TESE DE DOUTORADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

“A REFERENCE ARCHITECTURE FOR DESIGNING ADM-BASED MODERNIZATION TOOLS”

ALUNO: Bruno Marinho Santos
ORIENTADOR: Prof. Dr. Valter Vieira de Camargo

São Carlos
Fevereiro/2023

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

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

A REFERENCE ARCHITECTURE FOR DESIGNING ADM-BASED MODERNIZATION TOOLS

BRUNO MARINHO SANTOS

ORIENTADOR: PROF. DR. VALTER VIEIRA DE CAMARGO

São Carlos - SP

Fevereiro/2023

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

A REFERENCE ARCHITECTURE FOR DESIGNING ADM-BASED MODERNIZATION TOOLS

BRUNO MARINHO SANTOS

Doctoral dissertation submitted to the Graduate Program in Computer Science at the Federal University of São Carlos, as part of the requirements for obtaining the title of Doctor of Computer Science.
Concentration Area: Software Engineering.
Advisor: Prof. Ph.D. Valter Vieira de Camargo

São Carlos - SP
February/2023

UNIVERSIDADE FEDERAL DE SÃO CARLOS

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

Folha de Aprovação

Defesa de Tese de Doutorado do candidato Bruno Marinho Santos, realizada em 17/02/2023.

Comissão Julgadora:

Prof. Dr. Valter Vieira de Camargo (UFSCar)

Prof. Dr. Fabiano Cutigi Ferrari (UFSCar)

Prof. Dr. Daniel Lucrédio (UFSCar)

Profa. Dra. Maria Istela Cagnin Machado (UFMS)

Prof. Dr. Glauco de Figueiredo Carneiro (UFS)

O Relatório de Defesa assinado pelos membros da Comissão Julgadora encontra-se arquivado junto ao Programa de Pós-Graduação em Ciência da Computação.

To my husband, mother, father, sister and brother.

We stand undefined
Can't be drawn with a straight line
This will not be our ending
We are alive...
Don't you dare surrender"

Imperfection, *Evanescence*

Acknowledgements

First, I thank God for all the opportunities and for the people that have been placed in my life. To my husband André for being a loving supporter and strength.

To my family, especially my mother Magaly, my father, my sister Tayana and my brother Julio Cezar.

To my cousins Cristiano and Isabela.

To my friends who always held hands with me (Nair, Matheus, Mirela, André, Jullyane, Joyce).

To my advisor, Valter Vieira de Camargo, for his dedication, help, professionalism and guidelines.

To my psychiatrist Dr. Leonardo and psychologist Beatriz for the mental support.

CNPQ for financial support.

I thank everyone who, directly or indirectly, contributed to my formation.

Abstract

Software systems are constantly changing and this process generally makes it increasingly difficult for organizations to maintain. To help in this context, *Object Management Group* (OMG) proposed *Architecture-Driven Modernization* (ADM) that defends the realization of reengineering processes following the model-oriented architecture. One of ADM's main contributions is its sets of metamodels and its conceptual architecture. The construction of modernization tools that use the concepts of ADM have a greater chance of being interoperable. However, there are not many works in the literature that help in the construction of modernization tools based on ADM. Thus, the main objective of this Ph.D. research was to develop a Reference Architecture (RA) that supports the creation and evolution of modernization tools that are based on ADM's concepts and standards. The RA is formed by a set of diagrams representing architectural visions. For the creation of the RADM that is our Reference Architecture for ADM-based Modernization Tools, ProSA-RA was used, a process that systematizes the design, representation and evaluation of reference architectures. This Ph.D. research was evaluated through a questionnaire with ADM specialists (software developers and researchers) and software architects whose main objective was to obtain feedbacks on the acceptance of the Reference Architecture that was developed. The questionnaire that was applied brought as main results the following conclusions: (i) 100% of the participants agreed that the RA is clear and well described and (ii) 75% agreed that the RA is useful for create instances of different types of modernization tools.

Keywords: Architecture-Driven Modernization, Modernization Tool, Reference Architecture, Taxonomy, ProSA-RA.

Resumo

Sistemas de software estão em constante mudanças e esse processo geralmente faz com que se tornem cada vez mais difíceis de serem mantidos pelas organizações. Para ajudar nesse contexto a *Object Management Group* (OMG) propôs a *Architecture-Driven Modernization* (ADM) que defende a realização dos processos de reengenharia seguindo o padrão da arquitetura dirigida a modelos. Uma das principais contribuições da ADM são seus conjuntos de metamodelos e sua arquitetura conceitual. A construção de ferramentas de modernização que se utilizam dos conceitos da ADM possuem uma maior chance de serem interoperáveis. Contudo, não existem muitos trabalhos na literatura que auxiliam na construção de ferramentas de modernização baseadas em ADM. Dessa forma, o principal objetivo desta pesquisa foi desenvolver uma Arquitetura de Referência (AR) que apoia a criação e evolução de ferramentas de modernização que são baseadas nos conceitos e padrões da ADM. A AR é formada por um conjunto de diagramas representando visões arquiteturais. Para a criação da RADM que é nossa Arquiterura de Referência para ferramentas de modernização baseadas em ADM, foi utilizado o ProSA-RA, um processo que sistematiza o projeto, representação e avaliação de arquiteturas de referência. Este doutorado foi avaliado por meio de um questionário com especialistas em ADM (desenvolvedores de software e pesquisadores) e arquitetos de software que teve como principal objetivo obter comentários sobre a aceitação da Arquitetura de Referência que foi desenvolvida. O questionário que foi aplicado trouxe como principais resultados as seguintes conclusões: (i) 100% dos participantes afirmaram que a AR é clara e bem descrita e (ii) 75% concordaram que a AR é útil para criar instâncias de diferentes tipos de ferramentas de modernização.

Palavras-chave: Architecture-Driven Modernization, Ferramenta de Mondernização, Arquitetura de Referência, Taxonomia, ProSA-RA.

List of Figures

1.1 Main scenario for using the results of this Ph.D research. 5

2.1 General structure of ADM 10
2.2 Architecture-Driven Modernization Blueprint. 12
2.3 Layers, packages and separation of concerns in KDM (Adapted from KDM (2011)). 14
2.4 KDM domains and compliance levels KDM (2011). 16
2.5 Reference Architecture Composition 19
2.6 Quality attributes for a Software product 20
2.7 Steps to create an RA in a way that the checklist can be applied 22

3.1 KDM-RE's architecture (Durelli, 2016). 29
3.2 Approach to business process recovery - MARBLE (Pérez-Castillo et al., 2011a). 31
3.3 Approach to transforming legacy systems to the cloud - CloudMig (Frey e Hasselbring, 2011). 31
3.4 Workflow of refactorings proposed by Einarsson e Neukirchen (2012). . . . 33
3.5 RACOoN plug-in's architecture (Van Der Straeten et al., 2007). 34
3.6 Overview of the use case refactoring tool proposed by Ren et al. (2004). . . 36
3.7 Refactoring workflow proposed by Moghadam e Cinneide (2012). 37

4.1 Modernization Tools Taxonomy - Conceptual Diagram 59
4.2 Methodology to establish a Taxonomy 61

5.1 MVC view 71
5.2 Internal Components view 72
5.3 ADM Pipes and Filters Architectural Overview 75
5.4 Reverse Engineering view 76
5.5 General Activity Diagram View 77
5.6 Instance Manager Activity Diagram View 78
5.7 Flaws Detector Activity Diagram View 79
5.8 Metamodel Instance Discoverer View 81

5.9 Metric Calculator Activity Diagram View 82
5.10 Reverse Engineering Component Diagram View 83
5.11 Restructuring view 84
5.12 General Activity Diagram View 84
5.13 Instance Manager Activity Diagram View 85
5.14 Refactoring Activity Diagram View 86
5.15 Restructuring Component Diagram View 88
5.16 Forward Engineering View 88
5.17 General Activity Diagram View 89
5.18 Metamodel Instance and Source Code Discovery View 90
5.19 Metric and Measurement Calculation View 91
5.20 Forward Engineering Component Diagram View 92

List of Tables

2.1 Quality attributes of a Software 21
2.2 Stages, Number of questions and Evaluation Topics of the FERA checklist 24
2.3 Classification of model transformations 26

4.1 Modernization Scenarios 58
4.2 Comparison between MT's type 60
4.3 Concepts elicited from Step T-1 61
4.4 Classification of Computational Support for Software Modernization 66

5.1 Requirements 96
5.2 Acceptance of the Reference Architecture views 98
5.3 Overall acceptance of the Reference Architecture 98

List of Abbreviations and Acronyms

ADM *Architecture-Driven Modernization*
ADM-TF *Architecture-Driven Modernization Task Force*
AEP *Automated Enhancement Points*
AFP *Automated Function Points*
ASTM *Abstract Syntax Tree Metamodel*
ATAM *Architecture Tradeoff Analysis Method*
ADM-MT ADM-Modernization Tools
Common-MT Common Modernization Tools
RA Reference Architecture
FRA Futuristic Reference Architecture
PRA Practical Reference Architecture
CEM *Cloud Environment Model*
CIM *Computing Independent Model*
DLs *Description Logics*
FERA *Framework for Evaluation of Reference Architecture*
MT Modernization Tool
KDM *Knowledge Discovery Metamodel*
KDM-MT KDM Modernization Tools
KDM-RE *Knowledge Discovery Model - Refactoring Environment*
OMG-MT OMG Modernization Tools
M2M *Model-to-Model Transformations*
MDA *Model Driven Architecture*
MOF *Meta Object Facility*
OCL *Object Constraint Language*
OMG *Object Management Group*
PIM *Platform Independent Model*
PSM *Platform Specific Model*
QVT *Query/View/Transformation*
QVTO *QVT Operational Mappings*
SAAS *Software as a Service*
SBVR *Semantics for Business Vocabulary and Rules*

SMM *Structured Metrics Metamodel*
SOA *Service-Oriented Architecture*
SM *Systematic Mapping*
UML *Unified Modelling Language*
SRM *Structured Refactoring Metamodel*

Contents

1 Introduction 1
- 1.1 Context 1
- 1.2 Problem Statement 3
- 1.3 Contributions 4
- 1.4 Usage Scenario 5
- 1.5 AdvanSE Group Research 6
- 1.6 Research Roadmap 6
- 1.7 Outline 6

2 Background 8
- 2.1 Software Modernization 8
- 2.2 Architecture-Driven Modernization 9
- 2.3 Knowledge Discovery Metamodel 13
  - 2.3.1 KDM Compliance and Domains 15
- 2.4 Reference Architectures 17
  - 2.4.1 Reference Architecture Types 18
  - 2.4.2 Reference Architecture Attributes 19
  - 2.4.3 Reference Architecture Evaluation 21
- 2.5 Model Refactoring 24
- 2.6 Final Considerations 26

3 Related Works and Systematic Mappings 28
- 3.1 Modernization tools that use ADM standards 28
- 3.2 Tools that apply refactorings in UML 32
- 3.3 Systematic Mapping 39
  - 3.3.1 Systematic Mapping on ADM 39
    - 3.3.1.1 Group 1 - Discussions around ADM papers 41
    - 3.3.1.2 Group 2 - Modernization Tools papers 43
  - 3.3.2 Systematic Mapping on Code and Model Refactoring tools 47
- 3.4 Final Considerations 53

4 A Taxonomy for Classifying Modernization Tools in ADM Context **54**
4.1 Initial Considerations 54
4.2 The Taxonomy 55
4.3 Methodology for Building the Taxonomy 62
4.3.1 Step T-1: Information Source Investigation 62
4.3.2 Step T-2: Artifacts Analysis and Categorization 63
4.3.3 Step T-3: Taxonomy Establishment 64
4.3.4 Step T-4: Taxonomy Evaluation 64
4.4 Threats to validity 67
4.5 Final Considerations 67

5 A Reference Architecture for ADM-Based Modernization Tools **68**
5.1 Initial Considerations 68
5.2 The Reference Architecture 68
5.3 Structural Views 70
5.3.1 MVC View 70
5.3.2 Internal Components View 72
5.4 Data Flow View 74
5.4.1 Pipes and Filter View 74
5.5 Dynamic Views 76
5.5.1 Reverse Engineering Views 76
5.5.1.1 General Activity Diagram View 76
5.5.1.2 Instance Manager Activity Diagram View 78
5.5.1.3 Flaws Detector Activity Diagram View 79
5.5.1.4 Metamodel Instance Discoverer View 80
5.5.1.5 Metric Calculator Activity Diagram View 80
5.5.1.6 Reverse Engineering Component Diagram View 82
5.5.2 Restructuring Views 83
5.5.2.1 General Activity Diagram View 83
5.5.2.2 Instance Manager Activity Diagram View 85
5.5.2.3 Refactoring Activity Diagram View 85
5.5.2.4 Restructuring Component Diagram View 87
5.5.3 Forward Engineering Views 87
5.5.3.1 General Activity Diagram View 88
5.5.3.2 Metamodel Instance and Source Code Discovery View . . 89
5.5.3.3 Metric and Measurement Calculation View 91
5.5.3.4 Forward Engineering Component Diagram View 92
5.6 RA usage Guidelines 92
5.7 Methodology Employed 93
5.7.1 Information Source Investigation 94
5.7.2 Architectural Analysis and Requirements 95
5.8 Evaluation 97
5.8.1 Evaluating the RA Acceptance 97
5.8.2 Evaluating the RA Overall acceptance 98

5.8.3 Evaluating Discussions 99
5.9 Threats to Validity 99
5.10 Final Considerations 100

6 Conclusions 101
6.1 Contributions 101
6.2 Limitations 103
6.3 Future Work 103
6.4 List of Publications 103
6.5 Next Publications 105

Bibliography 106

A Details of Systematic Mapping on ADM 116
A.1 Final selected papers: 116
A.1.1 Group 1: Discussions around ADM papers 116
A.1.2 Group 2 - Modernization Tools papers 118
A.2 ADM Search String: 122

B Details of Systematic Mapping on Code and Model Refactoring Tools 123
B.1 Final selected papers: 123

C Taxonomy's Evaluation Strategy 127

Chapter 1

Introduction

1.1 Context

Software reengineering is theme that always will be of some importance in software engineering field (Ulrich e Newcomb, 2010a). This happens because as the time passes, any system become legacy; making the maintenance very difficult, the integration with modern systems hard and the productivity of the team low. However, although the topic of software reengineering has been researched for several years, some studies shows that the number of reengineering projects that fails is still high (Kiran Mallidi et al., 2021; Wolfart et al., 2021). Other studies presented by the Standish group also reveals that many companies suffer with legacy systems whose maintenance costs are extremely high (StandishGroup., 2014).

With this in mind, two main alternatives appear. The first is the complete replacement of the system with a new one and the second is the conduction of a reengineering process with the objective of modernizing the existing system and making maintenance and evolution tasks go back to acceptable levels of cost and effort (Arnold, 1993; Ludewig e Lichter, 2023).

Software Modernization (aka Software Reengineering) is an alternative for extending the life of the system - it involves three main processes: reverse engineering, restructuring and forward engineering. With the help of these three processes, it is possible to recover

knowledge from the existing legacy system, performing refactorings and optimizations, and finally, obtaining an improved version of the same system, preserving the knowledge and business rules (Chikofsky e Cross, 1990).

In 2003, the *Object Management Group* (OMG) created a task force called *Architecture-Driven Modernization Task Force* (ADM-TF) to investigate the large number of failures in reengineering/modernization projects. According to ADM-TF, the main problem is the lack of a standard model for representing systems to be modernized. Without this standard, companies need to develop their own modernization solutions (like Modernization Tools and algorithms) that only act over their proprietary models, making the sharing of these solutions a very difficult task among tools. The problem with proprietary solutions is that the knowledge around modernization is not captured, shared and largely disseminated. For example, so that a modernization tool can import refactoring algorithms for applying them in their own projects, these algorithms must have been developed in such a way that it know/recognize a common structure (a standard model), which must be also known internally by the tool.

In order to solve the standardization problem in the representation of existing system, the OMG task force proposed a metamodel called *Knowledge Discovery Metamodel* (KDM). KDM was created in order to be able to represent all the characteristics of a software system, encompassing both low-level implementation details (source code and system actions) and high-level concepts (architectural modules, conceptual elements , business processes and infrastructure). The main objective is to make this metamodel the most disseminated standard for representing systems in modernization tools. In other words, if researchers and professionals develop modernization solutions (mining and analysis algorithms, transformations, analytical tools, metrics and others) that know and act over KDM-represented systems instead of proprietary metamodels or specific languages, these solutions could be more easily reused/shared among all tools that recognize the KDM metamodel as their main form of representation (OMG, 2016).

Although OMG advocates adopting KDM as the base metamodel in modernization tools, it is not clear in the literature how KDM can be used in tools. To date, there is no knowledge available in the literature on how modernization tools should be designed in the context of ADM. What one can find are modernization tool architectures that are built without an explicit architectural model to guide (KDMAnalytics, 2017; Pérez-Castillo et al., 2011a).

A possible solution for this lack of guidelines is through the use of Reference Architectures (RA) (Bodziony e Wrembel, 2021; Martin et al., 2021; Rutledge e Italiaander, 2021; Tummers et al., 2021). A RA is formed by a set of artifacts that help in the development

of software solutions that aim to solve a given problem. With the help of an RA, it is possible to develop solutions to recurring problems in a standardized way, thus avoiding rework in the elaboration and design of a software system (Eickelmann e Richardson, 1996; Nakagawa et al., 2011). RA can be developed based on existing software solutions and/or knowledge of the literature, thus forming a set of practices and design decisions that assist in the development of new tools and in the modernization of existing solutions (Nakagawa et al., 2013).

As RA serve to assist in the development of systems in a given domain and to solve a specific problem. Research around RA usually involve the publication of guidelines, diagrams that provide architectural views and other artifacts that can contribute to the body of knowledge of the domain. A given RA may involve a set of quality attributes that are directly linked to the problem to be solved. These attributes are fundamental for choosing or not a RA and may vary from usability to performance of the generated system. RA have built-in knowledge from other works that have already been carried out and also rely on the help of domain expert knowledge, that is, one of the main benefits of their use is that it minimizes the incidence of problems that a developer may have when preparing a solution to the problem we are facing (Eickelmann e Richardson, 1996; Galster e Avgeriou, 2011).

1.2 Problem Statement

The main motivation for this thesis is the lack of guidelines on how to employ KDM, and other ADM metamodels, inside modernization tools. The problem of not having these clear and well-specified guidelines lead companies and researchers to develop modernization tools that only know and manipulate proprietary models, constraining the reuse of solutions that work with these models. Failure to reuse existing solutions can lead to the failure of systems modernization projects, such as the cases presented in research by Sneed (2005); StandishGroup. (2014); Ulrich e Newcomb (2010a).

One of the reasons why creating new ADM-based MT is challenging comes from the fact that it involves several modernization processes and each one of them has its peculiarities such as the model abstraction level, the purpose of the MT and how the components should behavior internally to provide an interoperability that could make possible its reuse.

Therefore, this doctoral research aims to answer the following question: *How do to design interoperable and modularized ADM-based modernization tools that incorporate modernization solutions?* A study that demonstrates how to use KDM (and other ADM metamodels) in modernization tools is important to increase the adoption by academic

research and business environments. The creation of tools that are concerned with concepts such as reuse and interoperability, defended by ADM, have more chances of being used by several researchers and practitioners and consequently would contribute to update the state of the art.

Therefore, we have proposed a reference architecture for building ADM-based modernization tools that is composed of i) Diagrams about MT structure/architecture and ii) A taxonomy that supports the MT classification.

1.3 Contributions

The main contribution of this Ph.D. research are two.

Contribution 1. *RADM - a Reference Architecture for ADM-based Modernization Tools:*

Several modernization tools were proposed in the literature, however, the publications that presented this kind of tool did not aim to systematize their development process. This scenario results in the lack of organization in the designing process. Aiming to mitigate this research problem, we developed a reference architecture for ADM-based MTs that gathers the knowledge of official ADM content and the knowledge presented in the literature in order to systematize the way that modernization tools are designed and developed. We propose a set of conceptual findings and diagrams containing the expected flows of each process of a modernization tool in the different categories that it can be applicable. This contribution is reported in Chapter 5 and in paper (Santos et al., 2019b).

Contribution 2. *A Taxonomy for Software Modernization Tools:*

Along the development of this project, we have noticed a lack of consensus about several terms involving software modernization. Therefore, another contribution of this thesis is a taxonomy to help in classifying modernization tools. A study was carried out in order to categorize knowledge regarding tools that were intended to support any of the stages of the modernization process. As a main result of this study, a taxonomy for modernization tools was developed whose main objective is to help the understanding of modernization tools and how they can be categorized according to the stage they support. This contribution was reported in Chapter 4 and an initial discussion about modernization tools in (Santos et al., 2018).

1.4 Usage Scenario

In this section, we describe a possible usage scenario for the reference architecture developed in this thesis. In Figure 1.1, the usage scenario is from the perspective of the Modernization Engineer. In this scenario, the engineer uses the reference architecture to develop his own modernization tool. The RA has all the necessary knowledge to make this new tool to have the selected quality attributes, such as interoperability and co-existence. Another important point is that using this same RA, a modernization engineer can also restructure an existing modernization tool, simply following the RA artifacts. At this level of use it is also possible to be the beneficiary of existing tools, since existing modules from other reusable tools are more easily reused as they follow the same base structure.

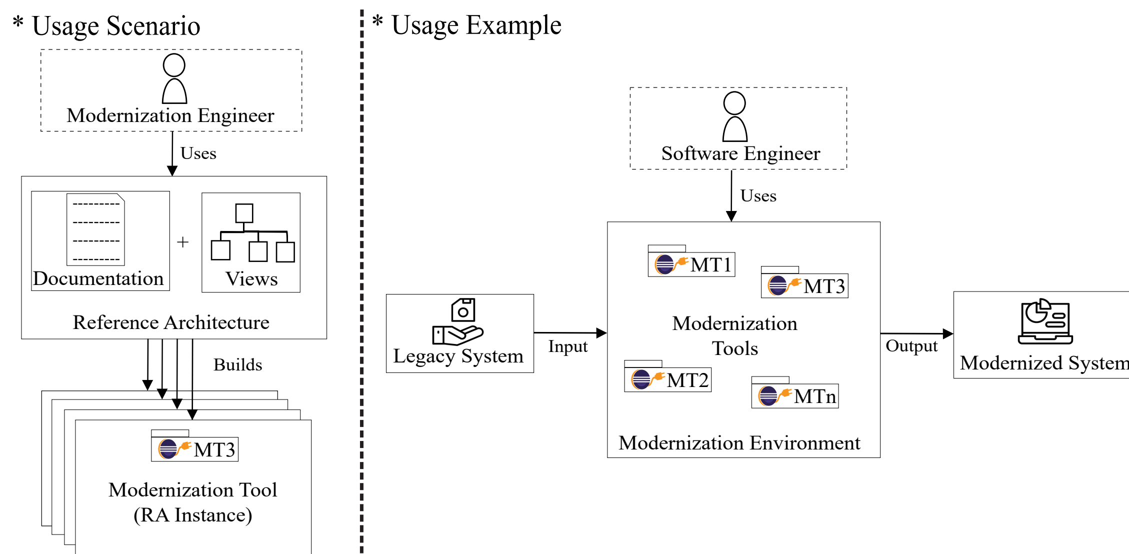

Figure 1.1: Main scenario for using the results of this Ph.D research.

In the second part of Figure 1.1 we present a possible applicability from the result of using our RA to build modernization tools. It is possible to see from the perspective of a Software Engineer how a modernization environment that was designed according to our reference architecture could benefit. In this usage example the software engineer uses a set of modernization tools to modernize a Legacy System into a Modernized System. As the RA can be used to support the creation of several MTs and usually the software engineer needs to use more than one MT we claim that these MT are part of a Modernization Environment, in which we could see several tools with specific roles supporting the modernization/reengineering projects. In this usage example, the RA is

behind the architectural concepts that were used to build the modernization environment, enabling an interoperability between the involved MTs.

1.5 AdvanSE Group Research

The AdvanSE group (http://advanse.dc.ufscar.br/), which is part of the computing department of the Federal University of São Carlos, has advanced considerably in the ADM/KDM subject and so far, works of Metrics have been proposed for KDM (Honda, 2014), Light and Heavy KDM Extensions (Santos, 2014), KDM Instance Mining (Santibáñez, 2013), KDM Refactorings (Durelli, 2016), Architectural Compliance Checking for KDM (Chagas, 2016), Cloud Computing API Constraint Analysis Algorithm for KDM (COSTA, 2017), KDM model to PSM model Transformation Engines (Angulo et al., 2018), DSL to Specify Planned Architectures of Adaptive Systems (San Martín e Camargo, 2021) and Architectural Conformance Checking (Landi et al., 2021). Other themes, such as the one contained in this proposal, are being researched in order to continue the studies already completed by the group. Thus, it is intended that this work can contribute to the state of the art in the subject[1].

1.6 Research Roadmap

Understand the state of the art in ADM. TODO

1.7 Outline

In addition to this chapter, this Ph.D. research is organized as follows:

- In Chapter 2, the main concepts related to system modernization advocated by the *Architecture-Driven Modernization* and its main metamodel *Knowledge Discovery Metamodel* are presented. This chapter also presents the refactoring for models and the concepts of reference architecture.
- In Chapter 3, we present the literature review of this Ph.D. research, including the executed systematic mappings.
- In Chapter 4, we present a discussion about modernization tools and a taxonomy for modernization tools.

[1] The complete list of publications can be seen in http://lattes.cnpq.br/6809743774407662

- In Chapter 5, a reference architecture for ADM-based modernization tools is presented. In this chapter we discus about the architectural views to build these kind of tools.

- Chapter 6 concludes this thesis, stating its main contributions, the publications generated during this work, and points out different opportunities for future work.

Chapter 2

Background

In this chapter we present the foundations of this PhD. research and how they are related to support the development of this thesis. In section 2.2, we discuss software modernization, presenting the main concepts related to this chapter. In the following sections, the main concepts that involve the most recent practices in systems modernization and reference architectures are shown. In Section 2.3 the Architecture-Driven Modernization is explained and in section 2.4 we discuss model refactoring. In Section 2.5 we present the main concepts of Reference Architecture. The Final Consideration are presented in Section 2.6.

2.1 Software Modernization

There is a huge variety of software system solutions spread across the most diverse sectors of industry and commerce, and they are used for the most diverse purposes. It is possible to say that every system, at some point, will need corrections, updates and improvements to continue meeting the needs of its users, a fact that opens up a wide variety of modernization tasks (Ulrich e Newcomb, 2010a). These technologies that are necessary for IT professionals to carry out these upgrades are available for use in projects and yet, new approaches that support this process are still emerging (Neubauer et al., 2017; Park et al., 2017).

Some technologies are made available through licensed or even free tools while others are controlled by service providers, which are the cases where companies provide their

own modernization service. In this case, these companies identify and drive modernization technologies and services that are critical to the success of a system modernization process (Gotti e Mbarki, 2016; Ulrich e Newcomb, 2010a).

When planning a modernization project, IT professionals can choose to use different approaches, such as hiring companies to carry out all or part of the process, using licensed or free tools and even developing their own tools. In practice, companies that need to modernize their systems do not have a modernization strategy, that is, they are often able to identify that they need to modernize, but they do not know how to conduct it. This is due to the fact that there is a lack of knowledge about the modernization process and its related techniques (Ulrich e Newcomb, 2010a).

Currently, one of the most used modernization methodologies and which has a solid base of concepts and approaches to modernization of existing systems is the *Architecture-Driven Modernization*. ADM has been available for a while and has faced many challenges as it has already been deployed and redeployed in many organizations (Ulrich e Newcomb, 2010a). These challenges are directly linked to standardization, more specifically, the lack of it. When a company implements ADM and is faced with a problem of lack of standardization, several other problems are generated and this has a direct influence on the rate of adoption of a given set of solutions that the company could use. However, as the very purpose of ADM is to create this standardization, several metamodels were and continue to be developed to solve this type of problem.

In parallel, the modernization of software could also benefit from reference architectures concepts that could provide ways of building systematized tools that deals with modernization processes.

2.2 Architecture-Driven Modernization

In 2003, the ADM *Task Force* was created and it was composed of several large companies, including large hardware vendors, system integrators, independent software vendors, and other organizations. ADM-TF built and issued a multi-stage modernization roadmap that laid out a blueprint for a series of modernization patterns (OMG, 2021).

OMG's focus was on creating a standard way of viewing unified information or metadata that a given tool would gather, manipulate and share with other tools. Within the context of modernization, this means defining a unified view of all artifacts involving existing systems, since every modernization tool gathers and stores information about existing systems in different ways (OMG, 2021).

The participation of these companies in the foundation of ADM was very important, since these companies had case knowledge, that is, they had experience of what worked and especially of what still required an effort to be solved regarding modernization of legacy systems. It is worth remembering that ADM is an approach that is constantly evolving and this is due to the fact that even today, several large companies have partnerships with OMG and offer products and services in the area of systems modernization (OMG, 2021).

ADM is a trend involving software reengineering processes and considers the use of standardized metamodels and *Model Driven Architecture* (MDA) concepts in its processes. The high number of failures in modernization projects (Berinato, 2003; Koch, 2002), was one of the main motivational factors for the creation of ADM. According to the OMG, the main reason for this problem was the lack of standardization, thus hindering the productivity of development teams, preventing the reuse of algorithms and techniques, and finally compromising the interoperability between modernization tools from different vendors.

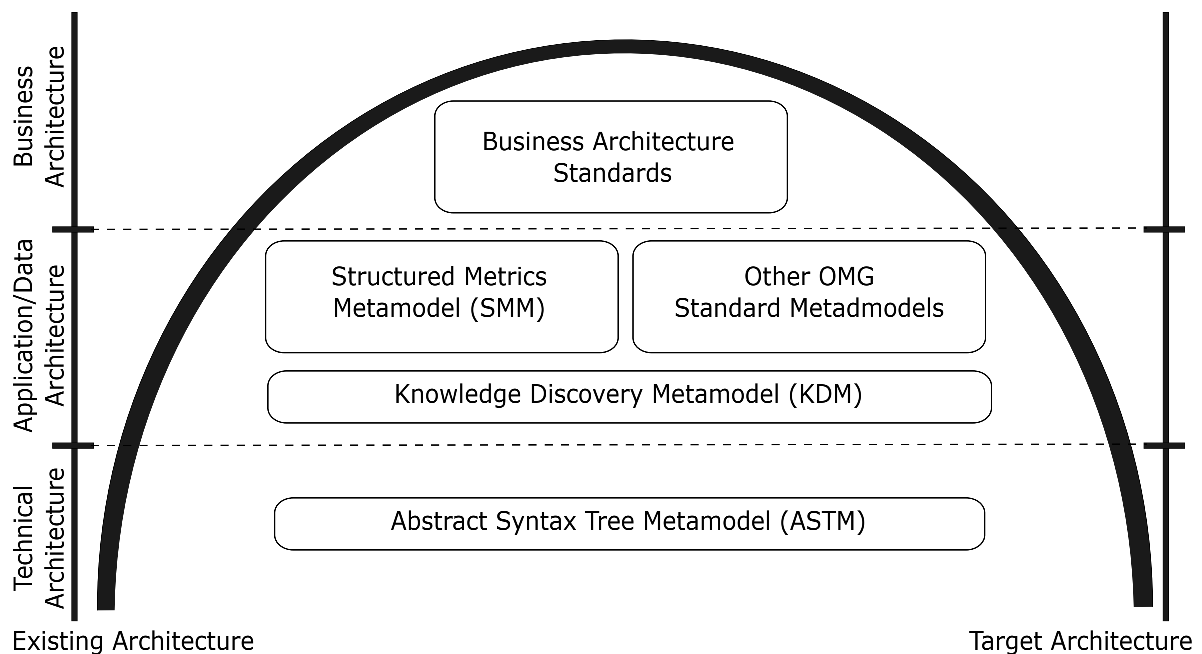

Figure 2.1: General structure of ADM (Adapted from KDM (2011)).

In Figure 2.1 we present several OMG patterns between the business and IT domains. At the technical level, refactorings are applied in order to change and update the technology that the system is built on. At the application level, the refactorings aim to restructure the application in order to improve some of the quality attributes, for example: maintenance, modularization and reuse. And at the business level, refactorings are more focused on making changes to system functionality, whether they are adding, removing or adapting

business rules. The output is a new target model without the problems that were identified earlier, which can be called a “modernized model”. In the forward engineering phase, models are once again subjected to a set of transformations with the aim of reaching the source code level again (Pérez-Castillo et al., 2011a).

Several standards support the evolution and transformation of existing business and IT architecture to target business and IT architectures. Each of these metamodels is responsible for reflecting different views of an architecture. The *Abstract Syntax Tree Metamodel* (ASTM) pattern is tied to the technical architecture of existing solutions and provides a more granular view of the architecture and supports automated transformations from existing languages and platforms to target languages and platforms. The pattern is a metamodel that aims to represent all aspects of an existing system (source code, configuration files, database and so on). KDM is an information exchange metamodel that facilitates interoperability between tools for any tool that captures or uses information about IT architecture. Even though it is possible to use information from an ASTM instance through KDM, both are metamodels that have different purposes and applicability.

OMG also provides other standards to support the system modernization process that are complementary to KDM as they serve different purposes and support modernizations regarding the architecture and data of an existing system. Such is the case of the *Structured Metrics Metamodel* (SMM), which is a metamodel for defining, representing and exchanging metric and measurement information related to any structured information model, such as the OMG standard *Meta Object Facility* (MOF) (OMG, 2021).

Business modeling patterns support the mapping between business architectures and systems architectures. Thus, it is possible to establish relationships between views from the KDM metamodel and views from the OMG *Semantics for Business Vocabulary and Rules* (SBVR) metamodel, since the KDM metamodel has metaclasses that can outline business views of a system. The SBVR defines business semantics and the rules associated with semantics and is able to represent other elements to provide a more complete view of the business model, being able to represent elements such as: business units, capabilities, processes, customers, partners and value chains, along with semantics and rules. So, even though it is possible to sketch business elements with KDM, it is only possible to get a complete picture through the use of the specific business patterns that lie in the business architecture layer.

As mentioned before, the modernization flow supported by ADM has three phases and it has the shape of a horseshoe, which are: Reverse Engineering, restructuring and forward engineering, as can be seen in Figure 2.2.

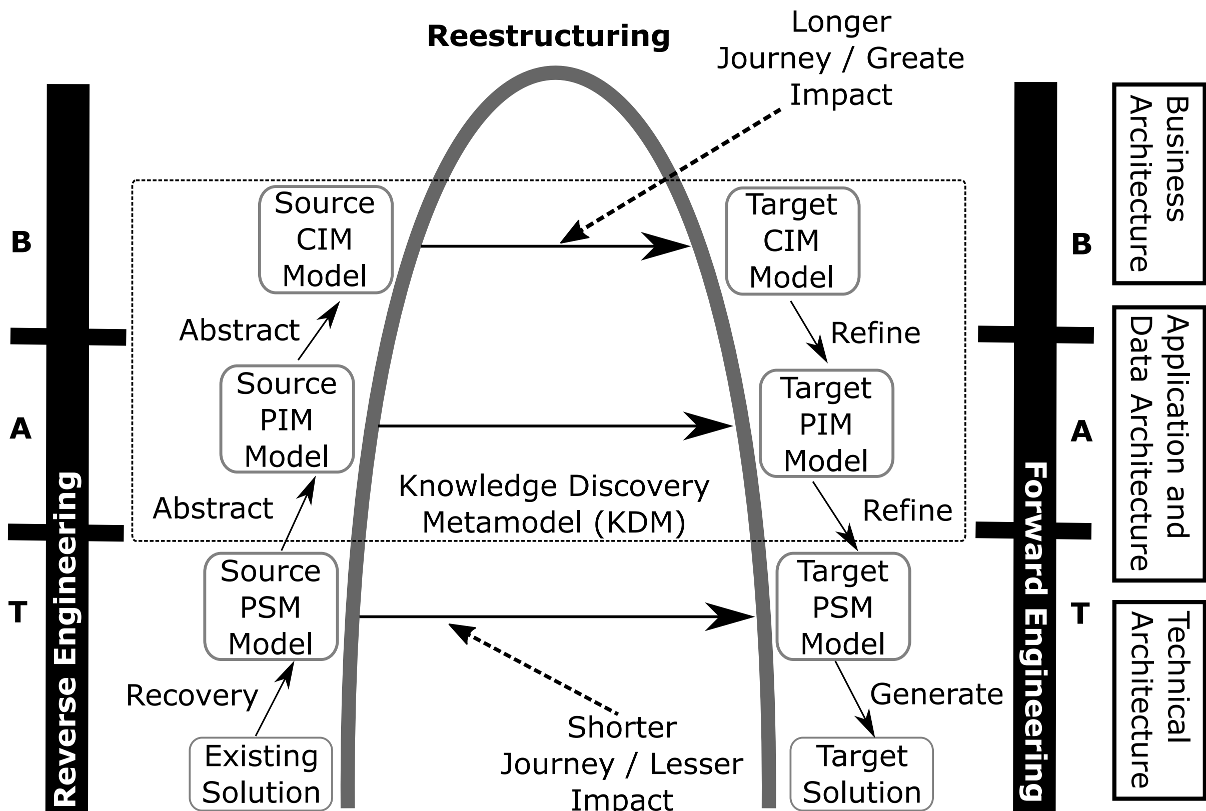

Figure 2.2: Architecture-Driven Modernization Blueprint.

