UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARCH-KDM 2.0: CHECAGEM DE CONFORMIDADE ARQUITETURAL EM PROJETOS DE MODERNIZAÇÃO DIRIGIDA A ARQUITETURA

ANDRÉ DE SOUZA LANDI

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos – SP

Março/2018

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ARCH-KDM 2.0: CHECAGEM DE CONFORMIDADE ARQUITETURAL EM PROJETOS DE MODERNIZAÇÃO DIRIGIDA A ARQUITETURA

ANDRÉ DE SOUZA LANDI

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

Março/2018

UNIVERSIDADE FEDERAL DE SÃO CARLOS

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 André de Souza Landi, realizada em 27/03/2018:

Prof. Dr. Valter Vieira de Camargo
UFSCar

Prof. Dr. Fabiano Cutigi Ferrari
UFSCar

Profa. Dra. Elisa Yumi Nakagawa
USP

Certifico que a defesa realizou-se com a participação à distância do(s) membro(s) Fabiano Cutigi Ferrari 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.

Prof. Dr. Valter Vieira de Camargo

Aos meus pais.

AGRADECIMENTOS

Agradeço primeiramente a Deus por ter me dado capacidade e a oportunidade para poder ingressar em uma pós-graduação, por me guiar diante da escuridão e me manter sempre em pé diante das dificuldades, pedras e buracos.

À minha família, especialmente a meus pais Clara Aparecida de Souza Landi e Ralph Landi e minha irmã Sabrina de Souza Landi que sempre me ajudaram, incentivaram e fizeram de tudo para que este sonho se realizasse.

Aos meus grandes amigos Diego Roberto Gonçalves de Pontes e Bruno Marinho Santos, estes que apesar de todas as dificuldades e/ou empecilhos nunca deixaram de me ajudar, compartilhar seus conhecimentos e serem fieis amigos.

À minha namorada Evelyn Andrade de Moraes e ao meu amigo Jefferson Rodrigo Santos Pedro por sempre me apoiarem e não me deixarem desistir nos momentos de fraqueza.

Ao meu orientador, Valter Vieira de Camargo, pela dedicação, pelo auxílio, pelo profissionalismo e pelas orientações. Ao CNPQ pelo apoio financeiro. E ao corpo docente do curso de pós-graduação, pela boa vontade, paciência, cobrança, respeito e principalmente por nunca deixar de acolher os alunos e passar seus conhecimentos aos mesmos.

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

O real conhecimento é saber a extensão da própria ignorância.

Confúcio

RESUMO

A Modernização Dirigida à Arquitetura (ADM - do inglês *Architecture-Driven Modernization*) é uma forma de reengenharia de software baseada em conceitos do Object Management Group (OMG) e padrões ISO. Existem diversos tipos de projetos de modernização, como conversão de linguagem para linguagem, migração de plataforma, integração entre aplicações e melhorias da aplicação. Uma atividade necessária em cenários do tipo melhorias da aplicação é a reconciliação arquitetural. Dentro da reconciliação arquitetural uma etapa importante é a Checagem de Conformidade Arquitetural (CCA), cujo objetivo é identificar os desvios arquiteturais. Este projeto de mestrado consiste na evolução de uma abordagem de CCA já apresentada em outro trabalho de mestrado chamado Arch-KDM, na elaboração de um novo apoio computacional para a abordagem e também na definição formal de desvios e violações arquiteturais. A Arch-KDM possui três etapas que apoiam a CCA sendo elas: a especificação de uma arquitetura planejada, a extração da arquitetura atual e por fim, a realização da checagem de conformidade. No decorrer deste projeto, foram realizadas evoluções relativas aos diversos pontos dessa abordagem, desde correções de bugs e elementos empregados de forma errônea até a evolução da abordagem acrescentando uma nova etapa. Dessa forma, a abordagem é reapresentada neste documento com suporte a uma ferramenta denominada Arch-KDM 2.0 que inclui as evoluções e correções. Foram realizadas duas avaliações, uma no sentido de avaliar a primeira etapa da abordagem por meio de um estudo empírico avaliando e validando como uma etapa apropriada para a especificação de arquiteturas planejadas. A segunda foi a realização de um estudo empírico avaliando a precisão, *recall* e *f-measure* do apoio computacional. Os resultados obtidos foram muito promissores e satisfatórios em ambas as avaliações, de forma que comprovou-se que a primeira etapa da abordagem é apropriada para a especificação de arquiteturas planejadas e que o apoio computacional foi capaz de obter uma precisão de 78,99% comparada a manual, que é considerada muito alta.

Palavras-chave: ADM, KDM, CCA, DSL, reconciliação arquitetural, desvio arquitetural, violação arquitetural, checagem de conformidade arquitetural

ABSTRACT

Architecture-Driven Modernization (ADM) is a software reengineering process based on Object Management Group (OMG) concepts and yours ISO standards. There are several types of modernization projects, such as language-to-language conversion, platform migration, application integration, and application improvements. One necessary activity in scenario of application improvements is architectural reconciliation. An important step of the architectural reconciliation is the Architectural Conformance Checking (ACC) whose objective is to identify architectural drifts. This master's project consists of three main parts, the evolution of an ACC approach already presented in another project called Arch-KDM; a new computational support for the approach; and a formal definition of architectural drifts and violations. The original Arch-KDM approach has three steps that support the ACC proccess. They are the specification of a planned architecture; the extraction of the current architecture; and the accomplishment of the conformance checking. In the course of this project, there were evolutions related to several points of this approach like bug fixes, erroneously use of elements and a new step. Thus, the approach is re-presented in this document with a tool called Arch-KDM 2.0 that includes all evolutions and fixes. Two evaluations were carried out, one in order to evaluate the first step of the approach by means of an empirical study evaluating as an appropriate stage for the specification of planned architectures. The second one was an empirical study evaluating the precision, recall and f-measure. The results obtained were very promising and satisfactory in both evaluations. It was verified that the first step of the approach is appropriate for the specification of planned architectures and for the second evaluation the computational support was able to obtain an accuracy of 78.99% against the manual accuracy, this value is considered a high accuracy.

Keywords: ADM, KDM, ACC, DSL, architectural reconciliation, architectural drift, architectural violation, architectural conformance checking

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ALGORITMOS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

1.1 Contexto

Arquitetura de software é uma área dentro da Engenharia de Software que estuda a estrutura de sistemas do ponto de vista de componentes e módulos de nível mais alto de abstração e suas relações. A preocupação na forma de representar deixa de ser apenas entre objetos e passa a ser em nível de camadas, componentes, subsistemas, etc. Há diversos estudos sobre arquitetura e diversas propostas de arquiteturas para domínios específicos. A escolha correta de uma arquitetura para um determinado sistema impacta significativamente alguns de seus atributos como manutenibilidade, evolutibilidade e desempenho (Krueger, 1992; Shiva; Shala, 2007).

Sistemas legados são aqueles que já passaram por inúmeras atividades de manutenção e que sua arquitetura atual possivelmente não corresponde mais à arquitetura que foi planejada no início do desenvolvimento (Knodel et al., 2006; Visaggio, 2001). Dessa forma, os atributos de qualidade desses sistemas não podem ser garantidos. A arquitetura planejada de um sistema é um artefato que descreve as regras de como deve ser a relação entre os principais módulos do sistema, isto é, quais módulos podem se comunicar e quais não podem (Deiters et al., 2009; Knodel; Popescu, 2007).

Em um sistema de software, após anos de anos de manutenção, possivelmente essas regras arquiteturais são violadas e são inseridos inúmeros "desvios na arquitetura", levando a um problema conhecido como erosão arquitetural (Demeyer, 2008; Perry; Wolf, 1992). Diante da dessa erosão, a técnica de reconciliação arquitetural é uma opção para corrigir os desvios presentes na arquitetura atual de forma que o sistema volte a ter a arquitetura planejada no início do desenvolvimento e, consequentemente, satisfaça os atributos de qualidade desejados (Avgeriou; Guelfi, 2005; Grunbacher et al., 2003).

O primeiro passo dessa técnica pode ser feito com técnicas de Checagem de Conformidade Arquitetural (CCA). Há diversas pesquisas que já foram realizadas sobre CCA com resultados bastante promissores (Bittencourt et al., 2010; Deiters et al., 2009; Maffort et al., 2013; Rahimi; Khosravi, 2010; Wang; Wu et al., 2009). A técnica base deste projeto foi desenvolvida no laboratório AdvanSE e é chamada Arch-KDM (Chagas, 2016). A Arch-KDM consiste em uma abordagem de realização da CCA no contexto da Modernização Dirigida a Arquitetura utilizando-se um metamodelo denominado KDM. Conforme o objetivo desta dissertação, a partir de agora o enfoque será apenas na CCA.

Quando uma organização identifica que seu sistema de software demanda muitos recursos para se manter operacional, é recomendado que a mesma opte por realizar uma reengenharia desse sistema. Essa reengenharia, muitas vezes, é mais vantajosa do que descartar o sistema e adquirir um novo. Nessa linha de pesquisa, destaca-se a Modernização Dirigida a Arquitetura (ADM - *Architecture-Driven Modernization*), iniciada pelo OMG em 2003, cujo principal objetivo é padronizar os tradicionais processos de reengenharia com o apoio de metamodelos padrões ISO e os conceitos da MDA (*Model-Driven Architecture*).

O principal metamodelo da ADM é o KDM (*Knowledge Discovery Metamodel*), que foi criado com a intenção de ser capaz de representar todas as características, artefatos e partes de um sistema de software completo, por exemplo, código fonte, eventos, interfaces, plataforma de implantação, detalhes arquiteturais, regras de negócio, etc. Um processo típico de modernização no contexto da ADM começa com a recuperação de uma instância do metamodelo KDM que representa o sistema legado. Prossegue-se por meio de análises, refatorações e otimizações sobre essa instância e é finalizado por meio da geração do código fonte do sistema modernizado.

Há diversos cenários de modernização possíveis (Dogru, 2010), por exemplo, conversão de linguagem para linguagem, migração de plataforma, integração entre aplicações e melhorias da aplicação. Uma atividade necessária em cenários do tipo melhorias da aplicação é a Reconciliação Arquitetural e, consequentemente, a CCA tratada nesta dissertação. A principal ideia da ADM é que ferramentas que têm o intuito de realizar alguma modernização de software adotem o metamodelo KDM. Isso faz com que as ferramentas sejam desenvolvidas seguindo padrões e sejam mais interoperáveis.

1.2 Motivações

As principais motivações que justificam a elaboração desta dissertação são:

- **Carência de pesquisas acerca de checagem de conformidade arquitetural no contexto da ADM.** Como o próprio nome diz, a ADM possui enfoque em modernizações arquiteturais e para isso tem como base principal o metamodelo KDM (*Knowledge Discovery Metamodel*). Até o presente momento, não foram encontradas pesquisas sobre a CCA que leve em conta esse metamodelo, exceto o trabalho que precede o presente (Chagas, 2016). A maior parte concentra-se em UML (Avgeriou; Guelfi, 2005; Ivkovic; Kontogiannis, 2006), modelos proprietários (Allier et al., 2011; Bourquin; Keller, 2007) ou outras técnicas (Hannemann; Murphy; Kiczales, 2005; Herold; Mair, 2014; Terra et al., 2012). A utilização desses outros modelos restringe a capacidade das refatorações arquiteturais já que eles não são capazes de representar todas as abstrações e conceitos do nível arquitetural como os metamodelos padrões da OMG para ADM.

- **Difundir e possibilitar facilidades na utilização da CCA.** Conforme citado anteriormente, o processo de erosão arquitetural ocasiona diversos problemas a um sistema, e muitas vezes, é muito custoso corrigi-lo. Esta dissertação também tem como motivação a possibilidade de auxiliar a identificação das violações arquiteturais existentes no sistema legado, para que, em seguida, possam ser aplicadas refatorações.

- **Carência de pesquisas acerca de formalização dos termos da CCA.** Não foram encontrados na literatura trabalhos que realizam uma formalização dos conceitos e termos utilizados na CCA, como Violação Arquitetural e Desvio Arquitetural. Também não foram encontradas formalizações no contexto da ADM para a CCA.

- **Limitações existentes na abordagem Arch-KDM original.** A abordagem Arch-KDM (Chagas, 2016) foi desenvolvida na forma de um protótipo e apesar de operar adequadamente para alguns casos, como arquiteturas planejadas sem composição, a ferramenta não estava completa e operava em apenas oito das metaclasses do KDM que representam interações entre o código de um sistema (*Calls*, *HasType*, *Creates*, *Extends*, *Implements*, *ExceptionFlow* e *HasValue*). Apesar de identificar violações arquiteturais, os resultados não estavam adequados para sua inclusão em um cenário de modernização do tipo melhoria de aplicações, uma vez que os resultados apresentados não proporcionavam uma visualização de forma que fosse possível encontrá-los no código do sistema analisado. Portanto, identificaram-se limitações nessa abordagem que impossibilitavam seu uso por completo, de forma a acarretar possíveis falhas em sua execução. No decorrer desta dissertação, observou-se que cada uma das etapas da abordagem original necessitava de evoluções no sentido de modificar, atualizar e corrigir as limitações identificadas.

1.3 Objetivos

Os objetivos a serem alcançados por esta dissertação são:

- **Contribuir para a comunidade da ADM, apresentando uma investigação sobre a utilização do metamodelo KDM como base para o processo de CCA.** A pesquisa realizada nesta dissertação alimenta o corpo de conhecimento sobre ADM e o KDM fazendo com que essa metodologia de reengenharia de software se torne cada vez mais madura. A abordagem apresentada por este projeto mostra como o KDM pode ser usado no contexto da CCA, trazendo evidências de que sua aplicação é factível.

- **Apresentar uma formalização para os termos utilizados na CCA**. Um dos objetivos desta dissertação é apresentar uma definição de termos comuns no processo de CCA e que são de grande importância para o contexto da ADM. Dessa forma, o objetivo é apresentar uma formalização dos conceitos da Checagem de Conformidade Arquitetural, das Violações Arquiteturais e dos Desvios Arquiteturais.

- **Evoluir uma abordagem de CCA desenvolvida anteriormente.** O principal objetivo desta dissertação é evoluir uma abordagem de CCA que está em andamento pelo grupo de pesquisa AdvanSE (*Advanced Research on Software Engineering*) da Universidade Federal de São Carlos (UFSCar). Com esse objetivo em mente, esta dissertação visa evoluir as etapas da abordagem, criar uma nova etapa para a abordagem e, conforme citado anteriormente, prover uma formalização para os termos comuns da CCA e presentes na abordagem.

- **Disponibilizar um apoio computacional que apoie a utilização da abordagem.** Esse objetivo visa ao término desta dissertação prover um apoio computacional sem as limitações encontradas na abordagem anterior e, posteriormente, disponibilizado à comunidade por meio dos sites do grupo de pesquisa[1].

1.4 A Abordagem Arch-KDM

O projeto Arch-KDM está sendo desenvolvido desde 2013, quando foi iniciado no mestrado do aluno Fernando Bezerra Chagas. Esse projeto teve também a participação de outros alunos do laboratório AdvanSE, em especial do aluno Rafael Durelli, o qual contribuiu de maneira positiva à este projeto. Dessa forma, esta dissertação consiste na evolução dessa abordagem.

[1] "http://advanse.dc.ufscar.br/index.php/tools/arch-kdm" e "http://advanse.dc.ufscar.br/index.php/tools/dcl-kdm"

A abordagem Arch-KDM consiste na CCA no contexto da ADM, com o auxilio do KDM. O principal objetivo dessa abordagem é identificar desvios arquiteturais entre um modelo de arquitetura planejada e outro modelo que representa a arquitetura atual de um determinado sistema. Essa abordagem foi totalmente desenvolvida utilizando os conceitos da ADM e o KDM foi utilizado em sua forma original, sem modificações ou extensões.

A Figura 1.1 ilustra as etapas da abordagem do ponto de vista de sua utilização. Como pode ser observado, ela é composta por quatro etapas, sendo que as três primeiras já existiam na abordagem original e a etapa IV foi adicionada nesta dissertação. A Etapa I é denominada de "Especificação da Arquitetura Planejada" e tem o objetivo de criar uma especificação de arquitetura planejada que será utilizada nas demais etapas da abordagem. A execução dessa etapa é apoiada por uma DSL (*Domain-Specific Language*) denominada DCL-KDM (Landi et al., 2017) e é a responsável por definir uma especificação de arquitetura em alto nível de abstração. Ao final dessa etapa, uma instância do metamodelo KDM é gerada e que representa a arquitetura planejada do sistema.

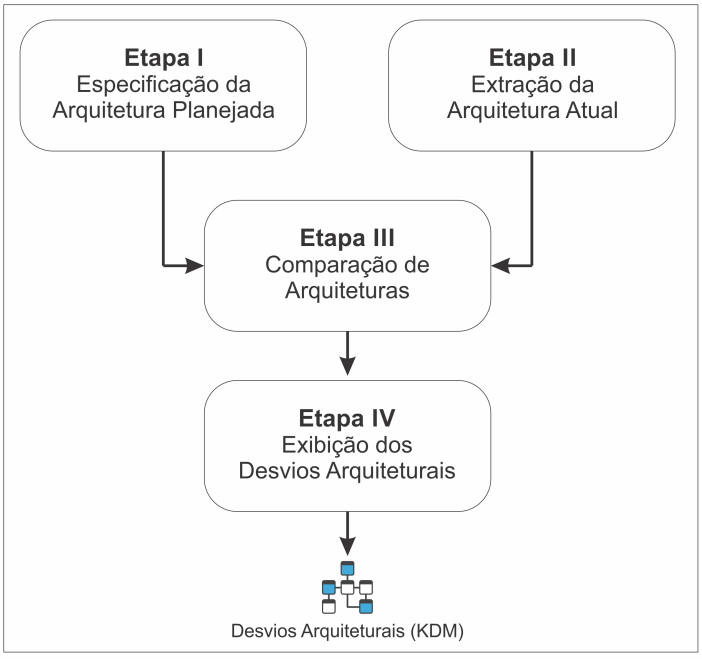

Figura 1.1: Abordagem Arch-KDM 2.0 resumida. Fonte: Elaborado pelo autor

A Etapa II é denominada "Extração da Arquitetura Atual" e tem o objetivo de criar/recuperar a arquitetura atual do sistema a ser analisado. Sua execução é apoiada por uma ferramenta já existente denominada *MoDisco* (Bruneliere et al., 2010; Brunelière et al., 2014) e um processo de mapeamento. O *MoDisco* realiza o processo de extração de informações do código transformando em modelos e o mapeamento é realizado para relacionar o código e a arquitetura planejada. Ao final dessa etapa, tem-se um modelo em KDM que representa o código e sua

relação com a arquitetura planejada do sistema.

A Etapa III é denominada de "Comparação de Arquiteturas" e tem o objetivo de realizar a Checagem de Conformidade Arquitetural propriamente dita. Ao final dessa etapa, tem-se um modelo em KDM que contem as violações arquiteturais do sistema.

A Etapa IV é denominada de "Exibição dos Desvios Arquiteturais" e tem o objetivo de exibir os problemas arquiteturais encontrados pelo apoio computacional e auxiliar o engenheiro de software a identificar os desvios arquiteturais do sistema.

1.5 Organização da Dissertação

No Capítulo 2, são descritas as fundamentações teóricas necessárias para o entendimento da abordagem e de suas evoluções/extensões. No Capítulo 3, são diferenciados os termos violação arquitetural e desvio arquitetural para o contexto deste trabalho, bem como apresentado uma definição formal para os mesmos. No Capítulo 4, é apresentada a abordagem após as evoluções, correções e extensão. No Capítulo 5, é apresentado um estudo de caso em formato de cenário de uso e escrito em forma de narrativa. No Capítulo 6, são apresentadas as avaliações realizadas. No Capítulo 7, são apresentadas a conclusão, contribuições, limitações e trabalhos futuros.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações Iniciais

No decorrer deste capítulo, são descritos os principais conceitos necessários para o entendimento desta dissertação. Na Seção 2.2 são descritos os principais conceitos e entendimentos relacionados à Arquitetura de Software e o problema denominado Erosão Arquitetural. Os conceitos relacionados a *ADM* são apresentados na Seção 2.5. Na Seção 2.6 é apresentado o principal metamodelo da ADM, o *KDM* bem como seus Pacotes *Code*, *Action* e *Structure*. Por fim, na Seção 2.7 é apresentada uma abordagem denominada *MoDisco* que apoia a utilização da ADM e do metamodelo KDM.

2.2 Arquitetura de Software e Erosão Arquitetural

O termo Arquitetura de Software consiste nos grandes módulos de um sistema e suas inter-relações, sem considerar detalhes de menor nível de abstração (Shaw; Garlan, 1996). Bosch (2000) afirmam que a especificação da arquitetura de software pode ser considerada uma das principais partes do ciclo de vida de um sistema. Isso se deve ao fato de que decisões tomadas em nível arquitetural podem possuir impactos diretos sobre o sucesso ou fracasso do sistema. Shaw e Garlan (1996) afirma que a arquitetura de software é a responsável por transformar o estudo do sistema de software estudando-os como um todo, além de seus componentes e relacionamentos.

Bass (2012) complementa as definições anteriores justificando a importância e a necessidade da arquitetura de software. Essas justificativas são: (i) a arquitetura pode ser considerada a ponte entre os requisitos e objetivos do sistema e a sua implementação; (ii) a arquitetura se

responsabiliza por atender os aspectos técnicos do sistema tão bem quanto os requisitos e objetivos; (iii) a arquitetura deve ser um modelo simples, eficaz e inteligente de estruturar o sistema e de como os seus componentes se relacionam; e (iv) a arquitetura auxilia a antecipar algumas decisões como, por exemplo, tempo de desenvolvimento, manutenção, restrições de desenvolvimento, custo e estrutura organizacional, enfatizando atributos de qualidade de software.

A representação da arquitetura de um software pode variar conforme o objetivo que se deseja observar. Essas representações geralmente são divididas em três tipos (Braga, 2013): (i) informais; (ii) semi-formais; e (iii) formais. A representação (i) informal é considerada a mais simples e rápida de se elaborar. Isto se deve ao fato de ser composta apenas por descrições textuais e diagramas de caixas e linhas. As descrições textuais são elaboradas em linguagem natural e fornecem detalhes apenas dos principais componentes da arquitetura. Já os diagramas de caixas e linhas são amplamente difundidos e simples. Nesse tipo de diagrama, as caixas representam os componentes e as linhas os relacionamentos entre os componentes (Garlan et al., 2010). Esse tipo de representação oferece como ponto positivo a linguagem natural, o que ocasiona um fácil entendimento e elaboração. Já os diagramas aparentam ser fáceis e intuitivos. Como ponto negativo tem-se que, provavelmente, o texto elaborado demandará tempo para compreender e encontrar informações específicas de componentes e suas relações, uma vez que estarão espalhados pela descrição. Já o diagrama por aparentar a simplicidade não terá detalhes estando incompletos e, muitas vezes, imprecisos.

A representação (ii) semi-formal é um detalhamento maior da representação informal onde são atribuídos significados a elementos de descrição da arquitetura. Exemplos para este tipo de representação é a UML (*Unified Modeling Language*) e o modelo de visões arquiteturais 4+1. A UML é uma notação de modelagem para sistemas de software orientados a objetos (Booch et al., 2005). O modelo 4+1 utiliza a representação em quatro visões concorrentes, no qual cada visão tem um conjunto de interesses específicos de um grupo de participantes do sistema, como programadores, especialistas, clientes e usuários finais (Kruchten, 1995). Como ponto positivo para representações semi-formais, tem-se que a UML, por exemplo, é muito difundida e a maioria dos desenvolvedores está familiarizada com seus diagramas e representações. A 4+1, por exemplo, pode trazer múltiplas visões da arquitetura. Já como pontos negativos tem-se a independência entre os modelos da 4+1 podendo impossibilitar a padronização. Quanto a UML, pode-se gerar diagramas muito complexos, que são difíceis para analisar, ou muito simples, que são insuficientes para análises.

Por fim, tem-se a representação (iii) formal considerada uma das melhores formas de se representar uma arquitetura. Isso se deve pela existência de ADLs (*Architecture Description*

Language) para domínios específicos ou para um propósito geral (Medvidovic; Taylor, 2000). Uma ADL define uma linguagem para especificar a estrutura de alto nível do sistema sem entrar em detalhes de implementação. Isso se caracteriza como formal devido a especificação possuir uma sintaxe bem definida, onde seus elementos também têm significados bem definidos.

Quando se inicia o desenvolvimento de um sistema, geralmente, é idealizada uma determinada arquitetura em uma das formas que foram citadas anteriormente. Entretanto, conforme o passar do tempo a arquitetura do sistema se desvia da que foi planejada inicialmente. Erosão arquitetural é o termo utilizado na literatura para esse tipo de problema (Perry; Wolf, 1992).

A erosão arquitetural é um dos problemas mais recorrentes que os arquitetos de software sempre enfrentam no decorrer de suas carreiras. Geralmente esse problema impacta negativamente alguns atributos de qualidade, como manutenibilidade, extensibilidade, evolução e reutilização. Em suma, a erosão arquitetural ocasiona diferenças entre a arquitetura atual e planejada de um sistema. Cada ponto de diferença entre as arquiteturas é conhecido como desvio arquitetural (Avgeriou; Guelfi, 2005; Avgeriou et al., 2005; Bourquin; Keller, 2007; Terra et al., 2012).

Esses desvios geralmente seguem padrões, dessa forma, quando comparados a padrões ou estilos arquiteturais como o MVC (*Model-View-Controller*), ficam evidentes (Buschmann et al., 1996). Por exemplo, algum componente da camada *View* acessando algum componente da camada *Model*, o que gera então um desvio no estilo arquitetural e, portanto torna a implementação do estilo incorreta. Esses padrões de forma geral podem receber um nome, o que permite serem agrupados em categorias. Essas categorias podem ser reconhecidas como *smells* arquiteturais, uma vez que não são erros do sistema, mas são indícios de algo que não foi bem realizado. Lippert e Roock (2006) afirma que desvios arquiteturais são como os *bad smells* de código fonte, ou seja, uma indicação de um problema, mas que ocorrem em uma granularidade maior que o de código fonte.

Portanto, no decorrer desta dissertação, quando se faz menção a erosão arquitetural entende-se como o fenômeno de quando uma implementação e arquitetura de um sistema se diferencia da arquitetura planejada no início do desenvolvimento desse sistema. Sua prevenção pode ser uma tarefa difícil e algumas vezes inviável. Estudos de caso da indústria, como os de O'Brien, Stoermer e Verhoef (2002) e de Bourquin e Keller (2007), demonstram que, em sua maioria, o controle da erosão arquitetural é realizado como uma atividade de reparação ao invés de uma atividade de prevenção. Quando se verifica que existe a erosão arquitetural de fato, o processo de reengenharia de software pode auxiliar na sua reparação uma vez que envolve a recuperação da arquitetura atual e a reparar/reconciliar ficando em conformidade com a arquitetura planejada (Silva, 2014).

2.3 Controle da Erosão Arquitetural e a Reconciliação Arquitetural

Existem diversas definições na literatura acerca de sistemas legados. Paradauskas e Laurikaitis (2015) definem um sistema legado como qualquer sistema de informação que se opõe a modificações e evoluções, para atender alterações de regras de negócio. Ulrich (2002) indica que sistemas legados são definidos como sendo produtos ativos independentes de características como plataforma operante, linguagem de programação ou tempo de vida. Hunt e Thomas (2002) afirmam que tudo e todo sistema se torna legado, um código fonte, por exemplo, já se torna legado assim que acaba de ser escrito.

Atualmente, muitas empresas possuem sistemas ditos legados que, mesmo sendo obsoletos, são responsáveis por funções críticas e possuem muito valor agregado (Sommerville, 2003). Esse fato se deve, uma vez que, esses sistemas têm embutido em si muito conhecimento e regras do negócio que não estão presentes em nenhum outro lugar (Sommerville, 2003). Esse conhecimento embutido, geralmente tem origem por meio da prolongação da vida útil do sistema por meio de manutenções que vão de correções até a inclusão de funcionalidades. O que, consequentemente, dificulta o descarte ou substituição do sistema, levando as empresas a lidar com os problemas como a erosão do sistema. Alguns exemplos de tais erosões enfrentadas por essas organizações são (Paradauskas; Laurikaitis, 2015): (i) uso de tecnologias obsoletas, que torna a manutenção custosa e onerosa; (ii) falta de documentação, que leva ao pouco entendimento do sistema, dificultando sua manutenção; e (iii) dificuldade de integração com outros sistemas, de modo que interfaces e conexões não possuem definições.

Conforme comentado anteriormente, o controle e prevenção da erosão arquitetural pode ser uma tarefa difícil e às vezes inviável. Entretanto, é possível e plausível de ser realizado. Em geral, as abordagens que realizam essa atividade se dividem baseando-se nos diferentes pontos de controle a ser realizado (Silva; Balasubramaniam, 2012). A primeira são as abordagens de controle que visam diminuir as ocorrências da erosão arquitetural. A segunda são as abordagens de controle que visam prevenir a erosão arquitetural, não deixando que haja seu surgimento. E por fim, as abordagens que visam reparar a erosão arquitetural de um sistema que já está deteriorado.

Cada uma das três abordagens citadas tem estratégias diferentes de alcançar seu principal objetivo. A Figura 2.1 elenca as três abordagens e as suas principais estratégias de execução segundo Silva e Balasubramaniam (2012).

No contexto desta dissertação, é focado apenas na abordagem de reparação da erosão ar-

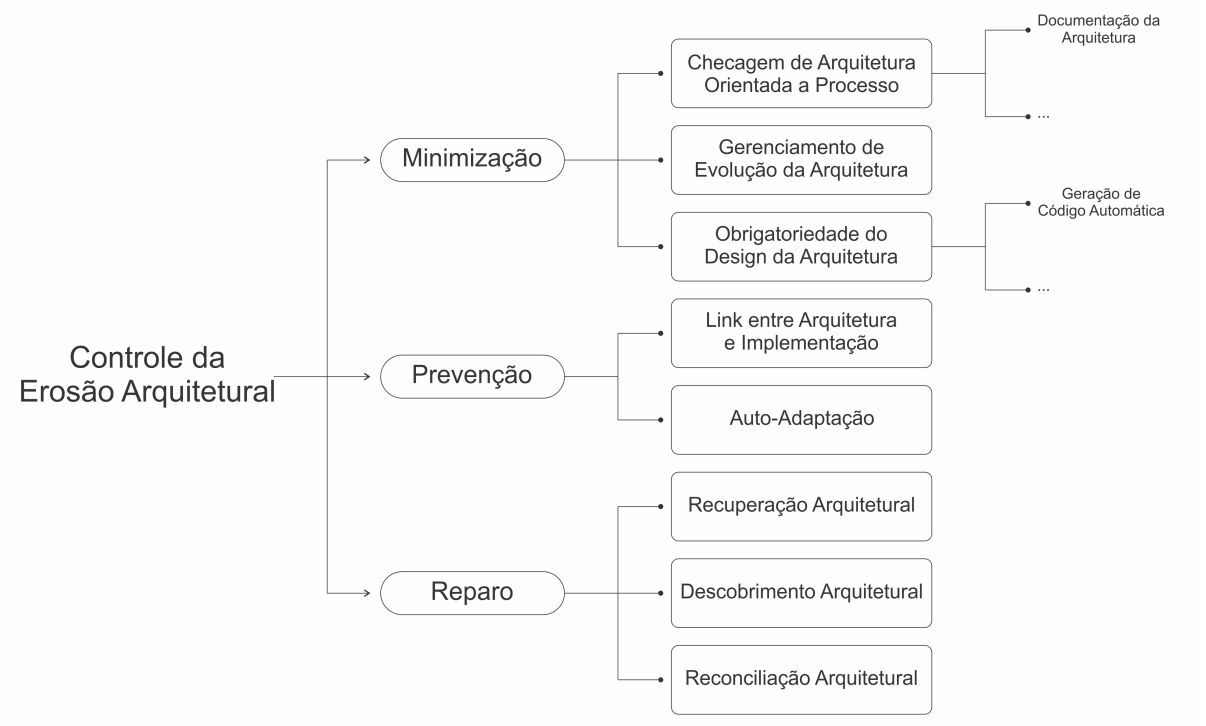

Figura 2.1: Abordagens de controle de Erosão Arquitetural. Fonte: Adaptado de Silva e Balasubramaniam (2012)

quitetural. A reparação da erosão arquitetural basicamente se divide em três técnicas que se complementam. A Recuperação da Arquitetura que envolve a extração da arquitetura a partir do código fonte e outros artefatos de software (Silva; Balasubramaniam, 2012). O Descobrimento da Arquitetura que é a técnica utilizada para elaborar uma arquitetura planejada a partir de propriedades emergentes do sistema e outros artefatos de software (Silva; Balasubramaniam, 2012). E por fim, a Reconciliação Arquitetural que é a técnica que auxilia a reduzir a diferença entre a arquitetura atual do sistema e a arquitetura planejada no início do desenvolvimento (Silva; Balasubramaniam, 2012).

A Reconciliação Arquitetural (RA) de certo modo agrupa a Recuperação da Arquitetura e o Descobrimento da Arquitetura. Isso ocorre porque a reconciliação arquitetural objetiva corrigir os desvios arquiteturais decorrentes de uma comparação entre a Arquitetura Planejada e a Arquitetura Atual de um sistema legado.

Esse processo atua de forma que a arquitetura deteriorada de um sistema seja corrigida, levando o sistema a voltar a ter a arquitetura que foi planejada inicialmente. Esses dois artefatos do processo, caso não existam por outros meios, podem ser obtidos a partir dos processos citados de Recuperação e Descobrimento da Arquitetura do sistema (Silva; Balasubramaniam, 2012).

A Reconciliação Arquitetural é composta por duas etapas principais (Avgeriou; Guelfi, 2005;

Avgeriou et al., 2005; Grunbacher et al., 2003), a primeira é denominada de CCA, cujo objetivo é identificar os desvios arquiteturais. A segunda é denominada de Refatoração Arquitetural, cujo objetivo é corrigir os desvios arquiteturais encontrados. Conforme o objetivo e motivação desta dissertação, a partir deste momento, é focado apenas na primeira etapa da RA, a CCA. Durante a seção seguinte, serão apresentados mais detalhes dessa etapa.

2.4 Checagem de Conformidade Arquitetural

Por meio do conceito de erosão arquitetural e de desvios na arquitetura é que surgiu a atividade de CCA. Segundo Knodel e Popescu (2007), essa atividade pode ser definida como sendo uma das atividades principais no controle de qualidade de software, cujo principal objetivo é revelar as diferenças entre uma arquitetura de software que foi planejada e sua real implementação. Ou seja, a CCA expõe as relações e restrições impostas pela arquitetura planejada de um software que são violadas pela real implementação do sistema.

Como resultado de uma CCA, pode-se obter três tipos de informações que podem ser relevantes ao processo de Reconciliação Arquitetural. Esses três tipos de informações são: (i) relações que foram planejadas e implementadas. Essas relações são denominadas convergências e indicam que a implementação é compatível com a arquitetura planejada. As (ii) relações que não são permitidas, porém existem na implementação (desvios arquiteturais). Essas relações são denominadas divergências e indicam que a implementação não é compatível com o modelo arquitetural planejado. As (iii) relações que não foram especificadas e foram implementadas. Essas relações são denominadas ausências e indicam que as relações na implementação não foram encontradas na arquitetura planejada. A Figura 2.2 exemplifica, simploriamente, a execução de uma CCA e esses três tipos de relações.

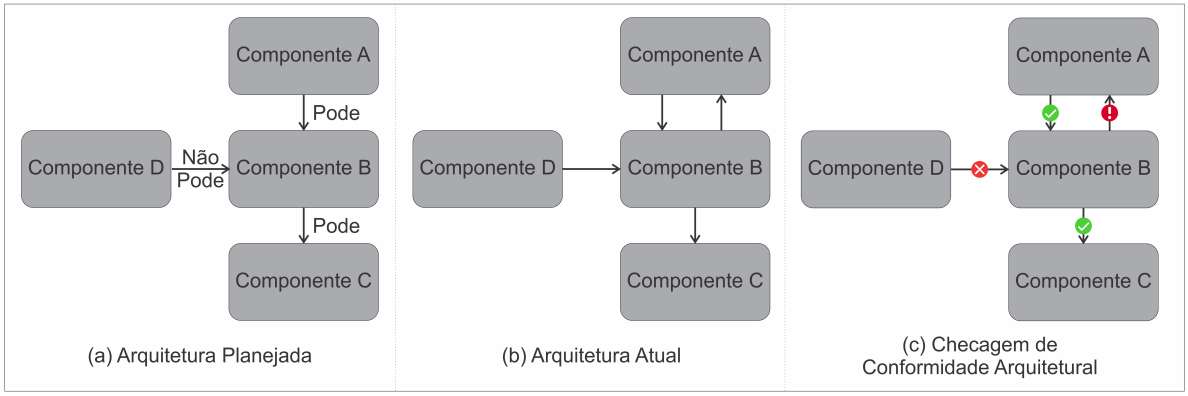

Figura 2.2: Exemplo simples de uma CCA. Fonte: Elaborado pelo autor

A Figura 2.2 é dividida em três partes. A parte (a) da figura representa a arquitetura pla-

nejada de um determinado sistema. Nessa arquitetura é possível notar que foram definidas três regras básicas que são: (i) Componente A pode acessar Componente B; (ii) Componente D não pode acessar Componente B; e (iii) Componente B pode acessar Componente C. A parte (b) da figura representa a arquitetura atual do sistema, ou seja, como o sistema esta implementado atualmente. Nessa arquitetura é possível ver que existem 4 relacionamentos entre os componentes. Por fim, a parte (c) da figura representa a execução da CCA e a identificação dos tipos de informações citadas anteriormente. Nessa parte da figura, observam-se três itens gráficos: (i) um sinal de correto na cor verde que representa a convergência; (ii) um sinal de errado na cor vermelha que representa as divergências (desvios arquiteturais); e (iii) um sinal de atenção na cor vermelha que representa as ausências.

Para executar a CCA e se obter esses três tipos de relações basicamente existem dois tipos de verificação. A verificação estática, na qual se compara o código fonte com a visão arquitetural do sistema e a verificação dinâmica, na qual se compara a execução do código fonte com a visão arquitetural do sistema. Knodel e Popescu (2007) comparam algumas técnicas de CCA e concluem que modelos de reflexão e regras de relações de conformidade são as duas principais técnicas de realização da CCA em sua forma estática.

Modelos de Reflexão é uma técnica proposta para ajudar a utilização de um modelo de alto nível de um sistema como uma lente para ver o modelo de código fonte. Geralmente, essa técnica é aplicada quando existe pouca ou nenhuma informação sobre o sistema e sua arquitetura (Murphy; Notkin; Sullivan, 2001).

Já as Regras de Relações de Conformidade especificam restrições entre os elementos arquiteturais. Restrições tais quais permitem, proíbem ou impõem as relações entre os elementos. Em comparação com os modelos de reflexão, as relações podem ser verificadas sem definir elementos arquiteturais (Knodel; Popescu, 2007). Cada regra de relação de conformidade geralmente é composta por um tipo de relação, um elemento de origem, um elemento de destino e o tipo da regra de conformidade. Nessas regras, os elementos de origem e destino são definidos por uma expressão regular que representa os nomes de cada um. O tipo de regra de conformidade determina se a relação entre os componentes é permitida, proibida ou se é obrigada a existir. Por fim, o tipo de relação define qual o tipo de dependência existe entre os elementos, como uma chamada de método, uma declaração de variável, etc. Por exemplo, a regra de relação ”Componente D é proibido Componente B”define que o componente D está proibido de acessar elementos pertencentes ao componente B.

2.5 Modernização Dirigida a Arquitetura

Com o decorrer do tempo, a reengenharia tem se tornado o principal meio para se realizar manutenções e evoluções em sistemas legados (Bianchi et al., 2001). Isso se deve ao fato do processo de reengenharia preservar ao máximo o conhecimento embutido no sistema. Dessa forma, tornando possíveis diversos tipos de alterações de maneira confiável, rápida e, muitas vezes, mais fáceis. Além de possuir um custo de manutenção relativamente aceitável (Bennett, 1995).

Contudo, estudos realizados pelo *Standish Group* International (2010), Sneed (2005, 2008) e Demeyer (2005) estimam que cerca de metade dos projetos de reengenharia falham. Essa alta quantidade de falhas, segundo Sneed (2005), se deve a dois principais motivos: (i) a dificuldade em automatizar os processos, o que aumenta os custos; e (ii) a falta de padronização e formalismos, portanto, dificilmente uma tarefa realizada em um projeto poderá ser reutilizada em outros. Sneed (2005) afirma que, devido essa falta de padronização, o desenvolvimento de ferramentas que automatizam o processo de reengenharia se torna inviável. Consequentemente, as organizações se sentem forçadas a optar por soluções *ad hocs*, essas por sua vez imaturas, porém, de menor custo. Soluções tais quais tem suas próprias ferramentas, algoritmos e metamodelos de representação que dificultam a troca de conhecimento.

Com tais problemas em mente, em 2003, o *Object Management Group* (OMG) iniciou esforços no sentido de padronizar os tradicionais processos de reengenharia de software. Por meio da *Architecture-Driven Modernization Task Force* (ADMTF) a OMG propôs então a *Architecture-Driven Modernization* (ADM). Segundo a OMG os principais objetivos da ADM são (Omg, 2009): (i) melhorar os tradicionais processos de modernização de software; (ii) consolidar melhores práticas de modernização, a fim de permitir que este processo seja bem-sucedido; (iii) tornar aplicações existentes mais ágeis; (iv) permitir a revitalização de aplicações já existentes; e (v) alavancar o uso de padrões de modelagem e a iniciativa do *Model-Driven Archtecture* (MDA) e do *Model-Driven Development* (MDD).

Um dos principais pilares da ADM está no uso de modelos, ou seja, a ADM é uma abordagem baseada em modelos e metamodelos, o que pode diminuir o nível de complexidade, uma vez que o nível de abstração é mais alto (Omg, 2016b). Portanto, a ideia foi unir os conceitos da MDA/MDD, como o PSM (*Platform Specific Model*), o PIM (*Platform Independent Model*) e o CIM (*Computational Independent Model*) e da Engenharia Reversa com alguns metamodelos padrões. Esses metamodelos criados são: o ASTM (*Abstract Syntax Tree Metamodel*); o SMM (*Software Metrics Metamodel*); o ADMPR (*ADM Pattern Recognition Specification*);

o ADMVS (*ADM Visualization Specification*); o ADMRS (*ADM Refactoring Specification*); o ADMTM (*ADM Transformation Metamodel*) e o KDM (*Knowledge Discovery Metamodel*). Até o presente momento, os metamodelos já disponibilizados são o SMM, o ASTM e o KDM, enquanto os demais, ainda encontram-se em elaboração. A Tabela 2.1 mostra a situação atual dos metamodelos da ADM.

