UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DRIV-UML: VISUALIZAÇÃO DE NÃO-CONFORMIDADES ARQUITETURAIS EM UML NO CONTEXTO DA MODERNIZAÇÃO DIRIGIDA A ARQUITETURA

BRUNO CÉSAR GASPARINI

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP

Abril/2018

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DRIV-UML: VISUALIZAÇÃO DE NÃO-CONFORMIDADES ARQUITETURAIS EM UML NO CONTEXTO DA MODERNIZAÇÃO DIRIGIDA A ARQUITETURA

BRUNO CÉSAR GASPARINI

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

Orientador: Prof. Dr. Valter Vieira de Camargo

São Carlos - SP

Abril/2018

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

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Bruno Cesar Gasparini, realizada em 02/04/2018:

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Auri Marcelo Rizzo Vincenzi
UFSCar

Prof. Dr. Rafael Serapilha Durelli
UFLA

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Rafael Serapilha Durelli e, depois das arguições e deliberações realizadas, o(s) participante(s) à distância está(ao) de acordo com o conteúdo do parecer da banca examinadora redigido neste relatório de defesa.

A minha esposa

AGRADECIMENTOS

Agradeço primeiramente a Deus pela oportunidade concedida de ingressar nessa pós-graduação, por me dar forças e me guiar em todos os momentos da minha vida.

A minha esposa Fernanda Menezes de Souza Gasparini pelo carinho, incentivo e apoio incondicional, não apenas durante o período acadêmico, mas também em todo o tempo que estivemos juntos.

A minha família, especialmente meus pais Carlos Aparecido Gasparini e Fátima Cristina Rodrigues Gasparini pelo amor, cuidado e exemplar criação que me capacitaram para a realização desse sonho.

Ao meu orientador Valter Vieira de Camargo pela dedicação, paciência, profissionalismo e orientação.

A meus companheiros de laboratório por estarem sempre dispostos ajudar e compartilhar de seus conhecimentos.

A CNPQ pelo apoio financeiro. A todo o corpo docente da Universidade Federal de São Carlos.

Agradeço a todos que, diretamente e indiretamente contribuíram com a minha formação.

Cada sonho que você deixa para trás, é

um pedaço do seu futuro que deixa de existir.

Steve Jobs

RESUMO

Como um meio de lidar com sistemas legados, a reengenharia é uma tendência na engenharia de software. Nesse contexto a Modernização dirigida a Arquitetura surgiu como uma forma de se estabelecer consenso entre as diferentes técnicas de reengenharia. Sistemas computacionais são acometidos a uma série de modificações e melhorias, contudo, quando não considerada referência principal, a arquitetura do software pode se deteriorar e ao longo do tempo, se desviar da arquitetura planejada. Neste cenário, este projeto tem como objetivo auxiliar o processo de checagem de conformidade arquitetural, exibindo graficamente em diagramas UML, as não-conformidades arquiteturais detectadas pela ferramenta de Checagem de Conformidade Arquitetural (CCA) ArchKDM. Para realizar tal tarefa, a abordagem recebe um conjunto de violações identificadas pela ferramenta e, a partir dessas, produz os diagramas UML de classes e pacotes. Na abordagem os desvios arquiteturais são ilustrados por meio de relacionamentos UML entre elementos de código (camadas, componentes, classes, etc.). Os elementos de código por sua vez, são ilustrados na forma de elementos UML. É importante mencionar que o propósito da UML no contexto da abordagem é de exibir apenas desvios arquiteturais. A abordagem visa duas perspectivas diferentes de exibição, uma de menor granularidade (fina), cuja a base é um digrama de classes e outra de maior granularidade (grossa), por meio de um diagrama de pacotes. O arquivo de saída da ArchKDM é uma instância do metamodelo KDM contendo as não-conformidades arquiteturais, portanto, com o propósito de auxiliar a abordagem, foram desenvolvidos dois *discoverers* que são executados como uma extensão da ArchKDM para produzir os diagramas. Este projeto é avaliado por um estudo empírico cujo objetivo é expor a ferramenta a um conjunto de sistemas reais utilizados em produção.

Palavras-chave: ADM, KDM, UML, reconciliação arquitetural, desvio arquitetural, diagrama, engenharia dirigida a modelo, transformação de modelos, ATL, não-conformidade arquitetural.

ABSTRACT

As a means of dealing with legacy systems, reengineering has become a trend in software engineering. In this context Architecture-Driven Modernization has emerged as a standard between the different reengineering techniques. Computer systems are affected by a considerable amount of modifications and improvements, however if not considered as a main concern, the software architecture may deteriorate over time and deviate from the intended architecture. In this scenario, this project aims to assist the architectural conformance checking process, displaying graphically by means of UML diagrams, the architectural nonconformities detected by the Architecture Conformance Checking (ACC) tool ArchKDM. In order to accomplish this task, the approach receives a set of violations identified by this tool and, from these, produce UML class and package diagrams. On this approach, the architectural drifts are shown by means of UML relationships between code elements (layers, components, classes, etc.). The code elements are illustrated as UML elements. It is important to mention that UML role in this approach context is to display only architectural drifts. The approach aims two different abstraction perspectives, a fine-grained, based on a class diagrams and another coarse-grained, based on package diagrams. The CCA tool that supports the approach is the ArchKDM. ArchKDM output file is an instance of KDM metamodel containing architectural nonconformities. Thus, in order to support the approach, two discoverers for Modisco plugin will be developed. These discoveries are in charge of recovering and building diagrams from the metamodel instance. This project is evaluated by an empiric study whose purpose is to expose the tool to a diversified set of, already in production, softwares.

Keywords: ADM, KDM, UML, Architectural Reconciliation, diagram, Model-Driven Engineering, Model-To-Model Transformation, ATL, architectural nonconformities.

LISTA DE FIGURAS

Figura 2.1 - Modernização de perspectiva técnica (ULRICH e KHUSIDMAN, 2007)...........18
Figura 2.2 - Modernização de perspectiva de aplicação (ULRICH e KHUSIDMAN, 2007) ..19
Figura 2.3 - Processo de modernização proposto pela ADM 20
Figura 2.4 - Modernização de Perspectiva de Negócio (ULRICH e KHUSIDMAN, 2007)....21
Figura 2.5 - Estrutura do KDM 22
Figura 2.6 - Domínios e Níveis de conformidade do KDM (OMG, 2011) 23
Figura 2.7 - Diagrama de classes do pacote Code (OMG, 2011) 24
Figura 2.8 - Código-fonte de uma classe em java. Adaptado de Durelli (2016) 27
Figura 2.9 - Diagrama de objetos da instância KDM. Adaptado de Durelli (2016) 27
Figura 2.10 - Diagrama de classes do pacote Action. (OMG, 2011).......................................28
Figura 2.11 - Método para ilustração do pacote *Action*. Adaptado de Durelli (2016).............30
Figura 2.12 - Diagrama de objetos para ilustração do pacote *Action* . Adaptado de Durelli (2016) 30
Figura 2.13 - Diagrama de classes do pacote *Structure* (OMG, 2011)...................................32
Figura 2.14 - Exemplo de uma arquitetura simples, representada em código (1), instância pacote Code (2) e visualização arquitetural (3). Adaptado de Durelli (2016) 33
Figura 2.15 - Instância do KDM que representa a Figura 12. Adaptado de Durelli (2016) 34
Figura 2.16 – Código-fonte de associação 35
Figura 2.17 - Associação...36
Figura 2.18 - Código-fonte de dependência...36
Figura 2.19 - Dependência 36
Figura 2.20 - Código-fonte de generalização..37
Figura 2.21 - Generalização 37
Figura 2.22 - Código-fonte de realização 37
Figura 2.23 – Realização..37
Figura 2.24 - Funcionamento geral do Modisco. (BRUNELIÈRE, 2011) 38
Figura 2.25 - Exemplo de Transformação de modelos 40
Figura 2.26 - Código ATL da transformação da Figura 19. Adaptado de Allilaire e Jouault (2007)..41

Figura 3.1 - Matriz de Estrutura de Dependências (DSM - *Dependency-Structure Matrix*) 44
Figura 3.2 - Modelo MVC usando primitivas e elementos de design. (KAMAL e AVGERIOU, 2008) 45
Figura 3.3 - Diagrama de sequência 46
Figura 3.4 - Arquitetura do pilfer 47
Figura 4.1 - Checagem de conformidade arquitetural da ArchKDM 49
Figura 4.2 - Cenário MVC hipotético 52
Figura 4.3 – Modelo KDM ilustrado na perspectiva da IDE Eclipse 52
Figura 4.4 - Modelo KDM (apenas não-conformidades) ilustrado na perspectiva da IDE Eclipse 53
Figura 5.1 - Processo de produção dos diagramas a partir do ArchKDM 55
Figura 5.2 – Exibição de Não-conformidades arquiteturais em UML 56
Figura 5.3 - Sistema hipotético 58
Figura 5.4 – ModelClass 58
Figura 5.5 – ControllerClassA 58
Figura 5.6 - ControllerClassB 59
Figura 5.7 - Diagrama de classes 59
Figura 5.8 - Direção de dependências 60
Figura 5.9 - Wizard de CCA da ArchKDM. 61
Figura 5.10 - Não Conformidades Arquiteturais 61
Figura 5.11 - Função de checagem de relacionamento. 63
Figura 5.12 - Agrupamento de KDMRelationships relacionados 63
Figura 5.13 - Implementação da regra de dependências (Discovery de Desvios entre Elementos Arquiteturais) 67
Figura 5.14 - Implementação da regra de dependência (Discovery de Desvios em Classes) 68
Figura 6.1 - Arquitetura planejada do sistema LabSys 73
Figura 6.2 - Oráculo do LabSys 74
Figura 6.3 – Arquitetura Planejada do MyAppointments 76
Figura 6.4 - Pré-execução do algoritmo de agrupamento (LabSys) 77
Figura 6.5 - Pós-execução do algoritmo de agrupamento (LabSys) 78
Figura 6.6 - Pré-execução do algoritmo de agrupamento (MyAppointments) 79
Figura 6.7 - Pós-execução do algoritmo de agrupamento (MyAppointments) 79
Figura 6.8 - Elementos do diagrama UML para o sistema LabSys 80

Figura 6.9 - Diagrama UML de pacotes para o sistema LabSys 81
Figura 6.10 - Diagrama de desvios em classes para o sistema MyAppointments 81
Figura 6.11 - Diagrama de pacotes para o sistema MyAppointments 82

LISTA DE TABELAS

Tabela 2.1 - *CodeRelationships* do KDM 25
Tabela 2.2 - Mapeamento de elementos do Código-Fonte e elementos KDM. (Adaptado de Santos 2014) 26
Tabela 2.3 - Correspondência entre código e metaclasses no pacote Code. 27
Tabela 2.4 - ActionRelationships do KDM 29
Tabela 2.5 - Correspondência entre código e metaclasses do pacote *Action*. 31
Tabela 5.1 - Helpers 64
Tabela 5.2 - Rules 65
Tabela 6.1 - Desvios inseridos no sistema MyAppointments 75
Tabela 6.2 - Desvios arquiteturais do LabSys 80
Tabela 6.3 - Dados do algoritmo de agrupamento 83
Tabela 6.4 - Relação de desvios identificados 83
Tabela 6.5 - Matriz de dependências do sistema LabSys 84

LISTA DE ABREVIATURAS E SIGLAS

UML - Unified Modeling Language

ADM - Architecture-Driven Modernization

OMG - Object Management Group

ADMTF - Architecture-Driven Modernization Task Force

MDA - Model-Driven Architecture

MDE - Model-Driven Engineering

PIM - Platform Independent Model

PSM - Platform Specific Model

CIM - Computation Independent Model

KDM - Knowledge Discovery Metamodel

SMM - Software Metrics Metamodel

ASTM - Abstract Syntax Tree Metamodel

MOF - Meta Object Facility

XMI - Metadata Interchange

XML - eXtensible Markup Language

UI - User Interface

ATL - Atlas Transformation Language

IDE - Integrated Development Environment

SUMÁRIO

CAPÍTULO 1 - INTRODUÇÃO **13**

1.1 Contexto 13

1.2 Motivações 15

1.3 Objetivos 15

CAPÍTULO 2 - FUNDAMENTAÇÃO TEÓRICA **16**

Considerações Iniciais 16

2.1 Modernização Dirigida a Arquitetura 16

2.1.1 Modernização de Perspectiva Técnica 18

2.1.2 Modernização de Perspectiva de Aplicação 18

2.1.3 Modernização de Perspectiva de Negócio 20

2.2 O metamodelo KDM 21

2.2.1 Pacote Code 24

2.2.2 Pacote Action 28

2.2.3 Pacote Structure 31

2.3 Detalhes de Representação UML 35

2.4 A Ferramenta de Engenharia Reversa Modisco 37

2.5 Linguagem de Transformação de Modelos ATL 39

Considerações Finais 42

CAPÍTULO 3 - TRABALHOS RELACIONADOS **43**

Considerações Iniciais 43

3.1 Visualização de Detalhes Arquiteturais 43

3.2 Recuperação de Modelos UML 45

Considerações Finais 47

CAPÍTULO 4 - CHECAGEM DE CONFORMIDADE ARQUITETURAL COM ARCHKDM **48**

Considerações Iniciais 48

4.1 Checagem de Conformidade Arquitetural 48

4.2 ArchKDM 49

4.2.1 Não Conformidades Arquiteturais 51
Considerações Finais 53
CAPÍTULO 5 - A ABORDAGEM DRIV-UML 54
Considerações Iniciais 54
5.1 Abordagem DriV-UML 54
5.2 Estratégia Adotada para Representar Desvios Arquiteturais em UML 56
5.3 Exemplo de Uso da Abordagem 57
5.4 Detalhes de Implementação 62
5.4.1 Algoritmo de Agrupamento 62
5.4.1.1 Critério de Agrupamento 62
5.4.2 Discoveries 64
5.4.3 Helpers e Rules 64
Considerações Finais 69
CAPÍTULO 6 - AVALIAÇÃO 70
6.1 Estudo Empírico 70
6.1.1 Definição do Estudo Empírico 70
6.1.1.1 Objetivo 70
6.1.1.2 Objeto de estudo 71
6.1.1.3 Abordagem Quantitativa 71
6.1.1.4 Planejamento do Estudo 72
6.1.1.5 Seleção de Contexto 72
6.1.1.6 Instrumentação 73
6.1.2 Operação 73
6.1.2.1 Preparação do LabSys 73
6.1.2.2 Preparação do Sistema MyAppointments 75
6.1.2.3 Execução 76
6.2 Análise do Processo de Agrupamento 76
6.2.1 Sistema LabSys 77
6.2.2 Sistema MyAppointments 78
6.3 Análise da Execução dos Discoveries 79
6.3.1 Sistema LabSys 79
6.3.2 Sistema MyAppointments 81

6.4 Resultados obtidos 82

CAPÍTULO 7 - CONCLUSÕES...85

7.1 Contribuições 86
7.2 Limitações..86

CAPÍTULO 8 - REFERÊNCIAS BIBLIOGRÁFICAS 88

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Warren (1999) descreve "sistema legado" como um sistema cujos custos de manutenção encontram-se muito altos, mas que ainda permanece em operação em virtude de seu papel crítico na organização. Brodie e StoneBraker (1993) afirmam que sistemas legados comumente são resistentes à modificação e evolução, ou seja, apresentam baixa manutenibilidade e não são flexíveis quanto à implementação de novos recursos e funcionalidades. Nesse cenário a reengenharia surge como uma alternativa por apresentar menor custo que uma substituição, por tornar o sistema mais responsivo a mudanças e também por eliminar o risco da perda do conhecimento de negócio adquirido ao longo do tempo de operação do software (WARREN, 1999).

Nesse contexto, em 2003, a OMG propôs a Modernização Dirigida a Arquitetura (ADM – *Architecture-Driven Modernization*) com o objetivo de promover o consenso nos métodos de reengenharia (OMG ADMTF, 2012). Um cenário típico de modernização que segue os princípios da ADM tem seu início no processo engenharia reversa do sistema. Essa etapa visa recuperar um modelo do sistema que represente o software de forma independente da linguagem de programação ou plataforma utilizada. Prossegue com um processo de refatorações e melhorias sob o modelo recuperado na etapa anterior, com o intuito de eliminar as deficiências do sistema original. Por fim tem-se a produção do código fonte a partir do modelo refatorado em um procedimento denominado engenharia avante. Com o intuito de alavancar a definição de modelo independente de plataforma (PIM - *Platform Independent Model*) da Arquitetura Dirigida a Modelos (MDA – *Model Driven Architecture*), a OMG

desenvolveu o KDM (*Knowledge Discovery Metamodel*), desde então esse metamodelo tem se difundido, tornando-se o principal metamodelo da ADM. O propósito do KDM é representar, na forma de modelo, características e artefatos que compõem um software, tais como, código-fonte, arquitetura, plataforma, regra de negócio.

A arquitetura do software é um artefato de grande importância e tem como objetivo prover a estrutura dos componentes do sistema, bem como seus relacionamentos, a fim de adequar os requerimentos comportamentais, não funcionais, de qualidade e ciclo de vida da aplicação (DEITERS, DOHRMANN, *et al.*, 2009; IEEE, 2000). Contudo, ao longo do tempo, a arquitetura do sistema legado tende a se deteriorar e definições estabelecidas inicialmente na arquitetura do software são violadas em um processo conhecido como erosão arquitetural. Como um meio de se avaliar a conformidade arquitetural de um software, diversos autores (CHAGAS, 2016; MAFFORT, VALENTE, *et al.*, 2013; BITTENCOURT, SOUZA, *et al.*, 2010; DEITERS, DOHRMANN, *et al.*, 2009; RAHIMI e KHOSRAVI, 2010) propõem o processo de Checagem de Conformidade Arquitetural (CCA). Streekmann (2012) define CCA como o processo de identificar dependências de código fonte que não estão em conformidade com a arquitetura planejada. O princípio de uma CCA se baseia em mapear um determinado código fonte à elementos arquiteturais (camadas, componentes, sistemas e subsistemas) obtendo-se assim a arquitetura atual do software. Em seguida, a fim de se identificar as não-conformidades arquiteturais existentes, compara-se a arquitetural atual com a arquitetura planejada.

Desenvolvido no laboratório AdvanSE, o ArchKDM[1] é a ferramenta escolhida como pilar desse projeto. Embora essa ferramenta tenha sido selecionada como estudo de caso, a abordagem do projeto visa averiguar a adequabilidade de se utilizar UML para visualização de desvios arquiteturais. A ArchKDM foi desenvolvida como um plugin para a IDE Eclipse[2] e utiliza o metamodelo KDM como base para entrada e saída de dados. O formato KDM por sua vez é textual e contém uma densa quantidade de informações relacionadas ao código fonte, o que torna difícil a visualização e análise dos dados produzidos pelo ArchKDM. Pensando nisso, a abordagem proposta nesse projeto visa contribuir com a forma na qual o resultado de uma CCA é apresentado ao engenheiro de software. No contexto desse projeto vislumbra-se o uso da UML para esse propósito.

---

[1] Disponível em: http://advanse.dc.ufscar.br/index.php/tools/arch-kdm
[2] Disponível em: http://www.eclipse.org/downloads

1.2 Motivações

As motivações que norteiam este trabalho estão listadas abaixo:

- Dificuldade de se visualizar desvios arquiteturais em processos de checagem de conformidade arquitetural;
- Escassez de estudos acerca de visualização de não-conformidades arquiteturais em UML no contexto da ADM. Embora muitos autores (CHAGAS, 2016; MAFFORT, VALENTE, *et al*., 2013; BITTENCOURT, SOUZA, *et al*., 2010; DEITERS, DOHRMANN, *et al*., 2009; RAHIMI e KHOSRAVI, 2010) proponham o processo de checagem de conformidade arquitetural, poucas dão ênfase na forma com que as não-conformidades são apresentadas ao engenheiro de software. Na literatura algumas abordagens, como por exemplo a proposta de Pruijt, Köppe, *et al*. (2014), utilizam notações UML na representação da Arquitetura Planejada e Arquitetura Atual, contudo, após o processo de CCA, as violações são exibidas de maneira textual.

1.3 Objetivos

Este projeto de mestrado possui os seguintes objetivos:

- Facilitar a visualização das não-conformidades arquiteturais existentes em um sistema possibilitando que engenheiros de software analisem, graficamente, detalhes arquiteturais de um determinado software permitindo assim, que projetos de modernização, sejam feitos de forma mais adequada. Por meio de uma representação gráfica o engenheiro poderá realizar uma análise mais detalhada do sistema em busca de problemas que em muitos casos, não estão explícitos em um projeto quando analisado diretamente pelo código fonte.
- Fornecer um apoio computacional baseado em UML para visualização de não-conformidades arquiteturais no contexto do projeto ArchKDM.
- Contribuir para a ADM por meio do fornecimento de um aparato ferramental que facilite a condução de modernizações arquiteturais.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Considerações Iniciais

Na Seção 2.2 é apresentada a Modernização Dirigida a Arquitetura e suas perspectivas. Na Seção 2.3 o metamodelo KDM é descrito. Na Seção 2.4 é apresentado o processo de Checagem de Conformidade Arquitetural da ferramenta ArchKDM bem como as possíveis não-conformidades arquiteturais detectadas pela mesma. Na Seção 2.5 é discutido conceitos de relacionamentos UML. Na Seção 2.6 é discutido o *plugin* Modisco. Por fim na Seção 2.7 é apresentada a linguagem ATL.

2.1 Modernização Dirigida a Arquitetura

Estudos realizados pelo *Standish Group International* (2008) estimam que apenas 53% dos projetos de reengenharia são finalizados com sucesso e dentro do prazo estimado. Dentre os 47% restantes, até 8% são descontinuados. Em seu relatório *Reengineering Project Failure Analysis*, Bergey *et al*. (1999) faz referência a um conjunto de potenciais problemas responsáveis por tais números, dentre eles merecem destaque os seguintes:

- A organização inadvertidamente adota uma estratégia de reengenharia falha ou incompleta;
- A força de trabalho está fadada a tecnologias antigas com programas de treinamento inadequados;

- A organização não tem seu software legado sob controle;
- A arquitetura do software não é uma consideração primária do processo de reengenharia;
- Não há noção de um separado e distinto processo de reengenharia;
- Não há um planejamento adequado ou não há determinação suficiente para seguir os planos.

Sneed (2005) atribui a causa a outros dois problemas básicos: (i) falta de padronização e a (ii) não-automatização do processo, sendo este último considerado pelo autor, uma consequência do primeiro. Em outros termos, Sneed afirma que por consequência da falta de padronização, o desenvolvimento de ferramentas de automatização do processo de reengenharia se torna impraticável, consequentemente, as organizações se sentem forçadas a optar por soluções *adhocs*, essas por sua vez imaturas, porém de menor custo.

Com tais problemas em mente, tornou-se evidente a necessidade de um conjunto de especificações e padrões consolidados que auxiliassem o engenheiro de software no processo de reengenharia. Com a iniciativa de criar tais especificações e promover o consenso na modernização de aplicações existentes (OMG ADMTF, 2012) em 2003 por meio da ADMTF, a OMG idealizou a ADM. Segundo a OMG (2012), os objetivos principais da ADM são:

- Consolidar melhores práticas de modernização a fim de permitir que este processo seja bem-sucedido.
- Tornar aplicações existentes mais ágeis.
- Permitir a revitalização de aplicações já existentes.
- Alavancar o uso de padrões de modelagem e a iniciativa MDA (*Model-Driven architecture*).

O processo de modernização proposto pela ADM é baseado na recuperação e transformação de modelos, sendo dividido em domínios, que por usa vez são subdivididos em perspectivas arquiteturais. Os domínios são Domínio de Negócio (DN) e Domínio de tecnologia (DT). O DN é composto apenas pela perspectiva arquitetural de negócio, já o DT é subdividido em outras duas perspectivas: Aplicação e Técnica. É importante ressaltar que esse processo é frequentemente retratado por meio do Modelo de Reengenharia Ferradura (*Horseshoe*) proposto por Kazman *et al*. (1998) e compartilha o paradigma de evolução incremental imposto pelo mesmo. Detalhes a respeito das três perspectivas citadas serão discutidas nos tópicos seguintes.

2.1.1 Modernização de Perspectiva Técnica

A modernização com perspectiva de arquitetura técnica é historicamente a mais utilizada, sendo comumente motivada pelo risco de obsolescência, custos, usabilidade e fatores relacionados a mudanças físicas (ULRICH e KHUSIDMAN, 2007). Exemplos de modernizações técnicas são: mudança da plataforma suportada, troca da linguagem de programação (desde que não ocorra modificação do paradigma da linguagem original), substituição da interface do usuário, etc.

