado na página 178.

REIMANN, J. **Generic Quality-Aware Refactoring and Co-Refactoring in Heterogeneous Model Environments**. Tese (Doutorado), Dresden, Alemanha, 2015. Citado 3 vezes nas páginas 124, 125 e 187.

REIMANN, J.; SEIFERT, M.; ASSMANN, U. Role-based generic model refactoring. In: **Model Driven Engineering Languages and Systems**. [S.l.]: Springer, 2010. p. 78–92. Citado 2 vezes nas páginas 187 e 191.

ROBERTS, D.; BRANT, J.; JOHNSON, R. A refactoring tool for smalltalk. **Urbana**, v. 51, p. 61801, 1997. Citado 2 vezes nas páginas 189 e 190.

ROBERTS, D. B. **Practical Analysis for Refactoring**. Tese (Doutorado), Champaign, IL, USA, 1999. Citado 3 vezes nas páginas 45, 46 e 139.

RODRÍGUEZ-ECHEVERRÍA, R.; CLEMENTE, P. J.; PRECIADO, J. C.; SANCHEZ-FIGUEROA, F. Modernization of legacy web applications into rich internet applications. In: **11th International Conference on Current Trends in Web Engineering**. [S.l.: s.n.], 2012. Citado na página 81.

ROSENBERG, L. H. Software reengineering. **Software, IEEE**, n. 5, p. 103 –111, 1996. Citado na página 55.

SALEM, R. B.; GRANGEL, R.; BOUREY, J.-P. A comparison of model transformation tools: Application for transforming {GRAI} extended actigrams into {UML} activity diagrams. **Computers in Industry**, v. 59, p. 682 – 693, 2008. Enterprise Integration and Interoperability in Manufacturing Systems. Citado 2 vezes nas páginas 33 e 52.

SANTIBANEZ, D.; DURELLI, R. S.; MARINHO, B.; CAMARGO, V. V. de. A Combined Approach for Concern Identification in KDM models. In: **Latin-American Workshop on Aspect-Oriented Software Development**. [S.l.: s.n.], 2013. Citado 2 vezes nas páginas 32 e 82.

____. CCKDM - A Concern Mining Tool for Assisting in the Architecture-Driven Modernization Process. In: **Session Tools - CBSoft (Congresso Brasileiro de Software)**. [S.l.: s.n.], 2013. Citado na página 82.

____. A Combined Approach for Concern Identification in KDM models. **Journal of the Brazilian Computer Society - JBCS**, n. 5, p. 32–56, 2015. Citado 2 vezes nas páginas 32 e 82.

SANTOS, B. M. **EXTENSÕES DO METAMODELO KDM PARA APOIAR MODERNIZAÇÕES ORIENTADAS A ASPECTOS DE SISTEMAS LEGADOS**. Dissertação (Mestrado) — Universidade Federal de São Carlos - UFSCAR, São Carlos, 2014. Citado na página 59.

SCHMIDT, D. Guest editor's introduction: Model-driven engineering. **Computer**, p. 25–31, 2006. Citado 2 vezes nas páginas 41 e 42.

SEN, S.; MOHA, N.; MAHÉ, V.; BARAIS, O.; BAUDRY, B.; JÉZÉQUEL, J.-M. Reusable model transformations. **Software & Systems Modeling**, Springer, v. 11, n. 1, p. 111–125, 2012. Citado 4 vezes nas páginas 122, 125, 155 e 156.

SENDALL, S.; KOZACZYNSKI, W. Model transformation: The heart and soul of model-driven software development. **IEEE Software**, IEEE Computer Society Press, Los Alamitos, CA, USA, p. 42–45, 2003. Citado na página 210.

SENG, O.; STAMMEL, J.; BURKHART, D. Search-based determination of refactorings for improving the class structure of object-oriented systems. In: **Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation**. [S.l.: s.n.], 2006. p. 1909–1916. Citado na página 197.

SHAHSHAHANI, P. M. Extending the knowledge discovery metamodel to support aspect-oriented programming. In: . [S.l.: s.n.], 2011. Citado na página 83.

SMARTQVT. **SmartQVT**. 2015. Disponível em: <http://sourceforge.net/projects/smartqvt/>. Acesso em: 23/02/2015. Citado na página 50.

SNEED, H. Estimating the costs of a reengineering project. In: **Reverse Engineering, 12th Working Conference on**. [S.l.: s.n.], 2005. p. 9–119. Citado na página 52.