Tabela 2.1: Situação atual dos metamodelos da ADM. Fonte: www.omg.org/spec/category/software-modernization

Metamodelo	Situação	Versão	Ano
ADMPR	em Desenvolvimento		
ADMRS	em Desenvolvimento		
ADMVS	em Desenvolvimento		
ASTM	Disponível	1.0	2011
KDM	Disponível	1.4	2016
SMM	Disponível	1.1.1	2016
ADMTM	em Desenvolvimento		

O processo de modernização da ADM é baseado na recuperação de código fonte, transformação de modelos e geração de código fonte. Sendo assim, dividido em dois domínios, que por usa vez são divididos em perspectivas arquiteturais. Os domínios são: (i) Domínio de Negócio que é dividido em apenas uma perspectiva, a perspectiva de negócios; e (ii) Domínio de Tecnologia que é dividido em duas perspectivas, a perspectiva de aplicação e de técnica. A perspectiva técnica do domínio de tecnologia é comumente utilizada e motivada pelo risco de obsolescência, custos e mudanças físicas (Ulrich; Khusidman, 2007). Esse tipo de modernização pode ser exemplificado por mudanças de plataformas, troca de linguagem, etc..

Geralmente, essa modernização é realizada diretamente em código fonte ou com modelos PSMs. Já a perspectiva de aplicação do domínio de tecnologia é comumente utilizada quando se ocorre alterações de forma que o sistema precise ser reimplementado ou ter alguma mudança pesada em sua arquitetura como um todo. Geralmente, essa modernização é realizada com o auxílio de modelos PIMs. Por fim, o domínio de negócio e sua perspectiva envolve a solução mais abrangente que transcende os modelos de aplicação e técnico para modelos arquiteturais de negócio. Essa perspectiva é comumente relacionada ao conceito do CIM, que apresenta o sistema de um ponto de vista mais abstrato possível (Truyen, 2006).

É importante ressaltar que esse processo é frequentemente retratado por meio do Modelo de Reengenharia *Horseshoe* (Ferradura) e do paradigma de evolução incremental proposto por

Kazman, Woods e Carrière (1998). A Figura 2.3 mostra o processo de modernização proposto pela ADM baseado no modelo *Horseshoe*.

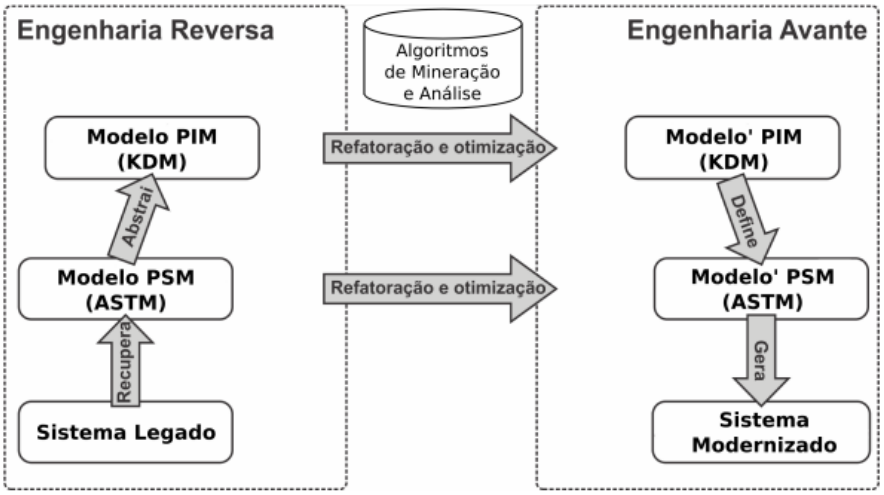

Figura 2.3: Processo *Horseshoe* de Modernização proposto pela ADM. Fonte: Adaptado de OMG (2009)

Como a maioria dos processos de reengenharia, o objetivo é resolver os problemas existentes no sistema legado obtendo-se posteriormente, uma nova versão modernizada. Para tanto, o ciclo começa com a engenharia reversa do sistema legado transformando-o em uma instância do metamodelo ASTM (PSM). Posteriormente, essa instância é transformada em uma instância do KDM (PIM). Com base nessa instância, aplicam-se algoritmos de mineração e de análise com o objetivo de identificar os problemas existentes no sistema legado. Em seguida, a etapa de refatoração e otimização é conduzida e uma nova instância do metamodelo KDM livre dos problemas identificados é gerada. Uma tecnologia comumente empregada nessa etapa é o ATL (*Atlas Transformation Language*), que é uma linguagem de transformação de modelos. Após a obtenção de um KDM modernizado, prossegue-se no ciclo da engenharia avante, gerando-se novamente uma instância do ASTM e posteriormente o código fonte do sistema antes legado e, agora modernizado.

2.6 Metamodelo de Descoberta de Conhecimento

Em 2009, o OMG, desenvolveu o KDM baseando-se nos seguintes dois fatos: (i) diversas ferramentas de mineração são capazes de extrair conhecimento de código fonte de sistemas; e (ii) essas mesmas ferramentas, em algum momento, necessitam armazenar esse conhecimento em algum lugar. Dessa forma, o KDM tornou-se o principal dos metamodelos da ADM, pois é capaz de representar o sistema legado a ser modernizado de forma independente de plataforma

e linguagem. Além disso, esse metamodelo tornou-se padrão ISO também 2009 (ISO/IEC 19506). O objetivo mais amplo do KDM é prover uma padronização do armazenamento de conhecimento recuperado de um sistema, promovendo a interoperabilidade entre ferramentas de modernização e, facilitando a troca de informações e cooperação entre elas (Omg, 2009). O KDM é baseado no modelo *Meta-Object Facility* (MOF) e utiliza o padrão XML (*Extensible Markup Language*) em conformidade com o esquema XMI (*XML Metadata Interchange*) para armazenar o conhecimento do software (Omg, 2016b).

O KDM também pode ser considerado uma família de metamodelos, pois ele é composto por metamodelos menores divididos em camadas e pacotes. Existem quatro camadas básicas no metamodelo e doze pacotes distribuídos entre elas. As quatro camadas são: (i) *Infrastructure Layer* (Camada de Infraestrutura) que contém os pacotes Core, Kdm e Source; a (ii) *Program Elements Layer* (Camada de Elementos de Programa) que contém os pacotes Code e Action; a (iii) *Runtime Resource Layer* (Camada de Recursos de Tempo de Execução) que contém os pacotes Platform, UI, Event e Data; e por fim (iv) *Abstractions Layer* (Camada de Abstração) que contém os pacotes Structure, Conceptual e Build. A Figura 2.4 apresenta graficamente essas quatro camadas e os doze pacotes.

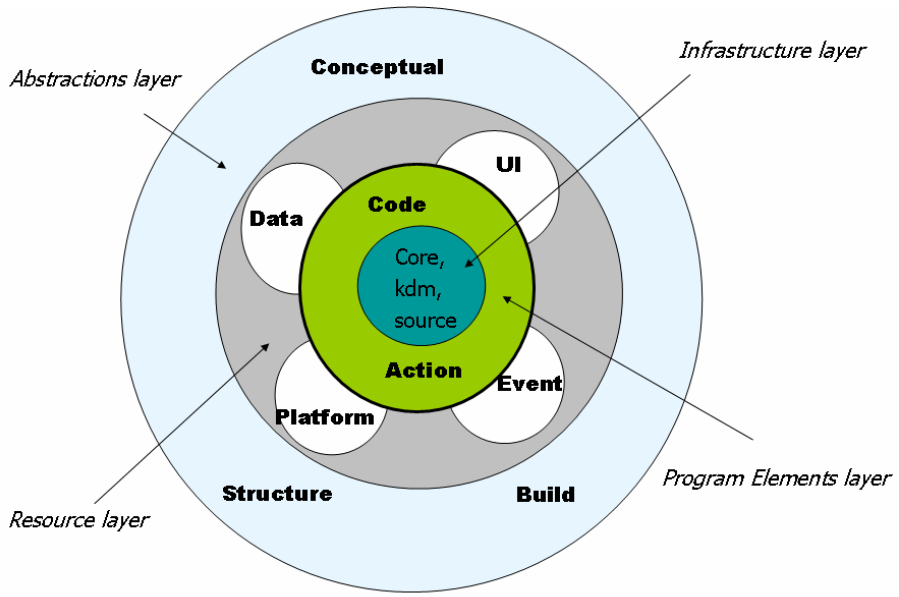

Figura 2.4: Camadas e Pacotes do metamodelo KDM. Fonte: OMG (2009)

Uma característica importante do metamodelo KDM a ser ressaltada é que todas as camadas interagem entre si, ou seja, elas são conectadas, consequentemente, se uma mudança for realizada em uma camada, a mesma deverá ser propagada para outras camadas. Isso é importante para manter todas as camadas sincronizadas e consistentes, preservando a estrutura sintática e semântica do metamodelo. Outra característica importante do metamodelo KDM é

o agrupamento do conhecimento gerado pela mineração em domínios, pelo qual cada domino representa uma visão arquitetural diferente do sistema. No total são determinados pelo KDM um total de dez domínios, cada qual com seu próprio conjunto de elementos bem definidos para representá-lo, ou seja, cada domínio é representado por um ou mais pacotes do KDM (Omg, 2016b).

No contexto deste trabalho, os pacotes que serão abordados com maior frequência são o Code, Action e Structure. O pacote Code são os elementos que representam elementos de código fonte e suas associações. O pacote Action são os elementos que representam as descrições do comportamento do código fonte, como por exemplo, operadores, fluxo de controle e declarações. Por fim, o pacote Structure são os elementos que representam a arquitetura estrutural do sistema legado, como por exemplo, componentes e camadas. As Subseções 2.6.1,2.6.2 e 2.6.3 dão detalhes acerca de cada um dos três pacotes citados.

2.6.1 Pacote Code

O pacote *Code* define metaclasses com o intuito de representar elementos em nível de implementação e suas associações. Este pacote também inclui metaclasses que representam elementos de programa comuns e suportados por várias linguagens de programação, como: classes, macros, procedimentos, tipos de dados, *templates* e protótipos (Omg, 2016a). Ao se gerar uma instância do metamodelo KDM, tem-se que cada elemento do pacote Code faz jus a um construto em uma linguagem de programação. O pacote *Code* possui um total de 24 diagrama de classes, o diagrama principal, `CodeModel`, está ilustrado na Figura 2.5.

A metaclasse `CodeModel` representa um recipiente para instâncias de elementos de código, isto é, essa metaclasse é um modelo que armazena um conjunto de elementos do sistema. É importante mencionar que os elementos do `CodeModel` são dependentes dos pacotes *Kdm*, *Source* e *Core*. Os elementos de código fonte de um sistema, por sua vez, se dividem em duas metaclasses distintas: (i) `AbstractCodeElement`, representando os símbolos e definições da linguagem; e (ii) `AbstractionCodeRelatioship`, representando os relacionamentos estabelecidos entre os elementos de código.

Em sua totalidade o pacote Code possui 90 metaclasses e todos os elementos abstratos para representar o código fonte por completo. Na Tabela 2.2 é possível observar alguns elementos de código fonte e seu respectivo mapeamento para metaclasse do KDM. É possível observar que alguns elementos de código são diretamente e facilmente mapeados para metaclasses, como classes, interfaces e métodos. Entretanto, outros elementos, suas associações e funcionamento demandam um maior conhecimento do metamodelo e do pacote Code bem como suas meta-

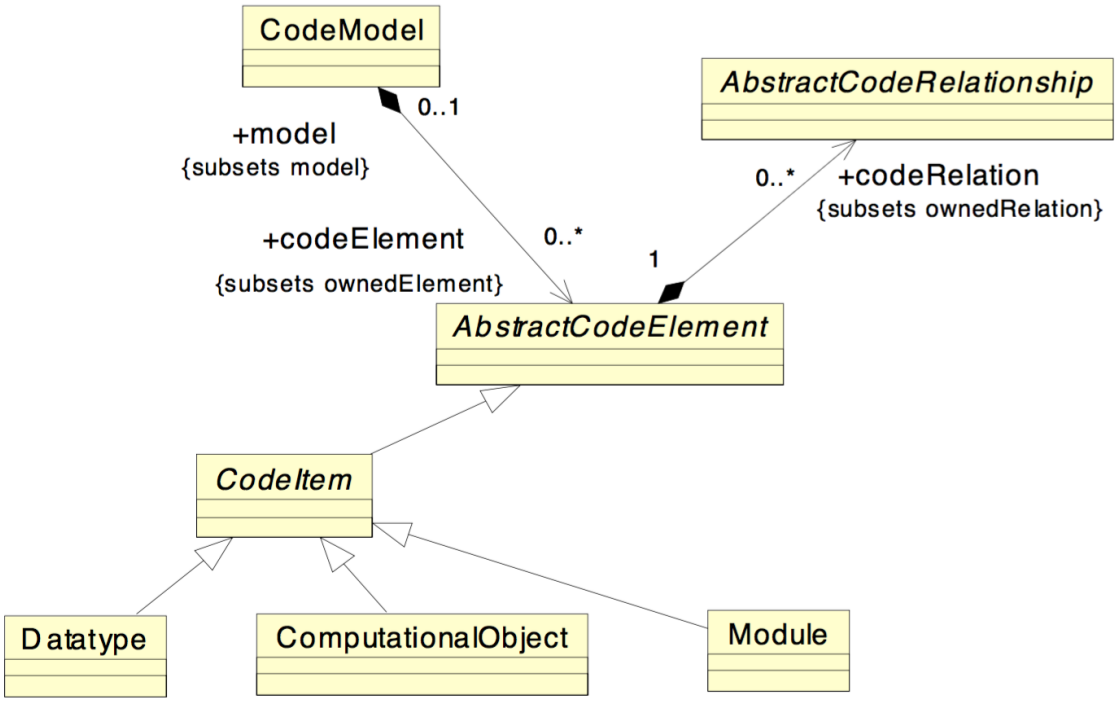

Figura 2.5: Diagrama `CodeModel` do Pacote *Code* do KDM. Fonte: OMG (2016a)

classes.

Tabela 2.2: Mapeamento de elementos de código-fonte para KDM. Adaptado de Santos (2014)

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

Com a finalidade de entender como o KDM é utilizado para representar o código fonte, no Código 2.1 e na Figura 2.6 encontram-se um exemplo simplificado de um código em Java e sua respectiva instância em KDM.

O Código 2.1 apresenta uma classe `Car` (Linha 2), pertencente ao pacote `com.br.model` (Linha 1), com um atributo privado definido como `name` do tipo `String` (Linha 4) e um método de acesso a esse atributo, de nome `getName` cujo tipo de retorno também é um `String` (Linha 6). Por fim, a classe `Car` possui como ancestral uma classe abstrata chamada `Vehicle` (Linha

2). A instância KDM está ilustrada na Figura 2.6 em um diagrama de instâncias de metaclasses. Como pode ser observado no diagrama, a metaclasse raiz é a `Segment`, que representa um conjunto significativo de fatos sobre um sistema. Cada `Segment` pode incluir uma ou mais instâncias de modelos do KDM, como por exemplo, o `CodeModel`, o `StructuralModel` e o `ConceptualModel`. A instância de `CodeModel` contêm um `Package` (Figura 2.6 - 1), dois `ClassUnits` (Figura 2.6 - 2 e 4), um `StorableUnit` (Figura 2.6 - 5), um `MethodUnit` (Figura 2.6 - 6) e um `Extends`(Figura 2.6 - 3).

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

Código 2.1: Código simples em Java. Adaptado de Durelli (2016)

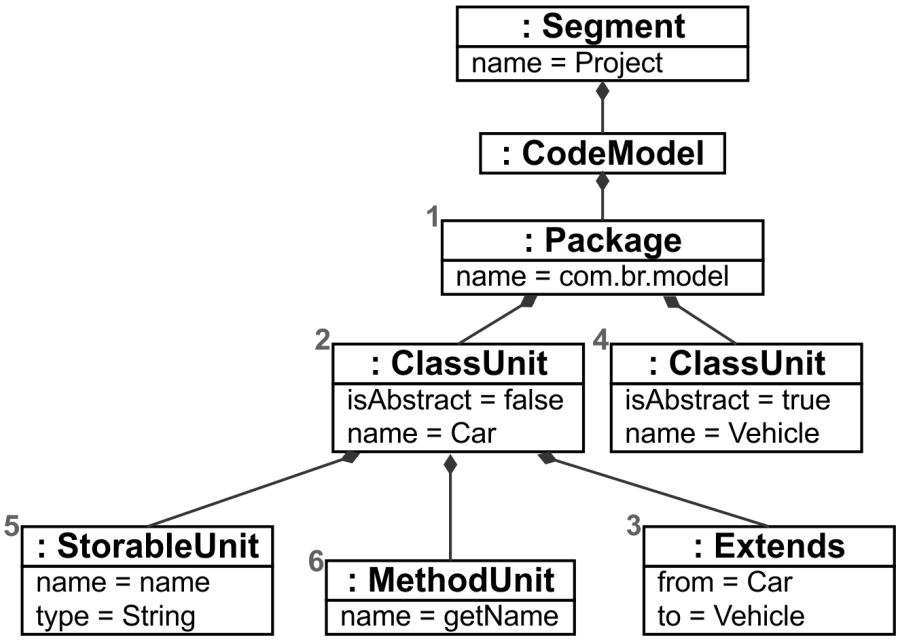

Figura 2.6: Instância em KDM do Código 2.1. Fonte: Adaptado de Durelli (2016)

Portanto, cada uma das estruturas estáticas do código fonte tem uma metaclasse específica em KDM que à representa conforme observado. Nesse exemplo, foram utilizadas apenas metaclasses para representar estruturas e construções estáticas. No entanto, o KDM possui um pacote que permite a representação de construções dinâmicas, em outras palavras, um pacote cuja finalidade é permitir e representar o comportamento do código fonte, ou seja, as relações dinâmicas de execução. Este pacote é o pacote `Action` descrito na subseção a seguir.

2.6.2 Pacote Action

O pacote *Action* define metaclasses com o intuito de representar unidades comportamentais em nível de implementação. Exemplos simples e comuns desses comportamentos são: declarações, operadores e condições de fluxo (Omg, 2016a). Ao se gerar uma instância do metamodelo KDM, tem-se que cada elemento do pacote *Action* faz jus a um comportamento em uma linguagem de programação. O pacote *Action* possui um total de 11 diagramas de classes, o diagrama principal, `ActionModel`, pode ser observado na Figura 2.7.

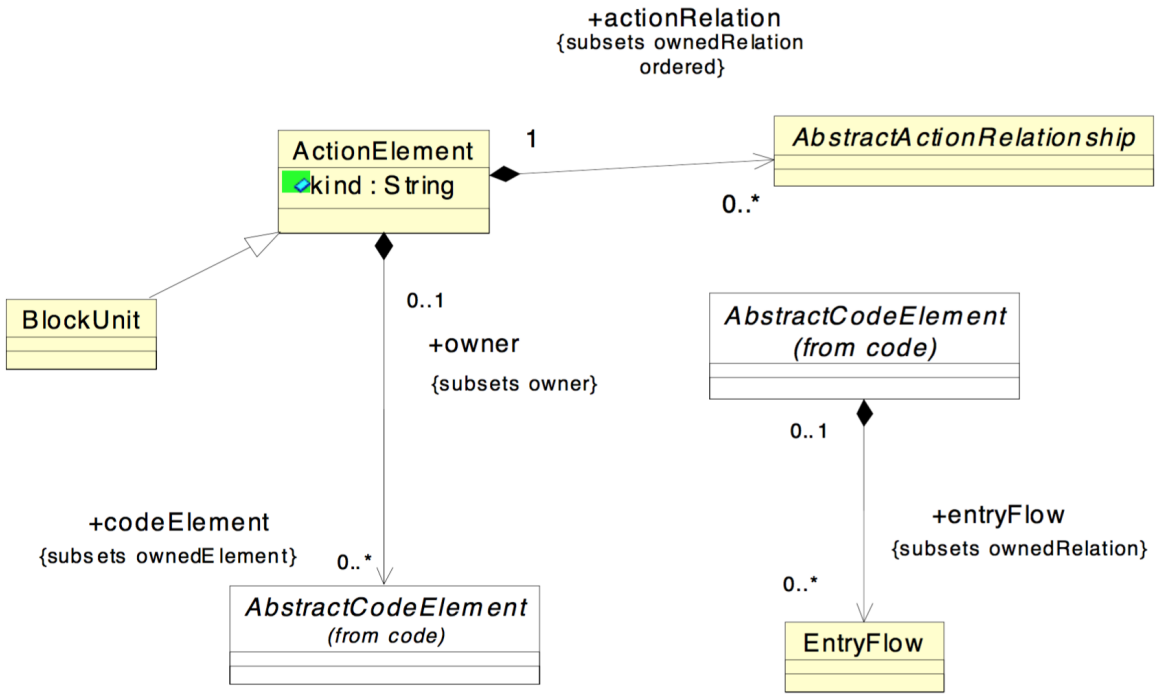

Figura 2.7: Diagrama `ActionModel` do Pacote *Action* do KDM. Fonte: OMG (2016a)

Diferentemente do diagrama do pacote *Code*, o `ActionModel` não possui uma metaclasse do tipo `Model`. No caso do pacote *Action*, essa ausência é intencional devido ao fato do pacote *Action* complementar e pacote *Code*, ou seja, seu objetivo é descrever o comportamento dos elementos do pacote *Code*. Portanto, nota-se um relacionamento de uma agregação de elementos de código ao elemento de ação (`ActionElement`). Conforme Figura 2.7, a metaclasse `ActionElement` é utilizada para descrever uma unidade básica de comportamento e possui um meta-atributo denominado `codeElement` do tipo `AbstractCodeElement` para representar o elemento gerador desse comportamento. É importante ressaltar que o `ActionElement` também pode armazenar outros elementos de ação. Isso é possível devido ao fato de que os elementos armazenados pela metaclasse são tipados como `AbstractCodeElement`, que por sua vez, é a metaclasse ancestral tanto de `CodeElement` como de `ActionElement`.

Assim como no pacote *Code*, existem alguns elementos facilmente mapeados como chamadas e criações que são representados pelas metaclasses `Calls` e `Creates`, respectivamente. Outros elementos necessitam de um maior conhecimento sobre o KDM e o pacote *Action*, por exemplo, a metaclasse `EntryFlow` que representa a ação de entrada de um elemento de código fonte.

Com a finalidade de entender como o KDM é utilizado para representar o comportamento do código fonte, no Código 2.2 e na Figura 2.8 encontra-se um exemplo simplificado de um código em Java e sua respectiva instância em KDM. Nota-se que o losango na cor cinza e anexado com três pontos (...) representa que outras metaclasses não são mostradas para simplificar a figura.

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

Código 2.2: Código simples em Java. Adaptado de Durelli (2016)

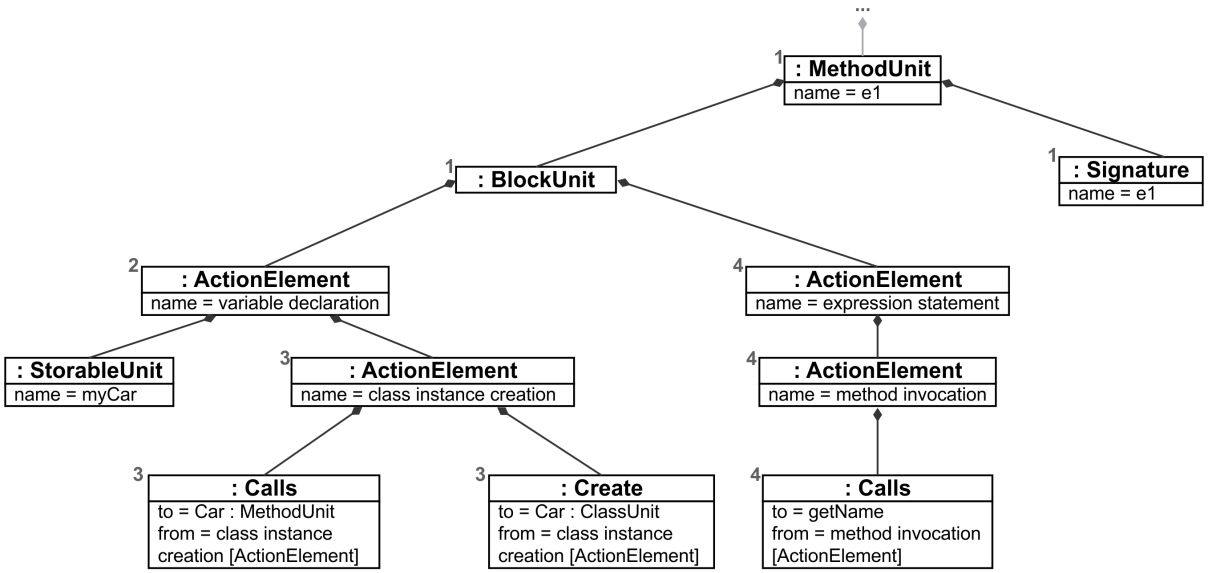

Figura 2.8: Instância em KDM do Código 2.2. Fonte: Adaptado de Durelli (2016)

O Código 2.2 apresenta um método `e1` (Linha 2) pertencente a uma classe qualquer, que possui duas linhas de código fonte. A primeira linha de código fonte realiza a declaração de uma variável de nome `myCar` e a instância (Linha 4). A segunda linha de código fonte realiza a chamada do método `getName` da variável `myCar` (Linha 5). A instância KDM está ilustrada na Figura 2.8, em um diagrama de instâncias de metaclasses. Como pode ser observado no diagrama,

existem metaclasses numeradas. As metaclasses com a numeração 1 são um `MethodUnit`, um `BlockUnit` e um `Signature`. Essas metaclasses representam uma declaração, corpo e assinatura de um método, no caso do método `e1`. Mais especificamente, a metaclasse `MethodUnit` é usada para representar o método. A metaclasse `BlockUnit` representa o escopo do método relacionado aos `ActionElement` que representam as linhas de código fonte. Já a metaclasse `Signature` representa a assinatura do método, com o nome do método, todos os parâmetros, o retorno, exceções, etc.

A metaclasse com a numeração 2 é um `ActionElement` do tipo declaração de variável. Esse `ActionElement`, por sua vez, é divido em dois filhos, uma instância de `StorableUnit` que representa a variável `myCar` e as metaclasses com a numeração 3. Essas metaclasses com a numeração 3 representam a instanciação da variável `myCar`. No caso tem-se um `ActionElement` do tipo criação de instância de classe, um `Calls` que representa a chamada do método construtor da classe e um `Create` que representa a classe que está sendo instanciada. As metaclasses com a numeração 4, por sua vez, representam a chamada do método `getName`. Neste caso, tem-se um `ActionElement` do tipo linha de código fonte, um `ActionElement` do tipo chamada de método e, por fim, o `Calls` representando a chamada do método `getName` propriamente dito.

Portanto, cada uma das estruturas dinâmicas do código fonte tem uma metaclasse especifica em KDM que a representa conforme observado. Neste exemplo, foram utilizadas metaclasses básicas para representar uma ação dinâmica comum em desenvolvimento de software. No entanto, o KDM possui uma vasta gama de metaclasses para representar ambos os aspectos de um código fonte com os Pacotes *Action* e *Code*.

2.6.3 Pacote Structure

O pacote *Structure* define metaclasses com o intuito de representar a organização do sistema em unidades de alto nível arquitetural e suas associações. Unidades arquiteturais tais quais como subsistemas, camadas e componentes. Este pacote também define a rastreabilidade desses elementos para outras metaclasses do próprio metamodelo KDM. O pacote *Structure* é composto por três diagramas de classes, sendo quinze metaclasses, e depende dos pacotes *Core* e *Kdm*, além de trabalhar em conjunto com os pacotes *Code*, *Data*, *Platform*, *UI* e *Inventory* (Omg, 2016a). O principal diagrama do pacote *Structure*, `StructureModel` está ilustrado na Figura 2.9.

A metaclasse `StructureModel` representa um recipiente para instâncias de elementos arquiteturais, ou seja, essa metaclasse é um modelo que armazena um conjunto de elementos da arquitetura do sistema. Essa classe possui um conjunto de elementos estruturais (`StructureElements`)

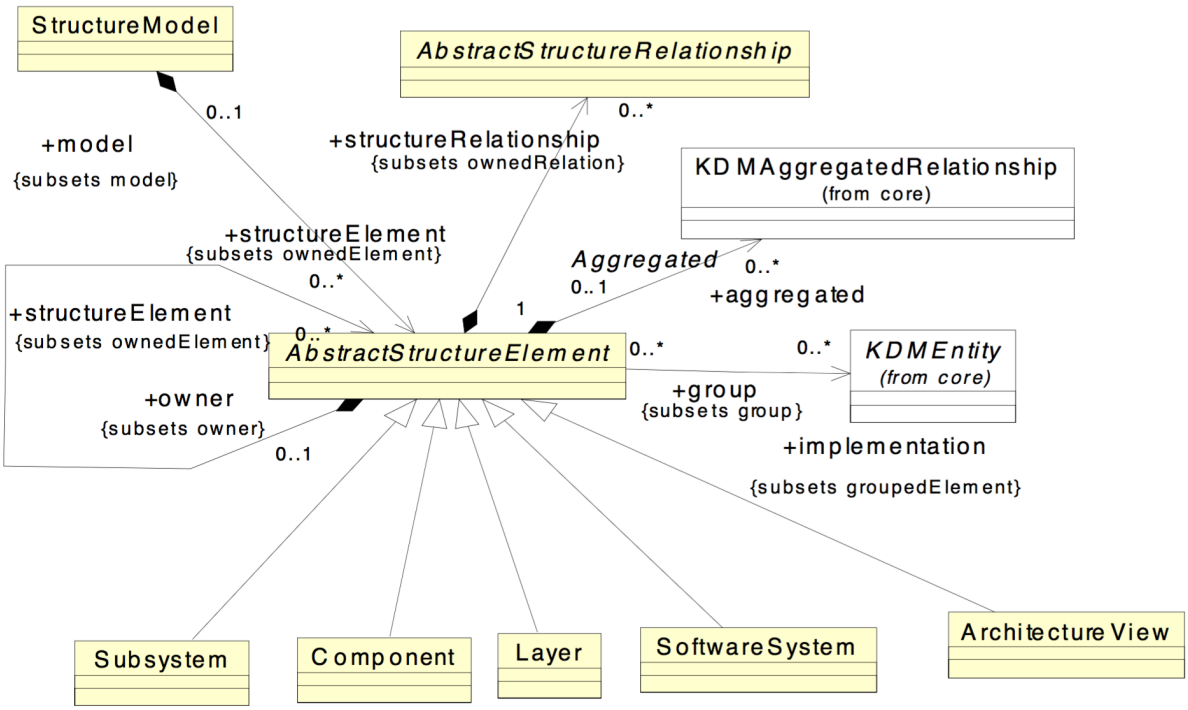

Figura 2.9: Diagrama `StructureModel` do Pacote *Structure* do KDM. Fonte: OMG (2016a)

que representam os elementos da arquitetura do software. A classe `AbstractStructureElement` representa um ancestral comum para cada um dos elementos arquiteturais e pode se especializar em elementos comuns de estilos arquiteturais (Garlan; Shaw, 1994; Shaw; Garlan, 1996) como `Subsystem`, `Component`, `Layer` e `SoftwareSystem`.

Assim como nos demais pacotes, existem alguns elementos facilmente mapeados como camadas, subsistemas e componentes. Outros elementos necessitam de um maior conhecimento sobre o pacote, por exemplo, a metaclasse `AbstractStructureElement` citada anteriormente. Essa metaclasse, além de servir como base para os elementos estruturais, define quatro associações a serem herdadas por cada elemento estrutural. A primeira associação representa elementos arquitetural pertencentes à outro elemento arquitetural, isto é, um elemento pode conter outros elementos em sua composição, gerando uma hierarquia entre os elementos. A segunda associação representa um conjunto de relacionamentos em nível arquitetural do próprio elemento. A terceira associação representa a relação entre dois elementos arquiteturais distintos. Por fim, a quarta associação representa um conjunto de elementos concretos que representam aquela abstração, ou seja, elementos de código fonte representados pelo pacote *Code*.

Com a finalidade de entender como o KDM é utilizado para representar o elementos arquiteturais do sistema, na Figura 2.10 e na Figura 2.11 encontra-se um exemplo simplificado de uma abstração gráfica de uma arquitetura de um sistema e sua respectiva instância em KDM.

Nota-se que o losango na cor cinza com filiação três pontos ("...") representa que outras metaclasses não são mostradas para simplificar a figura.

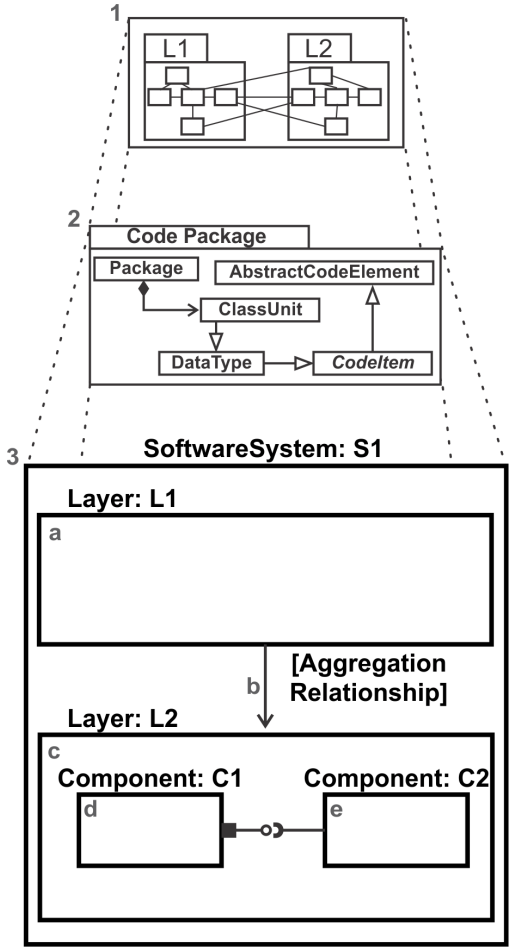

Figura 2.10: Exemplo de uma arquitetura simples, representada em código (1), instância pacote *Code* (2) e visualização arquitetural (3). Fonte: Adaptado de Durelli (2016)

Na Figura 2.10 está exemplificado uma arquitetura de um sistema hipotético utilizando-se dos elementos arquiteturais disponíveis no KDM. Para facilitar o entendimento, a imagem foi dividida em três níveis de abstração. O primeiro nível representa o sistema como código fonte separado em dois pacotes (`L1` e `L2`). Cada pacote por sua vez contém um diferente conjunto de elementos de código fonte (como classes e interfaces) e seus relacionamentos. Relacionamentos tais que ocorrem entre elementos do próprio pacote ou com elementos do outro pacote.

O segundo nível apresenta o sistema representado por meios do pacote *Code* no qual as metaclasses do KDM são usadas para representar os artefatos de baixo nível como as classes, pacotes, atributos e etc. Por fim tem-se o terceiro nível que apresenta uma perspectiva do pacote *Structure*. Nesse ponto, a arquitetura do software é representada em seu mais elevado nível de abstração. Por exemplo, no topo da composição do sistema se encontra a metaclasse do elemento arquitetural `SoftwareSystem S1`, sendo este subdividido em duas `Layers L1` e `L2`.

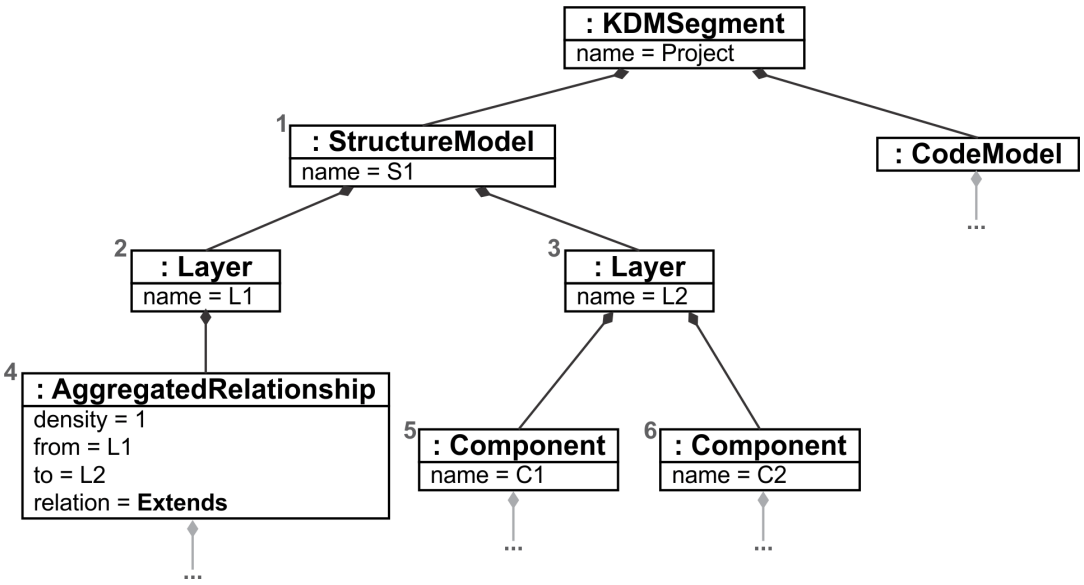

Figura 2.11: Instância em KDM da Figura 2.10. Fonte: Adaptado de Durelli (2016)

A `Layer L2` por sua vez é composta por dois `Component` (`C1` e `C2`), onde `C1` compartilha uma interface com `C2`. Além disso, observa-se que `Layer L1` possui um relacionamento com `L2`, onde a primeira pode acessar a última diretamente. Esse relacionamento é representado pela metaclasse `AggregatedRelationship`.

Na Figura 2.11 está exemplificado uma instância do Pacote *Structure* do KDM em forma de árvore. Essa instância representa a arquitetura ilustrada no nível 3 da Figura 2.10. Em uma instância do metamodelo KDM todos os elementos arquiteturais são derivadas de `StructureModel` (Figura 2.11 numeração 1). Logo abaixo do elemento arquitetural mais alto, tem-se as duas camadas representadas pela metaclasse `Layer` (Figura 2.11 numeração 2 e 3). Logo abaixo da `Layer L2` tem-se a representação dos componentes `C1` e `C2` pela metaclasse `Component` (Figura 2.11 numeração 5 e 6). Assim como demonstrado no nível 3 da Figura 2.10 (item b) a existência de um relacionamento entre as duas camadas, partindo da camada `L1` para a camada `L2`, é possível observar a instância da mesma associação na Figura 2.11 numeração 4.

Essa associação é representada pela metaclasse `AggregatedRelationship` e possui meta-atributos para informar dados sobre o relacionamento entre um elemento destino (`to`) e um elemento origem (`from`), que neste exemplo são `L1` (origem) e `L2` (destino). Os outros meta-atributos representados nessa metaclasse são `desity` e `relation`. O primeiro faz referência à densidade da relação, ou seja, o número de relações primitivas entre a origem e o destino, que neste exemplo é a densidade um. Já o segundo representa o tipo de relacionamento primitivo que os envolvidos têm, no caso deste exemplo é o tipo `Extends`, ou seja, neste contexto só existe o tipo de relacionamento “herança” entre as camadas.

2.7 Ferramenta *MoDisco*

Atualmente, um dos trabalhos mais importantes publicados no contexto da ADM e de utilização do KDM e foi a ferramenta chamada *MoDisco*, desenvolvida pela equipe *AtlanMod* (*Ecole des mines de Nantes (EMN) & Institut National de Recherche en Informatique et en Automatique (INRIA)*) (Bruneliere et al., 2010; Brunelière et al., 2014). Essa dissertação é considerado como colaborativo e é dedicado à engenharia reversa, a equipe *AtlanMod* criou o *MoDisco* em 2006 e desde então a ferramenta vem recebendo contribuições da comunidade e sendo melhorada com o passar do tempo. A empresa *Mia-Software* ingressou no projeto em 2008 e foi considerada oficialmente parte dele, contribuindo com seu desenvolvimento.

O *MoDisco* foi desenvolvido como um projeto integrado à IDE (*Integrated Development Environment*) Eclipse foi construído utilizando-se de base o *Eclipse Modeling Framework* (EMF) e oferece um conjunto de componentes reusáveis, extensíveis e baseados em modelos da ADM, principalmente o KDM, e que auxiliam o processo de engenharia reversa de sistemas legados. Basicamente o *MoDisco* recupera o código fonte legado, base de dados entre outros artefatos legados e representa os mesmos com o metamodelo KDM. Um de seus principais objetivos é ser adaptável para diferentes cenários, facilitando sua utilização por uma base de usuários potencialmente maior (Brunelière et al., 2014). A Figura 2.12 representa um segundo objetivo do *MoDisco* que é a representação de uma grande gama de artefatos de um sistema legado.

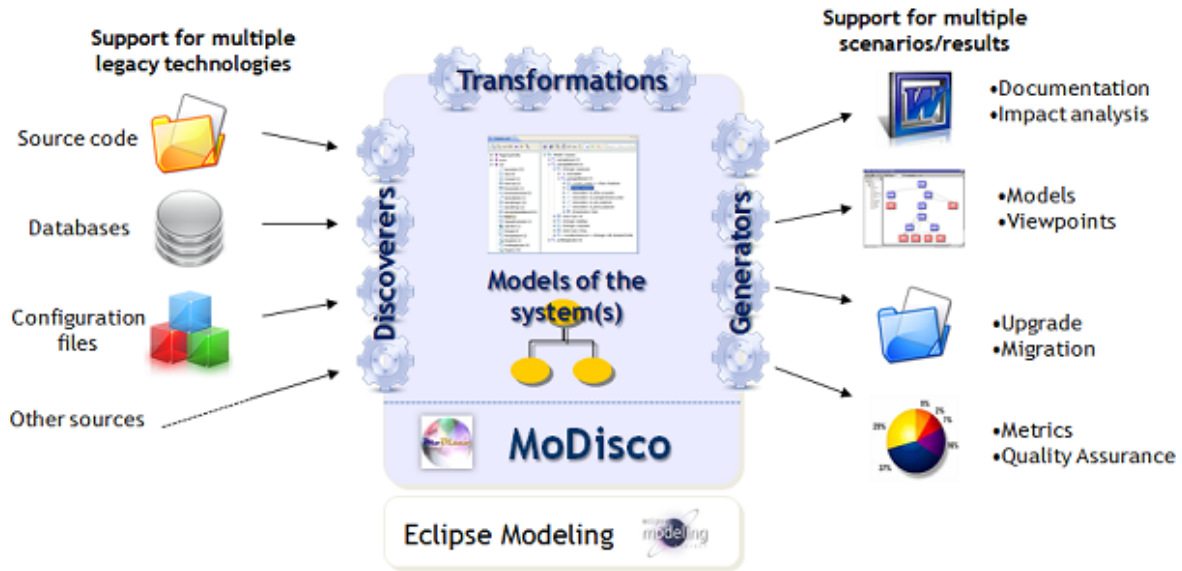

Figura 2.12: Visão geral do *MoDisco*. Fonte: Brunelière et al. (2014)

Contudo, uma das limitações dessa ferramenta é o suporte à aplicação de refatorações, manipulação automática de elementos da instância KDM e recuperação automática de informações da instância KDM. Dessa forma, o *MoDisco* não é capaz de aplicar refatorações de forma

automática, pois refatorações, geralmente, necessitam de interação do usuário para fornecer as informações necessárias. A ferramenta também não é capaz de realizar manipulações automáticas e simples de elementos em uma instancia do KDM. Por meio de sua biblioteca, é possível utilizar o KDM e transformá-lo em objetos Java para manipulação porém, a ferramenta não oferece nenhum suporte para tais manipulações. Também é digno de nota, a não possibilidade de se realizar recuperação de informações automáticas da instância do KDM através de biblioteca.