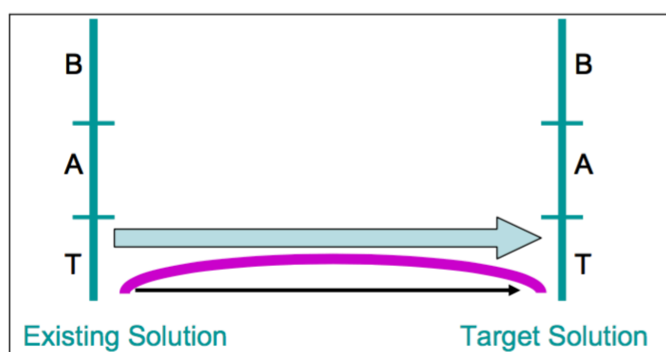

Figura 2.1 - Modernização de perspectiva técnica (ULRICH e KHUSIDMAN, 2007)

Na Figura 2.1 está uma ilustração do processo no modelo *Horseshoe*. Como pode-se notar, o processo de modernização é dividido em três perspectivas: *Business* (B), compondo o DN, *Application* (A) e *Technical* (T), que por sua vez compõem o DT. A abordagem do modelo *Horseshoe* propõe que, dado um sistema, esse chamado de solução existente (*existing solution*), o mesmo seja submetido a um conjunto pré-definido de modificações e melhorias, até que a solução alvo (*target solution*) seja correspondida. A solução alvo, nesse caso, é uma versão modernizada e livre dos problemas encontrados na solução existente. É importante ressaltar que na modernização de perspectiva técnica, as melhorias impostas sobre o sistema não atingem o nível de aplicação, ou seja, não impactam nem modificam a arquitetura do software e também que, nesse nível do processo, os modelos propostos pela iniciativa MDA (PSM, PIM, CIM) ainda não são aplicados.

2.1.2 Modernização de Perspectiva de Aplicação

A modernização na perspectiva de arquitetura de aplicação ocorre quando o design do software está envolvido e apenas as alterações de perspectiva técnica não são suficientes para a solução dos problemas detectados no software. Nesse ponto se torna necessária uma nova

implementação do sistema e de sua arquitetura. Ulrich e Khusidman (2007) cita como exemplo do uso desse processo o cenário no qual, dado um sistema, composto por um conjunto de aplicações, surge a necessidade de reconstruí-lo na forma de um aplicativo único e comum, com modelo de dados redesenhado e acoplado à plataforma de migração. Uma ilustração da modernização na perspectiva da aplicação é ilustrada por meio do modelo *Horseshoe* na Figura 2.2.

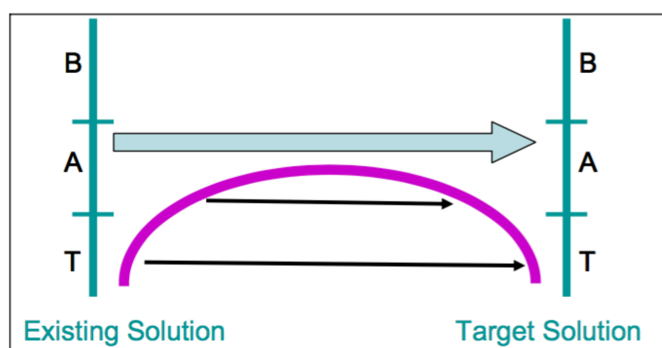

Figura 2.2 - Modernização de perspectiva de aplicação (ULRICH e KHUSIDMAN, 2007)

Neste nível da abordagem, surgem os conceitos de modelos da MDA, especificamente os modelos: *Platform-Independent Model* (PIM) e *Platform-Specific Model* (PSM). Pastor e Molina (2007) se referem a PIM como um modelo que descreve a aplicação sem detalhes de implementação, ou seja, um modelo que apresenta o sistema em um nível intermediário de abstração (PÉREZ-CASTILLO, GUZMÁN e PIATTINI, 2011) e mais próximo da regra de negócio. Por sua vez, o PSM representa o sistema de forma mais refinada, incluindo detalhes de implementação (PASTOR e MOLINA, 2007) sendo assim mais próximo ao código (KLEPPE, WARMER e BAST, 2003) e em um nível de abstração de menor ordem.

Com o propósito de concretizar a utilização dos conceitos de PIM e PSM e padronizar o processo de modernização, a OMG desenvolveu seus representantes na forma de metamodelos, são eles o KDM e ASTM. A Figura 2.3 ilustra o processo com os respectivos conceitos (PIM e PSM) da MDA apropriadamente aplicados (ULRICH e KHUSIDMAN, 2007).

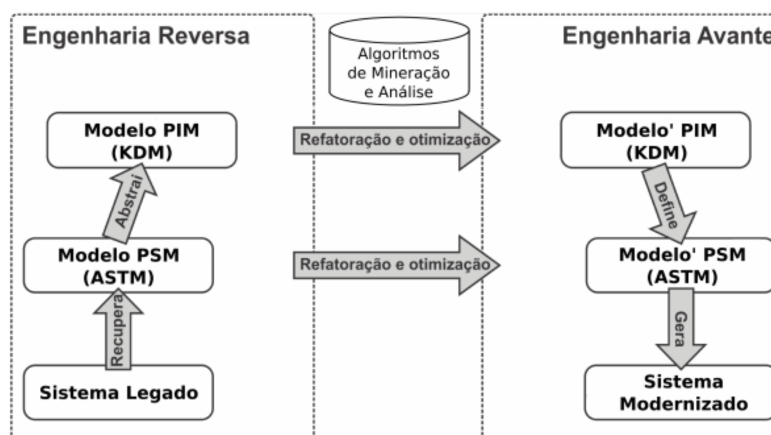

Figura 2.3 - Processo de modernização proposto pela ADM

O processo tem seu início com a engenharia reversa do sistema legado. Nesse momento vislumbra-se recuperar uma instância do metamodelo ASTM (PSM). Essa etapa é automatizada e realizada de forma transparente por algumas ferramentas, como por exemplo, o Modisco. Em seguida a instância do ASTM (PSM) é abstraída em uma instância do metamodelo KDM (PIM). Neste ponto, com o propósito de se identificar problemas do sistema legado, aplicam-se Algoritmos de Mineração e Análise sob esta instância. Em seguida inicia-se o processo de refatoração e otimização, com o intuito de corrigir os problemas identificados na etapa anterior. Por fim, com um KDM refatorado e otimizado, realiza-se a engenharia avante. No processo de engenharia avante será definido uma instância do metamodelo ASTM, este que por sua vez, servirá como base para a geração do código modernizado.

2.1.3 Modernização de Perspectiva de Negócio

A perspectiva de negócio envolve a solução mais abrangente e compreende além dos modelos arquiteturais de negócio, os modelos de aplicação e técnico (ULRICH e KHUSIDMAN, 2007). Ou seja, o processo visa a modernização de todos os níveis de abstração. Neste contexto surge o conceito de *Computation-Independent Model* (CIM), frequentemente chamado de modelo de negócio ou de domínio (OMG, 2003). O CIM apresenta o sistema do ponto de vista mais abstrato de todo o processo, agindo como ponte entre o sistema e os especialistas de negócio (TRUYEN, 2006). No modelo *Horseshoe* o percurso do processo de modernização ultrapassa o limite do DT e atinge o DN, na Figura 2.4 está ilustrado esse cenário.

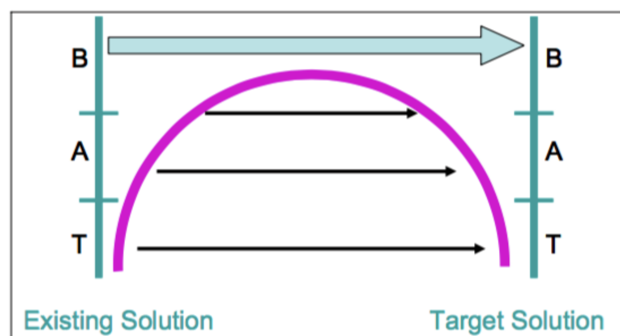

Figura 2.4 - Modernização de Perspectiva de Negócio (ULRICH e KHUSIDMAN, 2007)

2.2 O metamodelo KDM

Um amplo conjunto de ferramentas de mineração são capazes de extrair conhecimento de um software, nesse contexto, surge a necessidade do desenvolvimento de um meio de se armazenar as informações recuperadas. Cientes desse fato, a OMG em 2009, concebeu o KDM (*Knowledge Discovery Metamodel*). O objetivo do KDM é prover uma forma padronizada de se armazenar o conhecimento de um dado sistema, promovendo a interoperabilidade entre ferramentas de modernização (OMG, 2016), facilitando assim, a troca de informações e cooperação entre elas.O KDM é descrito pelo modelo *Meta-Object Facility (MOF)* e utiliza o padrão XML em conformidade com o esquema KDM XMI para armazenar os elementos do software (OMG, 2011)

A organização do KDM é feita nas seguintes camadas: (i) *Infrastructure Layer* (CI - Camada de Infraestrutura, (ii) *Program Elements Layer* (CEP - Camada de Elementos do Programa), (iii) *Runtime Resource Layer* (CRTP - Camada de Recurso de Tempo de Execução), (iv) *Abstraction Layer* (CA - Camada de Abstração). Cada camada por sua vez, é subdividida em pacotes. O CI é composto pelos pacotes *Core*, *"kdm"* e *Source*, o CEP se divide em *Action* e *Code*, o CRTE contém os pacotes *Platform*, UI, *Event* e *Data* e a CA os pacotes *Structure*, *Conceptual* e *Build*. Na Figura 2.5 está retratada a organização descrita acima.

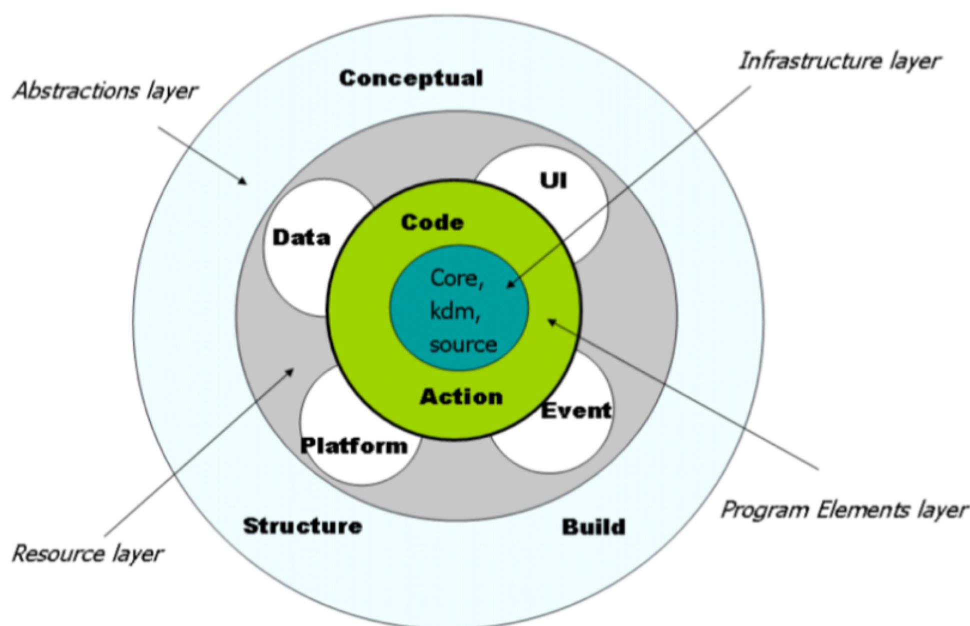

Figura 2.5 - Estrutura do KDM (OMG, 2011)

No KDM, fatos distintos do conhecimento de uma aplicação são agrupadas em domínios, cada domínio define um ponto de vista arquitetural (OMG, 2011) e um interesse específico. A linguagem para se descrever cada domínio está definida no conjunto de pacotes que representam este domínio, ou seja, o conjunto de pacotes que contém os elementos necessários para representar o respectivo domínio. Por exemplo, o conjunto de pacotes *Code* e *Action* possuem os elementos (variáveis, funções e procedimentos) que são utilizados para descrever o Domínio de Código (*Code Domain*). Outro exemplo é o pacote *Structure*, nesse pacote se encontram os elementos (sistemas, subsistemas e componentes) utilizados para representar o Domínio de Estrutura (*Structure Domain*). A vantagem em separar o conhecimento do sistema em domínios está no fato de que, desse modo, o usuário não precisa se preocupar com domínios não relacionados à sua área de estudo enquanto utiliza o KDM. Na Figura 2.6 estão representados os Domínios existentes no KDM bem como os níveis de conformidade que uma ferramenta de modernização pode aderir.

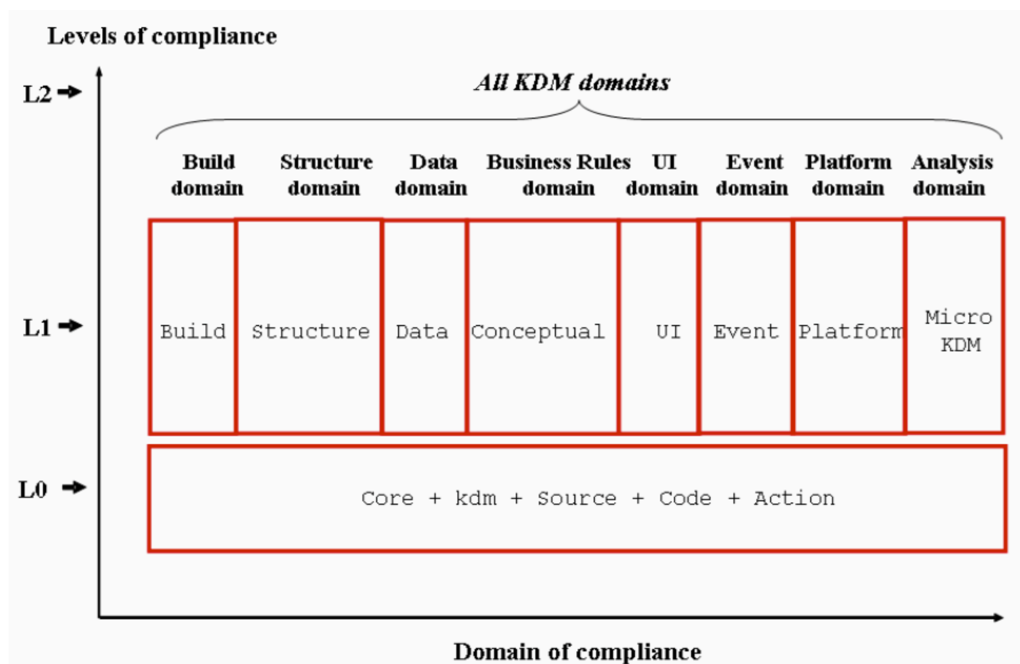

Figura 2.6 - Domínios e Níveis de conformidade do KDM (OMG, 2011)

O nível de conformidade L0 é composto pelo Domínio de Código (*Code domain*) que por sua vez é formado pelos pacotes de Infraestrutura do KDM (*Core*, *kdm* e *Source*) e os pacotes *Code* e *Action*. O nível L0 representa o nível de conformidade básico para qualquer ferramenta de software, isso quer dizer que, se uma dada ferramenta possui intenção de dar suporte a qualquer domínio dos níveis de conformidade superiores (L1 ou L2), ela deve obrigatoriamente estar em conformidade com L0.

O nível de conformidade L1 representa aspectos mais específicos de uma aplicação e estende as capabilidades do nível L0 (OMG, 2011). Esse nível permite o desenvolvimento de aplicações que possuem diferentes focos, isso é possível em consequência da separação dos domínios por interesses (*concerns*), como mencionado anteriormente. Um exemplo prático seria o desenvolvimento de uma ferramenta de teste de UI, nesse caso o foco se daria apenas no Domínio de Código e de UI. Por não serem pertinentes ao desenvolvimento da ferramenta, os outros domínios não precisam, obrigatoriamente, estarem envolvidos. Os domínios que compreendem o nível L1 são:

- Domínio de Construção (*Build domain*): composto pelo pacote *Build*.
- Domínio de Estrutura (*Structure domain*): composto pelo pacote *Structure*.
- Domínio de Dados (*Data domain*): composto pelo pacote *Data*.
- Domínio de regras de negócio (*Business Rules domain*): composto pelo pacote Conceptual.
- Domínio de Interface do Usuário (*UI domain*): composto pelo pacote *UI*.

- Domínio de Evento (*Event domain*): composto pelo pacote *Event*.
- Domínio de Plataforma (*Platform domain*): composto pelo pacote *Platform*.
- Domínio de Análise (*Analysis domain*): composto pelo pacote *MicroKDM*.

O nível de conformidade L2 representa o suporte de todos os domínios do KDM disponíveis. Uma ferramenta em conformidade com este nível, deve necessariamente dar suporte a todos os domínios do KDM. Este projeto está no escopo dos Domínios de Código (nível L0) e de Estrutura (nível L1) logo, a fundamentação teórica irá discutir detalhes apenas dos pacotes *Code*, *Action* e *Structure*.

2.2.1 Pacote Code

O pacote *Code* possui um conjunto de metaclasses cujo objetivo é descrever o software em nível de código (implementação). Nesse pacote se encontram um amplo número de metaclasses que representam elementos adotados de forma consensual por uma quantidade considerável de linguagens de programação, tais como: Classes, Tipos de dados, *Actions*, Macros, *templates* e protótipos. Em uma instância do KDM, cada elemento do pacote *Code* é equivalente a um elemento ou estrutura da linguagem de programação utilizada no software. O pacote *Code* possui um total de 24 diagrama de classes, o diagrama principal, *CodeModel*, está ilustrado na Figura 2.7.

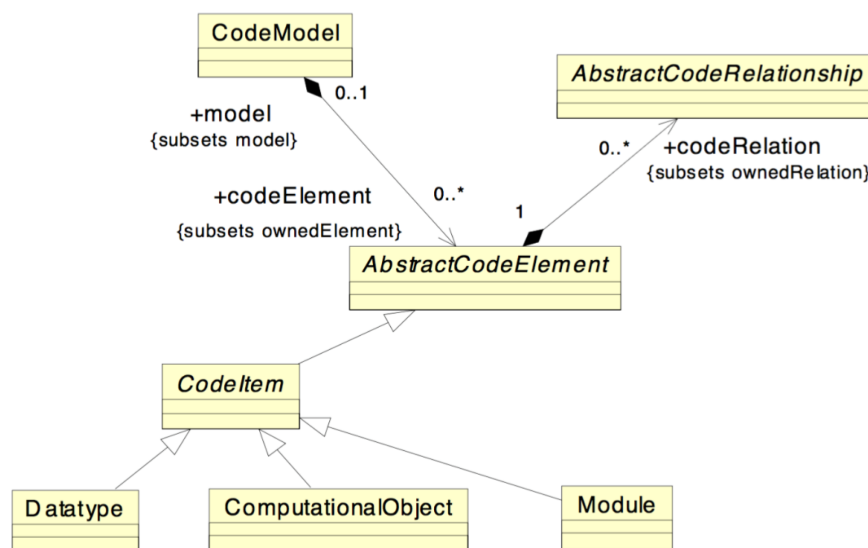

Figura 2.7 - Diagrama de classes do pacote Code (OMG, 2011)

A metaclasse *CodeModel* representa um recipiente para instâncias de elementos de código (*code elements*), ou seja, é um modelo que armazena um conjunto de fatos do sistema.

É importante mencionar que tanto os elementos de código quanto o *CodeModel* são dependentes dos pacotes *kdm*, *Source* e *Core* (Figura 2.5). Os elementos de código se subdividem em duas metaclasses: (i) *CodeItem*, representando os símbolos e definições da linguagem e (ii) *AbstractionCodeRelatioship* que define uma metaclasse abstrata para representar os relacionamentos estabelecidos entre os elementos de código. Os relacionamentos de código existentes no KDM estão sintetizados na Tabela 2.1. O elemento *CodeItem* se subdivide em outras três subclasses: (i) *DataType*, para descrever os tipos de dados, (ii) *Module*, que representa arquivos compartilhados, unidades compiladas e componentes binários, por fim, (iii) *ComputationalObject* para caracterizar definições de linguagem tais como procedimentos e variáveis (OMG, 2011) métodos e funções.

Tabela 2.1 - *CodeRelationships* do KDM

InstanceOf	Representa um relacionamento entre uma referência a um tipo de dados parametrizado ou uma entidade parametrizada (por exemplo, um método genérico), à correspondente declaração na classe parametrizada.
ParameterTo	Elemento do metamodelo que representa o parâmetro de tipo real no contexto de uma referência a uma entidade parametrizada.
Implements	Elemento do metamodelo que representa o relacionamento de implementação entre um *CodeItem* (uma classe por exemplo) e uma *InterfaceUnit*. (Equivalente ao implements da linguagem Java)
ImplementationOf	Representa o a associação entre a declaração e a definição de um objeto computacional (*ComputationObject*).
HasType	Elemento específico do metamodelo que representa a relação semântica entre o elemento de dados e seu elemento de tipo.
HasValue	Elemento específico do metamodelo que representa a relação semântica entre o elemento de dados e seu elemento de dados de inicialização.
Extends	Representa a relação semântica entre duas classes, onde uma (chamada de descendente) estende as capabilidades de outra por meio de herança.
Expands	Essa classe representa o relacionamento entre um elemento da metaclasse *MacroUnit* e outro *MacroUnit* ou de um elemento *MacroDirective* e um *MacroUnit*.
GeneratedFrom	Representa o relacionamento entre um bloco de elemento de código que não foi originalmente desenvolvido por desenvolvedores, mas são produzidos como resultado do pré-processamento de uma diretiva de pré-processo.
Includes	Classe que representa o relacionamento de uma diretiva *include* para uma *SharedUnit*, essa que por sua vez representa os elementos de código sendo incluídos.
VariantTo	Representa o relacionamento entre variantes de uma linha de produção de software que apresentam compilação condicional. Essa metaclasse conecta a diretiva condicional a cada ramificação da diretiva de compilação condicional.

Redefines	Representa o relacionamento onde um *MacroUnit* e outro *MacroUnit* onde o primeiro membro é uma redefinição do segundo.
VisibleIn	É um elemento específico do metamodelo que representa a relação entre dois itens de código onde um provê visibilidade de contexto restrita para o outro.
Imports	Esse elemento do metamodelo representa uma associação entre dois itens de código (*CodeItem*) onde um importa definições de outro. (Exemplo: *import* usado em Java)
CodeRelationship	Classe genérica para ser usado como base para novos relacionamentos de código por meio do mecanismo de extensão do KDM.

O pacote *Code* possui um amplo conjunto de metaclasses cujo objetivo é representar cada elemento do código fonte. É importante mencionar que em alguns casos é possível identificar facilmente os elementos de código e sua metaclasse correspondente, como por exemplo a classe *ClassUnit*, que nesse contexto representa Classes, ou *InterfaceUnit* que representa Interfaces. Contudo, alguns elementos podem não se apresentar de forma tão óbvia variando de acordo com a implementação do *Discovery* KDM. Por exemplo o *StorageUnit*, que na implementação do *Discovery* KDM do Modisco, além de representar atributos também descrevem variáveis locais. A Tabela 2.2 exibe o mapeamento entre alguns elementos de código e suas respectivas metaclasses.

Tabela 2.2 - Mapeamento de elementos do Código-Fonte e elementos KDM. (Adaptado de Santos 2014)

Elemento de Código-Fonte	**Metaclasse do KDM**
Classe	ClassUnit
Interface	InterfaceUnit
Método	MethodUnit
Atributos/Variáveis	StorableUnit
Parâmetro	ParameterUnit
Associação	KdmRelationship

Com o propósito de facilitar o entendimento, na Figura 2.9 está ilustrado uma instância KDM do código-fonte em Java descrito na Figura 2.8. A instância KDM está ilustrada em um diagrama de objetos de metaclasses (instância de metaclasses). O agregado *Segment* representa um container para os modelos KDM dessa instância KDM. É importante ressaltar que esse agregado pode armazenar modelos com especialidades distintas, como por exemplo: *CodeMode*, *StructureModel*.

```java
1  package com.br.model;
2
3  public class Car extends Vehicle{
4
5      private String name;
6
7      public String getName(){
8          ...
9      }
10
11 }
12
```

Figura 2.8 - Código-fonte de uma classe em java. Adaptado de Durelli (2016)

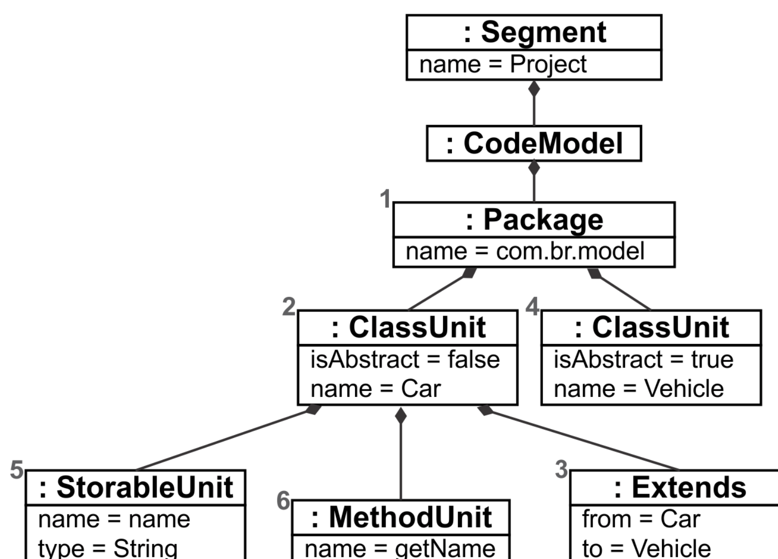

Figura 2.9 - Diagrama de objetos da instância KDM. Adaptado de Durelli (2016)

O código-fonte na Figura 2.8 apresenta uma classe *Car* (Linha 3), pertencente ao pacote *com.br.model* (Linha 1), com um atributo privado definido como *name* do tipo String (Linha 5). Nota-se também um método de acesso a esse atributo, de nome *getName* cujo tipo de retorno também é um *String* (Linha 7). Por fim, a classe *Car* possui como ancestral uma classe abstrata chamada *Vehicle* (Linha 3). A instância do KDM destaca um Package (1), dois ClassUnits (2 e 4), um StorableUnit (5), um MethodUnit (6) e um Extends (3). Os itens na Tabela 2.3 apresentam o relacionamento entre os elementos de código e suas respectivas instâncias de metaclasse.