Starting from the **reverse engineering** in the bottom left corner, the knowledge is extracted from an existing solution and a Platform Specific Model (PSM) is generated. In PSM there are meta-data related to a specific platform and programming language. This abstraction level is also known as technical architecture and platform migration and language-to-language conversion are examples of modernization that could be performed in this architecture level.

The source PSM can also serve as a base to the creation of a Platform Independent Model (PIM). The PIM has a higher abstraction level if compared to PSM because there are no specific platform or language information. Also known as Application and Data Architecture, in the PIM level is possible to perform modernization of the type: Service Oriented Architecture Transformation and Data Architecture Migration, for example.

The Computing Independent Model (CIM) is generate from a PIM and it is the highest abstraction level in a modernization process. In this level the business rules of a software system can be represented as a computer independent way. The CIM level is also known as Business Architecture, thus Data Warehouse Deployment and Application Portfolio Management are examples of the modernization scenarios that can be performed in this abstraction level.

In **restructuring** phase refactorings can be performed in the software system in order to get an improved version of the software system structure always preserving its original behavior. Refactorings are model transformations that can improve the design, maintenance and reuse of existing software systems (Durelli, 2016). According to Pérez-Castillo et al. (2011a) and Sadovykh et al. (2009) the restructuring phase can occur in any abstraction level (PSM, PIM and CIM) and the higher the abstraction level the greater the impact of changes in the software system.

The **forward engineering** phase is triggered when the refactorings were performed. It consists in a set of transformations to reach the source code level again.

2.3 Knowledge Discovery Metamodel

KDM is an ISO standard (Pérez-Castillo et al., 2011a) and it is the first of a series of specifications related to ADM activities. KDM assists in projects involving existing software systems facilitating interoperability and data exchange between tools produced by different vendors.

A common feature of many tools that deal with ADM challenges is that they analyze artifacts from existing systems (e.g. source code modules, database descriptions, and so on) to gain explicit knowledge of some software system (Ulrich e Newcomb, 2010a).

KDM metamodel provides a unified ontology and information exchange format that facilitates the exchange of information contained within tool models representing existing software. This metamodel is able to represent physical and logical software artifacts as well as their relationships at the most varied levels of abstraction. The primary objective of KDM is to enable a universal exchange format that allows interoperability between modernization tools, services and their respective intermediate representations. This metamodel also allows developing neutral content for different vendors (standards, rules, metrics, etc.) to be used in modernizations based on the KDM standard instead of proprietary intermediate representations of software systems (Ulrich e Newcomb, 2010a).

Using this representation, it is possible to exchange systems representation between platforms and languages in order to analyze, standardize and transform existing software systems (OMG, 2021). The KDM metamodel is formed by 12 (twelve) packages, organized in four layers: infrastructure, program elements, runtime resources and abstractions.

In Figure 2.3 we present the KDM architecture, and shows how the layers are related, which packages belong to each layer and the separation of interests in KDM. Each layer builds on the previous layer, so they are organized into packages that define a set of metamodel elements, whose purpose is to represent a specific and independent interest

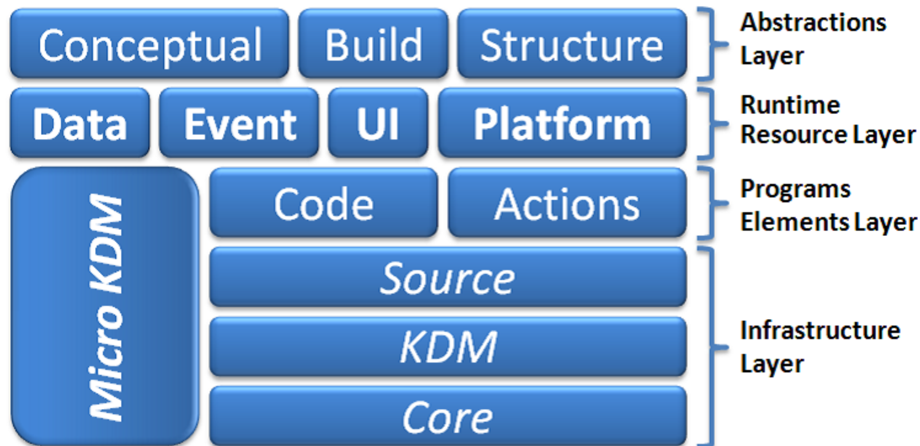

Figure 2.3: Layers, packages and separation of concerns in KDM (Adapted from KDM (2011)).

of knowledge related to legacy systems (Pérez-Castillo et al., 2011a). The infrastructure layer, which is at the lowest level of abstraction, defines a set of concepts used throughout the KDM specification, providing a common core for all other packages. The program elements layer provides a broad set of metamodel elements in order to provide a language-independent intermediate representation for various constructs defined by common programming languages. This layer represents implementation-level program elements and their associations, so the program elements layer represents the logical view of a legacy system (Pérez-Castillo et al., 2011a).

The runtime resource layer allows the representation of high value knowledge about legacy systems and their operating environment, that is, it focuses on what is not contained in the source code. The runtime resources layer represents the resources managed by the runtime platform, providing abstract resource actions to manage the other resources. Each package in this layer defines specific entities and containers to represent legacy system resources and also specific structural relationships between resources. The abstraction layer defines a set of elements capable of representing abstractions of specific domains and applications, as well as artifacts related to the process of building the existing system. Abstraction layer metamodel elements provide various containers and groups for other metamodel elements (Pérez-Castillo et al., 2011a).

Finally, the micro KDM package aims to refine the semantics defined in the KDM ontology. The micro KDM package is mainly applied to the action elements, found in the action package. Micro KDM is a set of compliance rules, additional guidelines for building and interpreting high-fidelity KDM views, suitable for performing (Pérez-Castillo et al., 2011a) static analysis.

2.3.1 KDM Compliance and Domains

KDM is a metamodel with a wide scope that covers many applications, platforms and programming languages, however, not all of its capabilities are equally applicable for all platforms, applications or program languages. The main objective of KDM is to offer the ability to exchange information between models of different tools and thus facilitate cooperation between tool providers so that it is possible to reuse different solutions in different projects. To achieve interoperability and integration of information from different tools is that the KDM metamodel specification defines several levels of compliance, in this way, it increases the probability that two or more compatible tools will be able to work together. KDM follows the principles of separation of concerns to allow the selection of only those parts of the metamodel that are of interest for the development of a specific tool. This separation of concerns can also be called KDM domains (KDM, 2011; Pérez-Castillo et al., 2011a).

A system is composed of several interests and each one of them has relevant information about a certain domain, each domain can be represented by a different KDM domain, as can be seen in Figure 2.4. Each KDM domain defines an architectural point of view. Each point of view of a domain is defined by its own KDM package, which has specific metaclasses to represent the system elements corresponding to a given point of view. The metaclasses defined by all KDM packages constitute an ontology for describing existing systems. For example, the *Code* and *Action* packages define the point of view for the source code domain that represents source code elements of a system, such as variables, procedures, and declarations. The *Structure* package defines the viewpoint for the structure domain that represents architectural elements of the system, such as subsystems and components. The package *Conceptual* corresponds to the business rules domain and defines the point of view to represent behavioral elements of a system, such as characteristics and business rules (KDM, 2011; Pérez-Castillo et al., 2011a).

Since the same system can have different points of view, it may be necessary to maintain a traceability between them, so that a better understanding and control of the representation in KDM can be obtained. In this way, KDM formally allows traceability between domains, that is, it is possible to represent the existing links between different domains natively, without the need to use other methods/metamodels outside KDM (KDM, 2011).

As you can see in Figure 2.4, there are 8 domains in the KDM metamodel, which are: *Code*, *Build*, *Structure*, *Data*, *Business Rules*, *UI*, *Event*, *Platform* and *micro KDM*. Please notice that the *Code* domain contemplates all L0 KDM packages: Core, kdm,

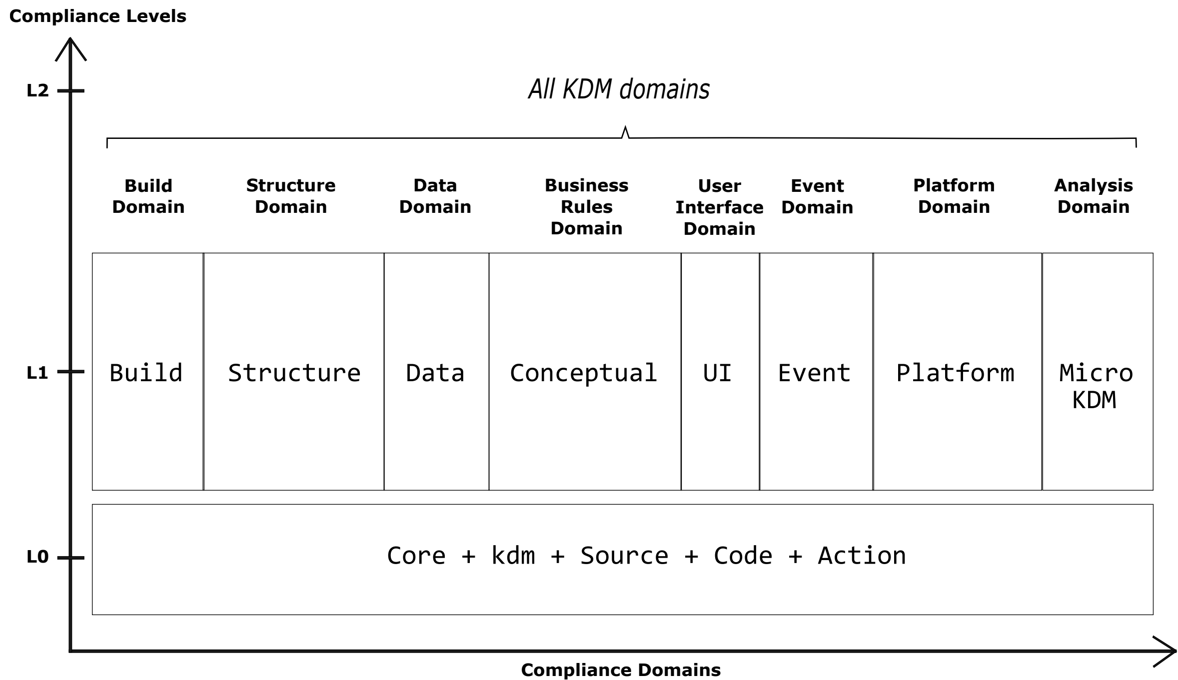

Figure 2.4: KDM domains and compliance levels KDM (2011).

source, code and action. These domains represent the basis for building KDM compliance levels. On the other hand, these levels of compliance mean that you only need to worry about the parts of KDM that are necessary to carry out your activities. Each domain is self-sufficient to represent a certain point of view of the system, if another domain is needed to represent a different point of view, there will be no major impacts on the representation that already exists. Consequently, if the intention is to develop a tool that only involves the *Structure* domain, for example, it is not necessary to have full knowledge of the KDM metamodel to use it efficiently. However, most of the KDM domains were partitioned into multiple increments, that is, each one is responsible for adding more knowledge to the previous domain, in such a way that they are complementary to each other (KDM, 2011; Pérez-Castillo et al., 2011a).

This division into domains aims to make it easier to learn and to use the KDM metamodel. However, these domains do not necessarily mean that each one represents a level of compliance, as this would generate an excessive number of levels of compliance and it would complicate the interoperability between the tools that would implement these domains. Still in Figure 2.4, the three levels provided for in the formal specification of KDM (KDM, 2011) are represented.

Compliance level L0 has the following KDM packages: *Core*, *kdm*, *Source*, *Code* and *Action*. This level provides an entry level of knowledge discovery capability, that is, it

represents a common denominator that can serve as a basis for interoperability between different categories of tools that use KDM. For a tool to be considered at compliance level L0, it is important that it fully support all metaclasses of packages mentioned at this level (Pérez-Castillo et al., 2011a).

The L1 compliance level supports all L0 packages and adds the following Build, Structure, Data, Conceptual, UI, Event, Platform, and MicroKDM packages. This level represents the layers in which modernization tools can be complementary as long as their focus is on different interests. For example, it is possible to have a modernization tool that supports the Level 0 packages and also the Structure package and another tool that also supports the L0 packages and the Conceptual package. As these tools support different KDM interests and are based on L0, we claim that they are complementary and interoperable. For a tool to be considered Level 1 conformance for a given domain (KDM metamodel package), the tool must fully support all metaclasses defined by the package for that domain and satisfy all semantics and constraints specified by the domain (Pérez-Castillo et al., 2011a).

The L2 compliance level is basically the union of levels 1 for all KDM domains, that is, for a modernization tool to be at the L2 level of compliance, it must fully support all packages and metaclasses of the KDM metamodel (Pérez-Castillo et al., 2011a).

2.4 Reference Architectures

A Reference Architecture is a structure that provides a software system functionalities characterization from the perspective of technology, application or problem of a specific domain. A RA can be used in three different contexts: (I) to help in the development of new systems‘s concrete architecture, (II) to help in evolution/modernization of software systems that are from the same domain, or (III) to help in the standardization and interoperability of software systems (Galster e Avgeriou, 2011).

Thereby a RA can be used as a guide to improve the chances of successfully develop a software system, it also can be considered as the first essential step to the development of application frameworks. Application frameworks are standard structures which aim to support the software systems development (Nakagawa et al., 2014a).

The proposal of a RA to a specific domain is not a trivial task because it requires a deep knowledge about this specific domain. Thus, the creation process of a RA involves several steps and it demands the analysis of several artifacts like software systems, concrete architectures, scientific papers, technical reports and other documents that have embed knowledge about the specific domain of the RA that will be built. The reuse of these

artifacts when used together with the domain experts knowledge are one of the success keys to the development of a RA (Galster e Avgeriou, 2011; Nakagawa et al., 2014a).

2.4.1 Reference Architecture Types

The creation of a reference architecture, can be started after the identification of an architecture problem. However, is not always possible to have or use systems that help in this analysis. Thus, in this way, we can drive the architecture creation in two main types: Futuristic Reference Architecture (FRA) and Practical Reference Architecture (PRA) (Angelov et al., 2008b).

Futuristic Reference Architectures are defined with the help of existing research that does not have practical experiences of concrete architectures, that is, it is when systems are not available to extract knowledge from a specific domain. Thus, We can imply that these reference architectures are oriented to research and it are mainly based on existing reference models and architectural patterns. The objective of this type of reference architecture is an attempt to "look to the future" in a way that it is possible to foresee some important principles that may be part of the design of a concrete architectures for a domain (Angelov et al., 2008b). Examples of FRA can be seen in Angelov e Grefen (2008), Norta (2007) and Wu (2002).

Practical Reference Architectures are guided by practical experiences in which concrete architectures represent the main sources of information. PRAs, in general, consider the legacy systems in their project and/or creation. Usually it is developed by organizations responsible for standardization to facilitate development within a specific domain or by groups of large companies operating in the same domain to establish or enforce standards (Angelov et al., 2008b). Examples of PRA can be seen in Zimmermann (1980), Hollingsworth et al. (2004) and Grefen e de Vries (1998).

Besides this division proposed by Angelov et al. (2008b), it is more common to find reference architectures that consider both types. As can be seen in the work of Nakagawa et al. (2014b), any form of knowledge extraction can be used for the elaboration of a reference architecture. Even in cases where there is a lack of legacy systems in the domain, it is possible to investigate systems from nearby domains in order to extract patterns that can assist in the creation process. In Figure 2.5 the union between the main methods of information extraction used in both types are represented.

In Figure 2.5 we can see that concrete architectures are part of the main elements of the knowledge present in a reference architecture, as well as reference models and architectural patterns, thus reinforcing what is proposed by Nakagawa et al. (2014b).

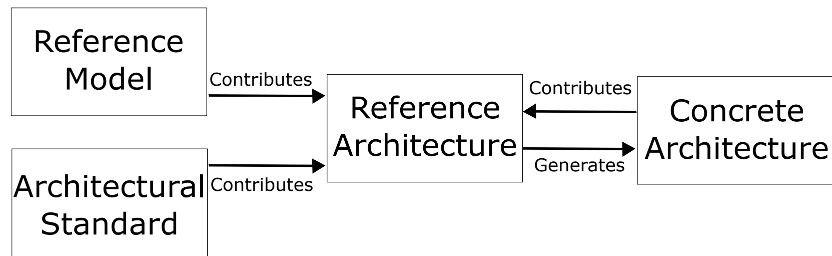

Figure 2.5: Reference Architecture Composition (Adapted from Angelov et al. (2008b)).

2.4.2 Reference Architecture Attributes

Nowadays, there is a wide variety of systems available on the market and in the scientific community. These systems have wide spread business rules and functionalities, being used in different areas. To meet the requirements for these systems, it is necessary to choose the quality attributes that best satisfy its demand. Quality attributes can be considered like a system requirements and when a system meets these attributes they can be considered as high quality systems (ISO/IEC25010, 2011; Nakagawa, 2006).

In the context of Reference Architectures (RA), quality attributes are fundamental because their definition can be considered as decisive factor for their creation and/or adoption. RA that have a certain set of quality attributes, in short, transmit these characteristics to the systems that are developed based on their guidelines.

A quality model is formed by a set of quality attributes, this set is the basis for assessing the quality of a system. The quality attributes chosen for the model is responsible for determining which attributes will be taken into account when assessing a determined system. Thus, the quality of a system corresponds to the degree to which a system meets the needs established by its target audience/model (ISO/IEC25010, 2011).

ISO/IEC 25010 is an ISO standard available since 2011 for software product quality. This standard defines models for assessing the quality of software and systems. The use of ISO/IEC 25010 has a fundamental importance in the elaboration of a Reference Architecture, once it provides a consistent terminology for the specification, measurement and quality evaluation of systems. Some of the main benefits when we use/apply a quality model in our software can include (ISO/IEC25010, 2011):

- easiness during the requirements discovery;
- validation of the scope of a requirement;
- identification of the design objectives;

- identification of the test objectives;
- identification of quality control criteria as part of quality assurance;
- identification of the acceptance criteria;
- definition of measures for quality attributes.

ISO/IEC 25010 emerged as a proposal to replace the ISO/IEC 9126 standard. The 25010 has as main novelties the addition of two new features, ”security” and ”compatibility” (ISO/IEC25010, 2011). In Figure 2.6 are presented all 8 quality attributes and their sub-attributes related to the quality of the software product based on ISO/IEC 25010.

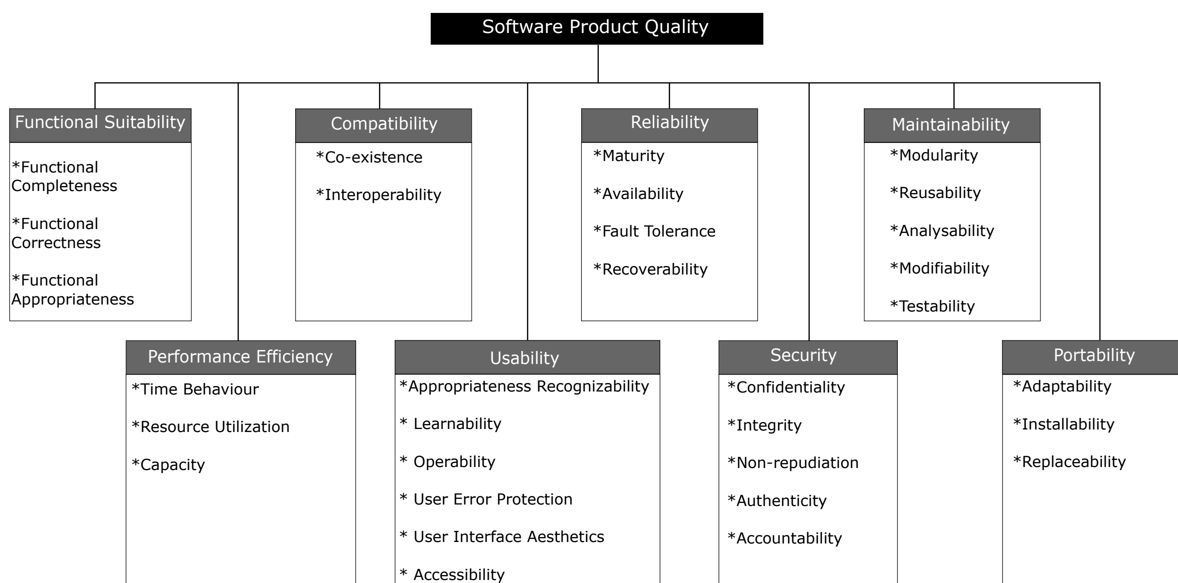

Figure 2.6: Quality attributes for a Software product (Adapted from ISO/IEC25010 (2011).

Each quality attribute has a set of other attributes inside it, here we will call them as sub-attributes, as we can see in Figure 2.6. When all sub-attributes within a specific attribute are met, it is possible to state that the system has this quality attribute. For example, if a system is able to coexist (Coexistence sub-attribute) and is interoperable (Interoperability sub-attribute) with other systems, it can be said that it is a system compatible with others (Compatibility quality attribute).

There are cases in which not all sub-attributes are met. For these cases, it can be said that the attribute has been partially met, however the sub-attributes it meets is part of its knowledge. For instance, it is possible that a system is partially portable (Portability quality attribute) and meets only the Adaptability sub-attribute. This means

that only the Adaptability was evaluated in the context of this system. In the specification of ISO/IEC25010 (2011) the term "degree" is used to assist in the description of all 8 attributes, as can be seen in Table 2.1.

Table 2.1: Quality attributes of a Software (Adapted from ISO/IEC25010 (2011))

Attribute	Description
Functional Capacity	Degree of attendance with the needs of the product/system when used under certain conditions.
Performance Efficiency	Degree of performance relative to the amount of resources that are used under established conditions.
Compatibility	Degree of exchanging of information between products, systems or components, and/or perform its functions as required, while sharing the same hardware or software environment.
Usability	Degree of a product/system to be used by specific users to achieve specific objectives with effectiveness, efficiency and satisfaction during a specific using context.
Reliability	Degree of a product/system to performs specific functions under specific conditions, for a specific period of time.
Assuredness	Degree of a product/system to protects information and data in a way that the communication with other products/systems have the same degree of data access appropriate to their types of authorization levels.
Maintainability	Degree of effectiveness and efficiency with which a product/system can be modified to be improved, fixed or adapted to changes in environment and in the requirements.
Portability	Degree of effectiveness and efficiency with which a product/system can be transferred from one hardware, software or other operational environment to another one.

The term degree present in Table 2.1 can be understood as a measure that represents a specific level of satisfaction that a given system has in relation to a specific quality attribute. It is important to understand the purpose of each quality attribute, as well as its sub-attributes, for the correct adoption in a given architecture.

2.4.3 Reference Architecture Evaluation

In the process of creating a RA all stages are essential, however, the evaluation stage requires a greater effort. In this stage that it is possible to identify the strengths and, especially, the weaknesses of the architecture. The consistent evaluation of an RA increases the chances of its use by researchers and system developers. In other words, a well-planned assessment of a reference architecture is an incentive for its wide adoption.

In the context of evaluating a reference architectures, several authors have used assessment approaches for common software architectures, making adaptations so that they can be applied in a RA. Babar e Gorton (2004) compare four evaluation methods for

scenario-based architectures using an assessment framework. This framework considers each method from four points of view, that are context, domain experts, structure and reliability.

Angelov e Grefen (2008) adapted an existing method for evaluating concrete architectures and quality attributes (such as applicability, usability, feasibility and automation). They use classic techniques of logic, such as comparisons between functionalities, elaboration of scenarios based on experts feedback and the use of methods for evaluating a set of qualities (*Architecture Trade-off Analysis Method* - ATAM).

Evaluating a reference architecture has not been a trivial step when considering methods that are not specific for this purpose. Using classic techniques employed in the evaluation of software architectures in reference architectures requires adaptations as well as a greater effort.

The methodology proposed by Santos et al. (2013) is called Framework for Evaluation of Reference Architecture (FERA) and is dedicated exclusively to evaluate reference architectures, regardless of the domain. This methodology can be used either alone or in conjunction with other evaluation methodologies. Since the checklist approach can be considered simple, cheap, flexible and it is still capable of carrying out a high-level assessment. It also can save a good amount of resources and time by discovering defects and clearing doubts in a early stage of the reference architecture life cycle. In order to have its effect maximized, it is recommended applying it during the preparation of RA, as at this stage the defects can be identified and corrected before obtaining a final version. An example can be seen in Figure 2.7.

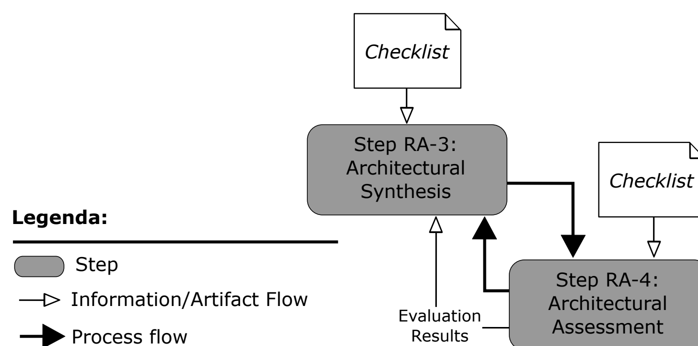

Figure 2.7: Steps to create an RA in a way that the checklist can be applied (Adapted from Santos et al. (2013)).

According to Santos et al. (2013), when using the FERA checklist it is possible to obtain a set of information about the reference architecture being developed, like:

- the RA is adequately represented, that is, the RA provides general information about potential risks, limitations and scope;
- the RA has an appropriate set of architectural views to its representation;
- the documentation of the RA has important information, such as architectural decisions, best practices/guidelines, policies/rules, international standards and interfaces between modules;
- the RA considers the quality attributes related and important to its domain;
- it is easy to create/instantiate a system based on the RA; and
- what could be changed on any topic like documentation to get a better RA information.

The results of this evaluation can be used to improve the architectural documentation of reference architectures. The checklist proposed by FERA consists of 93 multiple-choice questions whose answers can range from ”completely satisfactory” to ”totally unsatisfactory”. Each question has a field for adding comments from those who are evaluating the RA. This questionnaire must be answered by a set of people who represent the interested parties, which can be architects, domain experts, analysts, software project manager, designers/developers, testers and other parties interested in guarantee the quality of the RA (Nakagawa et al., 2014b).

The FERA checklist is structured in four stages, which are: (1) General Information; (2) Raising Discussions; (3) Conclusion of the General Analysis; and (4) Domain Specific Questions. The stages 1 and 4 are divided into two more groups, where each group has specific questions that are targeted to a specific audience. The complete layout of the stages, groups, questions and subjects covered can be seen in Table 2.2 (Santos et al., 2013).

Based on Table 2.2, it is possible to extract information about what information each question group is responsible for evaluating. For example, in the first group of the first stage there are nine questions that evaluate the overall information, the points of view, the views, the models, the stakeholders and the concerns. In the second group of the first stage, there are 16 questions regarding legal regulation, questions related to ISO 42010[1],

[1]ISO/IEC/IEEE 42010:2011 addresses the creation, analysis and support of system architectures through the use of architectural descriptions. This standard defines a conceptual model for the architecture description, specifies the content required for the description, and also specifies the content necessary for architectural views, architectural *frameworks* and architectural description languages.

Table 2.2: Stages, Number of questions and Evaluation Topics of the FERA checklist (Adapted from Santos et al. (2013)).

Stage	Group	Questions	Group Evaluation Topics
1	1	9	Overview information, points of view, views, models, stakeholders and interests.
	2	16	Legal regulation, compliance with ISO 42010, design/development issues, domain specific issues, artifact compliance and quality attributes.
2	1	52	Specific questions for stakeholders.
3	1	3	Overall analysis conclusion.
4	1	11	Specific questions related to hardware.
	2	3	Specific questions related to software

design/development problems, domain specific issues, artifacts compliance and quality attributes.

2.5 Model Refactoring

The main mechanism used to materialize the modernization process advocated by ADM are the transformations from legacy software metamodel instances into modernized ones. These transformations can also be understood as Refactorings which are transformations used to improve the structure of a system, in such a way that its behavior is preserved even after successive (Van Der Straeten et al., 2007) transformations. Refactorings allow you to restructure, realign, modularize, that is, redo the source code and the way an existing tool is used, thus, refactorings can be understood as a process that aims to redistribute functionality to improve an existing system.

According to Opdyke (1992), refactorings can be understood as a set of transformations that, when applied to an existing system, enable its restructuring in order to improve its design, evolution and reuse. Refactorings are important allies in the process of restructuring a system, however, it is important to keep in mind that the changes made do not impact the behavior of the system's functionalities. Chikofsky e Cross (1990) state that the restructuring process consists of altering a system in such a way that its internal structure is improved, however, the external behavior of the source code should not be modified.

The use of refactorings can bring a number of benefits to a system. These benefits will depend on the type of change you want to make, for example, in the maintenance stage of a system it is possible to apply refactorings that help the source code to become more readable or make the interests of the system to be modularized (Pérez-Castillo et al., 2011a). Several authors have proposed sets of refactorings that can range from domain-specific to more generic refactorings. One of the best known refactoring catalogs

is the one proposed by Fowler e Beck (1999), it proposes source code refactorings for the object-oriented paradigm.

Refactorings provide assistance in the modernization of systems, since through them it is possible to perform business rules extraction, cross-platform migrations and other modernization scenarios (Pérez-Castillo et al., 2011a). In the context of ADM, refactorings are performed at the model level, as ADM uses the MDA concepts (OMG, 2021). Model refactorings have the same goals as a source code refactoring, the main difference being the level of abstraction.

Model refactorings can be understood as model transformations and its main objective is to improve the model's structure, as well as to preserve its internal characteristics, that is, it does not differ from other types of refactorings (Einarsson e Neukirchen, 2012; Van Der Straeten et al., 2007). However, refactorings applied to models have a greater challenge, since there is greater diversity in relation to the level of abstraction that refactorings can be applied. Thus, if a refactoring for models was developed to act at one level of abstraction, it cannot be applied at another level without undergoing an adaptation. Another problem is that when refactorings are developed to be applied to a model coming from a proprietary metamodel, the chances of reusing this refactoring are drastically reduced. This fact reinforces the importance of adopting standard metamodels, since the refactoring developer itself can benefit from the reuse of existing refactorings.

Authors such as Mens (2008), Mohamed et al. (2010) and Mens et al. (2007) state that model transformation plays a fundamental role in approaches that use MDA principles, as it allows model manipulation in an automatic way. Transformations consist of the automatic generation of a target model, based on a source model, in such a way that this transformation is defined by means of a set of transformation rules (Mens e Van Gorp, 2007). According to Mens e Van Gorp (2007), there are three possible classifications for model transformations, which are: Vertical or Horizontal, Endogenous or Exogenous and Bidirectional.

In Table 2.3, transformations between source and target models can happen at one or more abstraction levels and there are cases in which more than one classification can be used at the same time, that is, they can be complementary. For example, when there is a refactoring capable of transforming a PIM model represented in KDM into a CIM target model also in KDM and this refactoring is capable of doing the opposite process, that is, transforming the CIM model into PIM, both in KDM, we can affirm that this refactoring is vertical, endogenous and bidirectional.

Table 2.3: Classification of model transformations (Adopted from Durelli (2016))

Classification	Name	Description
1	Vertical	In the vertical transformation, there is a change in the level of abstraction in the models and this change can be either to increase or to decrease the level of abstraction.
	Horizontal	A horizontal transformation keeps source and target models at the same level of abstraction.
2	Endogenous	In endogenous transformations, the models involved are expressed in the same modeling language .
	Exogenous	In exogenous transformations, the models that participate in the transformation are from different languages.
3	Bidirectional	A bidirectional transformation can either generate target models using source models as a basis, or generate source models using target models.

According to Mens e Tourwe (2004), performing the refactoring process for models is quite complex and involves a series of activities that must be performed so that a refactoring can be considered to serve its purposes.

The first activity is to select an appropriate model for applying refactoring (Selecting a Model). The second activity is to select an appropriate model transformation system to specify the model and transformation rules for the refactoring (Model Transformation System). The third activity is to identify excerpts within the model that need to be refactored (Model Smells[2]). The fourth activity is to select the appropriate refactorings that can be applied to the stretches that were identified in the previous activity (Selection of Model Refactorings). The fifth activity is to verify if the model's behavior will be preserved right after the refactorings are applied (Knowledge Preservation). The sixth activity is to apply refactorings (Application of Refactorings) and the seventh is to assess the effects of refactorings on software quality (Assessment of Refactoring Effects). The eighth and final activity is to check whether the consistency between the refactored model, other software models and source code has been preserved (Model Synchronization) Mens e Tourwe (2004).

2.6 Final Considerations

A Reference Architecture is a structure that provides a software system functionalities characterization from the perspective of technology, application or problem of a specific domain. A RA can be used in three different contexts: (I) to help in the development of new systems‘s concrete architecture, (II) to help in evolution/modernization of software systems

[2]Model Smells represent indications that there may be problems that need to be addressed in a model and that it is necessary to review it (Fowler e Beck, 1999).

that are from the same domain, or (III) to help in the standardization and interoperability of software systems (Galster e Avgeriou, 2011).

Thereby a RA can be used as a guide to improve the chances of successfully develop a software system, it also can be considered as the first essential step to the development of application frameworks. Application frameworks are standard structures which aim to support the software systems development (Nakagawa et al., 2014a).

The proposal of a RA to a specific domain is not a trivial task because it requires a deep knowledge about this specific domain. Thus, the creation process of a RA involves several steps and it demands the analysis of several artifacts like software systems, concrete architectures, scientific papers, technical reports and other documents that have embed knowledge about the specific domain of the RA that will be built. The reuse of these artifacts when used together with the domain experts knowledge are one of the success keys to the development of a RA (Galster e Avgeriou, 2011; Nakagawa et al., 2014a).

Chapter

3

Related Works and Systematic Mappings

This section presents the works we consider the most related to our research and those that have colaborated in some degree to the RA we have developed. Up to this moment there are no reference architectures specifically devoted to assist in building neither general modernization tools nor ADM-based modernization tools. So we present some initiatives that have worked around modernization tools since these papers usually present contributions around the architecture of these tools.

3.1 Modernization tools that use ADM standards

The first related work is the tool presented by Durelli (2016) which is an approach to perform model refactorings based on the catalog of refactorings from Fowler e Beck (1999). The Knowledge Discovery Model-Refactoring Environment (KDM-RE) tool supports modernization processes and it uses internally the Unified Modeling Language (UML), the KDM and the Structured Refactoring Metamodel (SRM) metamodels.

Durelli (2016) approach's has the following steps:

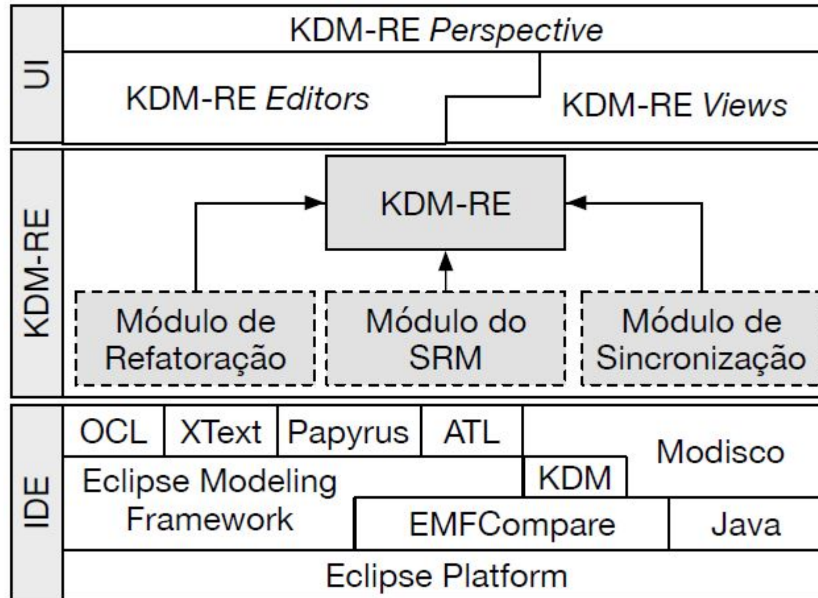

Figure 3.1: KDM-RE's architecture (Durelli, 2016).

- initially, the software engineer must convert the system to be modernized to an instance of KDM metamodel;
- the KDM instance should be converted to an instance of the UML metamodel;
- the UML instance can then be visualized through the UML class diagram, which is used as a graphical interface during the modernization process;
- the software engineer can then interact with the class diagram and choose a set of refactorings to be applied;
- after choosing a refactoring, the engineer must provide information for the correct execution of the refactoring. This information is sent to a transformation language that is responsible for performing the refactoring in KDM instance.

The tool's architecture can be seen in Figure 3.1. The first layer of the KDM-RE tool groups Eclipse IDE resources and It counts with the help of other plug-ins available in the literature for the creation of the three main modules of the tool. The KDM-RE layer has three main modules, Refactoring, SRM and Synchronization. The Refactoring Module is responsible for applying refactorings transparently in to KDM instances, the SRM Module is responsible for instantiating and to reusing the SRM metamodel, and the Synchronization Module is responsible for keeping consistent and propagating changes after applying refactorings to instances of the KDM metamodel. The last layer is called UI and is responsible for the tool's graphical interface.

Based on a systematic mapping conducted by our research group (Durelli et al., 2014), we have identified two main works that present contributions related to tool support. The first is the work of Pérez-Castillo et al. (2011b); Pérez-Castillo et al. (2011a) and the second is the related work is from Frey e Hasselbring (2011); Frey et al. (2013). Both of these works are described next.