No contexto desta dissertação foi utilizado a ferramenta *MoDisco* para recuperar as informações do código fonte legado escrito em Java e transformá-lo em uma instância do metamodelo KDM. Sem o auxílio dessa ferramenta todo o sistema legado, escrito em Java, deveria ser transformado em uma instância do KDM de forma manual, o que poderia atrasar esta pesquisa, uma vez que toda a manipulação do KDM foi possível devido a existência da ferramenta e de sua biblioteca base (para Java) de manipulação de elementos do metamodelo KDM.

2.8 Considerações Finais

Neste capítulo, foram apresentados os principais conceitos para o desenvolvimento e compreensão desta dissertação. Dentre eles, foram apresentados os conceitos de arquitetura de software e suas formas de representação, assim como conceitos fundamentais para a especificação e recuperação de arquiteturas de software. Foi apresentado também, um problema recorrente no contexto da arquitetura de software, denominado erosão, que é uma das principais motivações para o desenvolvimento da abordagem que é apresentada nos capítulos seguintes. Em seguida, apresentou-se possíveis maneiras de controlar a erosão arquitetural e o processo de reconciliação arquitetural, que é responsável pelo reparo desse problema. Apresentou-se também que a reconciliação arquitetural tem duas etapas, sendo que a primeira, a Checagem de Conformidade Arquitetural, é a que será focada nessa dissertação. A CCA que é responsável por realizar a identificação dos desvios arquiteturais do software.

Abrindo uma nova linha de conceitos, foi apresentada a ADM e o KDM e seus principais conceitos para a realização de modernizações em sistemas. Dessa forma, foi apresentada a base do contexto que esta dissertação está envolta, também foi apresentada uma ferramenta que apoiam a utilização dos conceitos da ADM e a utilização do KDM. Abordagens essas, que, sem elas não seria possível realizar a condução desta dissertação.

Capítulo 3

DEFINIÇÃO FORMAL DOS CONCEITOS DE VIOLAÇÃO ARQUITETURAL E DESVIO ARQUITETURAL

3.1 Considerações Iniciais

Neste capítulo, o objetivo é explicar como os termos Violação Arquitetural e Desvio Arquitetural foram tratados nesta dissertação e por que foi necessário diferenciar e definir esses termos. Na Seção 3.2 são descritos os conceitos de cada um dos termos. Na Seção 3.3 são apresentados os termos baseados em teoria de conjuntos para a realização de uma definição formal do que é um Desvio Arquitetural e do processo de Checagem de Conformidade Arquitetural. Por fim, na Seção 3.4 é apresentada a formalização desenvolvida.

3.2 Violações e Desvios Arquiteturais

Com o andamento do projeto, notou-se que seria necessário diferenciar e definir formalmente os termos Violação Arquitetural e Desvio Arquitetural. Dessa forma, baseando-se em um estudo aprofundado das características de representação do KDM, notou-se características únicas da representação de um código em KDM. Por exemplo, o KDM representa cada característica de uma determinada linha de código por meio de ações instanciadas em metaclasses, ou seja, instâncias das metaclasses do Pacote *Action* e do Pacote *Code* do KDM.

De forma geral, para esta dissertação, os termos Violação Arquitetural e Desvio Arquitetural podem ser definidos da seguinte forma: i) Violações Arquiteturais são os relacionamentos primitivos (granularidade muito baixa) de uma linha de código e; ii) Desvios Arquiteturais são

agrupamentos de violações arquiteturais, portanto, representam algo concreto do sistema como uma linha de código fonte.

Para facilitar o entendimento dessa representação, no Código 3.1 observa-se uma linha de código que representa uma declaração de variável e sua instanciação e na Figura 3.1 observa-se a estrutura de instanciação das metaclasses do KDM quando representam essa linha de código.

Como pode ser observado na Figura 3.1, uma linha de código é representada por um conjunto de metaclasses do KDM que se relacionam. Observando apenas os elementos do KDM separadamente, não é possível identificar uma linha de código específica. Isto é, assumindo que essa linha de código e a representação em KDM sejam de um desvio da Arquitetura, a instância da metaclasse `Calls` é uma violação e a instância da metaclasse `Creates` é outra violação, e o conjunto de todas elas é o desvio arquitetural. Dessa forma, apenas com a instância da metaclasse `Calls`, por exemplo, não é possível identificar características o suficiente que indicam que a instância represente a linha de código inteira.

```
1 ...
2 A variavelA = new A();
3 ...
```

Código 3.1: Código simples em Java

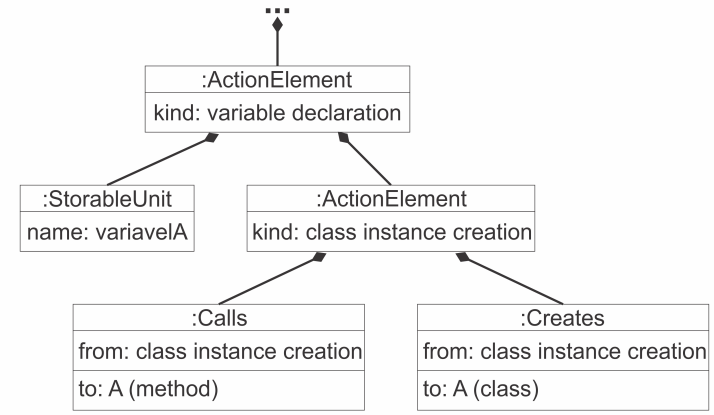

Figura 3.1: Representação do Código 3.1 em KDM. Fonte: Elaborado pelo autor

Como observado no Código 3.1 e na Figura 3.1, em uma comparação e tradução livre, chega-se a conclusão que a instância da metaclasse `Creates` representa o código "`new`" e assim por diante, conforme apresentado na Tabela 3.1.[1]

Quando se trata apenas do processo de CCA no contexto da ADM, essa representação de violações está correta e proporciona uma forma de visualização de problemas arquiteturais do

[1] A tabela apresenta outros elementos do KDM e suas representações em código que foram omitidos da Figura 3.1 por questões de visualização.

Tabela 3.1: Representação direta entre os elementos do Código 3.1 e da Figura 3.1

Código	Metaclasses do KDM
variavelA	StorableUnit(name: variavelA)
A variavelA = new A()	ActionElement(kind: class instance creation)
A()	Calls(from: ActionElement(kind: class instance creation), to: MethodUnit(A))
new	Creates(from: ActionElement(kind: class instance creation), to: ClassUnit(A)
variavelA =	Writes(from: ActionElement(kind: class instance creation), to: StorableUnit(name: variavelA))
import A;	Imports (from: ClassUnit(?), to: ClassUnit(A)

sistema. Entretanto, quando se pretende visualizar os problemas e encontrá-los diretamente no código, chega-se a um impasse de representação e ao problema de identificação comentado anteriormente.

De forma geral, não é possível identificar uma linha de código apenas com informações como "`new`" ou "`A()`", por exemplo. Portanto, essa é a principal motivação que gerou a inclusão dessa separação de conceitos nesta dissertação.

Na abordagem que será apresentada no Capítulo 4, a epata que utiliza essa separação de conceitos denomina-se "Visualização de Desvios Arquiteturais", e tem como objetivo, a realização de uma análise das violações arquiteturais encontradas, agrupando-as em um desvio que possivelmente pode ser visualizado no código e, posteriormente, refatorado e/ou corrigido.

Para auxiliar essa definição e separação de conceitos, definiu-se cada termo e processo da abordagem por teoria dos conjuntos. A Seção 3.3 descreve os termos definidos, a Seção 3.4 descreve a formalização para a CCA e define Desvios Arquiteturais.

3.3 Definição de Termos Utilizados na Formalização

Para definir formalmente o que é um desvio arquitetural baseou-se em dois âmbitos, sendo o primeiro a essência dos relacionamentos citados anteriormente e o segundo o contexto e características da CCA. Dessa forma, definiu-se os termos necessários para a formalização baseando-se nos conceitos de teoria de conjuntos. Primeiramente, foram definidos alguns termos frequentes no processo CCA, sendo eles: Sistema Legado, Código do Sistema, Arquitetura do Sistema e Relacionamentos do Sistema.

O primeiro termo, Sistema Legado, dá origem aos demais, ou seja, o termo Sistema Legado é composto por três itens que representam, respectivamente, o código, a arquitetura e os relacionamentos do sistema. Isto é, um Sistema Legado denominado $SisLeg_{name}$ é uma 3-tupla composta por dois conjuntos e um elemento que é uma 2-tupla. Os conjuntos são Sis_{cod} que representa código do sistema e Sis_{arch} que representa a arquitetura do sistema. A 2-tupla é denominada $Rel_{SisLeg_{name}}$ e representa os relacionamentos do sistema. Esse elemento é composto por dois conjuntos que representam os relacionamentos de código e os relacionamentos de arquitetura. Dessa forma, as Equações 3.1 e 3.2 definem um $SisLeg_{name}$.

$$SisLeg_{name} = (Sis_{cod}, Sis_{arch}, Rel_{SisLeg_{name}}) \; onde \tag{3.1}$$

$$Rel_{SisLeg_{name}} = (Rel_{cod}, Rel_{arch}) \tag{3.2}$$

O conjunto Sis_{cod} é composto pelos elementos de código do sistema, onde cada elemento de código pode ser definido como qualquer tipo de declaração, uso, definição ou linha de código (Meffert, 2006). Ou seja, um elemento de código pode ser qualquer tipo de expressão da linguagem de programação do sistema. Como exemplos dessas expressões tem-se as classes, pacotes, variáveis, chamadas de métodos e implementações de interfaces. Dessa forma, as Equações 3.3 e 3.4 definem o conjunto Sis_{cod}.

$$Sis_{cod} = \{codeElem_i | codeElem_i \in implementação\ do\ sistema\} \; onde \tag{3.3}$$

$$codeElem_i \in \{pacotes,\ classes,\ métodos,\ chamadas\ de\ métodos,...\} \tag{3.4}$$

Para a definição do conjunto Sis_{arch}, utiliza-se o conceito de arquitetura de software definido por Shaw e Garlan (1996). Shaw e Garlan (1996) declara a arquitetura de software como sendo os grandes módulos de um sistema e suas inter-relações. Sendo assim, para esta formalização, define-se um elemento arquitetural como um desses grandes módulos especificados em estilos arquiteturais, por exemplo camadas e componentes. Dessa forma, as Equações 3.5 e 3.6 definem o conjunto Sis_{arch}.

$$Sis_{arch} = \{archElem_i | archElem_i \in arquitetura\ do\ sistema\} \; onde \tag{3.5}$$

$$archElem_i \in \{grandes\ módulos\ do\ sistema\} \tag{3.6}$$

Apesar da definição de elemento arquitetural ser concisa, baseando-se nas suas abstrações, um elemento arquitetural é composto por código. Isto é, a materialização de uma arquitetura e dos elementos arquiteturais são o código do sistema. Sendo assim, ambos os elementos, arquiteturais e de código, possuem uma relação direta, representando o mesmo sistema em níveis de abstração diferentes. Dessa forma, as Equações 3.7, 3.8 e 3.9 definem o relacionamento entre elementos de código e elementos arquiteturais.

$$archElem_i = \{codeElem_i | codeElem_i \in Sis_{cod}\}\ de\ forma\ que \tag{3.7}$$

$$\{\nexists\ codeElem_i | codeElem_i \cap Sis_{cod} = \{\emptyset\}\}\ e \tag{3.8}$$

$$\{\nexists\ codeElem_i | \left(\sum_{n=1}^{m} |codeElem_i \cap archElem_n| \right) \neq 1\} \tag{3.9}$$

A 2-tupla $Rel_{SisLeg_{name}}$, como comentado anteriormente, é composta pelos relacionamentos do sistema. Cada elemento da 2-tupla é um conjunto e representa um tipos de relacionamento, sendo Rel_{cod} representando os relacionamentos de código e Rel_{arch} os relacionamentos arquiteturais. Para o conjunto Rel_{cod}, cada elemento é uma 2-tupla onde é relacionado a origem do relacionamento e o destino do relacionamento. Já para o conjunto Rel_{arch} cada elemento é uma 3-tupla onde é relacionado a origem do relacionamento, o destino do relacionamento e o sua representação em nível de código.

Independentemente do conjunto (Rel_{cod} ou Rel_{arch}), os relacionamentos são tipos definidos a partir do nível de abstração mais baixo encontrados durante a pesquisa e no KDM. Os resultados obtidos durante a pesquisa de tipos foram extensas, totalizando um total de 34 tipos. Esses tipos foram separados em três categorias sendo elas: i) os relacionamentos de código, ou seja, relacionamentos que são facilmente observados diretamente no código de um sistema; ii) os relacionamentos de ação, portanto, relacionamentos que são difíceis de se enxergar diretamente no código do sistema e; iii) os relacionamentos entre os elementos arquiteturais;

As descrições de tipo definido, junto com exemplos podem ser encontrados no Apêndice A. Neste apêndice são encontradas diversas tabelas que explicam e exemplificam essas categorias. Nas Tabelas A.1, A.2 e A.3 são elencados os tipos da primeira categoria. Nas Tabelas A.4, A.5,

A.6 e A.7 são elencados os tipos da segunda categoria e por fim, na Tabela A.8 são elencados os tipos da terceira categoria.

Uma vez apresentada a definição que, por abstração, um elemento arquitetural é composto por elementos de código conclui-se que ambos os conjuntos contém a mesma quantidade de relacionamentos. Sendo assim, no caso do conjunto Rel_{cod} a origem e destino são elementos de código e do conjunto Rel_{arch} a origem e destino são elementos arquiteturais, e possuem as mesmas quantidades. Dessa forma, as Equações 3.10, 3.11 e 3.12 definem o conjunto Rel_{cod} e as Equações 3.13, 3.14 e 3.15 definem o conjunto Rel_{arch}.

$$Rel_{cod} = \{typeRel_i | typeRel_i \in AllRelTypes\} \; onde \tag{3.10}$$

$$AllRelTypes = \{InterfaceImplementation, HasType, ElementCalls, DataReading, ...\} \; e \tag{3.11}$$

$$typeRel_i = (Ori, Dest) \; onde \; Ori \in Sis_{cod} \; e \; Dest \in Sis_{cod} \tag{3.12}$$

$$Rel_{arch} = \{typeRel_i | typeRel_i \in AllRelTypes\} \; onde \tag{3.13}$$

$$AllRelTypes = \{AggregatedRelationship, HasType, ElementCalls, DataReading, ...\} \; e \tag{3.14}$$

$$typeRel_i = (Ori, Dest, typeRel_{Cod}) \; onde \; Ori \in Sis_{arch} \; , \; Dest \in Sis_{arch} \; e \; typeRel_{Cod} \in Rel_{cod} \tag{3.15}$$

Formalizado esses termos base, na seção seguinte é apresentada a formalização do processo de CCA e a definição de um Desvio Arquitetural.

3.4 Formalização da Checagem de Conformidade Arquitetural e Desvios Arquiteturais

Com base nos termos definidos na seção anterior, é possível definir formalmente a CCA utilizada no contexto desta dissertação e, posteriormente, definir formalmente um Desvio Arquitetural. Para a primeira etapa da CCA, a especificação da arquitetura planejada, pode-se utilizar uma variação da definição do termo Rel_{arch} e do termo Sis_{arch}. Ou seja, realizando a substituição do conjunto que contém os elementos reais do sistema, por um conjunto contendo os elementos planejados do sistema, pode-se obter a definição de uma arquitetura planejada. Sendo assim, é possível obter uma instancia de $SisLeg_{name}$ que representa a arquitetura planejada. Dessa forma, as Equações 3.16, 3.17, 3.18 e 3.19 definem um sistema legado quando usado para arquitetura planejada.

$$SisLeg_{namePlan} = (Sis_{codPlan}, Sis_{archPlan}, Rel_{SisLeg_{namePlan}}) \; onde \tag{3.16}$$

$$Sis_{codPlan} = \{\emptyset\} \; e \; Sis_{archPlan} = \{archElemPlan_i | archElemPlan_i \in arquitetura \; planejada\} \; e \tag{3.17}$$

$$Rel_{SisLeg_{namePlan}} = (Rel_{codPlan}, Rel_{archPlan}) \; onde \tag{3.18}$$

$$Rel_{codPlan} = \{\emptyset\} \; e \; Rel_{archPlan} = \{typeRel_i | typeRel_i \in relacionamentos \; planejados\} \tag{3.19}$$

Para a segunda etapa da CCA, a extração da arquitetura atual, é necessário apenas instanciar os termos definidos na seção anterior. Definida a forma de utilização e representação da primeira e segunda etapas da CCA, é possível formalizar a terceira etapa, a comparação de arquiteturas. Como resultado de um processo comum de CCA, tem-se os desvios arquiteturais de um sistema. Entretanto, conforme explicado na Seção 3.2 para o contexto desta dissertação, o resultado do processo identifica as violações arquiteturais de um sistema.

A execução e identificação de violações podem ocorrer utilizando-se de diversas técnicas. No contexto desta dissertação o método utilizado é a diferença de conjuntos, sendo esta realizada entre os conjuntos que representam a Arquitetura Atual e a Arquitetura Planejada. Em termos de teoria de conjuntos este processo se dá pela realização da subtração da Arquitetura

Planejada da Arquitetura Atual. O resultado desta subtração é incluída em um conjunto denominado $SisViolations_{name}$ e definido pela Equação 3.20 .

$$SisViolations_{name} = SisLeg_{name} - SisLeg_{namePlan} \tag{3.20}$$

É importante mencionar que devido ao sistema possuir diversas visões, e nesta formalização diversos conjuntos, para identificar as violações arquiteturais utiliza-se especificamente os conjuntos $Rel_{archPlan}$ de $SisLeg_{namePlan}$ e Rel_{arch} de $SisLeg_{name}$. Dessa forma, as violações arquiteturais são os tipos de relacionamento que não existem na arquitetura planejada, mas existem na arquitetura atual. As Equações 3.21, 3.22, 3.23, 3.24, 3.25 e 3.26 descrevem a subtração destes conjuntos.

$$SisViolations_{name} = Rel_{arch} - Rel_{archPlan}\ onde \tag{3.21}$$

$$SisViolations_{name} = \{typeRel_i, typeRel_n, ...\}\ de\ forma\ que \tag{3.22}$$

$$typeRel_i \in SisViolations_{name} \leftrightarrow \tag{3.23}$$

$$(typeRel_{(Ori,Dest,typeRel_{cod})} \in Rel_{arch}) \notin Rel_{archPlan}\ de\ modo\ que\ a\ comparação\ seja \tag{3.24}$$

$$typeRel_{(Ori_{Rel_{arch}},Dest_{Rel_{arch}},typeRel_{cod,Rel_{arch}})} = typeRel_{(Ori_{Rel_{archPlan}},Dest_{Rel_{archPlan}},typeRel_{cod,Rel_{archPlan}})} \leftrightarrow \tag{3.25}$$

$$Ori_{Rel_{arch}} = Ori_{Rel_{archPlan}}\ e\ Dest_{Rel_{arch}} = Dest_{Rel_{archPlan}} \tag{3.26}$$

Para facilitar o entendimento e criar um exemplo de utilização da formalização, define-se, hipoteticamente, um sistema `S1S` que possui quatro elementos arquiteturais (`layer l1`, `layer l2`, `layer l3` e `componente c1`). Em termos de código, esse sistema também possui 4 elementos (`class c1`, `class c2`, `class c3` e `interface i1`) e inúmeros relacionamentos entre esses elementos. Sua arquitetura planejada, possui os mesmos quatro elementos arquiteturais e algumas regras de comunicação entre eles. Representando graficamente esse sistema hipotético

tem-se a Figura 3.2 para o $SisLeg_{namePlann}$ e a Figura 3.3 para o $SisLeg_{name}$.

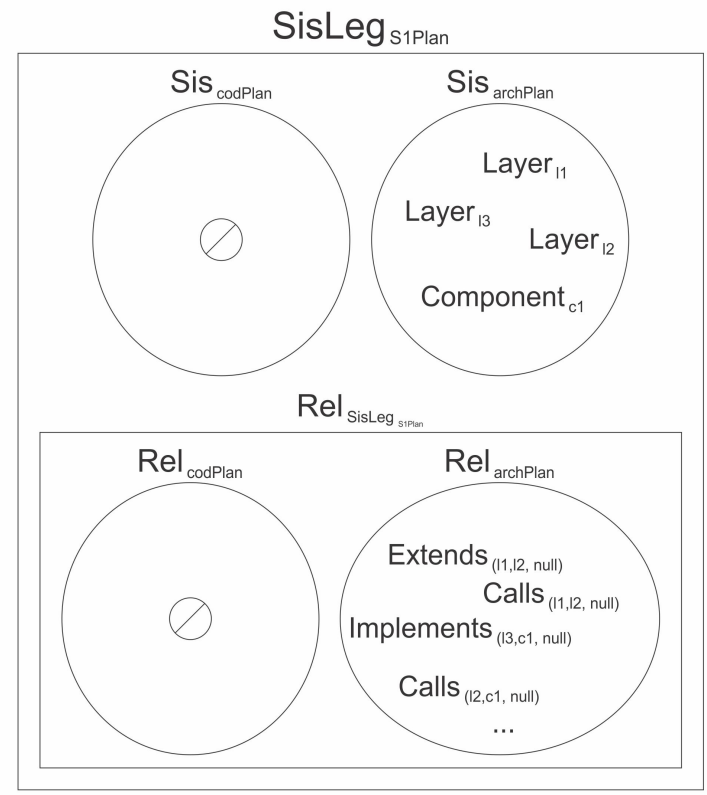

Figura 3.2: Exemplo gráfico da arquitetura planejada. Fonte: Elaborado pelo autor

Com ambas as definições é possível obter-se o conjunto $SisViolations_{name}$ a partir da subtração dos conjuntos de relacionamentos. A Figura 3.4 representa graficamente essa subtração. Como pode ser observado, o conjunto resultante possui apenas os relacionamentos que representam violações arquiteturais.

Realizada a subtração, o conjunto $SisViolations_{name}$ contém as violações identificadas, porém não contém os desvios arquiteturais. Como comentado anteriormente, apenas com as violações que foram identificadas, não é possível identificar em termos de código qual é o problema arquitetural, sendo assim, não é possível continuar o processo Reconciliação Arquitetural identificando no código e, consequentemente, resolvendo os problemas arquiteturais. Sendo assim, necessitou-se de uma definição que fosse capaz de melhorar a representação, dessa forma, definiu-se que agrupando as violações e suas informações era possível representar e identificar os problemas arquiteturais.

Portanto, verificou-se que observando qualquer tipo de contexto, as violações se relacionam. Um exemplo é o contexto de causa e efeito, nos quais as violações arquiteturais em sua forma simples podem estar relacionadas umas com as outras. Portanto, uma violação do tipo $Imports_{(Ori.Dest.Call_{cod})}$, por exemplo, pode estar relacionada a uma do tipo $Extends_{(Ori.Dest.Call_{cod})}$,

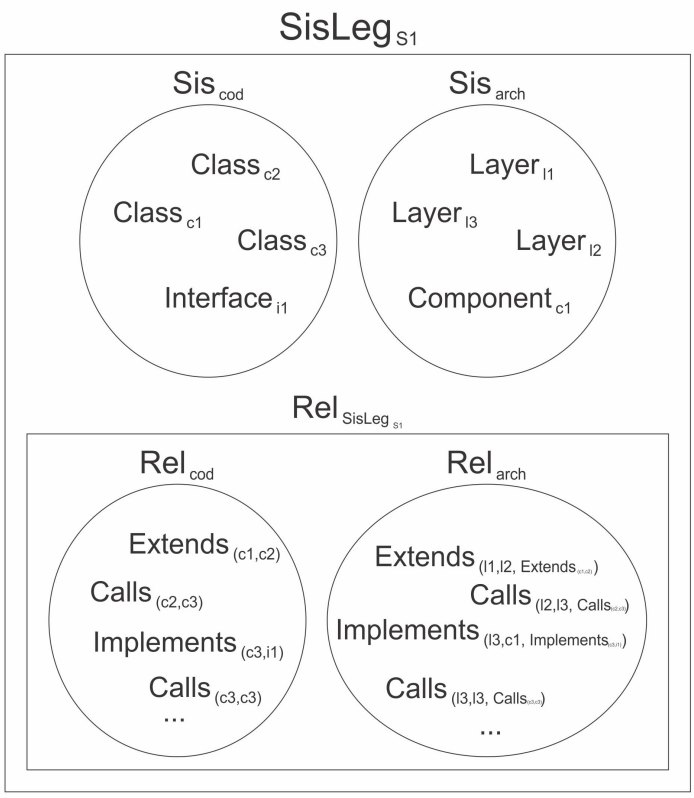

Figura 3.3: Exemplo gráfico da arquitetura atual. Fonte: Elaborado pelo autor

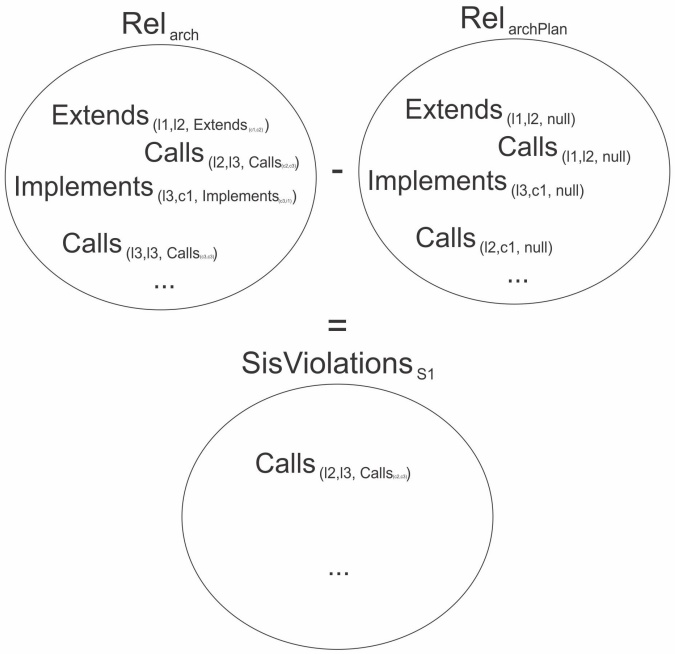

Figura 3.4: Exemplo gráfico da CCA. Fonte: Elaborado pelo autor

no qual ambas representam uma herança de classes.

Aprofundando-se na pesquisa, foram identificadas três formas distintas que as violações

podem ser relacionadas. A primeira forma é o relacionamento mútuo, onde uma violação se relaciona diretamente a outra e vice-versa. A segunda é o relacionamento unidirecional simples, onde uma violação se relaciona apenas a uma única outra. E por fim o relacionamento de um para muitos, onde uma violação se relaciona com diversas outras violações mas o oposto pode não ser verdade.

A partir deste princípio, definiu-se o desvio arquitetural como sendo um conjunto composto por violações arquiteturais que se relacionam. Com essa definição, um sistema passaria a ter Desvios Arquiteturais e não apenas mais Violações Arquiteturais. Dessa forma, o conjunto $ArchDrifts_{name}$ foi definido e é composto por um ou mais conjuntos $Drifts$ que por sua vez são compostos por uma ou mais violações arquiteturais identificadas no conjunto $SisViolations_{name}$. As Equações 3.27, 3.28, 3.29, 3.30 caracterizam essa definição.

$$ArchDrifts_{name} = \{drift_i, drift_n, ...\}\ onde \tag{3.27}$$

$$drift_i = \{typeRel_{(Ori,Dest,typeRel_{cod})} | typeRel_{(Ori,Dest,typeRel_{cod})} \in SisViolations_{name}\}\ de\ modo\ que \tag{3.28}$$

$$\{\nexists\, typeRel_{(Ori,Dest,typeRel_{cod})} | typeRel_{(Ori,Dest,typeRel_{cod})} \cap SisViolations_{name} = \{\emptyset\}\}\ e \tag{3.29}$$

$$\{\nexists\, typeRel_{(Ori,Dest,typeRel_{cod})} | \left(\sum_{n=1}^{m} |typeRel_{(Ori,Dest,typeRel_{cod})} \cap drift_n| \right) \neq 1\} \tag{3.30}$$

3.5 Considerações Finais

Neste capítulo foi apresentada uma importante divisão e separação de termos que geralmente na literatura são denominados como sinônimos. Essa separação é importante no contexto deste trabalho devido à forma como o KDM trabalha e representa os elementos de código de um sistema. Depois de separado e exemplificado os termos, apresentou-se uma formalização de termos baseado em teoria de conjuntos para auxiliar a formalização do processo de CCA e do Desvio Arquitetural. Por fim, apresentou-se uma formalização do processo de CCA e a formalização de Desvios Arquiteturais para corroborar a decisão de separação dos termos

Violação e Desvio, bem como ter uma base teórica para a implementação do apoio computacional apresentado no capítulo seguinte em conjunto com a abordagem.

Capítulo 4

ARCH-KDM 2.0: CHECAGEM DE CONFORMIDADE ARQUITETURAL NO CONTEXTO DA ADM

4.1 Considerações Iniciais

Neste capítulo é apresentada a abordagem e a metodologia para a realização da checagem de conformidade arquitetural no contexto da ADM. Na Seção 4.2, é apresentada a visão geral da abordagem Arch-KDM 2.0. Na Seção 4.3 são apresentadas as etapas da abordagem e as evoluções realizadas em cada uma de forma sucinta. Por fim, na Seção 4.4, as considerações finais sobre este capítulo são apresentadas.

4.2 As Etapas da Abordagem Arch-KDM 2.0

A Arch-KDM 2.0 é uma abordagem iniciada em um trabalho de mestrado anterior (Chagas, 2016), continuada pelo autor desta dissertação e realizada no laboratório de pesquisa AdvanSE (*Advanced Research on Software Engineering*), da Universidade Federal de São Carlos (UFSCar).

Essa abordagem consiste na CCA no contexto da ADM e com o auxílio do KDM. O principal objetivo dessa abordagem é identificar violações arquiteturais entre um modelo de arquitetura planejada e outro modelo que representa a arquitetura atual de um determinado sistema, ambos representados como instâncias do KDM. Essa abordagem foi totalmente desenvolvida utilizando os conceitos da ADM e o KDM foi utilizado em sua forma original, sem modificações ou extensões.

A Figura 4.1 apresenta a atual abordagem Arch-KDM 2.0. Como pode ser observado, ela é composta por quatro etapas, sendo que as três primeiras já existiam na abordagem, ou seja, a etapa IV foi adicionada neste projeto de mestrado. A Etapa I é denominada de "Especificação da Arquitetura Planejada". A execução dessa etapa é apoiada por uma DSL (*Domain-Specific Language*) e um algoritmo. A DSL é denominada DCL-KDM (Landi et al., 2017) e é a responsável por definir uma especificação de arquitetura em alto nível. O algoritmo é denominado "PASerializer-KDM" e é o responsável por realizar a leitura da arquitetura especificada e transformá-la em uma instância do metamodelo KDM.

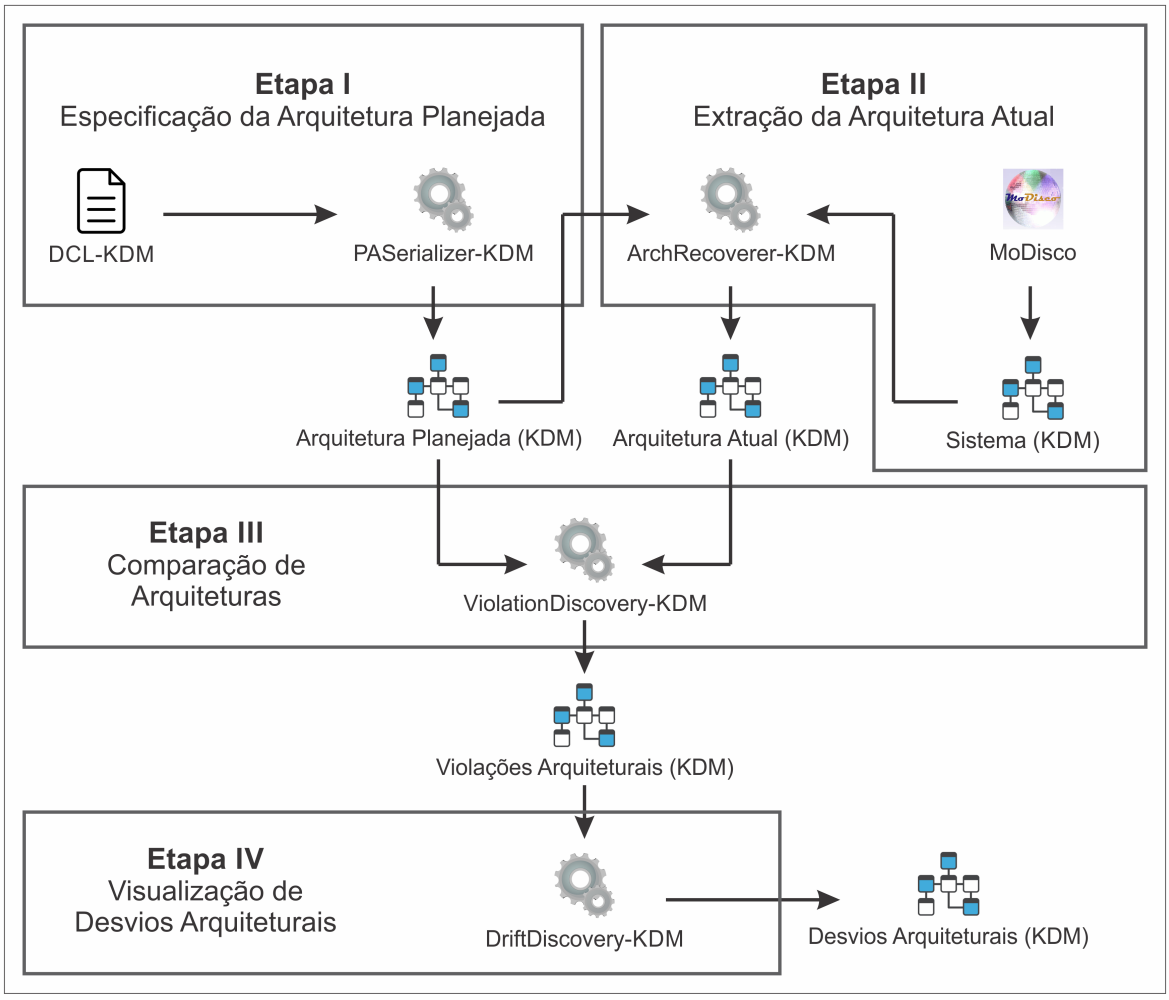

Figura 4.1: Abordagem Arch-KDM 2.0. Fonte: Elaborado pelo autor

A Etapa II é denominada "Extração da Arquitetura Atual", sua execução depende da ferramenta *MoDisco* (Bruneliere et al., 2010; Brunelière et al., 2014) e de um um algoritmo evoluído / melhorado no contexto deste trabalho chamado de "ArchRecoverer-KDM". Esse algoritmo recebe duas entradas; a instância do KDM gerada pelo *MoDisco* e a instância KDM gerada na Etapa I. Com base nessas duas entradas, o algoritmo é capaz de realizar um mapeamento entre o sistema real e sua arquitetura planejada e gerar uma nova instância do KDM que representa a

arquitetura atual do sistema.

A Etapa III é denominada de "Comparação de Arquiteturas". A execução dessa etapa é apoiada por um algoritmo. O algoritmo denominado "ViolationDiscovery-KDM" recebe como entrada duas instâncias do KDM, que representam a arquitetura atual e a arquitetura planejada, realiza o processo de CCA e gera uma instância do KDM que representa as violações arquiteturais do sistema.

A Etapa IV é denominada de "Visualização de Desvios Arquiteturais". A execução dessa etapa é apoiada por dois algoritmos, um implementado pelo autor deste trabalho e outro por Gasparini (2018). Entretanto, essa etapa provê suporte a extensão e acoplamento de novos algoritmos que realizem o descobrimento de desvios arquiteturais. O algoritmo implementado pelo autor deste projeto é denominado "DriftDiscovery-KDM" e é baseado na combinação de Matriz de Proximidade e a clusterização pelo algoritmo DBSCAN (Bouckaert et al., 2010; Garner, 1995; Borah; Bhattacharyya, 2004; Ester et al., 1996). O algoritmo recebe como entrada as violações arquiteturais encontradas na etapa anterior, realiza um processamento de agrupamento e disponibiliza seus resultados. É importante salientar que, no contexto desta dissertação, existem diferenças teóricas entre desvio arquitetural e violação arquitetural. A explicação e definição de cada termo serão apresentadas na Seção 4.3.4.

A evolução da abordagem e o apoio ferramental Arch-KDM 2.0, desenvolvidos neste trabalho, foram obtidos após várias alterações e evoluções no primeiro trabalho apresentado por Chagas (2016). Cada etapa da abordagem original foi analisada e oportunidades de melhoria foram encontradas. Nas Tabelas 4.1, 4.3 e 4.4 das seções seguintes são listadas as limitações encontradas e uma breve descrição de cada uma. Como envolvem vários detalhes técnicos, as explicações são detalhadas nos Apêndices A, B e C.

Além das evoluções e melhorias, realizou-se também a extensão da abordagem adicionando uma nova etapa. A motivação para a inclusão dessa nova etapa gira em torno de peculiaridades do KDM e também da opção de incluí-la no contexto da Reconciliação Arquitetural (RA). Maiores detalhes sobre essas motivações, necessidades e suas explicações são descritas na Seção 4.3.4.

Na Figura 4.2 pode ser observada esquematicamente a arquitetura do apoio ferramental original da abordagem antes deste trabalho ser iniciado (Item A), e após sua conclusão com o apoio ferramental refeito (Item B). A figura apresenta uma visão lógica distribuída em duas camadas, a primeira camada constitui-se no ambiente de desenvolvimento Eclipse com as ferramentas que foram empregadas no desenvolvimento do apoio computacional da abordagem. A segunda camada constitui-se da estrutura organizacional do apoio computacional.

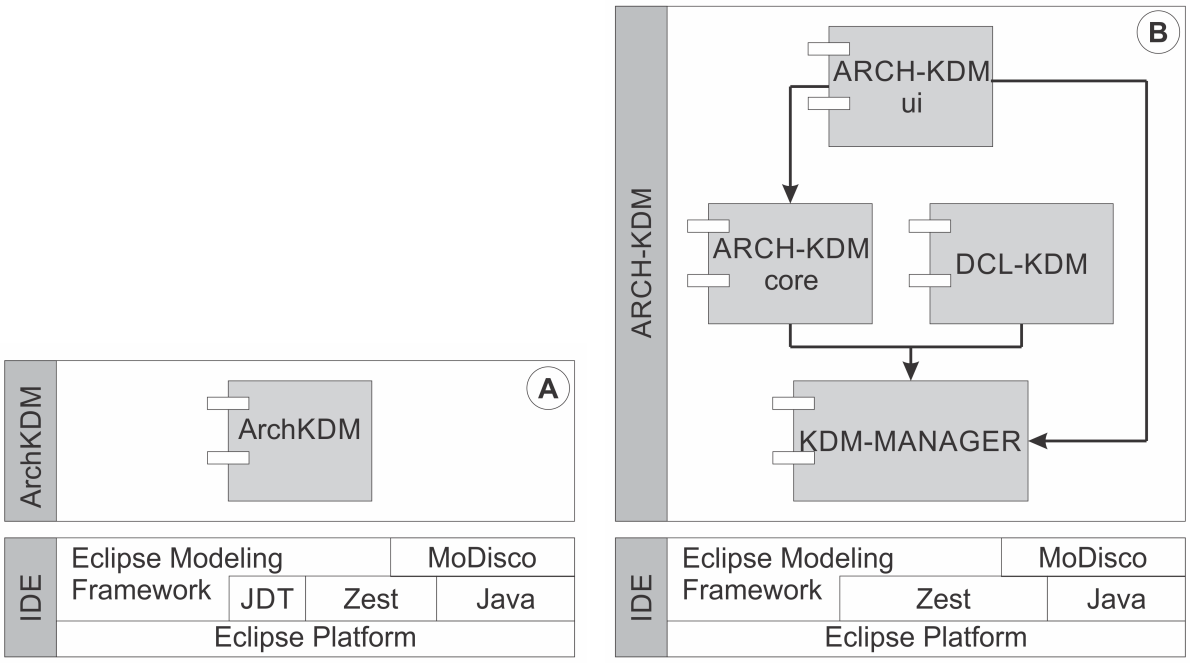

Figura 4.2: Arquitetura esquemática da Arch-KDM 1.0 (Item A) e 2.0 (Item B). Fonte: Elaborado pelo autor

Nas próximas seções, cada uma das quatro etapas da Arch-KDM são detalhadas. Em cada seção encontra-se uma tabela que apresenta as evoluções que foram realizadas na versão anterior da Arch-KDM para que a versão 2.0 fosse desenvolvida.

4.3 Visão Geral das Etapas e Evoluções Realizadas

Nesta seção serão apresentadas as quatro etapas da abordagem e suas evoluções. Na Subseção 4.3.1 serão apresentadas as evoluções e a etapa de especificação da arquitetura planejada. Na Subseção 4.3.2 serão apresentadas as evoluções e a etapa de extração da arquitetura atual de um sistema. Na Subseção 4.3.3 serão apresentadas as evoluções e a etapa da realização da CCA. Por fim, na Subseção 4.3.4 será apresentada a etapa de exibição/visualização dos desvios arquiteturais identificados.

4.3.1 Especificação da Arquitetura Planejada com a DCL-KDM

Com o intuito de realizar a especificação de uma arquitetura planejada para um sistema, essa etapa é composta por duas partes principais: i) uma *Architecture Description Language* (ADL) chamada DCL-KDM ii) um algoritmo de serialização da arquitetura como uma instância do KDM denominado *PASerializer-KDM*.

A criação da DCL-KDM foi motivada por dois objetivos: i) ter uma ADL que desse suporte a especificação de arquiteturas no contexto da ADM; e ii) ter uma ADL que também pudesse ser utilizada em outros contextos que não o da ADM com poucas adaptações (Landi et al., 2017).

A DCL-KDM é uma extensão de outra ADL da literatura denominada DCL, proposta inicialmente por Terra e Valente (2009). A DCL foi estendida para contemplar estilos arquiteturais e para instanciar a arquitetura planejada de sistemas no formato KDM. Portanto, o utilizador passa a ter a segurança de que sua especificação segue as regras impostas para cada estilo arquitetural.

A DCL-KDM faz uso de duas técnicas para que haja a minimização de erros, que são: (i) a utilização de uma sintaxe própria da linguagem, que obriga o arquiteto a definir os elementos de acordo com a mesma; e (ii) o uso do x-text[1], um *Framework* para o desenvolvimento de linguagens de programação, que obriga o usuário a realizar a especificação na ordem exata em que cada termo da DCL-KDM deve ser escrita por meio de sua gramática.

Na Tabela 4.1 podem ser observadas as evoluções realizadas nessa etapa, bem como uma breve descrição. Como envolve vários detalhes técnicos, as explicações são detalhadas no Apêndice B.

No intuito de exemplificar melhor a aplicação da DCL-KDM após as evoluções, a Figura 4.3 representa uma arquitetura planejada hipotética. Nesta figura, estão representados elementos arquiteturais e as regras de acesso entre eles. Cada elemento da figura nomeado com o padrão `[tipo de elemento] : [nome do elemento]` representa um elemento arquitetural distinto. As setas entre esses elemento representam as regras de acesso entre eles.