STAROŃ, M.; KUŹNIARZ, L. Implementing uml model transformations for mda. In: **11th Nordic Workshop on Programming and Software Development and Tools**. [S.l.: s.n.], 2004. p. 141–152. Citado 2 vezes nas páginas 33 e 111.

ŠTOLC, M.; POLÁŠEK, I. A visual based framework for the model refactoring techniques. In: **Proc. of 8th International Symposium on Applied Machine Intelligence and Informatics, SAMI**. [S.l.: s.n.], 2010. Citado 2 vezes nas páginas 190 e 191.

STRAETEN, R. V. D.; D'HONDT, M. Model refactorings through rule-based inconsistency resolution. In: ACM. **Proceedings of the 2006 ACM symposium on Applied computing**. [S.l.], 2006. p. 1210–1217. Citado 3 vezes nas páginas 189, 190 e 191.

STRAETEN, R. V. D.; MENS, T.; BAELEN, S. V. Challenges in model-driven software engineering. In: **Models in Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2009. p. 35–47. Citado na página 47.

SUN, Y.; WHITE, J.; GRAY, J. Model transformation by demonstration. In: **Model Driven Engineering Languages and Systems**. [S.l.]: Springer, 2009. p. 712–726. Citado 3 vezes nas páginas 123, 124 e 125.

THORSTEN, A.; PAWEL, S.; GABRIELE, T. Emf metrics: Specification and calculation of model metrics within the eclipse modeling framework. In: **Proceedings of the 9th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)**. [S.l.: s.n.], 2010. p. 1–5. Citado na página 201.

TICHELAAR, S.; DUCASSE, S.; DEMEYER, S.; NIERSTRASZ, O. A meta-model for language-independent refactoring. In: IEEE. **Principles of Software Evolution, 2000. Proceedings. International Symposium on**. [S.l.], 2000. p. 154–164. Citado 7 vezes nas páginas 122, 123, 125, 155, 156, 188 e 189.

ULRICH, W. M.; NEWCOMB, P. **Information Systems Transformation: Architecture-Driven Modernization Case Studies**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. Citado na página 33.

UML. **Unified Modeling Language^{TM}(UML)**. 2015. Disponível em: <http://www.omg.org/spec/UML/>. Acesso em: 23/03/2015. Citado 2 vezes nas páginas 39 e 43.

VIATRA. **Viatra**. 2015. Disponível em: <https://www.eclipse.org/viatra/>. Acesso em: 01/02/2015. Citado na página 50.

WAKE, W. C. **Refactoring Workbook**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. Citado 2 vezes nas páginas 45 e 47.

WIERINGA, R.; MAIDEN, N.; MEAD, N.; ROLLAND, C. Requirements engineering paper classification and evaluation criteria: A proposal and a discussion. **Requir. Eng.**, Springer-Verlag New York, Inc., Secaucus, NJ, USA, p. 102–107, 2005. Citado na página 77.

WOHLIN, C.; RUNESON, P.; HöST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLéN, A. **Experimentation in software engineering: an introduction**. Norwell, MA, USA: Kluwer Academic Publishers, 2012. Citado 3 vezes nas páginas 195, 198 e 199.

ZHANG, J.; LIN, Y.; GRAY, J. Generic and domain-specific model refactoring using a model transformation engine. In: **Model-Driven Software Development**. [S.l.]: Springer Berlin Heidelberg, 2005. p. 199–217. Citado 4 vezes nas páginas 92, 123, 125 e 167.

APÊNDICE

A

REGRAS PRÉ-DEFINIDAS PARA SINCRONIZAR O KDM

A.1 Introdução

Este apêndice apresenta as regras pré-definidas que são utilizadas no módulo de sincronização da ferramenta KDM-RE. O módulo de sincronização apresentado no Capítulo 6, Seção 6.5 utiliza regras pré-definidas que são iniciadas de acordo a(s) refatoração(ões) aplicada(s) na instância do metamodelo KDM. Todas as propagações especificadas no módulo de sincronização são pré-definidas para serem disparadas após a aplicação de específicas refatorações. Todas as propagações são definidas com base nas mudanças realizadas em uma determinada instância da metaclasses do metamodelo KDM. Além disso, todas as regras pré-definidas foram implementadas em ATL. Nas Tabelas 24, 25, 26 e 27 todas as regras de programação pré-definidas são ilustradas e explicadas.

Tabela 24 – Propagações definidas para refatorações realizadas em instâncias da metaclasse `Package`.