Tabela 2.3 - Correspondência entre código e metaclasses no pacote Code.

Elemento do código-fonte	**Instância da metaclasse no KDM**
Pacote *com.br.model* (Linha 1)	1. Package (name: “com.br.model”)
Classe *Car* (Linha 3)	2. ClassUnit (isAbstract: false, name: “Car”)
Classe *Vehicle* (Linha 3)	4. ClassUnit (isAbstract: true, name: “Vehicle”)

Atributo *name* (Linha 5)	5. StorableUnit (name: "name", type: "String"
Método *getName* (Linha 7)	6. MethodUnit (name: "getName")
Herança Car *extends Vehicle* (Linha 3)	3. Extends (from: "Car", to: "Vehicle")

2.2.2 Pacote Action

O pacote *Action* define um conjunto de elementos do metamodelo cujo propósito é representar as descrições de comportamento de nível de implementação determinadas pelas linguagens de programação, como por exemplo declarações, operadores, condições, bem como suas associações, como por exemplo fluxo e controle de dados (OMG, 2011).

O pacote *Action* é formado por um conjunto de 11 diagramas sendo o principal deles o *ActionModel*. Diferentemente do diagrama *CodeModel*, o *ActionModel* não possui uma metaclasse recipiente do tipo *Model*. O motivo da ausência dessa metaclasse se deve ao fato de que o *ActionModel* foi criado com o intuito de complementar e pacote *Code*, ou seja, seu objetivo é de descrever o comportamento dos elementos de código deste pacote. Com isso em mente, nota-se um relacionamento entre o pacote *Action* e *Code*. Esse relacionamento se trata de uma agregação de elementos de código (AbstractCodeElement) ao elemento de ação (ActionElement). Como pode ser visto na Figura 2.10 a classe *ActionElement* possui um atributo denominado *codeElement*. Esse atributo é tipado com a classe *AbstractCodeElement*, e que por sua vez é proveniente do pacote *Code*.

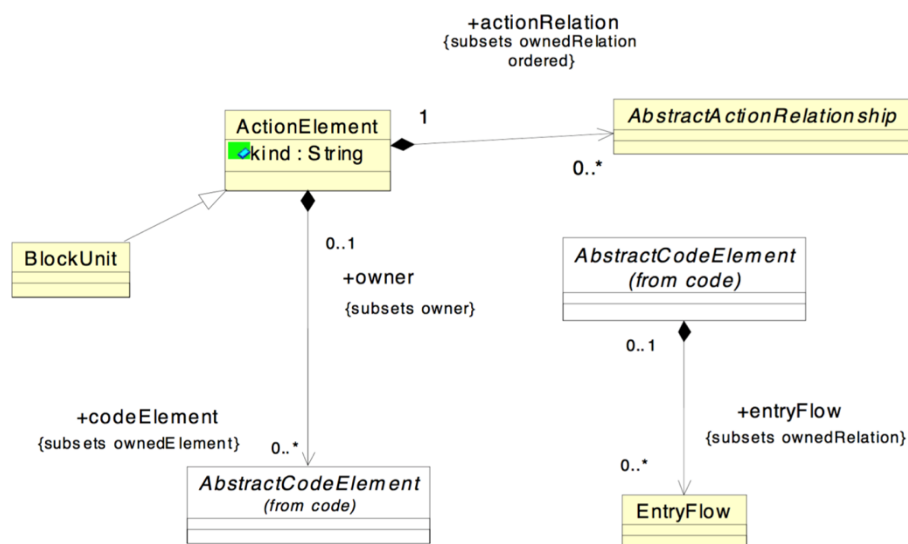

Figura 2.10 - Diagrama de classes do pacote Action. (OMG, 2011)

A metaclasse *ActionElement* é classe utilizada para se descrever uma unidade básica de comportamento (OMG, 2011). Como pode-se notar na Figura 2.10, essa metaclasse é um container para elementos de código (*CodeElement*) e relacionamento de ações, representado pela metaclasse *AbstractActionRelationship*. É importante ressaltar que além de elementos de código, o *ActionElement* pode armazenar inclusive, outros elementos de ação. Isso é possível devido ao fato de que os elementos armazenados pela metaclasse *ActionElement* são tipados como *AbstractCodeElement*, que por sua vez, é a metaclasse ancestral tanto de *CodeElement*, como também da *ActionElement*. Na Tabela 2.4 foi sintetizado, de acordo com a OMG (2011), as metaclasses de relacionamento (*ActionRelationships*), existentes no KDM,.

Tabela 2.4 - ActionRelationships do KDM

ControlFlow	Elemento de modelagem genérico que representa a relação de fluxo de controle entre dois elementos de ação (*ActionElements*).
EntryFlow	Elemento de modelagem que representa um fluxo inicial de controle dentro de um elemento de controle (por exemplo, um *MethodUnit*) para o primeiro elemento de ação.
Calls	Representa o relacionamento entre um elemento de ação (Ex: invocação) e um elemento de controle (Ex: método).
Dispatches	É um elemento de modelagem que representa um relacionamento entre um elemento de ação e um item de dados.
Reads	Representa o relacionamento entre um fluxo de dados de um elemento de dado para um elemento de ação. (Ex: acesso de leitura do elemento de dados)
Writes	Representa o fluxo de dados de um elemento de ação para um elemento de dados. (Ex: acesso de escrita a um elemento de dados).
Addresses	Representa o acesso a estrutura de dados complexas, onde os relacionamentos *Reads* ou *Writes* são aplicados ao endereço dessa estrutura.
Creates	Representa a associação entre um elemento de ação e um tipo de dado, em que o elemento de ação cria uma nova instância desse tipo.
ExitFlow	Elemento do metamodelo que representa um fluxo de controle implícito que deve ser invocado quando o bloco em execução foi terminado normalmente ou abruptamente. (Ex: sessão `finally` em um bloco `try{}catch{}finally{}` em Java)
Throws	Representa a associação entre um elemento de ação, que lança uma determinada exceção, e o elemento de dados associado a essa exceção.
UsesType	Representa uma conversão de tipo (*typecast*) realizada por um elemento de ação.
ActionRelationship	Classe genérica para ser usado como base para novos relacionamentos de ação por meio do mecanismo de extensão do KDM.

Na Figura 2.12 está ilustrado o KDM, na perspectiva do pacote *action*, para o código-fonte na Figura 2.11. A ilustração é feita por uma estrutura de árvore, onde a raiz é o elemento de código *MethodUnit*. É importante observar que quanto maior a profundidade de um nó, maior é a granularidade da representação do código, contudo, esta granularidade pode variar. O motivo disso é que, segundo KDMAnalytics (2016), a responsabilidade de selecionar a granularidade dos elementos de ação é pertinente ao implementador do *discovery* KDM, portanto, *discoveries* provenientes de diferentes fornecedores, podem apresentar diferentes granularidades. No exemplo da Figura 2.12, apenas os elementos que representam o método e1 (Linha 12) são retratados.

```
11      ...
12      public void e1 () {
13          Car myCar = new Car();
14          myCar.getName();
15      }
16      ...
```

Figura 2.11 - Método para ilustração do pacote *Action*. Adaptado de Durelli (2016)

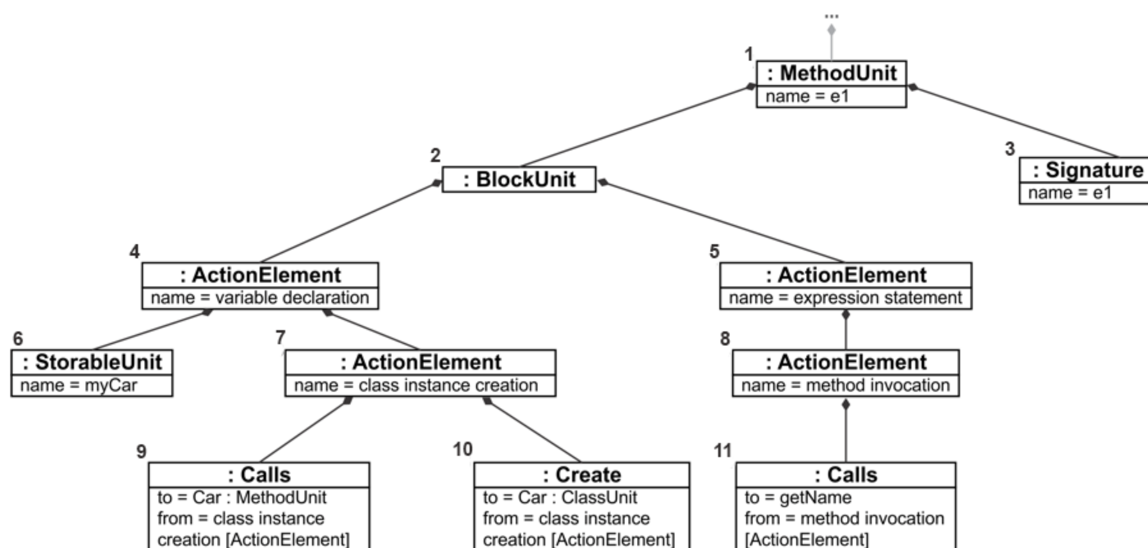

Figura 2.12 - Diagrama de objetos para ilustração do pacote *Action* . Adaptado de Durelli (2016)

O objeto *MethodUnit* (1) é a metaclasse que representa o método como um todo, dentro deste se encontram a assinatura do método, ilustrado pelo objeto *Signature* (3) e o bloco de código, ilustrado pelo objeto *BlockUnit* (2). A descrição de cada elemento do bloco de código encontra-se abaixo e na Tabela 2.5 estão relacionados os elementos ao trecho de código fonte (Figura 2.11) correspondente:

- ActionElement (4): indica que dentro do escopo do *BlockUnit* (3) está ocorrendo a declaração de variável.

- StorableUnit (6): é o elemento de código que representa a variável da declaração, armazenando o nome da respectiva variável (*myCar*) no atributo name.
- ActionElement (7): representa a ação que está sendo atribuída a variável *myCar*, no caso do exemplo, a instanciação de uma classe.
- Calls (9): indica a invocação de um método, neste caso o construtor *Car*, este que por sua vez é referenciado pelo atributo *to*. O atributo *from* representa a origem da invocação, neste exemplo, a instância da classe.
- Create (10): diz que a invocação realizada em d) é uma instanciação de classe, neste caso a classe *Car*, referenciada pelo atributo *to*. O atributo *from* representa a origem da invocação, neste exemplo, a instância da classe.
- ActionElement (5): indica que uma expressão está sendo executada.
- ActionElement (8): indica que a expressão executada é a invocação de um método.
- Calls (1): representa o método que está sendo invocado, nesse caso o método getName, indicado pelo atributo *to*. O atributo *from* diz a origem da invocação, neste exemplo, a instância da classe.

Tabela 2.5 - Correspondência entre código e metaclasses do pacote *Action*.

Elemento do código-fonte	Instância da metaclasse no KDM
`Car myCar = new Car();`	4. *ActionElement* (name: “variable declaration”)
`Car myCar`	6. *StorableUnit* (name: ”myCar”)
`new Car()`	7. *ActionElement* (name: ”class instance creation”):
`new`	9. *Create* (to: “Car: ClassUnit”, from: “class instance”)
`Car()`	10. *Calls* (to: “Car: MethodUnit”, from: “class instance”)
`myCar.getName();`	5. *ActionElement*(name: “expression statement”):
`myCar.getName()`	8. *ActionElement*(name: “method invocation”):):
`getName()`	11. *Calls*(to: “getName”, from: “method invocation”)

2.2.3 Pacote Structure

O pacote *Structure* define componentes para descrever a organização de um software em um alto nível de abstração (KDMANALYTICS, 2016), ou seja, seu objetivo é apresentar aspectos arquiteturais do sistema, tais como camadas, componentes e subsistemas, bem como ilustrar o relacionamento existente entre eles. O pacote *Structure* é composto por 3 diagramas de classes, depende dos pacotes core e kdm (OMG, 2011) e trabalha em conjunto com os

pacotes *Code*, *Data*, *Platform*, *UI* e *Inventory*. O principal diagrama do pacote *Structure*, *StructureModel* está ilustrado na Figura 2.13

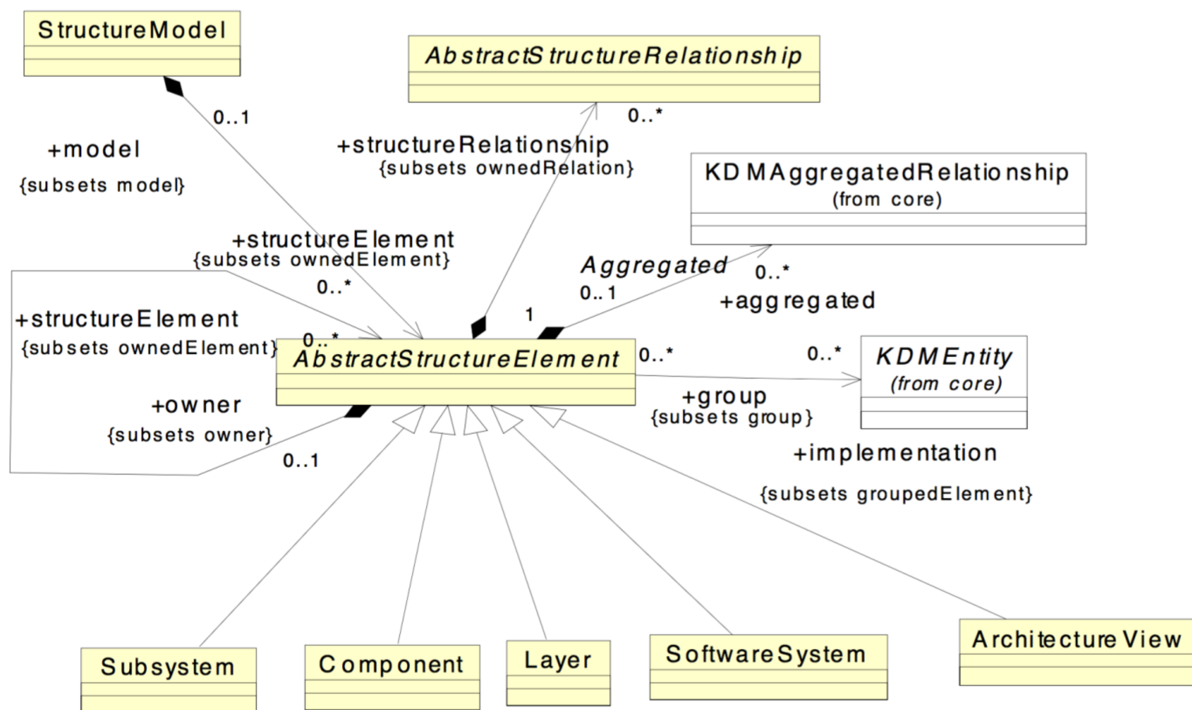

Figura 2.13 - Diagrama de classes do pacote *Structure* (OMG, 2011)

Seguindo o padrão do KDM o diagrama possui uma classe do tipo *Model* denominada *StructureModel* que caracteriza o modelo do diagrama. Essa classe possui um conjunto de elementos estruturais (*StructureElements*) que representam partes da arquitetura do software. A classe *AbstractStructureElement* representa um ancestral comum para cada um desses elementos e pode se especializar nas seguintes metaclasses:

- *SoftwareSystem*: Constitui a organização de todo o software, agindo como um ponto de encontro para todos os pacotes do sistema diretamente, ou indiretamente por meio de elementos estruturais.
- *Subsystem*: Descreve os subsistemas que compõem o *SoftwareSystem*.
- *Layer*: Consiste em partes do *Subsystem* que representam uma camada de software.
- *Component*: Corresponde, direta ou indiretamente a recursos de código.
- *ArchitectureView*: Representa o empacotamento lógico dos artefatos de software relacionados a uma visão particular do sistema.

Além de servir como metaclasse base para os elementos estruturais, a classe *AbstractStructureElement* define 4 associações a serem herdadas por cada elemento estrutural. A primeira associação é denominada *structureElement: AbstractStructureElement[0..*]* e

representa elementos estruturais pertencentes àquele elemento, logo, conclui-se que, um elemento estrutural pode conter outros elementos em sua composição. A segunda associação é chamada de *structureRelationship: AbstractStructureRelationship[0..*]*, e constitui os relacionamentos em nível arquitetural do respectivo elemento. A terceira associação, aggregated: AggregatedRelationship[0..*], possui o propósito de relacionar diferentes elementos do KDM, sejam esses relacionamentos abstratos ou concretos. Por fim, a última associação da metaclasse *AbstractStructureElement* é a *implementation*: *KDMEntity[0..*]*, cujo objetivo é especificar elementos computacionais de código (do pacote *Code*, ou seja, *Package*, *ClassUnit*, *InterfaceUnit*, etc.) que compõe o elemento estrutural.

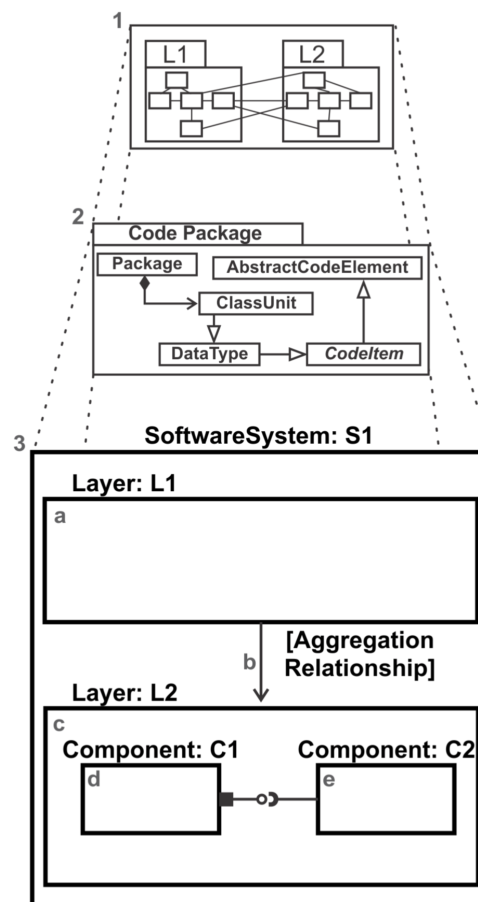

Figura 2.14 - Exemplo de uma arquitetura simples, representada em código (1), instância pacote Code (2) e visualização arquitetural (3). Adaptado de Durelli (2016)

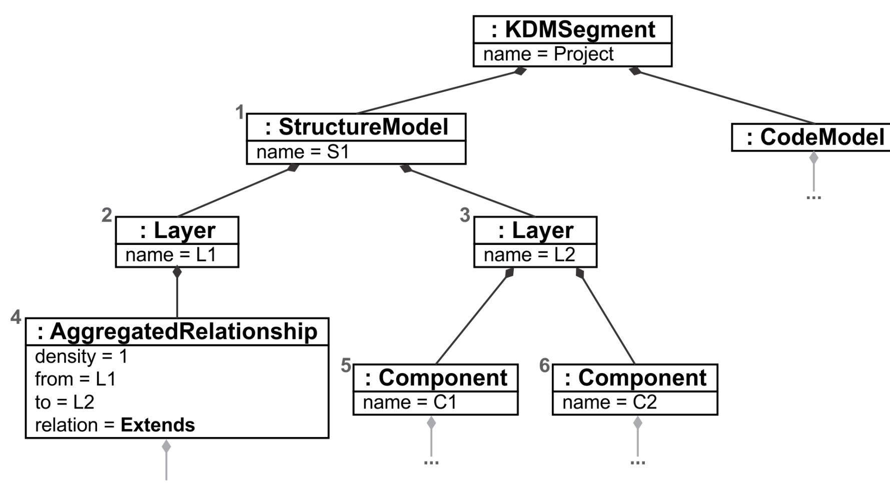

Figura 2.15 - Instância do KDM que representa a Figura 12. Adaptado de Durelli (2016)

Na Figura 2.14 está exemplificado uma possível arquitetura utilizando os elementos estruturais do KDM. A imagem foi dividida em três níveis, o nível 1 representa o sistema a nível de código-fonte. L1 e L2 representam dois pacotes, cada um contendo um conjunto de artefatos físicos (classes e interfaces) e seus relacionamentos. Nota-se que estes relacionamentos por sua vez ocorrem tanto entre objetos de um mesmo pacote, bem como entre artefatos de pacotes diferentes. O nível 2 apresenta o sistema no contexto do pacote *Code*, onde instâncias de *Package* correspondem as camadas, e os objetos das metaclasses do KDM representam os artefatos do sistema. O nível 3 ilustra o sistema na perspectiva do pacote Structure. Nesse ponto vislumbra-se a arquitetura do software em seu mais elevado nível de abstração.

No topo da hierarquia se encontra o elemento estrutural *SystemSoftware*, sendo este subdividido em duas camadas L1 e L2. A camada L2 por sua vez contém dois componentes, C1 e C2, onde C1 compartilha uma interface com C2. Além disso, observa-se que camada L1 possui um relacionamento com L2, onde a primeira pode acessar a última diretamente. Esse relacionamento é representado pela metaclasse *AggregatedRelationship*.

Na Figura 2.15 está exemplificado uma instância do KDM que corresponde ao código-fonte na Figura 2.14. Por questões de simplificação, algumas metaclasses (representadas por "...") não são exibidas na figura. As metaclasses *Layer* e *Component* representam as camadas do sistema e os componentes do sistema respectivamente. Observa-se que a camada L1 está associada a um agregado de relacionamentos (*AggregatedRelationship)*, esta associação representa o relacionamento que a camada possui com outro elemento estrutural do modelo, nesse contexto, a camada L2. O atributo *density* indica a quantidade de relações primitivas entre os elementos envolvidos. Os atributos *from* e *to* referenciam respectivamente os elementos

estruturais de origem e de destino, por fim, o atributo *relation* que indica qual a forma de acesso permitida entre as camadas, no exemplo da figura, a camada L1 possui autorização de herdar elementos de L2.

2.3 Detalhes de Representação UML

A UML (*Unified Modeling Language*) foi desenvolvida com o propósito de fornecer a arquitetos e engenheiros, ferramentas para análise, design e implementação de software, bem como oferecer suporte à modelagem de negócio. (OMG, 2015). Em 1997, foi adotada pela OMG como um padrão, a fim de unificar os métodos de modelagem de objetos e promover interoperabilidade entre ferramentas visuais de modelagem.

Segundo a OMG (2015), para que o sucesso no compartilhamento de modelos entre ferramentas distintas seja devidamente alcançado, um acordo semântico e sintático é fundamental. Com isso em mente, um conjunto de elementos pré-definidos que representem conceitos e componentes de software foi criado e tem sido amplamente utilizado nos modelos UML. Nesse projeto alguns desses conceitos serão responsáveis por denotar não-conformidades arquiteturais. Seus respectivos significados, serão discutidos a seguir. São eles: associação, dependência, generalização e realização.