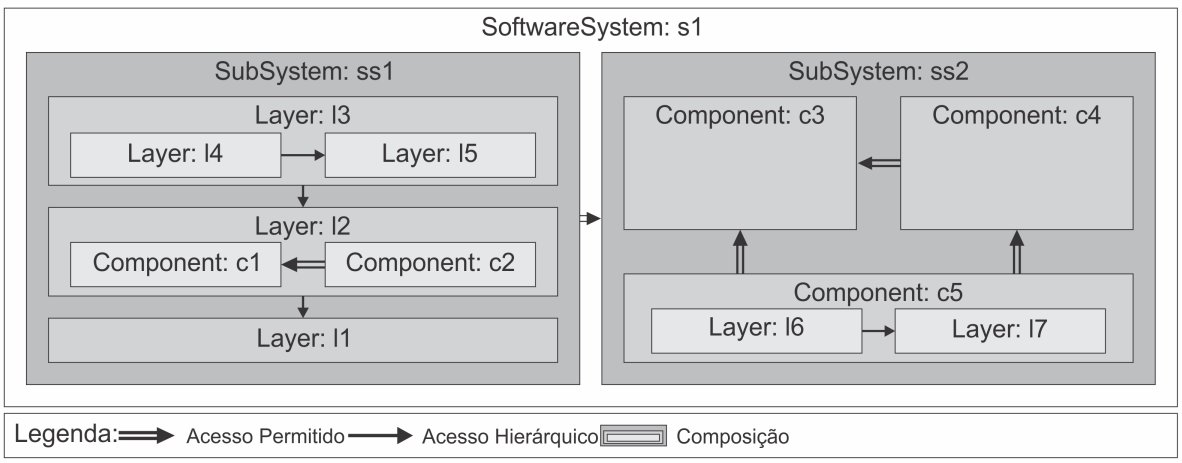

Figura 4.3: Exemplo de arquitetura planejada. Fonte: Elaborado pelo autor

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

Tabela 4.1: Evoluções realizadas na "Etapa I - Especificação da Arquitetura Planejada"

Limitação	Descrição / Solução
Inconsistência semântica	**Descrição:** Existiam inconsistências semânticas entre os termos hierarquia e composição, tornando o processo de serialização da Arquitetura Planejada inconsistente. **Solução:** Separou-se logicamente e semanticamente os conceitos de hierarquia e composição de elementos.
Carência de restrições	**Descrição:** Existiam restrições arquiteturais que não eram contempladas com a implementação inicial. Um exemplo são restrições do tipo "can" para indicar acesso entre mais de um elemento sem o modificador "only". **Solução:** Atualização da gramática e implementação desse tipo de restrição para a DCL-KDM.
Carência de metaclasses	**Descrição:** Existia um subconjunto fixo de 8 metaclasses do KDM (`Calls`, `UsesType`, `Creates`, `Extends`, `Implements`, `HasValue`, `Imports`, `HasType`) que eram utilizados para a geração da arquitetura planejada, sendo que este subconjunto era difícil de ser alterado na implementação original. **Solução:** Ao se reestruturar a implementação parametrizou-se e implementou-se todas as demais metaclasses pertinentes a etapa.
Implementação complexa	**Descrição:** A versão inicial foi implementada sem seguir boas práticas e, consequentemente, possuía alta complexidade de entendimento. **Solução:** Reimplementação de toda a etapa seguindo boas práticas de programação, acrescentando um fluxo de controle e padrões de projeto.

Quanto aos elementos arquiteturais, por exemplo, pode-se observar que se tem um sistema de software `s1` representado pelo retângulo maior, que é composto por dois subsistemas (`ss1` e `ss2`) representados por retângulos menores. Cada subsistema por sua vez é composto por componentes/*layers*, sendo cada componente/*layer* composto por outros componentes/*layers*.

Quanto às regras de acesso entre os elementos, tem-se definido três tipos de representações distintas. A primeira são as setas simples que representam as regras de acesso entre as *layers*. Esse tipo de seta representa o nível hierárquico entre as *layers*, de modo que fique evidente e fácil de se gerar as regras de acesso automaticamente. A segunda são as setas duplas que representam as regras de acesso entre quaisquer elementos. Por exemplo, na figura pode-se dizer que o elemento `c2` pode acessar o elemento `c1`. A terceira é a representação de composição entre os elementos, ou seja, a composição é representada pelo agrupamento e posicionamento entre os elementos representados por quadrados dentro uns dos outros.

Nesse tipo de notação utilizado, tem-se diferença entre a presença ou não das setas entre

os elementos. Isto é, a presença das setas representa a permissão e a ausência das setas a proibição de comunicação entre os elementos. Sendo assim, um exemplo é a permissão entre a comunicação entre o elemento `c4` e `c3`, porém a proibição da comunicação entre o elemento `c3` e `c4`, visto que não existem setas nesse sentido.

A arquitetura exemplificada na Figura 4.3 em DCL-KDM é descrita conforme o Código 4.1. Uma especificação arquitetural em DCL-KDM é dividida em dois blocos, o primeiro bloco contendo os elementos arquiteturais e o segundo bloco contendo as restrições arquiteturais. No primeiro bloco, denominado `architecturalElements`, são descritos todos os elementos arquiteturais que representam a arquitetura planejada e sua composição. Por exemplo, na linha 4 do Código 4.1 tem-se a especificação da `Layer:l3` da Figura 4.3. Nessa linha pode-se observar a declaração do elemento, a definição de seu nível hierárquico e de seu elemento contêiner. Nesse exemplo, essa `layer` possui o maior nível hierárquico da especificação, pois quanto maior o número de seu `level`, mais alta ela é na hierarquia.

```
1 architeturalElements {
2     subSystem ss1;
3     subSystem ss2;
4     layer l3, level 3, inSubSystem: ss1;
5     layer l2, level 2, inSubSystem: ss1;
6     layer l1, level 1, inSubSystem: ss1;
7     layer l4, level 2, inLayer: l3;
8     layer l5, level 1, inLayer: l3;
9     layer l6, level 2, inComponent: c5;
10    layer l7, level 1, inComponent: c5;
11    component c1, inLayer: l2;
12    component c2, inLayer: l2;
13    component c3, inSubSystem: ss2;
14    component c4, inSubSystem: ss2;
15    component c5, inSubSystem: ss2;
16 }restrictions {
17    c2 can-depend-only c1;
18    c4 can-depend-only c3;
19    c5 can-depend c4;
20    c5 can-depend c3;
21    ss1 can-depend ss2;
22 }
```

Código 4.1: Arquitetura exemplo da Figura 4.3 em DCL-KDM

No segundo bloco denominado `restrictions` são especificadas todas as regras de acesso que a descrição arquitetural possui. Isto é, neste bloco são descritas todas as regras definidas na arquitetura e representadas na Figura 4.3 como setas duplas. Por exemplo, na linha 19 do Código 4.1 tem-se a declaração da seta dupla entre os elementos `c5` e `c4`, representando que o primeiro elemento pode acessar/depender do segundo elemento.

De modo geral, a sintaxe da DCL-KDM é simples e dividida em dois padrões, o padrão dos elementos arquiteturais e o padrão das restrições arquiteturais. A especificação de elementos arquiteturais constitui-se de quatro palavras-chave para descrever elementos arquiteturais e duas para representar hierarquia e composição automáticas. Essas palavras são `layer`, `component`, `subsystem` e `module` para elementos arquiteturais e palavras com radical `in` e `level` para automatização de hierarquia e composição. A seguir tem-se uma explicação das palavras `layer`, `component` e `subsystem` utilizadas no exemplo da Figura 4.3 e Código 4.1.

A palavra-chave `layer` representa uma camada na arquitetura de um sistema. No Código 4.1 entre as linhas 4 e 10 tem-se alguns exemplos de especificação de camadas. A expressão básica de especificação de uma camada é dividida em duas partes obrigatórias e uma opcional. As partes obrigatórias são o uso da palavra-chave `layer`, seguida do seu nome e a segunda parte é o uso da palavra-chave `level` seguida de um número que represente sua hierarquia. Comparando-se ao exemplo dado tem-se na linha 4 "`layer l3`" para a primeira parte, seguido de virgula e "`level 3`" para a segunda parte. A parte opcional da especificação entra na especificação de composições onde pode-se ou não especificar que a camada está em certa posição da arquitetura utilizando-se as palavras-chave com radical `in`. Comparando-se ao exemplo tem-se na linha 4 "`inSubSystem ss1`" para representar que a camada `l3` é parte da composição do subsistema `ss1`.

A palavra-chave `component` representa um componente na arquitetura do sistema. No Código 4.1 entre as linhas 11 e 15 tem-se alguns exemplos de especificação de componentes. A expressão básica de especificação de um componente é dividida em uma parte obrigatória e outra opcional. A parte obrigatório é o uso da palavra-chave `component`, seguida do nome do componente. Comparando-se ao exemplo dado, tem-se na linha 12 "`component c2`". A parte opcional assim como para camadas é o posicionamento do componente na composição da arquitetura pelas palavras-chave com o radical `in`. No exemplo, tem-se "`inLayer l2`" para representar que o componente faz parte da composição da camada `l2`.

Por fim, tem-se a palavra-chave `subSystem` representando um subsistema de software. No código 4.1 entre as linhas 2 e 3 tem-se dois exemplos de especificação de um subsistema. A expressão para especificação do subsistema é dividida em uma parte obrigatória e outra opcional e segue o mesmo padrão de componentes. A parte obrigatória sendo o uso da palavra-chave `subSystem` e a opcional o posicionamento na composição de elementos.

O segundo padrão de sintaxe são as restrições arquiteturais. A especificação de restrições arquiteturais constitui-se de dois conjuntos de palavras concatenadas cada qual com seus significados e uma palavra especifica (`only`) que auxilia uma palavra (`can`).

O primeiro grupo de palavras-chave são `can`, `cannot` e `must`. Esse conjunto de palavras representa a acessibilidade entre os elementos arquiteturais. A palavra-chave `can` representa acessibilidades que podem acontecer, ou seja, um elemento pode acessar outro elemento. A palavra-chave `cannot` representa acessibilidades que não podem acontecer, ou seja, um elemento não pode acessar outro elemento. Por fim, a palavra-chave `must` representa acessibilidades que devem acontecer, ou seja, um elemento deve acessar outro elemento.

O segundo grupo de palavras-chave estão descritas na Tabela 4.2 e representam a tipificação da acessibilidade. Essas palavras-chave são utilizadas em sequência às do primeiro grupo e determinam um tipo específico de acessibilidade que deve ocorrer.

Tabela 4.2: Palavras-chave que tipificam o tipo de acesso

Palavra-Chave	Descrição
access	Acesso de métodos, atributos e classes
declare	Declaração de variáveis
handle	Acesso e declaração de métodos e variáveis
create	Criação de objetos
extend	Extensão de classes e interfaces
implement	Implementação de classes e interfaces
derive	Extensão e implementação de classes e interfaces
throw	Lançamento de exceções
useannotation	Uso de anotações de código
depend	Todos os tipos descritos anteriormente

Esses dois grupos de palavras-chave são utilizados para a definição de restrições entre os elementos arquiteturais. Atualmente a DCL-KDM possui três formas distintas de declaração dessas restrições. Duas delas podem ser observadas no Código 4.1 nas linhas 18 e 19. A terceira possibilidade é oposta a restrição exemplificada na linha 18, no qual a palavra-chave `only` é utilizada antes da palavra-chave `can`, dessa forma, a restrição ficaria igual a "`only c4 can-depend c3`". Traduzindo ambas as regras para português, visando diferenciar o uso da palavra-chave `only` tem-se para o exemplo na linha 18 a frase "o elemento c4 pode depender apenas do elemento c3", já para o segundo exemplo citado tem-se a frase "somente o elemento c4 pode depender do elemento c3".

De modo geral, apesar dessa diferenciação do uso da palavra-chave `only`, a especificação das restrições é dividida em três partes obrigatórias. A primeira parte obrigatória, como exemplificado anteriormente, pode ser especificada de duas formas, sendo a primeira a especificação

direta do elemento arquitetural origem da restrição e a segunda sendo o uso da palavra-chave `only` seguido do elemento origem da restrição. Um exemplo para esta primeira parte é a declaração `c2` na linha 17 do Código 4.1.

A segunda parte obrigatória é a especificação do tipo de acesso e da acessibilidade que ocorrerá entre os elementos. Para essa especificação deve-se utilizar os grupos de palavras-chave apresentados anteriormente sendo três possibilidades de acesso (`can`, `cannot` e `must`) e as dez possibilidades de acessibilidade apresentadas na Tabela 4.2. Um exemplo para esta especificação é a expressão `can-depend` nas linhas 19, 20 e 21 do Código 4.1. Por fim, a última parte obrigatória é a especificação do elemento arquitetural alvo da restrição arquitetural. Essa parte é apenas a definição do elemento, como por exemplo `ss2` na linha 21 do Código 4.1.

Uma vez que planejou-se uma arquitetura com a DCL-KDM, é necessário transformar a especificação em uma instância do KDM. Este processo se faz necessário para que essa especificação possa ser utilizada nas etapas posteriores do processo de Checagem de Conformidade Arquitetural. Para tanto, criou-se um algoritmo de conversão entre a especificação em DCL-KDM e o KDM, metamodelo base da abordagem.

Na Figura 4.4 encontra-se um exemplo gráfico da instância do KDM gerado pelo algoritmo. No Código 4.2 encontra-se o mesmo exemplo reduzido, visto que a instância gerada é um arquivo XMI com aproximadamente 350 linhas.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <kdm:Segment xmi:version="2.0" [...] name="Planned Architecture">
3   <model xsi:type="structure:StructureModel" name="Planned Architecture">
4     <structureElement xsi:type="structure:Subsystem" name="ss1">
5       <structureElement xsi:type="structure:Layer" name="l3"  [...]>
6         <aggregated from="//@model.0/@structureElement.0/@structureElement.0"
7             to="//@model.0/@structureElement.0/@structureElement.1"
8             relation="//@model.1/@codeElement.0/@codeElement.0/@actionRelation.0[...]"
9             density="7"/>
10        [...]
11      </structureElement>
12      <structureElement xsi:type="structure:Layer" name="l2" [...]>
13        <aggregated from="//@model.0/@structureElement.0/@structureElement.1"
14            to="//@model.0/@structureElement.0/@structureElement.2"
15            relation="//@model.1/@codeElement.1/@codeElement.0/@actionRelation.0[...]"
16            density="7"/>
17        [...]
18      </structureElement>
19      <structureElement xsi:type="structure:Layer" name="l1" [...]/>
20    </structureElement>
21  </model>
22  <model xsi:type="code:CodeModel"> [...] </model>
23 </kdm:Segment>
```

Código 4.2: Parte da instância do KDM gerada pelo algoritmo

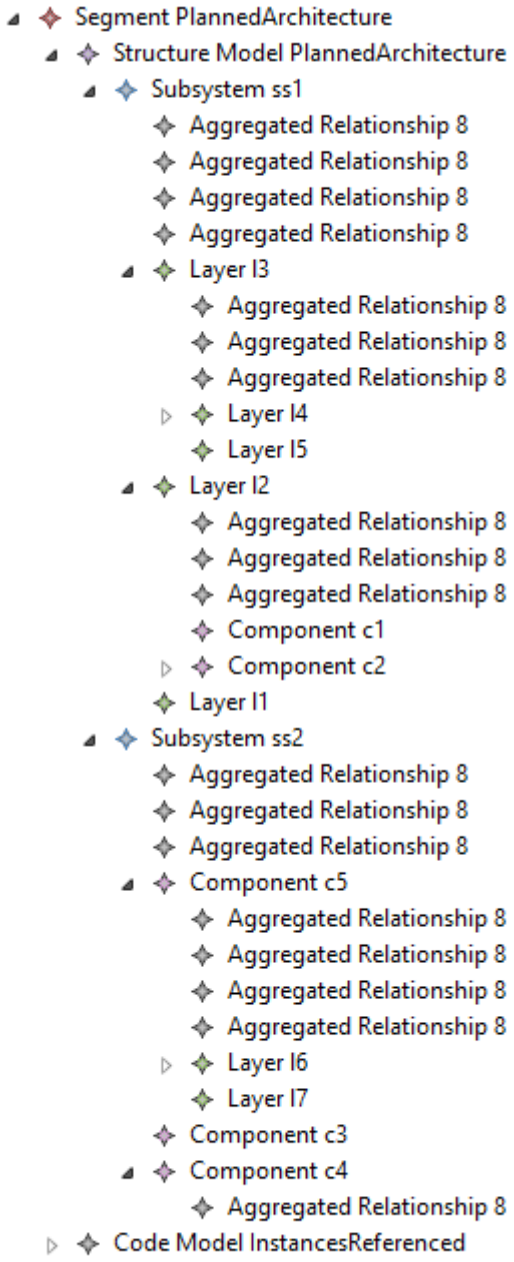

Figura 4.4: Instância do KDM que representa a arquitetura planejada. Fonte: Elaborado pelo autor

Esse código possui alguns exemplos das metaclasses do pacote *Structure*. Nas linhas 4, 5, 12 e 19 pode-se observar os elementos arquiteturais `ss1` (subsistema, metaclasse `Subsystem`), `l3`, `l2` e `l1` (camadas, metaclasse `Layer`). Já as restrições arquiteturais são representadas pela metaclasse `AggregatedRelationship`, que pode ser observada nas linhas 6 e 13. Realizando uma comparação aos elementos obrigatórios de uma restrição arquitetural definida pela DCL-KDM tem-se a direcionalidade e elementos da restrição representadas pelos atributos `from` e `to`, os tipos de acessibilidade representados pelo atributo `relation` e por fim o tipo de acesso representado pela existência ou ausência da metaclasse no elemento arquitetural.

4.3.2 Extração da Arquitetura Atual

Com o intuito de obter uma especificação que represente a arquitetura atual do sistema, essa etapa possui dois passos que são realizados internamente e que são invisíveis ao usuário, que são: i) recuperação de uma instância do KDM que representa o software em análise e; ii) um algoritmo de mapeamento para a instância gerada.

A primeira parte é auxiliada por uma ferramenta já desenvolvida apresentada no Capítulo 2 Seção 2.7 e denominada *MoDisco*. O *MoDisco* realiza a extração e mineração de informações de um sistema escrito na linguagem de programação Java transformando essa informação em uma instância do KDM. Na Figura 4.5 tem-se um exemplo gráfico de instância do KDM gerado por esta ferramenta para um sistema exemplo fictício utilizado para testes do desenvolvimento da abordagem.

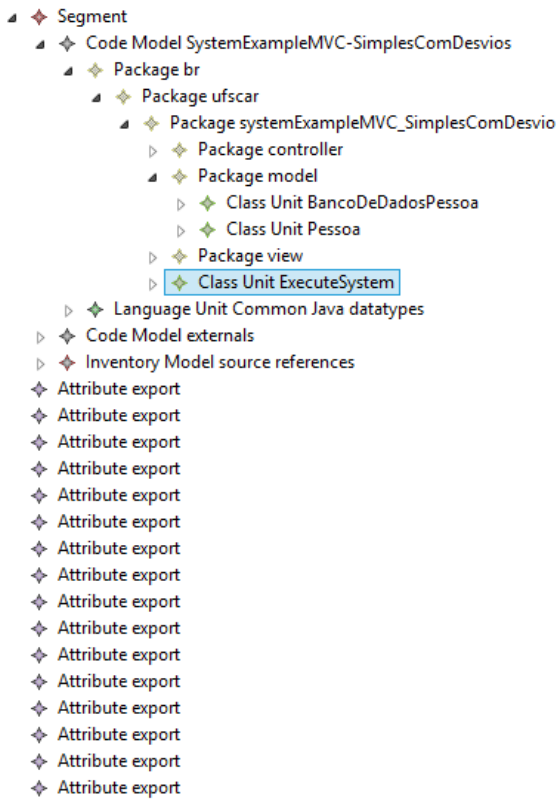

Figura 4.5: Exemplo instância KDM gerado pelo MoDisco. Fonte: Elaborado pelo autor

Conforme citado anteriormente, o arquivo que representa a instância do KDM é um arquivo de texto com a extensão XMI e com tamanho relativo ao seu conteúdo, neste exemplo o arquivo contém 5360 linhas. Observa-se na Figura 4.5 que o exemplo possui 3 pacotes principais (`model`, `controller` e `view`) em uma hierarquia de pacotes iniciada com um radical (`br`, `ufscar`, `systemExampleMVC_SimplesComDesvio`)

A segunda parte desta etapa é a utilização de um algoritmo que realiza um mapeamento entre a instância gerada do sistema e a arquitetura planejada, que é obtida por meio do uso/instanciação da DCL-KDM na etapa anterior. Este mapeamento é realizado entre os elementos arquiteturais definidos na instância do Pacote *Structure* do KDM, que representa a arquitetura planejada, e, os elementos na instância do Pacote *Code* do KDM que representa a arquitetura atual do sistema. Na Tabela 4.3 podem ser observadas as evoluções realizadas nesta etapa, principalmente neste algoritmo, bem como uma breve descrição. Como envolvem vários detalhes técnicos, as explicações são detalhadas no Apêndice C.

Tabela 4.3: Evoluções realizadas na "Etapa II - Extração da Arquitetura Atual"

Limitação	Descrição / Solução
Carência de elementos	**Descrição:** A implementação original só reconhecia dois elementos de código (`Class` e `Package`), dessa forma, nem todas as metaclasses do KDM que representavam elementos de código eram usadas nesta etapa. **Solução:** Reimplementação e inclusão do reconhecimento dos elementos de código `Enumaration` e `Interface`.
Carência de metaclasses	**Descrição:** Existia um subconjunto fixo de 8 metaclasses do KDM (`Calls`, `UsesType`, `Creates`, `Extends`, `Implements`, `HasValue`, `Imports`, `HasType`) que eram utilizados para a realização do processo de mapeamento da arquitetura atual, sendo que este subconjunto limitava o processo posteriormente. **Solução:** Reimplementação dos algoritmos utilizando todas as metaclasses do KDM pertinentes a etapa.
Implementação com falha	**Descrição:** Existiam falhas de execução dos algoritmos desta etapa para padrões específicos de entradas. A implementação dos algoritmos além de possibilitar falhas era altamente complexa. **Solução:** Reimplementação dos algoritmos diminuindo sua complexidade e correção das falhas encontradas.
Implementação complexa	**Descrição:** A versão inicial foi implementada sem seguir boas práticas e, consequentemente, possuía alta complexidade de entendimento. **Solução:** Reimplementação de toda a etapa seguindo boas práticas de programação, acrescentando um fluxo de controle e padrões de projeto.

O mapeamento, de forma geral, deve informar quais elementos de código da arquitetura atual são a implementação do elemento arquitetural da PA. Para tanto, essa atividade é realizada com o auxílio de uma interface gráfica simples, que auxilia a visualização e definição do mapeamento. Esse mapeamento possibilita um nível de granularidade relativamente baixo, uma vez que além de pacotes é possível mapear classes, enumerações e interfaces.

Na Figura 4.6 pode ser observada a interface gráfica citada. Esta interface é dividida em três partes principais. A primeira é denominada "*Architectural Elements of the PA*", essa parte da interface é composta pela árvore de elementos arquiteturais que representa a arquitetura planejada elaborada na etapa anterior. Neste momento é importante salientar que só são utilizados os elementos arquiteturais da arquitetura planejada, ou seja, não são utilizadas as restrições arquiteturais. A segunda é denominada "*Code Elements*", essa parte é composta pela árvore de elementos de código que representam o código recuperado do sistema. Por fim, a terceira é denominada "*Mapped Elements*", essa parte é composta pelos elementos que foram mapeados por meio da seleção em ambas as outras duas partes e mediante o clique do botão "*Map Element*".

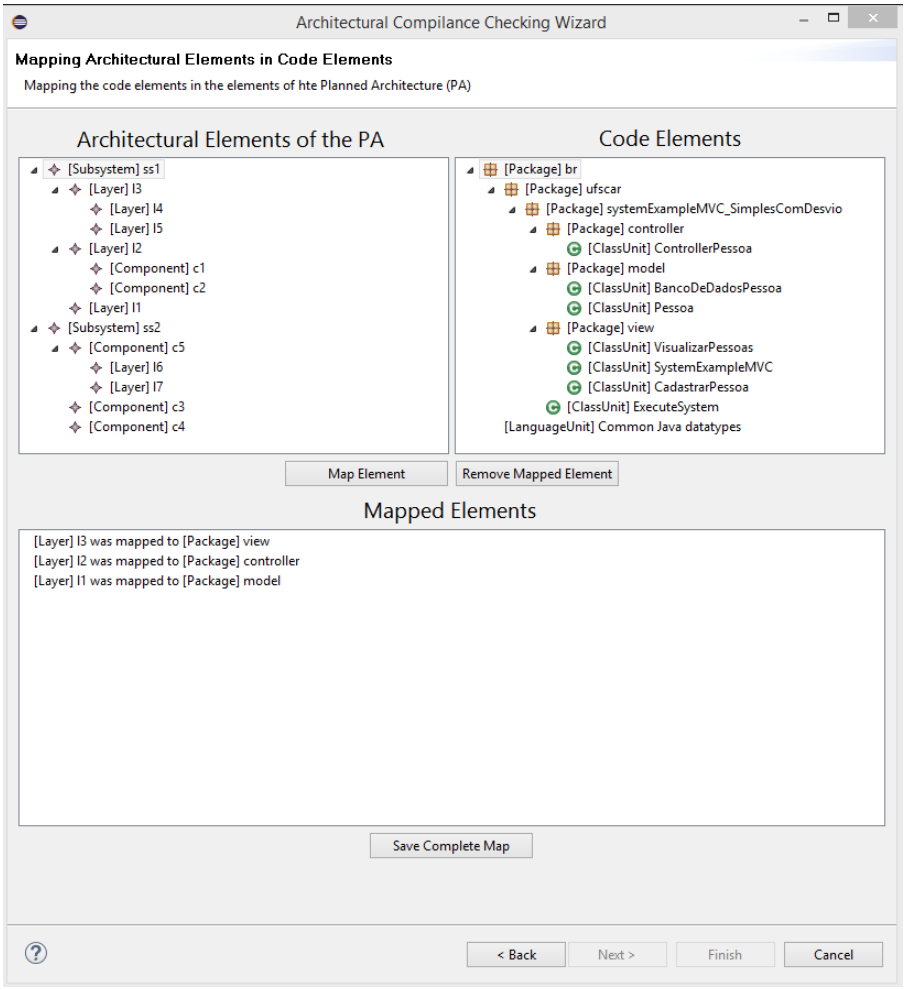

Figura 4.6: Interface gráfica do mapeamento entre Arquitetura Planejada e Atual. Fonte: Elaborado pelo autor

Após esse mapeamento, uma instância do Pacote *Structure* do KDM é adicionada à instância do KDM que representa o sistema legado. Na Figura 4.6 observa-se um mapeamento hipotético entre a arquitetura planejada e o sistema exemplo no campo *Mapped Elements*. Na Figura 4.7 observa-se o resultado do mapeamento após processado e gerado a instância do Pacote *Structure* e, na Figura 4.8 observa-se as propriedades mapeadas dos elementos arquiteturais sublinhados.

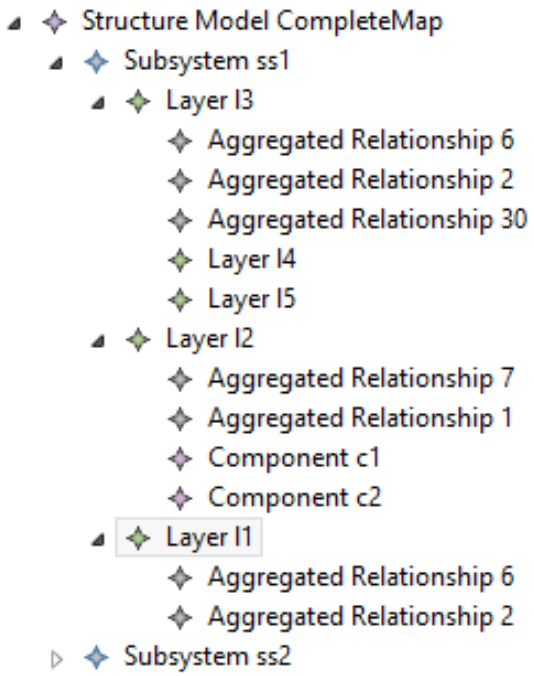

Figura 4.7: Exemplo de instância KDM do mapeamento. Fonte: Elaborado pelo autor

Property	Value
Group	
Grouped Element	
Implementation	Package view
In Aggregated	Aggregated Relationship 30
Inbound	
Model	
Name	l3
Out Aggregated	Aggregated Relationship 6, Aggregated Relationship 2, Aggregated Relationship 30
Outbound	
Owner	
Stereotype	

Property	Value
Group	
Grouped Element	
Implementation	Package controller
In Aggregated	Aggregated Relationship 2, Aggregated Relationship 1, Aggregated Relationship 2
Inbound	
Model	
Name	l2
Out Aggregated	Aggregated Relationship 7, Aggregated Relationship 1
Outbound	
Owner	
Stereotype	

Property	Value
Group	
Grouped Element	
Implementation	Package model
In Aggregated	Aggregated Relationship 6, Aggregated Relationship 7, Aggregated Relationship 6
Inbound	
Model	
Name	l1
Out Aggregated	Aggregated Relationship 6, Aggregated Relationship 2
Outbound	
Owner	
Stereotype	

Figura 4.8: Propriedades dos elementos arquiteturais mapeados. Fonte: Elaborado pelo autor

Esse mapeamento realizado e que pode ser observado na Figura 4.7 é separado em dois algoritmos. O primeiro algoritmo pode ser observado no Algoritmo 1 e é responsável por realizar um mapeamento inicial direto entre elemento arquitetural e elemento de código. Esse algoritmo realiza a inclusão no elemento arquitetural do elemento de código correspondente

definido no mapeamento.

Algoritmo 1: MAPEAMENTO INICIAL DA ARQUITETURA ATUAL APÓS AS EVOLUÇÕES

Entrada: Elementos Arquiteturais Selecionados, Mapa com o Mapeamento de Código, Instância do KDM do Sistema
Saída: Instância do KDM com o Mapeamento Inicial

```
1  início
2      para cada elemArqui ∈ arquiteturaPlanejadaElementosSelecionados faça
3          elemArqui.implementacao ← mapa.get(elemArqui.nome)
4      fim
5  fim
```

O segundo algoritmo pode ser observado nos Algoritmos 2, 3 e 4 e é responsável por realizar um mapeamento profundo dos elementos de código relacionados. Esse mapeamento avalia o mapeamento inicial realizado e analisa cada elemento de código que está em cada elemento arquitetural, com o fim de procurar no sistema todos os relacionamentos existentes que tem como partida ou chegada esse elemento de código. Dessa forma, é possível encontrar todos os relacionamentos de um elemento arquitetural com outro, sendo possível atualizar a metaclasse `AggregatedRelationship`, obtendo-se assim um mapeamento completo dos elementos arquiteturais e código do sistema.

Algoritmo 2: MAPEAMENTO DA ARQUITETURA ATUAL APÓS AS EVOLUÇÕES

Entrada: Instância do KDM da Arquitetura Planejada com os Elementos de Código mapeados, Instância do KDM do Sistema
Saída: Instância do KDM da Arquitetura Atual

```
1  início
2      para cada elemArqui ∈ arquiteturaPlanejada faça
3          para cada elemCod ∈ elemArqui faça
4              para cada tipoRel ∈ todosTiposRel faça
5                  todosRelElemCodPorElemArch ←
                     RECRELELEM(tipoRel, elemCod, arquiteturaPlanejada)
6                  INSIRAOUATUALIZEAGGREGATED(elemArqui,
7                  todosRelElemCodPorElemArch)
8              fim
9          fim
10     fim
11 fim
```

Algoritmo 3: MÉTODO recRelElem APÓS AS EVOLUÇÕES

Entrada: Tipo de Relacionamento, Elemento de código para análise, KDM da Arquitetura Planejada
Saída: Relacionamentos do elemento de código analisado por Elemento Arquitetural

```
1 início
2 |  todosRel ← tipoRel.verificarEsseTipoDeRelacaoNo(elemCod, KDMSistema)
3 fim
4 retorna
  MAPEARPARAELEMENTOSARQUITETURAIS(todosRel, arquiteturaPlanejada)
```

Algoritmo 4: MÉTODO insiraOuAtualizeAggregated APÓS AS EVOLUÇÕES

Entrada: Elemento arquitetural analisado, Relacionamentos do elemento arquitetural analisado por Elemento Arquitetural

```
1  início
2  |  para cada elemArquiDestino ∈ todosRelElemCodPorElemArch faça
3  |  |  se elemArquiDestino ∈ elemArqui.todosAggregatedRel.Destino então
4  |  |  |  elemArqui.aggregatedRel.adiciona(
5  |  |  |  todosRelElemCodPorElemArch(elemArquiDestino)
6  |  |  |  elemArqui.aggregatedRel.atualizaDensidade()
7  |  |  fim
8  |  |  senão
9  |  |  |  elemArqui.criaAggregated(elementoArquiDestino)
10 |  |  |  elemArqui.aggregatedRel.adiciona(
11 |  |  |  todosRelElemCodPorElemArch(elemArquiDestino)
12 |  |  |  elemArqui.aggregatedRel.atualizaDensidade()
13 |  |  fim
14 |  fim
15 fim
```

4.3.3 Comparação Arquitetural

Com o intuito de realizar a CCA, essa etapa é composta por uma única parte que é realização da CCA a partir das instâncias do KDM que representam a arquitetura planejada e a arquitetura atual geradas pelas etapas anteriores. O principal objetivo dessa etapa é a possibilidade de se realizar a comparação entre ambas as instâncias para a identificação de possíveis violações arquiteturais. Na Tabela 4.4 podem ser observadas as evoluções realizadas nessa etapa, bem como uma breve descrição. Como envolve vários detalhes técnicos, as explicações são detalhadas no Apêndice D.

De forma simples, o formato do processo de Checagem de Conformidade Arquitetural adotado neste trabalho realiza uma comparação entre a Arquitetura Planejada e a Arquitetura Atual para obter-se as violações arquiteturais. Para facilitar o entendimento, na Figura 4.9 pode ser

Tabela 4.4: Evoluções realizadas na "Etapa III - Checagem de Conformidade Arquitetural"

Limitação	Descrição / Solução
Carência de elementos	**Descrição:** A implementação original só reconhecia dois elementos de código (`Class` e `Package`), dessa forma, nem todas as metaclasses do KDM que representavam elementos de código eram usadas nesta etapa. **Solução:** Reimplementação e inclusão do reconhecimento dos elementos de código `Enumaration` e `Interface`.
Carência de metaclasses	**Descrição:** Existia um subconjunto fixo de 8 metaclasses do KDM (`Calls`, `UsesType`, `Creates`, `Extends`, `Implements`, `HasValue`, `Imports`, `HasType`) que eram utilizados para a realização do processo de mapeamento da arquitetura atual, sendo que este subconjunto limitava o processo posteriormente. **Solução:** Reimplementação dos algoritmos utilizando todas as metaclasses do KDM pertinentes a etapa.
Implementação com falha	**Descrição:** Existiam falhas de execução dos algoritmos desta etapa para padrões específicos de entradas e não abrangia todas as metaclasses do KDM, consequentemente mesmo que incluídas nas etapas anteriores, não seriam utilizadas nesta etapa. A implementação dos algoritmos além de possibilitar falhas era altamente complexa. **Solução:** Reimplementação dos algoritmos diminuindo sua complexidade, corrigindo as falhas encontradas e incluindo todas as metaclasses.
Carência de Interface	**Descrição:** Existia a falta de um fluxo de execução do protótipo para a abordagem, não haviam interfaces gráficas ou explicações que orientassem a utilização da ferramenta. **Solução:** Desenvolveu-se interfaces gráficas controlando e facilitando a utilização da ferramenta.
Implementação complexa	**Descrição:** A versão inicial foi implementada sem seguir boas práticas e, consequentemente, possuía alta complexidade de entendimento. **Solução:** Reimplementação de toda a etapa seguindo boas práticas de programação, acrescentando um fluxo de controle e padrões de projeto.

observado um exemplo gráfico simples da realização da CCA adotada.

Como pode ser observado na Figura 4.9 e conforme informações já citadas anteriormente, observa-se uma arquitetura planejada para duas camadas, (`Controller` e `Model`) e a definição de duas restrições arquiteturais explícitas e duas implícitas sendo elas: i) `Controller` pode realizar acessos do tipo `call` em `Model`; ii) `Controller` pode realizar acessos do tipo `import` em `Model`; iii) `Controller` não pode realizar nenhum tipo de acesso exceto `call` e `import` em `Model`; iv) `Model` não pode realizar nenhum tipo de acesso em `Controller`.

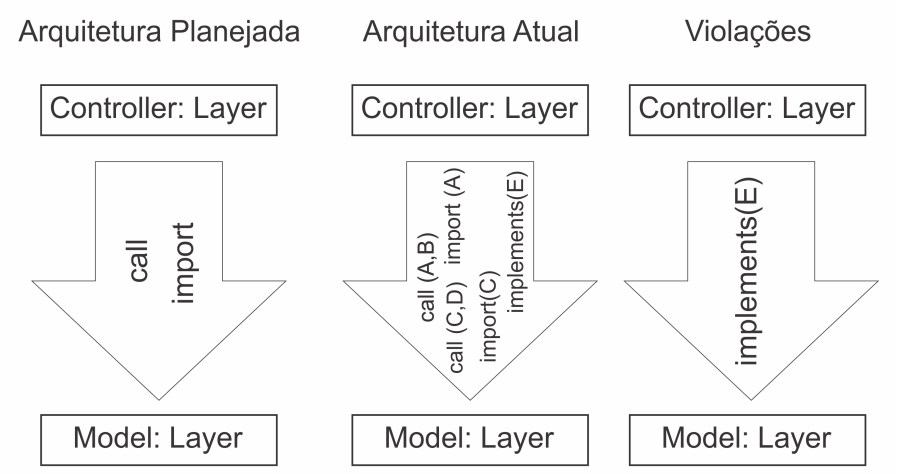

Figura 4.9: Exemplo gráfico da checagem de conformidade arquitetural. Fonte: Elaborado pelo autor

Observa-se também na figura, um exemplo de arquitetura atual de um sistema no qual a camada `Controller` realiza cinco acessos a camada `Model`. Cada acesso representando um tipo diferente e tendo diferentes elementos envolvidos. Por exemplo, o acesso `call(A,B)` representa uma chamada de método entre A e B.

Por fim, tem-se a representação das violações resultantes da realização da CCA. Nessa representação, tem-se o acesso denominado `implements(E)` como violação uma vez que este acesso existe, porém, não foi definido a permissividade de acessos do tipo `implements` entre as duas camadas.

Dessa forma, para realizar o processo de CCA em KDM fez-se necessário a criação de um algoritmo que realiza a verificação de quais metaclasses estão instanciadas na arquitetura planejada e quais metaclasses existem na arquitetura atual para que seja possível verificar as metaclasses que não foram previstas e, portanto, são violações. Esse algoritmo é representado nos Algoritmos 5 e 6.

4.3.4 Visualização de Desvios Arquiteturais

Com o intuito de preparar a abordagem para o processo de Reconciliação Arquitetural (RA) e conforme a realização desse estudo, observou-se a necessidade dessa etapa. A motivação principal para a inclusão dessa etapa na abordagem são as peculiaridades do KDM e o processo de RA.

Depois de realizada a etapa anterior, obtêm-se em formato de instância do KDM as violações arquiteturais. Cada violação arquitetural gerada representa uma pequena parte do problema ar-

Algoritmo 5: ALGORITMO DE CHECAGEM DE CONFORMIDADE ARQUITETURAL APÓS AS EVOLUÇÕES

Entrada: Instância do KDM da Arquitetura Planejada, Instância do KDM da Arquitetura Atual
Saída: Instância do KDM da Arquitetura Atual com um novo modelo composto por violações

```
1  início
2      aggregatedsPossiveis ← GETAGGREGATEDSFROM(arquiPlan)
3      aggregatedsAtuais ← GETAGGREGATEDSFROM(arquiAtual)
4      para cada aggregatedsPossivel ∈ aggregatedsPossiveis faça
5          origem ← aggregatedsPossivel.getFrom
6          destino ← aggregatedsPossivel.getTo
7          relacionamentosQuePodem ← aggregatedsPossivel.getRelation
8          aggregatedsAtuais ←
             RECAGGREGATEDSIGUAISATUAIS(destino, origem, aggregatedsAtuais)
9          para cada aggregatedAtual ∈ aggregatedsAtuais faça
10             EXCLUIRELACIONAMENTOS(relacionamentosQuePodem, aggregatedAtual)
11         fim
12     fim
13 fim
14 retorna arquiAtual
```

Algoritmo 6: ALGORITMO DE CHECAGEM DE CONFORMIDADE ARQUITETURAL APÓS AS EVOLUÇÕES - MÉTODO EXCLUIRELACIONAMENTOS

Entrada: Relacionamentos que podem existir baseado na arquitetura planejada, Instâncias da metaclasse `AggregatedRelationship` para serem validadas pela CCA