Início da Tabela		
Refatoração	Propagação	
`Add Package`	`Code Package`	Cria uma instância da metaclasse `Package`
	`Action Package`	Não se aplica

Continuação da Tabela 24		
Refatoração	Propagação	
	Structure Package	**Cenário 1**: Cria-se um elemento arquitetural (Layer, Component, etc). Deve-se associar a instância da metaclasse Package criada por meio da associação implementation do elemento arquitetural criado. **Cenário 2**: Já existe um elemento arquitetural (Layer, Component, etc) para representar em nível arquitetural a instância da metaclasse Package criada. Assim, apenas deve-se associar a instância da metaclasse Package criada por meio da associação implementation do elemento arquitetural.
	Data Package	Não se aplica
	Conceptual Package	**Cenário 1**: Cria-se uma instância da metaclasse ScenarioUnit. Deve-se associar a instância da metaclasse Package criada por meio da associação implementation da metaclasse ScenarioUnit criado. **Cenário 2**: Já existe uma instância da metaclasse ScenarioUnit para representar em nível de regra de negócio a instância da metaclasse Package criada. Assim, apenas deve-se associar a instância da metaclasse Package criada por meio da associação implementation da metaclasse ScenarioUnit.
delete Package	Code Package	**Cenário 1**: Verifica se a instância da metaclasse Package contém ClassUnits e/ou InterfaceUnits por meio da associação codeElement . Caso positivo, deve-se mover todas as ClassUnits e/ou InterfaceUnit para outra instância da metaclasse Package. **Cenário 2**: Verifica se a instância da metaclasse Package contém ClassUnits e/ou InterfaceUnits por meio da associação codeElement. Caso negativo, a instância da metaclasse Package pode ser deletada.
	Action Package	Não se aplica
	Structure Package	Caso a instância de Package a ser deleta contenha uma representação em nível arquitetural, então deve-se remover a referência do pacote deletado que está contido na associação implementation do elemento arquitetural.
	Data Package	Não se aplica

Continuação da Tabela 24		
Refatoração	Propagação	
	`Conceptual Package`	Caso o pacote a ser deletado contenha uma representação em nível de regras de negócio, então, deve-se também remover a referência do pacote deletado que está contido na associação `implementation` da metaclasse `ScenarioUnit`
`change Package`	`Code Package`	Altera os meta-atributos da instância da metaclasse `Package`.
	`Action Package`	Não se aplica
	`Structure Package`	Caso o pacote a ser alterado contenha uma representação em nível arquitetural, então, deve-se alterar o meta-atributo `name` da instância da metaclasse `Layer`.
	`Data Package`	Não se aplica.
	`Conceptual Package`	Caso o pacote a ser alterado contenha uma representação em nível de regras de negócio, então deve-se alterar o meta-atributo `name` da instância da metaclasse `ScenarioUnit`.
Fim da Tabela		

Na Tabela 25, as regras que regem as propagações pré-definidas para refatorações realizadas em instâncias da metaclasse `ClassUnit` são apresentadas.

Tabela 25 – Propagações definidas para refatorações realizadas em instâncias da metaclasse `ClassUnit`.

Início da Tabela		
Refatoração	Propagação	
`add ClassUnit`	`Code Package`	Cria-se uma instância da metaclasse `ClassUnit`
	`Action Package`	Não se aplica
	`Structure Package`	Não se aplica
	`Data Package`	Deve-se criar uma instância da metaclasse `RelationalTable`. Além disso, deve-se também especificar que o meta-atributo `name` de ambas as metaclasses (`ClassUnit` criada e `RelationalTable`) terão o mesmo valor. Deve-se também criar criar uma instância da metaclasse `UniqueKey`.