Associação: Uma classe é composta por propriedades, essas por sua vez, representam características da classe. Fowler (2003) descreve propriedade como um conceito único que pode aparecer em duas formas diferentes: atributos ou associações. Atributos são características que não envolvem duas classes, estando comumente relacionado a uma variável de tipo primitivo. Associação por sua vez, envolvem duas classes, onde uma possui uma referência para a outra. O código-fonte em Java na Figura 2.16 ilustra uma associação entre duas classes.

```
1 package associacao;
2
3 public class ClasseA {
4     ClasseB associationA;
5 }
```

Figura 2.16 – Código-fonte de associação

Como pode-se notar, a *ClasseA* possui uma referência, chamada *associationA* (Linha 4), para a *ClasseB*, logo, *ClasseA* está associada a *ClasseB*. Na UML, associações são representadas por uma seta de linha contínua conectando as classes envolvidas (Figura 2.17).

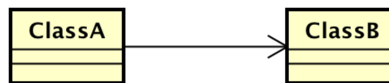

Figura 2.17 - Associação

Dependência: Booch, Rumbaugh e Jacobson (1998) definem dependência como um relacionamento semântico entre dois elementos, na qual a mudança em um (independente) pode afetar a semântica do outro elementos (dependente). As seguintes situações em uma classe podem indicar uma dependência:

- Invocação de uma operação (método) de outra classe
- Uso de outra uma classe como tipo de parâmetro em um método.
- Uma classe utiliza os dados (atributos) de outra classe
- Uso de outra classe como retorno em um método

O código-fonte em Java na Figura 2.18 ilustra uma dependência causada pelo uso de outra classe como retorno de um método. Nesse exemplo, método *methodA* (Linha 5). Na UML a representação de uma dependência é feita por uma seta de linha tracejada (Figura 2.19).

```
1  package dependencia;
2
3  public class ClasseA {
4
5      public ClasseB methodA() {
6          return null;
7      }
8  }
```

Figura 2.18 - Código-fonte de dependência

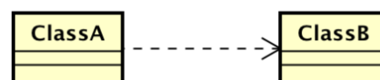

Figura 2.19 - Dependência

Generalização: Generalização é o processo de identificar características comuns entre conceitos (classes) diferentes e extraí-los para um conceito de propósito geral (superclasse) (LARMAN, 2001). O código-fonte em Java na Figura 2.20 exemplifica uma generalização, onde uma classe chamada *ClasseA* estende a outra denominada *ClasseB* (Linha 3). É importante mencionar que nessa linguagem, a generalização é conhecida também como herança. O relacionamento de generalização entre a classe e a superclasse é representada por uma seta triangular de linha contínua, como pode ser visto na Figura 2.21.

```
1 package generalizacao;
2
3 public class ClasseA extends ClasseB {
4
5 }
```

Figura 2.20 - Código-fonte de generalização

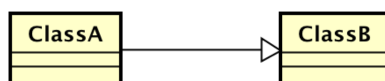

Figura 2.21 - Generalização

Realização: Diferentemente de uma classe, Fowler (2003) define *tipo* como uma abstração menos delimitada a implementação. Ao processo de implementação de um *tipo* tem-se o relacionamento de realização. Em um diagrama UML, a realização é representada por uma seta triangular de linha tracejada (Figura 2.23). Vale ressaltar que a relação entre classe de implementação e interface, esta última que por sua vez é uma aplicação prática similar a *tipo e* bastante comum em algumas linguagens de programação, também recebe o nome de realização. O código-fonte na Figura 2.22 exemplifica uma realização (Linha 3) em Java.

```
1 package realizacao;
2
3 public class ClasseA implements Interface1 {
4
5 }
```

Figura 2.22 - Código-fonte de realização

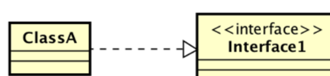

Figura 2.23 – Realização

2.4 A Ferramenta de Engenharia Reversa Modisco

O Modisco foi desenvolvido como uma ferramenta integrada à IDE Eclipse (*plugin*) e oferece um conjunto de componentes reusáveis, extensíveis e baseados em modelos cujo objetivo é facilitar a elaboração de soluções para engenharia reversa de sistemas legados (BRUNELIÈRE, 2011). No contexto desse projeto, o Modisco servirá como plataforma para os *discoveries* propostos. O Modisco é uma ferramenta fundamental para a conclusão desse trabalho, além disso, a contribuição desse componente com a ADM é significativa e seu uso tem se difundido progressivamente (BRUNELIÈRE, 2011), sendo considerada referência de

implementação pela *Architecture-Driven Modernization* (ADM) *Task Force* para os metamodelos *Knowledge Discovery Metamodel* (KDM), *Software Measurement Metamodel* (SMM) e *Abstract Syntax Tree Metamodel* (ASTM). Como parte de um projeto colaborativo dedicada à engenharia reversa, o time AtlanMod Team (INRIA & Ecole des Mines de Nantes) criou o Modisco em meados de 2006, desde então a ferramenta vem sendo aperfeiçoada. A empresa Mia-Software ingressou no projeto e em 2008 foi considerada oficialmente parte dele, contribuindo consideravelmente com seu desenvolvimento.

O suporte a uma determinada tecnologia é dado por meio de metamodelos, o modisco oferece nativamente suporte a linguagem java, sendo capaz de extrair informações (conhecimento) da estrutura global de qualquer software escrito nessa linguagem.

Na Figura 2.1 é exibida uma visão geral do funcionamento da ferramenta.

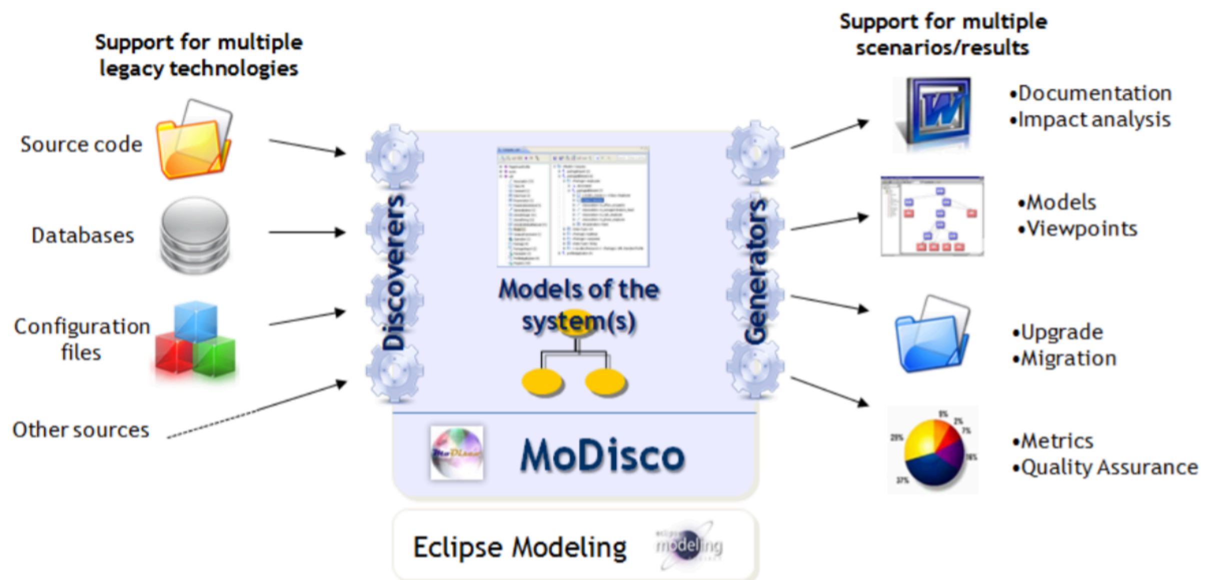

Figura 2.24 - Funcionamento geral do Modisco. (BRUNELIÈRE, 2011)

É importante mencionar que o conhecimento recuperado do sistema legado é representado na forma de modelo, logo, para se obter tal modelo, é necessário o uso de um mecanismo denominado *discoverer*. O Modisco possui um amplo conjunto de *discoverers*, todos são acessíveis por meio do *Discovery Manager*. Além disso, em consequência de sua extensibilidade, a ferramenta permite que novos *discoverers* sejam registrados e acoplados à sua interface. Como pode ser visto na Figura 2.24, o objetivo do *discoverer* é realizar a engenharia reversa e recuperar conhecimento das partes que compõem o software – código-fonte (*Source code*), bancos de dados (*Databases*), arquivos de configurações (*Configuration files*) – e representar esse conhecimento na forma de um modelo. Esse modelo será usado pelo engenheiro de software e, por meio da engenharia avante, será convertido novamente em um

software, dessa vez modernizado. Além dos *discoverers*, o Modisco possui também os chamados *Generators*. O *generator* funciona no sentido oposto ao de um *discoverer*, ou seja, seu objetivo é produzir dados - tais como métricas, documentação, análises de impacto – a partir do modelo. Esses dados terão o propósito de auxiliar o engenheiro de software no processo de modernização. Contudo, é importante mencionar que os *generators* não se limitam apenas à mineração de dados, eles podem também contribuir com a engenharia avante no processo de geração de artefatos do sistema modernizado, como por exemplo, código fonte.

Atualmente o Modisco possui um *discoverer* para produção de diagramas de classe UML, porém o mesmo não exibe informações visuais importantes. Um exemplo disso, é o fato do Modisco não ser capaz de identificar dependências em algumas situações, tais como:

- Quando a classe cliente utiliza o nome totalmente qualificado da classe fornecedora
- Quando a classe cliente e fornecedora se encontram no mesmo pacote.

Outro problema identificado é a ausência de um indicador do trecho de código que causou a dependência. Com isso, se uma dependência é causada por tipos diferentes de relacionamentos de código, como por exemplo uma chamada de método (*Call*), criação de objeto (*Create*) ou uso de parâmetro (*HasType*), o engenheiro de software não possui suporte visual algum para identifica-la. Essa questão foi tratada na abordagem desse projeto através de anotações com um pseudocódigo da dependência. Além disso, o *Discoverer* do Modisco possui um propósito mais geral não dando enfoque às não-conformidades arquiteturais logo, detalhes arquiteturais do pacote *Structure*, tais como camadas, componentes, sistemas e subsistemas não são ilustrados.

2.5 Linguagem de Transformação de Modelos ATL

O conceito de transformações de modelos se difundiu na Engenharia Dirigida a Modelos (MDE), com isso, o número de linguagens específicas de domínio voltadas a essa tarefa cresceu substancialmente nas comunidades de pesquisa e na indústria de software (JOUAULT e KURTEV, 2005). Como alternativa, em 2008 o grupo de pesquisa ATLAS group LINA & INRIA Nantes propôs a *Atlas Transformation Language* (ATL).

Embora predominantemente declarativa, alguns componentes da linguagem são escritos na forma imperativa, logo, a linguagem é considerada de paradigma híbrido. Em sua parcela

declarativa, a ATL é composta por um conjunto de regras denominada *rules* e funciona baseada na noção de "regra correspondida" (JOUAULT e KURTEV, 2005). Cada *rule* é composta por um padrão a ser correspondido no modelo de origem e um padrão à ser criado quando uma correspondência é encontrada. Na parcela imperativa a ATL disponibiliza dois componentes: (i) *called rule*, que funciona como um procedimento a ser executado podendo ser explicitamente invocado e (ii) *action block*, que por sua vez é um conjunto de instruções imperativas. Na Figura 2.25 está representado um exemplo de transformação de um modelo chamado *Model Family* para outro modelo denominado *Model Persons*.

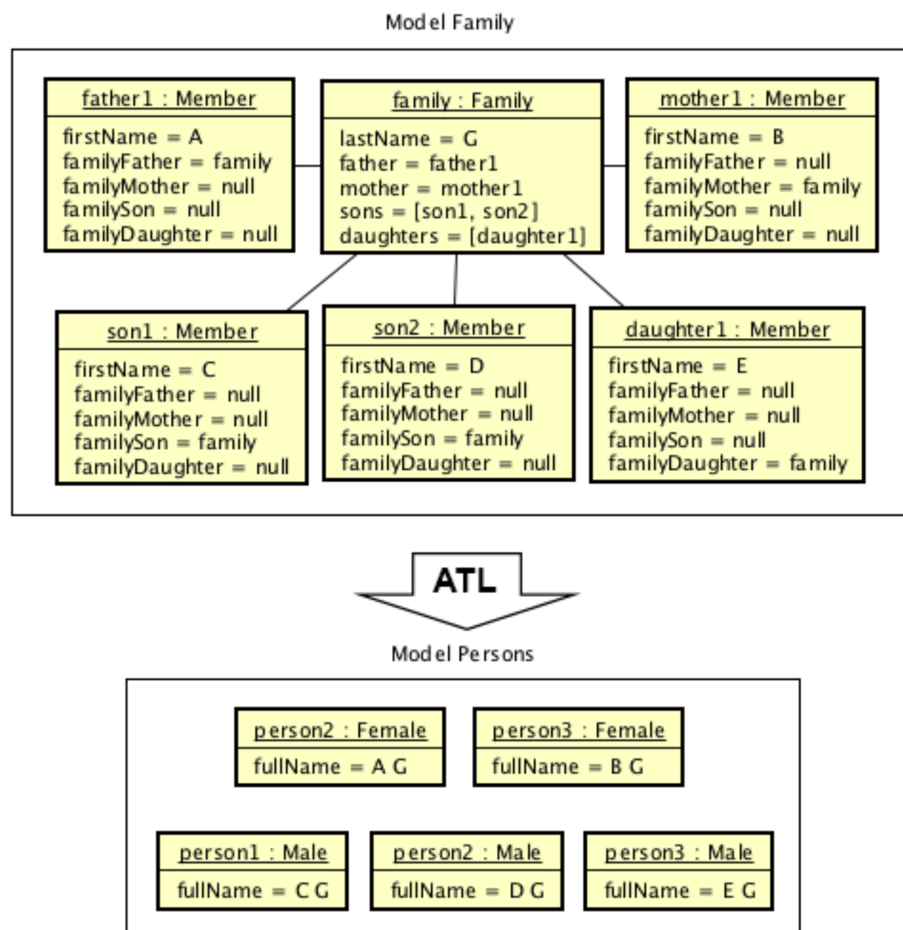

Figura 2.25 - Exemplo de Transformação de modelos

O modelo *Model Family* é composto por seis instâncias, sendo que a instância *family* pertence à classe *Family* e as instâncias *father1*, *mother1*, *daughter1*, *son1* e *son2* pertencem a classe *Member*. A instância *family* possui cinco atributos: *lastName*, *father*, *mother*, *sons* e *daughters*. Os atributos *father* e *mother* fazem referência às instâncias *mother1* e *father1* respectivamente. O atributo *sons* possui uma lista constituída de referências às instâncias *son1* e *son2*. Por fim, o atributo *daughters* é composto de uma lista com apenas uma referência à

instância *daughter1*. Cada instância da classe Member possui cinco atributos, *firstName*, *familyFather*, *familyMother*, *familySon*, *familyDaughter*. É importante ressaltar que dentre os quatro últimos parâmetros da classe *Member*, apenas um deles possui uma referência a instância *family*.

O modelo *Model Persons* apresenta um conjunto de cinco instâncias: *person1* e *person2*, pertencentes a classe *Female*, *person3*, *person4*, *person5*, por sua vez, associados à classe *Male*. Nesse modelo cada instância possui um atributo chamado *fullName*. Esse atributo é composto pelo valor do atributo *firstName* de cada uma das instâncias da classe *Member* do modelo *Model Family*, concatenado ao atributo *lastName* da instância *family*, também do modelo *Model Family*. No código-fonte da Figura 2.26 está escrito a transformação *ATL* apresentada na Figura 2.25.

```
1  -- @path Family=/Family2Persons/Family.ecore
2  -- @path Persons=/Family2Persons/Persons.ecore
3
4  module Family2Persons;
5  create OUT : Persons from IN : Family;
6
7⊕ helper context Family!Member def: familyName : String =
21
22⊕ helper context Family!Member def: isFemale() : Boolean =
32
33⊖ rule Member2Male {
34      from
35          s : Family!Member (not s.isFemale())
36      to
37          t : Persons!Male (
38              fullName <- s.firstName + ' ' + s.familyName
39          )
40 }
41
42⊖ rule Member2Female {
43      from
44          s : Family!Member (s.isFemale())
45      to
46          t : Persons!Female (
47              fullName <- s.firstName + ' ' + s.familyName
48          )
49 }
```

Figura 2.26 - Código ATL da transformação da Figura 19. Adaptado de Allilaire e Jouault (2007)

Os arquivos *Family.ecore* (Linha 1) e *Persons.ecore* (Linha 2) representam os modelos que estão envolvidos na transformação, *Families2Persons* (Linha 4) é o nome da transformação e *Persons* e *Family* (Linha 5) são os modelos de saída e entrada respectivamente. Na Linha 7 e 22 duas funções auxiliares são criadas, (i) *familyName* (Linha 7) que retorna o atributo *lastName* da instância de *Family* associada à instância de *Member* e (ii) *isFemale* (Linha 22) que indica se a instância *Member*, no modelo de origem, deverá ser transformada em uma

instância de *Female*, no modelo de destino. Por fim são definidas duas regras (*rules*), *Member2Male* (Linha 33) e *Member2Female* (Linha 42).

A palavra reservada *from* (Linha 34 e 43), em ambas as regras, define um padrão a ser correspondido e a palavra *to* (Linha 36 e 45) um padrão de transformação à ser executado. No padrão a ser correspondido da regra *Member2Male* é definido que, para todas as instâncias de *Member* do modelo *Family* cuja função *isFemale* retornar falso, o seu padrão de transformação será executado. No caso da regra *Member2Female*, o mesmo ocorrerá caso a função retorne verdadeiro.

O padrão de transformação para a regra *Member2Male* define que uma instância *Male* será criada no modelo *Persons*. Nessa instância, o atributo *fullName,* será resultado da concatenação do atributo *firstName,* da instância de *Member* correspondida, com o resultado da função *familyName*, também da instância de *Member* correspondida. Por fim, de forma análoga, o mesmo ocorre para a regra *Member2Female*. Contudo neste caso, no lugar de uma instância de *Male*, será criada uma instância de *Female* para o modelo *Persons*.

Considerações Finais

Neste capítulo foram apresentados conceitos essenciais para o entendimento do projeto. Foram apresentados os domínios e as três perspectivas da ADM (Técnica, de Aplicação e de Negócio), que contextualizam o projeto. Em seguida é descrito o metamodelo KDM e os pacotes base (*Code*, *Action* e *Structure*) da instância que servirá de fonte para as transformações ATL. Foi retratado conceitos de relacionamentos UML (Associação, Dependência, Generalização e Realização) que serão utilizados na abordagem para representação das não-conformidades arquiteturais. Por fim foram apresentadas a ferramenta de engenharia reversa Modisco e a linguagem de transformação de modelos ATL.

Capítulo 3

TRABALHOS RELACIONADOS

Considerações Iniciais

Os trabalhos relacionados foram divididos em duas categorias: Visualização de detalhes arquiteturais e recuperação de modelos UML. A primeira categoria (Seção 3.2) apresentam trabalhos que, de alguma forma, envolvem a exibição de detalhes da arquitetura de um software. É importante mencionar que esses detalhes não estão necessariamente relacionados a não-conformidades arquiteturais, o propósito dessa seção é ilustrar possíveis formas de se apresentar informações arquiteturais. A segunda categoria (Seção 3.3), é composta por trabalhos cujo objetivo é recuperar diagramas UML de um determinado software. Vale ressaltar que, nesse caso também, os diagramas não necessariamente possuem o objetivo de ilustrar não-conformidades arquiteturais, mas sim, demonstrar diferentes abordagens para produção de modelos UML.

3.1 Visualização de Detalhes Arquiteturais

Sangal, Jordan, *et al*. (2005) sugerem o uso de modelos de dependência para manter a integridade arquitetural do software. Embora o foco do trabalho não seja especificamente visualização de não-conformidades arquiteturais, os autores propõem um método para ilustrar dependências entre componentes da arquitetura de um software. A abordagem é baseada em

uma matriz denominada Matriz de Estrutura de Dependências (DSM - *Dependency-Structure Matrix*).

Figura 3.1 - Matriz de Estrutura de Dependências (DSM - *Dependency-Structure Matrix*)

Na Figura 3.1 está ilustrado o exemplo um sistema cuja arquitetura se divide em cinco camadas *application* (1), *model* (2), *domain* (3), *framework* (4) e *util* (5). As camadas estão dispostas em uma matriz 5 por 5, onde cada linha e coluna representam as camadas da arquitetura do software. A intersecção entre a linha e a coluna representa a quantidade de referências explícitas entre uma camada e outra, ou seja, o número de dependências entre elas. Nesse caso, verifica-se uma dependência entre as camadas 1 e 2 com 37 referências, 2 e 3 com 29 referências, 3 e 4 com 42 referências, por fim 4 e 5 com 13 referências. Embora o método de exibição seja eficiente para a proposta do trabalho dos autores, de restringir o surgimento de não-conformidades arquiteturais e manter a arquitetura planejada, quando o objetivo é visualizar quais são as não-conformidades do sistema, a abordagem não é eficiente. A razão disso se deve ao fato de que por não ilustrar especificamente a ação em código que causou a violação, o engenheiro de software se viria obrigado a procurar por essa ação, consumindo mais tempo.

Kamal e Avgeriou (2008) propõem uma abordagem de visualização de detalhes arquiteturais baseada na definição de primitivas arquiteturais por meio da UML. Para realizar tal tarefa, os autores fazem uso de mecanismos de extensão da UML para representar os elementos arquiteturais.

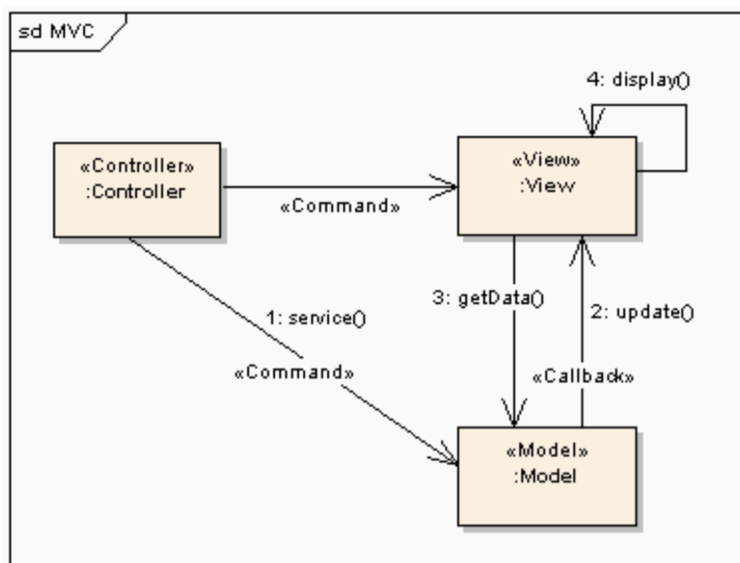

Figura 3.2 - Modelo MVC usando primitivas e elementos de design. (KAMAL e AVGERIOU, 2008)

Segundo Mehta e Medvidovic (2003) primitivas arquiteturais são conceitos subjacentes compartilhados por diferentes estilos de arquitetura. No exemplo ilustrado na Figura 3.2 vislumbra-se uma arquitetura MVC, onde cada camada é representada por um estereótipo UML, e duas primitivas: *Command* (<<Command>>) e *Callback* (<< Callback >>). A primitiva *Command* representa tipicamente uma invocação de uma função ou procedimento de um objeto alvo (KAMAL e AVGERIOU, 2008). Essa primitiva pode ser encontrada, por exemplo, no padrão CQRS (Command Query Responsibility Segregation). A primitiva Callback representa a interação entre dois objetos onde B passa uma referência de sua própria instância para A, a fim de que em tempo de execução A, invoque métodos de B. Esse princípio é frequentemente visto em arquiteturas que utilizam o conceito de Inversão de Controle (IoC - Inversion of Control).

3.2 Recuperação de Modelos UML

Martinez, Pereira e Favre (2014) propõem a recuperação de diagramas UML de sequência baseada nos conceitos de engenharia reversa propostos pela ADM. O método utiliza como entrada uma instância KDM recuperada pelo *plugin Modisco*. Por fim, um modelo, na forma de diagrama de sequência, é produzido por meio de um conjunto de regras de transformação ATL.

A abordagem busca por métodos públicos relevantes, em uma dada classe, e os transforma em diagramas distintos. A referência para objeto dono do método principal – método que deu origem ao diagrama - é selecionado como o primeiro *lifeline* e, por sua vez, as referências aos objetos secundários – objetos usados pelo método principal - dão origem aos *lifelines* seguintes. Na Figura 3.3 está ilustrado o exemplo de um diagrama recuperado pela abordagem.