```
1  início
2      para cada relParaExcluir ∈ relacionamentosQuePodem faça
3          para cada relacionamentoAtual ∈ aggregatedAtual.getRelation faça
4              se relParaExcluir.class = relacionamentoAtual.class então
5                  aggregatedAtual.remove(relacionamentoAtual)
6                  aggregatedAtual.updateDensity
7              fim
8          fim
9      fim
10 fim
```

quitetural identificado, sendo necessário realizar um processo de identificação de similaridade para obter-se o desvio arquitetural como um todo.

Na Figura 4.10 pode ser observado um exemplo de instância do KDM gerado pela etapa anterior. Conforme pode ser observado na figura, existem duas instâncias da metaclasse de relacionamento entre elementos arquiteturais (`AggregatedRelationship`) que agrupam todas as

violações encontradas no processo da etapa anterior. Por exemplo, a instância com oito elementos contém quatro violações do tipo `call`, duas violações do tipo `create` e duas violações do tipo `hasType`.

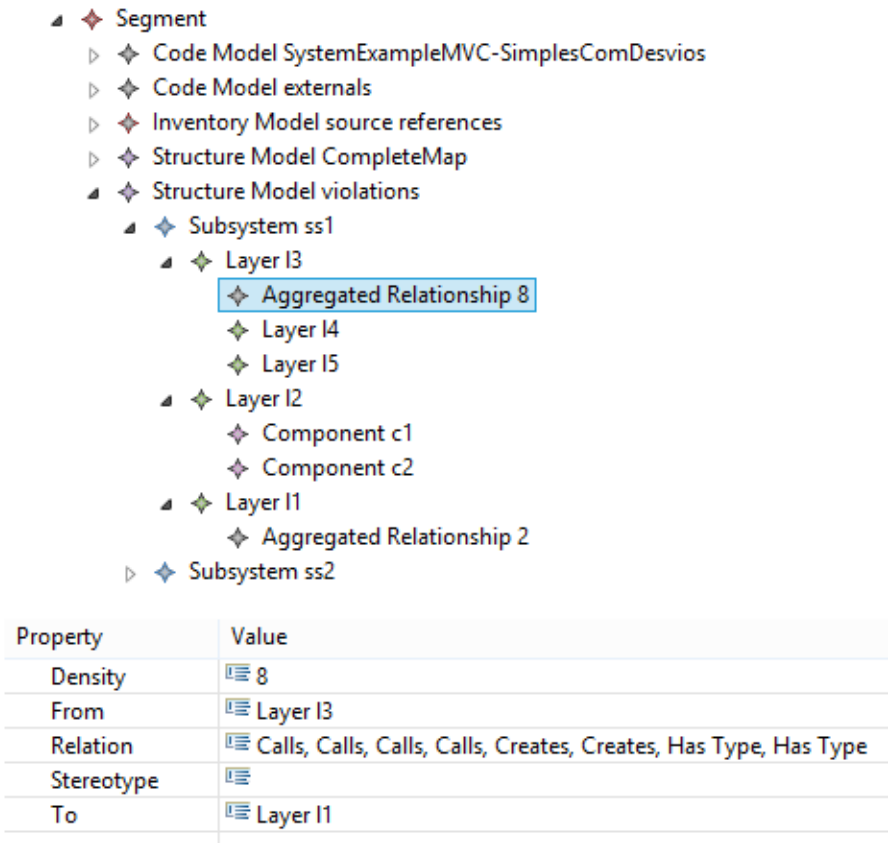

Figura 4.10: Exemplo de instância gerada na etapa anterior. Fonte: Elaborado pelo autor

Apesar das violações estarem agrupadas em instâncias da metaclasse de relacionamento (`AggregatedRelationship`), são geradas sem agrupar as violações, ou seja, da maneira que são apresentadas não representam desvios arquiteturais, representam apenas problemas pontuais em uma granularidade muito fina. Para que seja possível transformar essas violações em baixa granularidade em um desvio arquitetural que pode ser observado no código fonte, é necessário a realização de um processo de agrupamento de violações correlacionadas.

Para tanto, existem diversas formas de realizar esse agrupamento. O agrupamento implementado como prova de conceito é baseado na combinação de Matriz de Proximidade e a clusterização pelo algoritmo DBSCAN com sua implementação padrão disponibilizado pela ferramenta Weka (Bouckaert et al., 2010; Garner, 1995; Borah; Bhattacharyya, 2004; Ester et al., 1996). A matriz é baseada no caminho da hierarquia de origem e destino dos elementos arquiteturais. Outro exemplo de agrupamento é o realizado por um membro do grupo de pesquisa AdvanSE, para a abordagem Arch-KDM 2.0 (Gasparini, 2018). Gasparini (2018) realiza um trabalho de visualização de desvios arquiteturais em UML baseados nas violações encontradas pela Arch-

KDM 2.0. Sua implementação utiliza uma abordagem diferente realizando uma comparação direta com as violações encontradas e que possuem sua origem como uma classe comum na hierarquia. Na Figura 4.11 pode ser observado um exemplo de resultado gerado pela execução da combinação de Matriz de Proximidade e a clusterização pelo DBSCAN. Neste exemplo o algoritmo gerou seis desvios arquiteturais do total de 10 violações encontradas pela etapa de comparação de arquiteturas.

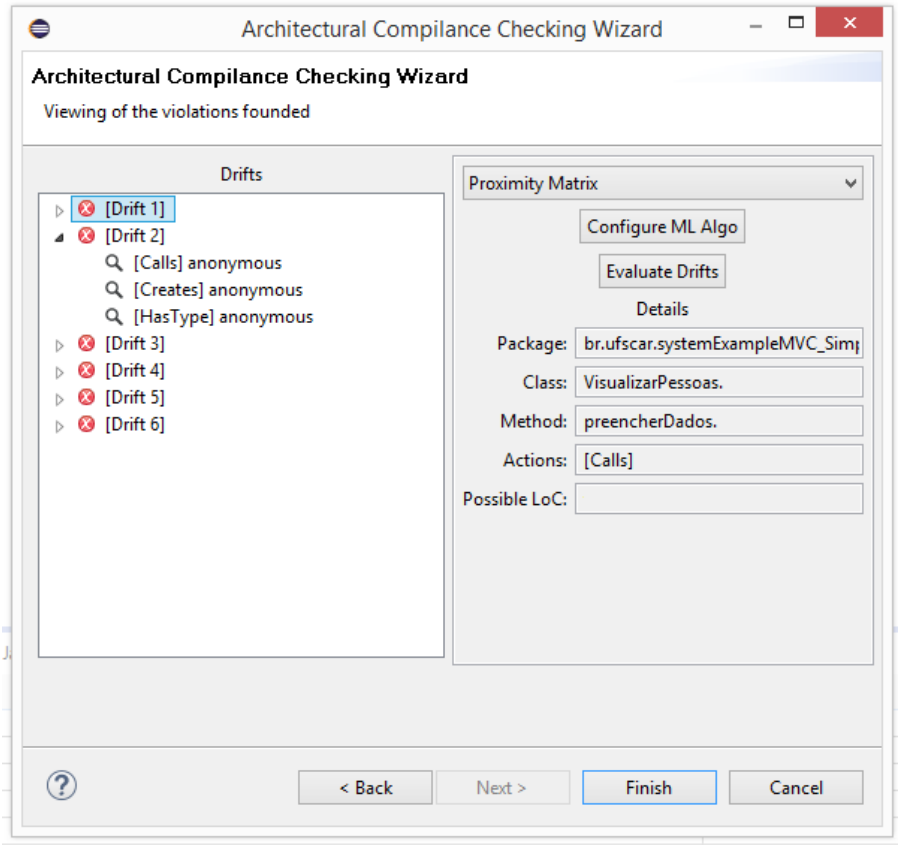

Figura 4.11: Exemplo de agrupamento de violações. Fonte: Elaborado pelo autor

4.4 Considerações Finais

Neste capítulo foram abordados alguns pontos importantes em relação a este trabalho, tais como as evoluções realizadas na primeira versão da Arch-KDM, a DCL-KDM, os algoritmos utilizados no processo de extração da arquitetura atual do sistema e os algoritmos de execução da CCA no contexto da ADM. Arch-KDM 2.0 é a extensão, correção e preparação da abordagem para a realização da Checagem de Conformidade para o processo de Reconciliação Arquitetural. A DCL-KDM é a linguagem específica de domínio que sofreu algumas evoluções para a realização da especificação da arquitetura planejada, a qual é responsável por definir restrições entre os elementos arquiteturais. É importante destacar que a DCL-KDM sofreu modificações

em relação a primeira versão da DCL-KDM quanto a restrições arquiteturais que é capaz de representar.

Foi apresentada uma visão geral do contexto que envolve a abordagem de checagem e mostradas as etapas de uma modernização que este trabalho se enquadra, o que é necessário para a abordagem funcionar no contexto da ADM e também como é possível identificar dependências no metamodelo KDM. Com isso, foi apresentada a checagem de conformidade arquitetural realizada entre a arquitetura atual e planejada de um sistema. Por fim, foi apresentado a forma de realizar a visualização de violações e desvios arquiteturais, bem como a definição teórica e matemática (baseada em teoria de conjuntos) de cada um dos dois termos.

Capítulo 5

ARCH-KDM 2.0: CENÁRIO DE UTILIZAÇÃO DA ABORDAGEM

5.1 Considerações Iniciais

Neste capítulo é apresentado um estudo de caso da utilização da abordagem no contexto da ADM. Este capítulo foi escrito no formato de uma narração, de forma a facilitar ao leitor o entendimento e interpretação das atividades que acarretaram a utilização da abordagem. Na Seção 5.2, é narrada a motivação e cenário que um engenheiro/arquiteto de software pode possuir para realizar a utilização da abordagem apresentada nesta dissertação. A Seção 5.3, especificamente, deixa de ser uma narrativa e apresenta o sistema *myAppointments* utilizado para a construção do cenário de uso. Na Seção 5.4, é narrada a utilização do apoio computacional para a primeira etapa da abordagem. Na Seção 5.5, é narrada a utilização do apoio computacional para a segunda etapa da abordagem. Na Seção 5.6, é narrada a utilização do apoio computacional para a terceira e quarta etapa da abordagem. Por fim, na Seção 5.7, são apresentadas as considerações finais deste capítulo.

5.2 Cenário de Uso

O sistema *myAppointments* é utilizado internamente por uma determinada empresa durante muitos anos. Por este motivo, o sistema já passou por muitas manutenções e evoluções. Seu custo de manutenção com o passar do tempo tornou-se cada vez mais alto e as alterações, inclusões e exclusões demandam muito tempo para serem feitas.

André era o engenheiro de software responsável pela infraestrutura de TI desta empresa, era o principal responsável pelo sistema. Diante deste desafio, André realizou uma análise no sis-

tema e concluiu que existia a possibilidade do sistema estar repleto de problemas arquiteturais. Dessa forma, ele decidiu pesquisar sobre formas e abordagens de identificar esses problemas arquiteturais.

Por meio das pesquisas sobre a checagem de conformidade arquitetural, André encontrou na literatura uma abordagem denominada Arch-KDM. Dessa forma André estudou a abordagem e viu que ela possui um apoio ferramental e é dividida em quatro etapas. A Etapa I é denominada de "Especificação da Arquitetura Planejada", a segunda etapa é denominada "Extração da Arquitetura Atual", a terceira etapa é denominada "Comparação de Arquiteturas" e, por fim, a quarta etapa é denominada "Visualização de Desvios Arquiteturais". Dessa forma, após a análise da abordagem, André conclui que ela pode solucionar seu problema e identificar quais são os problemas arquiteturais do sistema.

5.3 Detalhes do Sistema *myAppointments*

O sistema escolhido para a realização desse cenário de uso é denominado *myAppointments* (Pinto; Terra, 2015; Passos et al., 2010). Este sistema foi implementado por engenheiros de software do Grupo de Engenharia de Software da universidade Pontifícia Universidade Católica de Minas Gerais e foi disponibilizado pelos autores dos trabalhos Pinto e Terra (2015) e Passos et al. (2010). Sua principal função é realizar o gerenciamento de informações pessoais de uma agenda.

Este sistema foi desenvolvido respeitando fielmente e estritamente a arquitetura planejada de forma que não existam desvios arquiteturais no mesmo. Dessa forma, sendo possível a realização controlada de testes baseados em inserção de desvios arquiteturais voltados a demonstração de automatização (Pinto; Terra, 2015; Passos et al., 2010).

A arquitetura planejada do sistema pode ser observada na Figura 5.1 e embora o sistema seja simples e pequeno, ele utiliza as principais restrições de dependência envolvendo o padrão MVC. Além dessas restrições o sistema ainda possui as seguintes restrições arquiteturais (Pinto; Terra, 2015; Passos et al., 2010):

- **RA1**: Somente a camada `View` pode depender dos componentes providos pelo `AWT/Swing`.
- **RA2**: Somente os `DAOs` da camada `Model` podem depender dos serviços de banco de dados. Uma exceção é concedida para a classe model.DB, responsável por controlar as conexões do banco de dados.

- **RA3**: A camada `View` pode depender apenas dos serviços providos por ela mesma, pela camada `Controller` e pelo pacote `Util` (por exemplo, para dissociar a apresentação dos dados do acesso aos dados, componentes do `View` não podem acessar componentes do `Model` diretamente).
- **RA4**: `Domain Objects` não devem depender dos módulos `DAO`, `Controller` e `View`.
- **RA5**: Classes `DAO` podem depender somente de `Domain Objects`, das classes `Model` autorizadas a utilizar os serviços de banco de dados (como o model.DB), quanto do pacote `Util`.
- **RA6**: O pacote `Util` não pode depender de nenhuma classe específica do código fonte do sistema.

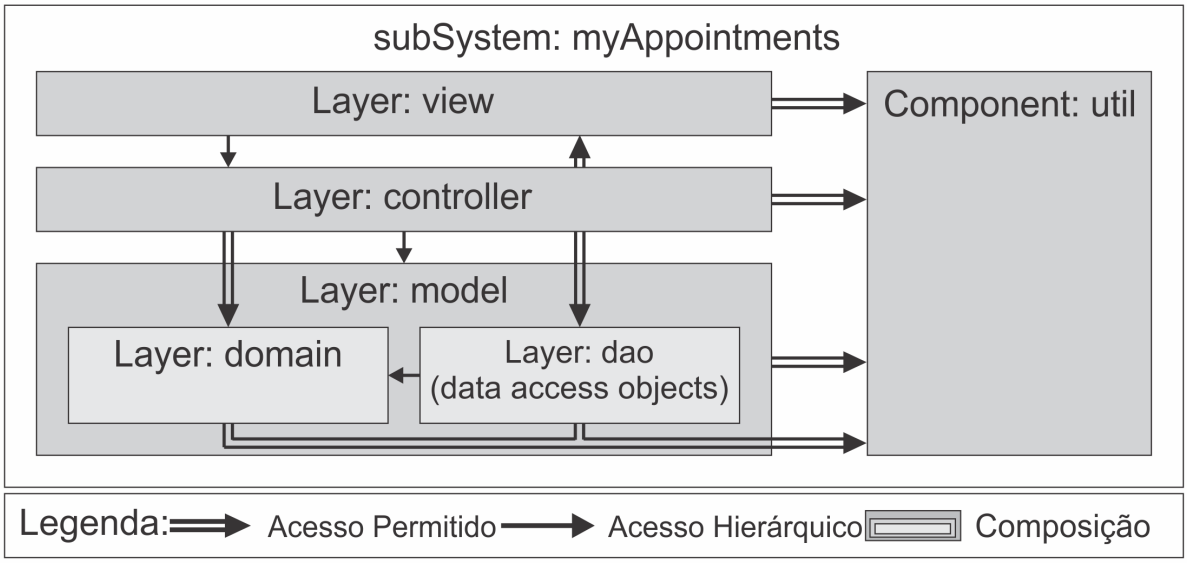

Figura 5.1: Arquitetura Planejada do sistema *myAppointments*. Fonte: Adaptado de (Pinto; Terra, 2015; Passos et al., 2010)

Para a apresentação deste cenário do ponto de vista de um usuário final da abordagem, realizou-se a inserção de três desvios arquiteturais de maneira controlada no sistema. O primeiro desvio arquitetural viola a restrição **RA3**. No Código 5.1 pode ser observado o desvio incluído no sistema. Como pode ser observado foi realizada a inclusão de um atributo na classe `Appointment` e um método, `firstDrift`, que retorna uma nova instância para esse atributo.

```
1 ...
2
3 import myappointments.model.domain.Appointment; *
4
5 public class AgendaView extends AbstractAgendaView {
6
```

```
7     private Appointment firstDrift; *
8
9     public AgendaView(AgendaController controller) {
10        ...
11    }
12
13    public Appointment firstDrift(){    *
14        firstDrift = new Appointment(); *
15        return firstDrift;              *
16    }                                   *
17
18 ...
```

Código 5.1: Código do primeiro desvio inserido no sistema

```
1  ...
2
3  import myappointments.model.AgendaDAO;  *
4  import myappointments.model.DAOCommand; *
5  import myappointments.model.DB;         *
6  import java.sql.PreparedStatement;      *
7  import java.sql.Timestamp;              *
8
9  ...
10
11 public class Appointment  {
12
13    ...
14
15    public void addAppointment(AgendaDAO agendaDAO) throws Exception {          *
16        if (getDate().compareTo
17                (DateUtils.getCurrentDate()) < 0)                              *
18            throw new IllegalArgumentException
19                    ("Date field cannot be prior to current date and time!") ;  *
20
21        String update =
22                "INSERT INTO APPOINTMENT (TITLE, NOTE, TIME) " +
23                "VALUES (?, ?, ?)" ;                                           *
24
25        PreparedStatement stmt = DB.getConnection().prepareStatement(update) ;  *
26        stmt.setString(1, getTitle()) ;                                        *
27        stmt.setString(2, getNote()) ;                                         *
28        stmt.setTimestamp(3, new Timestamp(getDate().getTime())) ;             *
29        stmt.executeUpdate();                                                  *
30        stmt.close() ;                                                         *
31
32        agendaDAO.notifyObservers(new DAOCommand(DAOCommand.Type.INSERT, this));*
33    }                                                                          *
34
35 ...
```

Código 5.2: Código do segundo desvio inserido no sistema

```
1 ...
```

```
2
3 import myappointments.controller.IController;
4 import myappointments.model.AbstractAgendaDAO;
5 import myappointments.model.domain.Appointment;
6 import myappointments.view.View;
7
8 ...
9
10 public class DateUtils extends AbstractAgendaDAO implements IController{
11
12     ...
13
14     @Override
15     public View getView() {
16         return null;
17     }
18
19     @Override
20     public void start() {
21     }
22
23     @Override
24     public void removeAppointment(Date date) throws Exception {
25     }
26
27     @Override
28     public Appointment getAppointment(Date date) throws Exception {
29         return null;
30     }
31
32     @Override
33     public List<Appointment> getAppointments(int day, int month, int year) throws
           Exception {
34         return null;
35     }
36
37     @Override
38     public void saveAppointment(Appointment oldAppointment, Appointment newAppointment)
            throws Exception {
39     }
40
41 ...
```

Código 5.3: Código do segundo desvio inserido no sistema

O segundo desvio viola a restrição **RA4**. Para esse segundo desvio, foi realizada uma refatoração da classe `AgendaDAO` empregando o `move method` para transferir o método `addAppointment` para a classe `Appointment`. No Código 5.2 pode ser observado o código movido destacado com "*".

O terceiro desvio viola a restrição **RA6**. Para esse terceiro desvio, foi realizada a inclusão

de uma extensão e implementação de uma classe do pacote `Util`. Também foram incluídos os métodos sobrescritos referentes à extensão e implementação. No Código 5.3 pode ser observado o código movido.

5.4 Especificação da Arquitetura Planejada

Ao iniciar a primeira etapa da abordagem, André utilizou uma DSL denominada DCL-KDM para criar a arquitetura planejada do sistema. Uma vez que André já possuía a arquitetura planejada do sistema (Figura 5.1 - Seção 5.3) foi possível apenas realizar sua descrição utilizando a DCL-KDM.

A descrição elaborada em DCL-KDM por André, é auxiliada por um editor de texto do apoio computacional desenvolvido. Este editor de texto possui *autocomplete* e validações sintáticas baseadas na gramática da DCL-KDM. Na Figura 5.2 é possível observar algumas características do editor, como por exemplo as opções de auto complemento de palavras e a identificação de erros na especificação. Dessa forma, com o auxilio do editor e baseando-se na arquitetura planejada do sistema, André conseguiu gerar a descrição arquitetural em DCL-KDM observada no Código 5.4.

```
1 architeturalElements{
2     subSystem myAppointments;
3
4     layer view, level 3, inSubSystem: myAppointments;
5
6     layer controller, level 2, inSubSystem: myAppointments;
7
8     layer model, level 1, inSubSystem: myAppointments;
9     layer dao, level 2, inLayer: model;
10    layer domain, level 1, inLayer: model;
11
12    component util, inSubSystem: myAppointments;
13
14 }restrictions{
15    view can-depend util;
16
17    controller can-depend util;
18
19    model can-depend util;
20    dao can-depend util;
21    domain can-depend util;
22
23    controller can-depend view;
24    controller can-depend dao;
25    controller can-depend domain;
26 }
```

Código 5.4: Especificação do myAppointment em DCL-KDM

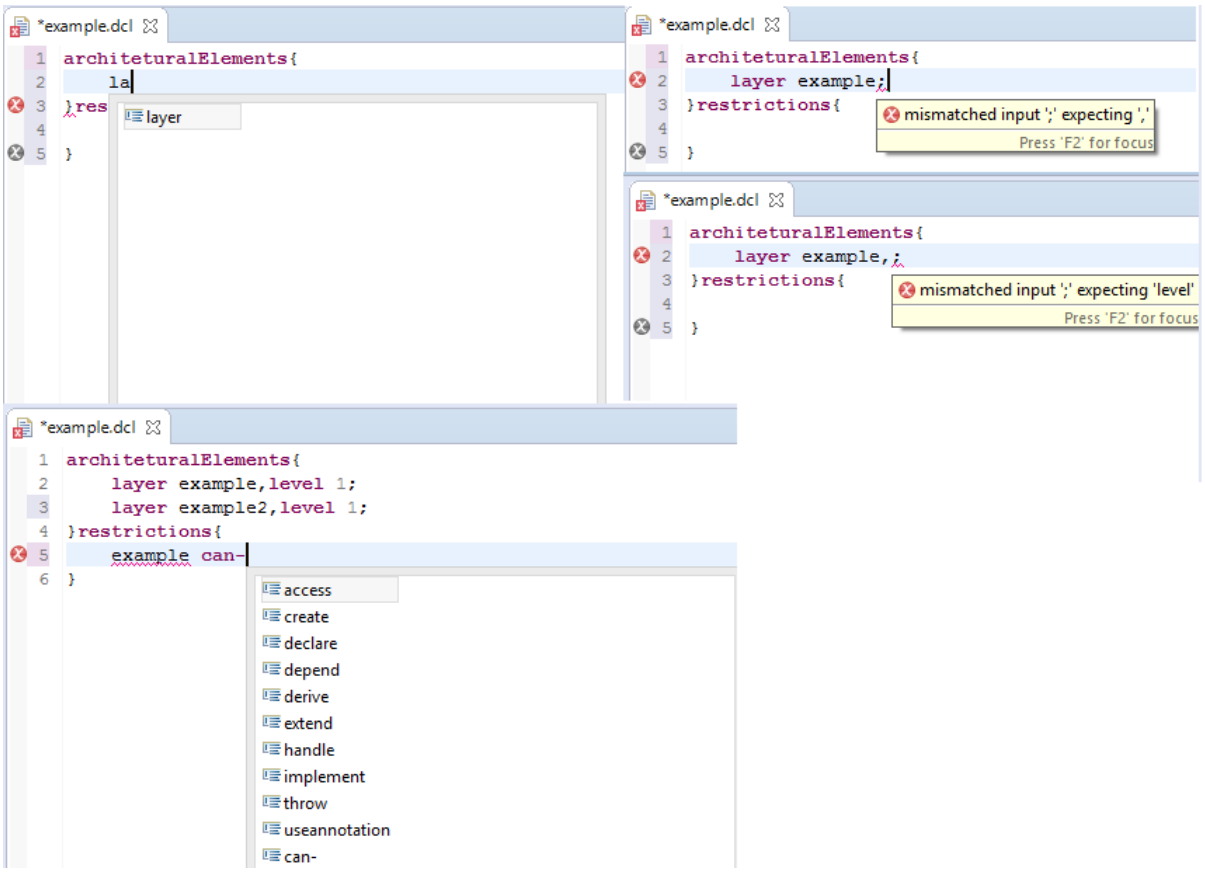

Figura 5.2: Exemplo das características do editor de texto da DCL-KDM. Fonte: Elaborado pelo autor

Depois de realizada a descrição arquitetural, André precisou criar sua representação em KDM. Para tanto, André utilizou outra funcionalidade do apoio computacional clicando com o botão auxiliar no editor de texto e no menu denominado "`DCL-KDM`" clicou no submenu denominado "`Generate KDM file from DCL`". Ambos menus podem ser observados na Figura 5.3.

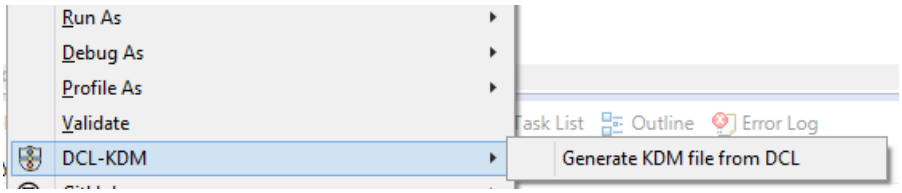

Figura 5.3: Menus de geração do KDM a partir da descrição. Fonte: Elaborado pelo autor

Ao realizar esta ação, André observou o arquivo que foi gerado no mesmo diretório do arquivo da DCL-KDM, dessa forma, armazenando-o e obtendo sua arquitetura planejada.

5.5 Extração da Arquitetura Atual

Ao iniciar a segunda etapa da abordagem, André precisou realizar a utilização de outro apoio computacional denominado *MoDisco* para gerar uma instância de seu sistema no metamodelo que a abordagem utiliza. Para realizar geração da instância do KDM com esta ferramenta, depois de instalá-la, André clicou com o botão auxiliar a partir da raiz do projeto de seu sistema, selecionou o menu "*Discovery*", selecionou o submenu "*Discoverers*" e clicou no item "*Discover KDM Code Model from Java Project...*". Na Figura 5.4 observa-se este conjunto de menus e item.

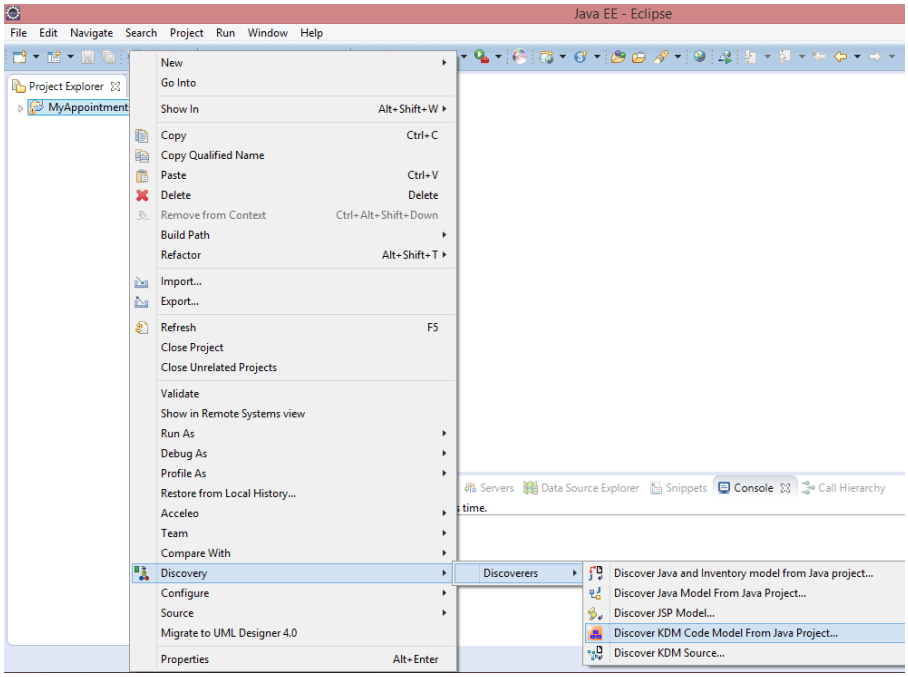

Figura 5.4: Menus de geração do KDM a partir do *MoDisco*. Fonte: Elaborado pelo autor

Depois de clicado no item, André precisou configurar um popup de seleção de informações para a geração da instância do KDM. Na Figura 5.5 observa-se este popup.

Depois de configurado e selecionado o botão "OK" um arquivo foi gerado na raiz do projeto (Figura 5.6), André, então obteve sua arquitetura atual do sistema.

Com os arquivos em mãos (arquitetura planejada e atual), André voltou ao apoio computacional proporcionado pela abordagem e iniciou um *wizard* que foi instalado pelo apoio computacional no menu "*Architectural Reconciliation*" e clicando no item "*Architectural Conformance Checking Wizard*". Na Figura 5.7 observa-se este menu e item.

Depois de clicar no item, André precisou configurar outro popup do *wizard*. Este popup pode ser observado na Figura 5.8. Ao ler as informações contidas no popup, como André sabia

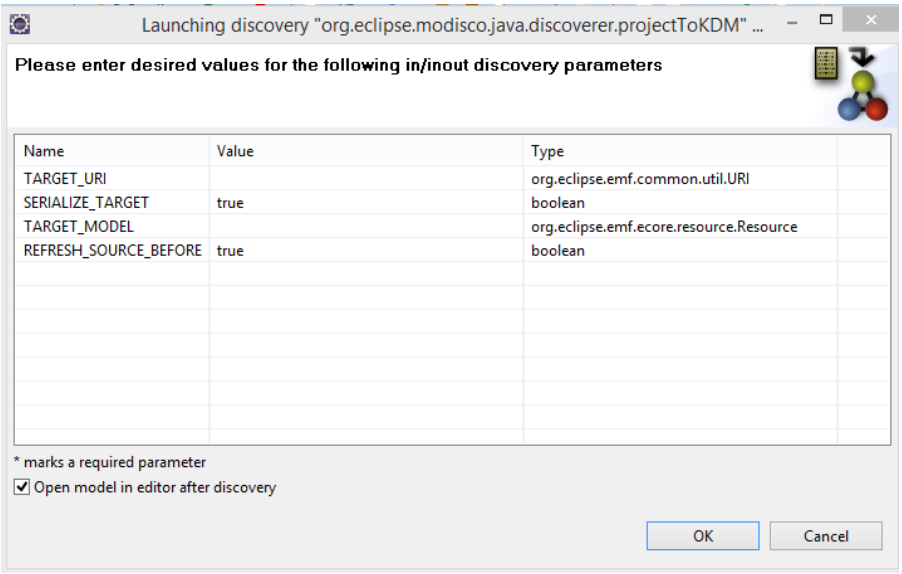

Figura 5.5: Popup aberto para configuração da geração da instância. Fonte: Elaborado pelo autor

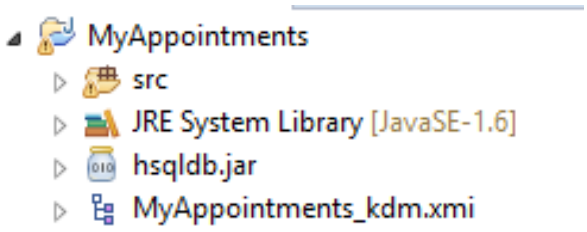

Figura 5.6: Arquivo gerado pelo *MoDisco*. Fonte: Elaborado pelo autor

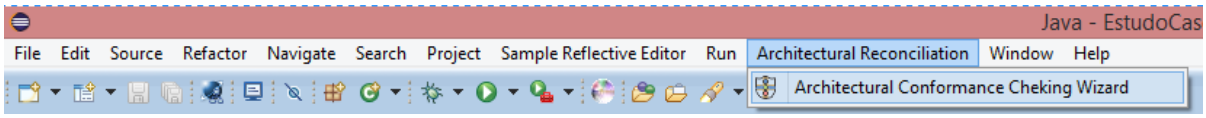

Figura 5.7: Menus para a utilização do *wizard* da abordagem. Fonte: Elaborado pelo autor

que o *wizard* era composto por passos, clicou no botão para prosseguir para o próximo passo. Nesse próximo passo, André observou alguns campos a serem preenchidos (Figura 5.9), sendo eles: i) a instância do KDM que representa a arquitetura planejada do sistema; ii) o tipo da arquitetura atual que será selecionada, tendo-se duas opções sendo elas arquitetura atual crua gerada por um sistema como o *MoDisco* ou uma arquitetura atual já mapeada e; iii) a instância do KDM que representa a arquitetura atual ou o código do sistema do sistema.

Depois de configurado as informações, André passou para o próximo passo e precisou configurar outro popup (Figura 5.10). Nesse popup, André precisou selecionar a instância do metamodelo KDM contendo o código do sistema, dessa forma, André selecionou o modelo que representava o sistema, o modelo denominado "`MyAppointments`".

Ao selecionar o modelo, André foi apresentado ao passo de mapeamento entre a arquitetura

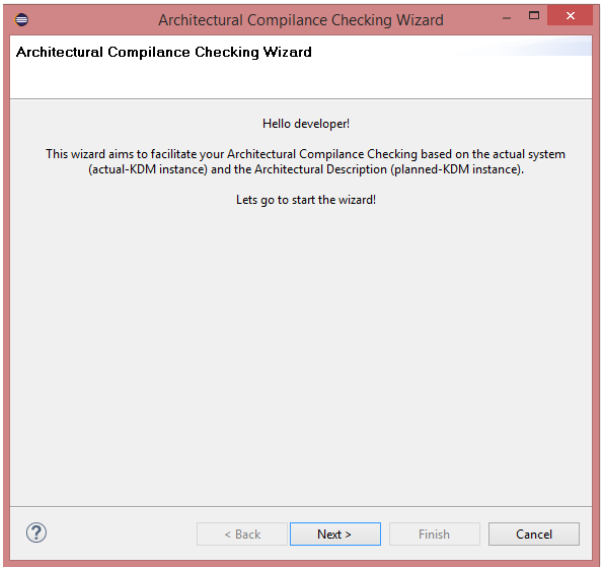

Figura 5.8: Interface inicial do *wizard*. Fonte: Elaborado pelo autor

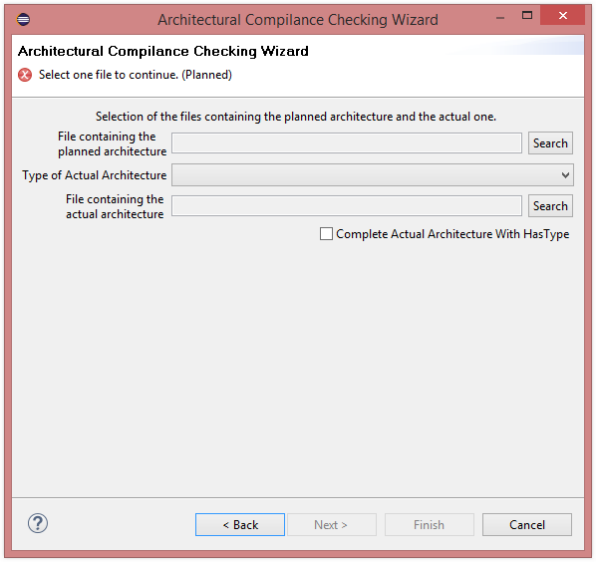

Figura 5.9: Interface de configuração das entradas da abordagem. Fonte: Elaborado pelo autor

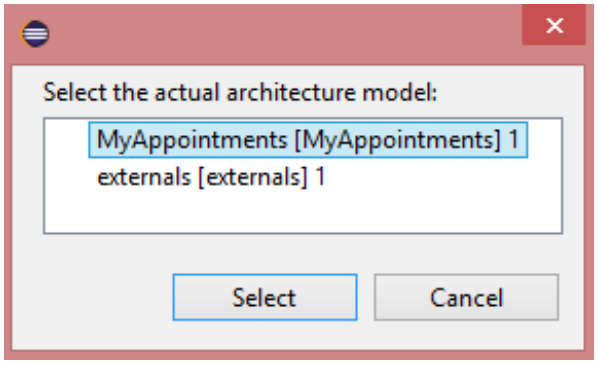

Figura 5.10: Popup de seleção de modelo. Fonte: Elaborado pelo autor

atual e a arquitetura planejada do sistema (Figura 5.11). André observou que a interface era dividida em três partes: i) uma parte denominada *Architectural Elements of the PA* que continha os elementos arquiteturais que havia descrito na DCL-KDM; ii) uma parte denominada *Code Elements* que continha os elementos do código de seu sistema e; iii) uma parte denominada *Mapped Elements* que estava em branco.

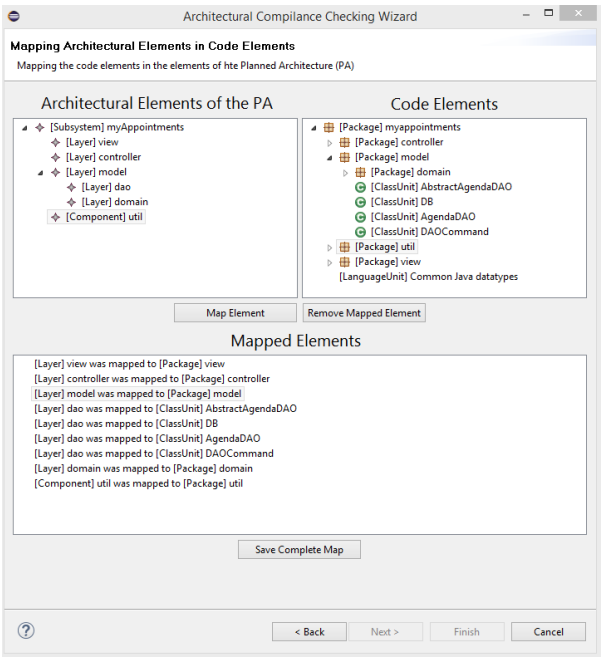

Figura 5.11: Interface de mapeamento entre a arquitetura planejada e o código fonte. Fonte: Elaborado pelo autor

Dessa forma, André realizou o mapeamento entre a arquitetura planejada e o código clicando em um elemento de código, em um elemento arquitetural e depois clicando no botão *Map Element*, preenchendo assim a parte denominada *Mapped Elements* com o mapeamento. Dessa forma, após o mapeamento, André obteve sua arquitetura atual mapeada com a planejada.

5.6 Comparação de Arquiteturas e Visualização de Desvios

André, ao continuar no *wizard*, passou para o próximo passo iniciando a terceira etapa da abordagem. Ao clicar no botão "*Initiate Processing*" André esperou por alguns segundos a execução do algoritmo de checagem de conformidade arquitetural (Figura 5.12);

Depois de finalizado, foi habilitado o botão para o próximo passo. Em continuidade ao *wizard*, no próximo passo, André precisou configurar algumas informações (Figura 5.13). André, dentre as duas opções de algoritmos para serem utilizados no agrupamento de desvios seleci-

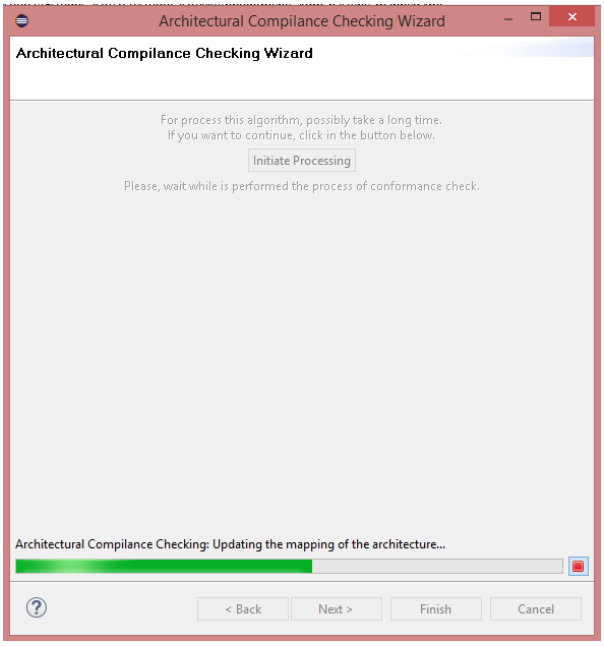

Figura 5.12: Interface do *wizard* para iniciar o algoritmo de CCA. Fonte: Elaborado pelo autor

onou o algoritmo de Matriz de Proximidade e ao clicar no botão "`Configure ML Algo`" configurou alguns parâmetros do algoritmo. Depois de selecionado e configurado, André, clicou no botão "*Evaluate Drifts*" e observou que foram gerados 9 itens com o padrão de nome "*Drift* [número]".

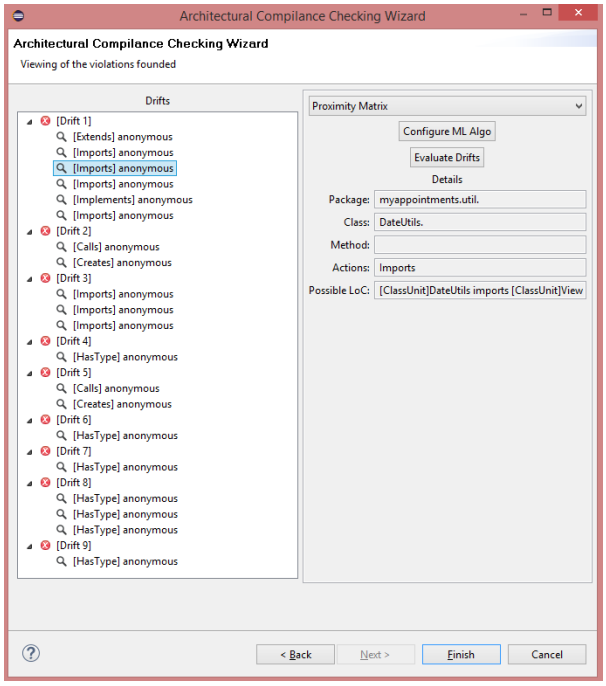

Figura 5.13: Interface de configuração do algoritmo de descoberta de desvios arquiteturais. Fonte: Elaborado pelo autor

Depois de um tempo analisando os agrupamentos, algumas tabelas encontradas na aborda-

gem sobre as metaclasses do KDM (Apêndice A - Tabelas A.1, A.2, A.3, A.4, A.5, A.6 e A.8), as informações disponíveis na interface do *wizard* e o código do sistema, André conseguiu reconhecer as linhas de código que foram definidas como os nove desvios apresentados pelo apoio ferramental. Dessa forma, André elaborou uma tabela para poder reconhecer os desvios arquiteturais que seu sistema possuía (Tabela 5.1).

Tabela 5.1: Código que representa cada desvio arquitetural encontrado pela Arch-KDM 2.0

Desvio Arquitetural	Código do Desvio
Drift 1	Classe `DateUtils`: `import myappointments.controller.IController; import myappointments.model.AbstractAgendaDAO; import myappointments.model.domain.Appointment; import myappointments.view.View; extends AbstractAgendaDAO implements IController`
Drift 2	Classe `AgendaView`: `new Appointment()`
Drift 3	Classe `Appointment`: `import myappointments.model.AgendaDAO; import myappointments.model.DAOCommand; import myappointments.model.DB;`
Drift 4	Classe `Appointment`: `AgendaDAO agendaDAO;`
Drift 5	Classe `Appointment`: `new DAOCommand(DAOCommand.Type.INSERT, this);`
Drift 6	Classe `AgendaView`: `public Appointment firstDrift;`
Drift 7	Classe `AgendaView`: `private Appointment firstDrift;`
Drift 8	Classe `DateUtils`: `Appointment oldAppointment, Appointment newAppointment; public Appointment getAppointment;`
Drift 9	Classe `DateUtils`: `public View getView();`

Refletindo sobre os problemas arquiteturais antes desconhecidos e agora listados, André conseguiu definir e planejar ações que deveriam ser tomadas para sanar os problemas e, consequentemente, melhorar a qualidade de seu sistema, facilitando a manutenção e diminuindo os custos empregados nessas ações.

5.7 Considerações Finais

Neste capítulo foi exemplificado a abordagem Arch-KDM 2.0 durante o estudo de caso de utilização e reconhecimento de desvios arquiteturais. Foi apresentada uma visão geral do sistema *myAppointments* criado especificamente com o intuito de validar abordagens de CCA.

Também foi apresentado de forma detalhada como se utiliza a abordagem de forma a facilitar o entendimento do funcionamento da mesma. Com isso, ao final do capítulo foi apresentado o resultado da CCA para o sistema. Dessa forma, percebeu-se que a utilização da abordagem tornou-se simples com o apoio computacional. Também percebeu-se que foi de fácil entendimento sua utilização e os resultados apresentados pelo apoio computacional foram objetivos e de fácil interpretação, fazendo com que a utilização da abordagem obtivesse exito no objetivo que se propunha.

Capítulo 6

AVALIAÇÃO

6.1 Considerações Iniciais

Neste capítulo é apresentada a avaliação da DCL-KDM e da Arch-KDM 2.0. O objetivo da abordagem principal da dissertação é analisar as violações e desvios arquiteturais encontradas em sua execução. Portanto, o melhor resultado é detectar as violações e desvios com o menor número possível de falsos positivos e negativos. Sendo assim, para atingir o melhor resultado para a abordagem almeja-se três pontos principais: i) a especificação da arquitetura planejada de forma correta; ii) a extração da arquitetura atual e seu mapeamento de forma correta e; iii) a qualidade do algoritmo de checagem. Na Seção 6.2, é apresentada a avaliação realizada para a DCL-KDM. E na Seção 6.3 a avaliação realizada para a Arch-KDM 2.0. Por fim, na Seção 6.5, são apresentadas as considerações finais deste capítulo.

6.2 Avaliação da DCL-KDM

Nesta seção serão descritas as duas avaliações realizadas na DCL-KDM. A avaliação conduzida nessa dissertação seguiu as orientações de Wohlin et al. (2000). Dessa forma, as Tabelas 6.1 e 6.2 apresentam o GQM (*Goal*/*Question*/*Metric*) elaborado para cada uma das avaliações. Nas tabelas podem ser observados os principais itens do GQM e uma breve descrição de cada um para cada umas das avaliações.

Nas subseções seguintes são apresentados: a estratégia de realização da avaliação, os resultados e uma breve discussão para cada uma das duas avaliações propostas.

Tabela 6.1: GQM para a primeira avaliação da DCL-KDM

Item	Descrição
Objeto do Estudo	DCL-KDM
Objetivo / Propósito	Comparar a DCL-KDM com o KDM-SDK tendo objetivo de verificar qual dos dois prove o melhor suporte em termos de produtividade dos engenheiros de software
Perspectiva	Engenheiros de software que necessitam especificar uma Arquitetura Planejada (PA) para um projeto de modernização de software baseado na ADM
Foco de Qualidade	Facilidade de especificação de uma PA comparando a DCL-KDM com uma abordagem existente KDM-SDK em termos de linhas de código e qualidade das instâncias geradas do KDM
Contexto	Acadêmico
Questões de Pesquisa	**RQ1:**"Quais são os pros e contras do uso da DCL-KDM e do KDM-SDK para especificar PAs?" e **RQ2:**"A DCL-KDM contribui mais para a corretude e qualidade das instâncias geradas do KDM em comparação com o KDM-SDK?"

Tabela 6.2: GQM para a segunda avaliação da DCL-KDM

Item	Descrição
Objeto do Estudo	DCL-KDM
Objetivo / Propósito	Comparar a DCL-KDM com outras três técnicas disponíveis na literatura (DCL (Terra; Valente, 2009), Matriz de Dependência (DM - *Dependency Matrix*) da ferramenta JArchitect (Jarchitect, 2016) e Reflexão de Modelos (RM - *Reflexion Models*) da ferramenta SAVE (Duszynski; Knodel; Lindvall, 2009))).
Perspectiva	Arquitetos de Software que necessitam escolher uma DSL para especificar uma Arquitetura Planejada (PA) para qualquer projeto de modernização de software
Foco de Qualidade	Especificação da PA, o uso da ferramenta de suporte e o escopo da especificação
Contexto	Sistemas de software heterogêneos
Questões de Pesquisa	**RQ3:**"Em termos de prós e contras como é a especificação e uso em DCL-KDM comparada com as outras três técnicas?" e **RQ4:**"A DCL-KDM pode ser usada como linguagem para especificação de PAs como as demais técnicas da literatura em qualquer contexto ou apenas para projetos baseados em ADM?"

6.2.1 Primeira Avaliação da DCL-KDM

Para conduzir essa primeira avaliação, utilizou-se uma Arquitetura Planejada (PA) hipotética que pode ser observada na Figura 6.1. Essa PA foi utilizada por conter um nível de variedade de abstrações e combinações arquiteturais consideravelmente alta. Outro motivo de sua utilização é o fato de não ser comum que sistemas reais possuam uma variedade de abstrações e combinações arquiteturais, que necessitavam ser avaliadas, e consequentemente, não foi possível encontrar um que se enquadra-se no perfil. Dessa forma, pode-se afirmar que essa PA é mais adequada para exercitar a DCL-KDM do que um sistema real com poucas combinações arquiteturais.

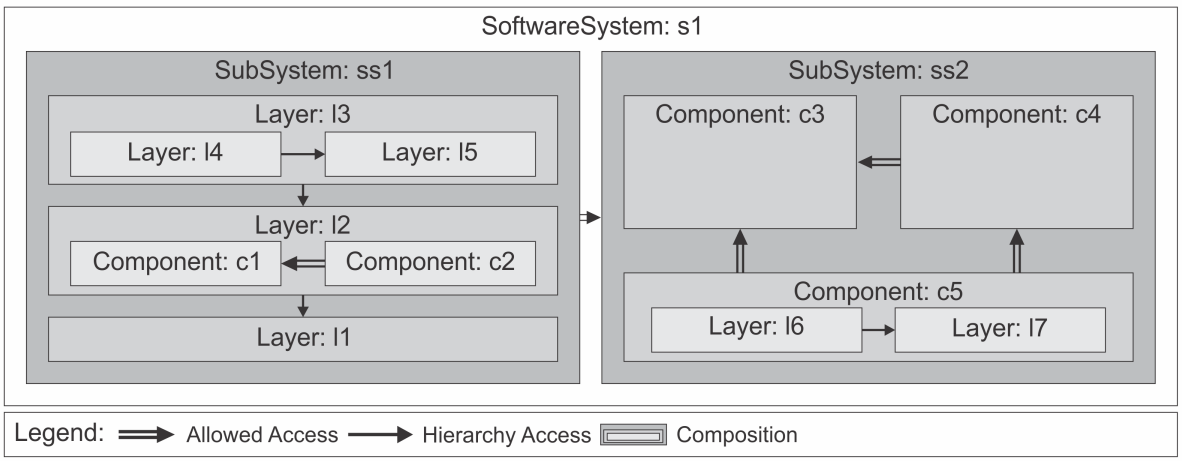

Figura 6.1: Arquitetura Hipotética para a primeira avaliação da DCL-KDM. Fonte: Elaborado pelo autor

Para a execução da avaliação, selecionou-se dois arquitetos de software com conhecimentos sobre KDM e ADM entre intermediário e avançado. Portanto, solicitou-se aos arquitetos que cada um realiza-se a especificação arquitetural utilizando-se a DCL-KDM e a API para o Eclipse KDM-SDK. A API KDM-SDK é um conjunto de classes em Java que representam o metamodelo KDM e dão suporte à geração manual de uma instância em Java.

Baseando-se na Figura 6.1, foi solicitado aos arquitetos que realizassem a especificação de três partes da arquitetura. A primeira parte é composta pelas camadas `l1`, `l2` e `l3` e suas restrições arquiteturais. A segunda parte é composta pelos componentes `c3`, `c4` e `c5` e suas restrições arquiteturais. Por fim, a última parte é composta pelo componente `c1` e `c2` dentro da camada `l2` e suas restrições arquiteturais. No Código 6.1 pode ser observado a especificação dessas três partes solicitadas em DCL-KDM. A utilização do KDM-SDK com Java gera um código maior que a utilização da DCL-KDM, portanto, no Código 6.2 pode ser observado apenas a especificação da primeira parte solicitada.