Pérez Castillo et al., (2011a, 2011b) present a tool-supported approach for recovering business processes from existing systems. As these authors employ KDM in their tool, the proposed approach is interesting from the point of view of this work. However, notice that the approach is concentrated only in the reverse engineering phase, not covering all phases of a complete modernization process.

The tool-supported approach proposed involves a set of three-phase transformations:

- The first set involve transformations that take into account artifacts of the legacy system. In this phase, it was used a specific metamodel for each artifact. Traditional reverse engineering techniques were used here, such as static analysis, dynamic analysis and formal concept analysis.
- The second set of transformations were developed to get a KDM instance from the model instances generated in the first step.
- The third, and last, set of transformations is responsible for obtaining the model representing the business processes. These transformations were based on a set of business patterns.

In Figure 3.2 we can see the complete business discovery process. Level 0 (L0) represents real-world legacy system artifacts, level 1 (L1) represents instances of PSM models, level 2 (L2) represents PIM models, and level 3 (L3) the CIM.

To transition from level 0 to 1, the approach provides a semi-automatic technique based on dynamic analysis combined with static analysis to analyze source code and generate an event log model. To transition from level 1 to 2, a set of model transformations is performed to transform the event log model into an instance of the KDM metamodel to describe the legacy system information, taking into account the runtime view, which can be used in most modernization projects. Finally, level 3 (L3) represents the end of the discovery process, that is, obtaining the system's business process model that was retrieved from the legacy source code. This model represents a CIM and conforms to the specifications of the BPMN metamodel.

Another work related to ours was proposed by Frey e Hasselbring (2011) and Frey et al. (2013). The proposal is a tool-supported approach called CloudMIG whose main objective

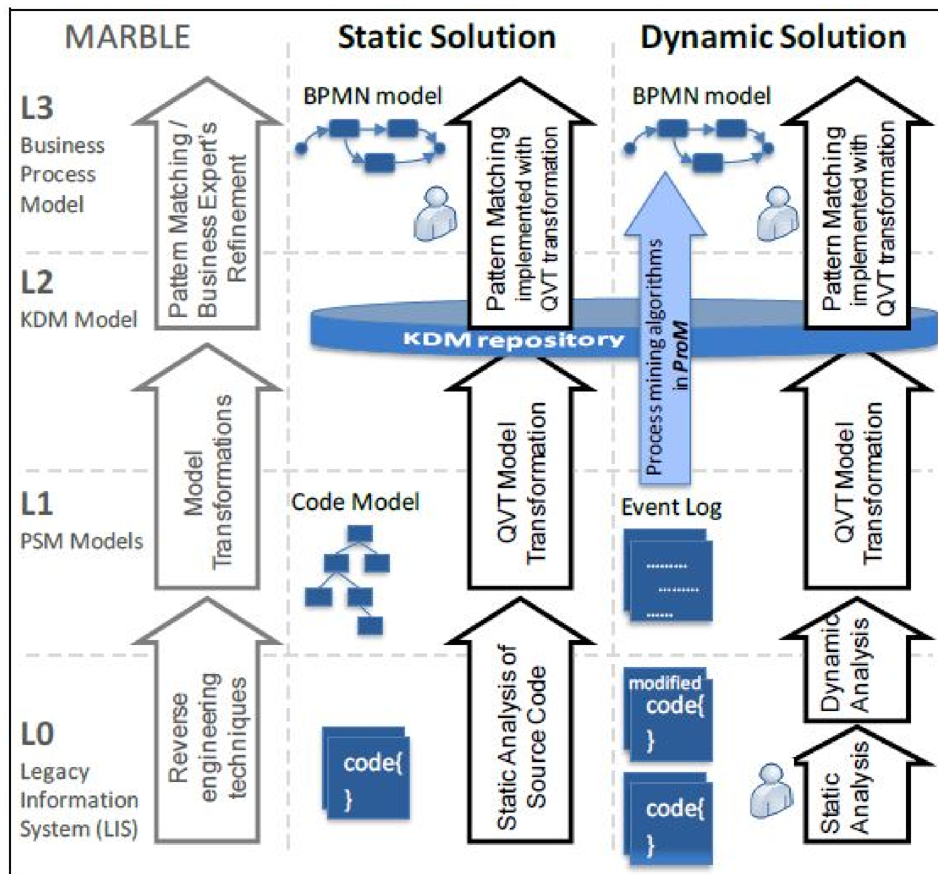

Figure 3.2: Approach to business process recovery - MARBLE (Pérez-Castillo et al., 2011a).

is to support the migration of legacy systems to Software as a Service (SaaS) in a semi-automatic way. This approach comprises six main steps (Figure 3.3), which are:

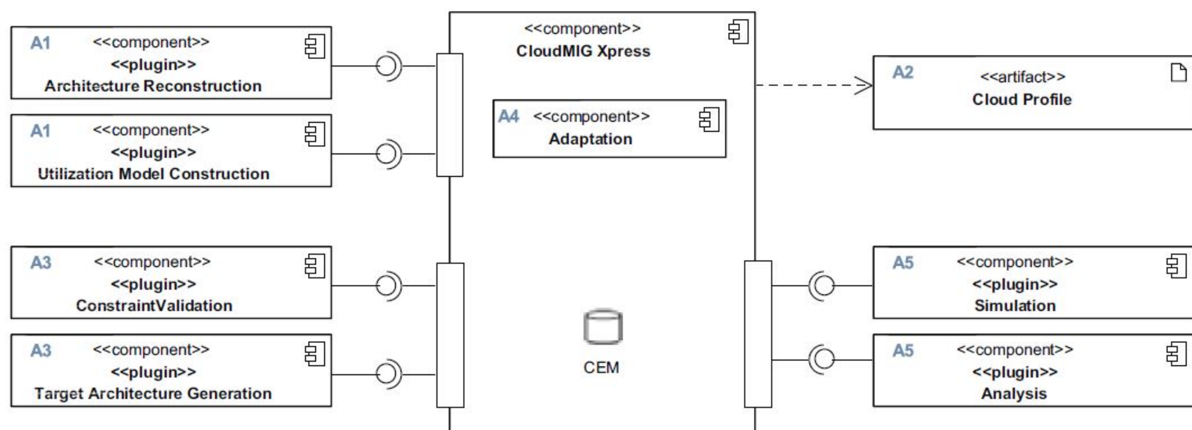

Figure 3.3: Approach to transforming legacy systems to the cloud - CloudMig (Frey e Hasselbring, 2011).

1. Extraction (A1): The extraction of the KDM architecture and models from the legacy system is included;

2. Selection (A2): An appropriate cloud environment model (*Cloud Environment Model* - CEM) is selected according to the characteristics of the system;

3. Generation (A3): A target architecture and a mapping model are generated;

4. Adaptation (A4): The adaptation activity allows an engineer to manually adjust the target architecture, performing changes where It was necessary;

5. Evaluation (A5): Static analyzes and runtime simulations of the target architecture are carried out;

6. Transformation (A6): The actual transformation of an existing system is carried out based on the target architecture that was generated to act in the cloud environment (A6 is not represented in the Figure, as it represents the system itself).

3.2 Tools that apply refactorings in UML

The second group of research are those that propose tools that apply refactorings in UML instances. The architectures of these tools contributed to the body of knowledge of our RA.

The work of Misbhauddin e Alshayeb (2015) was identified, in which a systematic review was performed in order to identify and to classify papers that propose refactorings for UML metamodel. The primary studies selected were grouped into six categories, namely: (1)Model Transformation Systems, (2)Model Smell Detection Strategies, (3)Support for UML Diagrams, (4)Model Behavior, (5)Model Quality and (6)Tooling Support.

The working group that will contribute to the development of the RA are the ones that provide Tooling Support for refactorings in the UML metamodel. The work of Misbhauddin e Alshayeb (2015) selected a total of 63 primary studies, in which 39 have some type of tooling support. Among the information that was extracted from this systematic review are the types of tools, the modeling environment and the level of automation.

Among these works, Einarsson e Neukirchen (2012) presents an approach to perform synchronous refactorings between an UML diagram and an UML model through transformations between models (*Model-to-Model Transformations* - M2M). According to Einarsson e Neukirchen (2012), a model is completely independent of its visualization, since in the model there is no information about the *layout* of the diagram. The proposed

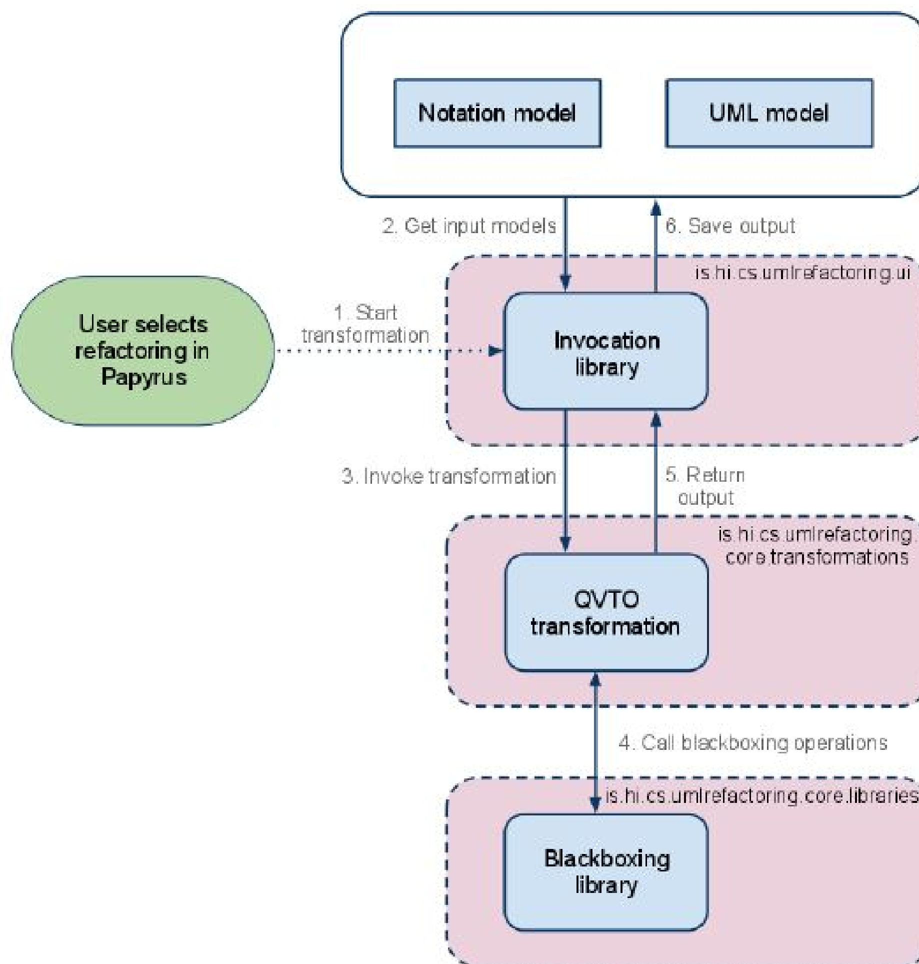

Figure 3.4: Workflow of refactorings proposed by Einarsson e Neukirchen (2012).

approach uses an existing API that allows the representation of UML diagrams for Eclipse called Papyrus. Papyrus is able to interpret an UML model (*.uml) and represent it as an UML diagram (*.notation). The complete approach has 6 (six) steps, which are:

1. *Start transformation*: The user selects the refactorings to be applied with the help of Papyrus and activates the *Invocation library* mechanism.

2. *Get input models*: At this moment, the invocation library (*Invocation library*) retrieves the *.uml file that contains the UML model and the *.notation file that contains the UML diagram and sends them to the next step;

3. *Invoke transformation*: These models are sent to the transformations module *QVT Operational Mappings* (QVTO) which selects the necessary transformation rules to perform the chosen refactorings;

4. *Call blackboxing operations*: This step is optional and can be invoked by the QVTO transform module, which can choose whether or not to call blackbox functions[1] (*black-box*);

5. *Return output*: In this step, the QVTO transformations were applied to the models and returned to the *Invocation library* module;

6. *Save output*: Finally, after executing the transformations, the invocation library saves the refactored models, replacing the original models, so that Papyrus can redo the reading of the models.

In the work of Van Der Straeten et al. (2007) mathematical formalisms are proposed for the preservation of behavior between sequence and state machine diagrams and their refactored versions. The approach was implemented through a plug-in called RACOoN and was developed to be used in an environment of tools that apply refactorings to UML models. In Figure 3.5 it is possible to see the architecture of the RACOoN tool and the other tools that make up the refactoring environment.

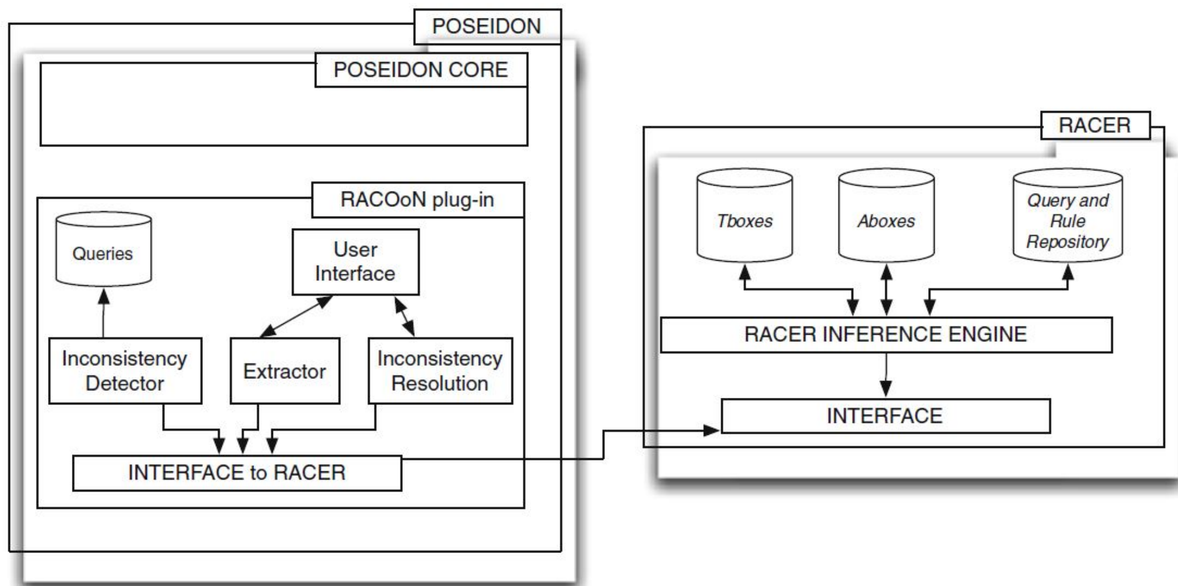

Figure 3.5: RACOoN plug-in's architecture (Van Der Straeten et al., 2007).

The Poseidon tool has a set of *queries* that look for and identify inconsistencies between UML models. The RACOoN tool was developed to serve as a plug-in to the Poseidon tool and It assists in the behavior preservation process. The RACER Tool is used in

[1]A blackbox function allows complex algorithms to be encoded in any programming language, since some algorithms are difficult to be implemented by constraint specification languages such as *Object Constraint Language* (OCL).

the approach to formally specify and to detect inconsistencies in UML models, with the support of a concepts collection and logic description functions (*Description Logics* - DLs[2]), which is a formal approach to perform consistency checks of behavior inheritance.

The RACOoN plugin has five main components to help preserve the behavior of refactored UML diagrams, which are:

- *User Interface*: Allows you to choose and perform detection of specific inconsistencies in UML models. It also allows the user to configure the tool and to load logic descriptions from UML metamodel in to RACER tool;
- *Extractor*: It has two functionalities, translate user-defined templates into *Abox* statements and load these statements into the RACER tool;
- *Inconsistency Detector*: It can be used for both detecting inheritance inconsistencies and detecting violations of preservation properties;
- *Inconsistency Resolution*: These are algorithms that aim to correct the inconsistencies that were identified in the verification process;
- *Interface to RACER*: It handles the communication between the components of the RACOoN tool and the RACER tool.

The work of Ren et al. (2004) describes a prototype tool for applying refactorings to use case models. In Figure 3.6 is represented the overview of the refactoring tool for use cases. The tool is composed of two subsystems, *Refactoring Framework* and *Prototype tool*. The *Refactoring Framework* subsystem is responsible for conducting the entire refactoring process, where each refactoring has a pre-condition that is a requirement of the condition package (*Condition*). At the beginning of a refactoring the precondition is checked by the *CodeModel* package which analyzes the use case model and determines whether the precondition is satisfied or not. If it is met, the *Refactoring* package applies the refactorings to the use-case model and triggers the *Change* package which is responsible for notifying the *CodeModel* package that the changes have been made and returns a backup, in case the changes have to be undone. The *Tool Interface* package is used to communicate with the *Prototype Tool* subsystem.

The subsystem *Prototype Tool* is used to evaluate the refactoring *framework*, through two main packages. The *Refactoring Tool GUI* package allows the user to start a refactoring

[2]DLs can be understood to be formalisms of the knowledge base and each DL consists of a pair of *Tbox* and *Abox*. A *Tbox* is used to introduce names for a complex concept and a *Abox* represents a set of individuals that are instances of concepts or represent the population of a given role in a formalism.

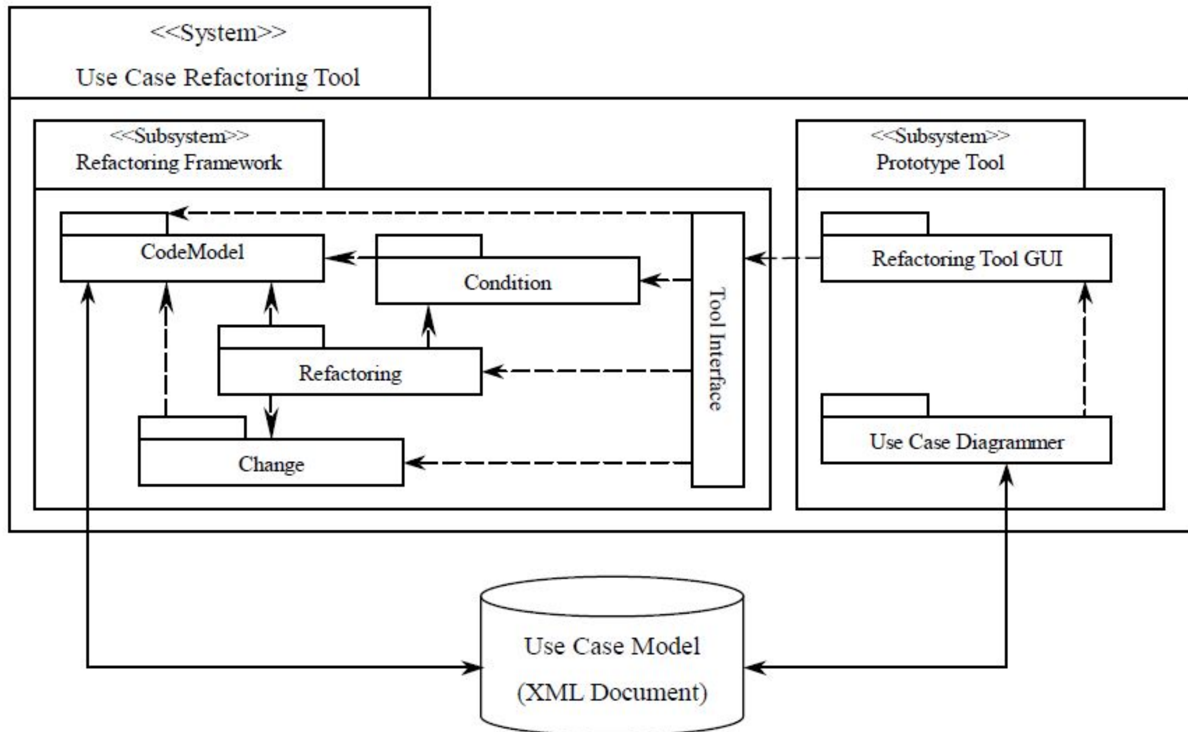

Figure 3.6: Overview of the use case refactoring tool proposed by Ren et al. (2004).

by providing corresponding input information. Afterwards, this package sends the requirements to the *Tool Interface* package of the *Refactoring Framework* subsystem to complete the refactoring. After applying the refactorings, the user can see the results through the *Use Case Diagrammer* package, which is responsible for showing the refactorings made in the use case diagram.

The approach proposed by Moghadam e Cinneide (2012) consists of refactoring systems based on both a desired project (target project) and an existing source code. The first step of the approach is to create the desired system design, based on the existing design, using a UML class model. Then the source code is refactored using the target system design. The source code resulting from these refactorings should have the same behavior as the unrefactored version, but the system design should be closer to the planned design.

In Figure 3.7 it is possible to see the main steps of the refactoring process proposed by Moghadam e Cinneide (2012). The approach is divided into two phases, the detection phase (*Detection Phase*) and the reification phase[3] (*Reification Phase*). The detection phase involves activities related to structural detection of differences between the two versions of the system and identification of the refactorings that will be used. Initially, an UML class model is extracted from the original source code (*Fact Extractor*) and is called

[3]Reification is a process that consists of transforming abstract concepts into concrete realities.

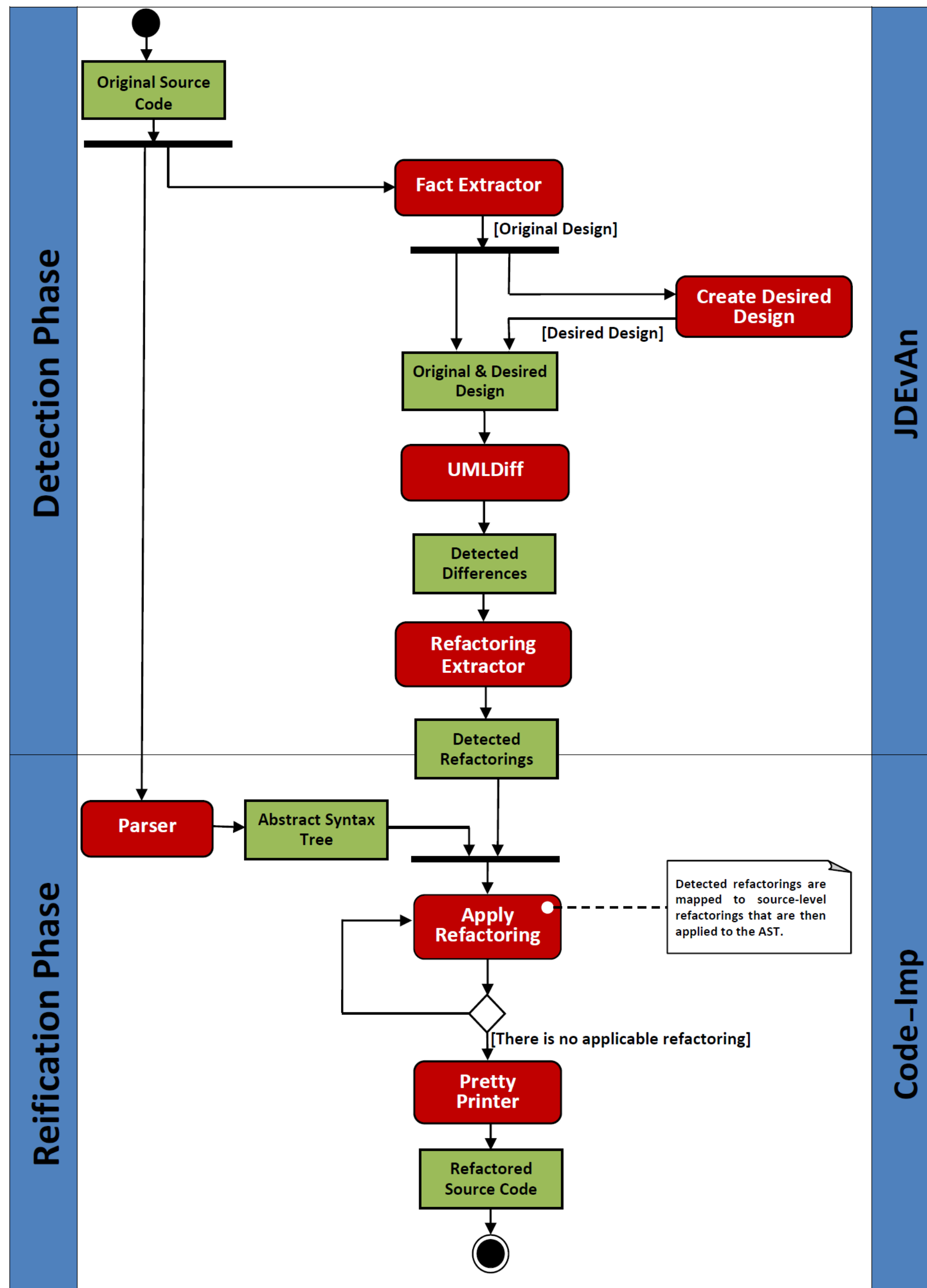

Figure 3.7: Refactoring workflow proposed by Moghadam e Cinneide (2012).

the original project. A second template, named project desired, is an updated version of the original that has been modified by a domain expert so that this system can meet new demands or simply fix identified issues. These two models are then compared by a differentiation algorithm (*UMLDiff*) and thus a relation of the differences that were found between the original version and the target version (*Detected Differences*) is generated.

These differences are then automatically categorized as detected refactorings (*Detected Refactorings*) through a set of predefined queries (*Refactoring Extractor*).

The reification step includes activities related to refactoring the source code of the original system based on the refactorings that were detected. The original source code is refactored using a heuristics approach based on detecting refactorings in order to get as close to the desired design. The initial step is to extract an abstract syntax tree (*Abstract Syntax Tree*) from the source code that contains all the necessary information for the transformation process. Then, the refactoring process is repeated until all the refactorings that were detected have been processed or there are no more refactorings that satisfy the requirements of the search technique. After applying the last possible refactoring, the abstract syntax tree is transformed into a refactored source code (*Refactored Source Code*).

The papers that were presented in this section characterize approaches that deal with instances of UML metamodel in the refactoring process. Many of these refactorings cannot be applied in other contexts outside the UML, however, the way these tools are structured, how the internal modules communicate with each other or among other tools are important for the construction of our RA. For instance, Einarsson e Neukirchen (2012) presents an example of how two models representing the same system can be refactored at the same time and how they can be manipulated. This paper also presents an alternative on how to proceed when it is necessary to develop *black-box* algorithms that are related to refactorings.

The paper of Van Der Straeten et al. (2007) presents an approach to preserve the behavior of refactored systems. The authors demonstrate through the tool architecture how the synergy can be made between tools that work together to solve a single problem. Thus, an important point is to know how to specify and design the input and output of each tool.

The paper of Ren et al. (2004), which presents an approach on how to apply refactorings to use case models, demonstrates the importance of providing a view of the changes caused by the refactorings and how these changes can be reversed if the user chooses to perform this process.

Finally, the paper of Moghadam e Cinneide (2012) demonstrates a complete modernization process, based on UML class diagrams, in which an improved/modernized system can be obtained from a legacy source code through the use of models .

Each of these works has different proposals and objectives, however, they are all related because they are dealing with model refactorings and because their approaches are complementary. The other publications found by Misbhauddin e Alshayeb (2015) were also

analyzed in order to identify architectural decisions that can contribute to the development of the RA proposed in Chapter 5.

3.3 Systematic Mapping

The starting point on collecting information to elaborate our reference architecture was the analysis of a systematic mapping (SM) on ADM presented by Durelli et al. (2014). Thus, in order to collect the recent papers about ADM we created a new systematic mapping based on the systematic mapping presented in (Durelli et al., 2014). We incremented the search string with new synonyms and new words. Also, we adjusted the protocol to include a broader range of papers, mainly by including other digital libraries and new research questions.

In addition, refactorings applied on metamodel instances play an important role when considering the modernization of software systems. Thus, we performed another systematic mapping to gather some of the existing refactoring tools in the literature to classify the approaches. In the following section, we present both systematic mappings.

3.3.1 Systematic Mapping on ADM

In this section, we present the research conducted in order to find relevant papers that present ADM related approaches. We followed the processes presented in Figure 3.8.

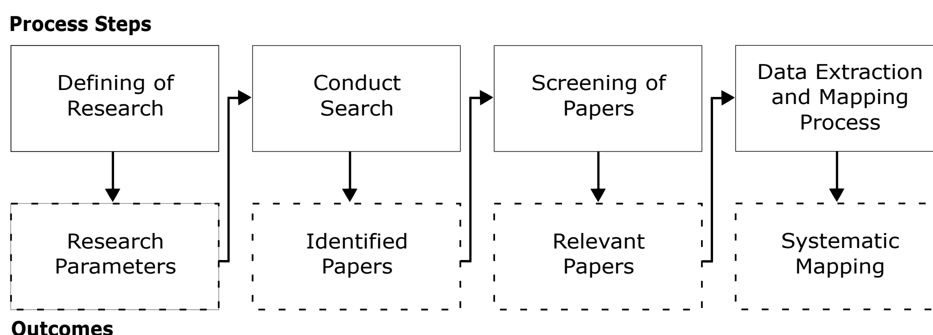

Figure 3.8: The systematic mapping process (Adapted from Durelli et al. (2014)).

Defining the Research. In this step, the definition of the systematic mapping protocol was elaborated, and the main produced artifact is the protocol which contains all the information needed to delimit the SM scope. The main information in our protocol are the Research Questions (RQs), the search string, the source list, the study selection criteria (inclusion and exclusion), and the data extraction form field.

Table 3.1: Search string for ADM Systematic Mapping.

(“Abstract Syntax Tree Metamodel” OR “Architecture-Driven Modernization” OR “Model-Driven Modernization” OR “Knowledge Discovery Metamodel” OR “Structured Metrics Metamodel” OR “Automated Enhancement Points” OR “Automated Function Points” OR “Structured Assurance Case Metamodel” OR “Structured Patterns Metamodel Standard”)

In this SM we elaborated research questions that could provide an overview of what is being studied in the literature from 2014 till 2018. Our RQs are as follows:

- RQ1 – What kind of study is being presented in research that involves ADM?
- RQ2 – What are the main functionalities/modules of the modernization tools presented in the papers?
- RQ3 - Which metamodels appear in publications?
- RQ4 - What are the focus area of these modernization tools (Reverse Engineering, Restructuring and Forward Engineering)?
- RQ5 – What are the main input and output artifacts presented?

In order to answer these questions we structured a search string broad enough to return the maximum ADM related papers as possible and it is represented in Table 3.1.

Conducting the Search. We applied the search string (Table 3.1) in electronic databases that are deemed as the most relevant scientific sources and therefore likely to contain important primary studies. We selected the following electronic databases: ACM, Engineering Village, IEEE Explorer, Scopus and Web of Science.

For this new SM, we considered only papers that were not presented in (Durelli et al., 2014) since they were already analysed in the context of this Ph.D. thesis. We also included new OMG modernization standards[4] to check if they were being somehow used in ADM context and two new digital libraries when comparing with (Durelli et al., 2014). In Table 3.1 we only presented the main keywords used for this search. The search strategy can be seen in Appendix A.2 - ADM Search String.

Screening of Papers. In order to determine which primary studies are relevant to answer our research questions, we applied a set of inclusion and exclusion criteria. The inclusion criterion applied was:

[4] Automated Enhancement Points (AEP); Automated Function Points (AFP); Structured Assurance Case Metamodel (SACM) and Structured Patterns Metamodel Standard (SPMS)

(i) It presents any kind of ADM-based approach or mention; and (ii) It uses any ADM metamodel.

Our exclusion criteria were:

(i) Already included in the last systematic mapping; (ii) Do not present enough information (Short papers); (iii) Introductory papers for books and workshops; (iv) It is not an ADM-based approach; (v) Not in English, Spanish or Portuguese; and (vi) This is not a paper.

We initially recovered 432 papers. Among them, 224 were identified as duplicated papers, and 159 were rejected by one of our exclusion criteria. We selected 49 papers to be classified and they were divided into two groups. The first group was composed of 14 papers that presented researches that involved ADM in general discussions or other applications that were not related about modernization tools. The second group was composed of 35 papers that presented modernization tools. The complete list of selected papers can be seen in Appendix A.1 - Final selected papers. The data extraction form field was composed of five fields:

- (i) What are the Focus Area (Modernization Phases) presented in the studies?
- (ii) What are the main functionalities/modules of the modernization tools?;
- (iii) What are the Metamodels employed in the study? and
- (iv and v) What are the main input and output artifacts presented?

Data Extraction and Mapping Process. As we divided the selected papers into two groups, we describe the data extractions and mapping processes of these two groups in the following sections.

3.3.1.1 Group 1 - Discussions around ADM papers

In this section we aim to respond RQ1 that is related to the studies that are being conducted about ADM that is not related to modernization tools.

In the 14 papers analysed we could see discussions about KDM extensions, analysis about KDM metamodel and also about ADM modernization involved processes. In following paragraphs we provide a short summary of each paper of this group.

Arcelli Fontana et al. (2017) takes KDM and other four metamodels that support reverse engineering process and their extensions available in the literature in order to describe and compare them. The main goal is to discuss about this kind of metamodels

in order provide a scenario where a software engineers could understand and choose the metamodel that better suit their need.

Santos et al. (2018) explore the refactoring process by displaying how this process is employed in the ADM modernization process. This paper also provides some examples and discussions about refactoring modernization tools available in the literature and industry.

Sabiri e Benabbou (2017) propose a new metamodel to support the migration of legacy software systems to the cloud to be used in the modernization process proposed by ADM. This metamodel is composed of three viewpoints (business, data and implementation, and infrastructure) that support engineers in the preliminary feasibility analysis of the migration process.

Durelli et al. (2014) present a Systematic Mapping on ADM. This paper was used as a starting point of the SM presented in this section and it was included as a parameter to ensure that our search string was retrieving all the papers we needed from digital libraries.

Pires e e Abreu (2018) present an approach to allow the generation of Unit tests by means of model transformations that uses KDM as intermediate representation for existing software systems.

Akodadi (2016) present a study that compare existing cloud migrations method that are model-driven. The final outcomes are a table displaying the pros and cons of each approach and the authors propose their own approach that considers the modernization processes based on ADM.

Mansurov e Campara (2004) present a development approach called Managed Architecture that is focused on evolution of existing software assets. This paper employs the concepts of ADM and it also raises a discussion about why UML is not the best metamodel to be used on Managed Architectures and why KDM is the chosen metamodel for this goal.

Alawneh e Hamou-Lhadj (2009) present a discussion about the software systems runtime representations. A survey of existing metamodels for software system representation is displayed and a proposal of KDM extension for representing execution traces is presented.

The next three papers discuss about KDM extensions. In Jácome e De Lara (2018) work is presented a tool to support metamodel extensions, including OMG standards such as KDM and Diagram Definition (DD). In Durak (2015) work is presented a KDM extension for representing existing simulation software to enable the ADM processes in this domain. And in Santos et al. (2019a) work we can see two approaches for KDM extensions to represent aspect-oriented source code in KDM instances. The first approach is the lightweight one that add the new representations by means of stereotypes/annotations on the existing KDM metaclasses and the other approach is the heavyweight that is

materialized by adding new metaclasses on the existing KDM metamodel. The common point of these approaches is that the authors claim that KDM lack of specific domain representations that could be solved with extensions on the referred metamodel.

This final set of papers is about Structured Assurance Case Metamodel (SACM). As mentioned before, we included other software representation metamodels from OMG to check if any author was using them in ADM context and if they could be used to support KDM. However, the three identified papers (de La Vara et al., 2017; Muram et al., 2018; de la Vara, 2014) are not the case and we decided to include them here only for register this finding.

3.3.1.2 Group 2 - Modernization Tools papers

In this section we aim to respond RQ2, RQ3, RQ4 and RQ5 that are related to the modernization tools that were analysed in the selected studies.

In Figure 3.9 we present the main functionalities identified in the modernization tools analysed. Not all the approaches presented an existing modernization tool, some of them was just a proposal. However, we decided to include them in this research since the main point of our reference architecture is to point out the architectural decisions.

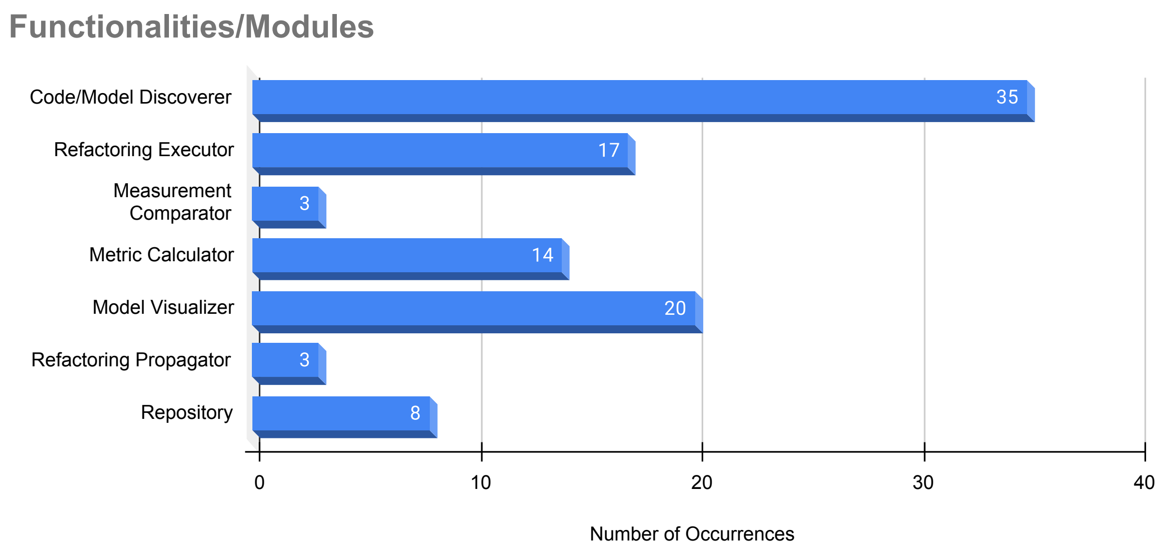

Figure 3.9: Modernization Tools main Functionalities/Modules.