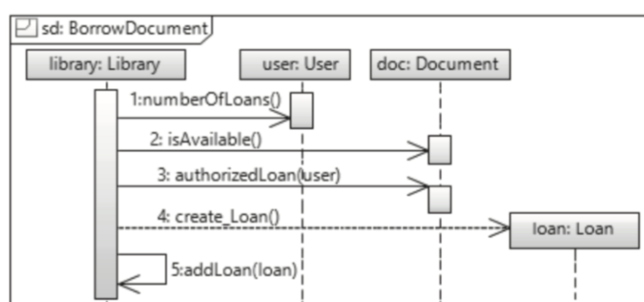

Figura 3.3 - Diagrama de sequência

Nesse exemplo é criado um diagrama para o método público *borrowDocument* de uma classe definida como *Library*. A referência proprietária *library: Library* é selecionada como *lifeline* inicial, as referências *user: User, doc: Document* e *loan: Loan* são escolhidas para os *lifelines* restantes. Por fim, do objeto referenciado por *library*, partem 5 mensagens (métodos) numberOfLoans, *isAvailable*, *authorizedLoan*, *create_Loan* e *add_Loan*.

Sutton e Maletic (2007) propõem uma abordagem mais específica para a recuperação de diagramas de classe a partir códigos em C++. Ao invés de criar um analisador (*parser*) especificamente para a linguagem C++, os autores optaram pelo desenvolvimento de um transformador que produz uma representação do código na forma de um modelo intermediário. O modelo intermediário em questão é uma instância do modelo *srcML* (*Source Code Markup Language*). Em seguida essa instância passa por uma ferramenta, criada com o propósito de apoiar a abordagem, denominada *pilfer*. A ferramenta é responsável em montar um grafo de relacionamento semântico e por fim, construir e serializar os elementos UML, por meio do OMF (*Open Modeling Framework*), esse que por sua vez, é uma implementação em C++ do MOF (*Meta-Object Facility*). O processo descrito está ilustrado na Figura 3.4.

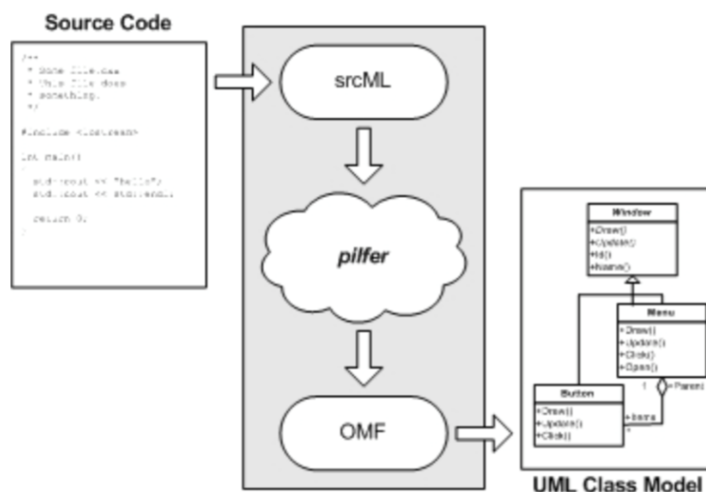

Figura 3.4 - Arquitetura do pilfer (SUTTON e MALETIC, 2007)

Considerações Finais

Nesse capítulo foram descritos trabalhos relacionados a visualização de detalhes arquiteturais. Foi apresentado uma abordagem que utiliza Matriz de Estrutura de Dependências para ilustrar as violações, e mais dois trabalhos que, assim como o projeto desenvolvido nesse mestrado, faz uso da UML para este propósito. É importante mencionar que embora esses trabalhos ilustrem detalhes arquiteturais de um sistema, apenas a primeira abordagem está relacionada a visualização de não conformidades arquiteturais.

Capítulo 4

Checagem de Conformidade Arquitetural com ArchKDM

Considerações Iniciais

A ArchKDM é a base da abordagem apresentada nesse trabalho. Com isso em mente, esse capítulo tem objetivo de explicar o processo de Checagem de Conformidade Arquitetural da ArchKDM. Na Seção (4.1) foi explicado o conceito de Checagem de Conformidade Arquitetural (CCA). Por fim na Seção (4.2) foi detalhada como é realizada a Checagem de Conformidade Arquitetural pela ArchKDM.

4.1 Checagem de Conformidade Arquitetural

Com o advento do conceito de erosão arquitetural e desvios na arquiteturais tornou-se evidente a necessidade um processo de Checagem de Conformidade Arquitetural (CCA), cujo objetivo é identificar elementos no software que violaram a arquitetura planejada inicialmente. O processo de CCA é uma atividade de suma importância e pode ser considerada como uma das principais etapas no controle de qualidade de software (KNODEL e POPESCU, 2007).

Com o processo de CCA é possível identificar três componentes importantes para o processo de Reconciliação Arquitetural: (i) relacionamentos planejados e implementados,

também conhecidas como convergências. As convergências representam as relações que estão de acordo com a arquitetura planejada. (ii) Relacionamentos existentes na arquitetura atual que não são permitidos, denominados de divergências ou desvios arquiteturais. Por fim os (iii) relacionamentos que não foram especificadas na arquitetura planejada, contudo não violam a arquitetura do software. Esses relacionamentos são também conhecidos como ausências.

4.2 ArchKDM

Desenvolvida no laboratório de pesquisa AdvanSE da Universidade Federal de São Carlos, a ArchKDM foi criada com o objetivo de fornecer apoio ferramental à ADM no processo de checagem de conformidade arquitetural (CCA). Seu funcionamento é baseado em três fases: I - Especificação da arquitetura planejada, II - Mapeamento e extração da arquitetura atual e III - Checagem de conformidade arquitetural (CHAGAS, 2016). Os artefatos gerados pelas fases I e II serão utilizados como entrada na fase III, essa por sua vez produzirá um KDM contendo as não-conformidades arquiteturais, bem como o CodeModel e ActionModel do sistema legado. A abordagem pode ser observada na Figura 4.1.

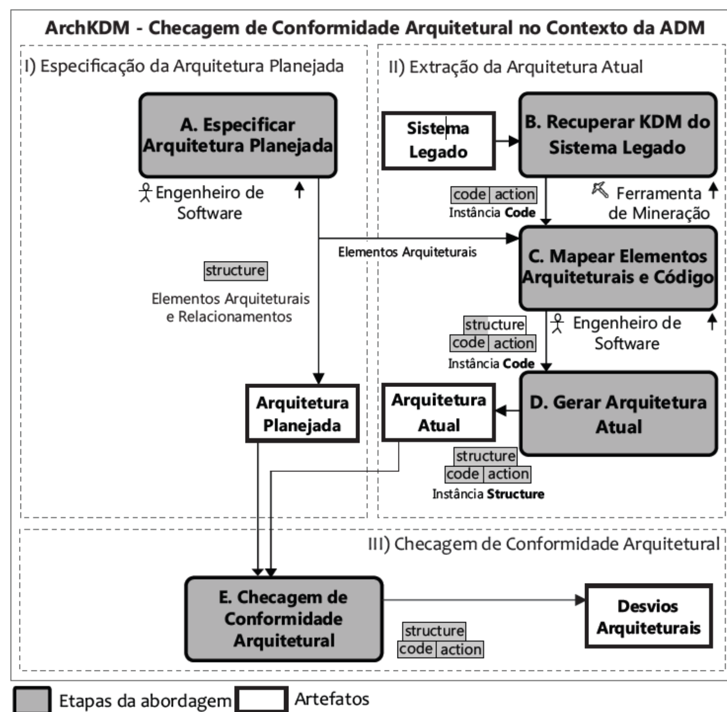

Figura 4.1 - Checagem de conformidade arquitetural da ArchKDM

A fase de Especificação da Arquitetura Planejada (fase I) possui apenas uma etapa: (A) Especificar a Arquitetura Planejada. Nessa etapa o engenheiro irá definir os elementos arquiteturais, tais como camadas (*layers)*, componentes (*component)* e subsistemas (*subsystems)* e as restrições arquiteturais que compõem o relacionamento entre esses elementos, tais como "pode apenas" (*only-can)*, "não pode" (*cannot)*, "deve" (*must)*, "acessa" (*access)*, "declara" (*declare)* e cria (*create)*.

A fase II (Mapeamento e extração da arquitetura) é subdividida em três etapas: (B) Recuperar KDM do sistema legado, (C) Mapear Elementos Arquiteturais e Código e (D) Gerar Arquitetura Atual. Na etapa (B), uma ferramenta de mineração recupera o KDM do sistema legado de forma automatizada. Em seguida inicia-se a etapa (C) que, por sua vez, é manual e deve ser realizada pelo engenheiro de software. Utilizando-se da arquitetura planejada, nessa etapa são associados os elementos de código e de ação, extraídos na etapa anterior, aos elementos arquiteturais. Por fim inicia-se a etapa (D), na qual um algoritmo analisará os elementos de código com o propósito de criar os relacionamentos entre os elementos arquiteturais. Esses relacionamentos arquiteturais são criados por meio de instâncias da classe *AggregatedRelationship*. Como resultado da segunda fase, um arquivo KDM com a arquitetural atual será criado.

Na fase checagem de conformidade arquitetural (III) a ferramenta faz a comparação dos artefatos resultantes da fase I e II verificando quais relacionamentos estão na arquitetura atual e não foram previstos na arquitetura planejada. Esses relacionamentos são adicionados a uma lista como não-conformidades (violações) arquiteturais. Cada não-conformidade é representada por um relacionamento esses que, por sua vez, podem ser tanto de código (CodeRelationships), como também de ação (*ActionRelationship*). A lista abaixo sintetiza as metaclasses especialistas dos relacionamentos de código (a-e) e dos relacionamentos de ação (f-h) utilizados pela ArchKDM para representar não-conformidades arquiteturais:

a) *Implements* (Realização): Violações nessa categoria representam implementação de interfaces. Estará classificada nesta categoria toda não-conformidade causada por uma classe que implementa a interface, esta por sua vez localizada em uma camada não acessível pela camada implementadora.
b) *Extends* (Generalização): Violações nessa categoria representam herança de classes. Estará classificada nesta categoria toda não-conformidade causada por uma classe que herda outra classe, definida em uma camada, não acessível pela camada herdeira.
c) *Imports* (Importação): Essa categoria representa violações causadas por importação. Se configuram nesta categoria toda não-conformidade causada por uma classe, interface ou

tipo de dados definidos em uma camada não acessível, e usados como tipo pela camada importadora.

d) *HasType* (Possui definição do tipo): Essa violação ocorre quando uma variável, atributo, parâmetro ou retorno de função está declarada com um tipo de dado não primitivo, cuja definição do mesmo se encontra em uma camada não acessível pela camada declarante.

e) *HasValue* (Inicialização): Essa categoria representa violações causadas na inicialização de uma variável. Ocorre quando o tipo de dados do item que está inicializando a variável, foi definido em uma camada não acessível pela camada declarante dessa variável.

f) *Calls* (Chamada): As violações classificadas com essa categoria indicam não-conformidades causadas pela invocação de um método, ou seja, a violação representa a invocação de um método, cuja definição está localizada em uma camada não acessível pela camada invocadora.

g) *Creates* (Instanciação): Violações causadas por instanciação. Essa violação ocorre quando classes definidas em uma camada destino não acessível são instanciadas na camada origem.

h) *UsesType* (Conversão de tipo): Quando um elemento de dados está sendo convertido (*typecasting*) para um outro tipo, cuja definição está em uma camada não acessível à camada do elemento convertido, a violação estará nesta categoria.

É importante mencionar que as classes especialistas dos relacionamentos, de ação e código, auxiliam na identificação do componente UML que melhor descreve a violação em um diagrama de classes. Por fim, com a conclusão da fase III, a ferramenta produzirá um arquivo KDM (XMI) contendo todas as não-conformidades detectadas na forma de relacionamentos de código e ação.

4.2.1 Não Conformidades Arquiteturais

Como forma de exemplificar não-conformidades arquiteturais, considere um software cujo padrão arquitetural é MVC (*Model-View-Controller*). Assume-se que a camada View só pode acessar a camada *Controller*, e essa por sua vez somente pode acessar a camada *Model*. Nesse exemplo as relações de acesso são bidirecionais, ou seja, se A pode acessar B, então B pode acessar A. O exemplo descrito está ilustrado na Figura 4.2. As setas representam direito de acesso, isto é, podem ser lidas como “pode acessar”

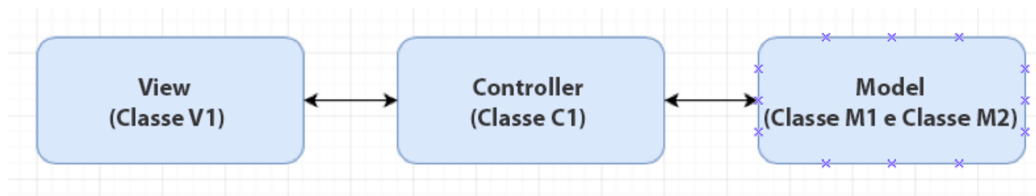

Figura 4.2 - Cenário MVC hipotético

Suponha que os relacionamentos entre os elementos de ação (*ActionElement*) e código das camadas estejam da seguinte forma:

- Classe V1 está na camada *View* (*Layer View*).
- Classe C1 está na camada *Controller* (*Layer Controller*).
- Classe M1 e M2 estão na camada *Model* (*Layer Model*).
- Classe V1 importa (1.f) classe M1.
- Classe V1 executa quatro métodos (1.a-d) de M1.
- Classe V1 importa (2.f) classe C1.
- Classe V1 executa dois métodos (2.a-b) da classe C1.
- Classe C1 importa classes M1 e M2 (3.e-f)
- Classe C1 executa dois métodos de M1 (3.a-b) e dois métodos de M2 (3.c-d).

Como já mencionado anteriormente, a saída da ArchKDM é um KDM logo, o formato do arquivo produzido é XMI. O modelo KDM que representa os relacionamentos citados acima está ilustrado na Figura 4.3.

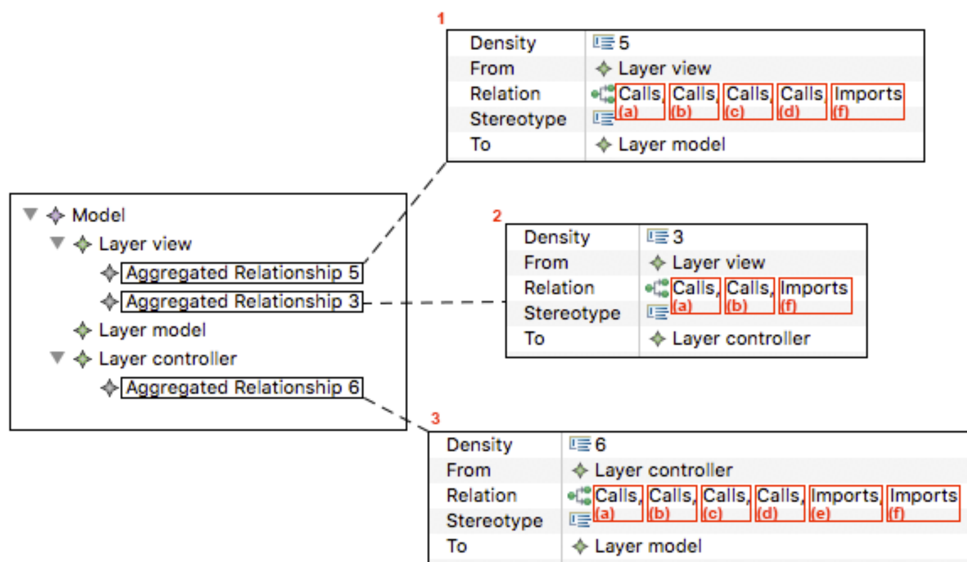

Figura 4.3 – Modelo KDM ilustrado na perspectiva da IDE Eclipse

Nota-se que existe um conjunto de relacionamentos (*Aggregated Relationship 5*) entre as camadas *View* e *Model* (*From: Layer view, To: Layer model)*. Contudo, foi definido na Figura 4.2 que os relacionamentos entre essas duas camadas não é permitido portanto, conclui-se que essa coleção se trata de um conjunto de não-conformidades, nesse caso, composta de quatro *Calls* e um *Imports*. No arquivo de saída gerado pela ArchKDM, além do modelo representado na Figura 4.3, se encontra também um modelo espelho (Figura 4.4), esse por sua vez armazena apenas os conjuntos de não-conformidades arquiteturais.

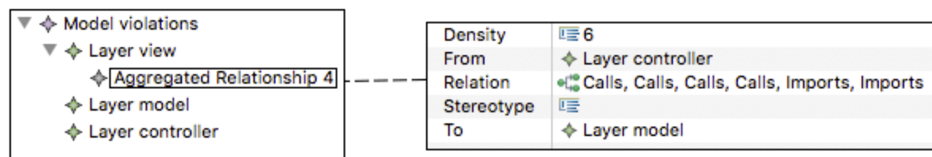

Figura 4.4 - Modelo KDM (apenas não-conformidades) ilustrado na perspectiva da IDE Eclipse

Considerações Finais

Nesse capítulo foi explicado o conceito de CCA e os três componentes os quais este processo permite identificar. Em seguida foi apresentada a ferramenta ArchKDM e descrita as etapas de uma CCA executada pela mesma. Por fim foi descrito o conceito de Não Conformidades arquiteturais, bem como o formato no qual a ArchKDM apresenta as não conformidades identificadas.

Capítulo 5

A ABORDAGEM DRIV-UML

Considerações Iniciais

Este capítulo tem o objetivo de apresentar a abordagem que foi desenvolvida neste projeto de mestrado. A Seção 5.1 dá uma visão geral da abordagem DiV-UML. A Seção 5.2 explica a estratégia utilizada no projeto para ilustrar desvios arquiteturais por meio da UML. A Seção 5.3 ilustra um exemplo do uso da ferramenta que apoia a abordagem, bem como detalha a interação dessa ferramenta com a ArchKDM. Por fim é dado detalhes específicos de implementação da ferramenta na Seção 5.4.

5.1 Abordagem DriV-UML

Como já comentado, o objetivo deste projeto é auxiliar engenheiros de software a analisarem não-conformidades arquiteturais por meio do uso de diagramas UML. Para que este objetivo fosse atingido foi desenvolvida a ferramenta DriV-UML que, por meio de diagramas de classes e de pacotes, ilustra as não-conformidades arquiteturais encontradas pela ferramenta de CCA ArchKDM.

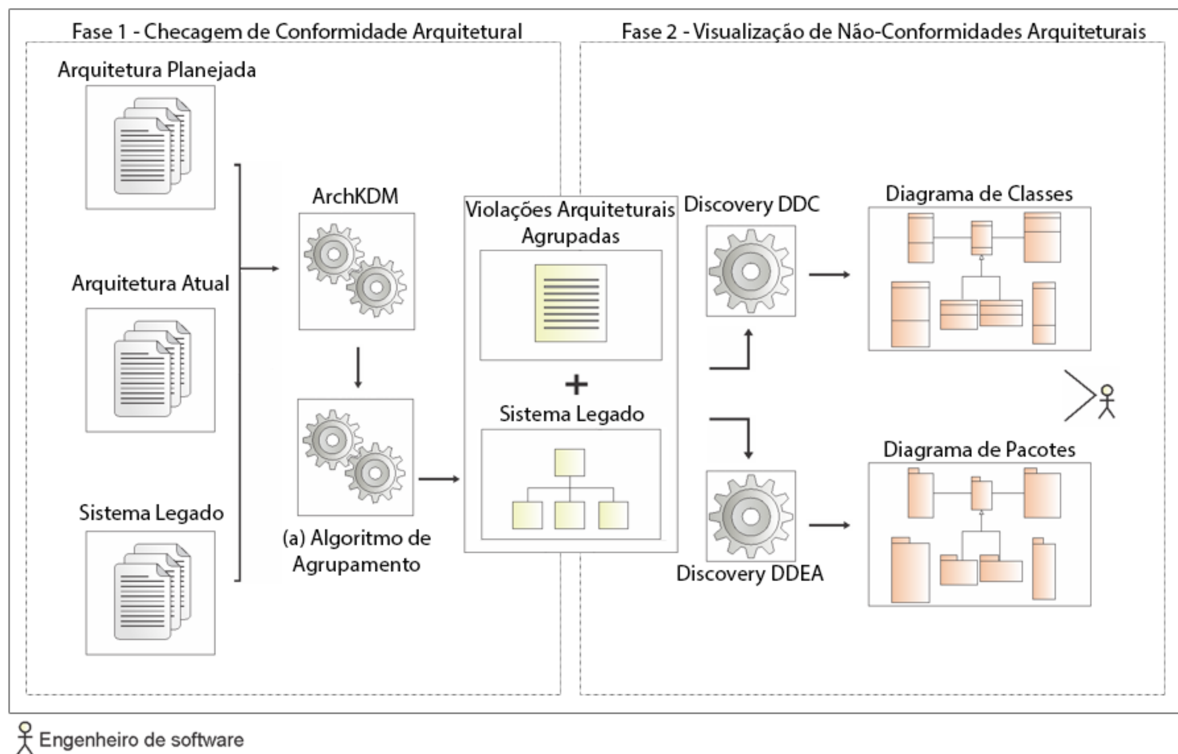

Figura 5.1 - Processo de produção dos diagramas a partir do ArchKDM

Como pode-se observar na Figura 5.1, o processo de produção dos diagramas é dividido em duas fases. A "Fase 1 - Checagem de Conformidade Arquitetural" tem como objetivo extrair do "Sistema legado" um KDM, junto com suas não-conformidades arquiteturais. A ferramenta recebe como entrada a "Arquitetura planejada", a "Arquitetura atual" e o "Sistema legado" e como saída produz um KDM contendo as informações necessárias, para o início da segunda etapa. Na "Fase 2 – Visualização de Não-Conformidades Arquiteturais" o KDM extraído na etapa anterior será a entrada para os *discoverers* DDC (Discovery de Desvios em Classes) e DDEA (Discovery de Desvios entre Elementos Arquiteturais*)*. Os discoveries são invocados pelo motor do ArchKDM que, por meio do lançador de transformações ATL do Modisco, executa as transformações. Juntamente com o algoritmo de agrupamento localizado na primeira fase, este trabalho de mestrado está localizado quase integralmente na segunda fase do processo. A primeira fase foi desenvolvida em um trabalho anterior do grupo (LANDI, 2018).

5.2 Estratégia Adotada para Representar Desvios Arquiteturais em UML

A estratégia adotada neste trabalho para exibir desvios arquiteturais na UML concentra-se apenas em diagramas de classe e de pacotes. O principal ponto da estratégia é o seguinte: *Cada relacionamento gráfico visível no diagrama representa pelo menos um desvio arquitetural*. Assim, se no diagrama de classe há uma dependência entre duas classes, isso significa que um desvio do tipo dependência existe entre elas. O mesmo ocorre para os de herança, realização e associação.

É importante mencionar que nesse projeto a UML é abordada com um propósito diferente do usual. A UML é usada com o intuito de exibir apenas desvios arquiteturais ao invés de toda a estrutura do sistema. A Figura 5.2 ilustra um exemplo de como a UML é usada neste projeto.

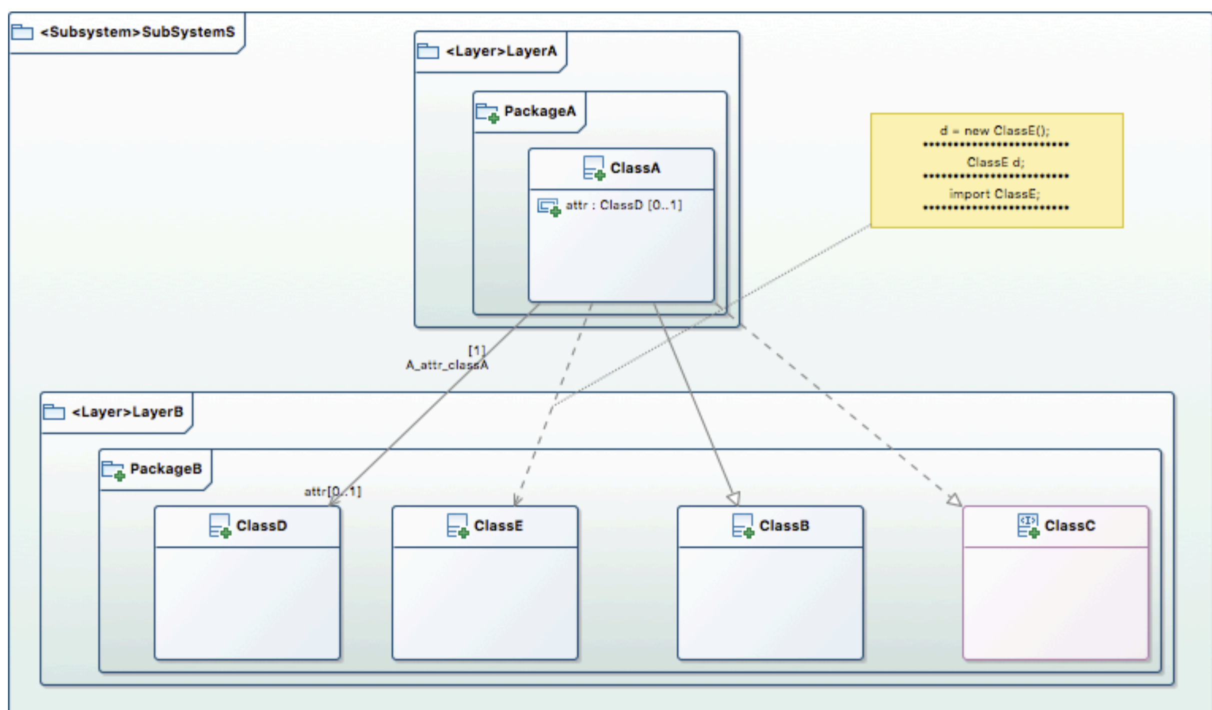

Figura 5.2 – Exibição de Não-conformidades arquiteturais em UML

Na Figura 5.2 está ilustrada duas camadas, LayerA e LayerB. A primeira é composta por um pacote, PackageA contento uma única classe, chamada de ClassA. A segunda camada é formada por um pacote, denominado PackageB que contém quatro classes: ClassB, ClassC, ClassD e ClassE. Nota-se quatro relacionamentos UML entre a classe da camada LayerA com as classes da LayerB. Esses relacionamentos representam não-conformidades arquiteturais, ou

seja, são relacionamentos que não deveriam existir na arquitetura do sistema. É importante mencionar que a semântica dos relacionamentos é preservada ou seja, se uma não-conformidade é representada por um relacionamento de generalização, isso quer dizer que a mesma foi causada por uma herança entre os elementos envolvidos. O mesmo princípio se aplica para os outros relacionamentos.