```
1 architeturalElements {
2     subSystem ss1;
3     subSystem ss2;
4     layer l3, level 3, inSubSystem: ss1;
5     layer l2, level 2, inSubSystem: ss1;
6     layer l1, level 1, inSubSystem: ss1;
7
8     component c1, inLayer: l2;
9     component c2, inLayer: l2;
10
11    component c3, inSubSystem: ss2;
12    component c4, inSubSystem: ss2;
13    component c5, inSubSystem: ss2;
14 }restrictions {
15    c2 can-depend-only c1;
16    c4 can-depend-only c3;
17    c5 can-depend c4;
18    c5 can-depend c3;
19
20    ss1 can-depend ss2;
21 }
```

Código 6.1: Especificação das três partes solicitadas em DCL-KDM

```
1  private void createFirstPartToEvaluated() {
2      Subsystem ss1 = StructureFactory.eINSTANCE.createSubsystem();
3      ss1.setName("ss1");
4      Layer l3 =StructureFactory.eINSTANCE.createLayer();
5      l3.setName("l3");
6      Layer l2 =StructureFactory.eINSTANCE.createLayer();
7      l2.setName("l2");
8      Layer l1 =StructureFactory.eINSTANCE.createLayer();
9      l1.setName("l1");
10     ss1.getStructureElement().add(l3);
11     ss1.getStructureElement().add(l2);
12     ss1.getStructureElement().add(l1);
13     List<KDMRelationship> lisfOfRelationships = new ArrayList<KDMRelationship>();
14     Calls relation = ActionFactory.eINSTANCE.createCalls();
15     lisfOfRelationships.add(relation);
16     [...]
17     AggregatedRelationship newRelationship = CoreFactory.eINSTANCE.
           createAggregatedRelationship();
18     newRelationship.setDensity(lisfOfRelationships.size());
19     newRelationship.setFrom(l3);
20     newRelationship.setTo(l2);
21     newRelationship.getRelation().addAll(lisfOfRelationships);
22     l3.getAggregated().add(newRelationship);
23     [...]
24 }
```

Código 6.2: Código da primeira parte da avaliação em KDM-SDK

Realizando-se uma comparação entre ambas as especificações tem-se que a linha 2 do

Código 6.1 e a linha 2-3 do Código 6.2 realizam a mesma função, declarar e instanciar um subsistema denominado `ss1`. As linhas 4-6 do Código 6.1 e as 4-12 do Código 6.2 declaram as camadas `l1`, `l2` e `l3`. Por fim, as linhas 13-22 do Código 6.2 criam os relacionamentos entre as camadas, que comparada a DCL-KDM não é necessário sua especificação devido a geração automática. É importante ressaltar que ambas as especificações geram a mesma instância do KDM, no Código 6.3 pode ser observado a mesma parte comparada anteriormente em XMI. Por exemplo, a linha 2 do Código 6.1, a linha 2-3 do Código 6.2 e a linha 4 do Código 6.3 representam a declaração do subsistema `ss1`.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <kdm:Segment xmi:version="2.0" [...] name="Planned Architecture">
3   <model xsi:type="structure:StructureModel" name="Planned Architecture">
4     <structureElement xsi:type="structure:Subsystem" name="ss1">
5       <structureElement xsi:type="structure:Layer" name="l3"  [...]>
6         <aggregated from="//@model.0/@structureElement.0/@structureElement.0"
7             to="//@model.0/@structureElement.0/@structureElement.1"
8             relation="//@model.1/@codeElement.0/@codeElement.0/@actionRelation.0 [...]"
9             density="7"/>
10        [...]
11      </structureElement>
12      <structureElement xsi:type="structure:Layer" name="l2" [...]>
13        <aggregated from="//@model.0/@structureElement.0/@structureElement.1"
14            to="//@model.0/@structureElement.0/@structureElement.2"
15            relation="//@model.1/@codeElement.1/@codeElement.0/@actionRelation.0 [...]"
16            density="7"/>
17        [...]
18      </structureElement>
19      <structureElement xsi:type="structure:Layer" name="l1" [...]/>
20    </structureElement>
21  </model>
22  <model xsi:type="code:CodeModel"> [...] </model>
23 </kdm:Segment>
```

Código 6.3: Parte da instância do KDM gerada por ambas as especificações

Para essa primeira avaliação, focou-se na métrica Linhas de Código (LoC - `Lines of Code`). Essa métrica apesar de subjetiva é uma métrica fácil e rápida para obter-se um parametro de produtividade. Dessa maneira, foi escolhida para prover de forma empírica uma estimativa inicial de esforço para a criação de uma mesma especificação nas duas abordagens, a DCL-KDM e KDM-SDK, para os dois arquitetos. Na Figura 6.2 podem ser observados os resultados da especificação de cada parte proposta para cada um dos arquitetos de software convidados.

Analisando as informações demonstradas e a utilização pelos arquitetos, é possível responder ambas as questões de pesquisa estipuladas para essa avaliação. Para a primeira questão de pesquisa, elaborou-se a Tabela 6.3 sumarizando os pontos favoráveis e desfavoráveis da DCL-

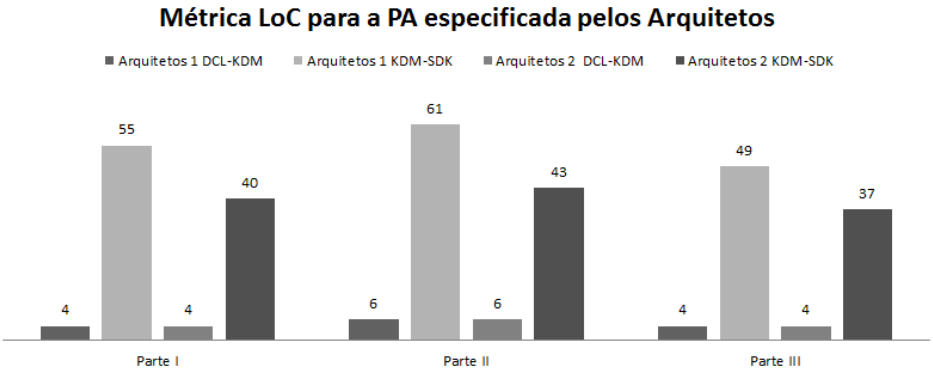

Figura 6.2: Resultado em termos de LoC para a PA especificada por arquiteto de software

KDM.

Tabela 6.3: Pontos favoráveis e desfavoráveis da DCL-KDM

Pontos Favoráveis	Pontos Desfavoráveis
A DCL-KDM instancia e serializa uma especificação automaticamente em KDM, tornando transparente para o arquiteto de software detalhes de instanciação, serialização e restrições do metamodelo.	A DCL-KDM oferece apenas um conjunto fixo de elementos arquiteturais, o que era esperado, uma vez que apesar de utilizar as metaclasses do pacote *Structure* do KDM, o KDM-SDK é uma API possibilitando, por meio de herança, a criação de quantos tipos de elementos arquiteturais forem necessários.
A DCL-KDM possibilita a especificação de um número consideravelmente alto de restrições arquiteturais entre elementos suportados pela DSL.	Na DCL-KDM existe um número restrito de possibilidades de realizar composições entre elementos arquiteturais, bem como a hierarquização entre eles, o que não existe no KDM-SDK.
Apesar da abordagem ser voltada e baseada na ADM e no KDM, o arquiteto de software que a estiver utilizando não precisa possuir conhecimentos avançados de ADM e do KDM, uma vez que sua utilização é simples.	O KDM-SDK não possui padronização de utilização, dessa forma a mesma especificação pode ser escrita de diversas formas diferentes, conforme pode ser observado no LoC para KDM-SDK na Figura 6.2. Dessa mesma forma se comporta a especificação de restrições arquiteturais, fatos tais quais podem não ser errados ou ruins, uma vez que fica nas mãos do arquiteto de software e seu conhecimento, mas dão liberdade ao arquiteto de especificar conforme deseja.
A DCL-KDM possibilita a geração automática de restrições arquiteturais pré estabelecidos como por exemplo, camadas.	

Para a segunda questão de pesquisa, observou-se no estudo empírico que a quantidade de erros e correções observados durante a especificação dos arquitetos diminui na utilização da DCL-KDM, enquanto a qualidade se mantém igual. A diminuição de erros e, consequentemente, correções durante a especificação se deram por duas principais razões; a primeira é o fato da responsabilidade de definição de restrições arquiteturais ser removida da responsabilidade do arquiteto de software, por exemplo, as restrições arquiteturais de camadas que não precisam ser especificadas, uma vez que são geradas automaticamente. A segunda razão é o fato de não possibilitar múltiplas formas de especificação. Como pode ser observado nos valores de Loc na Figura 6.2, cada arquiteto tem seu próprio estilo e lógica, tornando cada especificação pessoal e única. Fato que em DCL-KDM é evitado já que existe apenas uma forma de realizar a especificação e esta é suportada por uma gramática, evitando erros durante a especificação.

Quanto a qualidade se manter a mesma é devido as seguintes razões: i) a DCL-KDM encapsula em sua geração a utilização do KDM-SDK, tornando mais simples a geração, porém, mantendo a mesma qualidade da instância e ; ii) ambos os arquitetos que participaram desse estudo empírico tinham conhecimentos avançados de KDM e de sua serialização, mantendo assim um alto índice de qualidade da instância, fato que pode ser alterado caso o arquiteto não tenha conhecimentos avançados do metamodelo.

6.2.2 Segunda Avaliação da DCL-KDM

Para conduzir essa segunda avaliação, utilizou-se o mesmo sistema apresentado para o cenário de uso, que consiste basicamente no sistema apresentado no Capítulo 5 Seção 5.3 e sua arquitetura pode ser observada na Figura 5.1 (Capítulo 5 - Seção 5.4).

Essa avaliação compreende a especificação da arquitetura do sistema *myAppointments* na DCL-KDM (Landi et al., 2017), na DCL (Terra; Valente, 2009), em DM (Jarchitect, 2016) e em RM (Duszynski; Knodel; Lindvall, 2009). No Código 6.4 pode ser observado a especificação em DCL-KDM. No Código 6.5 pode ser observado a especificação em DCL. Na Figura 6.3 pode ser observado a especificação em DM utilizando-se a ferramenta JArchitect5. Por fim, na Figura 6.4 pode ser observado a especificação em RM utilizando-se a ferramenta SAVE.

```
1 architeturalElements{
2     subSystem myAppointments;
3
4     layer view, level 3, inSubSystem: myAppointments;
5
6     layer controller, level 2, inSubSystem: myAppointments;
7
8     layer model, level 1, inSubSystem: myAppointments;
9     layer dao, level 2, inLayer: model;
```

```
10      layer domain, level 1, inLayer: model;
11
12      component util, inSubSystem: myAppointments;
13
14  }restrictions{
15      view can-depend util;
16
17      controller can-depend util;
18
19      model can-depend util;
20      dao can-depend util;
21      domain can-depend util;
22
23      controller can-depend view;
24      controller can-depend dao;
25      controller can-depend domain;
26  }
```

Código 6.4: Especificação do myAppointment em DCL-KDM (Landi et al., 2017)

```
1  %Modules
2  module Controller:    myapp.controller.*
3  module View:          myapp.view.*
4  module Model:         myapp.model.**
5  module Domain:        myapp.model.domain.*
6  module Util:          myapp.util.*
7  module DAO:           "myapp.model.[a-zA-Z0-9/.]*DAO"
8  module JavaAwtSwing:  java.awt.**, javax.swing.**
9  module JavaSql:       java.sql.**
10 %Constraints
11 only View can-depend JavaAwtSwing
12 only DAO can-depend JavaSql
13 View cannot-depend Model
14 Domain can-depend-only $java
15 DAO can-depend-only Domain, Util, javaSql
16 Util cannot-depend $system
```

Código 6.5: Especificação do myAppointment em DCL (Terra; Valente, 2009)

Essa avaliação foi conduzida empiricamente focando apenas no uso de cada abordagem e em seus pontos favoráveis e desfavoráveis. Analisando os dados obtidos nos Códigos 6.4 e 6.5 e nas Figuras 6.3 e 6.4, é possível responder ambas as questões de pesquisa. Para responder a questão de pesquisa **RQ3** separou-se observações a respeito de cada uma das quatro técnicas apresentadas.

- **DCL-KDM**: Como ponto favorável, a DCL-KDM, comparadas as outras abordagens analisadas pode ser considerada intuitiva baseando-se nos auxílios proporcionados pelo apoio ferramental como o *autocomplete* de código. Dessa forma, possibilitando que o arquiteto de software, após entender a sintaxe da linguagem, consiga especificar qualquer PA que

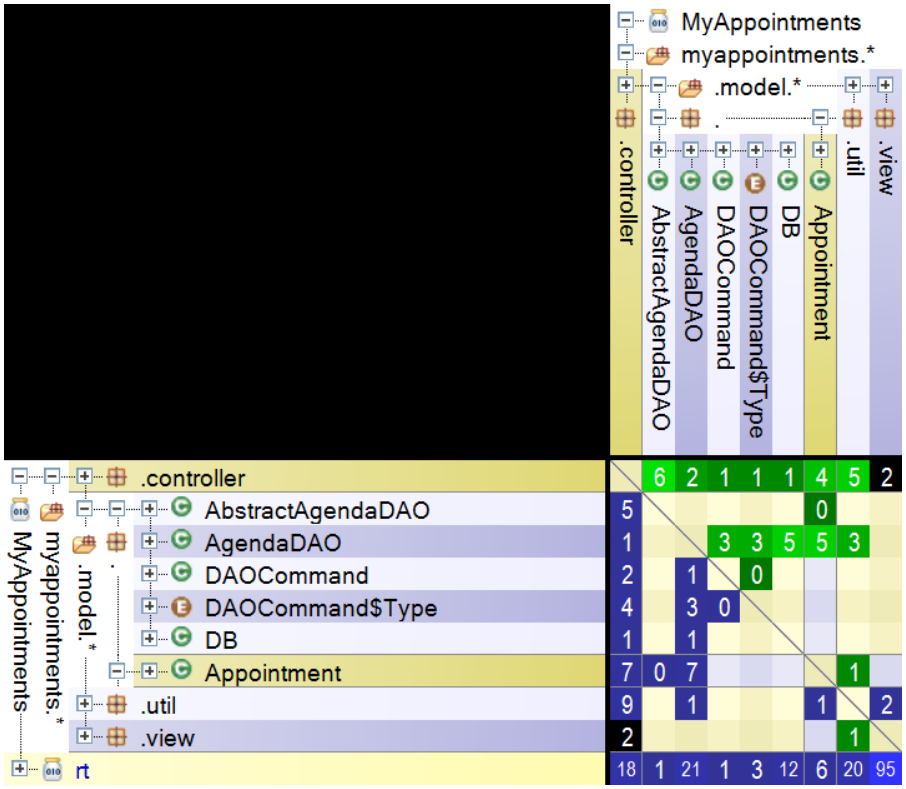

Figura 6.3: Especificação do myAppointment em DM (Jarchitect, 2016). Fonte: Elaborado pelo autor

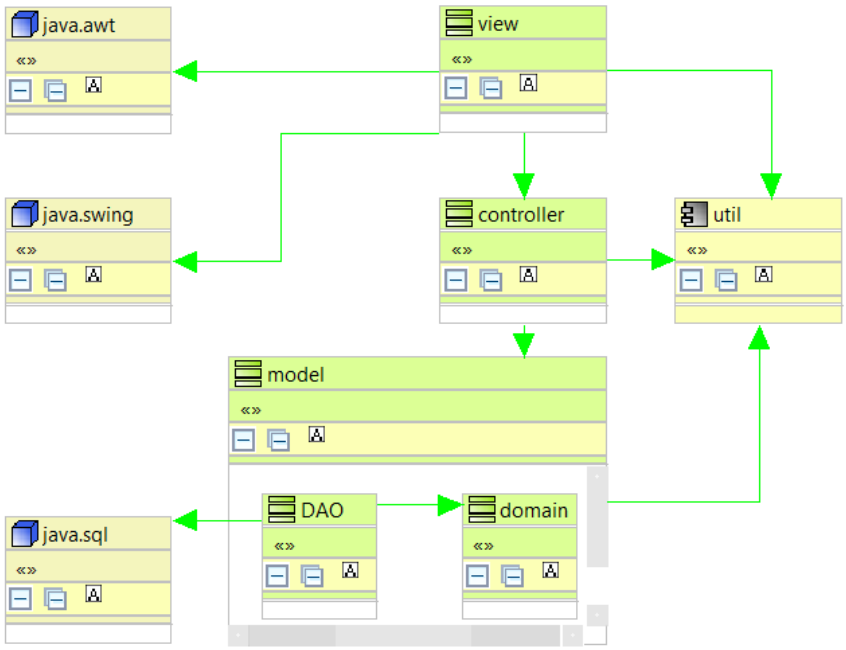

Figura 6.4: Especificação do myAppointment em RM (Duszynski; Knodel; Lindvall, 2009). Fonte: Elaborado pelo autor

necessite. Como ponto desfavorável tem-se o fato de que não foi possível especificar todas as restrições arquiteturais propostas pelos idealizadores do sistema.

- **DM**: Como ponto favorável a DM, sua visualização é compacta e possibilita uma visão geral e rápida do sistema e seus relacionamentos. Como ponto desfavorável, notou-se que as regras de linguagem da DM foram insuficientes para representar todas as restrições arquiteturais do sistema como, por exemplo, as baseadas em padrões de nomeclatura. Outro ponto extremamente desfavorável é o fato da DM necessitar intrinsecamente do código do sistema, levando a concluir que este tipo de abordagem dificulta a especificação e uso da DM na geração e serialização de PA.

- **DCL**: A especificação em DCL, é muito similar a especificação em DCL-KDM, o que é previsível, uma vez que a DCL-KDM foi baseada na DCL. Dessa forma, os pontos favoráveis e desfavoráveis de ambas são iguais. Entretanto, observou-se três diferenças interessantes entre ambas as abordagens. A primeira é a possibilidade de que na DCL-KDM é possível realizar a diferenciação entre elementos arquiteturais, o que não ocorre na DCL, que caracteriza todos os elementos arquiteturais como módulos. A segunda é a forma como o arquiteto de software observa a especificação do sistema, enquanto na DCL-KDM, a princípio, todas as comunicações entre elementos arquiteturais são proibidas e deve-se especificar as permissões de acesso, na DCL é o oposto. Por fim, a terceira diferença é quanto ao código do sistema, a DCL e a DM necessitam que o sistema tenha o código já escrito ou muito bem definido de forma que em sua especificação já seja realizado o mapeamento, já a DCL-KDM e a RM não necessitam dessa definição de código fonte.

- **RM**: Dentre as quatro técnicas escolhidas para a avaliação, o RM em conjunto com a ferramenta SAVE é a única que da suporte a um processo de Checagem de Conformidade Arquitetura por meios gráficos, o que é um grande ponto favorável a esta abordagem. O uso da ferramenta é intuitivo e gera diagramas de alto nível, como o da Figura 6.4. Entretanto, a ferramenta é proprietária, não estando aberta diretamente ao público e, consequentemente, utilizando modelos e metamodelos proprietários. Dessa forma, tendo seu principal ponto desfavorável com o fato de ser proprietário, ferindo assim os princípios da ADM de utilizar metamodelos padrões que facilitem a reusabilidade.

Realizando essa avaliação e observando o uso e resultados obtidos para as quatro técnicas da literatura escolhidas, conclui-se dois fatos que dão suporte à resposta para a questão de pesquisa **RQ4**. O primeiro fato é que a DCL-KDM, apesar de algumas limitações comparadas às demais técnicas, é uma opção aceitável e viável como escolha de abordagem para especificação de arquiteturas planejadas. O segundo fato é que a base da DSL, ou seja, a gramática da DCL-KDM pode ser convertida para qualquer outra técnica apenas criando novas conversões de

serialização, por exemplo outros metamodelos como o do UML, ou até mesmo o utilizado pela ferramenta SAVE. Baseando-se nesses dois fatos, conclui-se que a resposta para a pergunta "A DCL-KDM pode ser usada como linguagem para especificação de PAs como as demais técnicas da literatura em qualquer contexto, ou apenas para projetos baseados em ADM?" é sim, a DCL-KDM pode ser utilizada em outros contextos. Apesar da DCL-KDM ter sido gerada e baseada na ADM e respeitar o KDM, a DSL criada pode ser utilizada para a especificação de Arquiteturas Planejadas, independente do contexto do sistema e baseando-se nas suas características, também pode ser estendida e de forma reconhecer novos elementos e gerar instâncias de outros metamodelos diferentes do KDM.

6.3 Avaliação da Arch-KDM 2.0

Nessa seção será descrita a avaliação realizada na Arch-KDM 2.0. Uma vez que o principal objetivo da Arch-KDM 2.0 seja a identificação de violações arquiteturais e a descoberta dos desvios arquiteturais, o principal foco da avaliação é medir a precisão com o qual os desvios arquiteturais são descobertos. Para tal medição utilizou-se três métricas, a primeira é a precisão baseada nos desvios recuperados corretamente, a segunda é o *recall* baseado na porcentagem de desvios recuperados corretamente sobre o total de desvios do sistema e, por fim, a *f-measure* baseado na exatidão da recuperação realizando a ponderação entre a precisão e o *recall* (Makhoul et al., 1999; Landgrebe; Paclik; Duin, 2006; Roncero, 2010; Perez-castillo et al., 2011). Para realizar o cálculo das métricas escolhidas, foi necessário definir alguns termos relevantes que podem ser observados na Tabela 6.4.

Tabela 6.4: Termos utilizados para o cálculo das métricas de avaliação

Termo	Descrição
Ground Truth (GT)	Desvios arquiteturais que foram localizados pelo especialista de forma manual
True Positive (TP)	Desvios arquiteturais que foram descobertos pelo apoio computacional e estão presentes no GT
False Negative (FN)	Desvios arquiteturais que não foram descobertos pelo apoio computacional e estão presentes no GT
False Positive (FP)	Desvios arquiteturais que foram descobertos pelo apoio computacional e não estão presentes no GT

A métrica da precisão é baseada na descoberta de desvios corretamente. Durante a pesquisa, encontraram-se poucas referências na literatura definindo um valor mínimo que determine se um

sistema obteve uma alta, média ou baixa precisão. Entretanto, a presente dissertação se baseou nos resultados obtidos por Perez-Castillo et al. (2011) que definem valores em uma escala iniciando em muito baixa chegando a muito alta para a precisão de recuperação de informações de processos. Apesar de serem contextos diferentes, o autor deste trabalho julgou serem plausíveis de utilização devido ao fato da escala estar avaliando a precisão da recuperação e não seu contexto. Portanto, na Tabela 6.5 pode ser observado os parâmetros de definição apresentados por Perez-Castillo et al. (2011).

Tabela 6.5: Escala de nível de precisão para recuperação de informações. Fonte: Perez-Castillo et al. (2011)

Precisão	**Nível**
Precisão < 0.47	Muito Baixa
$0.47 <$ Precisão < 0.56	Baixa
$0.56 <$ Precisão < 0.63	Média
$0.63 <$ Precisão < 0.72	Alta
$0.72 <$ Precisão	Muito Alta

É importante salientar que esse tipo de avaliação é realizada de forma subjetiva e em conjunto com o(s) especialista(s) do domínio. A colaboração de especialistas é importante nesse tipo de avaliação pois ajuda a consolidar os resultados encontrados e a resolver situações de conflito. Para o cálculo da precisão baseado nos termos definidos anteriormente, foi utilizada a equação 6.1

$$Precisao(P) = \frac{TP}{GT + FP} \tag{6.1}$$

A métrica de abrangência (*recall*), é baseada no quanto o apoio computacional recupera de desvios arquiteturais descartáveis, ou seja, baseada na quantidade de desvios corretos descobertos sobre a quantidade de desvios incorretos recuperados. Para o cálculo de abrangência baseado nos termos definidos anteriormente é utilizada a equação 6.2

$$Abrangencia(R - Recall) = \frac{TP}{GT + FN} \tag{6.2}$$

Por fim, a métrica *f-measure* baseia-se nos valores de precisão e abrangência para realizar uma ponderação por meio da equação 6.3

$$f\text{-}measure(F) = \frac{2 \cdot P \cdot R}{P + R} \tag{6.3}$$

Nas subseções seguintes serão apresentados a metodologia e resultados da avaliação.

6.3.1 Metodologia aplicada na avaliação da Arch-KDM 2.0

Para esta avaliação, o primeiro passo realizado foi a escolha de um sistema a ser avaliado. O sistema escolhido foi *LabSys* (*Laboratory System*). Esse sistema foi implementado por engenheiros de software da Universidade Federal do Tocantins (UFT) e foi disponibilizado pelos seus desenvolvedores. Sua principal função é realizar o gerenciamento de informações e do uso de laboratórios da universidade. Esse sistema foi desenvolvido utilizando estilos arquiteturais diversos. O sistema não possuía uma arquitetura planejada, portanto, com o auxílio do código, da documentação e dos autores foi possível elaborar e definir sua arquitetura. Na Figura 6.5 pode ser observada a arquitetura planejada do sistema.

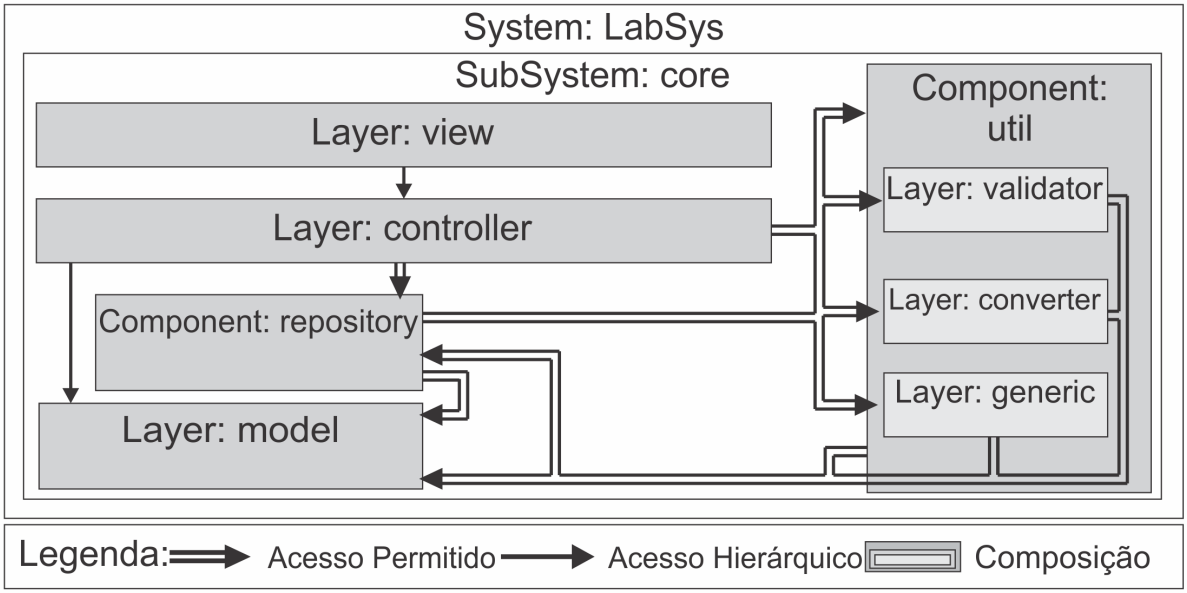

Figura 6.5: Arquitetura Planejada do Sistema *LabSys*. Fonte: Elaborado pelo autor

O segundo passo da avaliação foi a realização do reconhecimento manual dos desvios arquiteturais, gerando um oráculo para o sistema. O reconhecimento visou analisar todo o código fonte do sistema para encontrar manualmente os desvios arquiteturais. Essa análise foi executada manualmente seguindo três atividades e as repetindo duas vezes para cada classe do sistema, sendo as atividades: i) analisar linha por linha da classe; ii) caso a linha realize alguma referência a um objeto ou item que não seja da própria classe, verificar o destino, armazenar este par de origem e destino; iii) checar se o par armazenado é permitido ou não na arquitetura

planejada, em caso de não ser permitido, anotar numa planilha este desvio arquitetural.

O terceiro passo da avaliação foi a realização do reconhecimento automático dos desvios arquiteturais por meio do uso da Arch-KDM 2.0. Depois de realizado o reconhecimento automático, os resultados obtidos foram comparados com os resultados do oráculo. Com o auxilio das Tabelas A.1, A.2, A.3, A.4, A.5, A.6 e A.8 (Apêndice A) e das informações geradas pelo apoio computacional é possível reconhecer as linhas de código que foram definidas como os desvios arquiteturais. Dessa forma, foi possível reconhecer qual a linha de código e, consequentemente, compará-la com o oráculo para verificar sua identificação conforme os termos apresentados na Tabela 6.4.

O quarto passo da avaliação foi a validação dos resultados, com o objetivo de avaliar a interpretação do especialista na realização do oráculo e comparação com os desvios gerados manualmente, utilizou-se o Método do Júri (Fonseca et al., 2007; Matos, 2014). Este método consiste na utilização de um júri, formado de juízes que, analisam individualmente os resultados propostos pelo especialista. Depois da análise dos juízes foi verificado se houve a "concordância entre juízes". Segundo Matos (2014), a forma mais simples e prática de calcular a concordância é a utilização da Porcentagem de Concordância Absoluta (PAA - *Percentage of Absolute Agreement*). Esse cálculo é realizado dividindo o número de vezes que os jurados concordam pela quantidade total de itens avaliados. Para alguns autores, segundo Matos (2014), o valor mínimo aceitável para este tipo de avaliação é de 75% sendo que valores com mais de 90% são considerados altos.

A realização do Método do Júri foi dividida em quatro atividades. A primeira atividade foi a escolha de quatro voluntários para compor a mesa de jurados. Três voluntários tinham a atividade de realizar a avaliação e um voluntário a atividade de solucionar e avaliar em caso de desacordo entre os outros jurados. Os voluntários escolhidos são especialistas na área de computação e desenvolvimento de software. Seus nomes não serão divulgados por confidencialidade, porém, seus perfis compreendem i) especialista em desenvolvimento de software, mestrando em engenharia de software com ênfase em banco de dados e desenvolvedor para uma empresa multinacional do ramo da aviação. ii) especialista em desenvolvimento de software e mestrando em ciências da computação com ênfase em modernização de software; iii) especialista em privacidade e professor da Pontifícia Universidade Católica de Minas Gerais; iv) especialista em desenvolvimento de software e doutorando em ciências da computação com ênfase em modernização de software;

A segunda atividade foi a entrega e assinatura do "Termo de Consentimento Livre e Esclarecido" para que os jurados concordassem e registrassem suas participações na avaliação. A

terceira atividade foi a entrega dos documentos necessários aos jurados para sua participação, contendo o código do sistema analisado, o oráculo desenvolvido pelo especialista e o resultado gerado automaticamente pelo apoio computacional.

A quarta atividade foi a solicitação aos jurados para fazer uma análise do código do sistema e comparar com a arquitetura planejada do sistema, de forma que classificassem os desvios arquiteturais encontrados pelo especialista e pelo apoio ferramental. Depois de realizada sua avaliação, definiu-se ao especialista a avaliação de três itens em duas categorias, sendo elas "Concordo" se aprova-se a avaliação realizada pelo especialista e "Não Concordo" se reprova-se a avaliação realizada pelo especialista. Os itens definidos foram: i) desvios encontrados manualmente; ii) identificação de qual desvio encontrado manualmente representa o desvio encontrado automaticamente e; iii) a avaliação como um todo.

Por fim, o ultimo passo da avaliação foi a realização da análise dos resultados. Na próxima seção são apresentados os resultados obtidos ao aplicar a avaliação descrita nesta Seção.

6.3.2 Resultados obtidos na avaliação

Na Tabela 6.6 pode ser observado o resultado do segundo passo da avaliação. A tabela forma uma matriz com a quantidade de desvios encontrados manualmente entre os elementos arquiteturais da arquitetura planejada.

Tabela 6.6: Resultados da identificação manual de desvios arquiteturais do *LabSys*

	view	**controller**	**repository**	**model**	**util**	**validator**	**converter**	**generic**
view	0	0	0	0	0	0	0	0
controller	0	0	0	0	0	0	0	0
repository	0	0	0	0	0	0	0	0
model	0	0	0	0	41	0	24	0
util	0	0	0	0	0	0	0	0
validator	0	0	0	0	52	0	0	0
converter	0	0	0	0	0	0	0	0
generic	0	0	0	0	2	0	0	0

Os valores obtidos para a elaboração da Tabela 6.6 foram retirados da execução manual do processo de reconhecimento de desvios, conforme explicado anteriormente. Na Tabela 6.7 podem ser observadas três linhas do oráculo gerado. O oráculo completo pode ser acessado no link "https://www.dropbox.com/s/osvmd90u0injmf8/Or%C3%A1culoLabSys.xlsx?dl=0".

Tabela 6.7: Exemplo do oráculo do *LabSys*

Elemento Origem	Linha	Elemento Destino	Linha	Tipo
`Block.java`	3	`EntityConverter.java`	N/A	importação de classe
`Campus.java`	20	`EntityConverter.java`	N/A	implementação de classe
`Course.java`	98	`CoursePeriod.java`	N/A	retorno de método

Na Tabela 6.8 pode ser observado o resultado do terceiro passo da avaliação. A tabela forma uma matriz com a quantidade de desvios encontrados automaticamente entre os elementos arquiteturais da arquitetura planejada.

Tabela 6.8: Resultados da identificação automática de desvios arquiteturais do *LabSys*

	view	**controller**	**repository**	**model**	**util**	**validator**	**converter**	**generic**
view	0	0	0	0	0	0	0	0
controller	0	0	0	0	0	0	0	0
repository	0	0	0	0	0	0	0	0
model	0	0	0	0	41	0	24	0
util	0	0	0	0	0	0	0	0
validator	0	0	0	0	33	0	0	0
converter	0	0	0	0	0	0	0	0
generic	0	0	0	0	0	0	0	0

Sumarizando os dados encontrados em ambas as as identificações, o oráculo manual encontrou um total de 119 desvios arquiteturais, enquanto o apoio computacional encontrou 94 desvios arquiteturais. Na Tabela 6.9 pode ser observado o resultado do quarto passo da avaliação. Na Tabela 6.9 é possível observar que os jurados numerados de 1 a 3 concordaram e validaram a execução da avaliação. Dessa forma, foi possível realizar o último passo da avaliação.

Tabela 6.9: Resultado dos jurados por quesito de avaliação

Item avaliado	Jurado 1	Jurado 2	Jurado 3
Identificação Manual	Concordo	Concordo	Concordo
Relação de desvios: Manual X Auto.	Concordo	Concordo	Concordo
Avaliação	Concordo	Concordo	Concordo

Conforme apresentado anteriormente, utilizou-se as métricas de precisão, *recall* e *f-measure*. Na Tabela 6.10 podem ser observados os valores encontrados para cada um dos quatro termos necessários para o cálculo das métricas.

Tabela 6.10: Valores para os termos utilizados para o cálculo das métricas de avaliação

Termo	Valor
Ground Truth (GT)	119
True Positive (TP)	94
False Negative (FN)	25
False Positive (FP)	0

A partir desses valores, é possível a substituição nas equações de calculo e realizar a identificação dos valores para cada métrica conforme observado nas Equações 6.4, 6.5 e 6.6.

$$Precisao(P) = \frac{TP}{GT + FP} = \frac{94}{119 + 0} = \frac{94}{119} = 0,7899 \tag{6.4}$$

$$Abrangencia(R - Recall) = \frac{TP}{GT + FN} = \frac{94}{119 + 25} = \frac{94}{144} = 0,6527 \tag{6.5}$$

$$f\text{-}measure(F) = \frac{2 \cdot P \cdot R}{P + R} = \frac{2 \cdot 0,7899 \cdot 0,6527}{0,7899 + 0,6527} = \frac{1,0311}{1,4426} = 0,7147 \tag{6.6}$$

Considerando os desvios arquiteturais recuperados (TP), como observado, o valor da precisão dado pela Equação 6.4 foi de 0,7899 representando uma precisão de 78,99%. Utilizando-se dos valores estipulados e propostos por Perez-Castillo et al. (2011) (Tabela 6.5), pode-se afirmar que a técnica e o apoio computacional desenvolvidos podem ser utilizados com um certo grau de confiança, uma vez que atingiu o patamar de precisão muito alta segundo os autores.