We created standard modules names in order to group the functionalities/modules identified. For instance, the Code/Model Discoverer module is composed of any functionality that involved knowledge discovery, either in reverse engineering or in forward engineer-

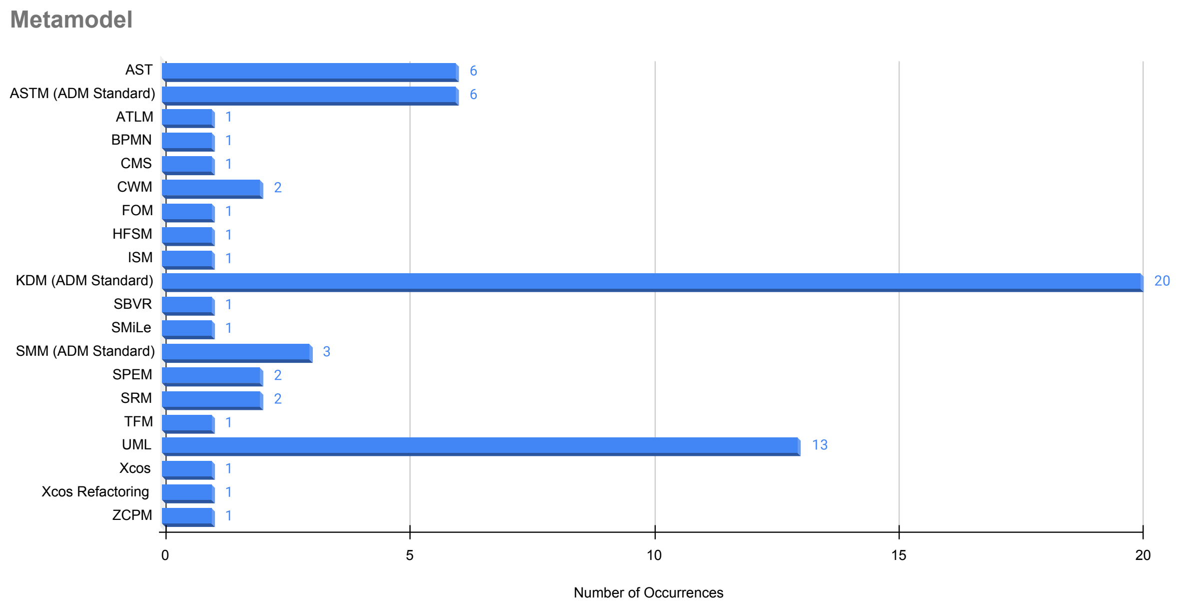

Figure 3.10: Metamodels identified in the modernization tools.

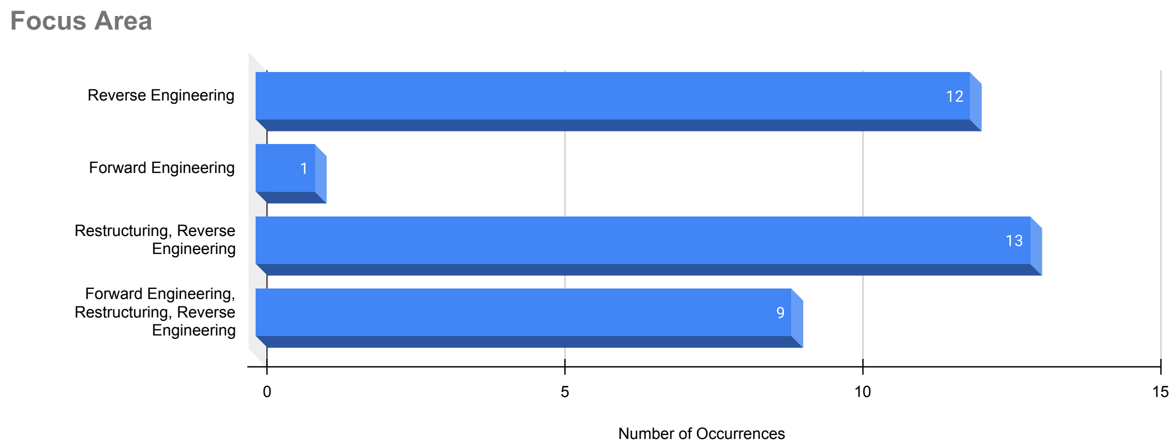

Figure 3.11: Focus Area of the analysed modernization tools.

ing. Another example is Measurement Comparator that grouped any functionality that evaluated measurements in order to support changes in software refactoring functionalities.

In Figure 3.9 all the MT analysed presented a Code/Model Discoverer step. This is a logical finding, since the knowledge discovery is a crucial process in ADM and this is a starting point of any other process inside a modernization project. We also could notice that Model Visualizer, Refactoring Executor and Metric Calculator functionalities followed Code/Model Discoverer functionality and presented the highest number of occurrences.

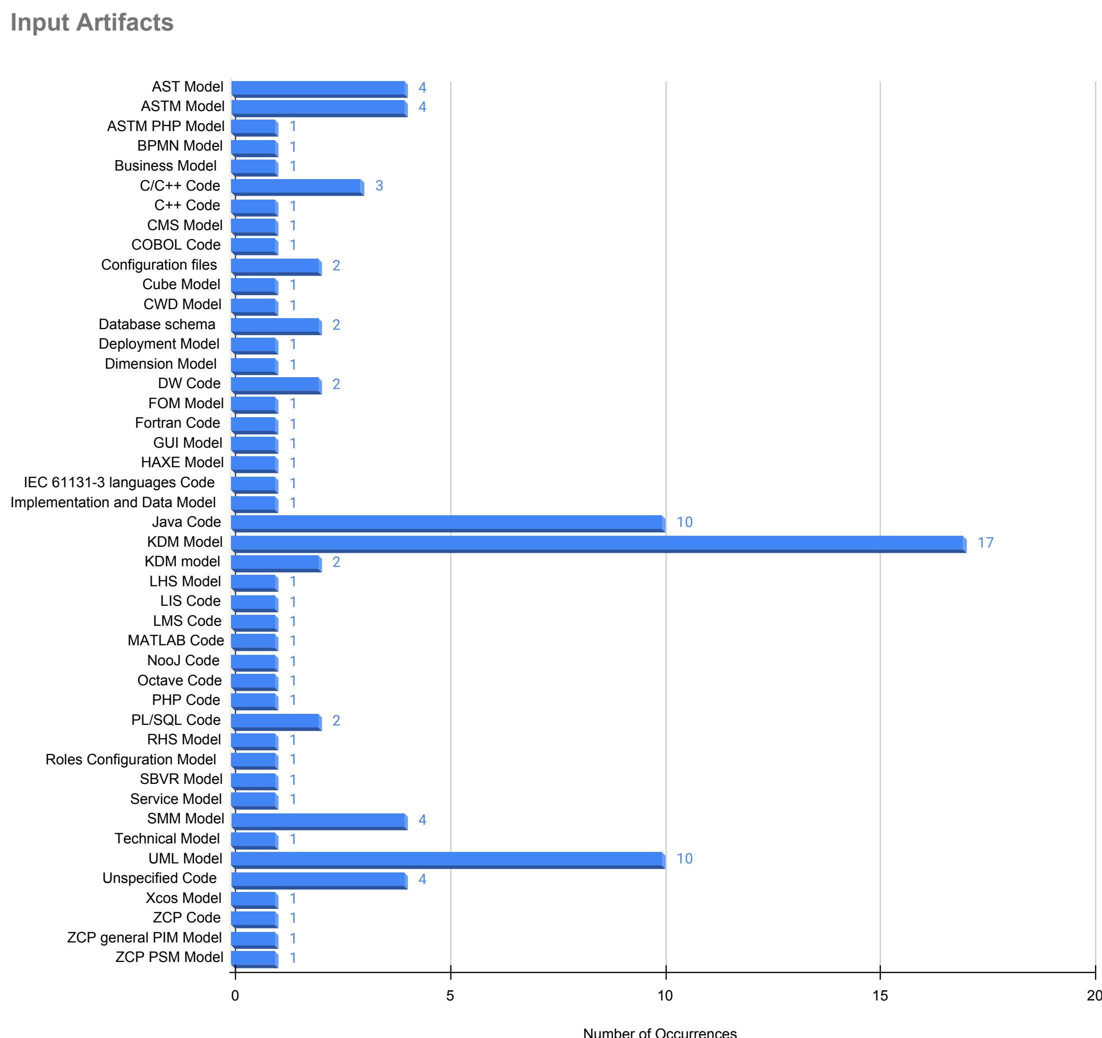

Figure 3.12: Modernization Tool's Input Artifacts.

Thus, Figure 3.9 summarizes the main functionalities/modules in the MT in order to answer RQ2.

In Figure 3.10 we present a list of all the identified metamodels in the MT analysed. Some of the occurencies were identified as standard metamodels, such as UML, BPMN and KDM. However, we added a tag **ADM standard** to mark the standard metamodels advocated by ADM, which are: KDM, SMM and ASTM. Our main goal was to identify how they were employed in modernization tools. KDM metamodel had the highest number of occurrences and it was used in two main contexts: (i) main metamodel in the tool supporting the refactorings; and (ii) auxiliary metamodel to support other higher abstraction level metamodels. KDM and UML had the highest number of occurrences,

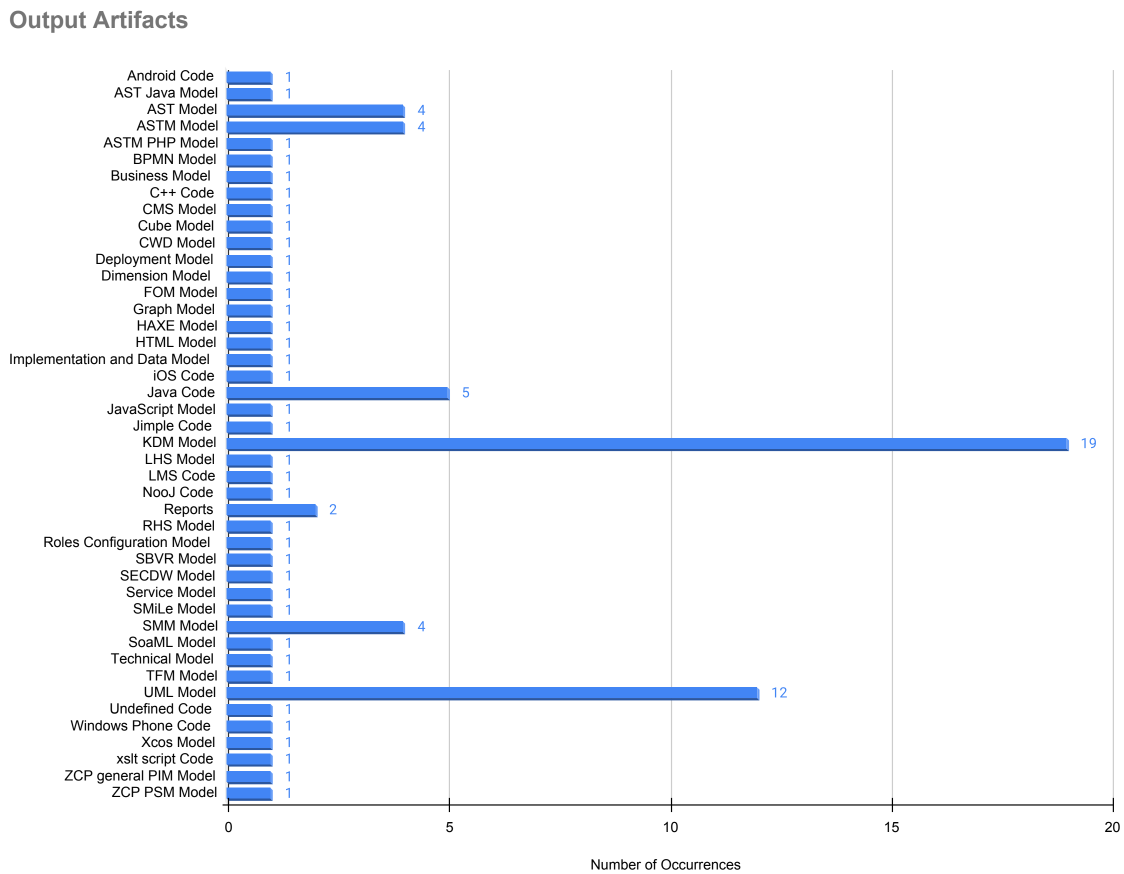

Figure 3.13: Modernization Tool's Output Artifacts.

20 and 13 respectively, and we notice that UML was mainly used for supporting the refactorings visualization. Figure 3.10 answers RQ3 by displaying the metamodels used in the modernization tools of group 2.

RQ4 aims to discover what are the main focus area of the modernization tools analysed. In Figure 3.11 It is possible to see that 25 of the tools displayed approaches to discovery knowledge from code or model by means of Reverse Engineering. This is the result of Reverse Engineering count(12) with Restructuring, Reverse Engineering count(13).

Since ADM is mainly focused on software modernization, we claim that restructuring tools need at least an initial model representation to start the modernization process. This explains that the highest occurrences involves Restructuring and Reverse Engineering in the same tool. Only 9 of the MTs displayed a full cycle of the modernization processes of ADM and only one tool proposed an approach to support the Forward Engineering process.

Finally, In Figure 3.12 and 3.13 we display the main input and output identified in order to answer RQ5. In both we could notice that KDM has the highest occurrences. Some tools only uses KDM as an input to generate other metamodel instance and other tools could generate the KDM instance and transform this instance in an improved one. The other occurrences are fragmented since each MT has one purpose and we analysed all the inputs and outputs from each modernization process involved in the tool. The process of identifying the inputs and outputs of each MT was scattered through the paper since sometime we could only retrieve these information in case study, conclusions or related works sections. This analysis gave us an overview of how KDM could be used in different modernization scenarios and how interoperable KDM is when used together with other metamodels.

3.3.2 Systematic Mapping on Code and Model Refactoring tools

In this section, we present the research conducted in order to find relevant papers that present refactoring tools in models or code. We followed the processes presented in Figure 3.8.

Defining the Research. In this step, we produced systematic mapping protocol artifact with all the information needed to delimit the SM scope. Here we present the Research Questions (RQs), the search string, the source list, the study selection criteria (inclusion and exclusion), and the data extraction form field.

The RQs guide the conduction of the research and impact directly the whole SM since the analysis aims to answer such questions. Our RQs are as follows:

- RQ1 – What are the main functionalities/modules of the refactoring tools?
- RQ2 - How the refactoring tools are presented in the papers and how they are classified?
- RQ3 - Which metamodels appear in publications?
- RQ4 – What are the main input and output artifacts presented?

In order to answer these questions we structured a string broad enough to return the maximum related papers as possible and it is represented in Table 3.2.

Conducting the Search. We applied the search string (Table 3.2) in electronic databases that are deemed as the most relevant scientific sources and therefore likely to contain

Table 3.2: Search string for code and model refactoring.

(“model refactoring” OR “code refactoring”) AND “tool”

important primary studies. We selected the following electronic databases: ACM, IEEE XPLORE, Scopus and Springer.

Screening of Papers. In order to determine which primary studies are relevant to answer our research questions, we applied a set of inclusion and exclusion criteria. The inclusion criterion applied was: the paper presents a refactoring tool. Our exclusion criteria were: (i) paper not written in English or Spanish; (ii) the focus is not in the refactoring tool; (iii) the paper does not present a refactoring tool, and; (iv) not enough information about the tool.

We initially recovered 398 papers. Among them, 35 were identified as duplicated papers, and 284 were rejected by one of our exclusion criteria. We read all the titles, abstracts, and keywords of the 79 papers. This was the first selection phase.

In the second selection phase, we read all the 79 selected papers to identify the papers that provided the tool with more relevant architectural information. The final selected set is composed of 30 papers[5]. The other 49 papers were excluded since their focus was not on the tool or there was insufficient information to extract based on our research questions. The data extraction form field was composed of six fields:

- (i) What are the main functionalities/modules of these tools;
- (ii) How the tools are presented;
- (iii) The classification;
- (iv) Metamodels employed; and
- (v and vi) What are the main input and output artifacts presented.

Data Extraction and Mapping Process. After reading and extracting all relevant data from the accepted papers we normalized the data in order to present them into graphics.

In Figure 3.14 we present the functionalities identified by our mapping in the tools. Since the main acceptance criterion was to be a refactoring tool, all 30 tools presented a Refactoring Executor module. The second most identified module was the Code/Model Discoverer. We claim that this is a crucial module when talking about refactoring tools

[5]The complete list of selected papers can be seen in Appendix B

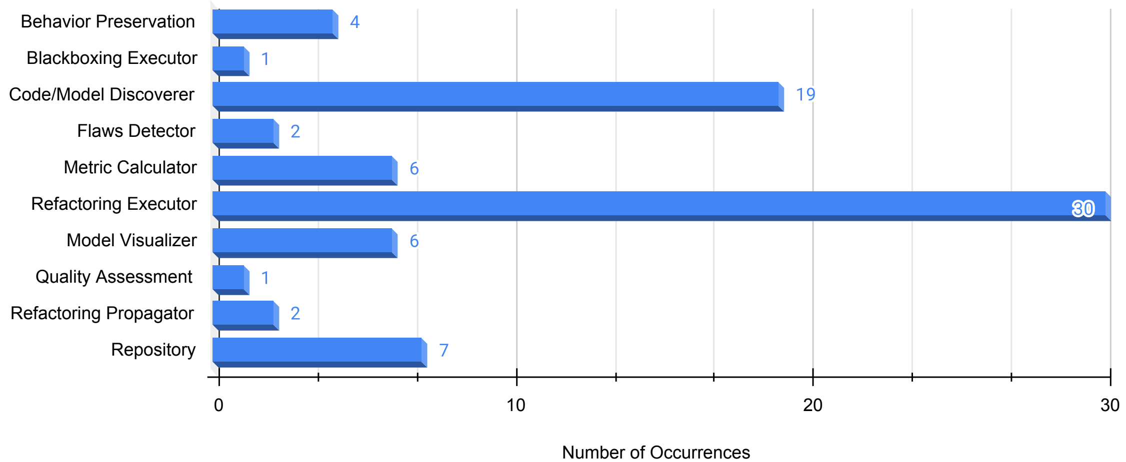

Figure 3.14: Main Functionalities/Modules.

because before applying refactorings in a source code or a metamodel instance, we need to either transform it from source code to model or transform model-to-model and/or apply some logic to analyze the as-is situation to decide the refactorings to be applied. Thus, in the category Code/Model Discoverer, we grouped functionalities that involved transformations among different abstraction levels and code/model analysis. Any tool that provided modules for refactorings or analysis visualizations were grouped in the Model Visualizer Module. Another example of grouping depicted in Figure 3.14 is the Repository that groups all tools' components that were responsible for storing transformation rules, source code, metamodel instances, or any other artifact. Thus, Figure 3.14 helps us answer RQ1 by displaying the main identified functionalities/modules in the analyzed tools.

In Figure 3.15 we show how the authors presented their tools in the papers. We concluded that most of the tools were presented as a set of tools, meaning that in some cases the author described a tool to perform a refactoring on a model, and also described a tool to visualize the result of the refactoring. In other cases, the authors described tools that worked together to achieve the goal. When more than one tool could be identified, we categorized the availability of these tools as a set. Also, there were papers that the functionalities were described as a mathematical or computational formalism. In this case, the authors claimed that there was a tool, but the focal point of the paper was to show how the refactorings could be mathematical or computational proved. Most of the tools were classified "as a set of tools" and "as an eclipse plug-in", followed by "as an algorithm" classification.

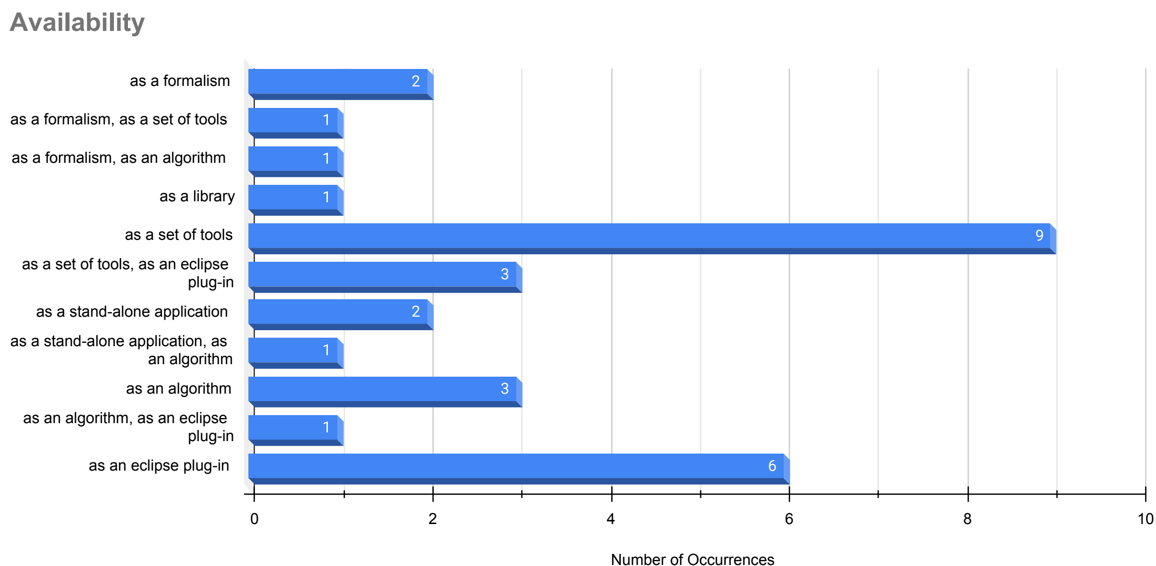

Figure 3.15: Classification about the tool's availability.

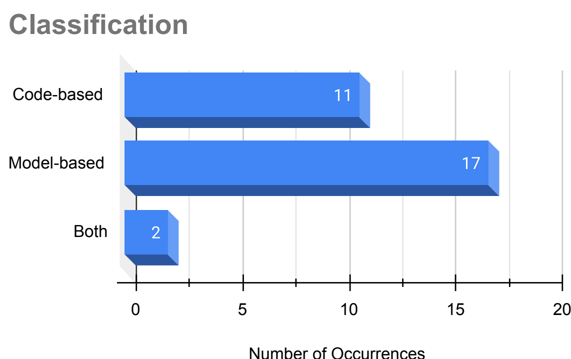

Figure 3.16: Tool's Classification regarding their abstraction level.

Still focusing on RQ2, in Figure 3.16 we also analyzed how the tools could be classified when considering the abstraction level of where the refactorings were applied. To do this classification, we considered as code-based tools the refactorings that were applied on a Platform-Specific Model such as Java Model, Abstract Syntax Trees (ASTs), and others. Model-based tools were considered when the abstraction level was higher than PSM, platform, or computation independent models, such as UML models, proprietary metamodels, and others. Thus, we concluded that 17 of the analyzed tools are model-based, 11 are code-based, and 2 are both code and model-based since refactoring is applied in different abstraction levels.

In Figure 3.17 we present the list of metamodels used in the tools. The most used representation is the AST model. We claim that code-based tools only need to use an

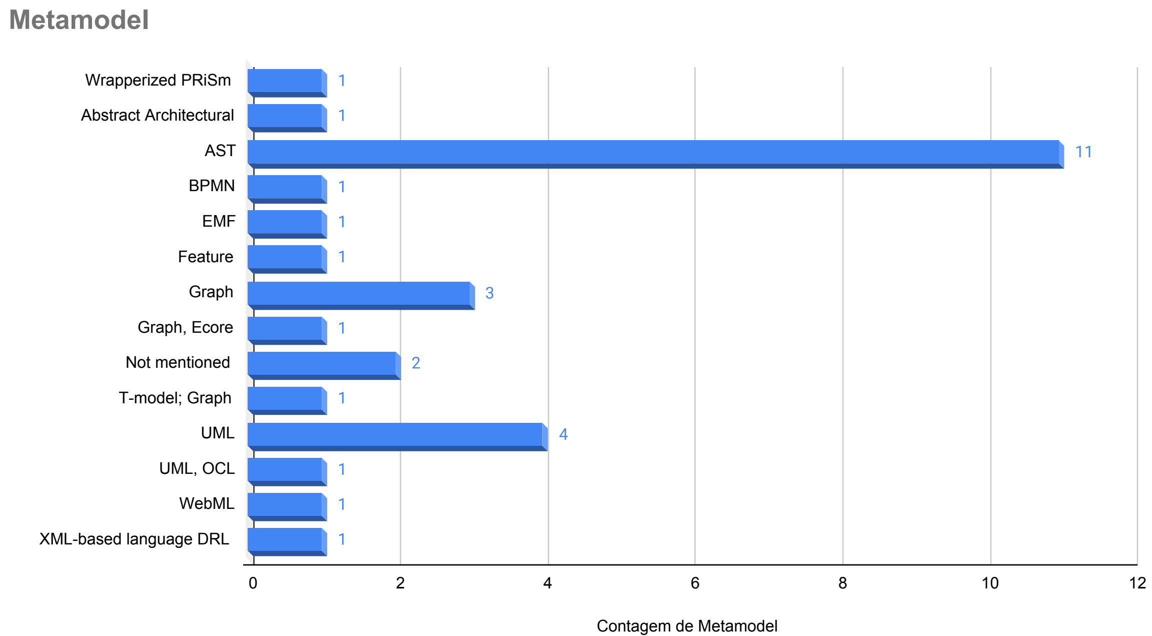

Figure 3.17: Metamodels identified in the refactoring tools.

AST representation, explaining why this is the metamodel with the highest occurrence. However, since ASTs are platform-specific representations, these ASTs are not compatible with each other since we could find C, C++, Java, and other ASTs representations.

We also noticed that standard metamodels appeared in model-based tools like EMF, UML, and BPMN. Nonetheless, we notice several other metamodel instances that will probably not be interoperable due to their particularities on representing the software systems to be refactored. Thus, answering RQ3, we have the predominance of AST models, and we could also see Graph, EMF, UML, BPMN, among other representations.

In Figures 3.18 and 3.19 we present the input and output artifacts from the analyzed tools. In Figure 3.18, answering RQ4, the most common inputs are Java code, Graph Notation, and UML model. In Figure 3.19, we observe the same pattern. Regarding the refactoring tools that we selected in this SM, we claim that they can represent how refactoring tools usually are not concerned about interoperability, as we observe in Figures 3.18 and 3.19. The authors appeared to be more concerned with the refactorings themselves than regarding how the tool could be reused in different contexts (languages and scopes). Another discussion we could point out is that when developing a refactoring tool, the researcher has to think about the refactorings to be performed and other modules that will support the refactorings. At the same time, they have to forget to design the tool's

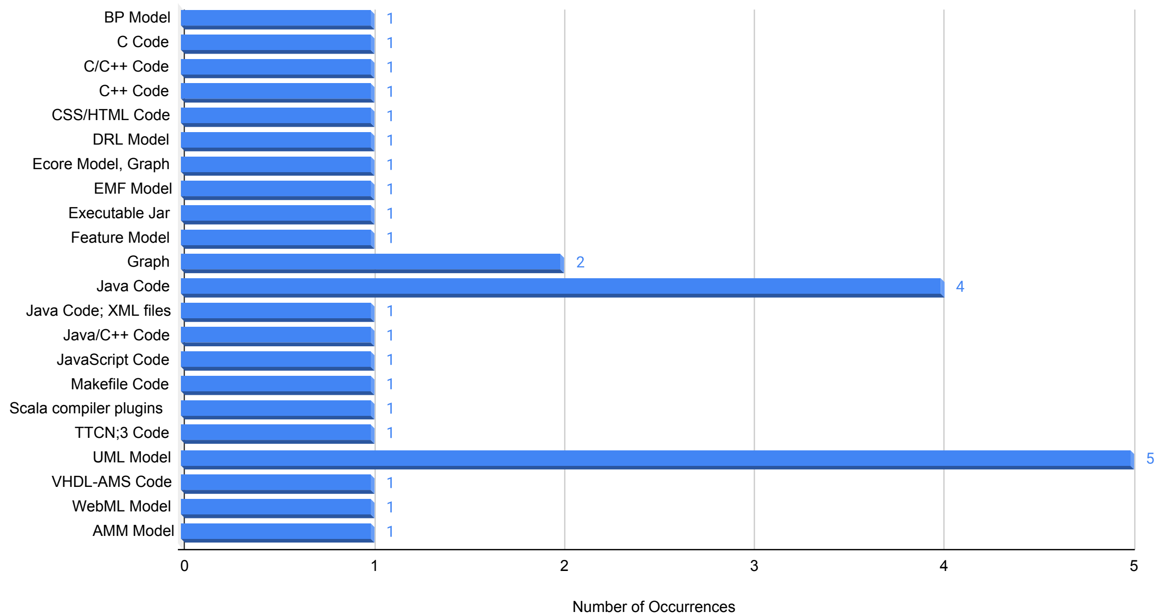

Figure 3.18: Refactoring Tool's Input Artifacts.

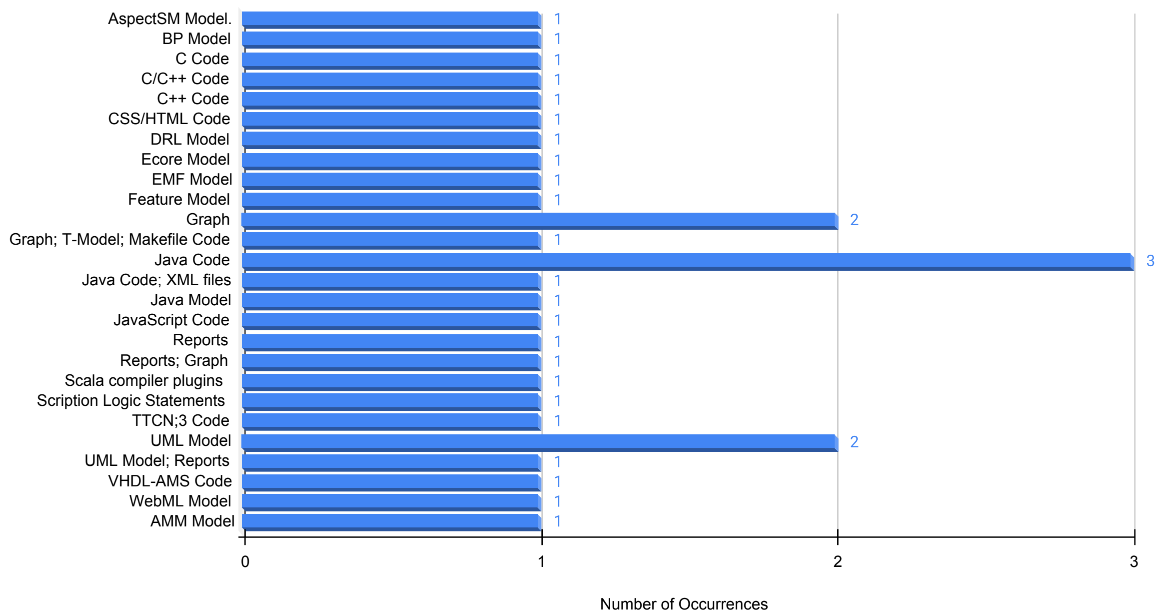

Figure 3.19: Refactoring Tool's Output Artifacts.

architecture considering a bigger picture where this tool could be reused or interoperable with other tools.

3.4 Final Considerations

In this chapter, the main groups of works related to this research were presented, which are the works that involved refactorings for UML metamodel and the works that use the concepts of ADM and KDM metamodels. These related works contributed directly to the development of the proposed RA, since they contributed with the theoretical basis to assist the RA construction process.

The approaches presented in section "Modernization tools that use ADM standards" contributed to the elaboration of this Ph.D. research since they all presented relevant descriptions about the modernization flow and how they are organized in their architecture. In summary, they all:

- Utilize ADM's modernization principles;
- Use the standard ISO KDM metamodel;
- Use a metamodel to view the information contained in a KDM instance;
- Apply transformations to models.

The approaches presented in section "Tools that apply refactorings in UML" contributed with their architecture and modules/functionalities. This set of tools contains three main common characteristics that are relevant to building our RA:

- The transformations are applied on metamodel instances level;
- They use at least one standard metamodel in the core of its algorithm (UML); and
- The transformations flow provided in theirs architecture has similar components to those expected in ADM-based modernization tools.

In addition, we claim that the systematic mappings on ADM and on code and model refactoring tools played a crucial part on the elaboration of our RA not only in the modules and functionaries but also in the architectural views elaboration presented in Chapter 5.

In Chapter 4, a taxonomy for modernization tools based on ADM is shown, which was also elaborated through the bibliographic survey carried out and described in this chapter.

Chapter 4

A Taxonomy for Classifying Modernization Tools in ADM Context

4.1 Initial Considerations

This chapter presents a taxonomy for classifying tools that are used in modernization contexts. The intention is to establish terms that really express the type of the tools, making the comprehension easier, as well as the knowledge sharing among developers and practitioners.

Although there are other works in the past that have already presented taxonomies and terminologies around the theme of software reengineering (Bourque e Abran, 1994; Chikofsky e Cross, 1990; M. Klein, 1994), there is a need for a taxonomy more focused on the current advances of software modernization, particularly focused on the ADM concepts. Therefore, our taxonomy has not the intention of competing with those other works, but just provide a conceptual framework so that software engineers can talk about modernization tools, in the ADM context, using the same vocabulary.

The proposed taxonomy is useful for classifying modernization tools considering their purposes, used metamodels, abstraction level, etc. Therefore, software and modernization engineers that work with modernization tools can have a better view of the existing types of tools they are working with.

We have detected only three works that have some relation with our taxonomy; the work of Bourque e Abran (Bourque e Abran, 1994), the work of Chikofsky (Chikofsky e Cross, 1990) and the work of Klein (M. Klein, 1994).

The work of Chikovsky, which is one of the first in this field, present basic definitions on reengineering, redocumentation, design recovery, reverse engineering and reestructuring. It is not focused on classifying tools for this context as is our case. Besides, at that time, tools for assisting reengineering project were very scarce.

The work of Bourque and Abran (M. Klein, 1994) are the closest to ours, since they present a taxonomy for software reengineering tools. The difference is that all the thirteen categories presented by them are devoted to the **functional goal** of the tool. For example, one cay say that a specific tool is a *Design Recovery Tool* or that another specific tool is a *Impact Analysis Tool*. We do not have this kind of classification in our taxonomy, but this can be considered in future works. The work of Klein (M. Klein, 1994) present six categories for reengineering tools but all the categories are very high-level. The difference is that is that our taxonomy is focused on classifying internal properties such as metamodels, abstraction level, and purpose, which can be applied in all sets of modernization tools.

4.2 The Taxonomy

Figure 4.1 presents a diagram that graphically represent our taxonomy. As we are using a class diagram, the known concepts of inheritance, aggregation, abstract classes, multiplicities, associations and reading direction in relationships are being used. The white rectangles are the **terms** of the taxonomy and the gray ones are the **classification labels** that acts as classifiers.

Regarding the terms, we have two main ones: **Modernization Environment** (ME) and **Modernization Tool** (MT), which are stereotyped as **main** in the diagram. MEnv and MTs are the most used terms, since they classify tools in this context.

> As can be seen in the diagram, we consider a ***Modernization Environment as an aggregation of one of more Modernization Tools***.

Computational Support (CompSu) is an abstract term, which in fact, are being used as the highest level term, i.e., both Modernization Environments and Modernization Tools are Computational Supports. Since CompSu is just representing the link between ME and MT, this is not a crucial term here.

Modernization Tool (MT) is the central term and also acts as a family of terms, as it is the root of a hierarchy. It is a concrete term because one can use it to designate a tool, for example, saying that "a specific reverse engineering tool is a Modernization Tool". It also acts as a family because it is the root of more specialized terms. Below one can find our definition for this term.

Modernization tools are tools that provide some level of automation for any of the steps of the modernization cycle.

As example, a MT can automatize (i) the generation of metamodel instances, (ii) the process of calculating metrics from legacy systems, and also (iii) the execution of model transformations/refactorings.

In the second hierarchy level of the MT, there are two subtypes: **Common Modernization Tools** (Common-MT) and **ADM-Modernization Tools** (ADM-MT or ADM-based Modernization Tools). The difference is that ADM-MTs employ ADM metamodels in their architecture but Common-MT do not. From an usage point of view, a Common-MT has the same functionality and goals of any other ADM-MT. However, they do not necessarily concern with the usage of standard metamodels. The works of Mercier et al. (2017) and Madisetti et al. (1999) can be cited as examples of Common-MT.

In the third level of the hierarchy one can find other two subtypes of ADM-MT that are **OMG Modernization Tools** and **KDM Modernization Tools**. The KDM-MT use KDM internally for representing systems to be modernized. However, OMG-MT are tools that use any other OMG metamodel internally. Examples of ADM-MT can be seen in the works: Canovas e Molina (2010); Son e Kim (2016); Ulrich e Newcomb (2010a).

The term Modernization Environment was created to represent a wider computational support, providing a set of functionalities that assist along a modernization process. Therefore, a modernization environment is usually composed of a number of modernization tools that support the entire modernization process, from reverse engineering to obtaining the modernized target system. In other words, a modernization environment has several tools that work together to perform a complete modernization. For example, reverse engineering tools, data mining and analysis, restructurings, and source code generation from modernized models.