Continuação da Tabela 25

Refatoração	Propagação	
	Conceptual Package	**Cenário 1**: Cria-se uma instância da metaclasse RuleUnit. Deve-se associar a instância da metaclasse ClassUnit criada por meio da associação implementation da metaclasse RuleUnit criada. **Cenário 2**: Já existe uma instância da metaclasse RuleUnit para representar em nível de regra de negócio a instância da metaclasse ClassUnit criada. Assim, apenas deve-se associar a instância da metaclasse ClassUnit criada por meio da associação implementation da metaclasse RuleUnit.
delete ClassUnit	Code Package	Deleta-se uma instância da metaclasse ClassUnit.
	Action Package	Devem-se remover todas as instâncias das metaclasses que utilizam a instância da metaclasse ClassUnit deletada. Por exemplo, todas as instâncias das metaclasses Calls, Extends, HasType, ParameterUnit devem ser removidas.
	Structure Package	Se a ClassUnit a ser removida está associada a uma instância Package por meio da associação codeElement, que, por sua vez, está associada a algum elemento arquitetural por meio da associação implementation, então, deve-se verificar se o elemento arquitetural possui uma instância da metaclasse AggregationRelationship. Em seguida, devem-se remover todas as instâncias de HasType, Calls, Extends, Implements, etc. que possuem relacionamento com a ClassUnit a ser removida. Em seguida, deve-se atualizar o meta-atributo density da metaclasse AggregationRelationship.
	Data Package	Deve-se identificar uma instância da metaclasse RelationalTable que possui o mesmo meta-atributo name da instância da metaclasse ClassUnit a ser removida. Então, deve-se também remover a instância da metaclasse RelationalTable

Continuação da Tabela 25		
Refatoração	Propagação	
	`Conceptual Package`	Deve-se identificar uma instância da metaclasse `RuleUnit` que possui o mesmo meta-atributo `name` da instância da metaclasse `ClassUnit` a ser removida. Em seguida, deve-se remover a instância da metaclasse `RuleUnit`.
`change ClassUnit`	`Code Package`	Renomeia o meta-atributo `name` da instância da metaclasse `ClassUnit`.
	`Action Package`	Devem-se alterar todas as metaclasses que utilizam a instância da metaclasse `ClassUnit` renomeada. Todas as associações `to` ou `from` das instâncias das metaclasses `Calls`, `Extends`, `HasType`, `ParameterUnit` devem ser renomeadas.
	`Structure Package`	Se a `ClassUnit` a ser renomeada está associada a uma instância `Package` por meio da associação `codeElement`, que, por sua vez, está associado a algum elemento arquitetural por meio da associação `implementation`, então, deve-se verificar se o elemento arquitetural possui uma instância da metaclasse `AggregationRelationship`. Em seguida, devem-se remover todas as associações `to` ou `from` das instâncias de `HasType`, `Calls`, `Extends`, `Implements`, etc. que possuem relacionamento com a `ClassUnit` a ser removida.
	`Data Package`	Deve-se identificar uma instância da metaclasse `RelationalTable` que possui o mesmo meta-atributo `name` da instância da metaclasse `ClassUnit` a ser renomeada. Em seguida, deve-se renomear o meta-atributo `name` da instância da metaclasse `RelationalTable`.
	`Conceptual Package`	Deve-se identificar uma instância da metaclasse `RuleUnit` que possui o mesmo meta-atributo `name` da instância da metaclasse `ClassUnit` a ser renomeada. Em seguida, deve-se renomear o meta-atributo `name` da instância da metaclasse `RuleUnit`.
Fim da Tabela		

As regras pré-definidas para as refatorações realizadas em instâncias da metaclasse `StorableUnit` são destacas na Tabela 26.

Tabela 26 – Propagações definidas para refatorações realizadas em instâncias metaclasse `StorableUnit`.

Início da Tabela		
Refatoração	Propagação	
`add StorableUnit`	`Code Package`	Cria-se uma instância da metaclasse `StorableUnit`
	`Action Package`	Não se aplica
	`Structure Package`	Não se aplica
	`Data Package`	Deve-se verificar se a instância da metaclasse `StorableUnit` criada está contida em uma instância da metaclasse `ClassUnit`, que, por sua vez, contém uma metaclasse `RelationalTable` correspondente. Caso afirmativo, deve-se criar uma instância da metaclasse `ColumnSet` e associar a instância da metaclasse `RelationalTable`.
	`Conceptual Package`	Não se aplica.
`delete StorableUnit`	`Code Package`	Deleta-se uma instância da metaclasse `StorableUnit`.
	`Action Package`	Devem-se remover todas as instâncias das metaclasses `Reads`, `Writes` e `Address` que utilizam a instância da metaclasse `StorableUnit` deletada.
	`Structure Package`	Se a `StorableUnit` a ser removida estiver contida em uma instância de `ClassUnit` por meio da associação `codeElement`, que, por sua vez, está associada a uma instância `Package` por meio da associação `codeElement`, a qual está associada a algum elemento arquitetural por meio do associação `implementation`, então, deve-se verificar se o elemento arquitetural possui uma instância da metaclasse `AggregationRelationship`. Em seguida, devem-se remover todas as instâncias de `Reads`, `Writes` e `Address` que utilizam a instância da metaclasse `StorableUnit` deletada. Em seguida, deve-se atualizar o meta-atributo `density` da metaclasse `AggregationRelationship`.