Como pode ser observado na Figura 5.2 alguns pacotes encontram-se rotulados com o rótulo <Layer>. Isso é possível porque a instância do KDM que serve de entrada para o algoritmo de agrupamento, e que é a saída do ArchKDM, possui elementos arquiteturais do sistema representados pelo pacote Structure. Como pode ser observado na Figura 2.13 (Página 32), o pacote Structure possui outros elementos arquiteturais como Componentes e Subsistemas. Assim como Layers, se o sistema em análise também possuísse outros elementos arquiteturais, os mesmos também seriam representados na forma de pacotes rotulados. Vale mencionar que elementos arquiteturais aninhados também são considerados. Ou seja, se uma camada possui outra camada aninhada, os pacotes que as representam são aninhados no diagrama UML.

A utilização desses rótulos permite representar de forma evidente as abstrações arquiteturais existentes no sistema. Isso faz com que o entendimento dos desvios arquiteturais seja mais facilmente compreendido pelo engenheiro de software.

É importante ressaltar que o pacote implementado por um elemento arquitetural é ilustrado no diagrama como um pacote interno. Por exemplo, na Figura 5.2, o elemento `PacoteA` está envolvido pelo pacote rotulado `LayerA`. Isso quer dizer que o `PacoteA` implementa `LayerA`. Se no mapeamento da arquitetura for definido que múltiplos pacotes e classes implementam uma mesma camada, todos esses pacotes e classes serão ilustrados no diagrama UML como elementos internos (filhos) do pacote que representa a respectiva camada.

Por último, também é importante ressaltar que se optou por anexar uma nota nos relacionamentos de dependência que representam desvios, essas notas contém um pseudocódigo Java equivalente ao trecho de código que causou a violação.

5.3 Exemplo de Uso da Abordagem

A imagem a seguir ilustra a estrutura de classes de um sistema hipotético que faz uso do padrão MVC.

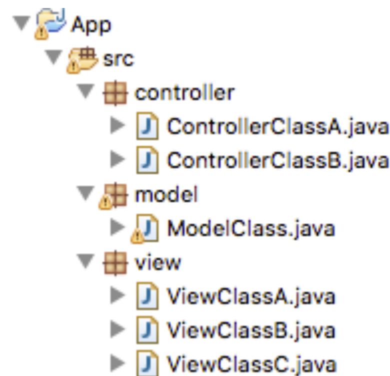

Figura 5.3 - Sistema hipotético

O sistema hipotético da Figura 5.3 é composto por cinco classes, sendo duas classes controladoras (*controllers*), uma de modelo (*model*) e três visão (*views*). As *views* são classes vazias criadas apenas com propósitos ilustrativos. A Figura 5.4, Figura 5.5 e Figura 5.6 exibem o código das classes ControllerClassA, ControllerClassB e ModelClass, respectivamente.

```
1  package model;
2
3  public class ModelClass extends view.ViewClassA {
4      view.ViewClassC attr;
5
6      void main() {
7          //Aggregated 2
8          view.ViewClassB d;
9          d = new view.ViewClassB();
10         System.out.println(d.toString());
11     }
12 }
```

Figura 5.4 – ModelClass

```
1  package controller;
2
3  import view.ViewClassB;
4
5  public class ControllerClassA {
6
7      public static void main(String[] args) {
8          new ViewClassB() {
9
10         };
11     }
12 }
```

Figura 5.5 – ControllerClassA

```
1 package controller;
2
3 public class ControllerClassB extends view.ViewClassA{
4
5 }
```

Figura 5.6 - ControllerClassB

Nota-se a existência de três relacionamentos na classe ModelClass (Figura 5.4): i) herança com a classe ViewClassA na linha 9, ii) associação com a classe ViewClassC na linha 10 e iii) dependência com a classe ViewClassB, dentro do método `main()` nas linhas 14 e 15. Na classe ControllerClassA (Figura 5.5) vislumbra-se uma dependência com ViewClassB na linha 8 e por fim, na classe ControllerClassB (Figura 5.6), uma herança com ViewClassA na linha 5.

A Figura 5.7 ilustra o diagrama de classes completo do sistema extraído pelo *discovery* UML nativo do Modisco. É importante ressaltar que uma dependência entre ModelClass e ViewClassB não foi identificada pelo *discovery*. Isso ocorre pois na sua atual implementação o *discovery* do Modisco não é capaz de identificar dependências quando, a referência à classe, é realizada por meio de seu nome totalmente qualificado.

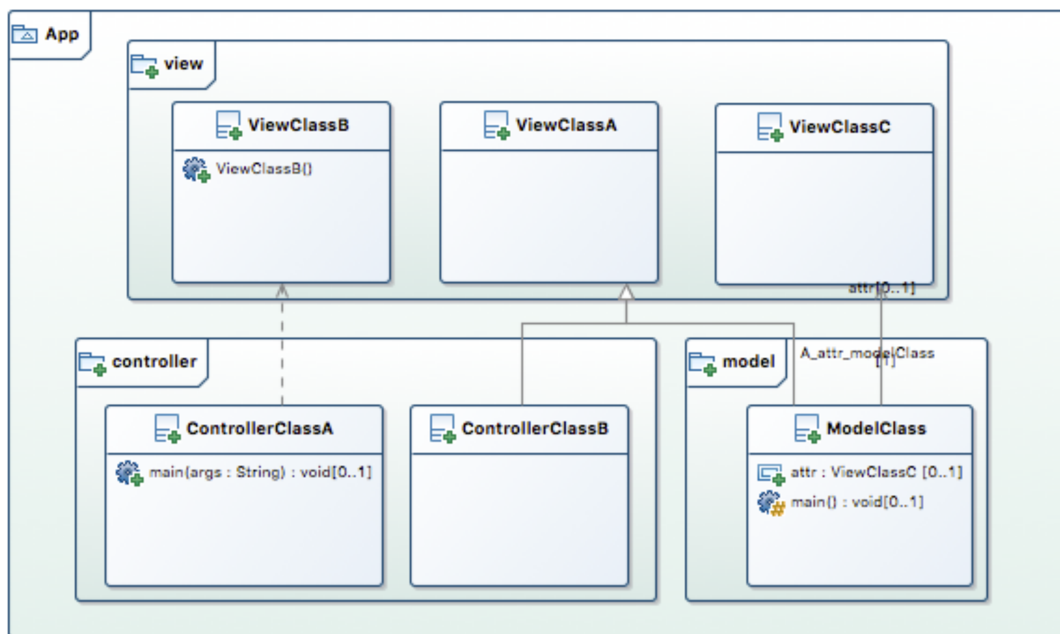

Figura 5.7 - Diagrama de classes

Segundo Fowler (2002), o padrão MVC estabelece a separação entre modelo e apresentação e um ponto chave nesta separação é a direção das dependências: a apresentação depende do modelo, mas o modelo não depende da apresentação. Com isso em mente, para a execução da CCA da ArchKDM, foram definidas as seguintes restrições da arquitetura planejada:

1) Controller pode depender da View
2) Controller pode depender da Model
3) View pode depender da Controller
4) View pode depender da Model
5) Model não pode depender da View
6) Model não pode depender da Controller

A Figura 5.8 ilustra graficamente a direção das dependências permitidas.

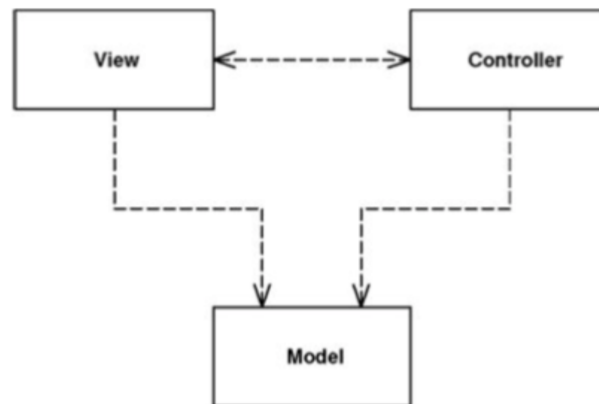

Figura 5.8 - Direção de dependências

Baseando-se nas restrições estabelecidas, nota-se uma violação do modelo MVC no sistema ilustrado na Figura 5.4 a qual a classe ModelClass possui três dependências com a camada View. Esses desvios devem, posteriormente, ser identificados pelo ArchKDM e mapeadas para o diagrama UML pelo DriV-UML.

O DriV-UML é composto por duas etapas principais, a primeira etapa se localiza dentro da Fase 1 do processo da ArchKDM (Figura 5.1). Essa etapa consiste no agrupamento das violações arquiteturais em desvios arquiteturais. Para essa tarefa, foi desenvolvida uma extensão na forma de um Algoritmo de Agrupamento, referenciado pela letra (a) na Figura 5.1. Esse algoritmo, denominado "ArchKDM2UML" deverá ser selecionado no Wizard da ArchKDM (Figura 5.9) e acionado pelo botão "Evaluate Drifts". A ArchKDM produzirá dois arquivos, um KDM completo com todas as violações arquiteturais e um segundo contendo essas mesmas violações, contudo, agrupadas em desvios. Na segunda etapa dois *discoveries* ATL são invocados automaticamente e os diagramas de classes e pacotes extraídos.

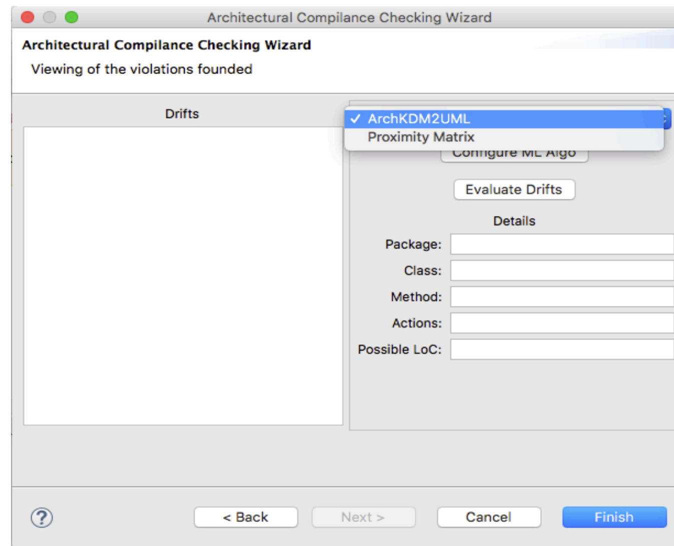

Figura 5.9 - Wizard de CCA da ArchKDM.

O DriV-UML gera como saída arquivos em conformidade com as especificações da UML 2.5 propostas pela OMG. O formato de arquivo base é XML e pode ser aberto em qualquer ferramenta que suporte essas especificações.

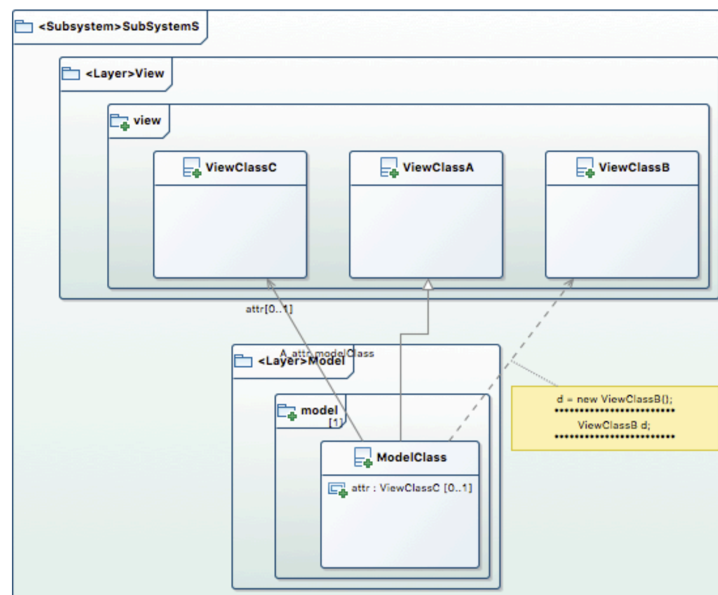

Figura 5.10 - Não Conformidades Arquiteturais

A Figura 5.10 exibe o arquivo UML final produzido pelo DriV-UML, nota-se que além da herança e associação que violam a arquitetura planejada, foi identificada a dependência que o *discovery* do Modisco não foi capaz de detectar.

Percebe-se também que os pacotes e classes foram mapeados em suas respectivas camadas, essas que por sua vez foram demarcadas pelo rótulo <Layer>. O subsistema que contém as camadas foi demarcado por <SubSystem>. O mapeamento tornou-se possível pelo fato de que, durante o processo de CCA, o engenheiro de software define quais pacotes e classes

compõem as camadas e subsistemas do software. Por fim, o DriV-UML produziu um pseudocódigo na dependência identificada. Esse pseudocódigo representa a linha de código responsável pela dependência.

5.4 Detalhes de Implementação

Para apoiar a abordagem proposta nesse projeto, foi criada uma ferramenta de produção de diagramas UML. A ferramenta foi integrada ao ArchKDM sendo composta por um algoritmo de agrupamento e dois *discoveries* UML. O algoritmo e os *discoveries* são detalhados nas seções seguintes.

5.4.1 Algoritmo de Agrupamento

O metamodelo KDM permite o armazenamento de informações em granularidade fina. Assim, em apenas um *statement* de código é possível que existam vários "relacionamentos de código (*KDMRelationships*)", isto é, instâncias de várias metaclasses. Por exemplo, em um *statement* que declara uma variável e inicializa a mesma com a instanciação de uma classe, é possível identificar pelo menos três relacionamentos de código: (i) tipagem de variável (*HasType*), (ii) instanciação de classe (*Creates*) e (iii) invocação do método construtor (*Calls*). Dessa forma, para que os diagramas UML não ficassem visualmente poluídos com dependências, um algoritmo de agrupamento foi criado com o intuito de agrupar relacionamentos. Dessa forma, ao invés de exibir cada relacionamento graficamente como uma dependência (uma linha no diagrama), cada grupo de relacionamentos é que se tornou uma dependência.

5.4.1.1 Critério de Agrupamento

O critério de agrupamento dos relacionamentos de código (*KDMRelationships)* é baseado na classe que envolve o elemento que faz referência, bem como o elemento referenciado pelo relacionamento. Isso quer dizer que, supondo dois *relacionamentos*: *relation_1* e *relation_2*. O atributo *relation_1.to* faz referência para um elemento E1 e *relation_2.to* para E2. Por sua vez o atributo *relation_2.from* possui uma referência para E3 e *relation_2.from* para E4. Se E1 e E2 são elementos que compõem uma mesma ClassUnitA, E3

e E4 elementos que compõem uma mesma ClassUnitB, então *relation_1* e *relation_2* serão agrupados. A Figura 5.11 exibe o código da função que faz a verificação descrita.

```
84  private static boolean isRelated(KDMRelationship relation_1, KDMRelationship relation_2) {
85      return getClassUnit(relation_1.getTo()) == getClassUnit(relation_2.getTo())
86          && getClassUnit(relation_1.getFrom()) == getClassUnit(relation_2.getFrom());
87  }
```

Figura 5.11 - Função de checagem de relacionamento.

```
89  public static void splitAggregatedByRelatedRelationships(EList<AbstractStructureElement> eList) {
90      for (AbstractStructureElement abstractStructureElement: eList) {
91          ArrayList<AggregatedGroup> aggregatedsGroupsList = new ArrayList<AggregatedGroup>();
92
93          for (AggregatedRelationship aggregatedRelationship: abstractStructureElement.getAggregated()) {
94              ArrayList<AggregatedRelationship> aggregatedsGroup = new ArrayList<AggregatedRelationship>();
95
96              for (int i = 0; i < aggregatedRelationship.getRelation().size(); i++) {
97                  KDMRelationship relation_1 = aggregatedRelationship.getRelation().get(i);
98
99                  for (int j = i; j < aggregatedRelationship.getRelation().size(); j++) {
100                     KDMRelationship relation_2 = aggregatedRelationship.getRelation().get(j);
101
102                     if (isRelated(relation_1, relation_2)) {
103                         AggregatedRelationship aggregated = getRelationshipGroup(relation_1, aggregatedsGroup);
104                         aggregated = (aggregated != null)?aggregated:getRelationshipGroup(relation_2, aggregatedsGroup);
105
106                         if (aggregated == null) {
107                             aggregated = CoreFactory.eINSTANCE.createAggregatedRelationship();
108                             aggregated.setFrom(aggregatedRelationship.getFrom());
109                             aggregated.setTo(aggregatedRelationship.getTo());
110                             aggregatedsGroup.add(aggregated);
111                         }
112
113                         if (!aggregated.getRelation().contains(relation_1))
114                             aggregated.getRelation().add(0, relation_1);
115                         if (!aggregated.getRelation().contains(relation_2))
116                             aggregated.getRelation().add(0, relation_2);
117                         aggregated.setDensity(aggregated.getRelation().size());
118                     }
119                 }
120             }
121
122             AggregatedGroup group = new AggregatedGroup(
123                     (AbstractStructureElement) aggregatedRelationship.getFrom(),
124                     (AbstractStructureElement) aggregatedRelationship.getTo(), aggregatedsGroup);
125
126             aggregatedsGroupsList.add(group);
127
128             group.getFrom().getOutAggregated().remove(aggregatedRelationship);
129             group.getTo().getInAggregated().remove(aggregatedRelationship);
130         }
131
132         abstractStructureElement.getAggregated().clear();
133         for(AggregatedGroup group: aggregatedsGroupsList)
134             for(AggregatedRelationship aggregated: group.getAggregatedGroup()) {
135                 abstractStructureElement.getAggregated().add(aggregated);
136                 group.getFrom().getOutAggregated().add(aggregated);
137                 group.getTo().getInAggregated().add(aggregated);
138             }
139
140         if (abstractStructureElement.getStructureElement().size() > 0)
141             splitAggregatedByRelatedRelationships(abstractStructureElement.getStructureElement());
142     }
143 }
144
```

Figura 5.12 - Agrupamento de KDMRelationships relacionados

Os relacionamentos identificados pelo ArchKDM são comparados entre si por meio de um laço bidimensional de complexidade N(N-1), como pode ser visto entre as linhas 96 e 120 da Figura 5.12. Relacionamentos que foram considerados relacionados pelo método `isRelated()` (Figura 5.11) por sua vez são separados em um agregado de relacionamentos (*AggregatedRelationships*) distintos, como pode ser visto na linha 110 da Figura 5.12. É importante mencionar que o algoritmo de agrupamento é executado recursivamente para todas

estruturas arquiteturais do sistema (subsistemas, camadas e componentes), até mesmo estruturas aninhadas. Essa recursão pode ser observada na linha 141 (Figura 5.12).

5.4.2 Discoveries

Foram criados dois discoveries, ambos foram desenvolvidos com a linguagem de transformação de modelos ATL e foram incorporados ao ArchKDM. O primeiro *discovery* denominado Discovery de Desvios em Classes (DDC) disponibiliza uma visão à nível de camadas, pacotes e classes. Nos diagramas produzidos por esse *discovery*, quatro tipos de relacionamentos serão possíveis: herança, realização, associação e dependência. O segundo *discovery*, denominado Discovery de Desvio entre Elementos Arquiteturais (DDEA) produz diagramas apenas a nível de camadas e pacotes. Nos diagramas produzidos por esse *discovery*, quaisquer relacionamentos existentes entre duas camadas serão reduzidos a uma única dependência, logo, apenas dependências serão visíveis. O objetivo do segundo *discovery* é dar ao engenheiro de software uma visão geral de quais camadas do sistema foram violadas.

5.4.3 Helpers e Rules

Os discoveries Discovery de Desvios em Classes (DDC) e Discovery de Desvio entre Elementos Arquiteturais (DDEA) compartilham juntos vinte Helpers, sendo 11 deles herdados do *discovery* UML do Modisco e nove deles acrescentados pela abordagem, a Tabela 5.1 contém a descrição de cada helper adicionado.

Tabela 5.1 - Helpers

Helper	Descrição
`getParent()`	Retorna o container imediato do elemento de contexto se e somente se, o respectivo elemento for descendente uma estrutura identificada como violação.
`isDescendantOf(element)`	Verifica se o elemento de contexto é descendente do argumento elemento.
`isEntityViolation()`	Verifica se uma determinada violação é um elemento de código. É importante mencionar que o ArchKDM não identifica violações arquiteturais que são elementos de código, apenas relacionamentos de código. Contudo, com base no KDM completo do sistema, esse método identifica se um dado elemento de código possui como descendente

	algum relacionamento identificado como violação pelo ArchKDM. Nesse caso, o respectivo elemento de código será classificado como violação.
`isRelationshipViolation()`	Verifica se uma determinada violação é um relacionamento de código.
`findViolatedStructure()`	Retorna a estrutura (Subsystem, Layer ou Component) que possui qualquer tipo de relacionamento de código identificado como violação pelo ArchKDM.
`getModelContainer()`	Retorna o CodeModel que envolve o elemento de contexto.
`getClassContainer()`	Retorna o ClassUnit que envolve o elemento de contexto.
`isViolation()`	Verifica se o elemento de contexto é uma violação, independentemente se o mesmo se trata de um elemento ou relacionamento de código.
`getViolation()`	Retorna o elemento de contexto, se o mesmo for uma violação.

As regras (rules) ATL definem os critérios de transformação entre os modelos, na implementação dos *discoveries* foram utilizadas 14 regras, uma herdada do Modisco e 13 desenvolvidas para o projeto. Tabela 5.2 descreve cada regra criada e em quais *discoveries* a mesma foi utilizada.

Tabela 5.2 - Rules

Regra	**Descrição**
`StructureModelToModel`	Transforma qualquer StructureModel do metamodelo KDM em um UML Model. StructureModels do pacote Structure do KDM são equivalentes ao elemento CodeModel do pacote Code. Utilizada no *discovery* DDC e DDEA.
`SubsystemToPackage`	Converte elementos Subsystem do KDM em Package UML. Utilizada no *discovery* DDC e DDEA.
`LayerToPackage`	Converte elementos Layer do KDM em Package UML. Camadas (Layers) que não possuem agregado de relacionamentos (AggregatedRelationships) de entrada ou saída serão ignorados pela regra. Isso quer dizer que, apenas camadas que possuem relacionamentos serão ilustradas no diagrama. Utilizada no *discovery* DDC e DDEA.
`ModuleToPackage`	Converte elementos do tipo Module do KDM em Package UML. Utilizada no *discovery* DDC e DDEA.