For instance, MoDisco (Brunelière et al., 2014) is a MEnv that supports the reverse engineering phase by allowing the knowledge discovery of java source code into different metamodel instances. With MoDisco it is possible to recover, for instance, UML and KDM models. MoDisco is composed of several discoverers that can be understood as MTs, and each discoverer deals with a specific metamodel.

Regarding the classification labels (gray rectangles), there are the following five ones: i) **Type** of the tool; ii) **Purpose** of the tool; iii) **Modernization Scenario** in which the tool is applicable; iv) **Metamodel** used by the tool and v) the **abstraction level** of the metamodel used.

The label **Type** is used for differentiate Modernization Tool from Modernization Environment.

Please notice that in Figure 4.1 we have some notations, which are: a) {*incomplete*} - indicates that the generalization set is not covering all possible instances; b) {*complete*} - indicates that the generalization set is covering all possible instances.

As previously said, our taxonomy involves five classification labels. Two labels are only applicable for Modernization Tools, which are: Metamodel and Metamodel Abstraction Level. The remainder ones (Type, Purpose and Modernization Scenario) are applicable for MTs and also for Modernization Environment.

The classification label **Modernization Scenario** aims at representing the scenarios in which MEnvs and MTs can be used. Modernization Scenarios are scenarios that show where and how a software system could be modernized, and they also help modernization teams on how to plan a modernization project (Ulrich e Newcomb, 2010b). MScens can act in three different architectural levels, which are: Technical, Application/Data, and Business Architecture. These architectural levels are important to define not only the set of MEnvs and MTs to be used but also the set of metamodels and their abstraction level that will be employed in the tools.

For instance, Language-to-Language Conversion is a Technical MScen responsible for converting a software system from one language to another. The main motivations for this scenario could be the obsolescence of a language, the existence of a requirement to enhance a functionality not supported in the current language, etc.

Purpose is another important classification level of our taxonomy. As purpose comes from Type, it represents that as Modernization Tools as Modernization Environments can have one or more purposes, which correspond to the classical phases of reengineering processes, which are: Reverse Engineering, Restructuring, and Forward Engineering.

Table 4.1: Modernization Scenarios (Adapted from Ricardo Pérez-Castillo e Piattini (2010))

Modernization Scenario	Description
Application Portfolio Management	This scenario aims to mine and represent metadata for all artifacts involved in an enterprise system. Legacy systems generally do not have much documentation and this scenario addresses this gap and can be applied in the process of modernizing applications/data or businesses.
Application Improvement	This scenario aims to improve the robustness, integrity, quality, consistency and/or performance of applications, but does not involve an effort to transform the architecture. This scenario is applied in the technical modernization processes.
Language-to-Language Conversion	This scenario deals with the conversion of one or more information systems from one programming language to another. This scenario does not involve a redesign of the system's functionalities and is applied in the technical modernization processes.
Platform Migration	This scenario moves systems from one platform to another due to platform obsolescence or to standardize the system to an organizational standard. This scenario is usually performed in conjunction with a language conversion and is also applied in technical modernization processes.
Non-Invasive Application Integration	This scenario is triggered when there is a need to bring legacy user interfaces to end users, replacing legacy front-ends with other front-ends, such as web-based interfaces, keeping system functionality intact. This scenario is applied in technical modernization processes.
Service Oriented Architecture Transformation	This scenario takes into account legacy systems that generally incorporate their functionality in a monolithic way and wish to migrate their functionality to *Service-Oriented Architecture* (SOA). This scenario can be applied to applications/data modernization processes.
Data Architecture Migration	This scenario moves one or more data structures to another, for example, moving from a non-relational file or database to the relational data architecture. This scenario can be applied to applications/data modernization processes.
System and Data Architecture Consolidation	This scenario is motivated by the need to build a single system from multiple autonomous systems that perform the same basic functions. It can be combined with other scenarios, such as model-oriented transformation, language change or platform migration. This scenario is applied to all applications/data modernization processes.
Data Warehouse Deployment	This scenario creates a common repository or data warehouse for business data, as well as the different ways to access that data. This scenario is usually realized in the processes of modernization of applications/data, but it can also be applied in the processes of business modernization.
Application Package Selection and Deployment	This scenario defines how legacy systems must be removed, integrated or configured to work with a specific component. This scenario is applied exclusively in the processes of modernization of applications/data.
Reusable Software Assets / Component Reuse	This modernization scenario helps to identify, capture and prepare functionalities and information resources of legacy systems for reuse. This scenario is also applied in the processes of modernization of applications/data.
Model-Driven Architecture (MDA) Transformation	This scenario converts a non-model driven environment to a model driven environment. Moving to MDA requires transforming legacy systems into a set of models that can be used to generate replacement systems. This scenario is applied in application/data modernization processes.
Software Assurance	This scenario aims to measure reliability in relation to the established business and security objectives. This scenario can be applied within application/data or technical update processes.

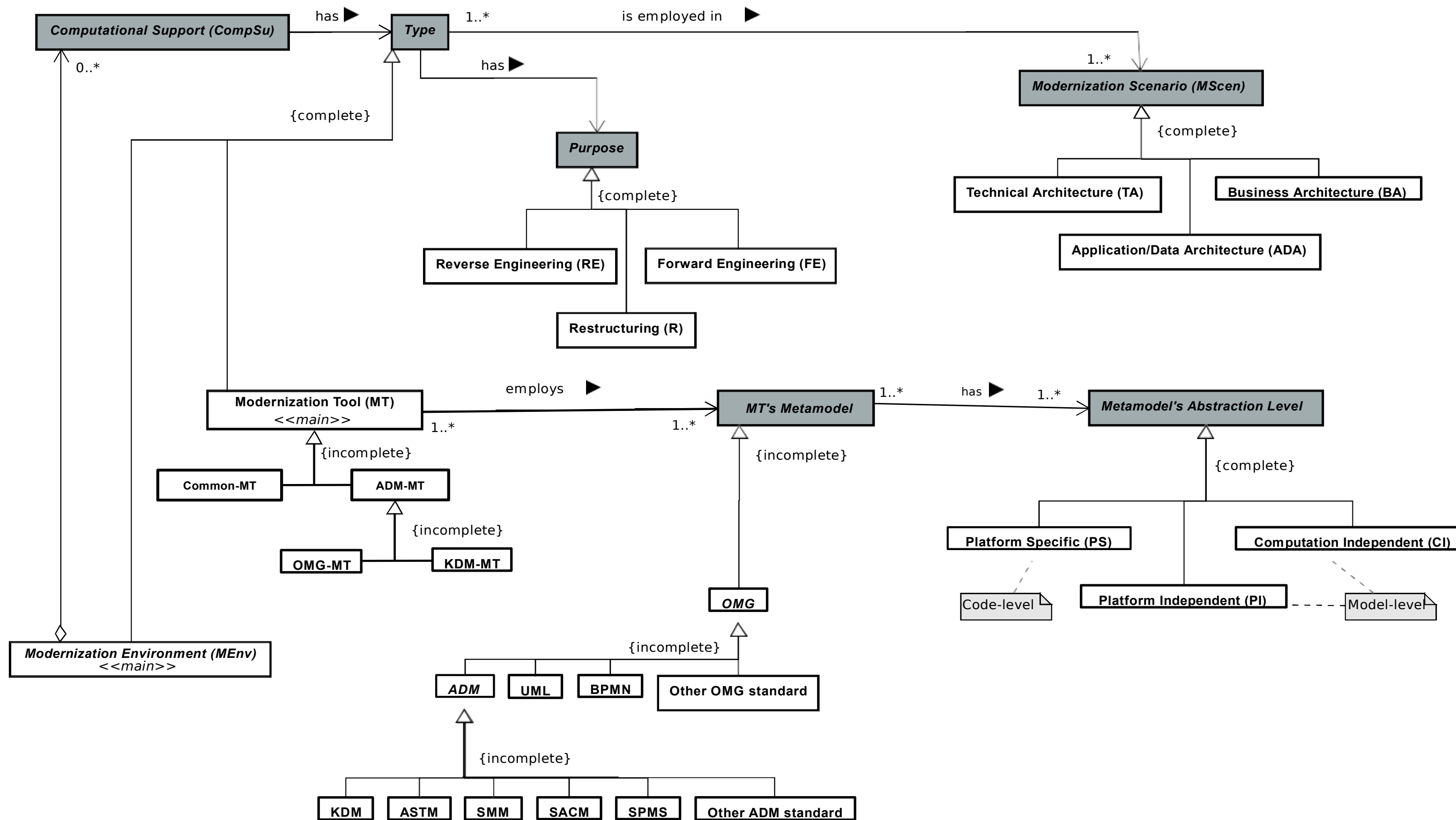

Figure 4.1: Modernization Tools Taxonomy - Conceptual Diagram

The classification label ***MT's Metamodel*** **(grey concept)** is used to classify MTs according to the metamodels used internally. There are two subgroups: ***OMG*** and ***ADM***. When a MT uses an OMG standard metamodel, it is possible to classify the tools as being **<*OMG*>-MTs**. For instance, if a tool has algorithms that operate with BPMN, the MT will be classified as BPMN-based MT, which means that this BPMN-based MT is prone to be interoperable with other BPMN-based MTs from other MEnvs.

Another possible classification for MT considers the group of standard metamodels proposed by ADM, which are the **ADM-based MTs**. These tools use at least one of the standard ADM metamodels. Thus, the classification of tools that use this kind of metamodel is **<*ADM-*> MT**. For instance, a KDM-based MT is a tool that uses the KDM metamodel in its internal mechanisms.

Another classification label is the **Metamodel's Abstraction Level** (grey concept). Knowing that MTs employ algorithms to process metamodel instances, it is possible to classify the tools according to the abstraction level in which the MTs operate. The three possible abstraction levels, according to MDA, are Platform Specific, Platform Independent, and Computation Independent. Classifying the Metamodel's Abstraction Level of a MT is directly linked to the metamodels that are being employed in the tool. Thus, it is possible to have more than one Metamodel's Abstraction Level in a single MT.

Since platform-specific models act at the source code level, we could also claim that these models are **code-level**. Now, considering the platform-specific and the computation-independent models, they act in higher model levels if compared to source code, where some fine-grained details can be omitted to provide a more abstract point of view of the same software system. Thus, we claim that they are **model-level**.

Table 4.2: Comparison between MT's type

MT	Uses Metamodel	Use ADM Standards	Uses OMG Standards	Uses KDM Standard
Common-MT	✓			
ADM-MT	✓	✓	✓	
OMG-MT	✓		✓	
KDM-MT	✓	✓	✓	✓

In Table 4.2, we have a comparison between the MT's types regarding the use of metamodels and in this table we can see the main difference in each type of modernization tool of our taxonomy. It is important to notice that in this comparison we are showing that a Common-MT is a MT that do not use any of the ADM/OMG standards but this tool can still be considered as part of an ADM modernization process, since the transformations can be applied in metamodel instances. This differentiation is important to identify the

level of interoperability that this tool could provide. Roughly speaking, we could state that a Common-MT would be a tool with lower chances of interoperability and a KDM-MT would be a tool with greater chances of interoperability when picturing about the complete modernization process.

Table 4.3: Concepts elicited from Step T-1

Concept	Name	Source (Derived from...)
C1	Reengineering	Set 1 and 2 of Step T-1
C2	Reverse Engineering	C1
C3	Restructuring	C1
C4	Forward Engineering	C1
C5	Model Driven Architecture	Set 1 and 2 of Step T-1
C6	PSM	C5
C7	PIM	C5
C8	CIM	C5
C9	Modernization Tools	Set 1 and 2 of Step T-1
C10	Modernization Scenarios	Set 2 of Step T-1
C11	Standard Metamodel	Set 1 and 2 of Step T-1
C12	Proprietary Metamodel	Set 2 of Step T-1
C13	Modernization Environment	C9 and C10
C14	Code-level	C5 and C6
C15	Model-level	C5, C7 and C8
C16	ADM-based metamodels	C11
C17	Computational Support	C9 and C13
C18	Technical Architecture	c10
C19	Application/Data Architecture	c10
C20	Business Architecture	c10
C21	Architecture-Driven Modernization	Set 1 of Step T-1

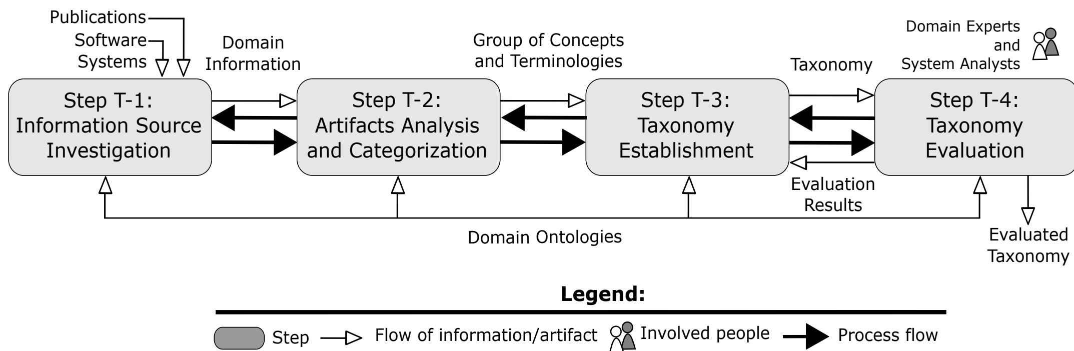

Figure 4.2: Methodology to establish a Taxonomy

4.3 Methodology for Building the Taxonomy

4.3.1 Step T-1: Information Source Investigation

The first step to establishing our taxonomy was to identify the sources of information to be used in the elicitation of concepts and terminologies. Different sources were considered, mainly related to the ADM domain. These sources can be classified into two sets: Set 1 - Formal specifications, glossaries, and white papers about ADM; Set 2 - Existing Modernization Tools identified in the literature.

In order to identify the concepts and terminologies about modernization tools available in the literature, we carried out a deep search on the OMG website[1] website, more specifically in the ADM part. In addition, a systematic mapping was considered to identify information sources (Durelli et al., 2014). Following, each set of information sources is described in detail:

- **Set 1 - Formal specifications, glossary and white papers about ADM:** An important source of information is the set of artifacts provided by OMG about ADM blueprint. We analyzed the ADM website, 4 white papers, 1 glossary, and formal specifications about software modernization[2]. As a result, we identified that since ADM involves the concepts of software reengineering and MDA[3], we should consider the terms: (i) Reverse Engineering; (ii) Restructuring; (iii) Forward Engineering; (iv) Platform Specific Model (PSM); (v) Platform Independent Model (PIM); and (vi) Computation Independent Model (CIM). We considered these terms as concepts associated with software modernization in the context of ADM to help us build out taxonomy.

- **Set 2 - Existing Modernization Tools identified in the literature:** In order to see how the concepts of ADM were being implemented in the real world, we analyzed existing modernization tools in the literature and industry. We analyzed the modernization tools presented in a systematic mapping on ADM (Durelli et al., 2014), and two books (Pérez-Castillo et al., 2011b; Ulrich e Newcomb, 2010b). The OMG/ADM

[1]Object Management Group (OMG) is dedicated to bringing together its international membership of end-users, vendors, government agencies, universities and research institutions to develop and revise standards as technologies change throughout the years. https://www.omg.org/

[2]This page provides a summary of OMG specifications that have either been formally published or are in the finalization process about software modernization: https://www.omg.org/spec/category/software-modernization

[3]Model Driven Architecture (MDA) is an approach to software design, development, and implementation spearheaded by the OMG that provides guidelines for structuring software specifications that are expressed as models.

makes available a list of vendors[4] that somehow employs the ADM concepts in its tools or that supported the OMG/ADM in its consolidation process. We analyzed these tools to understand their behavior and their role in the modernization process to support our taxonomy. As a result, we identified that modernization tools are directly related to the usage of the metamodel not only to representing legacy knowledge through reverse engineering but also metamodels that support the analysis of source codes and the other steps of software modernization advocated by ADM. Another important concept found by analyzing the books is the concept of Modernization Scenarios taken into consideration while composing our taxonomy.

4.3.2 Step T-2: Artifacts Analysis and Categorization

In this step, we analyzed all the information sources found in Step T-1, and as a result, we elicited the concepts presented in Table 4.3. These concepts represent the terms that are the basis of our taxonomy, but not all of them are explicit in the diagram that represents our taxonomy presented in Section 4.3.3. Model Driven Architecture, Standard Metamodel, Proprietary Metamodel, Code-level and Model-level are the concepts that are not explicit in the taxonomy, however they are incorporated in the other concepts. Model Driven Architecture, Code-level and Model-level are related to Metamodel's Abstraction Level. Standard Metamodel and Proprietary Metamodel are related to MT's Metamodel.

Based on the analysis performed in order to find the concepts in the previous step, we also elicited a set of constraints to help us in the taxonomy elaboration:

- **Constraint 1 - Purpose:** Modernization Tools should be categorized regarding the Reengineering (C1) step that it represents, i.e., its purpose that could be Reverse Engineering (C2), Restructuring (C3), or Forward Engineering (C4);
- **Constraint 2 - Abstraction Level:** Modernization tools should also be categorized regarding their abstraction level according to MDA (C5), which could be represented by PSM (C6) or Code-level (C14) and also PIM/CIM (C7 and C8) or Model-level (C15);
- **Constraint 3 - Modernization Concepts:** It should be specified in the taxonomy how the modernization concepts (Tool and Scenario) presented in Table 4.3 are related to each other;

[4]https://www.omg.org/adm/directory.htm

- **Constraint 4 - ADM standards**: While considering standard metamodels (C11), it should be implicit that there are specific metamodels that OMG designed to work with the ADM blueprint (C16).

While formulating and organizing these constraints, we identified the need for a concept that could involve modernization tools (C9) and modernization scenarios (C10). Thus, we created the **Modernization Environment** (C13) and **Computational Support** (C17) that encapsulate these two terms.

Note that several of these concepts are well known in software reengineering and MDA fields. The main reason to include them in our taxonomy is that they are related to software modernization and are also important concepts while characterizing modernization tools.

In general, the Step T-2 Artifacts Analysis and Categorization was important not only to identify the required concepts, but also to guide the study during the creation of our taxonomy, since we had to check if all constraints and concepts were being fulfilled, correctly represented, and described.

4.3.3 Step T-3: Taxonomy Establishment

In this step we gathered all concepts and constraints acquired in Steps T-2 in order to build the taxonomy of Modernization Tools that was presented in Section 4.2 - The Taxonomy.

4.3.4 Step T-4: Taxonomy Evaluation

In order to evaluate our taxonomy, we selected a set of papers presenting modernization tools, and then applied our taxonomy in order to classify the tools presented by those papers.

Our main goal in this initial evaluation was to check if some of the tools identified in chapter 3 could be categorized according to our RA taxonomy. The paper criteria selection was: 1 - the paper present a tool that uses KDM; 2 - the paper presents enough information about its architecture; 3 - tools that act in different modernization scenarios and purposes. About 1, we wanted to investigate KDM-based tools to check if out taxonomy could cover the different abstraction levels of KDM. About 2, in order to categorize correctly the tool we would need information about the main modules, input and outputs sources, internal behavior and main functionalities. About 3, to cover most of the possiblem categorization concepts we would need to get different modernization scenarios.

Table 4.4 presents nine (9) computational supports and their classifications according to our taxonomy. The first column contains the **Name** of the computational support,

and the second column has a short **Description** of each CompSu. The third, fourth and fifth columns classify computational supports according to their **Modernization Scenario**, **Type** and **Purpose**, respectively. Regarding **Modernization Scenario**, the valid values are: Business Architecture (BA), Application and Data Architecture (ADA) and Technical Architecture (TA). Regarding **Type** the valid values are: Modernization Tools (MT) and Modernization Envirionment (ME). Regarding **Purpose** the valid values are: Reverse Engineering (RE), Restructuring (R) and Forwards Engineering (FE).

The last three columns classify more internally the **Composee MTs**. The sixth column was added to provide more information about the **MT's Function** in order to facilitate the understanding of the classification provided in the seventh and eighth columns. The seventh column contains the classification of the tools that uses OMG standards: **MT's Metamodels**. Finally, the eighth column classifies the MT's Metamodels according to their abstraction level: **Metamodel's Abstraction Level**. The valid values of this field are: Platform-Specific (PS), Platform-Independent (PI) and Computation-Independent (CI).

According to Table 4.4, KDM-RE acts in the **application/data architecture** MScen as a **modernization environment** with the purpose of **restructuring**. KDM-RE is composed of three modernization tools that uses OMG standards, two **KDM-based** and one **UML-based**, which are **PI and CI**, and **PI**, respectively.

An important discussion here is on how to classify these CompSu as being MTs or MEnvs. To classify a CompSu it is necessary to understand how it works internally according to its metamodels and its functionalities. For instance, in Table 4.4 we have Arch-KDM (d. S. Landi et al., 2017), that works only with KDM metamodel and a DCL for KDM. The approach that Arch-KDM is inserted has several steps and functionalities such as Planned Architecture Specification, Current Architecture Extraction, and Architecture Comparison that return the found architectural drifts. Thus, we classify Arch-KDM as a MEnv since it involves more than one functionality.

The author of MoDisco (Brunelière et al., 2014) mentions this tool as being "a model-driven reverse engineering framework". This is due to the fact that MoDisco has several discoverers that act independently according to user usage. In this sense, each discoverer acts as an independent tool, but the set of discoverers/tools are encapsulated and interoperable in a MEnv that they call as being a model-driven framework. Thus, according to the proposed taxonomy, MoDisco is a **MEnv** that is composed of several **MTs**, and that has the **purpose** of **reverse engineering**.

Some MTs support completely a specific MEnv's purpose, such as CloudMIG and RUTE-K2J, and others that support partially, such as KDM-AO. Thus, we claim that CloudMIG and RUTE-K2J are MEnv composed of MTs and KDM-AO is a MT that can

Table 4.4: Classification of Computational Support for Software Modernization

Name	Description	Classification according to					
		MScen	Type	Purpose	Composee MTs		
					MT's Function	MT's Metamodel	Metamodel's Abstraction Level
KDM-RE	It implements the Fowler's refactoring catalog and allow modernization engineers to apply them in KDM instances.	ADA	MEnv	R	Refactoring Application	KDM-based	PI and CI
					Refactoring Propagator	KDM-based	PI and CI
					As-is visualize	UML-based	PI
RUTE-K2J	It takes a KDM instance as input and automatically generates a Java model from it.	ADA and BA	MT	FE	Transform KDM instances in Java model	KDM-based	PI and CI
MoDisco	It is able to retrieve information from legacy source code and databases and represent them as KDM and other metamodel instances.	TA and ADA	MEnv	RE	ASTM Discoverer	ASTM-based	PS
					UML Discoverer	UML-based	PI
					Metrics Application	SMM-based	PS, PI and CI
					KDM Discoverer	KDM-based	PI and CI
GAFEMO	It is a framework for the modernization of legacy systems using a model-driven and service-oriented approach using the features provided by gap-analysis techniques.	TA, ADA and BA	MEnv	RE, R and FE	Acquisition of logical model	ASTM-based	PS
					Refinement of logical model	KDM-based	PI and CI
					Refinement of business model	SBVR-based	CI
CloudMIG	It supports a semi-automatic legacy systems migration to cloud	TA and ADA	MEnv	RE	Metrics Calculation	SMM-based	PS, PI and CI
					Recovers KDM instances	KDM-based	PI and CI
MARBLE	It retrieves the business processes from existing systems using a set of model transformations.	TA, ADA and BA	MEnv	RE	Recovers KDM instances	KDM-based	PI and CI
					Recovers BPMN intances	BPMN-based	CI
CCKDM	It identifes crosscutting concerns in KDM instances. To do so the tool uses a combination of a concern library and a modified clustering algorithm.	ADA	MT	RE	Analyzes and marks KDM instances	KDM-based	PI and CI
KDM-AO	It implements a light and a heavyweight extensions that enables the instantiation of aspect-oriented concepts in KDM instances.	ADA	MT	RE	Enables the aspect oriented concepts instantiation	KDM-based	PI and CI
Arch-KDM	It helps in the conduction of Architecture-Conformance Checking (ACC) employing exclusively the KDM.	ADA	MEnv	RE	Planned Architecture Specification	KDM-based	PI and CI
					Current Architecture Extraction	KDM-based	PI and CI
					Architecture Comparison	KDM-based	PI and CI

be a composee of a MEnv, since only by itself, it has no power to perform completely at least one of the MEnv's purposes, such as: Reverse Engineering, Restructuring or Forward Engineering.

On the other hand, it is completely possible to have a MEnv composed of MTs and other MEnvs for different purposes. For instance, we could have a MEnv composed of

MoDisco, KDM-RE, and RUTE-K2J that would fulfill a complete modernization process with all three CompSu purposes.

4.4 Threats to validity

In this section we present the threat to validity of our preliminary taxonomy evaluation. The first point is that the selection and classification was performed by the author of this thesis and this could lead to a doubt about the categorization results that was made. We claim that even with all the information present in the taxonomy, these concepts could lead to some doubts in the categorization process. By providing this set of tools already categorized by the taxonomy´s creator we could support a second step of the evaluation process that will be performed by software engineers and modernization engineers that could use this initial evaluation as a reference.

4.5 Final Considerations

We claim that the general understanding of Modernization Tools is the key concept when talking about modernization with ADM, once all the standards that they advocate ground one of its main advantage, which is the interoperability. In this chapter, we presented a discussion about MT that aimed to improve the general comprehension and the main features that permeate MT.

In chapter 5, we present a reference architecture for software modernization tools that are based on ADM, in which we use the concepts presented in Chapter 4.

Chapter 5

A Reference Architecture for ADM-Based Modernization Tools

5.1 Initial Considerations

In this chapter we present RADM, which is the Reference Architecture we have developed for supporting the designing of ADM-based Modernization Tools. RADM is composed of 21 views, divided into Structural View, Data-Flow View and Dynamic Views. The primary purpose of these views is to serve as a basis for the structure of specific modernization tools. The main benefit of them is to show the main abstraction that must exist in these tools.

RADM is also composed of a set of activity and component diagrams that provide the inside dynamics of the subcomponents presented in the Structural and Data-Flow views.

5.2 The Reference Architecture

One of the major contribution of our reference architecture is to cover two vertents. The first one is to provide for software architects the main *abstractions* that must exist in ADM-based modernization tools. By identifying these abstractions, software architects can

design an architecture creating concrete modules that represent these abstractions. When these abstractions become evident in the source code, the evolution and maintenance will be facilitated. We also provide the relationships we believe are the most suitable among these abstractions, which can materialize themselves in concrete relations in the source code.

The another vertent is regarding the dynamics of the internal components. Our reference architecture also delivers activity diagrams that details the execution steps of the main components. Allowing software architects to understand the main steps of how their modernization tools should work.

In this section we provide an overview of the Reference Architecture we have created. We have organized this section in three main views:

- Structural View: MVC and Internal Components View;
- Data Flow Views: Pipes and Filters View; and
- Dynamic Views: Activities and Components Views.

The architectural views and its descriptions are presented here to provide a documentation of how the RA could be used. In the following sections we present a conceptual overview of the reference architecture that represents the complete Modernization Tool Architecture.

The Pipes and Filters view supports modernization engineers in the design process of modernization tools by enabling to understand the context, the main processes and input and output sources of the tool that is going to be developed. The MVC view provides an overview of the modernization environment. This view supports modernization engineers by providing two main sides, the user and the application, and also provides the main layers and how the communication could be established between the layers. And the Internal Components view that presents the components of the layers in the modernization application side. In this view we present the main components/functionalities of each modernization process and it is useful when the modernization engineer wants to design the functionalities of a modernization tool.

For the Dynamic views we will find a specific view for each one of the main components of our RADM. Here we will find the Reverse Engineering, Restructuring and Forward Engineering Views composed by their components, layers and repositories.

In order to represent these views we choose our own notation, that were based on the architectures found on the literature review, since it provide us a more free way of drawing

architectural views. However, the elements presented are based on KDM (Pérez-Castillo et al., 2011a).

5.3 Structural Views

This set of view aims to represent the structure of our RA by presenting a general view of modernization tools architecture. The two views of this set are:

- The Abstract Layered view (Figure 5.1) provides an overview of the modernization environment. This view supports modernization engineers by providing two main sides, the user and the application, and also provides the main layers and how the communication could be implemented between the layers of the two sides.
- The Concrete Layered view (Figure 5.2) presents the components of the layers in the modernization application side. In this view we present the main components/functionalities of each modernization process and it is useful when the modernization engineer wants to design the functionalities of a modernization tool.

5.3.1 MVC View

This view represents (Figure 5.1) how the layers communicate internally and how they should be designed. First of all, this view follows the Model-View-Controller (MVC) architectural pattern this is due to most of the existing modernization tools found in the literature followed a layered architecture and since MVC is a well-known and structured architectural pattern we claim that this is the one that should be considered while instantiating modernization tools. MVC framework supports an asynchronous technique that allows the execution of modernization tools to load very quickly models and to execute other components. In addition, MVC allows that modification in one specific component/layer does not affect the entire model.

In this view we considered two sides, the user side and the tool side. The user side is the environment that the modernization engineer are going to access the functionalities of the modernization tool that could either by a web browser or an Integrated Development Environment (IDE). The modernization application side is responsible for processing the requests of the user side. In this side, the front and back-end implementation should be stored. The functionalities should be grouped according to its purpose (Reverse Engineering, Restructuring or Forward Engineering) and it should be implemented as a component. The persistence layer should be separated from the model so each layer

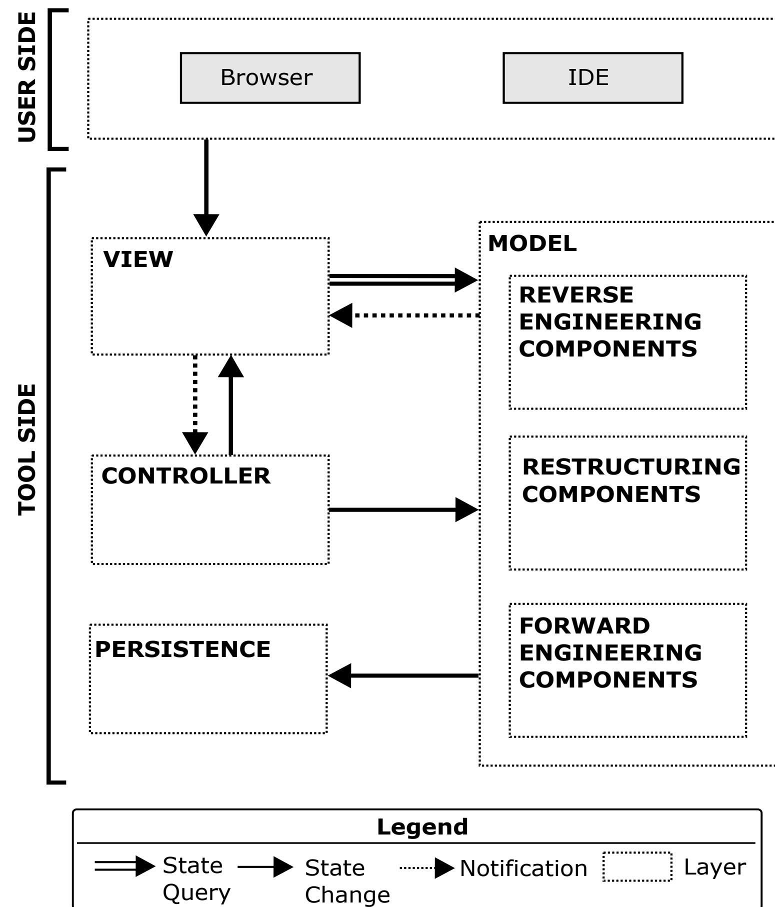

Figure 5.1: MVC view

could be responsible for implementing only its concern. The view layer is responsible for presenting the functionalities to the user and the controller layer is responsible for manage the requests from view layer.

In the legend of Figure 5.1 we present three notations to represent the relationship between layers. State Query change represents an action of a component in a layer that changes the current stage of another component. The State query represents a request to retrieve the current status of a component. Finally, the Notification represents a message exchange between layers and components that do not change the current state of a component.

Related requirements: G-4; G-8; and G-9[1].

[1]Please notice that these requirements are presented in Table 5.1

5.3.2 Internal Components View

A concrete view of modernization application side based on Figure 5.1, can be seen on Figure 5.2. In this view it is possible to see the components of each layer and how they communicate with each other.

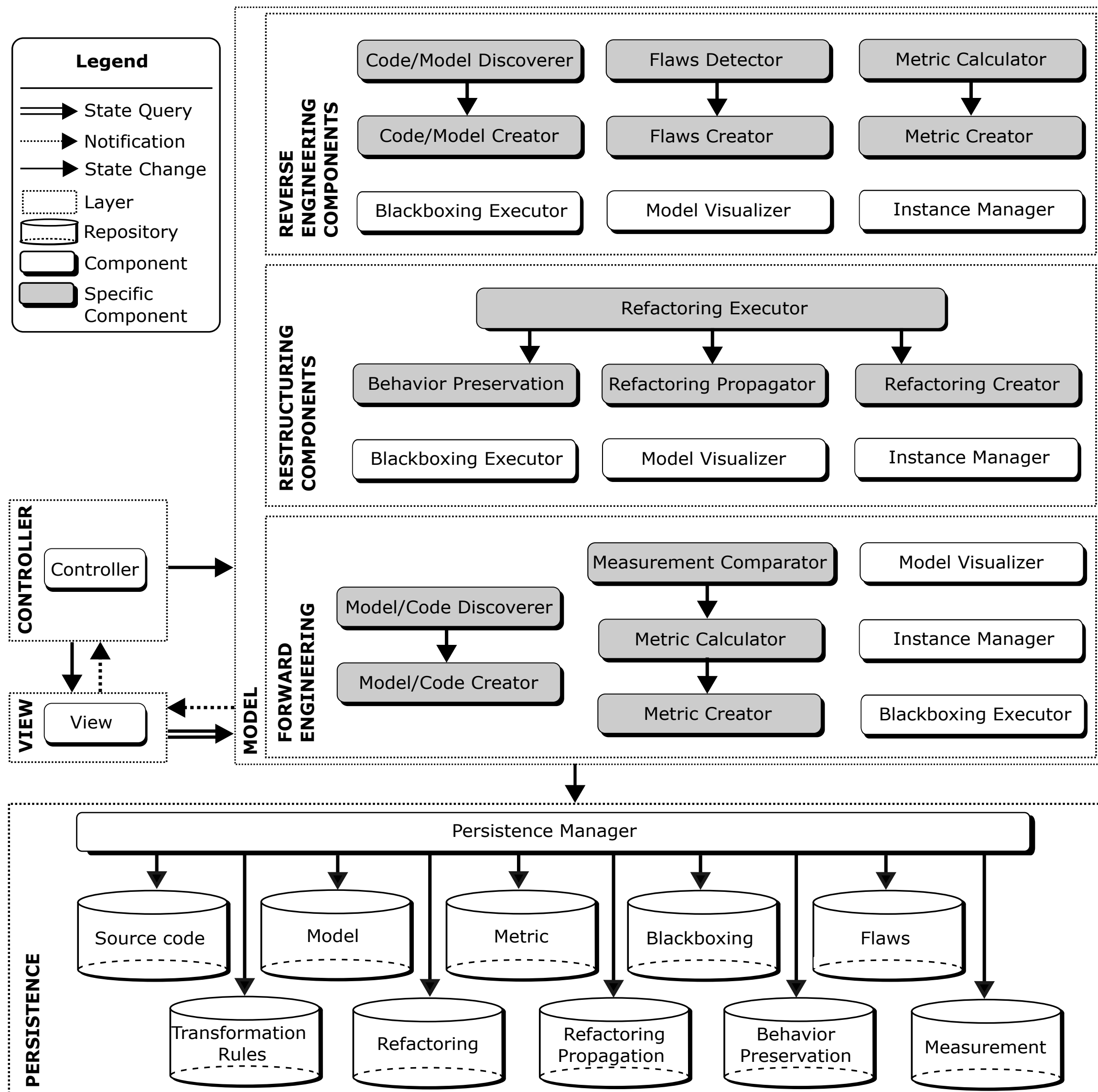

Figure 5.2: Internal Components view

Following we present the description of each layer.

- **View Layer:** Responsible for providing the user interface where a software engineer is going to perform one or more steps of the modernization process.

- **Controller Layer:** Responsible for processing the requests from the view layer and to redirect to the correct component in the model layer.

- **Model Layer:** It contains all the layers of the modernization tool that is organized in three main layers: Reverse Engineering; Restructuring; and Forward Engineering.

- **Reverse Engineering Components Layer:** It contains all components that support the Reverse Engineering process. Inside the Reverse Engineering layer there are the components Instance Manager(G-7 and RE-2), Flaws Detector (RE-3) and the Code/Model Discoverer that has components responsible for discover knowledge from source code (RE-1). The other components are the Blackboxing Executor (G-11), Metric Calculator (RE-4), Metric Creator (RE-5) and Model Visualizer (G-1)[2]. Please, notice that the gray components are specific components, meaning that they should exist only inside that particular layer. In the other hand, white components can be implemented in any of the three possible layers (Reverse, Restructuring and Forward).

- **Restructuring Components Layer:** It contains all components that support the Restructuring process. The component Refactoring Executor (R-1, R3 and R-4) is the main component that is responsible for the refactorings in model instances. The components Behavior preservation (R-5) and Refactoring Propagator (R-5) support the modernization process by improving the results of the performed refactorings. The Refactoring Creator (R-2) component allows the creation of new refactorings that will be applied in the model intances.