Continuação da Tabela 26		
Refatoração	Propagação	
	Data Package	Deve-se identificar uma instância da metaclasse ColumnSet que possui o mesmo meta-atributo name da instância da metaclasse StorableUnit a ser removida. Então, deve-se também remover a instância da metaclasse ColumnSet.
	Conceptual Package	Não se aplica.
change StorableUnit	Code Package	Renomeia o meta-atributo name da instância da metaclasse StorableUnit.
	Action Package	Devem-se alterar todas as metaclasses que utilizam a instância da metaclasse StorableUnit renomeada. Todas as associações to ou from das instâncias das metaclasses Reads, Writes e Address devem ser renomeadas.
	Structure Package	Se a StorableUnit a ser renomeada estiver contido em uma instância de ClassUnit por meio da associação codeElement, que, por sua vez, está associada a uma instância Package por meio da associação codeElement, a qual está associada a algum elemento arquitetural por meio da associação implementation, então, deve-se verificar se o elemento arquitetural possui uma instância da metaclasse AggregationRelationship. Em seguida, deve-se renomear todas as associações to ou from das instâncias das metaclasses Reads, Writes e Address que utilizam a instância da metaclasse StorableUnit renomeada.
	Data Package	Deve-se identificar uma instância da metaclasse ColumnSet que possui o mesmo meta-atributo name da instância da metaclasse StorableUnit a ser renomeada. Em seguida, deve-se renomear o meta-atributo name da instância da metaclasse ColumnSet.
	Conceptual Package	Não se aplica.
Fim da Tabela		

Finalmente, as regras pré-definidas para as refatorações realizadas em instâncias da metaclasse MethodUnit são apresentadas na Tabela 27.

Tabela 27 – Propagações definidas para refatorações realizadas em instâncias metaclasse `MethodUnit`.

Início da Tabela		
Refatoração	Propagação	
`add MethodUnit`	`Code Package`	Cria-se uma instância da metaclasse `MethodUnit`
	`Action Package`	Não se aplica
	`Structure Package`	Não se aplica
	`Data Package`	Não se aplica.
	`Conceptual Package`	Não se aplica.
`delete MethodUnit`	`Code Package`	Deleta-se uma instância da metaclasse `MethodUnit`.
	`Action Package`	Devem-se remover todas as instâncias das metaclasses `Calls` que utilizam a instância da metaclasse `MethodUnit` deletada.
	`Structure Package`	Se a `MethodUnit` a ser removida estiver contida em uma instância de `ClassUnit` por meio da associação `codeElement`, que, por sua vez, está associada a uma instância `Package` por meio da associação `codeElement`, a qual está associada a algum elemento arquitetural por meio da associação `implementation`, então devem-se verificar se o elemento arquitetural possui uma instância da metaclasse `AggregationRelationship`. Em seguida, deve-se remover todas as instâncias de `Calls` que utilizam a instância da metaclasse `StorableUnit` deletada. Em seguida, deve-se atualizar o meta-atributo `density` da metaclasse `AggregationRelationship`.
	`Data Package`	Não se aplica.
	`Conceptual Package`	Não se aplica.
`change MethodUnit`	`Code Package`	Renomeia o meta-atributo `name` da instância da metaclasse `MethodUnit`.
	`Action Package`	Devem-se alterar todas as metaclasses que utilizam a instância da metaclasse `MethodUnit` renomeada. Todas as associações `to` ou `from` das instâncias das metaclasses `Calls` devem ser renomeadas.

Continuação da Tabela 27		
Refatoração	Propagação	
	`Structure Package`	Se o `MethodUnit` a ser renomeado estiver contido em uma instância de `ClassUnit` por meio da associação `codeElement`, que, por sua vez, está associada a uma instância `Package` por meio da associação `codeElement`, a qual está associada a algum elemento arquitetural por meio da associação `implementation`, então, deve-se verificar se o elemento arquitetural possui uma instância da metaclasse `AggregationRelationship`. Em seguida, devem-se renomear todas as associações `to` ou `from` da instância da metaclasse `Calls`, as quais utilizam a instância da metaclasse `MethodUnit` renomeada.
	`Data Package`	Não se aplica.
	`Conceptual Package`	Não se aplica.
Fim da Tabela		