`ExtendsToGeneralization`	Transforma relacionamentos KDM de código do tipo Extends em Generalization (Herança). Utilizada apenas no *discovery* DDC.
`ImplementsToInterfaceRealization`	Transforma relacionamentos KDM de código do tipo Implements em Realização (Implementação). Utilizada apenas no *discovery* DDC.
`AggregatedRelationshipToDependency`	Transforma aggregatedRelationships em dependências. Esta foi a única regra cuja a abordagem de implementação foi diferente para cada um dos discoveries. A Figura 5.14 ilustra a implementação no *discovery* DDC e a Figura 5.13 no *discovery* DDEA.
`lazy createDependency`	Cria um link de dependência entre o objeto que faz referência e o objeto referenciado do objeto de contexto. Essa lazy rule é invocada em tempo de execução pela regra AggregatedRelationshipToDependency do *discovery* DDEA. Usada apenas no *discovery* DDEA.
`lazy DataElementToAssociation`	Cria o link de associação entre um atributo e a classe o qual o respectivo atributo foi tipado. Utilizada apenas no *discovery* DDC.
`StorableUnitToProperty`	Converte StorableUnits KDM em Properties. É importante mencionar que apenas StorableUnits que não são variáveis locais serão transformados. Nessa regra a lazy rule DataElementToAssociation é invocada para criar o link de associação entre o atributo e a classe de tipagem. Utilizada apenas no *discovery* DDC.
`InterfaceUnitToInterface`	Transforma InterfaceUnits KDM em Interfaces UML. Utilizada apenas no *discovery* DDC.
`ClassUnitToClass`	Transforma ClassUnits KDM em Class UML. Inner e Generic classes serão ignoradas. Utilizada apenas no *discovery* DDC.
`lazy createPseudoCode`	Cria as annotations com o pseudocódigo de uma determinada dependência.

No diagrama produzido pelo Discovery de Desvios entre Elementos Arquiteturais (DDEA), todos os relacionamentos entre dois elementos arquiteturais são sintetizados e representados por uma única dependência. Isso quer dizer que, independentemente de quantos relacionamentos (KDMRelationships) ou agregados de relacionamentos

(AggregatedRelationships) dois elementos arquiteturais possuam entre si, o diagrama exibirá apenas uma conexão de dependência. A regra ATL que implementa essa lógica no *discovery* DDEA, denominada `AggregatedRelationshipToDependency`, está ilustrada na Figura 5.13.

```
476⊖ rule AggregatedRelationshipToDependency {
477      from
478          src: kdm!AggregatedRelationship (
479              src.getModelContainer().name = 'violations' and src.relation -> notEmpty()
480          )
481      do {
482          if (thisModule.dependencies -> get(Tuple{fromEl = src.from, toEl = src.to}).oclIsUndefined()) {
483              thisModule.createDependency(src);
484          }
485          thisModule.dependencies <- thisModule.dependencies
486              -> including(Tuple{fromEl = src.from, toEl = src.to}, true);
487      }
488  }
```

Figura 5.13 - Implementação da regra de dependências (Discovery de Desvios entre Elementos Arquiteturais)

Para essa regra foi definida como padrão de correspondência "`src: kdm!AggregatedRelationship`" como pode ser visto na linha 478 da Figura 5.13. Um detalhe importante a ser mencionado é que devido ao fato de ser possível a existência de múltiplos agregados de relacionamentos entre dois elementos, essa regra de transformação pode ser invocada múltiplas vezes. Para evitar que múltiplas dependências entre dois elementos fossem criadas, foi necessário a criação de um mapa de ocorrências para cada dependência produzida, denominada "dependencies". Essa lista por sua vez é verificada a cada vez que a regra é correspondida e se, e somente se, dois dados elementos não estiverem mapeados, uma nova dependência é criada. Por fim a dependência criada é adicionada ao mapa a fim de que uma outra não seja produzida em uma posterior invocação da regra. Essa lógica está ilustrada entre as linhas 485 e 486 da Figura 5.13.

Como mencionado na Tabela 5.2, a regra de produção de dependências (`AggregatedRelationshipToDependency`) possui uma implementação diferente entre os discoveries DDC e DDEA. A Figura 5.14 ilustra a implementação da regra para o DDC.

```
626 rule AggregatedRelationshipToDependency {
627     from
628         src: kdm!AggregatedRelationship (
629             src.getModelContainer().name = 'violations' and src.relation -> exists(e |
630                     e.oclIsKindOf(kdm!Calls) or e.oclIsKindOf(kdm!Creates) or
631                     e.oclIsKindOf(kdm!UsesType) or e.oclIsKindOf(kdm!Imports) or
632                     e.oclIsKindOf(kdm!HasType) and not e.refImmediateComposite().
633                     refImmediateComposite().oclIsKindOf(kdm!ClassUnit))
634         )
635     using {
636         srcClass: kdm!DataType = src.relation -> first().from.getDataTypeContainer();
637         destClass: kdm!DataType = src.relation -> first().to.getDataTypeContainer();
638     }
639     to
640         tgt: uml!Dependency (
641             client <- srcClass.getViolation(),
642             supplier <- destClass.getViolation(),
643             ownedComment <- thisModule.createPseudoCode(src)
644         )
645     do {
646         srcClass.getParent().getViolation().packagedElement <- tgt;
647     }
648 }
```

Figura 5.14 - Implementação da regra de dependência (Discovery de Desvios em Classes)

No Discovery de Desvios em Classes cada agregado de relacionamentos pode se tornar uma dependência, contudo alguns critérios devem ser atendidos, são esses:

1) O agregado de relacionamentos deve possuir um relacionamento do tipo *HasType*, *Calls*, *UsesType*, *Imports* ou *Creates*.
2) Se agregado de relacionamentos contém um relacionamento do tipo *HasType*, o ancestral imediato do mesmo não pode ser um ClassUnit.

O primeiro critério garante que o agregado de relacionamentos possua um relacionamento que não seja herança ou realização. Por sua vez o segundo critério garante que o mesmo possua um relacionamento que não seja associação. Dessa forma é possível garantir que o respectivo agregado de relacionamentos possuirá pelo menos uma dependência. Esses critérios são ilustrados entre as linhas 629 e 633 da Figura 5.14.

Por fim, como pode-se notar na linha 643(Figura 5.14), a regra de dependências do DDC produz também um Pseudocódigo por meio da regra `createPseudoCode`. Esse pseudocódigo é ilustrado na forma de uma anotação UML e é produzido por meio de um processo de engenharia avante do KDM. Esse processo é composto por um laço que itera sob cada relacionamento e, por meio dos elementos do KDM, produz um código de sintaxe semelhante a linguagem Java.

Considerações Finais

No início do capítulo foi apresentada uma visão geral sobre a abordagem, a estratégia adotada para representar as não conformidades arquiteturais por meio da UML e um exemplo de uso da mesma. Em seguida foi detalhada a implementação do algoritmo de agrupamento bem como os critérios de agrupamento utilizados. Por fim, foi descrita a implementação de cada regra e *helper* que compõem os discoveries da abordagem.

Capítulo 6

AVALIAÇÃO

Neste capítulo é apresentada a avaliação da abordagem proposta. A efetividade da abordagem apresentada neste projeto de mestrado depende de dois pontos principais: (i) do correto agrupamento dos relacionamentos identificados pela ArchKDM e (ii) da qualidade do Discovery de Desvios entre Elementos Arquiteturais (DDEA) e Discovery de Desvios em Classes (DDC). Assim, neste capítulo apresentam-se uma análise do algoritmo de agrupamento e também uma avaliação dos *discoveries* UML. Os resultados obtidos nas análises são discutidos na Seção 6.4.

6.1 Estudo Empírico

O estudo empírico foi realizado com o propósito de verificar a confiabilidade dos diagramas produzidos pelos *discoveries* desenvolvidos nesse projeto. O planejamento do estudo foi realizado de acordo com as diretrizes propostas por Wohlin *et al*. (2000).

6.1.1 Definição do Estudo Empírico

6.1.1.1 Objetivo

O objetivo geral é investigar se a ferramenta é capaz de transformar todos desvios arquiteturais identificados pelo ArchKDM em elementos UML, bem como verificar se os elementos UML produzidos representam adequadamente os desvios. Para isso a avaliação foi dividida em duas etapas: (i) avaliação do algoritmo de agrupamento e (ii) avaliação dos *discoveries* UML

6.1.1.2 Objeto de estudo

Algoritmo de agrupamento e *discoveries* DDC e DDEA.

6.1.1.3 Abordagem Quantitativa

A corretude do algoritmo de agrupamento foi analisada no sentido de averiguar se sua execução não causada nenhum erro no conjunto de desvios identificados pela ferramenta ArchKDM. O algoritmo é então considerado correto se, após sua execução de agrupamento, todos os desvios continuavam presentes no conjunto. Assim, a corretude foi medida pela:

- Relação entre a quantidade de relacionamentos (*KDMRelationship*) agrupados e a quantidade total de relacionamentos. O objetivo dessa medida é investigar de forma preliminar se nenhum relacionamento identificado pelo ArchKDM foi perdido durante o processo de agrupamento do algoritmo.
- Relação entre a quantidade de relacionamentos replicados ou perdidos, sobre o número total de relacionamentos. Essa medida visa identificar se algum relacionamento foi perdido ou duplicado durante o agrupamento. Para isso, os relacionamentos agrupados pelo algoritmo foram comparados um a um com os relacionamentos identificados pelo ArchKDM. Se algum estivesse duplicado ou em falta, o mesmo seria contabilizado como replica ou ausência.
- Relação entre a quantidade de relacionamentos agrupados erroneamente e a quantidade total de relacionamentos. Como mencionado na Seção 5.4.1.1 cada relacionamento é agrupado de acordo com a classe que o referencia e a classe que o mesmo faz referência. Portanto se em um determinado grupo, existe algum relacionamento que não referencia ou não é referenciado pela mesma classe dos demais, este é classificado como agrupado erroneamente. A medida proposta nesse item visa avaliar a proporção de relacionamentos agrupados erroneamente.

A confiabilidade dos *discoveries* é avaliada considerando que eles devem mostrar graficamente todos os desvios arquiteturais que foram criados pelo algoritmo de agrupamento. Assim, a confiabilidade dos Discoveries foi medida pelos seguintes fatores:

- Relação de relacionamentos KDM reagrupados pelo algoritmo e que foram mapeados para relacionamentos UML pelos *discoveries*. Para o cálculo dessa medida foi necessário testar os *discoveries* com arquivos KDM de sistemas computacionais reais e com uma quantidade elevada de desvios arquiteturais. A

ideia nesse caso é investigar, para um determinado número de relacionamentos KDM reagrupados pelo algoritmo, quantos foram analisados pelo *discovery* e posteriormente transformados em elementos UML.

- Relação de tipos de desvios arquiteturais que os *discoveries* são capazes de identificar, com relação a quantidade de desvios arquiteturais que o ArchKDM consegue identificar. Para o cálculo dessa medida foi realizado inicialmente um levantamento de quais os tipos de desvios o ArchKDM identifica. O objetivo nesse caso é avaliar se os *discoveries* podem identificar todos esses tipos.

6.1.1.4 Planejamento do Estudo

Assumindo que o ArchKDM tenha extraído todos os relacionamentos de código que violam a arquitetura planejada do software. O planejamento da avaliação foi realizado de forma a responder as seguintes questões:

1) O Algoritmo de agrupamento funciona adequadamente? Ou seja, nenhum relacionamento é perdido, replicado ou agrupado erroneamente?

2) Todos os tipos de desvios que o ArchKDM é capaz de identificar, os *discoveries* conseguem ilustrar nos diagramas UML?

Para responder essas questões, foram coletadas as seguintes informações:

1. Relacionamentos que representam os desvios arquiteturais antes e após a execução do algoritmo.
2. Tipos de desvios identificados pelo DriV-UML
3. Diagrama de classe completo de uma CCA do ArchKDM

O item 1 foi usado para investigar se o algoritmo de agrupamento funciona de acordo com os critérios estabelecidos na Seção 5.4.1.1. No item 2, foi realizado um levantamento de quais os tipos de desvios o ArchKDM consegue identificar. No item 3 o ponto principal é identificar se o *discovery* UML é capaz de mapear todos os tipos de desvios para o relacionamento UML que o melhor representa.

6.1.1.5 Seleção de Contexto

Para essa avaliação foram usados dois sistemas reais: LabSys (*Laboratory System*) e MyAppointments. LabSys é um software de controle do uso de laboratório. É atualmente utilizado em produção na Universidade Federal do Tocantins (UFT). Por sua vez o MyAppointments é um sistema utilizado por Pinto e Terra (2015) em seu artigo “Processo de

Conformidade Arquitetural em Integração Contínua" e por Passos et al. (2010) no artigo "*Static Architecture Conformance Checking*". O sistema obedece uma arquitetura MVC, sendo adequado para um processo de avaliação controlada.

6.1.1.6 Instrumentação

Para essa avaliação, foi distribuída uma planilha excel aos membros de um grupo de voluntários. As planilhas contribuíram na construção do oráculo.

6.1.2 Operação

6.1.2.1 Preparação do LabSys

O sistema LabSys não possuía uma arquitetura planejada, contudo, por meio da documentação e cooperação dos desenvolvedores, foi definida uma arquitetura para o mesmo, que pode ser vista na Figura 6.1.

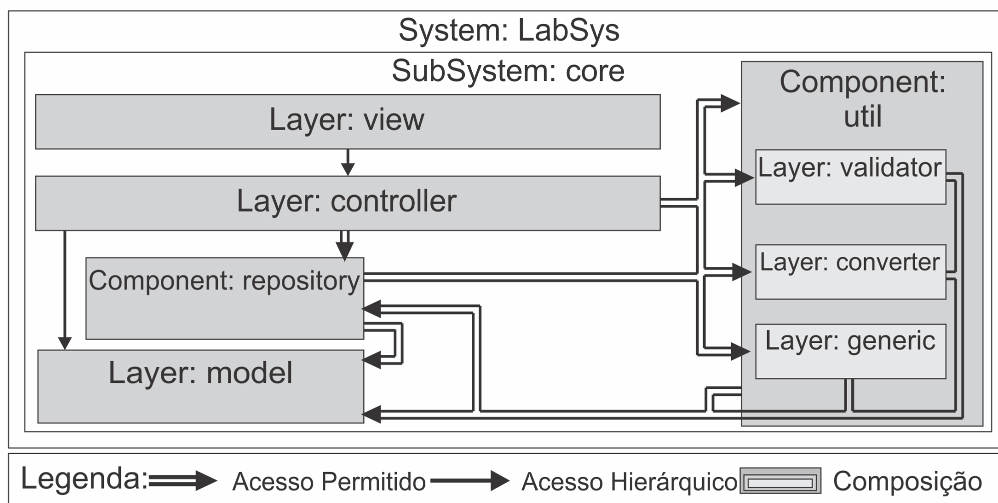

Figura 6.1 - Arquitetura planejada do sistema LabSys

De acordo com a Figura 6.1 nota-se uma arquitetura baseada no padrão arquitetural MVC. Além das camadas `Model`, `View` e `Controller` tem-se a componente `Repository` e `Util`. Esse último sendo composto por mais três camadas: `Validador`, `Converter` e `Generic`. As restrições de acesso entre os elementos arquiteturais do LabSys foram estabelecidas como pode ser visto na lista abaixo. Nota-se que o termo acesso aqui significa utilizar os elementos que compõem outro elemento arquitetural.

- `Controller` pode depender de `Repository`, `Util`, `Validator`, `Converter` e `Generic`
- `Repository` pode depender de `Model`, `Util`, `Validator`, `Converter` e `Generic`
- `Repository` pode ser acessado por `Util`, `Validator`, `Converter` e `Generic`
- `Model` pode ser acessado por `Util`, `Validator`, `Converter` e `Generic`

Para avaliação do LabSys também foi construído um oráculo contendo as violações arquiteturais do sistema. Essas violações foram identificadas manualmente por meio de uma análise linha a linha do código e os seguintes dados foram registrados:

(i) o arquivo e a linha de código da origem da violação;

(ii) o arquivo de destino da violação;

(iii) o elemento arquitetural violado;

(iv) o tipo da violação;

(v) o relacionamento UML que representa a violação;

A Figura 6.2 ilustra o exemplo de três registros do oráculo.

ID	Itens que não respeitam as Restrições Arquiteturais da model						
	Elemento origem		Elemento destino				
	Elemento de código	Linha	Elemento de código	Elemento Arquitetural	Linha	Tipo	Obs
1	Block	3	EntityConverter	converter		importacao	dependencia
2	Block	19	EntityConverter	converter		implementacao	realizacao
3	Campus	3	EntityConverter	converter		importacao	dependencia

Figura 6.2 - Oráculo do LabSys

Para construção do oráculo foram escolhidos quatro voluntários especialistas na área da computação e desenvolvimento de software: (i) um especialista em desenvolvimento de software e mestrando em ciências da computação com ênfase em modernização de software; (ii) um especialista em desenvolvimento de software e doutorando em ciências da computação com ênfase em modernização de software; (iii) um desenvolvedor de software com 10 anos de experiência na área e que trabalha atualmente em uma empresa do Setor Energético localizada na Suécia; (iv) um especialista em desenvolvimento de software com 12 anos de experiência na área, mestrando em ciências da computação com ênfase em modernização de software. Cada voluntário realizou uma análise linha a linha do sistema e posteriormente construiu um oráculo individual. Os oráculos foram comparados entre si e por fim um oráculo único foi produzido.

6.1.2.2 Preparação do Sistema MyAppointments

Para o entendimento dessa seção foi necessária uma análise do ArchKDM com o intuito de se obter os possíveis tipos de desvios que a ferramenta era capaz de identificar, são estes:

(i) Importação de classe; (ii) Declaração de atributo; (iii) Tipo de retorno de método; (iv) Tipo de parâmetro de método; (v) Implementação de interface; (vi) Herança; (vii) Declaração de variável local; (viii) Conversão de tipo; (ix) Checagem de tipo; (x) Instanciação, (xi) Chamada de método; (xii) Chamada de método estático e (xiii) Lançamento de exceção. Com isso em mente, foi inserido um único desvio de cada tipo no sistema MyAppointments. A ideia dessa abordagem é verificar se o *discovery* é capaz de identificar todos os tipos de desvios. A Tabela 6.1 exibe os desvios inseridos.

Tabela 6.1 - Desvios inseridos no sistema MyAppointments

	Origem		Destino	
Tipo do desvio	**Camada**	**Arquivo**	**Camada**	**Arquivo**
Importação de classe	util	DomainUtils.java	model	Appointment.java
Declaração de atributo	util	DomainUtils.java	model	Appointment.java
Tipo de retorno de método	util	DomainUtils.java	model	Appointment.java
Tipo de parâmetro de método	util	DomainUtils.java	model	Appointment.java
Implementação de interface	util	NewAppointment.java	model	AppointmentInterface.java
Herança	util	ExtendedAppointment.java	model	Appointment.java
Declaração de variável local	util	DomainUtils.java	model	Appointment.java
Conversão de tipo	util	DomainUtils.java	model	Appointment.java
Checagem de tipo	util	DomainUtils.java	model	Appointment.java
Instanciação	util	DomainUtils.java	model	AppointmentException.java
Chamada de método	util	DomainUtils.java	model	Appointment.java
Chamada de método estático	util	DomainUtils.java	model	AppointmentUtils.java
Lançamento de exceção	util	DomainUtils.java	model	AppointmentException.java

A arquitetura planejada definida para o MyAppointments de acordo com Pinto e Terra (2015) está ilustrada na Figura 6.3.

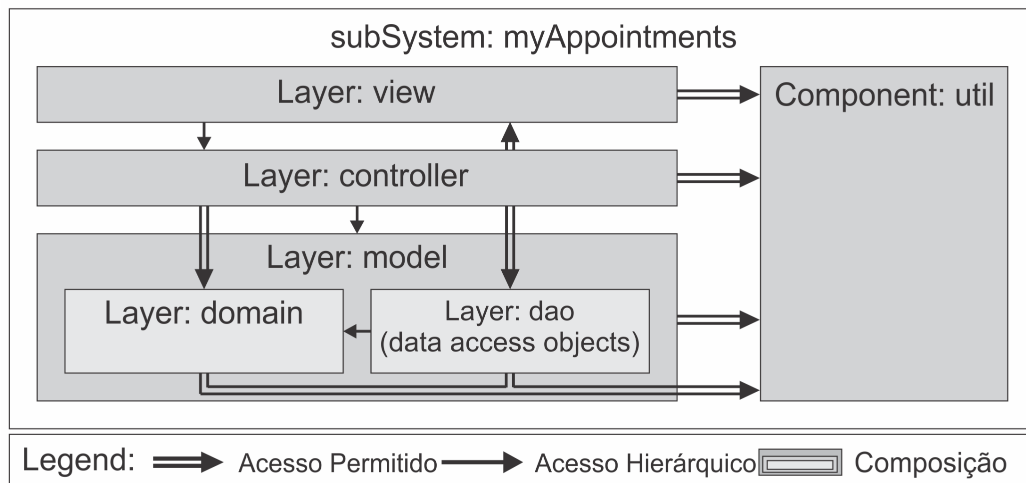

Figura 6.3 – Arquitetura Planejada do MyAppointments

O padrão arquitetural do sistema é composto por três camadas: Model, View e Controller. Foi adicionado também um componente, esse por sua vez denominado Util. A camada Model é composta por outras duas subcamadas: Domain e Dao. As definições de acesso são:

- `Controller` pode depender de `Util`, `View`, `Model`, `Dao` e `Domain`
- `Util` pode ser acessada por `Controller`, `View`, `Model`, `Dao` e `Domain`

6.1.2.3 Execução

Inicialmente foi utilizado o ArchKDM para mapear a arquitetura dos sistemas e extrair as violações arquiteturais. Em seguida, foi executado o algoritmo para agrupar as violações. Por fim, foram executados os *discoveries* ATL.

6.2 Análise do Processo de Agrupamento

O objetivo dessa etapa da avaliação é analisar o agrupamento de violações realizada pelo DriV-UML. Para isso foi utilizado o arquivo de saída do KDM contendo todos os relacionamentos identificados pelo ArchKDM para cada sistema.

6.2.1 Sistema LabSys

Por padrão o ArchKDM separa as violações em agregados de relacionamentos (*AggregatedRelationships*). Esses por sua vez são divididos em camadas de acordo com sua origem e destino. A Figura 6.4 ilustra os agregados identificados no LabSys após a CCA do ArchKDM e antes da execução do algoritmo de agrupamento.

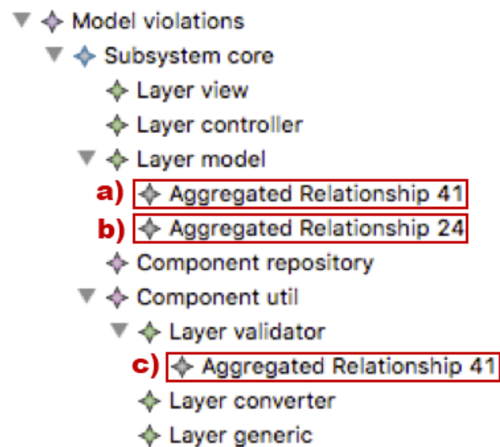

Figura 6.4 - Pré-execução do algoritmo de agrupamento (LabSys)

Como se pode notar, o ArchKDM criou três agregados de relacionamentos, referenciados na figura pelas letras: a, b e c. Os agregados localizados na camada `model`, marcados por (a) e (b), possuem 41 e 24 violações, respectivamente. O agregado ilustrado pela letra (c), localizado na camada `validator` dentro do componente `util`, possui quarenta e um violações. No total 98 violações foram identificadas.

Após a extração das violações arquiteturais pelo ArchKDM, o algoritmo de agrupamento foi executado. A Figura 6.5 ilustra os agregados identificados após a execução do algoritmo.

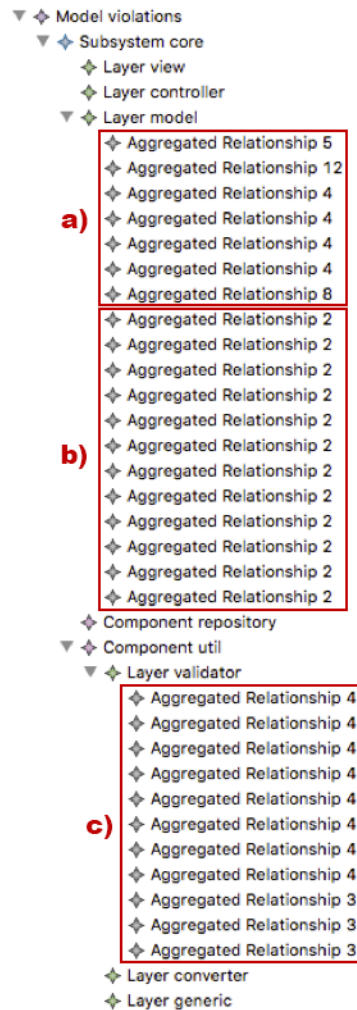

Figura 6.5 - Pós-execução do algoritmo de agrupamento (LabSys)

Como pode ser visto na Figura 6.5, após a execução do algoritmo as violações foram divididas e reagrupadas em novos agregados. No LabSys a camada `model` passou a ter 19 agregados de relacionamentos, referenciados pelas letras (a) e (b) e a camada `validator` 11, ilustrado pela letra (c). Vale ressaltar que a soma dos valores de (a), a soma de (b) e a soma (c) foram iguais aos valores de (a), (b) e (c) da Figura 6.4, respectivamente. A soma total de violações de todos os agregados foram 65 e 33 para as camadas `model` e `util`, respectivamente.