- **Forward Engineering Components Layer:** It contains all components that support the Forward Engineering process. The Metric Calculator (FE-3), Measurement Comparator (FE-4) and Metric Calculator supports the analysis of software modernization. The Model/Code Discoverer is responsible to convert model instances in to modernized source code (FE-1, FE-2 and G7).

- **Persistence Layer:** This layer can only be accessed by the model layer and it is responsible for managing all the persistence that the components may require. The repositories should be organized as follows: i)Model - stores all model instances that were obtained during the execution of the components of the different Components Layer; ii) Metric - stores all the metrics and model measurement results; iii) source code - store the legacy source code and the target source code; iv) Blackboxing -

[2]The information in parentheses ( ) represent the requirement described in Table 5.1

stores the blackboxing algorithms that are going to be used in the modernization tool implementation; v) Refactoring - stores the model transformations needed for performing a refactoring; vi) Flaws - stores the algorithms to detect the flaws and the model instances annotated with the detected flaws; vii) Transformation Rules - stores all transformation rules that are used by the components; viii) Behavior Preservation - stores the algorithms and transformation rules to apply behavior preservation; and ix) Measurement - stores the measurements that were discovered in the modernization process.

Related requirements: G-1; G-3; G-4; G-8; G-8.1; G-9; G-11; RE-1; RE-2; RE-3; RE-4; RE-5; R-1; R-2; R-3; R-4; R-5; FE-1; FE-2; FE-3; and FE-4.

5.4 Data Flow View

This view aims to represent the data flow of our RA by presenting the execution simulation of a modernization process.

- The Pipes and Filters view (Figure 5.3) supports modernization engineers in the designing process of modernization tools by enabling to understand the context, the main processes and input and output sources of the tool that is going to be developed.

5.4.1 Pipes and Filter View

First of all, Architecture-Driven Modernization blueprint provide us a horseshoe view that enable to see how the different modernization processes (Reverse Engineering, Restructuring and Forward Engineering) interact with each other, and pipes and filter architectural style could represent the ADM's horseshoe blueprint. This architectural view can be seen in Figure 5.3. Each filter can act independently and they do not need to know or inform status to another filter. This view enable to understand the architectural data flow by showing that each filter has a purpose and specific input and output formats. The main reason we choose pipes and filters to represent this data flow is that this architecture pattern allows the processing modules of a MT to be break down in to a set of independents steps, representing each step of the software reengineering. In addition, it also provides the flexibility to reorder the processing steps of the MT that allows the modernization engineer to add and remote processing steps inside each one of the filters.

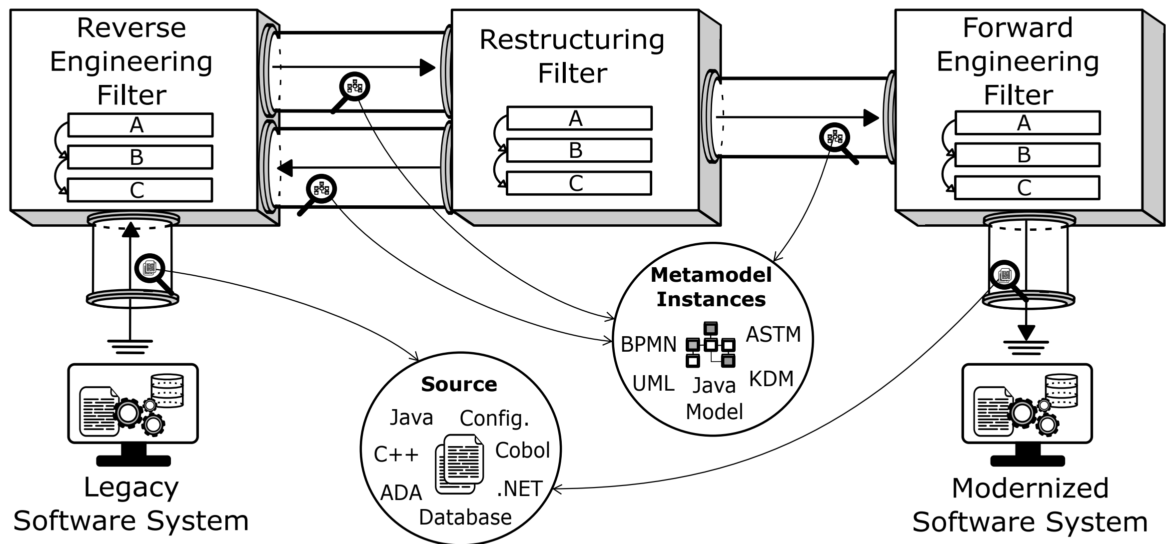

Figure 5.3: ADM Pipes and Filters Architectural Overview

Input files are provided to filter by means of pipes that accept specific types of files, for instance, the Reverse Engineering Filter has a purpose of abstracting knowledge from existing source files[3] in order to transforms this input in metamodel instances that are going to be consumed by another filter.

The Restructuring filter receive metamodel instances from Reverse Engineering filter to perform model transformations in order to improve and to fulfill the modernization goals. The restructuring filter output could serve as input to the reverse engineering filter or to the forward engineering filter. Usually, the reverse engineering filter consumes the output of the restructuring when another improvement should be performed in a different abstraction level[4]. When consumed by forward engineering filter the goal is to transform the modifications performed in restructuring filter in source code, completing the modernization process.

The Pipes and Filters view completes the MVC view since it is possible to see how each filter behave internally.

Related requirements: G-5; G-9 and G-10.

[3]Source files in modernization context could be represented by all files that composes a software system, such as: source code, configuration files, databases, and interfaces.

[4]There are three main abstraction level according to Model Driven Architecture: Platform Specific, Platform Independent and Computation Independent.

5.5 Dynamic Views

In this section we concentrate on the dynamic aspects of the Reverse Engineering, Restructuring; and Forward Engineering mentioned initially in Figure 5.2. In the following sections we present details of each modernization flow which are represented by a set of activity and a component diagram. Each activity diagram corresponds to a component in Figure 5.2.

5.5.1 Reverse Engineering Views

In Figure 5.4 is presented the Reverse Engineering view composed by its components, layers and repositories.

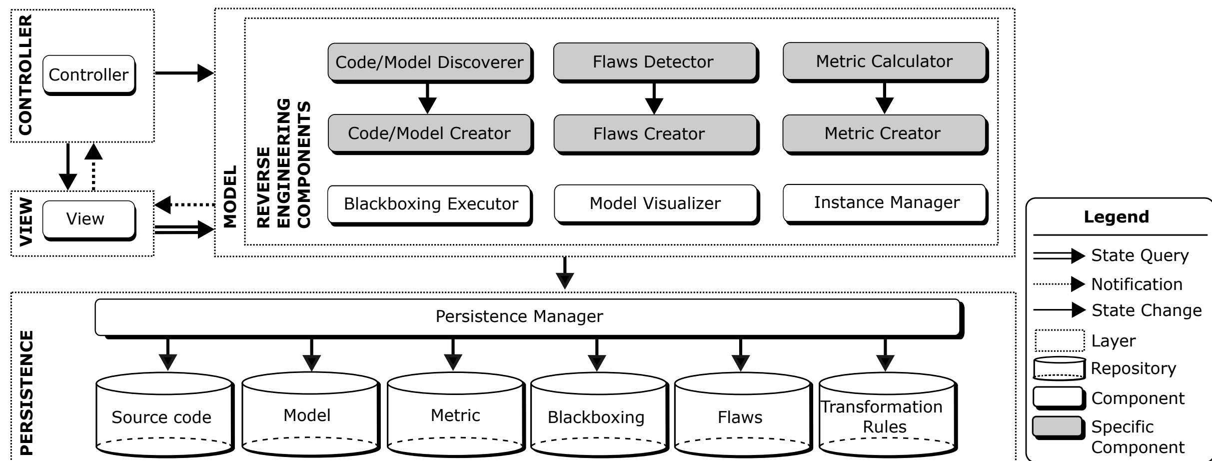

Figure 5.4: Reverse Engineering view

5.5.1.1 General Activity Diagram View

The first activity diagram is about the general flow of RE. As can be seen in Figure 5.5, the flow starts with the process of loading legacy source code in order to discover metamodel instances of it to finally store these information in the correspondent repository. The Discovery Instance activity can be executed as many times as needed until the desired abstraction level of the model instance is achieved. After the discovery instance activity the MT should store the discovered instance as a backup or to be reused by other algorithm. The modernization engineer can decide the better way of working with these instances and the load instance activity can be skipped if it is decided to use the current model that was recently discovered.

The RE process involves two other main functionalities that are flaws detection and metric calculation.

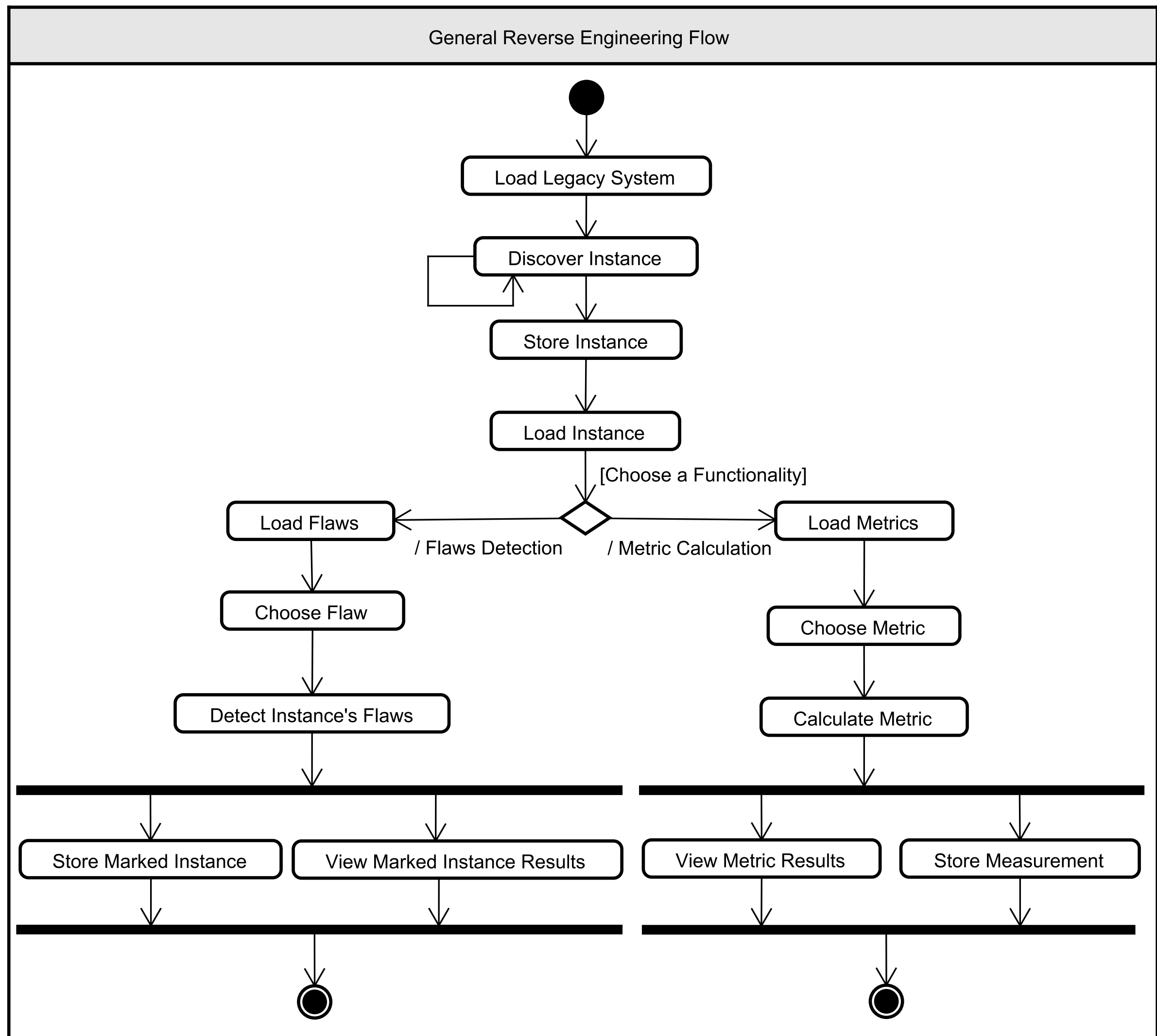

Figure 5.5: General Activity Diagram View

Flaws detection is the process of analysing an existing metamodel instance in order to detect bad code signs about specific concerns. To detect the flaws it is necessary to load the metamodel instance (Load Instance), load the existing flaws detector algorithms (Load Flaws), choose the one that better fits its purpose (Choose Flaw) and then apply it in the instance (Detect Instance's Flaws). After applying the flaw algorithm, the result should be stored (Store Marked Instance) and present to the software engineer the results (View Marked Instance Results).

Regarding metric calculation, it is the process of measure the software system considering a specific metric. To calculate a metric it is necessary to load the metamodel

instance (Load Instance), load the metrics available in the tool (Load Metric), select a metric (Choose Metric) and apply the metric in the metamodel instance (Calculate Metric). After applying the metric algorithm, the result should be stored (Store Measurement) and present to the software engineer the results (View Metric Results).

The followings activity diagrams present detailed information about the flow presented in Figure 5.5.

5.5.1.2 Instance Manager Activity Diagram View

This view presents the way that the instances and source codes should be handled in a RE modernization tool and can be seen in Figure 5.6.

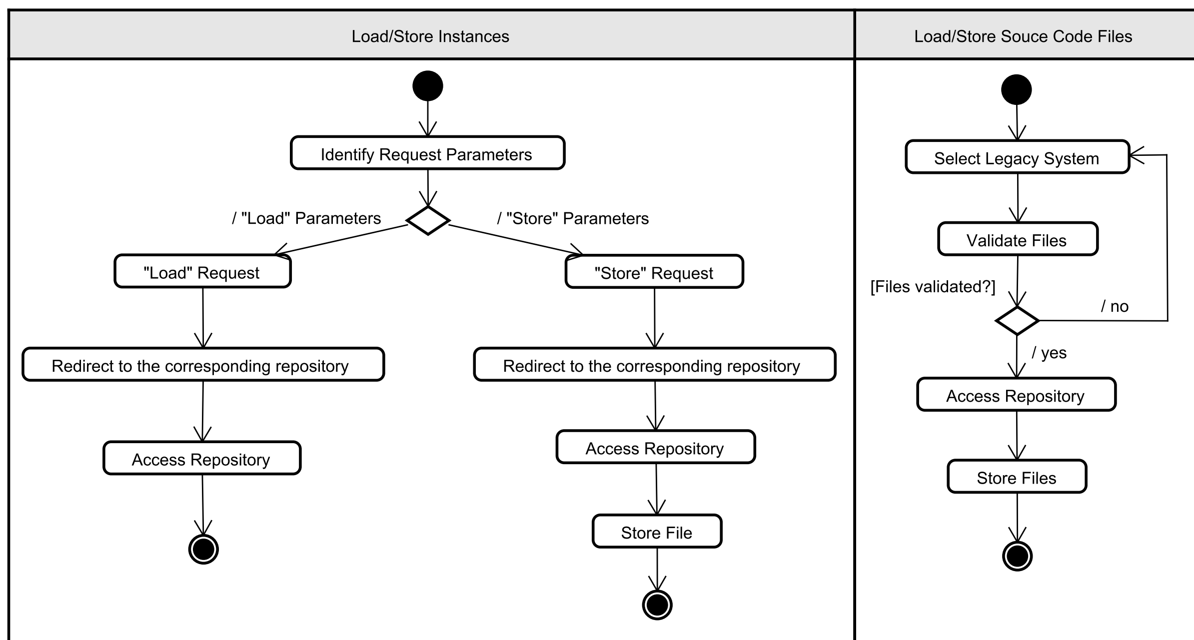

Figure 5.6: Instance Manager Activity Diagram View

Regarding metamodel instances there are two possible actions, load or store (Identify Request Parameters). The ”Load” request receives the instance parameters (name of the instance and corresponding repository) that will be loaded in order to redirect the request to the corresponding repository. The ”Store” request receives the name, the instance itself and the corresponding repository information to record in the corresponding repository.

Regarding the source code management, the first step is to import the files in the tool (Select Legacy System). The files should be validated in order to ensure that the user is importing the correct files (Validate Files). After the validation the source code repository should be accessed and the files should be stored.

5.5.1.3 Flaws Detector Activity Diagram View

In a RE modernization tool it is possible to detect flaws in metamodel instances. However, there are cases that the tools is responsible not only for the detecting process but also for creating the algorithms that are going to perform the flaws detection. We are considering that the flaws detected in metamodel instances are marked in the instance itself to be processed later on for restructuring tools.

The flow to detect flaws is as follows and can be seen in Figure 5.7. First, the instance that will be analysed should be loaded and then the existing flaws detector algorithms are loaded. The software engineer will decide which algorithm will be applied, however the needed flaw detector could not be available/exists then it could be created (explained in the following paragraph). If the wanted flaw algorithm is available, the flaw is selected (Choose Flaw) and then applied in the metamodel instance (Detect Instance's Flaws). The following steps are to store the marked instance and to view the marked instance results.

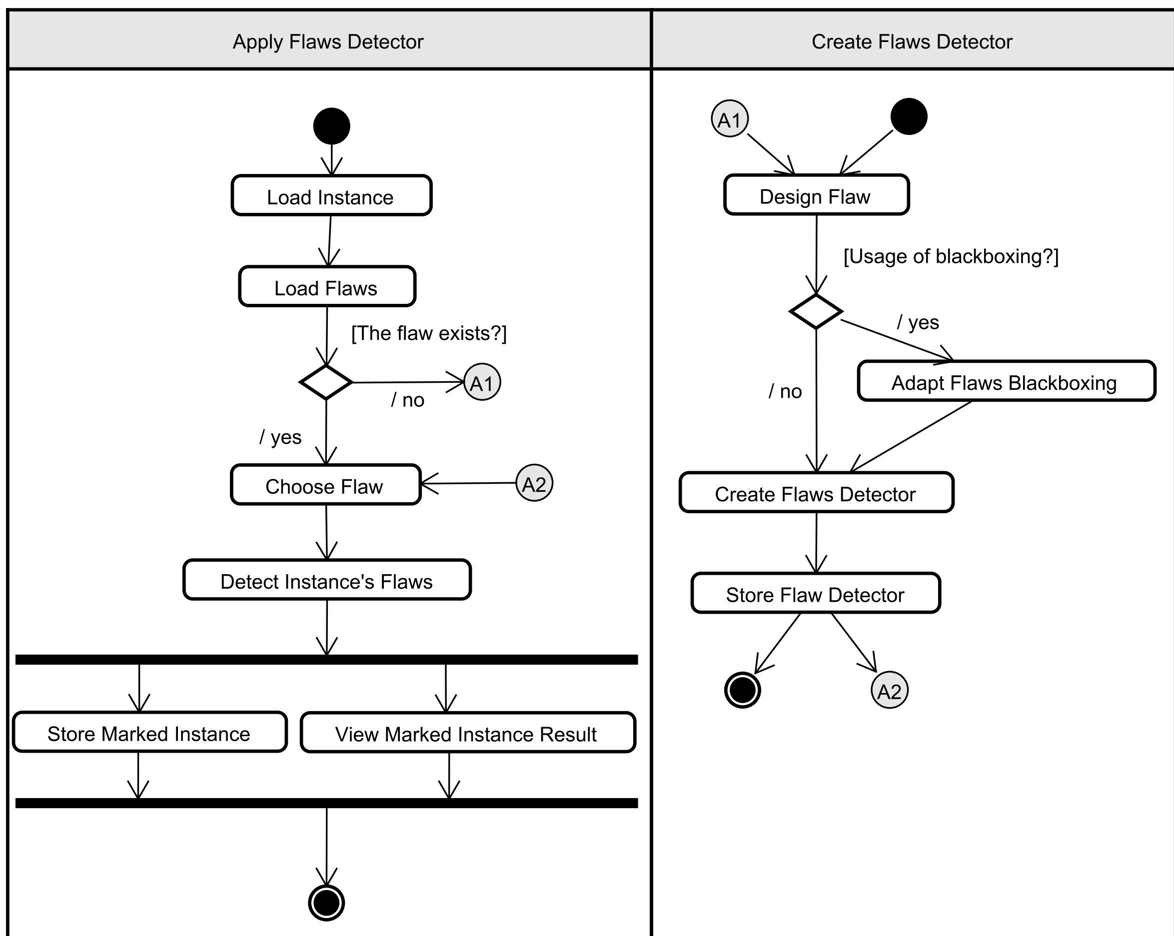

Figure 5.7: Flaws Detector Activity Diagram View

The flow to "create flaws" is as follows. The first step is to design what should be detected and how the algorithm to detect the flaw should behave. Sometimes it is possible to reuse an existing algorithm/tool to avoid rework, then the modernization engineer should design how this algorithm will be integrated in the tool. When the flaws detector algorithm is created, it should be stored in order to be available.

5.5.1.4 Metamodel Instance Discoverer View

The main functionality in the Reverse Engineering process is the metamodel instance discover process. The basic flow to create a discoverer tool can be started in two ways (Figure 5.8). The first one is by discovering an instance from source code and the second way is by discovering and instance from another instance. The discovery process for both are very similar an can be synthesized as follows. The fist step is to load source code/instance and the existing transformation rules. If the transformation rules needed do not exists it could be created, as presented in the following paragraph. After choosing the right transformation rules, they should be applied in order to get the needed metamodel instance. As metamodel instances are not meant to be processed by humans, the tool could provide ways to visualise according to a given context. The final step is to store the instance in its corresponding repository.

The flow to create a discoverer is as follows. The first step is to design how the algorithm to discover instances should behave and the metamodels that will be handled. Sometimes it is possible to reuse an existing algorithm/tool to avoid rework, then the modernization engineer should design how this algorithm will be integrated in the tool. The discovery process need two metamodels as input, one for the source instance (Select Source Metamodel) and another to get the target instance (Select Target Metamodel). A mapping should be developed in order to create the transformation rules. This step is responsible for specifying which element in the target metamodel corresponds to the source metamodel. This mapping enables the transformation rules development. The last step is to store the transformations rules to be used by the metamodel instance discoverer.

5.5.1.5 Metric Calculator Activity Diagram View

In a RE modernization tool it is possible to calculate metrics in metamodel instances. The flow to calculate metric is as follows (Figure 5.9). First, the instance that will be analysed should be loaded and then the existing metrics are loaded.

The software engineer will decide which algorithm will be applied, however the needed metric could not be available/exists then it could be created (explained in the following

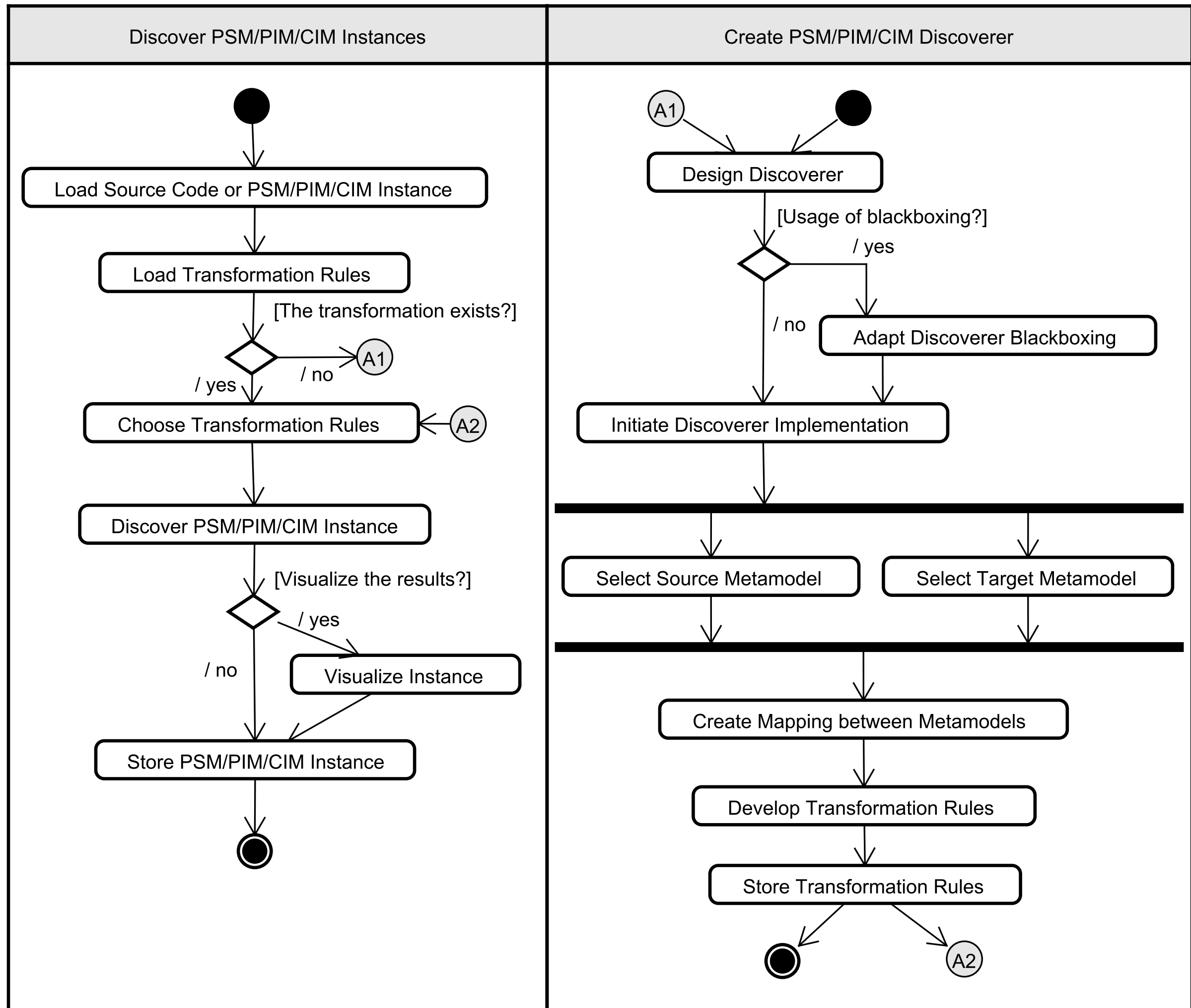

Figure 5.8: Metamodel Instance Discoverer View

paragraph). If the wanted metric is available, the metric is then selected (Choose Metric) and then applied in the metamodel instance (Apply Metric). The following steps are to store the measurement and to view the measurement results.

The flow to create a metric algorithm is as follows. The first step is to design the metric and how the algorithm to calculate the metric should behave. Sometimes it is possible to reuse an existing algorithm/tool to avoid rework, then the modernization engineer should design how this algorithm will be integrated in the tool. When the metric algorithm is created, it should be stored in order to be available.

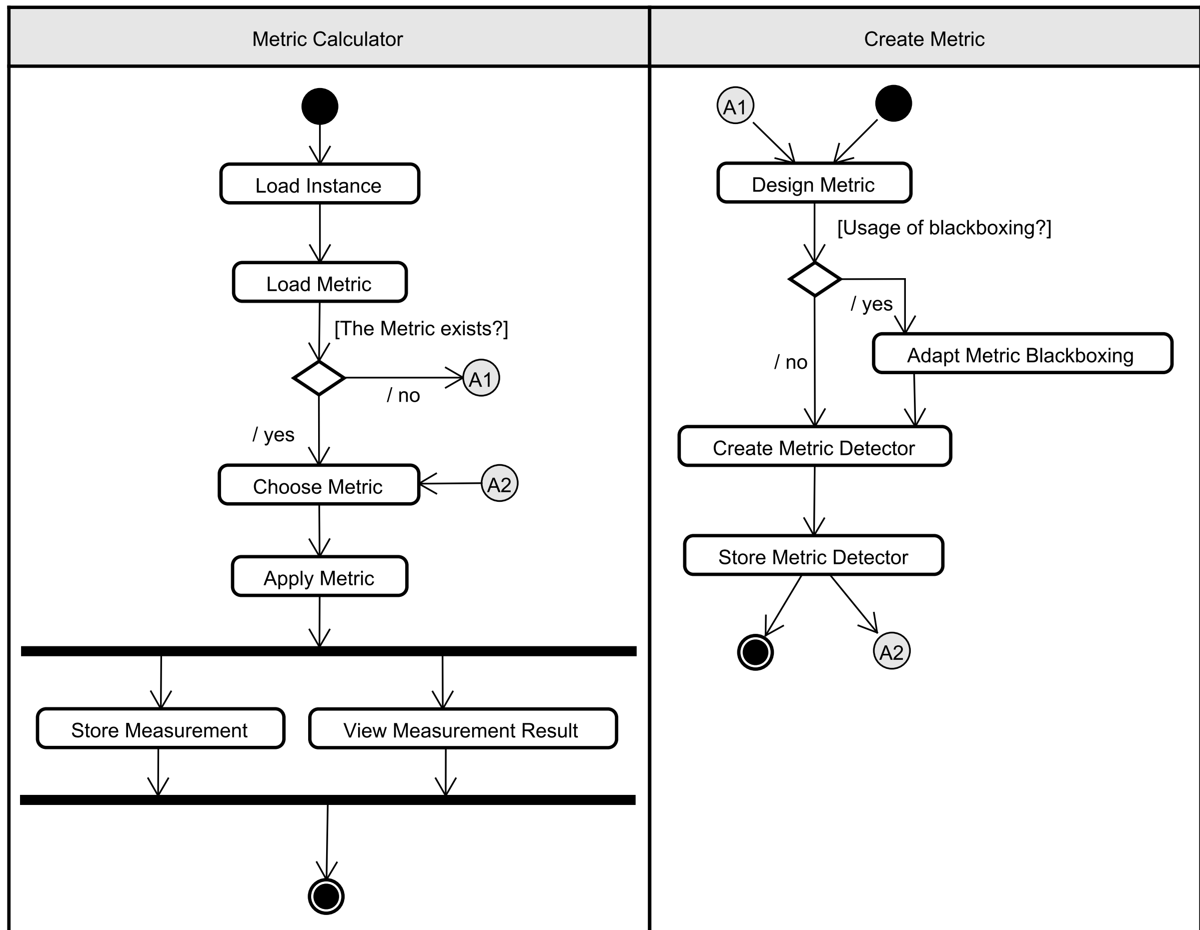

Figure 5.9: Metric Calculator Activity Diagram View

5.5.1.6 Reverse Engineering Component Diagram View

In Figure 5.10 we present the component diagram view of the reverse engineering view. The creation process of this diagram took in consideration the SOLID principles[5] (Martin et al., 2003). For instance, we covered the Single-responsibility principle by creating one component for each functionality so each one of them could have only one responsibility. we covered the Liskov substitution principle by creating one interface for each main component (Flaws Detector, Code/Model Discovery and Metric Calculator) so the controller component could use the reverse engineering components without knowing the inside content of each component.

The main point of this diagram is to present the idea of how the source code should be structured and how the components should communicate. In the figure we could

[5]Proposed by Robert C. Martin, SOLID is a set of principles that establishes best practices for developing software systems aiming to contribute by avoiding code smells, successfully refactoring code, and other advantages.

see that components should communicate though interfaces and not directly access each other. Another point is that components should not access the repository directly to standardize the way of accessing the repositories. By sending all the store requests to a Instance manager component we simplify the way we call this functionality inside the main components.

Regarding the reverse engineering component diagram in Figure 5.10 we display three main components: Flaws Detector, Code/Model Discovery, and Metric Calculator. Each one of these components has its own concern, i. e., should be self contained in the component.

Flaws Detector component clusters the functionalities that are related to the process of analysing/detecting flaws in model instances. The Code/Model Discovery is responsible for clustering the components that aim to support the conversion of source code or models for the reverse engineering. The Metric Calculator component is responsible for clustering all components that support the analysis of the software systems metrics data.

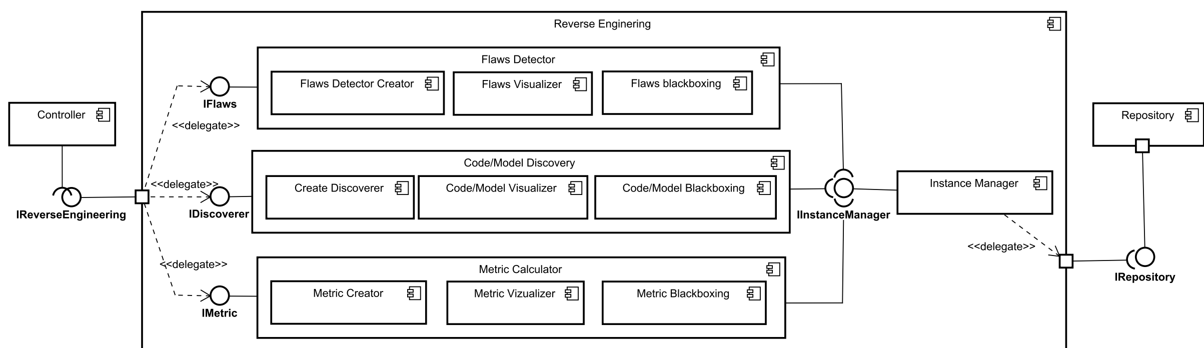

Figure 5.10: Reverse Engineering Component Diagram View

5.5.2 Restructuring Views

The diagrams set presented in this section is related to the restructuring step of the modernization flow. In Figure 5.11 is presented the Restructuring view composed by its components, layers and repositories.

5.5.2.1 General Activity Diagram View

In Figure 5.12 is represented the general flow of the restructuring process. The flow starts at loading the metamodel instance in order to execute the existing refactorings. One important validation after executing refactorings is to ensure that the behavior of the

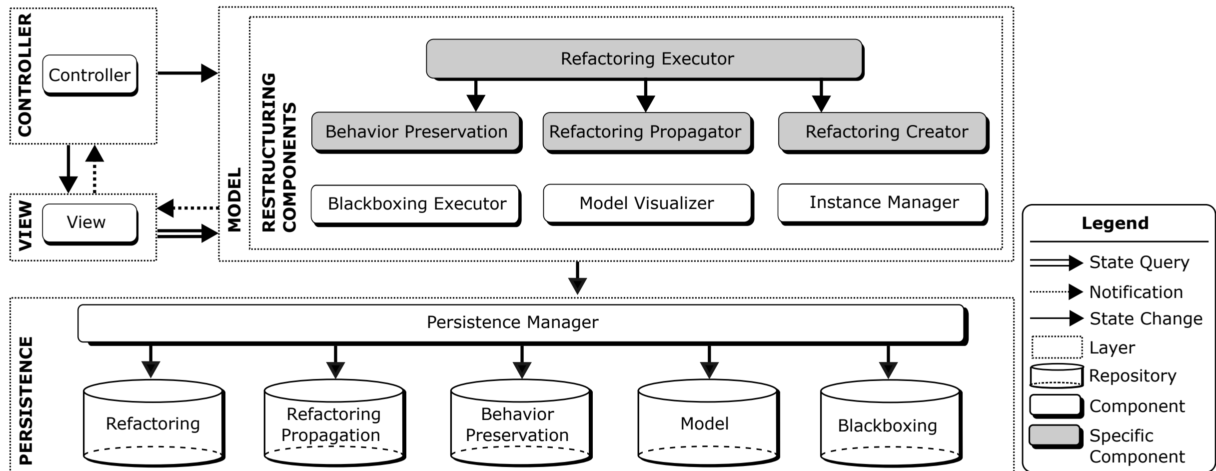

Figure 5.11: Restructuring view

refactored system is preserved, thus the modernization engineer has to execute a behavioral preservation algorithm.

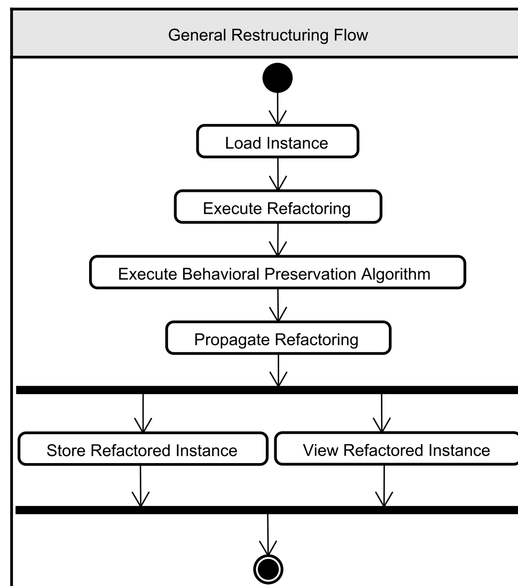

Figure 5.12: General Activity Diagram View

When the software system represented as a metamodel instance has more than one abstraction level, there might be needed a propagation of the changes to the others levels. After the propagation the tool should store the refactored instance and display the refactoring results to the user.

5.5.2.2 Instance Manager Activity Diagram View

This view presents the way that the instances should be handled in a Restructuring modernization tool. Regarding metamodel instances (Figure 5.13) there are two possible actions, load or store (Identify Request Parameters).