O cálculo do *recall* é a relação dos desvios encontrados sobre todos os desvios e os falsos

negativos. Como observado, o valor do *recall* dado pela Equação 6.5 foi de 0,6527 representando 65,27%. Isso significa que de todos os desvios arquiteturais, foram recuperadas 65,27%.

Por fim, tem-se a métrica *f-measure* utilizada para avaliar a exatidão realizando uma ponderação entre os valores de precisão e *recall*. Conforme observado na Equação 6.6 o valor de resultado é 0,7147 representando 71,47%. Essa métrica é um indicativo de desempenho de forma que quanto mais próximo do total for o resultado, ou seja, de 100%, melhor é o desempenho do objeto em análise (Roncero, 2010).

Analisando-se as métricas obtidas, é possível afirmar que a abordagem e o apoio computacional desenvolvido possuem resultados positivos e muito promissores. Analisando-se a fundo os desvios arquiteturais não encontrados pelo apoio computacional, foi possível observar que o algoritmo contém algumas falhas quanto a elementos de código representados no pacote *Code* do KDM. No caso especifico dessa avaliação, observou que os desvios não encontrados automaticamente foram ocasionados pelo fato de serem elementos de composição da metaclasse `TemplateUnit`. Acredita-se que, evoluindo o algoritmo para os outros elementos não reconhecidos, é possível atingir uma precisão de 100%.

6.4 Ameaças a validade

Em ambas as avaliações, foram elencadas duas ameças a validade. A primeira é que, apesar de demonstrar o uso do apoio computacional com dois sistemas (*myAppointments* e *LabSys*) e avaliar com um sistema, não é possível afirmar que a abordagem irá prover resultados equivalentes em outros sistemas, como de costume em estudos empíricos na engenharia de software (validação externa). A segunda ameaça é a dependência de engenheiros de software para avaliar os resultados. Apesar de os resultados terem sido avaliados por meio da aplicação do Método do Júri, como é possível em avaliações com humanos, os resultados podem ter sido afetados por algum grau de subjetividade (validação de construção).

Como contramedida para essas duas ameaças, será realizado um experimento comparando a utilização da ferramenta proposta por essa abordagem e outras ferramentas. Foi encontrada e analisada a possibilidade de utilização de ferramentas proprietárias, porém, não foi possível utilizá-las devido ao fato de serem ferramentas pagas. Também foram encontradas ferramentas abertas à comunidade e estão em processo de análise para sua utilização. Outra contramedida que está em análise é o contato com empresas de TI para a realização de experimentos com softwares reais de médio e/ou grande porte em Java para averiguar como a abordagem se comporta no âmbito empresarial, uma vez que foram averiguados apenas o comportamento em

ambiente acadêmico.

6.5 Considerações Finais

Neste capítulo foram apresentadas três avaliações realizadas durante este trabalho. As primeiras duas avaliações foram realizadas com o intuito de avaliar a DCL-KDM. Uma das avaliações realizadas foi a comparação da DCL-KDM com a API Java KDM-SDK. Nessa avaliação observou-se atributos como o código gerado e a qualidade da instância do metamodelo. Dessa forma, foi possível concluir que a DCL-KDM condiz com seu propósito em gerar instâncias do KDM com qualidade e de forma menos trabalhosa que a alternativa atual. Outra avaliação realizada visou comparar a DCL-KDM com outras três abordagens e ferramentas da literatura. Com os resultados da comparação, observou-se atributos como facilidade de uso e o resultado obtido da utilização das abordagens. Sendo assim, é possível concluir que a DCL-KDM comparada com as demais abordagens possui vantagens e desvantagens porém nenhuma é impactante o suficiente para inviabilizar sua utilização como uma forma de definir Arquiteturas Planejadas.

A última avaliação foi realizada com o intuito de avaliar a Arch-KDM 2.0. Nessa avaliação observou-se os atributos de qualidade de identificação automático precisão, abrangência e `f-measure`. Com os resultado obtidos, foi possível concluir que a abordagem possui uma precisão muito alta e que depois de novas evoluções e refinamentos, os algoritmos e a abordagem pode chegar a uma precisão de 100%. De forma geral, com os resultados de todas as avaliações, foi possível concluir de forma positiva que tanto a DCL-KDM quanto a Arch-KDM 2.0 conseguem auxiliar e conduzir um processo de CCA no contexto da ADM utilizando o metamodelo KDM.

Capítulo 7

CONCLUSÕES

O principal ponto desta dissertação é a utilização do metamodelo KDM no processo de CCA completo que compreende desde a etapa de geração da arquitetura planejada até a etapa de visualização dos desvios arquiteturais identificados pelas abordagem. No Capítulo 3 foi apresentada a definição e formalização de dois termos muito recorrentes neste trabalho, sendo eles "Violação Arquitetural" e "Desvio Arquitetural". Este capítulo é de fundamental importância no contexto da ADM e do KDM em que a abordagem está inserida.

No Capítulo 4 foram apresentadas as etapas do processo de CCA e as evoluções realizadas para se desenvolver a Arch-KDM 2.0. Este capítulo apresenta como as características do processo de CCA foram abordados no contexto do KDM bem como a versão atual da abordagem após as evoluções realizadas. No Capítulo 5 foi apresentado um cenário de uso detalhado da abordagem. Esse capítulo teve o intuito de apresentar ao leitor como é a utilização da abordagem e do apoio computacional desenvolvido. No Capítulo 6 foi apresentada uma avaliação do apoio computacional da abordagem para avaliar sua efetividade diante do que a abordagem propõem realizar.

Este trabalho como um todo foi a evolução e continuação do trabalho apresentado por Chagas (2016). Dessa forma, foi realizada a evolução da abordagem e a construção do apoio ferramental Arch-KDM 2.0. Tal evolução foi necessária devido às falhas e problemas encontrados durante a execução da abordagem e, diante da necessidade de inclusão da abordagem em um processo de Reconciliação Arquitetural (RA). Portanto, cada etapa da abordagem foi repassada e analisada de forma que oportunidades de melhoria fossem encontradas e elencadas.

Como mencionado anteriormente, cada uma das evoluções nas etapas da abordagem envolve vários detalhes técnicos e são detalhadas nos Apêndices B, C e D. Além das evoluções e melhorias, realizou-se também a extensão da abordagem adicionando uma nova etapa. A

motivação para a inclusão dessa nova etapa gira em torno de peculiaridades do KDM e da inclusão da abordagem no contexto e processo da Reconciliação Arquitetural (RA).

De modo geral, a avaliação apresentou resultados satisfatórios como os observados no capítulo de avaliação. A avaliação realizada obteve uma precisão de 78% na recuperação de desvios arquiteturais automaticamente. Dessa forma, é possível concluir que a abordagem prova-se capaz de realizar o processo de CCA e o apoio computacional desenvolvido tem capacidade de encontrar os desvios arquiteturais corretamente.

Conforme apresentado, a abordagem por ser uma evolução manteve as bases elencadas por Chagas (2016). Portanto, quanto a expressividade da abordagem, ela continua apresentando bons resultados quanto à qualidade de sua verificação de conformidade, definição de restrições e uso de estilos arquiteturais. Quanto à adequabilidade, a abordagem melhorou seu desempenho utilizando uma quantidade maior de elementos que o pacote estrutural do KDM disponibiliza, apesar de continuar com a limitação para alguns estilos arquiteturais devido ao KDM não conseguir representar, a menos que sejam utilizados estereótipos e/ou realizado uma extensão. Fatores como este dificulta a criação de abordagens para a checagem de conformidade arquitetural utilizando estilos arquiteturais, pois adaptar um metamodelo para um determinado estilo arquitetural dificulta sua reutilização por outras ferramentas. Por fim, quanto à aplicabilidade, a abordagem tornou-se replicável baseada na formalização e, consequentemente, melhor para ser aplicada após as evoluções, sendo possível disponibilizá-la para que a comunidade possa utilizá-la.

Outro ponto interessante de mencionar é a definição entre os termos violação arquitetural e desvio arquitetural elaborado na dissertação. Apesar de na literatura ambos os termos serem muito utilizados como sinônimos e não ser encontrada uma definição formal para eles, no contexto deste trabalho e da ADM/KDM foram necessárias suas definições. Para esse dissertação, conforme apresentado no Capítulo 3, uma violação arquitetural compreende os relacionamentos primitivos e simples de um elemento de código. Por exemplo, em uma linha de código de instanciação, a instanciação propriamente dita e a chamada do método construtor. Já um desvio arquitetural são os conjuntos de relacionamentos primitivos, ou seja, conjuntos de violações, que geralmente representam algo concreto do sistema e consequentemente que pode ser refatorado e corrigido para sanar o problema arquitetural que ele representa.

É importante relembrar a principal característica da abordagem, o uso do KDM e dos conceitos da ADM. Esta característica é importante pois devido a ela, a abordagem é capaz de atingir quaisquer linguagens de programação. Isto é possível pois todos os algoritmos desenvolvidos e evoluídos são dependentes do metamodelo e não da tecnologia empregada no sistema.

Além disso, como a ADM elaborou o KDM transformando-o em um padrão ISO, todas as abordagens e apoios computacionais desenvolvidos para ele são passíveis de reutilização, fato que não ocorre com algoritmos ou abordagens elaborados baseados em modelos proprietários ou linguagens pré-definidas.

Dessa forma, com a padronização do KDM e a criação, evolução e popularização de abordagens que o utilizam, ferramentas de modernização independentes de suas motivações, têm excelentes motivações para adotar o KDM ao invés de modelos proprietários. E o principal motivo é que com a popularização do KDM e da ADM cada vez mais existirão recursos disponíveis como algoritmos, refatorações e técnicas. Um exemplo são os algoritmos apresentados nesta dissertação, como foram baseados nos conceitos do KDM, independente da linguagem de programação utilizada no apoio computacional, desde que utilize-se o KDM os algoritmos poderam ser utilizados na ferramenta.

7.1 Contribuições

As principais contribuições deste trabalho são:

- A formalização da CCA utilizada nesta dissertação, bem como a formalização dos termos Violação Arquitetural e Desvio Arquitetural.
- A evolução de uma abordagem já existente na literatura de checagem de conformidade arquitetural utilizando o metamodelo KDM, sem a utilização de extensões no metamodelo, o que facilita e simplifica a utilização sua utilização, apesar de esta ser uma das vantagens do KDM.
- A evolução de uma linguagem especifica de domínio para especificação de arquiteturas e a realização de uma avaliação.
- Uma nova etapa para a abordagem e uma nova avaliação para a abordagem.
- Uma ferramenta de apoio computacional para a aplicação da abordagem de checagem de conformidade arquitetural no contexto da ADM de forma eficiente, denominada Arch-KDM 2.0.

7.2 Publicações

Durante o desenvolvimento deste projeto, foram realizadas as seguintes divulgações de trabalhos:

- Landi, André de S.; Chagas, Fernando ; Santos, Bruno M. ; Costa, Renato S. ; Durelli, Rafael ; Terra, Ricardo ; Camargo, Valter V. de . Supporting the Specification and Serialization of Planned Architectures in Architecture-Driven Modernization Context. In: 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2017, Turin. p. 327-337.

- Landi, A. S.; Camargo, V. V. . Recomendações de Refatorações Arquiteturais Baseadas em Análise de Impacto no contexto da ADM. In: VI Workshop de Teses e Dissertações do CBSoft, 2016, Maringá, PR. p. 63-69.

- Serikawa, Marcel A. ; Landi, André de S. ; Siqueira, Bento R. ; Costa, Renato S. ; Ferrari, Fabiano C. ; Menotti, Ricardo ; Camargo, Valter V. de . Towards the Characterization of Monitor Smells in Adaptive Systems. In: 2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS), 2016, Maringá. p. 51-60.

- Durelli, Rafael S. ; Viana, Matheus C. ; Landi, André de S. ; Durelli, Vinicius H. S. ; Delamaro, Marcio E. ; De Camargo, Valter V. . Improving the structure of KDM instances via refactorings. In: Proceedings of the 31st Brazilian Symposium on Software Engineering - SBES'17, 2017. Fortaleza, p. 174-1784.

- Paula, M. H. ; Serikawa, M. A. ; Landi, A. S. ; Santos, B. M. ; Costa, R. S. ; Camargo, V. V. . SARA-MR: Uma Arquitetura de Referência para Facilitar Manutenções em Sistemas Robóticos Autoadaptativos. In: IV Workshop on Software Visualization, Evolution and Maintenance (VEM 2016), 2016, Maringá. p. 81-88.

7.3 Limitações

Apesar de realizar todo um processo de evolução da abordagem e a construção de um novo apoio computacional, foi possível observar que em alguns pontos os algoritmos apresentados ainda necessitam de evoluções. Dessa forma, é possível afirmar que, apesar da evolução ter aumentado a quantidade de elementos reconhecidos, uma das limitações ainda é a falta de utilização de alguns elementos que o KDM disponibiliza. Um exemplo é o observado na

avaliação do apoio ferramental, no qual elementos do tipo `TemplateUnit` não são reconhecidos pelo algoritmo.

Outra limitação importante notada foi a utilização da ferramenta *MoDisco*. Atualmente, na literatura são poucas as ferramentas encontradas que realizam a conversão de código fonte para o metamodelo KDM, para Java o *MoDisco* foi a única ferramenta encontrada. Apesar da ferramenta estar consolidada na literatura e, ser capaz de converter uma ampla gama de código fonte para KDM, existem alguns pontos falhos que tornam a descoberta do KDM incompleta. Um exemplo é a metaclasse *HasType* do KDM, que não é reconhecida pelo *MoDisco*. Este fato, dependendo do sistema e suas atribuições pode impactar significativamente no resultado da abordagem, uma vez que a mesma necessita de uma instância fiel do sistema para sua execução.

Por fim, outras duas limitações encontradas dizem respeito a quantidade de restrições arquiteturais reconhecidas e ao reconhecimento de aplicações web. A primeira pode ser resumida no fato do apoio computacional não ser capaz de reconhecer restrições arquiteturais de itens externos ao sistema, isto é, não é possível criar restrições para elementos do próprio Java. Por exemplo restrições do tipo "somente o elemento DAO pode acessar elementos do conector mySQL". A outra é baseada na literatura onde não foram encontradas ferramentas que realizam o reconhecimento de código de linguagens web (HTML, PHP, XHTML, etc)e transforme em KDM, o que impossibilita a validação de componentes da interface destes sistemas.

7.4 Trabalhos Futuros

Para trabalhos futuros pretende-se continuar a evolução dos algoritmos apresentados, de forma a atingir seu melhor potencial. Pretende-se procurar e firmar parcerias com empresas do setor privado para realizar a utilização da abordagem e do apoio computacional em sistemas maiores. Outro trabalho futuro é a implementação e pesquisa de algoritmos ou combinação de algoritmos alternativos a combinação Matriz de Proximidade e DBSCAN para a realização da descoberta de desvios arquiteturais.

Pretende-se para trabalhos futuros evoluir as especificações de restrições da DCL-KDM para contemplar restrições para APIs e pacotes padrões da linguagem de programação utilizada. Bem como abrir uma nova linha de pesquisa e investigar a possibilidade de novas ferramentas que realizem o reconhecimento de código de linguagens web para o KDM.

Outro tópico de pesquisa categorizado como uma continuação deste projeto é a realização da identificação e mapeamento de desvios arquiteturais em KDM para a sugestão de possíveis refatorações. Refatorações tais quais poderiam ser realizadas de forma automática ou semi-

automática com o desenvolvimento de um apoio computacional. Esse trabalho futuro em particular já está sendo pesquisado e iniciado pelo grupo de pesquisa AdvanSE.

REFERÊNCIAS

ALLIER, S. et al. From object-oriented applications to component-oriented applications via component-oriented architecture. In: *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, 2011. p. 214–223.

AVGERIOU, P. et al. Evolution through architectural reconciliation. *Electronic Notes in Theoretical Computer Science*, Elsevier B.V., v. 127, n. 3, p. 165–181, 2005. ISSN 15710661.

AVGERIOU, P.; GUELFI, N. Resolving architectural mismatches of COTS through architectural reconciliation. *Lecture Notes in Computer Science*, v. 3412, p. 248–257, 2005. ISSN 03029743.

BASS, L. *Software architecture in practice*. Third edit. USA: Pearson Education, Inc., 2012. 640 p. ISBN 978-0-321-81573-6.

BENNETT, K. Legacy Systems: Coping with success. *IEEE Software*, v. 12, n. 1, p. 19–23, 1995. ISSN 07407459.

BIANCHI, A. et al. Iterative reengineering of legacy functions. *IEEE International Conference on Software Maintenance, ICSM*, v. 29, n. 3, p. 632–641, 2001. ISSN 00985589.

BITTENCOURT, R. A. et al. Improving automated mapping in reflexion models using information retrieval techniques. In: *2010 17th Working Conference on Reverse Engineering*, 2010. p. 163–172. ISSN 1095-1350.

BOOCH, G. et al. *The unified modeling language user guide*. 2nd. ed.. ed. [S.l.]: Addison-Wesley Professional,, 2005. ISBN 0321267974.

BORAH, B.; BHATTACHARYYA, D. K. An improved sampling-based dbscan for large spatial databases. In: *International Conference on Intelligent Sensing and Information Processing, 2004. Proceedings of*, 2004. p. 92–96.

BOSCH, J. *Design and Use of Software Architectures*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. ISBN 0-201-67494-7.

BOUCKAERT, R. R. et al. WEKA–experiences with a java open-source project. *Journal of Machine Learning Research*, v. 11, p. 2533–2541, 2010.

BOURQUIN, F.; KELLER, R. K. High-impact refactoring based on architecture violations. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, p. 149–158, 2007. ISSN 15345351.

BRAGA, T. H. *Recuperação da arquitetura de software para manutenção de sistemas*. 146 p. Tese (Doutorado) — UFMG, 2013.

BRUNELIERE, H. et al. Modisco: A generic and extensible framework for model driven reverse engineering. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2010. (ASE '10), p. 173–174. ISBN 978-1-4503-0116-9. Disponível em: <http://doi.acm.org/10.1145/1858996.1859032>.

BRUNELIèRE, H. et al. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, v. 56, n. 8, p. 1012 – 1032, 2014. ISSN 0950-5849. Disponível em: <http://www.sciencedirect.com/science/article/pii/S0950584914000883>.

BUSCHMANN, F. et al. Pattern-Oriented Software Architecture Volume 1: A System of Patterns. *Wiley*, Vol. 1, p. 476, 1996. ISSN 0007-1250.

CHAGAS, F. B. *Checagem de conformidade arquitetural na modernização orientada a arquitetura*. 81 p. Tese (Doutorado) — UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2016.

DEITERS, C. et al. Rule-based architectural compliance checks for enterprise architecture management. In: *2009 IEEE International Enterprise Distributed Object Computing Conference*, 2009. p. 183–192. ISSN 1541-7719.

DEMEYER, S. Refactor conditionals into polymorphism: What's the performance cost of introducing virtual calls ? *IEEE International Conference on Software Maintenance, ICSM*, v. 2005, p. 627–630, 2005. ISSN 1063-6773.

DEMEYER, S. Object-oriented reengineering. In: ____. *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 91–104. ISBN 978-3-540-76440-3. Disponível em: <https://doi.org/10.1007/978-3-540-76440-3_5>.

DOGRU, A. H. *Modern Software Engineering Concepts and Practices: Advanced Approaches: Advanced Approaches*. [S.l.]: IGI Global, 2010.

DURELLI, R. S. *Uma Abordagem para Criação, Reúso e Aplicação de Refatorações no Contexto da Modernização Dirigida a Arquitetura*. 231 p. Tese (Doutorado) — USP, 2016.

DUSZYNSKI, S.; KNODEL, J.; LINDVALL, M. Save: Software architecture visualization and evaluation. In: *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, 2009. p. 323–324. ISSN 1534-5351.

ESTER, M. et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In: *Kdd*, 1996. v. 96, n. 34, p. 226–231.

FONSECA, R. J. R. M. d. et al. Acordo inter-juízes: O caso do coeficiente kappa. *Laboratório de Psicologia*, Instituto Superior de Psicologia Aplicada, p. 81–90, 2007.

GARLAN, D. et al. *Documenting software architectures: views and beyond*. 2nd. ed.. ed. [S.l.]: Addison-Wesley Professional, 2010. 342 p. ISSN 0270-5257. ISBN 0321552687, 9780321552686.

GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

GARNER, S. Weka: The waikato environment for knowledge analysis. In: *Proc New Zealand Computer Science Research Students Conference*, 1995. p. 57–64.

GASPARINI, B. C. *VISUALIZAÇÃO DE NÃO-CONFORMIDADES ARQUITETURAIS EM UML NO CONTEXTO DA ADM*. 90 p. Tese (Doutorado) — UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2018.

GRUNBACHER et al. Reconciling software requirements and architectures with intermediatemodels. *Software and Systems Modeling*, v. 3, n. 3, p. 235–253, 2003. ISSN 1090-705X.

HANNEMANN, J.; MURPHY, G. C.; KICZALES, G. Role-based refactoring of crosscutting concerns. In: *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, 2005. (AOSD '05), p. 135–146. ISBN 1-59593-042-6. Disponível em: <http://doi.acm.org/10.1145/1052898.1052910>.

HEROLD, S.; MAIR, M. Recommending refactorings to re-establish architectural consistency. In: AVGERIOU, P.; ZDUN, U. (Ed.). *Software Architecture*, 2014. p. 390–397. ISBN 978-3-319-09970-5.

HUNT, A.; THOMAS, D. Software Archaeology. *IEEE Software*, v. 19, n. 2, p. 20–22, 2002. ISSN 0740-7459.

INTERNATIONAL, S. G. *Modernization: Clearing a pathway to success*. [S.l.], 2010. 10– p. Acessado em 20/12/2016. Disponível em https://www.standishgroup.com/sample_research_files/Modernization.pdf.

IVKOVIC, I.; KONTOGIANNIS, K. A framework for software architecture refactoring using model transformations and semantic annotations. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006. p. 10 pp.–144. ISSN 1534-5351.

JARCHITECT. *JArchitect 5: A tool to evaluate java code base*. 2016. Available in http://www.jarchitect.com/.

KAZMAN, R.; WOODS, S. G.; CARRIèRE, S. J. Requirements for integrating software architecture and reengineering models: Corum ii. In: *Proceedings of the Working Conference on Reverse Engineering (WCRE'98)*, 1998. (WCRE '98), p. 154–. ISBN 0-8186-8967-6.

KNODEL, J. et al. Static evaluation of software architectures. In: *Conference on Software Maintenance and Reengineering (CSMR'06)*, 2006. p. 10 pp.–294. ISSN 1534-5351.

KNODEL, J.; POPESCU, D. A comparison of static architecture compliance checking approaches. In: *Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture*, 2007. (WICSA '07), p. 12–. ISBN 0-7695-2744-2.

KRUCHTEN, P. B. 4+1 View Model of Architecture. *IEEE Software*, v. 12, n. 6, p. 42–50, 1995. ISSN 07407459.

KRUEGER, C. W. Software reuse. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 2, p. 131–183, jun. 1992. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/130844.130856>.

LANDGREBE, T. C. W.; PACLIK, P.; DUIN, R. P. W. Precision-recall operating characteristic (p-roc) curves in imprecise environments. In: *18th International Conference on Pattern Recognition (ICPR'06)*, 2006. v. 4, p. 123–127. ISSN 1051-4651.

LANDI, A. de S. et al. Supporting the specification and serialization of planned architectures in architecture-driven modernization context. In: *41st International Conference on Computers, Software and Applications (COMPSAC)*, 2017. p. 1–10.

LIPPERT, M.; ROOCK, S. *Refactoring in Large Software Projects: Performing Complex Restructurings Successfully*. [S.l.]: John Wiley & Sons, 2006. 286 p. ISBN 978-0-470-85892-9.

MAFFORT, C. et al. Heuristics for discovering architectural violations. In: *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013. p. 222–231. ISSN 1095-1350.

MAKHOUL, J. et al. Performance measures for information extraction. In: Herndon, va. *Proceedings of DARPA broadcast news workshop*, 1999. p. 249–252.

MATOS, D. A. S. Confiabilidade e concordância entre juízes: aplicações na área educacional. 2014.

MEDVIDOVIC, N.; TAYLOR, R. N. A classification and comparison framework for software architecture description languages. *IEEE Software Engineering*, v. 26, n. 1, p. 70–93, 2000. ISSN 00985589.

MEFFERT, K. Supporting design patterns with annotations. In: *ECBS*, 2006. v. 6, p. 437–445.

MURPHY, G. C.; NOTKIN, D.; SULLIVAN, K. J. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, v. 27, n. 4, p. 364–380, Apr 2001. ISSN 0098-5589.

O'BRIEN, L.; STOERMER, C.; VERHOEF, C. Software Architecture Reconstruction: Practice Needs and Current Approaches. *Technical report*, Pittsburgh, PA, Vol. 1, n. CMU/SEI-2002-TR-024, 2002.

OMG. *Architecture-Driven Modernization Standards Roadmap*. 2009. 423 p. Disponível em http://adm.omg.org/.

OMG. *Knowledge Discovery Meta-model (KDM)*. 2016. Disponível em http://www.omg.org/technology/kdm/, especificação disponível em http://www.omg.org/spec/KDM/.

OMG. *OMG ® Specifications BUSINESS MODELING SPECIFICATIONS*. 2016. Disponível em http://www.omg.org/spec/.

PARADAUSKAS, B.; LAURIKAITIS, A. Business knowledge extraction from legacy information systems. *Information technology and control*, v. 35, n. 3, 2015.

PASSOS, L. et al. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, v. 27, n. 5, p. 82–89, Sept 2010. ISSN 0740-7459.

PEREZ-CASTILLO, R. et al. Obtaining thresholds for the effectiveness of business process mining. In: *2011 International Symposium on Empirical Software Engineering and Measurement*, 2011. p. 453–462. ISSN 1949-3770.

PERRY, D.; WOLF, A. L. *Foundations for the study of software architecture*. 1992. 40–52 p.

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. *Best Paper*.

RAHIMI, R.; KHOSRAVI, R. Architecture conformance checking of multi-language applications. In: *ACS/IEEE International Conference on Computer Systems and Applications - AICCSA 2010*, 2010. p. 1–8. ISSN 2161-5322.

RONCERO, V. G. *Classificação semi-supervisionada de textos em ambientes distribuídos*. Tese (Doutorado) — Ph. D. dissertation, Universidade Federal do Rio de Janeiro, 2010.

SANTOS, B. M. *Estensões Do Metamodelo Kdm Para Apoiar Modernizações Orientadas a Aspectos De Sistemas Legados*. 122 p. Tese (Doutorado) — UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2014.

SHAW, M.; GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline*. USA: Prentice Hall, 1996. 242 p. ISBN 0131829572.

SHIVA, S. G.; SHALA, L. A. Software reuse: Research and practice. In: *Information Technology, 2007. ITNG '07. Fourth International Conference on*, 2007. p. 603–609.

SILVA, L. D. *Towards Controlling Software Architecture Erosion Through Runtime Conformance Monitoring*. Tese (Thesis) — University of St Andrews, 2014.

SILVA, L. de; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 85, n. 1, p. 132–151, jan. 2012. ISSN 0164-1212. Disponível em: <http://dx.doi.org/10.1016/j.jss.2011.07.036>.

SNEED, H. M. Estimating the costs of a reengineering project. *Proceedings - Working Conference on Reverse Engineering, WCRE*, v. 2005, p. 111–119, 2005. ISSN 10951350.

SNEED, H. M. Offering software maintenance as an offshore service. *IEEE International Conference on Software Maintenance, ICSM*, p. 1–5, 2008. ISSN 1063-6773.

SOMMERVILLE, I. *Engenharia de software*. [S.l.]: McGraw Hill Brasil, 2003. 580 p. ISSN 03029743. ISBN 85-88639-07-6.

TERRA, R.; VALENTE, M. T. A dependency constraint language to manage object-oriented software architectures. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., New York, NY, USA, v. 39, n. 12, p. 1073–1094, ago. 2009. ISSN 0038-0644. Disponível em: <http://dx.doi.org/10.1002/spe.v39:12>.

TERRA, R. et al. Recommending refactorings to reverse software architecture erosion. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, p. 335–340, 2012. ISSN 15345351.

TRUYEN, F. *The Fast Guide to Model Driven Architecture: The Basics of Model Driven Architecture*. 2006. 16 p. Disponível em http://www.omg.org/mda/mda_files/Cephas_MDA_Fast_Guide.pdf.

ULRICH, W. M. *Legacy Systems: Transformation Strategies*. [S.l.]: Prentice Hall, 2002.

ULRICH, W. M.; KHUSIDMAN, V. *Architecture-Driven Modernization: Transforming the Enterprise*. 2007. Disponível em http://adm.omg.org/.

VISAGGIO, G. Ageing of a data-intensive legacy system: symptoms and remedies. *Journal of Software Maintenance and Evolution: Research and Practice*, John Wiley & Sons, Ltd., v. 13, n. 5, p. 281–308, 2001. ISSN 1532-0618. Disponível em: <http://dx.doi.org/10.1002/smr.234>.

WANG, Y.-H.; WU, I. et al. Achieving high and consistent rendering performance of java awt/swing on multiple platforms. *Software: Practice and Experience*, Wiley Online Library, v. 39, n. 7, p. 701–736, 2009.

WOHLIN, C. et al. *Experimentation in software engineering: an introduction*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. ISBN 0-7923-8682-5.

GLOSSÁRIO

ADL – *Architecture Description Language*

ADMPR – *ADM Pattern Recognition Specification*

ADMRS – *ADM Refactoring Specification*

ADMTF – *Architecture-Driven Modernization Task Force*

ADMTM – *ADM Transformation Metamodel*

ADMVS – *ADM Visualization Specification*

ADM – *Architecture-Driven Modernization*

API – *Application Programming Interface*

ASTM – *Abstract Syntax Tree Metamodel*

ATL – *Atlas Transformation Language*

CCA – *Checagem de Conformidade Arquitetural*

CIM – *Computational Independent Model*

DSL – *Domain-Specific Language*

EMF – *Eclipse Modeling Framework*

IDE – *Integrated Development Environment*

KDM – *Knowledge Discovery Metamodel*

MDA – *Model-Driven Archtecture*

MDD – *Model-Driven Development*

MOF – *Meta-Object Facility*

MVC – *Model-View-Controller*

OMG – *Object Management Group*

PIM – *Platform Independent Model*

PSM – *Platform Specific Model*

RA – *Reconciliacão Arquitetural*

SMM – *Software Metrics Metamodel*

UML – *Unified Modeling Language*

XMI – *XML Metadata Interchange*

XML – *Extensible Markup Language*

Apendice A

TABELAS DE EXPLICAÇÃO/EXEMPLIFICAÇÃO DAS METACLASSES DO KDM

Tabela A.1: Relacionamento de código possíveis entre elementos de código (I)

Metaclasse	Descrição
InstanceOf	Esse tipo de relacionamento representa a "instanciação" de uma relação entre um elemento de código como uma classe e um Template. Ou seja, a ligação ente um método genérico e sua classe parametrizada. Por exemplo, em uma declaração do tipo "public <T >void method(T test)" e sua implementação como "public void method(String test)", esse relacionamento representa a ligação entre "T" e "String"
ParameterTo	Esse tipo de relacionamento representa a "parametrização" de uma relação entre um template e um elemento de código, como uma classe. Ou seja, a ligação entre uma classe parametrizada em um método genérico. Por exemplo, em uma chamada de um método declarado como "public <T >void sum(T a, T b)" e sua chamada sendo "sum(1, 2)", esse relacionamento representa a ligação entre "T" e "int"
Implements	Esse tipo de relacionamento representa a "implementação" de uma interface. Ou seja, a ligação entre uma classe e sua interface. Por exemplo, na declaração "public class A implements B", esse relacionamento representa a ligação entre "A" e "B".
ImplementsOf	Esse tipo de relacionamento representa a "implementação" de um método ou atributo de uma classe implementável. Ou seja, a ligação entre uma implementação e sua declaração. Por exemplo, uma interface possui o método "void do()" e em uma classe que a implementa tem-se "@override void do(){}", esse relacionamento representa a ligação entre o "do" da interface e sua implementação "@override do"

Tabela A.2: Relacionamento de código possíveis entre elementos de código (II)

Metaclasse	Descrição
HasType	Esse tipo de relacionamento representa a se um determinado elemento de código "tem o tipo" de outro determinado elemento de código. Esse relacionamento por si só acaba duplicando a informação no metamodelo KDM pois cada elemento possui um atributo denominado tipo, porém ele habilita a ligação entre esse tipo de relacionamento em outros pacotes do KDM, como o Structure. Por exemplo, em uma declaração como "A a", esse relacionamento representa a ligação entre "a" e seu tipo "A".
HasValue	Esse tipo de relacionamento representa o "valor de inicialização" de algum elemento de código. Ou seja, a ligação entre um elemento de código e seu valor inicial. Por exemplo, em uma linha de código como "A a = new A()", esse relacionamento representa a ligação entre "a" e seu valor "new A()", o que é diferente de "a = new A()", neste caso, esse relacionamento não se aplica pois está sendo realizado uma atribuição e não uma inicialização.
Extends	Esse tipo de relacionamento representa a herança entre dois elementos de código como classes. Ou seja, a ligação entre um elemento de código que é considerado filho de outro elemento. Por exemplo, na declaração "public interface A extends B", esse relacionamento é a ligação entre "A" e "B".
Expands	Esse tipo de relacionamento representa a relação entre definições de macros ou diretivas de macros que geram código que são comuns em algumas linguagens de programação. Ou seja, representa a ligação entre duas macros, ou uma macro e uma diretiva de macro. Por exemplo, tem-se as seguintes definições "#define GT(A,B) ((A) >(B))" e "#define GMAX(A,B) g=(GT(A,B) ? (A) : (B))", esse relacionamento é a ligação entre "GT" e seu uso em "GMAX"
GeneratedFrom	Esse tipo de relacionamento representa a relação entre o uso de uma macro, diretiva de macro ou uma variante default de uma linha de produto de software. Ou seja, representa a ligação entre o uso e sua correspondente definição. Por exemplo, utilizando-se as definições do exemplo de MacroRelationship tem-se o uso por "GMAX(p+q, r+s);", esse relacionamento é a ligação entre esse uso de "GMAX" e sua definição.
Includes	Esse tipo de relacionamento representa a relação entre uma determinada diretiva de inclusão e a unidade compartilhada que deve ser incluída. Ou seja, representa a ligação entre uma inclusão e o incluído. Por exemplo, em uma declaração "#include "a.h"", esse relacionamento representa a ligação da declaração de inclusão "#include" com o arquivo incluído "a.h".
VariantTo	Esse tipo de relacionamento representa a relação entre uma determinada diretiva condicional e sua variação quando se trabalha com linhas de produto de software. Ou seja, representa a ligação entre uma diretiva e sua variação. Por exemplo, na definição "IfDef UNIX g=1 #else g=2 #endif", esse relacionamento representa a ligação entre a diretiva condicional "IfDef UNIX" e sua variação "#else".

Tabela A.3: Relacionamento de código possíveis entre elementos de código (III)

Metaclasse	Descrição
Redefines	Esse tipo de relacionamento representa a relação entre uma determinada definição de macro, diretiva ou variante e sua posterior redefinição. Ou seja, representa a ligação entre uma redefinição e sua definição original. Por exemplo, em uma declaração "#define A 1" e sua posterior definição "#define A 2", esse relacionamento representa a ligação entre a definição "A 1" e sua redefinição para "A 2".
VisibleIn	Esse tipo de relacionamento representa a relação de visibilidade entre um elemento de código e seu contexto. Ou seja, representa a ligação de visibilidade de um elemento com o contexto. Esse tipo de relação geralmente é opcional devido ao fato de cada linguagem de programação tratar a semântica de visibilidade de uma forma diferente, porém esse tipo de relação proporciona a ligação com outros pacotes do KDM como o Structure. Por exemplo, na declaração "namespace space{ void func(){} }", esse relacionamento é a ligação entre o "namespace" e o elemento.
Imports	Esse tipo de relacionamento representa a relação entre um elemento de código, como uma classe, e outro elemento de código, como uma interface, onde uma importa as funcionalidades da outra. Ou seja, representa a ligação ente dois elementos de código por meio da "importação". Por exemplo, em uma classe "A" encontra-se o código "import package.B", esse relacionamento é a ligação entre "A" e "B", onde significa que "A" importa a classe "B".
CodeRelationship	Esse tipo de relacionamento representa uma relação genérica ente quaisquer dois elementos de código fonte que não se encaixam em nenhum dos tipos anteriores. Ou seja, um relacionamento qualquer que seja específico de algum padrão, paradigma ou linguagem que não se encaixa nos tipos anteriores.

Tabela A.4: Relacionamentos de ações possíveis entre elementos de código (I)

Metaclasse	Descrição
ControlFlow	Esse tipo de relacionamento representa a relação genérica do fluxo de controle entre dois elementos de ação do código-fonte. Ou seja, representa a ligação entre duas linhas de códigos e seu fluxo de controle. Esse tipo de relacionamento deve ser utilizado quando o fluxo de controle alvo da relação não se enquadra em um dos seguintes tipos: `InitializationFlow`, `StatementFlow`, `TrueConditionFlow`, `FalseConditionFlow`, `OptionConditionFlow`.
EntryFlow	Esse tipo de relacionamento representa a relação de fluxo de entrada de um elemento de código-fonte. Ou seja, representa a ligação entre um elemento e seu bloco de inicialização. Por exemplo, dentro de uma classe tem-se a declaração "{ print "text" }", esse relacionamento é a ligação da classe com este bloco de inicialização.

Tabela A.5: Relacionamentos de ações possíveis entre elementos de código (II)

Metaclasse	Descrição
Flow	Esse tipo de relacionamento representa a relação entre duas linhas de código-fonte e sua ordem. Ou seja, representa a ligação entre uma linha de código-fonte e a que vem imediatamente a seguir desta. Por exemplo, a definição de duas linhas como "`a = 1; b = a;`", esse relacionamento é a ligação entre a linha "`a = 1;`" e a linha "`b = a;`".
TrueFlow	Esse tipo de relacionamento representa a relação entre uma condição e seu fluxo de controle quando a mesma é considerada verdadeira. Ou seja, representa a ligação entre uma condição e um fluxo de código-fonte a ser executado quando a condição é verdadeira. Por exemplo, em uma condição `if`, esse relacionamento é a ligação entre o `if` e o bloco de código que é executado quando o `if` é verdadeiro.
FalseFlow	Esse tipo de relacionamento representa o oposto do tipo de relacionamento `TrueConditionFlow`. Ou seja, representa a ligação entre uma condição de um fluxo de código a ser executado quando a condição é falsa. Por exemplo, em uma condição `if`, esse relacionamento é a ligação entre o `if` e o bloco de código que é executado quando o `if` é falso.
GuardedFlow	Esse tipo de relacionamento representa a relação entre uma condição e cada uma de suas opções de fluxo de controle. Ou seja, representa a ligação entre uma condição e um possível fluxo de controle para essa condição. Por exemplo, em uma condição do tipo `switch`, esse relacionamento representa a ligação entre a condição do `switch` e cada um dos cases do mesmo.
Calls	Esse tipo de relacionamento representa a relação entre um elemento de código que invoca outro elemento de código. Ou seja, representa a ligação entre um elemento que chama outro elemento e o elemento chamado. Por exemplo, tem-se a seguinte declaração "`object1.toString()`", esse relacionamento é a ligação entre o "`object1`" e o método "`toString`".
Dispatches	Esse tipo de relação representa a relação entre um elemento de código e um tipo de dado. Ou seja, representa a ligação ente o uso e um elemento de dado, similar ao `ElementCalls`, diferenciando que o alvo da chamada é um elemento de dado. Por exemplo, em uma definição "`typedef int (*fp)(int i);`" e um uso "`fp pf; *pf(1);`", esse relacionamento é a ligação entre o uso e o tipo "`fp`".
Reads	Esse tipo de relacionamento representa a relação entre uma linha de código fonte e um elemento do tipo de dados. Ou seja, representa a ligação de uma linha de código e um elemento de dado que deve ser lido. Por exemplo, em uma declaração do tipo "`a = class1.b`", esse relacionamento é a ligação entre a linha "`a = classe1.b`" e a leitura do elemento de dado "`b`".

Tabela A.6: Relacionamentos de ações possíveis entre elementos de código (III)

Metaclasse	Descrição
Writes	Esse tipo de relacionamento representa a relação entre um elemento de dados e sua linha de código. Ou seja, representa a ligação entre um tipo de dado que vai receber um valor e sua linha de código. Por exemplo, em uma declaração do tipo "`a = class1.b`", esse relacionamento é a ligação entre o elemento de dado "`a`" que vai receber um valor de escrita e a linha "`a = classe1.b`".
Addresses	Esse tipo de relacionamento representa a relação de acesso para estruturas complexas de dados. Ou seja, representa a ligação entre um elemento que recebe a referência para outro elemento de dado. Por exemplo, a partir das seguintes definições "`typedef int (*fp) (int i ); int foo(int i){}`" e o uso sendo "`fp pf; pf = foo;`", esse relacionamento é a ligação entre o conteúdo de "`pf`" e o endereço de "`foo`".
Creates	Esse tipo de relacionamento representa a relação entre uma linha de código e um tipo de dado que é criado nela. Ou seja, representa a ligação entre uma linha de código que instancia um tipo, e esse tipo de dado instanciado. Por exemplo, na declaração "`a = new A();`", esse relacionamento é a ligação entre a linha de código e a classe que é instanciada nessa linha ("`A`").
ExitFlow	Esse tipo de relacionamento representa a relação entre um bloco de código-fonte que deve tentar ser executado e o bloco de código-fonte que deve ser executado após isso, ou seja, a ligação entre dois blocos de códigos-fonte que onde o segundo deve ser obrigatoriamente executado. Por exemplo, em um fluxo de controle do tipo `try/catch` existe o fluxo `finally`, esse relacionamento é a ligação entre o bloco `try/catch` e o `finally`.
ExceptionFlow	Esse tipo de relacionamento representa a relação entre um bloco ou linha de código-fonte que deve tentar executar e um bloco de código que lança uma exceção, ou seja, a ligação entre um bloco ou linha de código que sua execução pode levar a uma exceção e o bloco de código que executa a exceção. Por exemplo, em um fluxo de controle do tipo `try` existe o fluxo `catch`, esse relacionamento é a ligação entre o bloco `try` e o bloco `catch`.
Throws	Esse tipo de relacionamento representa a relação entre uma linha de código que lança uma exceção e o elemento de dados que é a exceção. Ou seja, representa a ligação entre a exceção e o elemento dados lançador. Por exemplo, na declaração "`throw e;`", esse relacionamento é a ligação entre a linha e o objeto "`e`" lançado como exceção.