6.2.2 Sistema MyAppointments

Para o MyAppointments, o ArchKDM produziu um arquivo KDM com apenas um agregado de relacionamento. Esse por sua vez é composto de quatorze violações no componente `util`, referenciados pela letra (a). A Figura 6.6 ilustra esse arquivo.

Figura 6.6 - Pré-execução do algoritmo de agrupamento (MyAppointments)

Depois da execução do algoritmo de agrupamento o arquivo KDM passou a ter cinco agregados de relacionamentos (Figura 6.7).

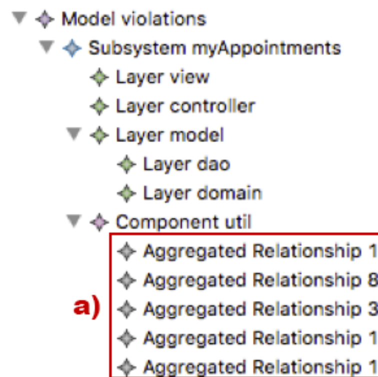

Figura 6.7 - Pós-execução do algoritmo de agrupamento (MyAppointments)

É importante ressaltar que embora a quantidade de agregados de relacionamentos tenha mudado, a quantia total de relacionamentos ainda é a mesma, quatorze. Ou seja, nenhum relacionamento foi perdido durante o reagrupamento.

6.3 Análise da Execução dos Discoveries

O objetivo dessa análise é avaliar a abrangência e confiabilidade dos *discoveries*. Para isso, foram coletados dados diretamente dos arquivos UML produzidos pelos *discoveries* DDC e DDEA, para os sistemas LabSys e MyAppointments.

6.3.1 Sistema LabSys

Por consequência da quantidade de classes violadas, o diagrama produzido pelo Discovery de Desvios em Classes não será exibido na integra. A Figura 6.8 ilustra os elementos identificados no arquivo UML pela visão do navegador XML do Modisco.

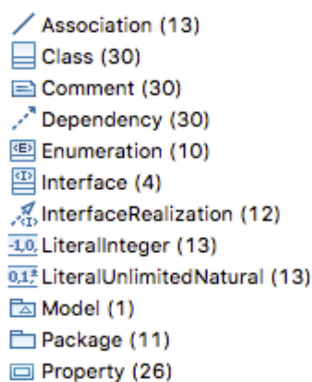

Figura 6.8 - Elementos do diagrama UML para o sistema LabSys

Foram identificados 13 desvios arquiteturais do tipo associação, 12 realizações e trinta dependências. É importante mencionar também que foram identificados 30 elementos do tipo comentário. Cada um desses por sua vez está associado a uma dependência. O motivo desse relacionamento possuir um comentário associado é que, em alguns casos, uma dependência pode representar mais do que apenas um único desvio arquitetural. Por sua vez o comentário detalha, por meio de um pseudocódigo, quais são os desvios arquiteturais que a dependência representa. A Tabela 6.2 exibe a relação da quantidade de desvios presentes nos comentários, por tipo de desvio:

Tabela 6.2 - Desvios arquiteturais do LabSys

Importação de classe	33
Tipo de retorno de método	10
Tipo de parâmetro de método	13
Instanciação	8
Chamada de método	1
Lançamento de exceção	8

Por sua vez o Discovery de Desvios entre Elementos Arquiteturais identificou que existem violações entre as camadas: *validator* e *util; model* e *util; model* e *converter*. O diagrama (Figura 6.9) produzido por esse *discovery* ilustra essas violações por meio de uma única dependência.

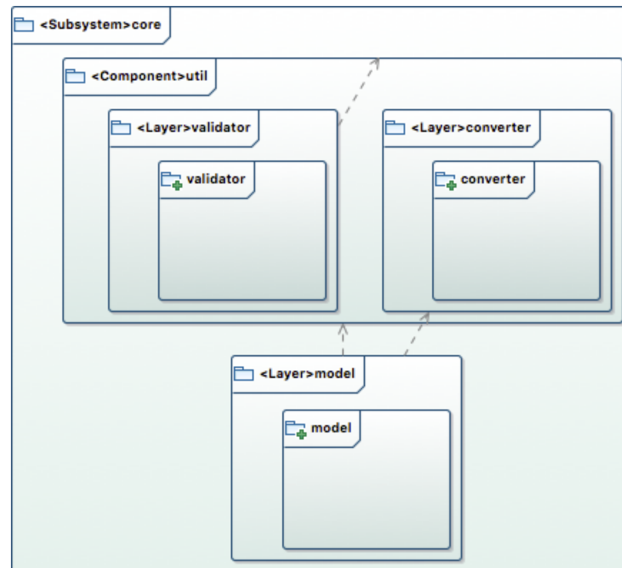

Figura 6.9 - Diagrama UML de pacotes para o sistema LabSys

6.3.2 Sistema MyAppointments

O diagrama obtido pelo DriV-UML para o sistema o MyAppointments está ilustrado abaixo.

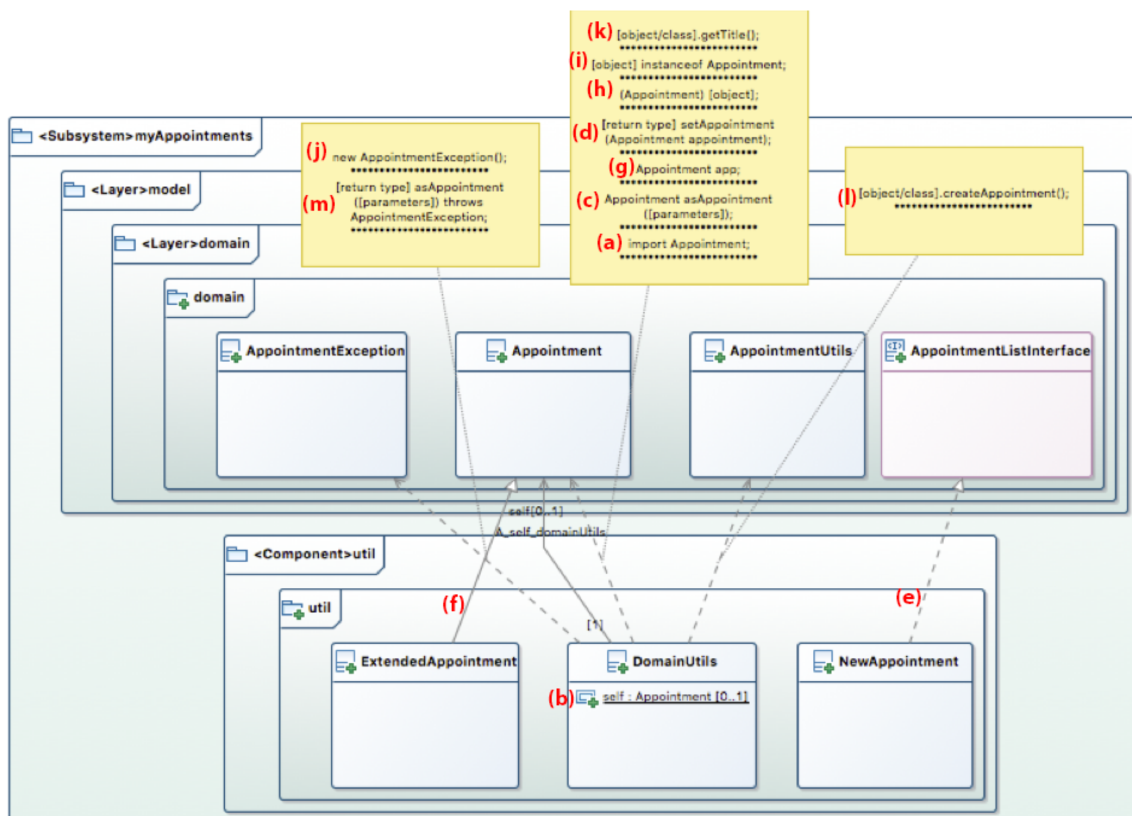

Figura 6.10 - Diagrama de desvios em classes para o sistema MyAppointments

Na Figura 6.10 pode-se notar os seguintes desvios arquiteturais:

(a) Importação de classe

(b) Declaração de atributo

(c) Tipo de retorno de método

(d) Tipo de parâmetro de método

(e) Implementação de interface

(f) Herança

(g) Declaração de variável local

(h) Conversão de tipo

(i) Checagem de tipo

(j) Instanciação

(k) Chamada de método

(l) Chamada de método estático

(m) Lançamento de exceção

Por fim, no diagrama de pacotes, foi identificada apenas uma única dependência entre as camadas *util* e *domain* (Figura 6.11).

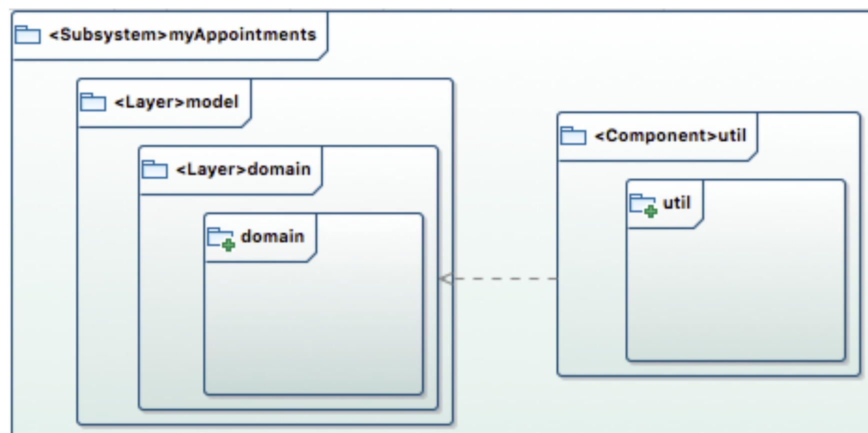

Figura 6.11 - Diagrama de pacotes para o sistema MyAppointments

6.4 Resultados obtidos

A Tabela 6.3 sintetiza os dados coletados na análise do algoritmo de agrupamento. Os dados foram separados em dois momentos: pré-execução e pós-execução do algoritmo.

Tabela 6.3 - Dados do algoritmo de agrupamento

	LabSys		MyAppointments	
	Pré-execução	Pós-execução	Pré-execução	Pós-execução
Quantidade de grupos	3	30	1	5
Total de violações agrupadas	98	98	14	14
Violações replicadas ou perdidas	0		0	
Violações agrupadas erroneamente	0		0	

Nota-se que a quantidade de grupos e de violações da pré-execução foram iguais aos valores encontrados na pós-execução do algoritmo em ambos os sistemas. Posteriormente cada violação dos grupos pós-execução foram conferidos um a um de acordo com os critérios estabelecidos no tópico 5.4.1.1. Nesse processo não foi identificada violações agrupadas erroneamente, replicadas ou perdidas (Tabela 6.3)

Para avaliar a execução do Discovery de Desvios em Classes no sistema LabSys, os desvios identificados durante a análise foram comparados aos desvios registrados no oráculo. No sistema MyAppointments por sua vez, os desvios foram comparados aos desvios inseridos controladamente durante a preparação da avaliação (6.1.2.1). A Tabela 6.4 sintetiza os dados obtidos para ambos os sistemas respectivamente com seus valores de referência.

Tabela 6.4 - Relação de desvios identificados

	LabSys		MyAppointments	
Tipos de desvios	**Oráculo**	**DriV-UML**	**Manualmente inseridos**	**DriV-UML**
Importação de classe	33	33	1	1
Declaração de atributo	13	13	1	1
Tipo de retorno de método	10	10	1	1
Tipo de parâmetro de método	13	13	1	1
Implementação de interface	12	12	1	1
Herança	0	0	1	1
Declaração de variável local	0	0	1	1
Conversão de tipo	0	0	1	1
Checagem de tipo	0	0	1	1
Instanciação	8	8	1	1
Chamada de método	1	1	1	1
Chamada de método estático	0	0	1	1
Lançamento de exceção	8	8	1	1

Como pode ser visto, tanto para o LabSys como para MyAppointments não houve diferença entre a quantidade de desvios obtidos pelo ArchKDM e DriV-UML. Com isso conclui-se que todos os desvios identificados pelo ArchKDM foram mapeados para um elemento UML.

Com o objetivo de se avaliar a execução do Discovery de Desvios entre Elementos Arquiteturais no LabSys, inicialmente foram identificados os elementos arquiteturais que possuíam desvios entre si no oráculo. Em seguida, esses elementos foram comparados aos elementos produzidos pelo *discovery* DDEA no diagrama da Figura 6.9. A Tabela 6.5 ilustra uma matriz de dependências com os resultados da análise do sistema LabSys.

Tabela 6.5 - Matriz de dependências do sistema LabSys

	Oráculo				**Discovery DDEA (Figura 6.9)**			
	Model	Validator	Util	Converter	Model	Validator	Util	Converter
Model	0	0	1	1	0	0	1	1
Validator	0	0	1	0	0	0	1	0
Util	0	0	0	0	0	0	0	0

Como pode-se notar, a camada Model possui uma dependência com a componente Util e a camada Converter, ambas dependências foram identificadas e adequadamente ilustradas na Figura 6.9

Capítulo 7

CONCLUSÕES

O tema principal dessa dissertação é o uso da UML como um meio de ilustrar desvios arquiteturais identificados por ferramentas de Checagem de Conformidade Arquitetural (CCA). No caso específico deste projeto, a ferramenta que faz a checagem de conformidade tem como saída uma instância do metamodelo KDM.

De maneira geral a avaliação realizada apresentou resultados satisfatórios tanto para o algoritmo de agrupamento quanto para os *discoveries* UML. Na avaliação do algoritmo de agrupamento notou-se que o conjunto de violações arquiteturais gerado pela ArchKDM não foi modificado após a execução do algoritmo, isto é, nenhuma violação foi replicada ou perdida (Tabela 6.3). Contudo, apenas essas informações não foram suficientes para se garantir que o algoritmo estava funcionando adequadamente. Sabendo-se disso, foi realizada uma análise manual onde cada grupo de violação foi analisado de modo que, se alguma violação pertencente a um determinado grupo não possuísse como origem e destino as mesmas classes, esse seria classificado como um desvio "agrupado erroneamente". Como mencionado na avaliação, nenhuma violação identificada como foi agrupada erroneamente (Tabela 6.3).

No caso da avaliação do Discovery de Diagramas em Classes (DDC), identificou-se que o Driv-UML foi capaz de produzir os mesmos desvios encontrados no oráculo do LabSys. Embora o *discovery* tenha ilustrado graficamente todos os desvios desse sistema, não foi possível garantir que a abordagem seria capaz de detectar todos os tipos de desvios que o ArchKDM consegue identificar. A razão disso foi o fato do LabSys não possuir todos os tipos de desvios em seu código. Em consequência disso, optou-se por uma avaliação controlada em um segundo sistema, denominado MyAppointments, cujo objetivo foi verificar se o *discovery* detectaria todos os tipos. Os resultados também foram satisfatórios, sendo que cada desvio

inserido manualmente no sistema MyAppointments foi adequadamente representado no diagrama da Figura 6.10.

Por fim, foi avaliado o *discovery* DDEA. Como mencionado anteriormente no terceiro parágrafo da Seção 5.4.3, nos diagramas produzidos por esse *discovery*, todos os desvios identificados entre dois elementos arquiteturais são sintetizados em um único relacionamento de dependência. Portanto, para avaliar os diagramas gerados por esse *discovery*, foi necessário identificar os elementos arquiteturais (camadas, componentes, etc.) envolvidos nos desvios e verificar se, entre eles, existia uma dependência nos diagramas. No caso do sistema MyAppointments os desvios foram inseridos controladamente entre as camadas Model e componente Util (Tabela 6.1) e foram devidamente ilustrados no diagrama da Figura 6.11. No caso do LabSys a Tabela 6.5 comparou a matriz de dependência entre os elementos arquiteturais identificados no oráculo e as dependências ilustradas no diagrama da Figura 6.9. Para ambos os sistemas, os resultados foram satisfatórios.

7.1 Contribuições

As principais contribuições desse trabalho são:

- Investigação sobre uma nova forma de se apresentar desvios arquiteturais de um sistema, que é usando a UML no seu formato original.
- Contribuição para a ADM (*Architecture-Driven Modernization*) no sentido de mostrar como um processo de checagem de conformidade arquitetural, inclusive com uma forma de exibir graficamente os desvios encontrados, pode ser conduzida.
- Uma ferramenta de apoio computacional que auxilie o engenheiro de software visualizar desvios arquiteturais, facilitando assim, o processo de reconciliação arquitetural.

7.2 Limitações

Embora o DriV-UML tenha ilustrado adequadamente os desvios arquiteturais, a abordagem possui limitações que estão atreladas às limitações do modelo UML, como por exemplo a questão de escalabilidade. Em outras palavras, a visualização de uma grande

quantidade de elementos UML em um único diagrama. Em sistemas com um elevado número de desvios arquiteturais, os diagramas UML visualizados por inteiro podem ficar poluídos. Em alguns casos, quando a quantidade de comentários com pseudocódigos é excessiva, as ferramentas de visualização UML podem não ter poder de processamento suficiente para renderizar todos os comentários. Essa limitação foi identificada na avaliação da abordagem com o sistema LabSys.

Outra limitação do DriV-UML é com relação a produção de elementos UML para classes aninhadas (*inner classes*). Classes aninhadas não são identificadas pelo *discovery*. Caso o ArchKDM identifique um desvio arquitetural, cuja origem ou destino seja uma classe aninhada, o DriV-UML produzirá um relacionamento UML apenas entre as classes que envolvem a classe aninhada, ou seja, o diagrama exibirá somente um relacionamento entre as classes de níveis mais elevados (*top most classes*).

Uma terceira limitação é a ausência de um experimento controlado que averiguasse a viabilidade de se mostrar desvios arquiteturais com a UML da forma como foi idealizado neste projeto. Embora as avaliações tenham mostrado bons resultados tanto para o algoritmo quanto para os *Discoveries*, não foi avaliada a abordagem de se utilizar os relacionamentos convencionais da UML como forma de desvios. Um dos problemas razoavelmente evidentes disso é, por exemplo, o uso do relacionamento de dependência. Como esse relacionamento possui um nível alto de abstração, ele agrupa vários tipos de violações arquiteturais. Dessa forma, não é possível diferenciar as violações visualmente, sendo necessário o uso de notas para isso.

REFERÊNCIAS BIBLIOGRÁFICAS

ALLILAIRE, F.; JOUAULT, F. ATL. *Eclipse.org*, 2007. Disponivel em: <http://www.eclipse.org/atl/documentation/old/ATLUseCase_Families2Persons.pdf>.

ARCHKDM. AdvanSE. Disponivel em: <http://advanse.dc.ufscar.br/index.php/tools/arch-kdm>.

BERGEY, J. et al. *Why Reengineering Projects Fail*. Software Engineering Institute, Carnegie Mellon University. Pittsburgh, p. 4. 1999. (CMU/SEI-99-TR-010).

BITTENCOURT, R. A. et al. *Improving automated mapping in reflexion models using information retrieval techniques*. 17th Working Conference on Reverse Engineering. 2010.

BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. *The Unified Modeling Language User Guide*. Addison-Wesley Professional; 2 edition, 1998.

BRODIE, M. L.; STONEBRAKER, M. DARWIN: On the Incremental Migration of Legacy Information Systems. *GTE Laboratories Incorporated*, v. TR-0222-10-92-165, p. 1, 1993.

BRUNELIÈRE, H. MoDisco in a Nutshell! How to Deal with your IT Legacy? Reverse Engineering using Models. *JavaTech Journal*, p. 21-24, 2011.

CHAGAS, F. B. *Dissertação de mestrado:* Checagem de Conformidade Arquitetural na Modernização Orientada a Arquitetura. São Carlos: 2016.

DEITERS, C. et al. *Rule-Based Architectural Compliance Checks for Enterprise Architecture Management*. 2009 IEEE International Enterprise Distributed Object Computing Conference. 2009. p. 183-192.

FOWLER, M. *Patterns of Enterprise Application Architecture*. 2002. 560 p.

FOWLER, M. *UML Distilled:* A Brief Guide to the Standard Object Modeling Language (3rd Edition). Boston: Addison-Wesley Longman Publishing Co., Inc., 2003.

IEEE. Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

JOUAULT, F.; KURTEV, I. *Transforming Models with ATL*. Verlin: Springer-Verlag, 2005. 128--138 p.

KAMAL, A. W.; AVGERIOU, P. *Modeling Architectural Patterns' Behavior Using Architectural Primitives*. Berlin: Springer Berlin Heidelberg, 2008. 164-179 p.

KAZMAN, R.; WOODS, S. G.; CARRIÈRE, S. J. *Requirements for Integrating Software Architecture and Reengineering Models:* CORUM II. Washington: IEEE Computer Society, 1998.

KDMANALYTICS. *KDM Analytics:* Working Together to Build Confidence, 2016. Disponivel em: <http://www.kdmanalytics.com/kdmspec/KDM_1.0_13_actionpkg.htm>. Acesso em: 2016.

KLEPPE, A.; WARMER, J.; BAST, W. *MDA Explained:* The Model Driven Architecture™ : Practice and Promise. Boston: Addison-Wesley Longman Publishing Co., Inc., 2003.

KNODEL, J.; POPESCU, D. *A comparison of static architecture compliance checking approaches*. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture. Mumbai: 2007. p. 12-12.

LANDI, A. D. S. *ArchKDM 2.0:* Checagem de Conformidade Aquitetural em Projetos de Modernização Dirigida a Arquitetura. 2018. 120 p.

LARMAN, C. *Applying UML and Patterns:* An Introduction to Object-Oriented Analysis and Design and Iterative Development (2nd Edition). Upper Saddle River: Prentice Hall PTR; 2 edition, 2001.

MAFFORT, C. et al. *Heuristics for Discovering Architectural Violations*. 2013 20th Working Conference on Reverse Engineering (WCRE). 2013. p. 222–231.

MARTINEZ, L.; PEREIRA, C.; FAVRE, L. *Recovering sequence diagrams from object-oriented code:* An ADM approach. 2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). 2014. p. 1-8.

MEHTA, N. R.; MEDVIDOVIC, N. Composing Architectural Styles From Architectural Primitives. *SIGSOFT Softw. Eng. Notes*, New York, September 2003.

OMG. MDA Guide Version. *OMG Homepage*, 2003. Disponivel em: <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>.

OMG. KDM Specification. *OMG*, 2011. Disponivel em: <http://www.omg.org/spec/KDM/1.3/PDF/>.

OMG. UML Specification, p. 43, 2015. Disponivel em: <http://www.omg.org/spec/UML/2.5/PDF>.

OMG ADMTF. Why do we need standards for the modernization of existing systems?, 2012. Disponivel em: <http://adm.omg.org/legacy/ADM_whitepaper.pdf>.

PÉREZ-CASTILLO, R.; GUZMÁN, I. G. R. D.; PIATTINI, M. Architecture-Driven Modernization. In: DOĞRU, A. H.; BIÇER, V. *Modern Software Engineering Concepts and Practices:* Advanced Approaches. 2011.

PASSOS, L. et al. Static Architecture-Conformance Checking: An Illustrative Overview. *IEEE Software*, v. 27, p. 82-89, setembro 2010.

PASTOR, O.; MOLINA, J. C. *Model-Driven Architecture in Practice:* A Software Production Environment Based on Conceptual Modeling. 2007.

PINTO, A. F.; TERRA, R. Processo de Conformidade Arquitetural em Integração Contínua. In: *2nd Latin-American School on Software Engineering (ELA-ES)*. 2015. p. 42-53.

PRUIJT, L. et al. *HUSACCT*: Architecture Compliance Checking with Rich Sets of Module and Rule Types. ASE. 2014.

RAHIMI, R.; KHOSRAVI, R. *Architecture conformance checking of multi-language applications*. ACS/IEEE International Conference on Computer Systems and Applications. 2010. p. 1-8.

SANGAL, N. et al. Using Dependency Models to Manage Complex Software Architecture, 2005.

SNEED, H. M. Estimating the costs of a reengineering project. *Working Conference on Reverse Engineering (WCRE),* 2005. 111-119.

STREEKMANN, N. *Clustering-Based Support for Software Architecture Restructuring*. Vieweg + Teubner Verlag, 2012.

SUTTON, A.; MALETIC, J. I. Recovering UML Class Models from C++: A Detailed Explanation. *Information and Software Technology*, 2007.

TRUYEN, F. The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture, p. 5, 2006. Disponivel em: <http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf>.

ULRICH, W.; KHUSIDMAN, V. Architecture-Driven Modernization: Transforming the Enterprise, 2007.

WARREN, I. The Renaissance of Legacy Systems - Method Support for Software-System Evolution. Secaucus: Springer-Verlag, 1999.

WOHLIN, C. et al. *Experimentation in Software Engineering:* An Introduction. Norwell: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.