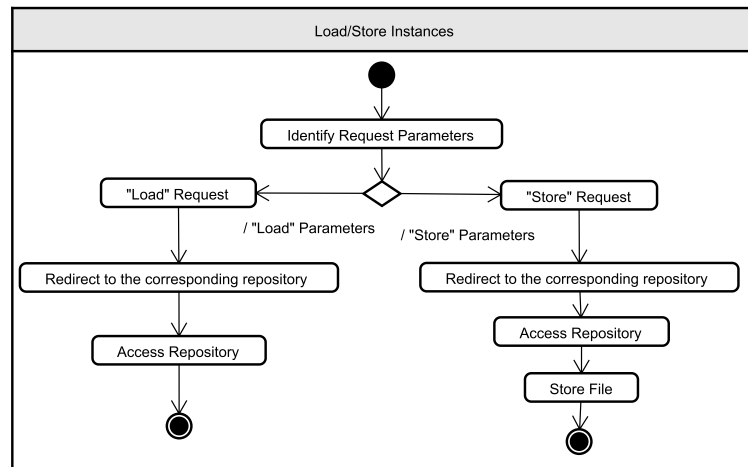

Figure 5.13: Instance Manager Activity Diagram View

The ”Load” request receives the instance parameters (name of the instance and corresponding repository) that will be loaded in order to redirect the request to the corresponding repository. The ”Store” request receives the name, the instance itself and the corresponding repository information to record in the corresponding repository.

5.5.2.3 Refactoring Activity Diagram View

In Figure 5.14 is displayed the complete restructuring flow of restructuring process. This view is composed of four flows: refactoring executor, refactoring creator, behavioral preservation and refactoring propagator creator.

The flow starts at loading the instance that is going to be refactored. The next step is to load the existing refactoring that could be applied in the software system instance. In this step, the tool could allow the implementation of a new refactoring. So, the flow named as Refactoring Creator (A1) represents the steps needed to create a new refactoring. After choosing the refactoring, the algorithm should apply it in the instance. The flow triggers two other flows, one allowing the user to see the refactorings results and the other one executing the behavioral preservation of the instance to ensure that the refactoring is not changing any implementation logic of the software system.

Regarding the behavioral preservation, each refactoring should have its behavioral preservation algorithm, since it is expected when dealing with refactorings. In this step, the

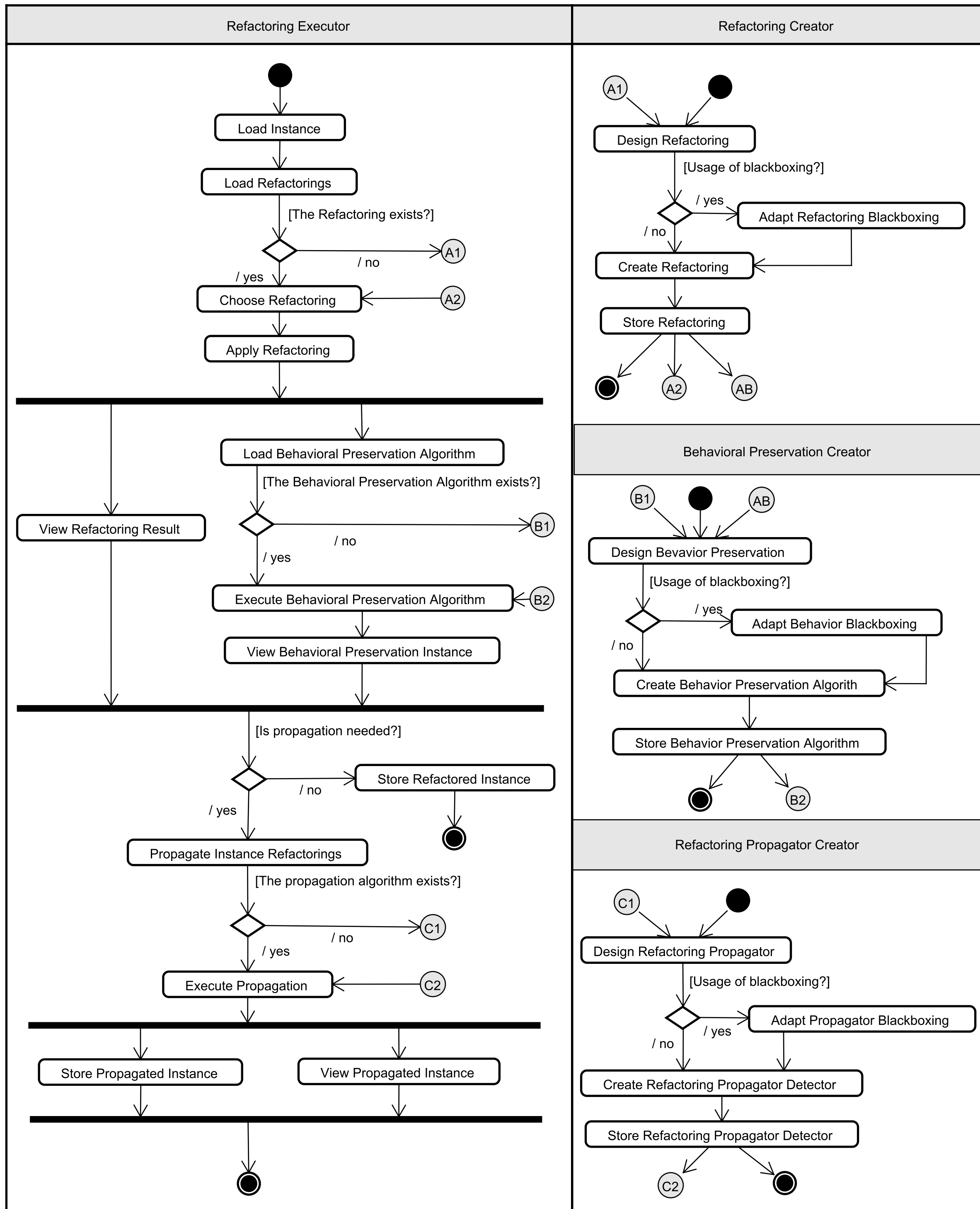

Figure 5.14: Refactoring Activity Diagram View

tool could allow the implementation of a new behavioral preservation. This new behavior preservation could be represented by means of transformation rules that would be applied in the refactored model instance to validate some specific behavior the modernization

engineer wants to preserve. So, the flow named as Behavioral Preservation Creator (B1) represents the steps needed to create a new refactoring. After choosing the corresponding behavioral preservation, the algorithm should apply it in the instance and in the end it should display the results.

There are cases that the instance contains more than one architectural view of the same software system, in that case, the changes performed by the refactoring algorithms should be propagated to the other architectural views in order to keep the instance consistently updated. To do so, the tool should support the propagation mechanism that is triggered by the propagation instance refactoring. In this step, the tool could allow the implementation of a new Refactoring Propagator. So, the flow named as Refactoring Propagator Creator (C1) represents the steps needed to create a new refactoring propagator. After choosing the corresponding refactoring propagator, the algorithm should apply it in the instance and in the end it should display the results and also store the the propagated instance.

Refactoring Creator (A1), **Behavioral Preservation Creator (B1)** and **Refactoring Propagator Creator (C1)** have the same implementation flow. First of all, the operating logic should be designed, which will evaluate if there is any existing algorithm/tool could be incorporated to then implement/create the complete operating logic. The final step of these flows is to store the service/tool/algorithm in a specific repository.

5.5.2.4 Restructuring Component Diagram View

In Figure 5.15 we show thee main components: Behavior Preservation, Refactoring, and Propagation. As mentioned before, each component should be self contained and also should be accessed only through its interface. Behavior Preservation component is responsible for clustering all components related to supporting the behavioral preservation logic. Refactoring component clusters all components related to applying refactoring to model instances, and Propagation component is related to all logic that aims to propagate the changes performed in the refactoring component.

5.5.3 Forward Engineering Views

In Figure 5.16 is presented the Forward Engineering view composed by its components, layers and repositories.

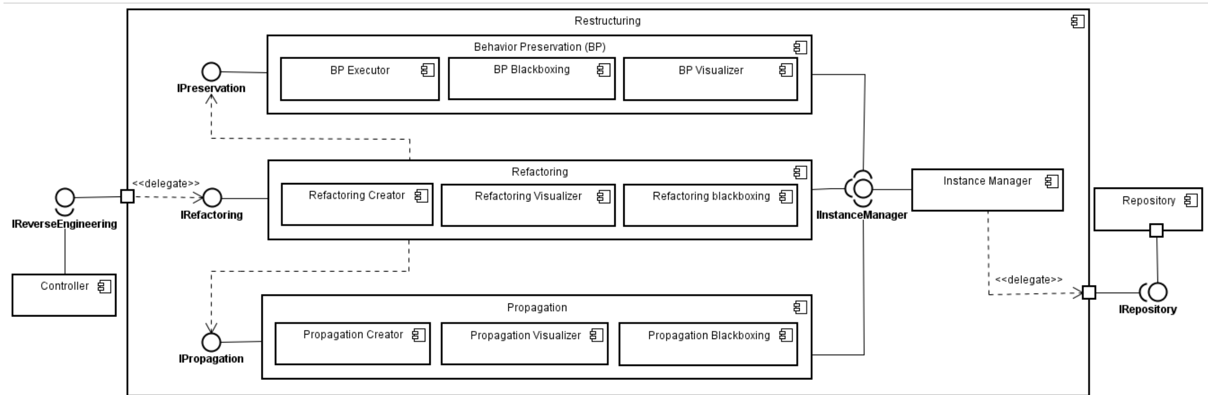

Figure 5.15: Restructuring Component Diagram View

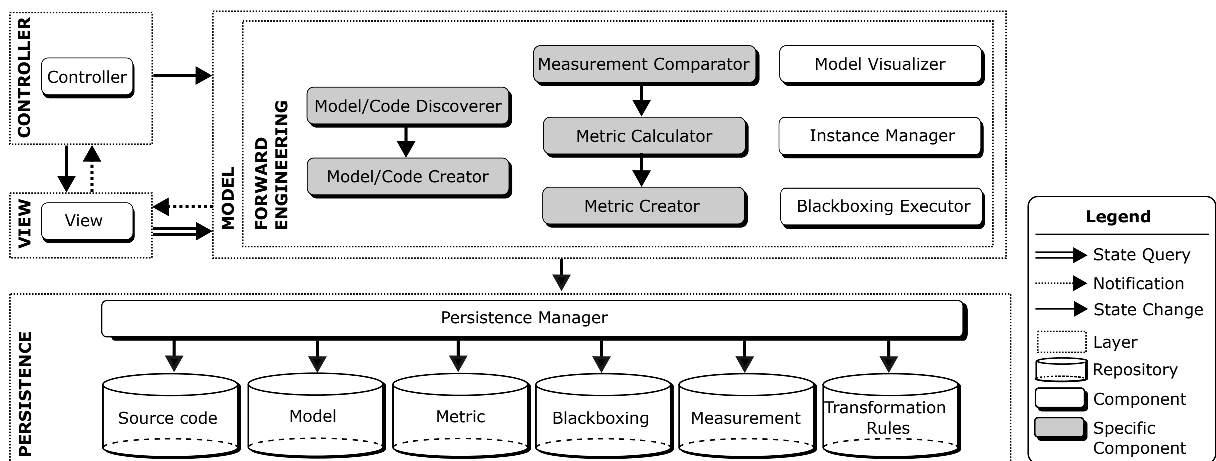

Figure 5.16: Forward Engineering View

5.5.3.1 General Activity Diagram View

In Figure 5.17 is presented the general flow of Forward Engineering process. The flow starts at loading the metamodel instance in which two main functionalities can be performed: Metric Calculation and instance/source code discovery.

Regarding metric calculation, it is the process of measure the software system considering a specific metric. To calculate a metric it is necessary to load the metrics available in the tool, select a metric (Choose Metric) and apply the metric in the metamodel instance (Calculate Metric). After applying the metric algorithm, the result should be stored (Store Measurement) and present to the software engineer the results (View Metric Results).

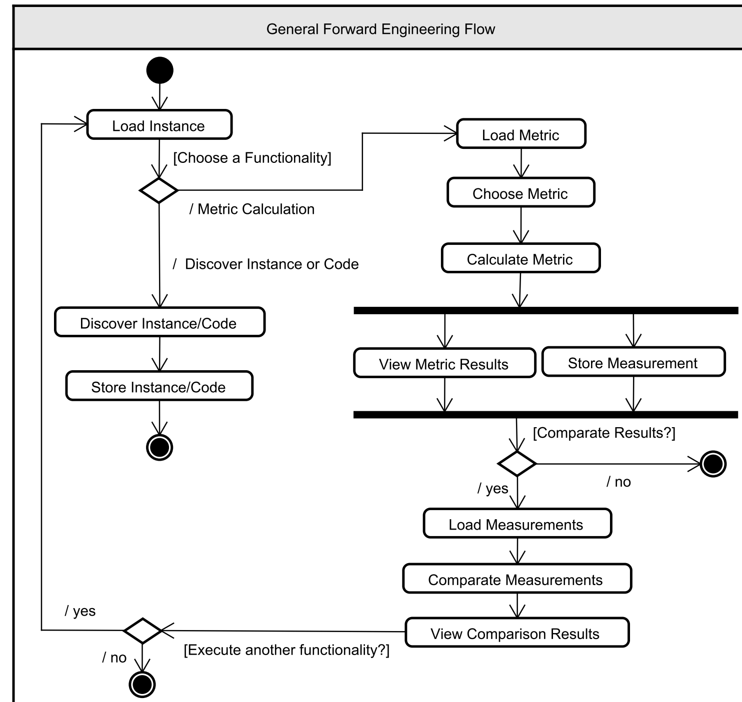

Figure 5.17: General Activity Diagram View

As Forward Engineering flow is the last one in the reengineering process, some metric calculation could have previously being performed in reverse engineering flow. Thus, as another functionality of FE flow, the software engineer could also perform a measurement comparison in order to extract some information about the refactoring that were performed. For this, the tool should load the measurements, execute the algorithm to compare the two metric results and present the comparison results.

After the measurement comparison, the tool should display the option of calculate another metric or perform instance/source code discovery. The instance/source code discovery consists in the process of analyse existing instances in order to discover lower level instances until it reaches the source code level. In the end of this flow the instances/source code discovered should be stores in its corresponding repository.

The followings activity diagrams present detailed information about the flow presented in Figure 5.16.

5.5.3.2 Metamodel Instance and Source Code Discovery View

The main functionality in the Forward Engineering process is the metamodel instance/-source code discover process. The basic flow to create a discoverer tool can be started in two ways and can be seen in Figure 5.18.

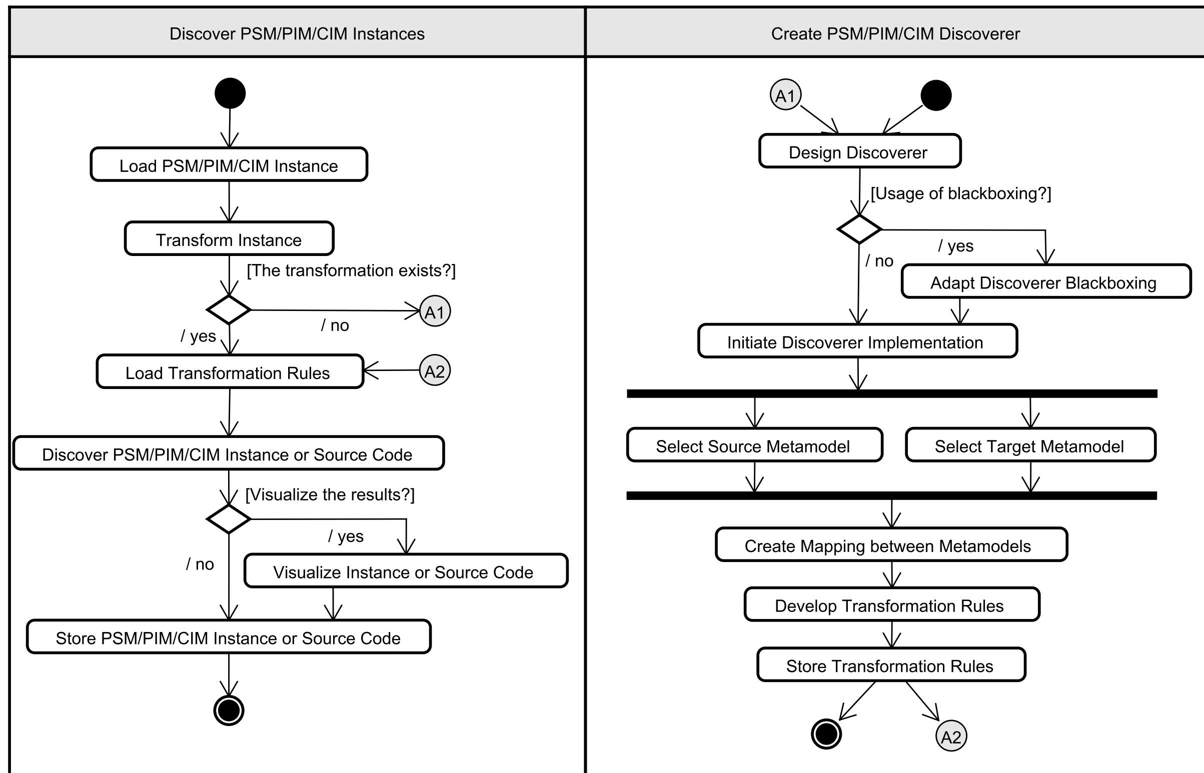

Figure 5.18: Metamodel Instance and Source Code Discovery View

The first one is by discovering source code from an instance and the second way is by discovering an instance from another instance. The discovery process for both are very similar an can be synthesized as follows. The fist step is to load the instance and the existing transformation rules. If the transformation rules needed do not exists it could be created, as presented in the following paragraph. After choosing the right transformation rules, they should be applied in order to get the needed metamodel instance. The final step of the discovery algorithm is to store the instance in its corresponding repository. As an optional action, the tool could offer a functionality to visualize the instance/source code.

The flow to create a discoverer is as follows. The first step is to design how the algorithm to discover instances should behave and the metamodels that will be handled. Sometimes it is possible to reuse an existing algorithm/tool to avoid rework, then the modernization engineer should design how this algorithm will be integrated in the tool. The discovery process need two metamodels as input, one for the source instance (Select Source Metamodel) and another to get the target instance (Select Target Metamodel). A mapping should be developed in order to create the transformation rules. This step is responsible for specifying which element in the target metamodel corresponds to the

source metamodel. This mapping enables the transformation rules development. The last step is to store the transformations rules to be used by the metamodel instance discoverer.

5.5.3.3 Metric and Measurement Calculation View

In a FE modernization tool it is possible to calculate metrics in metamodel instances. The flow to calculate metric is as follows and can be seen in Figure 5.19.

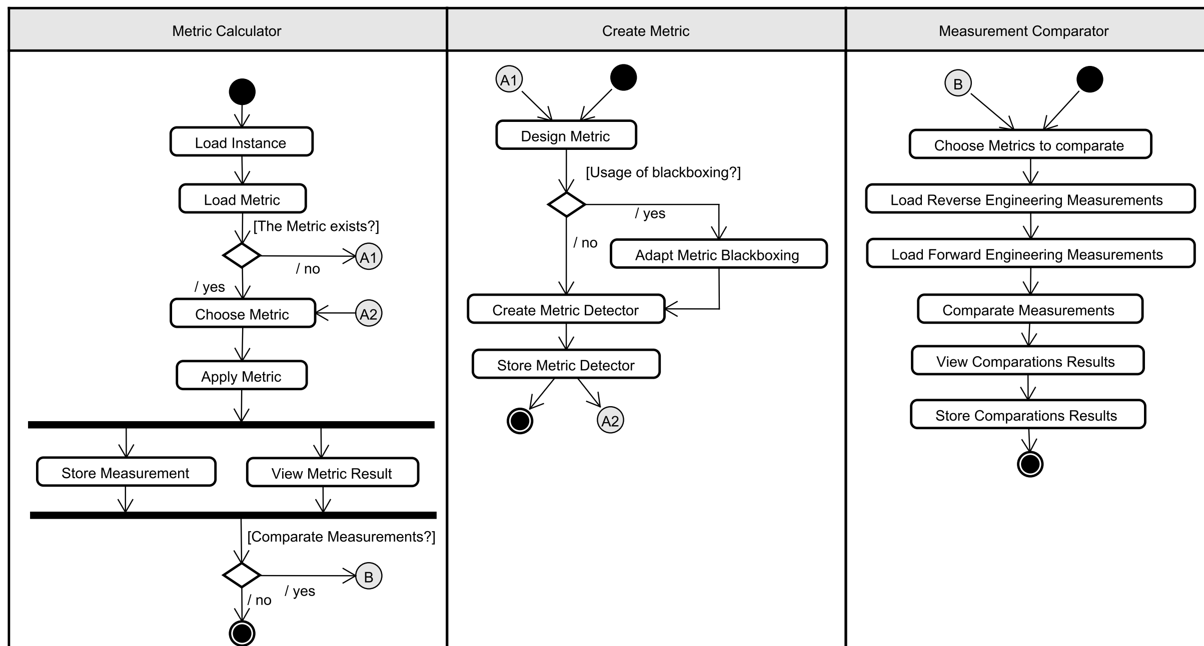

Figure 5.19: Metric and Measurement Calculation View

First, the instance that will be analysed should be loaded and then the existing metrics are loaded. The software engineer will decide which algorithm will be applied, however the needed metric could not be available/exists then it could be created (explained in the last paragraph). If the wanted metric is available, the metric is then selected (Choose Metric) and then applied in the metamodel instance (Apply Metric). The following steps are to store the measurement and to view the measurement results.

If the software engineer wants to compare measurements from reverse engineering and forward engineering flows a new algorithm should be called. Measurement comparator starts at choosing the metrics that are going to be considered and then the RE and FE measurements are loaded so the Comparator Measurement algorithm could analyze and show the results.

The flow to create a metric algorithm is as follows. The first step is to design the metric and how the algorithm to calculate the metric should behave. Sometimes it is possible to

reuse an existing algorithm/tool to avoid rework, then the modernization engineer should design how this algorithm will be integrated in the tool. When the metric algorithm is created, it should be stored in order to be available.

5.5.3.4 Forward Engineering Component Diagram View

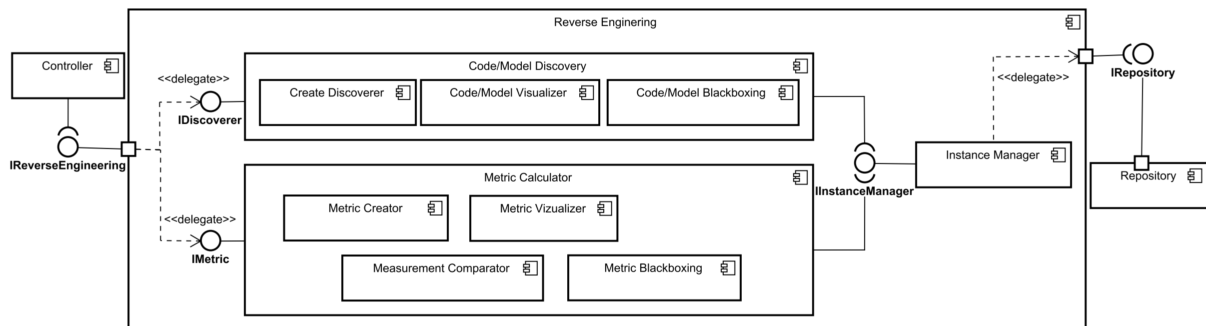

Figure 5.20: Forward Engineering Component Diagram View

In Figure 5.20 is represented the component diagram of Forward engineering. This diagram has two main components: Code/Model Discovery and Metric Calculator. The Code/Model Discovery is responsible for clustering the components that aim to support the conversion of source code or models for the forward engineering. The Metric Calculator component is responsible for clustering all components that support the analysis of the software systems metrics data.

5.6 RA usage Guidelines

The views presented in previous section summarize our RA by displaying the main layers, components, relationship and activities that should be present in a modernization tool. In this section we present some guidelines to help modernization engineers and software architects on the process of creating new interoperable modernization tools.

The first step on building a new modernization tool is to understand how it will be use, if by means a service request/Browser or by existing IDEs through plugins. This decision should not impact in MT architecture, however it will impact on how the service/functionality results of your tool will be delivered.

The second step is to understand where the purpose of your tools is focuses, if reverse engineering, restructuring or forward engineering. To support this understanding we have our taxonomy proposed on Chapter 4 and Figures 5.1, 5.2 and 5.3 in which software architects and modernization engineers can have a global view of the environment needed

to support the MT. Depending on the MT's purpose, more than one layer is impacted, thus, identifying the correct layers and components will support not only the correct design but also modularization and interoperability.

The third step is to determine which components and repositories will be part of the new MT' architecture. For instance, if a MT aims to perform a Language-to-Language Conversion (Modernization scenarios - Table 4.1) and the focus in only on the reverse engineering part, there is no need to create the other layers. However, if the focus is on the language-to-language transformation rules at least the restructuring and reverse engineering layers should exist, even if the mechanism used in the reverse engineering part already exists.

After deciding which layers and components will be part of the architecture, the fourth step is to identify the operations that each component will be responsible to execute. Each layer has its own set of activity diagram view that displays the main operations needed to execute the flow of a component. In an activity diagram view, all the activities represent actions/functionalities that could be translated in to methods/functions in the source code implementation.

Finally, the fifth step is adjust the selected components in to the specific component diagram view. These views can help modularize the source code that will be produced in step 4. The structure of these views could be translated in to the package structure of the source code or how each component could be available as a service.

5.7 Methodology Employed

In this section we present the methodology of **RADM** that is a **R**eference Architecture for **ADM**-based **M**odernization Tools. This RA aims to help modernization engineers in the process of building modernization tools that perform modernizations in ADM context. It is important to mention that the taxonomy presented in Chapter 4 provided the basic knowledge and naming convention for our RA and also supported the views with the expected flow for each component, as we had to analyze all possible modernization scenarios to build the taxonomy. The architectural views and its descriptions are presented in this chapter to provide a documentation of how the RA could be used.

The Family-Architecture Assessment Method (FAAM) and Architecture Trade-off Analysis Method (ATAM) are examples of evaluation methodologies for reference architecture (Angelov et al., 2008a). These methodologies rely on questioning techniques that take in consideration domain specialists and in ATAM we also have measuring techniques that support quantitative measurement in order to support the architecture's evaluation. In

order to establish our RADM, we adapted a methodology followed by Nakagawa et al. (2014a). The PROSA-RA methodology not only supports the evaluation step of a RA but also its creation contemplating the source investigation, architectural analysis of the sources and the evaluation itself. The following sections present the result of each step to the RADM creation.

5.7.1 Information Source Investigation

Following the approach proposed by Nakagawa et al. (2014a) in Step 1 (Information Source Investigation) we have collected official ADM papers, publications, and existing tools to help building our RA. Different sources were considered, mainly related to ADM domain. These sources can be classified into three sets: I) ADM publications; II) Existing Modernization Tools; and III) Existing Model and Code Refactoring Tools.

The analysis of publications concentrated on primary studies that presented any kind of modernization tool. The main goal was to extract which were the main functionalities and how this kind of tool behave while supporting the modernization process. In addition, we also have used a systematic mappings on architecture-driven modernization (Durelli et al., 2014) and a systematic review on UML model refactoring (Misbhauddin e Alshayeb, 2015). Another point to mention is that in order to identify the main architectural requirements about modernization tools available in the literature we also carried out a search on OMG website, more specifically in the ADM part.

Following, each set of information sources is described in more details.

- **Set I - ADM publications:** An important source of information is the set of artifacts provided by OMG about ADM blueprint. We analyzed the ADM website, 4 white papers, 1 glossary, 1 formal specifications about software modernization[6]; and the ISO/IEC 25010 (ISO/IEC25010, 2011). As a result, we extracted the architecture styles of ADM blueprint by means of its main concepts which are software Reengineering and Model Driven Architecture (MDA) which are pipes and filters; and layered architecture.

- **Set II - Existing modernization tools:** In order to see how ADM concepts were being implemented in researches and at the same time performing the identification of technical requirements we analyzed existing modernization tools in the literature and in the industry. We analyzed tools from a systematic mapping on ADM (Durelli et al., 2014), and two ADM books (Pérez-Castillo et al., 2011b; Ulrich e Newcomb, 2010b).

[6]This page provides a summary of OMG specifications about software modernization. https://www.omg.org/spec/category/software-modernization

We also extended the systematic mapping on ADM (Durelli et al., 2014) to consider publications till 2019. The OMG/ADM make available a list of vendors[7] that somehow employs the ADM concepts in its tools or they supported the ADM in its consolidation process. We analysed these tools in order to understand their behavior and their roles in the modernization process to extract common properties and requirements. As a result, we rise a set of architectural requirements to support the creation of our reference architecture. Another result is that based on this investigation we developed a Taxonomy for Software Modernization Tools to help in the classification of modernization tools and it will also be helpful to instantiate the RADM.

- **Set III - Existing Model and Code Refactoring Tools:** The restructuring phase of ADM blueprint is the most crucial when considering the modernization process. To help the building process of the architectural views of this phase, we performed a systematic mapping that took in consideration existing refactorings tools in the literature to provide a classification of the approaches and to extract the main functionalities of these tools. Another set of tools used here was a systematic review on UML model refactoring (Misbhauddin e Alshayeb, 2015). As a result, we complemented the requirements found in Set II specifically in the restructuring phase.

5.7.2 Architectural Analysis and Requirements

Based on the previous investigation, all the artifacts collected in Step 1 were analyzed in order to elicit the architectural requirements of software modernization tools. We created our reference architecture in a way that instances of it, i.e., modernization tools generated with its support, met some requirements. The identified set of requirements are distributed in five kinds: Global (G), Reverse Engineering (RE), Restructuring (R), Forward Engineering (FE), and Quality Attributes (QA). In table 5.1 are presented the requirements, in the first column we have an identifier (ID) and in the second column the description of each requirement.

The requirements that are common to all types of modernization tools are presented as Global, this set represents common requirements to the four other groups, this is, it is applicable to FE, R, QA and FE. The RE, R, and FE requirements are specific to each modernization tool domain which means that they only should be implemented if it fulfills the modernization tool that are going to be instantiated. The quality attributes (QA) can be understood as characteristics of a concrete architecture that will be created based on our RA. Our set of quality attributes (QA-1, QA-2 and QA-3 in Table 5.1) come from

[7]https://www.omg.org/adm/directory.htm

Table 5.1: Requirements

ID	Description
	Requirement Group - Quality Attributes (QA), Global (G), Forward Engineering (FE), Restructuring (R) and Forward Engineering (FE)
ID	**Description**
QA-1	The RA must guarantee a good **Reusability** level in its instances. This means the effort to reuse modules from on modernization tool into another should be facilitated and intuitive. This requirement seeks to make the building of modernization tools easier and also that better quality is reached by reusing existing modules.
QA-2	The RA must ensure **Interoperability** between the modernization tools generated with its support. This means that a modernization tool is able to process the output generated by another one.
QA-3	The RA must ensure the modernization tools generated with its support have a good level of **Modularity**. This means that the main concerns of these tools keep evident in the design. This requirements seeks to guarantee that maintenance actions are well localized and confined in specific modules.
G-1	The RA should provide a Visualization module. This module is responsible for presenting metamodel instances and results to the software engineer.
G-3	All the functionalities should be developed considering the concept of component and modules to guarantee the requirements **QA-3** and **QA-1**.
G-4	The RA should follow the layered architectural style. This style can facilitates the maintenance and mainly the implementation of different types of visualizations and the requirements **QA-3** and **QA-1**. In addition, this architectural style had the most occurrence on our literature review.
G-5	The architecture between the RE, R and FE modules should consider the pipes and filters architectural style since they have a specific architectural flow that matches with this style.
G-8	The RA should provide a module to manage all data persistence of the other modules to facilitate **QA-3** and **QA-1** by avoiding the persistence spread in other functionalities.
G-8.1	The RA should have separated repositories to the different artifacts in order to group only similar artifacts.
G-9	The RA should present the whole modernization process and how they communicate with each other.
G-10	The RA should present the the possible input and output files in each modernization process.
G-11	The RA should allow the usage of blackboxing algorithms. These algorithms have implementation in which their content is not explicit known, however the modernization engineer wants to integrate them in the tool.
RE-1	The RA should provide a module to perform model discovery, that is the process of discovering a model from source code.
RE-2	The RA should provide a module to perform model understanding, that is the process of discover new models from a current model.
RE-3	The RA should provide a module to perform flaws detection in instances to mark possible snippets candidates to be restructured.
RE-4	The RA should allows the metrics calculation in metamodels instances.
RE-5	The RA should allows the creation of metrics to be applied in metamodels instances.
R-1	The RA should provide a module to load metamodel instances that were analysed and marked in **Flaws Detection Module (RE-3)** in order to prepare them for the refactorings.
R-2	The RA should allow the creation of module that supports the creation of refactorings and how the new refactorings should be stored.
R-3	The RA should provide a module to execute existing refactorings (transformations rules) in metamodel instances.
R-4	The RA should provide a functionality to verify the behavior preservation of the refactorings.
R-5	The RA should provide a module to propagate the changes performed by the refactoring in the different metamodel instances viewpoints.
FE-1	The RA should provide a module to perform model discovery, that is the process of discovering a model from source code.
FE-2	The RA should provide a module to perform model understanding, that is the process of discover new models from a current model.
FE-3	The RA should allows the application of metrics in metamodels instances.
FE-4	The RA should provide a module to show the comparison of the metrics that were applied in reverse engineering and forward engineering modules in order to provide an analysis of the software system, represented in metamodel instances, after and before the restructuring phase.

mainly by the information from Set I, more specifically from ADM blueprint and ISO/IEC 25010 (ISO/IEC25010, 2011). These quality attributes also surfaced as we looked at the set of related papers and tools, as when we checked for reusability we often concluded that the tools were not reusable as they were not modular and interoperable enough to be reused. In general, this analysis and requirement elaboration was important not only to identify the concepts but also to guide us during the creation of our reference architecture once we had to check if all constraints and concepts were being fulfilled and correctly represented.

Next section we present the Architectural Synthesis which contains the architectural views that were developed to integrate the Reference Architecture.

5.8 Evaluation

Following the approach proposed by Nakagawa et al. (Nakagawa et al., 2014a) we evaluate our reference architecture performing a survey with ADM experts (software developers and researchers) and software architects. The evaluation had two mains stages(Shull et al., 2007): (i) Reading of the technical report about the RA; and (ii) Answering the questions in an on-line form[8] . The group had eight participants, two software architects with more than ten years of experience and six ADM experts that have worked directly with ADM; either by conducting research or developing ADM-based tools. The results obtained from the survey were divided in (a) Acceptance; and (b) Overall acceptance. The first one was focused on evaluate specific parts of the RA and the second one was focused on the content in general.

5.8.1 Evaluating the RA Acceptance

In order to evaluate the acceptance of our RA we wanted to know from the participants if the architectural views provided in the technical report were understandable and if they were able to fulfill its purpose. We proposed four affirmations and the participants has to answer using the following options: Strongly Agree (SA), Agree (A), Tend to Agree (TA), Neutral (N)[9], Tend to Disagree (TD), Disagree (D), and Strongly Disagree (SD).

The four affirmations were: AF1 - “The Pipes and Filters view provides an architectural overview of ADM blueprint”; AF2 - “The Abstract Layered view is enough to understand how the architectural layers communicate with each other”; AF3 - “The Abstract Layered

[8] Available at: https://www.dropbox.com/s/4m53smjnew87jet/ra-evaluation.pdf?dl=0

[9] A Neutral answer means that the participant does not feel confident to accept or deny a specific affirmation.

view is enough to understand how a modernization tool should be designed"; and AF4 - "The Concrete Layered view provides a complete overview of how the components should be designed inside of each layer".

In Table 5.2 we present the answers for each affirmation. In AF1, 100% agree that the pipes and filter view provides an architectural overview of ADM blueprint. About AF2, 87,5% agree that the abstract layered view is enough to understand how the architectural layers communicate with each other and only 12,5% tends to disagree about this affirmation. In AF3, 87,5% of the participants agree that is enough to understand how a modernization tool should be designed and 12,5% disagree about this affirmation. Finally, in AF4, 100% of the participants agree that the concrete layered view provides a complete overview of how the components should be designed inside of each layer.

Table 5.2: Acceptance of the Reference Architecture views

ID	SA	A	TA	N	TD	D	SD
AF1	37,5%	62,5%	0%	0%	0%	0%	0%
AF2	12,5%	75%	0%	0%	12,5%	0%	0%
AF3	25%	25%	37,5%	0%	0%	12,5%	0%
AF4	25%	62,5%	12,5%	0%	0%	0%	0%

5.8.2 Evaluating the RA Overall acceptance

In order to evaluate the overall acceptance of our RA we wanted to know from the participants if the quality of the technical report was acceptable and if they believe that the RA was broad enough to help the creation of different modernization tool. The two affirmations were: AF1 - "I believe the reference architecture is clear and well-described"; and AF2 - "I believe the reference architecture is useful for instantiating different types of modernization tools".

In Table 5.3 we present the answers for each affirmation. In AF1, 100% agree that the reference architecture is clear and well-described. About AF2, 75% agree that the reference architecture is useful for instantiating different types of modernization tools and 25% of the participants were neutral about this affirmation.

Table 5.3: Overall acceptance of the Reference Architecture

ID	SA	A	TA	N	TD	D	SD
AF1	25%	37,5%	37,5%	0%	0%	0%	0%
AF2	25%	12,5%	37,5%	25%	0%	0%	0%

5.8.3 Evaluating Discussions

The main focus of this evaluation was to get a perspective from people about the reference architecture. We opted for choosing a small group of highly qualified people in order to increase the chances of getting a trustful feedback since we were sure they had the necessary knowledge to evaluate the RA. As a result, we got a list of issues that are going to help us in the RA improvement process. For instance, some participants claimed that in some architectural views more information/explanation were needed in order to increase the understanding. For instance, one participant stated that the Structural views do not displayed how each component would communicate internally with each other. To mitigate this point we created two more views, Activity and Component diagrams, which are focused on displaying how each component can interact with other in order to complete the modernization process. The other comments we reflected in the improvement of the views and requirements descriptions.