Tabela A.7: Relacionamentos de ações possíveis entre elementos de código (IV)

Metaclasse	Descrição
CompliesTo	Esse tipo de relacionamento representa a relação entre um elemento de código de uma interface ou classe abstrata e sua real implementação. Ou seja, representa a ligação entre uma declaração de um elemento e seu uso. Por exemplo, em uma chamada de método de uma interface, esse relacionamento é a ligação entre o método da interface e seu uso.
UsesType	Esse tipo de relacionamento representa a relação entre uma linha de código que faz uma conversão de dados e o tipo do dado que esta sendo convertido, ou seja, a ligação entre uma linha de conversão e o dado destino dessa linha. Por exemplo, em uma declaração "`a = (A) b;`", esse relacionamento representa a ligação entre a linha de código e o tipo "`A`" da conversão.
ActionRelationship	Esse tipo de relacionamento representa uma relação genérica entre dois elementos de ações do código-fonte que não se encaixam em nenhum dos tipos anteriores, isto é, qualquer que seja específico de algum padrão, paradigma ou linguagem que não se encaixe nos anteriores.

Tabela A.8: Relacionamentos possíveis entre elementos arquiteturais

Metaclasse	Descrição
AggregatedRelationship	Esse tipo de relacionamento representa a relação entre dois elementos quaisquer e que tenham relações entre si. Essas relações são agrupadas nesse tipo de relacionamento por meio da sua forma mais simples como demonstradas nas tabelas 1 e 2. Porém em cada agrupamento deve-se manter seguir a regra que o elemento inicial do tipo de relacionamento é pertence ou é o mesmo do elemento inicial do relacionamento `AggregatedRelationship`, tal qual como o destino. Ou seja, esse tipo de relacionamento é um agrupamento de relacionamentos primitivos que ocorrem entre dois elementos e no qual se respeita a direção do relacionamento. Por exemplo, em um determinado sistema existem duas camadas A e B, essas camadas são representadas por dois pacotes distintos A e B. Entre esses pacotes existem relacionamentos de código e de ação entre si. Para representar esses relacionamentos no contexto das camadas, faz-se necessário encontrar todos os relacionamentos que partem do pacote A e terminam no pacote B. Esses relacionamento são agrupados por meio desse relacionamento e então cria-se um relacionamento partindo da camada A para a camada B e aloca-se os relacionamentos de ação e código neste relacionamento estrutural. Da mesma forma deve ser feito o oposto para se preencher os relacionamentos originários da camada B para a camada A.

Apendice B

EVOLUÇÕES REALIZADAS NA ETAPA ESPECIFICAÇÃO DA ARQUITETURA PLANEJADA

Em um estudo mais aprofundado sobre a primeira etapa da abordagem, incluindo suas características, implementação e utilização, observou-se que a mesma não atendia completamente as necessidades de criação de uma Arquitetura Planejada de forma plena. Desse modo, se fez necessário algumas evoluções em sua estrutura, implementação e características. Foram realizadas diversas evoluções e correções e cada uma tem suas motivações e justificativas, dessa forma a seguir são apresentadas as principais evoluções realizadas na primeira etapa da abordagem.

Uma evolução realizada foi a retirada da palavra-chave `interface` da DCL-KDM. Perante sua elaboração, a DCL-KDM foi idealizada de forma que componentes necessitassem de estruturas auxiliares denominadas interfaces e representadas pela palavra-chave `interface`. Porém, em uma análise realizada no KDM observou-se que para a representação de um elemento denominado `interface` apenas com o Pacote *Structure* do KDM, seria necessário realizar uma extensão (leve ou pesada) do metamodelo. Realizar uma extensão do metamodelo seria uma infração na ideia inicial da abordagem de usar apenas o pacote *Structure* para representar a Arquitetura Planejada, além de ferir a intenção da OMG de utilização do metamodelo, sempre que possível, sem realizar extensões. Dessa forma, durante esta análise optou-se por retirar a utilização da palavra-chave `interface` da DCL-KDM tornando a especificação de um componente mais simples e direta. Considera-se essa modificação como sendo uma modificação de melhoria a DCL-KDM devido a realizar uma melhora na utilização do metamodelo KDM, uma vez que retirou-se a necessidade de realizar uma extensão do mesmo.

Outra evolução realizada foi a separação dos conceitos de hierarquia de comunicação entre camadas e composição de elementos arquiteturais. Essa inconsistência de conceitos encontrava-

se no módulo de geração da instância do KDM, o que ocasionava problemas na serialização da arquitetura planejada. Inicialmente na versão original da DCL-KDM, esses conceitos estavam intrinsecamente ligados gerando instâncias do KDM inconsistentes e de difícil compreensão. Em uma especificação simples conforme o Código B.1, a versão original da DCL-KDM geraria uma instância sem a composição. Deixando assim, suas regras de comunicação definir a composição e hierarquia, dificultando o entendimento, não sendo fiel a especificação realizada e, consequentemente, não gerando as regras automaticamente para composição.

```
1 architeturalElements {
2     subSystem core;
3     layer view, level 3, inSubSystem: core;
4     layer controller, level 2, inSubSystem: core;
5     layer model, level 1, inSubSystem: core;
6
7     component repository, inLayer: model;
8
9     component validator;
10    component generic;
11    component converter;
12 } restrictions {
13 }
```

Código B.1: Exemplo de especificação em DCL-KDM

Na Figura B.1 pode ser observado como era a representação dos elementos arquiteturais em KDM da DCL-KDM original. Como pode ser observado, não existe o conceito de composição entre os elementos arquiteturais, consequentemente, não sendo fiel à especificação apresentada anteriormente. As restrições arquiteturais não estão representadas nessa figura, porém, também não eram geradas corretamente as permissões de comunicação entre um elemento arquitetural e sua composição de elementos. Em contrapartida, após a evolução realizada, os elementos arquiteturais passaram a ser posicionados corretamente, deixando apenas a hierarquia para ser definida pelas regras de comunicação conforme definido pela palavra chave `level`. Dessa forma, também foi possível gerar automaticamente as regras de comunicação entre os elementos de composição, o que não era gerado corretamente na versão original. Na Figura B.2 pode ser observado como está atualmente a geração da arquitetura planejada na DCL-KDM. Como pode ser observado, a instância do KDM está totalmente fiel a especificação, gerando corretamente a composição entre os elementos arquiteturais bem como algumas regras denotadas pelas instâncias da metaclasse `AggregatedRelationship`.

Outra evolução realizada foi um mapeamento completo entre as restrições disponibilizadas pela DCL-KDM e as metaclasses do KDM. Ao se utilizar a DCL-KDM observou-se um deficit no qual a versão atual abrangia apenas uma parte do metamodelo KDM. Em razão disso

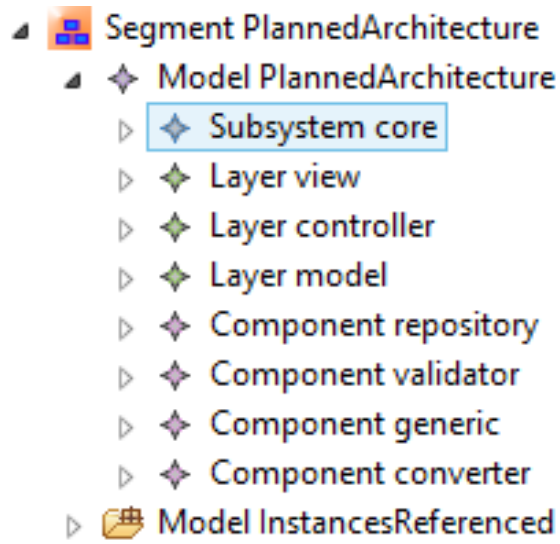

Figura B.1: Visualização gráfica de uma instância KDM gerada pela DCL-KDM original para o Código B.1. Fonte: Elaborado pelo autor

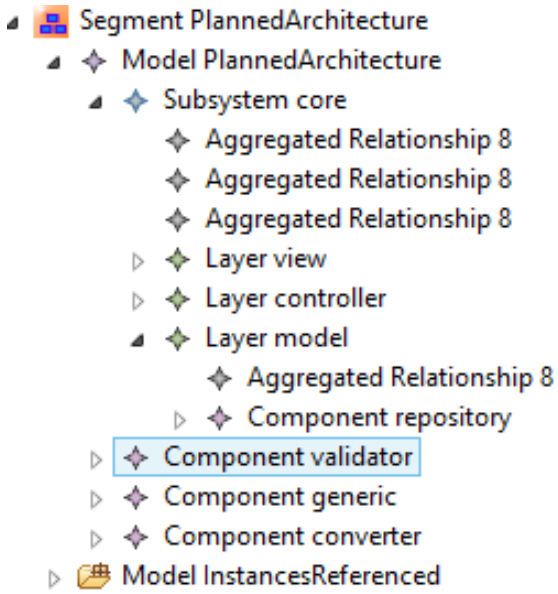

Figura B.2: Visualização gráfica de uma instância KDM gerada pela DCL-KDM atual para o Código B.1. Fonte: Elaborado pelo autor

realizou-se um aprofundamento de conhecimentos sobre o metamodelo KDM a ponto de observar que em sua totalidade existem 34 metaclasses que representam relacionamentos. Nas tabelas do Apêndice A estão descritas cada uma dessas metaclasses separadamente. Cada uma delas representa um relacionamento diferente e distinto, que um elemento pode conter com outro elemento em nível de código. Portanto, evoluiu-se a DCL-KDM para todos os 34 tipos de relacionamento, diferente da versão original que utilizava apenas 8 relacionamentos identificadas e demonstradas na Tabela B.1. A Tabela B.2 elenca os tipos de restrições disponíveis na DCL-KDM com as suas respectivas metaclasses representantes em KDM. Com esse novo mape-

amento, a geração da arquitetura planejada é completa perante as possibilidades proporcionadas pelo metamodelo KDM, não deixando possíveis relacionamentos sem serem alcançados.

Tabela B.1: Mapeamento entre restrições da DCL-KDM e metaclasses do KDM da versão Original

Dependência DCL-KDM	Metaclasse do KDM
Access	Calls
Declare	HasType
Handle	Calls, HasType
Create	Creates
Extend	Extends
Implement	Implements
Derive	Extends, Implements
Throw	Calls, ExceptionFlow
Useannotation	HasValue
Depend	Todas citadas.

Tabela B.2: Mapeamento entre as restrições da DCL-KDM e o metamodelo KDM da versão evoluída

Metaclasse	Descrição
access	Imports, Calls, Dispatches
declare	HasType, HasValue, Imports, Reads, Writes, Addresses, UsesType
handle	InstanceOf, ParameterTo, HasType, GeneratedFrom, Includes, VariantTo, Redefines, VisibleIn, Imports, ControlFlow, EntryFlow, Flow, TrueFlow, FalseFlow, GuardedFlow, Calls, Dispatches, Reads, Writes, Addresses, UsesType
create	HasType, HasValue, Imports, Calls, Addresses, Creates, UsesType
extend	Extends, Expands, Imports
implement	Implements, ImplementsOf, Imports, CompliesTo
derive	InstanceOf, ParameterTo, Implements, ImplementsOf, Extends, Expands, Includes, Redefines, Imports, CompliesTo
throw	Imports, ExitFlow, ExceptionFlow, Throws
useannotation	HasValue, Imports, Calls
depend	Todas as 34 meta-classes

Outra evolução realizada foi a reengenharia da implementação do protótipo de geração da arquitetura planejada a fim de transformá-lo em um plug-in para o Eclipse juntamente com a

DCL-KDM. Na Figura B.3 - Item A, pode-se observar a arquitetura base da DCL-KDM em sua versão original. Na mesma figura no Item B, pode-se observar a arquitetura da DCL-KDM após as evoluções realizadas. Em ambos os itens é apresentada a visão lógica da arquitetura da DCL-KDM e distribuída em duas camadas. A primeira camada denomina-se IDE e denota os recursos do Eclipse IDE para a boa utilização e implementação da DCL-KDM, que são: *Eclipse Modeling Framework* (EMF), *Xtext*, *MoDisco* e o *EMFCompare*. A segunda camada denomina-se DCL-KDM e representa os componentes que compõem a DCL-KDM.

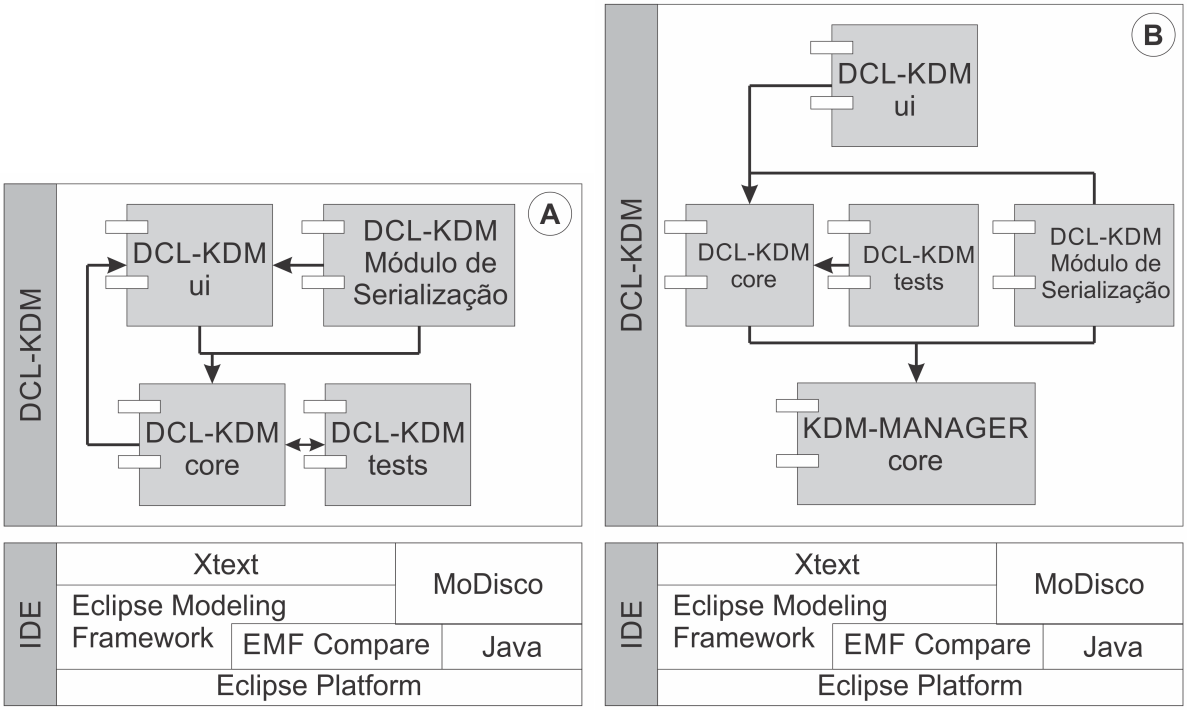

Figura B.3: Visualização gráfica e simplificada da arquitetura de componentes da DCL-KDM original (Item A) e atual (Item B). Fonte: Elaborado pelo autor

Na Figura B.4 pode ser observado o diagrama de classes e pacotes do módulo de Serialização original. Inicialmente, a implementação do protótipo era composta somente por duas classes. A classe `Activator` era responsável por realizar a inicialização do componente para o Eclipse IDE. A classe `ReadinDSLView` era a responsável por realizar toda a implementação do módulo de Serialização bem como sua interface com a DCL-KDM.

Conforme o andamento da realização da reengenharia, observou-se a necessidade de se realizar modificações na arquitetura da DCL, referente ao componente "DCL-KDM Módulo de Serialização" e o componente "DCL-KDM ui". Dessa forma, adaptou-se a interface do módulo de Serialização a interface da DCL-KDM podendo-se desacoplar a implementação do módulo de Serialização. Observou-se também que diversas atividades poderiam ser melhoradas, modularizadas e facilitadas com o uso de uma API. Portanto, optou-se neste momento pelo

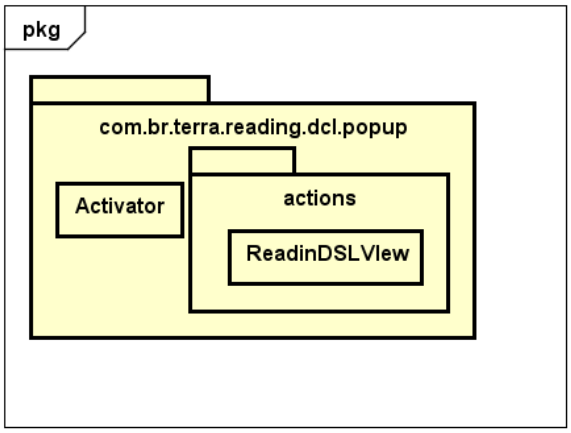

Figura B.4: Diagrama de classes e pacotes da versão original do módulo de serialização. Fonte: Elaborado pelo autor

uso da API "KDM-MANAGER" que facilita e auxilia a implementação e uso de elementos do KDM em Java. Na Figura B.3 - Item B, pode ser observado como a arquitetura atual da DCL-KDM ficou após sua reengenharia. Em relação ao código, observou-se diversos itens a serem continuados e pesquisados. Conforme foi sendo realizada a reengenharia do módulo, preocupou-se com atributos de qualidade como a reusabilidade, facilidade de uso, facilidade de entendimento e evoluibilidade, dessa forma, na Figura B.5 pode ser observada a versão atual da DCL-KDM.

Conforme pode ser observado na Figura B.5, a única classe do protótipo que realizava a serialização da arquitetura planejada deu lugar a 15 novas classes, uma enumeração e duas interfaces. Visando os atributos de qualidade mencionados, novas classes foram criadas e preparadas para serem evoluídas facilmente. A classe `Activator` ainda é a responsável por realizar a inicialização do componente para o Eclipse IDE, porém a única classe `ReadinDSLView` deu lugar as demais. A classe `DCL2KDM` é a responsável por coordenar a serialização, definindo o fluxo de processos a serem realizados para a serialização da arquitetura planejada. A classe `ReaderDCLEditor` é a responsável por realizar a leitura da especificação em DCL-KDM a partir do editor de texto.

A classe `ArchitecturalGenerator` é responsável por realizar a geração dos elementos arquiteturais em KDM bem como sua composição, enquanto a classe `RestrictionsGenerator` é a responsável por realizar a geração das restrições em formato de relacionamentos no KDM. Conforme ilustrado anteriormente, existem um total de 7 tipos de restrições no qual são possíveis na DCL-KDM. Dois deles são automaticamente gerados e cinco deles são manualmente gerados durante a especificação. Dessa forma, pensando-se no padrão de projeto *Strategy*, desenvolveu-se uma interface denominada `PatternRestrictionGenerator`, que representa a estratégia de

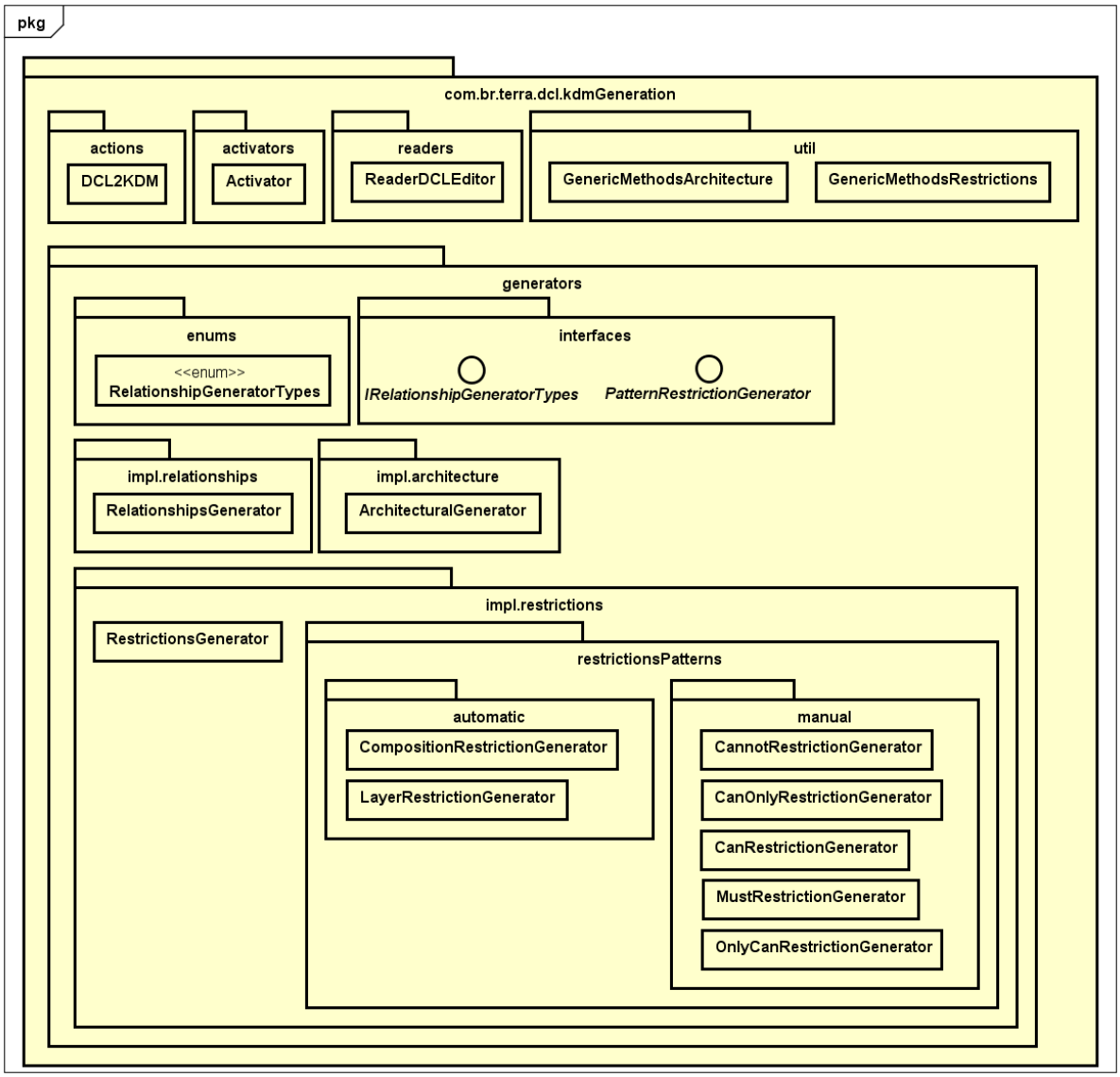

Figura B.5: Diagrama de classes e pacotes da versão atual do módulo de serialização. Fonte: Elaborado pelo autor

geração de restrições. Portanto, tem-se uma classe para cada um dos 7 tipos de regras gerados dentro do pacote `restrictionsPatterns`.

Conforme comentado anteriormente e mapeado na Tabela B.2, cada tipo de restrição pode variar em 10 opções de acessos, sendo cada opção representada por um diferente conjunto de metaclasses. Desse modo, fez-se a necessidade de outra estrutura auxiliar para realizar a composição das metaclasses para cada uma das opções de acessos. Essa estrutura auxiliar, diferentemente do conceito padrão do *Strategy*, foi implementada de uma forma complementar, utilizando-se uma enumeração.

Sendo assim, tem-se a interface denominada `IRelationshipGeneratorTypes`, no qual é implementada pela enumeração `RelationshipGeneratorTypes`, onde cada item da enumeração

realiza sua própria implementação da interface, facilitando assim a leitura e evoluibilidade do mapeamento realizado e demonstrado pela Tabela B.2.

Por fim, a última evolução exemplificada neste apêndice foi a inclusão de uma nova restrição entre os elementos arquiteturais. Tal restrição foi determinada pela diferença básica observada entre a DCL-KDM e a DCL de Terra e Valente (2009). No trabalho anterior a este, não é comentado sobre essa diferença, porém, ao se comparar e analisar as duas ADLs pode-se observar a diferença entre o formato de visualizar as restrições. Para a DCL (Terra; Valente, 2009), ao se declarar restrições arquiteturais entre os módulos definidos na especificação, o engenheiro deve se atentar que a especificação parte do princípio que todos os relacionamentos entre os módulos especificados são permitidos. Sendo assim, o engenheiro deve apenas especificar quais são os relacionamentos que não são permitidos durante sua especificação.

Já para a DCL-KDM, ao se declarar restrições arquiteturais entre os elementos arquiteturais definidos na especificação, o engenheiro deve se atentar que a especificação parte do princípio que todos os relacionamentos entre os elementos especificados são negados, ou seja, não permitidos. Sendo assim, o engenheiro deve apenas especificar quais são os relacionamentos que são permitidos ou estritamente proibidos. Essa diferença entre as duas ADLs, apesar de importante e mudar a lógica de especificação das restrições, pelos testes realizados não impactaram significativamente para o sucesso ou fracasso da especificação. Esse tipo de diferença pode impactar apenas com base no estilo do sistema empregado. Ou seja, por um lado, se o sistema possui muitas restrições utilizando a DCL-KDM pode ser ligeiramente mais fácil a especificação, uma vez que a princípio, toda comunicação não é permitida. Por outro lado, se o sistema possui poucas restrições e muitas permissões, a utilização da DCL pode ser ligeiramente mais fácil, uma vez que a princípio, toda a comunicação é permitida.

Com essa diferença em mente e as restrições entre ambas as ADLs, observou-se um problema ao se especificar comunicações simples em DCL-KDM. Esse problema que foi a motivação e a necessidade dessa evolução. A versão original da DCL-KDM não permitia a elaboração de comunicações múltiplas de maneira simples. Isto é, um exemplo do tipo "componente A pode se comunicar com os componentes B e C" não era possível de se especificar na DCL-KDM. Isto se deve ao fato de a versão original da DCL-KDM apenas oferecer a restrição do tipo "`nameElement`" `can-` "`accessType`" `-only` "`nameElement`" e do tipo `only` "`nameElement`" `can-` "`accessType`" "`nameElement`". Restrições tais quais representam que um elemento arquitetural pode acessar somente um outro elemento arquitetural e, somente um elemento arquitetural pode acessar um outro elemento arquitetural, respectivamente. Com o uso de apenas esses dois tipos de restrições que permitem a comunicação entre elementos arquiteturais, um

exemplo como o citado acima não seria possível de se especificar. Isto se deve ao fato que em ambos os casos as regras entrariam em conflito, uma vez que representam que somente um determinado elemento pode se comunicar com outro e vice-versa.

Apendice C

EVOLUÇÕES REALIZADAS NA ETAPA EXTRAÇÃO DA ARQUITETURA ATUAL

Em um estudo mais aprofundado sobre a segunda etapa da abordagem, incluindo suas características, implementação e utilização, observou-se que a mesma possuía problemas em seu algoritmo de mapeamento. Desse modo, se fez necessário algumas evoluções em sua implementação. Foram realizadas diversas evoluções e correções e cada uma tem suas motivações e justificativas, dessa forma, a seguir são apresentadas as principais evoluções realizadas na segunda etapa da abordagem.

Uma das evoluções realizadas foi a correção do mapeamento para as interfaces e a inclusão da possibilidade de mapeamento de enumerações. Inicialmente, o mapeamento apresentava Pacotes, Classes e Interfaces, porém, apenas utilizava pacotes e classes para efetivamente realizar o mapeamento, isto é, interfaces apesar de listada não era mapeada corretamente. Dessa forma, foi necessário realizar uma reestruturação do código e incluir as interfaces e enumerações para que fossem corretamente mapeadas.

Outra evolução realizada foi o ampliamento para o mapeamento completo de metaclasses que representam os relacionamentos em KDM. Conforme o mesmo caso da etapa anterior, a abordagem original apenas utilizava 8 das 34 metaclasses que o KDM proporciona para uso. Dessa forma, o algoritmo foi modificado para abranger todas as metaclasses realizando um mapeamento completo do sistema. Ambas as evoluções citadas podem ser observadas nas descrições realizadas no Capítulo 4.

Outra evolução realizada foi a reengenharia do protótipo original. A Figura C.1 demonstra a implementação original. Como pode ser observado na figura, a implementação era composta

por cinco classes, sendo elas uma classe de interface gráfica, duas ações para menus do Eclipse IDE, uma para domínio e uma classe útil. A classe `MapArchitectureElementsToCodeElements`, é a implementação para a chamada da classe de interface gráfica (`MappingArchitectureElements`) que realizava o mapeamento inicial entre os elementos de código-fonte e os elementos arquiteturais. A classe `MapItem` é uma classe de domínio utilizada durante o processo de mapeamento inicial realizado pela interface gráfica. A classe `ActionRecoveryArchitecture` é a implementação para a chamada de execução do algoritmo de mapeamento. Por fim, a classe `ReadingKDMFile` é uma *godclass* de utilitários, ou seja, uma classe que contém inúmeros métodos que realizam todo o processamento dos algoritmos.

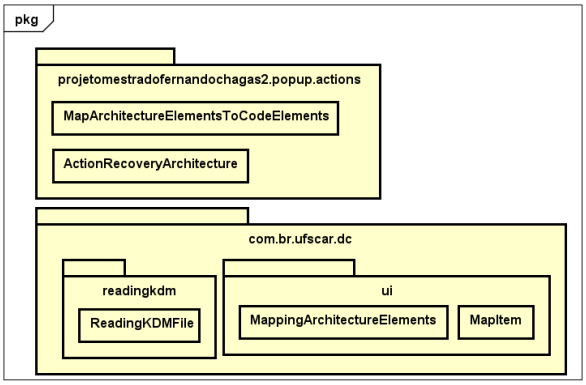

Figura C.1: Diagrama de classes e pacotes da versão original do mapeamento da Arquitetura Atual. Fonte: Elaborado pelo autor

Conforme citado anteriormente (Capítulo 4 - Seção 4.3), a implementação foi dividida em três componentes. A Figura C.2 e a Figura C.3 denotam parcialmente a implementação dos componentes UI e Core que substituíram a implementação original. Em ambas as figuras estão representados diagramas de pacotes e classes referentes a implementação do mapeamento da arquitetura atual do sistema. Neste caso, as cinco classes que realizavam o processo todo deram lugar a onze novas classes.

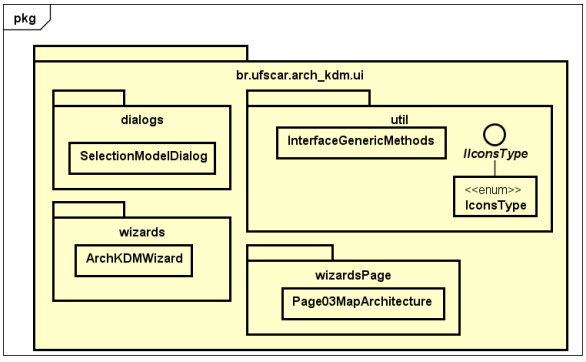

Figura C.2: Diagrama de classes e pacotes do componente UI. Fonte: Elaborado pelo autor

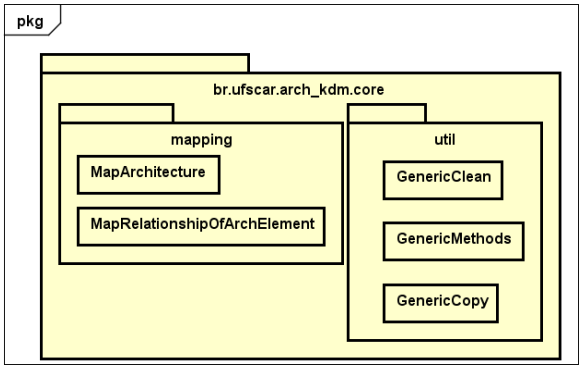

Figura C.3: Diagramas de classes e pacotes do componente Core. Fonte: Elaborado pelo autor

Na Figura C.2 encontram-se cinco novas classes, e uma interface. A classe `ArchKDMWizard` e `Page03MapArchitecture` são as responsáveis pela interface gráfica do mapeamento entre os elementos arquiteturais e os elementos de código fonte. Ambas as classes dependem essencialmente do componente core para realizar suas funções adequadamente. As classes dentro do pacote `util` e do pacote `dialogs` são classes auxiliares para facilitar e auxiliar na composição da interface gráfica do plug-in para o IDE Eclipse.

Na Figura C.3 encontram-se cinco novas classes. O mapeamento da arquitetura atual foi analisado e separado em duas classes principais. A classe `MapArchitecture` é a responsável por coordenar e realizar o mapeamento entre os elementos arquiteturais. A classe `MapRelationshipOfArchElement` é a responsável por coordenar e realizar o mapeamento dos relacionamentos entre os elementos arquiteturais. As classes dentro do pacote `util` são classes que contém métodos genéricos, que auxiliam não só o mapeamento da arquitetura atual, mas também os algoritmos das demais etapas da abordagem.

Por fim, a ultima evolução citada nesse apêndice é a elaboração e modificação do algoritmo de mapeamento. A elaboração e a modificação foram necessárias devido a evoluções anteriores que demandavam alterações no algoritmo para serem corretamente aplicadas. A evolução e os algoritmos já foram apresentados no Capítulo 4 na Seção 4.3.2.

Apendice D

EVOLUÇÕES REALIZADAS NA ETAPA CHECAGEM DE CONFORMIDADE ARQUITETURAL

Em um estudo mais aprofundado sobre a terceira etapa da abordagem, incluindo suas características, implementação e utilização, observou-se que a mesma possuía problemas em seu algoritmo de checagem de conformidade. Desse modo, se fez necessário algumas evoluções em sua implementação. Foram realizadas diversas evoluções e correções e cada uma tem suas motivações e justificativas, dessa forma, a seguir são apresentadas as principais evoluções realizadas na segunda etapa da abordagem.

De modo geral, as evoluções desta etapa vão de encontro com as evoluções realizadas nas etapas anteriores. Dessa forma, sendo iguais por exemplo a da etapa anterior, sendo elas i) a correção do mapeamento para as interfaces e a inclusão da possibilidade de mapeamento de enumerações e; ii) o mapeamento completo de metaclasses que representam os relacionamentos em KDM.

Outra evolução realizada foi a re-implementação do protótipo original. A Figura D.1 demonstra a implementação original. Como pode ser observado na figura, a implementação era composta apenas de duas classes. A classe `ArchitectureComplianceCheking` é a implementação da chamada de execução do algoritmo de mapeamento pelos menus do Eclipse IDE. A classe `ReadingKDMFile` é uma *godclass* de utilitários, ou seja, uma classe que contém inúmeros métodos que realizam todo o processamento dos algoritmos. Um desses métodos é o responsável pela realização da CCA.

Conforme citado anteriormente (Capítulo 4 - Seção 4.3), a implementação foi dividida em três componentes. A Figura D.2 denota parcialmente a implementação do componente core.

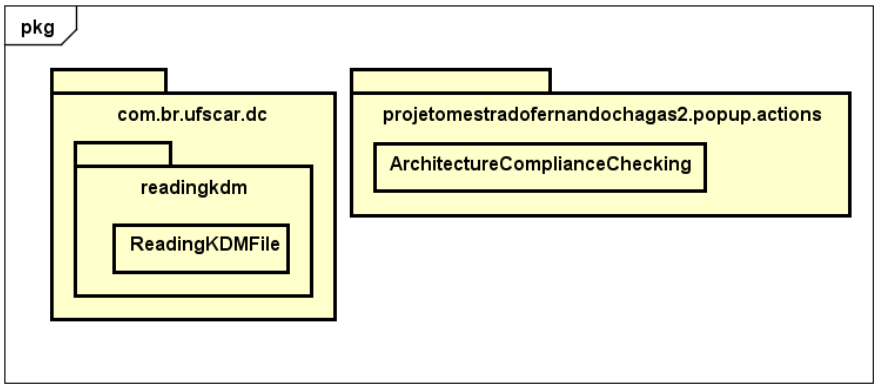

Figura D.1: Diagramas de classes e pacotes da versão original da Checagem de Conformidade. Fonte: Elaborado pelo autor

Essa figura é um diagrama de pacotes e classes da implementação atual dessa etapa.

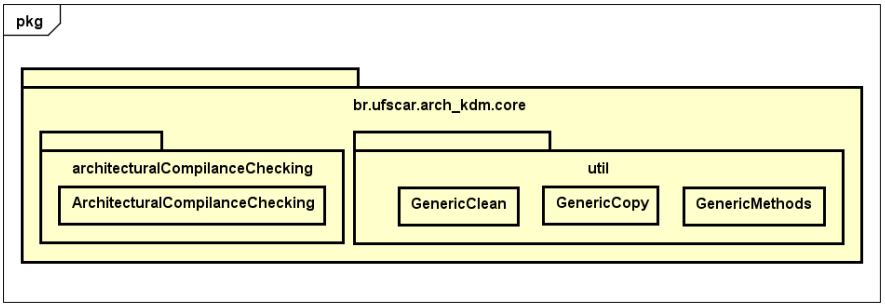

Figura D.2: Diagramas de classes e pacotes do componente Core referentes a checagem de conformidade. Fonte: Elaborado pelo autor

Na Figura D.2 encontram-se quatro classes. As classes dentro do pacote `util` são classes que contém métodos genéricos que auxiliam não só a CCA, bem como os algoritmos das demais etapas da abordagem. A classe `ArchitecturalCompilanceChecking` é a responsável por realizar a CCA. Essa classe é única e exclusivamente para realizar a checagem, diferentemente da versão anterior, que eram algoritmos espalhados dentro de uma *godclass* chamada `ReadingKDMFile`.

Outra evolução citada nesse apêndice é a elaboração e modificação do algoritmo de checagem de conformidade. A elaboração e a modificação foram necessárias devido a evoluções anteriores que demandavam alterações no algoritmo para serem corretamente aplicadas. A evolução e os algoritmos já foram apresentados no Capítulo 4 na Seção 4.3.3.

Por fim, a última evolução apresentada neste apêndice foi a realização de uma implementação de interface gráfica para o protótipo. Na versão inicial do protótipo desenvolvido, não existiam interfaces gráficas que auxiliassem o processo, com exceção de uma interface de mapeamento. Todas as etapas e fases eram compostas por itens nos menus do Eclipse IDE, sem explicações ou

identificações. Portanto, o protótipo não era passível de uso por pessoas que desconhecessem sua implementação e estruturação.

Dessa forma, observou-se a necessidade de realização de toda uma reestruturação que torna-se possível a utilização da ferramenta por qualquer engenheiro de software. Portanto, criou-se o componente `ui` da arquitetura demonstrada no Capítulo 4. As Figuras D.3 e D.4 são diagramas de classes e pacotes que representam a implementação original dos itens nos menus do Eclipse IDE e a implementação atual de utilização no formato de Wizard do Eclipse IDE.

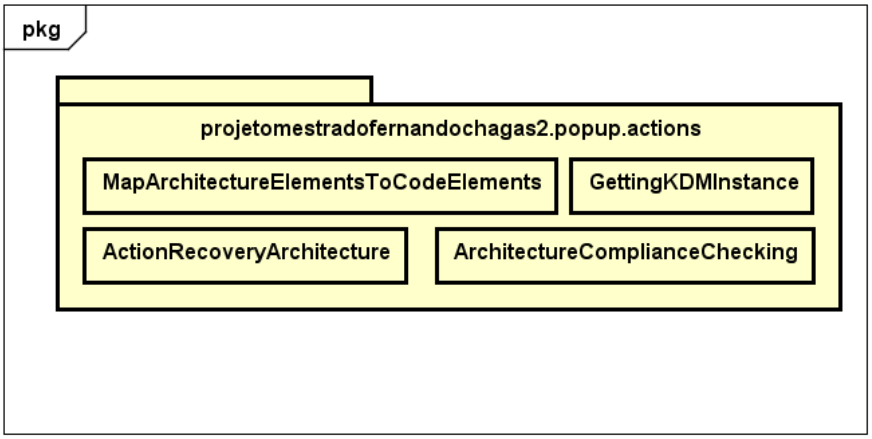

Figura D.3: Diagramas de classes e pacotes da Interface Gráfica da versão original. Fonte: Elaborado pelo autor

Conforme pode ser observado na Figura D.3, existiam apenas quatro classes, sendo cada uma responsável por uma atividade. A classe `GettingKDMInstance` representa a geração da arquitetura planejada. A classe `MapArchitectureElementsToCodeElements` representa o mapeamento inicial com o auíilio do engenheiro de software. A classe `ActionRecoveryArchitecture` representa a geração da arquitetura atual. Por fim, a classe `ArchitectureComplianceCheking` representa a realização da CCA. Apesar de os itens nos menus do Eclipse IDE estarem separados não existia nenhuma informação que orientasse o usuário durante o uso. Portanto, levando usuários que não conhecessem à implementação a utilização errônea.

Com o intuito de corrigir essa limitação, criou-se um novo componente composto de um wizzard que orienta e auxilia o usuário a realizar a CCA corretamente. Conforme pode ser observado na Figura D.4, as classes dentro do pacote `wizards` e `wizardsPage` são as implementações do *wizard* contendo uma página para cada tipo de atividade realizada no processo. Uma visualização da Interface Gráfica e de sua utilização é encontrada no Capítulo 5, no qual é apresentado um cenário de uso da abordagem.

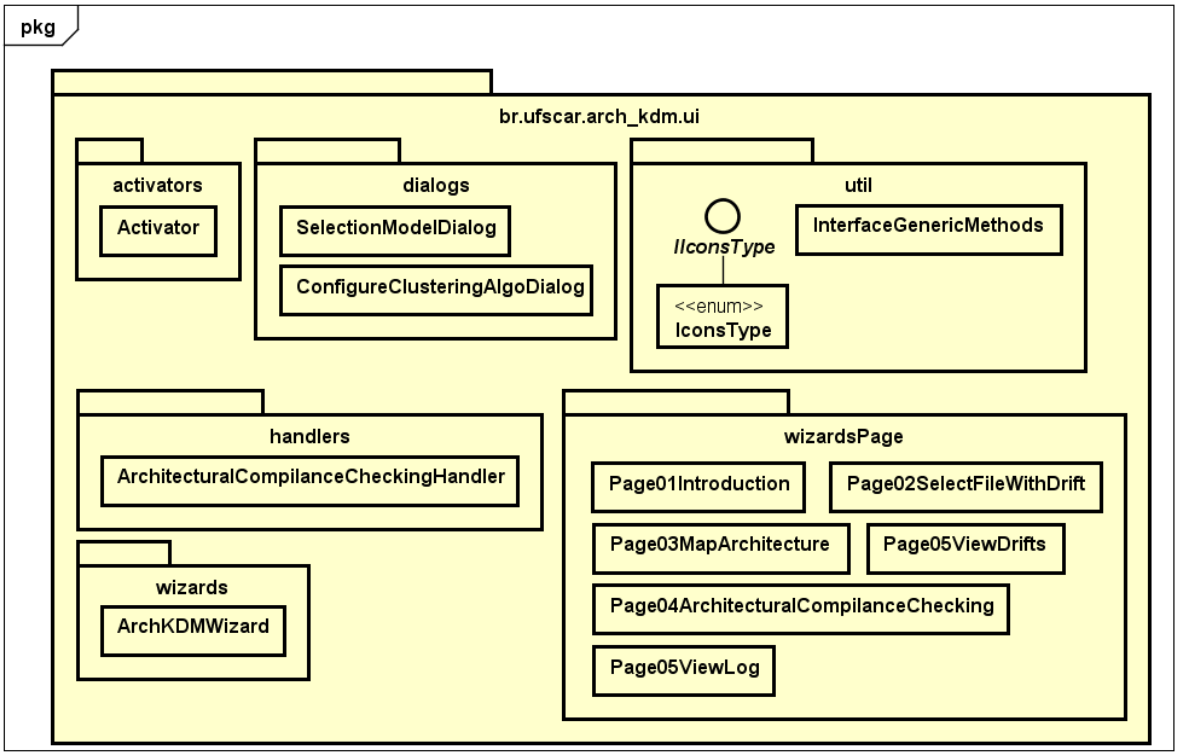

Figura D.4: Diagramas de classes e pacotes do componente UI. Fonte: Elaborado pelo autor