Another interesting issues pointed out was that some participants suggested to add constraints on the architectural views in order to provide some guidelines on what should not be performed to ensure the quality of the architectural instances.

It is important to remember that since we are proposing a RA, it is out of our scope providing some lower level implementation details. Thus, the architectural views are not supposed to suggest how the layers and components should be implemented or which technologies should be used. The RA should be used as a reference guide to understand and to design modernization tools.

5.9 Threats to Validity

In this section we present the points that could affect the conclusion of our preliminary evaluation.

Internal Validity - refers to whether there is sufficient evidence to support our conclusions. We claim that the RA can support the designing of MT. However, in this thesis we were not able to instantiate our RA to create a concrete MT. The point is that in the context of our research group we had several opportunities to design and build several MTs and the knowledge contained in the RA and Taxonomy has a summarized version of the required information we needed back then in order to build our tools. We claim that if this kind of knowledge already existed we would not face some of the issues during our development, or at least the RA could facilitate and speed up the MT creation.

External validity - refers to the generalizibility of the treatment/condition outcomes. Our evaluation was carried out with eight participants that contemplated masters and PhD students with experience in ADM and some software architects. The evaluation considered only two of the three main set views (Structural view, Data flow view and Dynamic views). Thus, it is not possible to affirm that the overall acceptance we got from the evaluation can be applied to the entire RA. In order to mitigate this we claim that the Dynamic views set is a derivation of the other views and the main activities inside were already presented in the requirements table that was also shared with the participants. In addition, we proposed in section 6 in Future Works section an extension/improvement of our RA evaluation.

5.10 Final Considerations

Reference architectures are architectural proposals with the objective of solving a problem in a specific domain area, that take into consideration existing/current work. Our RA presented in this chapter contains both architectural information from existing modernization tools and the knowledge available in the scientific literature.

The architectural views presented here contemplate the complete modernization processes described by ADM, in which they represents an specific point of view that aims to support modernization engineers in the process of creating new modernization tools. Our evaluation main focus was to provide an overview of professionals and academic researcher's general acceptance in order to validate the architectural views here provided.

In Chapter 6 we present our conclusions, contributions, list of publications, and future works.

Chapter 6

Conclusions

In this Ph.D. research, the focus was on the development of a reference architecture that supports modernization engineers on building and designing modernization tools. Specifically, we presented two main contributions that is discussed in the following section.

This chapter is organized as follows. Section 6.1 states the main contributions of this thesis. Section 6.2 presents the main limitations of this thesis. Section 6.3 lists the future works we envision. Section 6.4 summarizes the list of publications resulted from this Ph.D. research and related works. Finally, in Section 6.5 we present the next publications we are working on.

6.1 Contributions

The development of this thesis resulted in two main contributions: i) the taxonomy for Modernization Tools and ii) the reference architecture.

The taxonomy aims at classifying the concepts arround modernization tools and how to use this information. As stated in Chapter 4, we have noticed a lack of consensus in this area and the existence of a taxonomy can guide modernization engineers in the understanding process of modernization tools needs.

The reference architecture is the main product of this thesis. As presented in Chapter 5, it consists in three main views where each one concentrates on a specific part of the MT architecture.

Other more specific and parallel contributions were published in papers. For example, in paper Santos et al. (2018), published in IEEE Software, we present an overview of software modernization, providing modernization tools examples and some refactoring guidance. And in paper Santos et al. (2019c), published in SBES, we present an snippet of our reference architecture and a preliminary evaluation on the first two sets of views.

Another aspect that it is important to raise up is regarding the evaluation. The evaluation of a reference architecture is a big challenge. Although we had no resources for conducting a more complete evaluation, the evaluation performed was enough for giving us a first impression on the quality of the RA, in Future Work section we present the next steps to improve our RA evaluation.

As a **general contribution** of this Ph.D. research consists in a set of artifacts regarding modernization tools that helps the understanding and development of MT according to three main software engineering concepts: Model-Driven Architecture, Software Reengineering and Architecture-Driven Modernization.

We claim that the technical documentation provided in this thesis can clarifies the main characteristics of the development of modernization tools makes the creation process more planned, also supports the interoperability between these tools derived from our RA.

We are also able to say that when creating modernization tools with the support of a documentations that has the relevant information about modernization area in a single place the modernization engineers can focus much more in **what should be developed** and less effort on **how it should be developed**. By **"what should be developed"** we mean the process of building the algorithms of the MT's components. By **"how it should be developed"**, we mean the process on how the architecture of the tools should be planned and developed.

Recapping our research question from Chapter 1: "How do to design interoperable and modularized ADM-based modernization tools that incorporate modernization solutions?" we claim that our contributions support our thesis that modernization tools would be more easily designed if supported by a set of artifacts that summarizes all knowledge involving this domain. In this thesis we did not cover the creation of a modernization tool with the support of our RA. However, from previous works we already have experience designing and building this kind of tools and the main obstacle was to design MT having in mind all possible involved concepts and yet build in such way the MT could interoperate with other MTs.

6.2 Limitations

As main limitations we can mention the following topics:

- A more profound evaluation to investigate if the level of details provided in the architectural views of our RA are enough to support the creation of new MTs;
- An evaluation of the proposed taxonomy to check if the artifacts contain enough information to understand the modernization tools field;
- Since we do not create new MT by using our RA and Taxonomy, we should perform experiments to check if the taxonomy and the RA are enough to design new MT;

6.3 Future Work

This section aims at mitigating the limitations raised in Section 6.2. As future works we envision the following opportunities:

- Apply the Delphi Method and Focus group using academic researchers and professionals working in the IT area for evaluating the proposed Taxonomy and the Reference Architecture. An initial methodological strategy is already in development, as can be seen in Appendix C.
- Submit the proposed reference architecture to a reference architecture specialist and modernization engineers in order to get feedback to improve the our research results.
- Implementation of modernization tools based entirely on the proposed Reference Architecture.
- Update the systematic mapping on ADM to update, if necessary, our reference architecture.

6.4 List of Publications

1. LANDI, A.; MARTÍN, D. S.; **SANTOS, B. M.**; CUNHA, W.; DURELLI, R.; CAMARGO, V. Architectural conformance checking for KDM-represented systems. Journal of Systems and Software, p. 1-29, 2021. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121221002132

2. SANTIBANEZ, D. G. S. M. ; ANGULO, G. C. ; **SANTOS, B. M.**; HONDA, RAPHAEL; CAMARGO, VALTER V. DE . Specification and Use of Concern Metrics for Supporting Modularity-Oriented Modernizations. SOFTWARE QUALITY JOURNAL, v. 1, p. 1-1, 2020.

3. **SANTOS, B. M.**; DE SOUZA LANDI, ANDRÉ ; DE GUZMÁN, IGNACIO GARCÍA-RODRÍGUEZ ; PIATTINI, MARIO ; de Camargo, Valter Vieira . Towards a Reference Architecture for ADM-based Modernization Tools. In: the XXXIII Brazilian Symposium, 2019, Salvador. Proceedings of the XXXIII Brazilian Symposium on Software Engineering - SBES 2019. New York: ACM Press, 2019. p. 114-1. (Santos et al., 2019c)

4. **SANTOS, B. M.**; SANTIBANEZ, D. G. S. M. ; HONDA, RAPHAEL ; CAMARGO, VALTER V. DE . Concern Metrics for Modularity-Oriented Modernizations. In: 12th International Conference on the Quality of Information and Communications Technology, 2019, Ciudad Real. QUATIC, 2019. v. 1. p. 1-1.

5. **SANTOS, B. M.**; LANDI, ANDRÉ DE S. ; SANTIBÁÑEZ, DANIEL S. ; DURELLI, RAFAEL S. ; DE CAMARGO, VALTER V. . Evaluating the Extension Mechanisms of the Knowledge Discovery Metamodel for Aspect-Oriented Modernizations. JOURNAL OF SYSTEMS AND SOFTWARE, v. 1, p. 1, 2018.

6. **SANTOS, B. M.**; DE GUZMAN, IGNACIO GARCIA-RODRIGUEZ ; DE CAMARGO, VALTER V. ; PIATTINI, MARIO ; EBERT, CHRISTOF . Software Refactoring for System Modernization. IEEE SOFTWARE, v. 35, p. 62-67, 2018.

7. ANGULO, G.; MARTÍN, D. SAN; **SANTOS, B. M.**; FERRARI, F. C.; DE CAMARGO, V. V.. An Approach for Creating KDM2PSM Transformation Engines in ADM Context. In: the VII Brazilian Symposium on Software Components, Architectures, and Reuse, 2018, São Carlos. Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse on - SBCARS '18, 2018. p. 92.

8. LANDI, ANDRE DE S. ; CHAGAS, FERNANDO ; **SANTOS, B. 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. 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2017. p. 327.

9. **SANTOS, B. M.**; CAMARGO, V. V. . A Reference Architecture for KDM-based Modernization Tools. In: VI Workshop de Teses e Dissertações do CBSoft, 2016, Maringá - PR. VI Workshop de Teses e Dissertações do CBSoft (WTDSoft 2016), 2016. p. 1-9.

10. PAULA, M. H. ; SERIKAWA, M. A. ; LANDI, A. S. ; **SANTOS, B. M.**; COSTA, R. S. ; CAMARGO, V. V. . SARAMR: Uma Arquitetura de Referência para Facilitar Manutenções em Sistemas Robóticos Autoadaptativos. In: IV Workshop on Software Visualization, Evolution and Maintenance, 2016, Maringá - PR. IV Workshop on Software Visualization, Evolution and Maintenance, 2016. p. 1-8.

6.5 Next Publications

We are working on two new papers, which are:

- A Taxonomy for Modernization Tools. This paper aims at presenting our taxonomy and display some usage scenarios, as can be seen in Chapter 4.
- A Reference Architecture for Designing Modernization Tools. This paper aims at presenting a more complete view of our RA, including the Dynamic views that were not covered in Santos et al. (2019c).

In addition, we are also planning other papers derived from the new evaluations yet to be executed, as can be seen in Appendix C.

Bibliography

Akodadi, K. A survey of cloud migration methods: A comparison and proposition, p. 1–7. 2016.

Alawneh, L.; Hamou-Lhadj, A. Execution traces: A new domain that requires the creation of a standard metamodel. In: *International Conference on Advanced Software Engineering and Its Applications*, Springer, p. 253–263, 2009.

Angelov, S.; Grefen, P. An e-contracting reference architecture. *J. Syst. Softw.*, v. 81, n. 11, p. 1816–1844, 2008.

Angelov, S.; Trienekens, J. J.; Grefen, P. Towards a method for the evaluation of reference architectures: Experiences from a case. In: *Software Architecture: Second European Conference, ECSA 2008 Paphos, Cyprus, September 29-October 1, 2008 Proceedings 2*, Springer, p. 225–240, 2008a.

Angelov, S.; Trienekens, J. J. M.; Grefen, P. *Towards a method for the evaluation of reference architectures: Experiences from a case* Berlin, Heidelberg: Springer Berlin Heidelberg, p. 225–240, 2008b.

Angulo, G.; Martín, D. S.; Santos, B.; Ferrari, F. C.; de Camargo, V. V. An approach for creating kdm2psm transformation engines in adm context: The rute-k2j case. In: *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse*, p. 92–101, 2018.

Arcelli Fontana, F.; Raibulet, C.; Zanoni, M. Alternatives to the knowledge discovery metamodel: An investigation. *International Journal of Software Engineering and Knowledge Engineering*, v. 27, n. 07, p. 1097–1128, 2017.

ARNOLD, R. S. *Software reengineering.* IEEE Computer Society Press, 1993.

BABAR, M. A.; GORTON, I. Comparison of scenario-based software architecture evaluation methods. In: *11th Asia-Pacific Software Engineering Conference*, p. 600–607, 2004.

BERINATO, S. *A rash of it failures.* 2003.
Disponível em `http://www.cio.com/archive/061503/tl_health.html`

BODZIONY, M.; WREMBEL, R. Reference architecture for running large scale data integration experiments. In: *International Conference on Database and Expert Systems Applications*, Springer, p. 3–9, 2021.

BOURQUE, P.; ABRAN, A. An innovative software reengineering tools workshop—a test of market maturity and lessons learned. *SIGSOFT Softw. Eng. Notes*, v. 19, n. 3, p. 30–34, 1994.
Disponível em `http://doi.acm.org/10.1145/182824.182829`

BRUNELIÈRE, H.; CABOT, J.; DUPÉ, G.; MADIOT, F. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, v. 56, n. 8, p. 1012 – 1032, 2014.
Disponível em `http://www.sciencedirect.com/science/article/pii/S0950584914000883`

CANOVAS, J.; MOLINA, J. An architecture-driven modernization tool for calculating metrics. *IEEE Software*, v. 27, n. 4, p. 37–43, 2010.

CHAGAS, F. B. *Checagem de conformidade arquitetural na modernização orientada a arquitetura.* MSc Dissertation, UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2016.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE software*, v. 7, n. 1, p. 13–17, 1990.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, v. 7, n. 1, p. 13–17, 1990.

COSTA, R. S. *Uma Abordagem para Identificação de Violações Arquiteturais em Processos de Migração de Plataformas de Nuvem.* MSc Dissertation, UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2017.

DURAK, U. Extending the knowledge discovery metamodel for architecture-driven simulation modernization. *Simulation*, v. 91, n. 12, p. 1052–1067, 2015.

DURELLI, R. S. *Uma abordagem para criação, reúso e aplicação de refatorações no contexto da modernização dirigida a arquitetura [online].* PhD Thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo. Tese de Doutorado em Ciências de Computação e Matemática Computacional. [acesso 2017-07-12]. Disponível em: <http://www.teses.usp.br/teses/disponiveis/55/55134/tde-29092016-145938/>, São Carlos, 2016.

DURELLI, R. S.; SANTIBÁÑEZ, D. S. M.; MARINHO, B.; HONDA, R.; DELAMARO, M. E.; ANQUETIL, N.; DE CAMARGO, V. V. A mapping study on architecture-driven modernization. In: *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*, p. 577–584, 2014.

EICKELMANN, N. S.; RICHARDSON, D. J. An evaluation of software test environment architectures. In: *Proceedings of the 18th International Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, p. 353–364, 1996 (*ICSE '96*, v.1).
Disponível em `http://dl.acm.org/citation.cfm?id=227726.227798`

EINARSSON, H. T.; NEUKIRCHEN, H. An approach and tool for synchronous refactoring of uml diagrams and models using model-to-model transformations. In: *Proceedings of the Fifth Workshop on Refactoring Tools*, New York, NY, USA: ACM, p. 16–23, 2012 (*WRT '12*, v.1).
Disponível em `http://doi.acm.org/10.1145/2328876.2328879`

FOWLER, M.; BECK, K. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999.

FREY, S.; HASSELBRING, W. An extensible architecture for detecting violations of a cloud environment's constraints during legacy software system migration. In: *2011 15th European Conference on Software Maintenance and Reengineering*, p. 269–278, 2011.

FREY, S.; HASSELBRING, W.; SCHNOOR, B. Automatic conformance checking for migrating software systems to cloud infrastructures and platforms. *Journal of Software: Evolution and Process*, v. 25, n. 10, p. 1089–1115, 2013.

GALSTER, M.; AVGERIOU, P. Empirically-grounded reference architectures: a proposal. In: *Proceedings of the joint ACM SIGSOFT conference–QoSA and ACM SIGSOFT*

symposium–ISARCS on Quality of software architectures–QoSA and architecting critical systems–ISARCS, ACM, p. 153–158, 2011.

Gotti, Z.; Mbarki, S. Gui structure and behavior from java source code analysis. In: *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, p. 251–256, 2016.

Grefen, P.; de Vries, R. R. A reference architecture for workflow management systems. *Data & Knowledge Engineering*, v. 27, n. 1, p. 31–57, 1998.

Hollingsworth, D.; et al. The workflow reference model: 10 years on. In: *Fujitsu Services, UK; Technical Committee Chair of WfMC*, Citeseer, 2004.

Honda, R. R. *Modelagem e cômputo de métricas de interesse no contexto de modernização de sistemas legados.* MSc Dissertation, UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2014.

ISO/IEC25010 International organization for standardization. iso/iec 25010:2011. systems and software engineering – systems and software quality requirements and evaluation (square) – system and software quality models. 2011.

Jácome, S.; De Lara, J. Controlling meta-model extensibility in model-driven engineering. *IEEE Access*, v. 6, p. 19923–19939, 2018.

KDM *Object management group (omg), knowledge discovery meta-model (kdm) version 1.3.* 2011.
Disponível em `http://www.omg.org/spec/KDM/1.3/`

KDMAnalytics KDM Analytics - Prioritize, Measure and Quantify CyberSecurity Risk. Disponível em `http://kdmanalytics.com/`, especificação disponível em `http://kdmanalytics.com/resources/standards/kdm/technical-overview/kdm-1-0-annotated-reference/front-matter/`, 2017.

Kiran Mallidi, R.; Sharma, M.; Singh, J. Legacy digital transformation: Tco and roi analysis. *International journal of electrical and computer engineering systems*, v. 12, n. 3, p. 163–170, 2021.

Koch, C. *Supply chain: Hershey's bittersweet lesson.* 2002.
Disponível em `http://www.cio.com/article/31518/Supply_Chain_Hershey_s_Bittersweet_Lesson`

de La Vara, J. L.; Génova, G.; Álvarez-Rodríguez, J. M.; Llorens, J. An analysis of safety evidence management with the structured assurance case metamodel. *Computer Standards & Interfaces*, v. 50, p. 179–198, 2017.

Landi, A.; Martín, D. S.; Santos, B.; Cunha, W.; Durelli, R.; Camargo, V. Architectural conformance checking for kdm-represented systems. *Journal of Systems and Software*, p. 1 – 29, 2021.
Disponível em `https://www.sciencedirect.com/science/article/pii/S0164121221002132`

Ludewig, J.; Lichter, H. *Software engineering: Grundlagen, menschen, prozesse, techniken*. dpunkt. verlag, 2023.

M. Klein, M. Reengineering methodologies and tools a prescription for enhancing succes. *Information Systems Management - ISM*, v. 11, p. 30–35, 1994.

Madisetti, V. K.; Jung, Y. K.; Khan, M. H.; Kim, J.; Finnessy, T. Reengineering legacy embedded systems. *IEEE Design Test of Computers*, v. 16, n. 2, p. 38–47, 1999.

Mansurov, N.; Campara, D. Managed architecture of existing code as a practical transition towards mda. In: *International Conference on the Unified Modeling Language*, Springer, p. 219–233, 2004.

Martin, D.; Kühl, N.; Schwenk, M. Towards a reference architecture for future industrial internet of things networks. *arXiv preprint arXiv:2109.00833*, 2021.

Martin, R. C.; Newkirk, J.; Koss, R. S. *Agile software development: principles, patterns, and practices*, v. 2. Prentice Hall Upper Saddle River, NJ, 2003.

Mens, T. Introduction and roadmap: History and challenges of software evolution. In: *Software evolution*, Springer, p. 1–11, 2008.

Mens, T.; Taentzer, G.; Müller, D. Challenges in model refactoring. In: *Proc. 1st Workshop on Refactoring Tools, University of Berlin*, p. 1–5, 2007.

Mens, T.; Tourwe, T. A survey of software refactoring. *IEEE Transactions on Software Engineering*, v. 30, n. 2, p. 126–139, 2004.

Mens, T.; Van Gorp, P. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, v. 152, p. 125–142, 2007.

MERCIER, D.; CHAWDHARY, A.; JONES, R. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, p. 497–501, 2017.

MISBHAUDDIN, M.; ALSHAYEB, M. Uml model refactoring: A systematic literature review. *Empirical Softw. Engg.*, v. 20, n. 1, p. 206–251, 2015.
Disponível em `http://dx.doi.org/10.1007/s10664-013-9283-7`

MOGHADAM, I. H.; CINNEIDE, M. O. Automated refactoring using design differencing. In: *2012 16th European Conference on Software Maintenance and Reengineering*, p. 43–52, 2012.

MOHAMED, M.; ROMDHANI, M.; GHÉDIRA, K. Classification of model refactoring approaches. *Journal of Object Technology*, v. 8, n. 6, p. 121–126, 2010.

MURAM, F. U.; GALLINA, B.; RODRÍGUEZ, L. G. Preventing omission of key evidence fallacy in process-based argumentations. In: *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, IEEE, p. 65–73, 2018.

NAKAGAWA, E. Y. *Uma Contribuição ao Projeto Arquitetural de Ambientes de Engenharia de Software*. PhD Thesis, Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo (ICMC/USP), 2006.

NAKAGAWA, E. Y.; ANTONINO, P. O.; BECKER, M. Exploring the use of reference architectures in the development of product line artifacts. In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*, New York, NY, USA: ACM, p. 28:1–28:8, 2011 (*SPLC '11*, v.1).
Disponível em `http://doi.acm.org/10.1145/2019136.2019168`

NAKAGAWA, E. Y.; BECKER, M.; MALDONADO, J. C. Towards a process to design product line architectures based on reference architectures. In: *Proceedings of the 17th International Software Product Line Conference*, New York, NY, USA: ACM, p. 157–161, 2013 (*SPLC '13*, v.1).
Disponível em `http://doi.acm.org/10.1145/2491627.2491651`

NAKAGAWA, E. Y.; GUESSI, M.; MALDONADO, J. C.; FEITOSA, D.; OQUENDO, F. Consolidating a process for the design, representation, and evaluation of reference

architectures. In: *2014 IEEE/IFIP Conference on Software Architecture*, Sydney, NSW, Australia: IEEE, p. 143–152, 2014a.

Nakagawa, E. Y.; Guessi, M.; Maldonado, J. C.; Feitosa, D.; Oquendo, F. Consolidating a process for the design, representation, and evaluation of reference architectures. In: *2014 IEEE/IFIP Conference on Software Architecture*, p. 143–152, 2014b.

Neubauer, P.; Bill, R.; Wimmer, M. Modernizing domain-specific languages with xmltext and intelledit. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, p. 565–566, 2017.

Norta, A. H. Exploring dynamic inter-organizational business process collaboration. 2007.

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

OMG *Object management group (omg) - architecture-driven modernization.* 2021. Disponível em `https://www.omg.org/adm/`

Opdyke, W. F. Refactoring object-oriented frameworks. 1992.

Park, G.; Chung, L.; Khan, L.; Park, S. A modeling framework for business process reengineering using big data analytics and a goal-orientation. In: *2017 11th International Conference on Research Challenges in Information Science (RCIS)*, p. 21–32, 2017.

Pérez-Castillo, R.; de Guzmán, I. G.-R.; Piattini, M. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Comput. Stand. Interfaces*, v. 33, n. 6, p. 519–532, 2011a.
Disponível em `http://dx.doi.org/10.1016/j.csi.2011.02.007`

Pérez-Castillo, R.; de Guzmán, I. G.-R.; Piattini, M.; Weber, B.; Places, A. S. An empirical comparison of static and dynamic business process mining. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*, New York, NY, USA: ACM, p. 272–279, 2011b (*SAC '11*, v.1).
Disponível em `http://doi.acm.org/10.1145/1982185.1982249`

Pires, J. P.; e Abreu, F. B. Knowledge discovery metamodel-based unit test cases generation. In: *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, p. 432–433, 2018.

Pérez-Castillo, R.; Fernández-Ropero, M.; d. Guzmán, I. G. R.; Piattini, M. Marble. a business process archeology tool. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, p. 578–581, 2011a.

Pérez-Castillo, R.; Guzmán, I. G. R. d.; Piattini, M. *Architecture-driven modernization* p. 75–103, 2011b.

Ren, S.; Butler, G.; Rui, K.; Xu, J.; Yu, W.; Luo, R. A prototype tool for use case refactoring. 2004.

Ricardo Pérez-Castillo, I. G. R. d. G.; Piattini, M. *Modern software engineering concepts and practices: Advanced approaches: Advanced approaches. chapter 4 - architecture-driven modernization.* IGI Global, 2010.

Rutledge, L.; Italiaander, R. Toward a reference architecture for traceability in sbvr-based systems. In: *Proceedings of the Seventh International Workshop on Controlled Natural Language (CNL 2020/21)*, 2021.

d. S. Landi, A.; Chagas, F.; Santos, B. M.; Costa, R. S.; Durelli, R.; Terra, R.; d. Camargo, V. V. Supporting the specification and serialization of planned architectures in architecture-driven modernization context. In: *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, p. 327–336, 2017.

Sabiri, K.; Benabbou, F. A legacy application meta-model for modernization. In: *Proceedings of the 2nd international Conference on Big Data, Cloud and Applications*, p. 1–6, 2017.

Sadovykh, A.; Vigier, L.; Hoffmann, A.; Grossmann, J.; Ritter, T.; Gomez, E.; Estekhin, O. Architecture driven modernization in practice #150; study results. In: *2009 14th IEEE International Conference on Engineering of Complex Computer Systems*, p. 50–57, 2009.

San Martín, D.; Camargo, V. A domain-specific language to specify planned architectures of adaptive systems. In: *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, SBCARS '21, New York, NY, USA: Association for Computing Machinery, p. 41–50, 2021 (*SBCARS '21*, v.).
Disponível em `https://doi.org/10.1145/3483899.3483903`

Santibáñez, D. G. S. M. *Mineração de interesses no processo de modernização dirigida a arquitetura.* MSc Dissertation, UNIVERSIDADE FEDERAL DE SÃO CARLOS, 2013.

Santos, B. M. *Estensões Do Metamodelo Kdm Para Apoiar Modernizações Orientadas a Aspectos De Sistemas Legados.* MSc Dissertation, UNIVERSIDADE FEDERAL DE SÃO CARLOS CENTRO, 2014.

Santos, B. M.; de Guzmán, I. G.-R.; de Camargo, V. V.; Piattini, M.; Ebert, C. Software refactoring for system modernization. *IEEE Software*, v. 35, n. 6, p. 62–67, 2018.

Santos, B. M.; Landi, A. d. S.; Santibanez, D. S.; Durelli, R. S.; de Camargo, V. V. Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations. *Journal of Systems and Software*, v. 149, p. 285–304, 2019a.

Santos, B. M.; de Souza Landi, A.; de Guzmán, I. G.-R.; Piattini, M.; de Camargo, V. V. Towards a reference architecture for adm-based modernization tools. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, New York, NY, USA: Association for Computing Machinery, p. 114–123, 2019b (*SBES 2019*, v.1). Disponível em `https://doi.org/10.1145/3350768.3350792`

Santos, B. M.; de Souza Landi, A.; de Guzmán, I. G.-R.; Piattini, M.; de Camargo, V. V. Towards a reference architecture for adm-based modernization tools. In: *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, New York, NY, USA: Association for Computing Machinery, p. 114–123, 2019c (*SBES 2019*, v.1). Disponível em `https://doi.org/10.1145/3350768.3350792`

Santos, J. F. M.; Guessi, M.; Galster, M.; Feitosa, D.; Nakagawa, E. Y. A checklist for evaluation of reference architectures of embedded systems (s). In: *International Conference on Software Engineering & Knowledge Engineering - SEKE*, p. 1–6, 2013.

Shull, F.; Singer, J.; Sjøberg, D. I. *Guide to advanced empirical software engineering.* Berlin, Heidelberg: Springer-Verlag, 2007.

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

Son, H. S.; Kim, R. Y. C. A method of handling metamodel based on xml database for sw visulization. In: *2016 International Conference on Platform Technology and Service (PlatCon)*, p. 1–3, 2016.

StandishGroup. *The standish group report - chaos summary 2014.* 2014. Disponível em `https://www.projectsmart.co.uk/white-papers/chaos-report.pdf`

Tummers, J.; Tobi, H.; Catal, C.; Tekinerdogan, B. Designing a reference architecture for health information systems. *BMC Medical Informatics and Decision Making*, v. 21, n. 1, p. 1–14, 2021.

Ulrich, W. M.; Newcomb, P. *Information systems transformation: Architecture-driven modernization case studies.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010a.

Ulrich, W. M.; Newcomb, P. H., eds. . The MK/OMG Press. Boston: Morgan Kaufmann, 419 - 429 p., 2010b.
Disponível em `http://www.sciencedirect.com/science/article/pii/B9780123749130000251`

Van Der Straeten, R.; Jonckers, V.; Mens, T. A formal approach to model refactoring and model refinement. *Software & Systems Modeling*, v. 6, n. 2, p. 139–162, 2007.
Disponível em `https://doi.org/10.1007/s10270-006-0025-9`

de la Vara, J. L. Current and necessary insights into sacm: An analysis based on past publications. In: *2014 IEEE 7th International Workshop on Requirements Engineering and Law (RELAW)*, IEEE, p. 10–13, 2014.

Wolfart, D.; Assunção, W. K.; da Silva, I. F.; Domingos, D. C.; Schmeing, E.; Villaca, G. L. D.; Paza, D. d. N. Modernizing legacy systems with microservices: A roadmap. In: *Evaluation and Assessment in Software Engineering*, p. 149–159, 2021.

Wu, H. *A reference architecture for adaptive hypermedia applications.* Technische Universiteit Eindhoven, 2002.

Zimmermann, H. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, v. 28, n. 4, p. 425–432, 1980.

Appendix

A

Details of Systematic Mapping on ADM

A.1 Final selected papers:

A.1.1 Group 1: Discussions around ADM papers

1. Arcelli Fontana, Francesca, Claudia Raibulet, and Marco Zanoni.“Alternatives to the knowledge discovery metamodel: An investigation.” International Journal of Software Engineering and Knowledge Engineering 27.07 (2017): 1097-1128.
2. Durak, Umut. “Extending the Knowledge Discovery Metamodel for architecture-driven simulation modernization.” Simulation 91.12 (2015): 1052-1067.
3. de la Vara, Jose Luis. “Current and necessary insights into SACM: An analysis based on past publications.” 2014 IEEE 7th International Workshop on Requirements Engineering and Law (RELAW). IEEE, 2014.
4. Jácome, Santiago, and Juan De Lara. “Controlling meta-model extensibility in model-driven engineering.” IEEE Access 6 (2018): 19923-19939.
5. Santos, Bruno M., et al. “Software refactoring for system modernization.” IEEE Software 35.6 (2018): 62-67.

6. Muram, Faiz UL, Barbara Gallina, and Laura Gómez Rodríguez. "Preventing omission of key evidence fallacy in process-based argumentations." 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC). IEEE, 2018.

7. Sabiri, K., and F. Benabbou. "A Legacy Application Meta-model for Modernization." Proceedings of the 2nd international Conference on Big Data, Cloud and Applications. 2017.

8. Durelli, Rafael S., et al. "A mapping study on architecture-driven modernization." Proceedings of the 2014 IEEE 15th international conference on information reuse and integration (IEEE IRI 2014). IEEE, 2014.

9. Pires, Joao Paulo, and Fernando Brito e Abreu. "Knowledge discovery metamodel-based unit test cases generation." 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2018.

10. Sabiri, Khadija, et al. "A survey of cloud migration methods: a comparison and proposition." International Journal of Advanced Computer Science and Applications 7.5 (2016): 598-604.

11. Santos, Bruno M., et al. "Evaluating the extension mechanisms of the knowledge discovery metamodel for aspect-oriented modernizations." Journal of Systems and Software 149 (2019): 285-304.

12. de La Vara, Jose Luis, et al. "An analysis of safety evidence management with the Structured Assurance Case Metamodel." Computer Standards Interfaces 50 (2017): 179-198.

13. Alawneh, Luay, and Abdelwahab Hamou-Lhadj. "Execution traces: A new domain that requires the creation of a standard metamodel." International Conference on Advanced Software Engineering and Its Applications. Springer, Berlin, Heidelberg, 2009.

14. Mansurov, Nikolai, and Djenana Campara. "Managed architecture of existing code as a practical transition towards MDA." International Conference on the Unified Modeling Language. Springer, Berlin, Heidelberg, 2004.

A.1.2 Group 2 - Modernization Tools papers

1. Durelli, Rafael S., et al. "Improving the structure of KDM instances via refactorings: An experimental study using KDM-RE." Proceedings of the 31st Brazilian Symposium on Software Engineering. 2017.

2. Chagas, Fernando, et al. "KDM as the Underlying Metamodel in Architecture-Conformance Checking." Proceedings of the 30th Brazilian Symposium on Software Engineering. 2016.

3. Landi, André de S., et al. "Supporting the specification and serialization of planned architectures in architecture-driven modernization context." 2017 IEEE 41st annual computer software and applications conference (COMPSAC). Vol. 1. IEEE, 2017.

4. Márquez, Luis, et al. "A framework for secure migration processes of legacy systems to the cloud." International Conference on Advanced Information Systems Engineering. Springer, Cham, 2015.

5. Durak, Umut. "Pragmatic model transformations for refactoring in Scilab/Xcos." International Journal of Modeling, Simulation, and Scientific Computing 7.01 (2016): 1541004.

6. Gotti, Zineb, et al. "NooJ App Optimization." International Conference on Automatic Processing of Natural-Language Electronic Texts with NooJ. Springer, Cham, 2018.

7. Trias, Feliu, et al. "Migrating traditional web applications to CMS-based web applications." Electronic Notes in Theoretical Computer Science 314 (2015): 23-44.

8. de Lima Mariano, Thiago, et al. "A Parser and a Software Visualization Environment to Support the Comprehension of MATLAB/Octave Programs." ICEIS (2). 2018.

9. Lavazza, Luigi. "Automated function points: Critical evaluation and discussion." 2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics. IEEE, 2015.

10. Khamal, Adil, et al. "An approach based on ADM for the generation of a meta-model modernized for LMS platforms." 2014 International Conference on Next Generation Networks and Services (NGNS). IEEE, 2014.

11. Sabiri, Khadija, et al. "Towards a cloud migration framework." 2015 Third World Conference on Complex Systems (WCCS). IEEE, 2015.

12. Grimmer, Andreas, et al. "Supporting program analysis for non-mainstream languages: experiences and lessons learned." 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

13. Ovchinnikova, Viktoria, and Erika Asnina. "The algorithm of transformation from UML sequence diagrams to the Topological Functioning Model." 2015 International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). IEEE, 2015.

14. Fleck, Günter, et al. "Experience report on building astm based tools for multi-language reverse engineering." 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). Vol. 1. IEEE, 2016.

15. Angulo, Guisella, et al. "An approach for creating kdm2psm transformation engines in adm context: The rute-k2j case." Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse. 2018.

16. Mamouni, Abdelaziz, and Abdelaziz Marzak. "ZCP Modernization by Recovering ZCM Models from Existing Platforms." Proceedings of the International Conference on Compute and Data Analysis. 2017.

17. López-Sanz, Marcos, et al. "Modernization of Information Systems at Red. es: An Approach Based on Gap Analysis and ADM." International Conference on Service-Oriented Computing. Springer, Cham, 2017.

18. Tu, Zhiying, Gregory Zacharewicz, and David Chen. "Building a high-level architecture federated interoperable framework from legacy information systems." International Journal of Computer Integrated Manufacturing 27.4 (2014): 313-332.

19. Martinez, Liliana, Claudia Pereira, and Liliana Favre. "Migrating c/c++ software to mobile platforms in the adm context." (2017).

20. Son, Hyun Seung, and R. Young Chul Kim. "A Method of Handling Metamodel Based on XML Database for SW Visulization." 2016 International Conference on Platform Technology and Service (PlatCon). IEEE, 2016.

21. Ellison, Martyn, Radu Calinescu, and Richard F. Paige. "Towards Platform Independent Database Modelling in Enterprise Systems." Federation of International Conferences on Software Technologies: Applications and Foundations. Springer, Cham, 2016.

22. Santibáñez, Daniel San Martín, Rafael Serapilha Durelli, and Valter Vieira de Camargo. "A combined approach for concern identification in KDM models." Journal of the Brazilian Computer Society 21.1 (2015): 1-20.

23. Rabelo, Luiz A. Pacini, et al. "An approach to business process recovery from source code." 2015 12th International Conference on Information Technology-New Generations. IEEE, 2015.

24. Durelli, Rafael S., et al. "Towards a refactoring catalogue for knowledge discovery metamodel." Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014). IEEE, 2014.

25. Normantas, Kestutis, and Olegas Vasilecas. "Extracting term units and fact units from existing databases using the Knowledge Discovery Metamodel." Journal of information science 40.4 (2014): 413-425.

26. Bagnato, Alessandra, and Jérôme Rocheteau. "Towards green metrics integration in the MEASURE platform." MeGSuS@ ESEM. 2018.

27. Dahab, Sarah A., Stephane Maag, and Xiaoping Che. "A software measurement framework guided by support vector machines." 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA). IEEE, 2017.

28. Martinez, Liliana, Claudia Pereira, and Liliana Favre. "Recovering sequence diagrams from object-oriented code: An ADM approach." 2014 9th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE). IEEE, 2014.

29. Limyr, Andreas, et al. "Semaphore–a model-based semantic mapping framework." International Conference on Business Process Management. Springer, Berlin, Heidelberg, 2006.

30. Reus, Thijs, Hans Geers, and Arie van Deursen. "Harvesting software systems for MDA-based reengineering." European Conference on Model Driven Architecture-Foundations and Applications. Springer, Berlin, Heidelberg, 2006.

31. Sadovykh, Andrey, et al. "REMICS-REuse and Migration of legacy applications to Interoperable Cloud Services." European Conference on a Service-Based Internet. Springer, Berlin, Heidelberg, 2011.

32. Blanco, Carlos, et al. “Towards a modernization process for Secure Data Warehouses.” International Conference on Data Warehousing and Knowledge Discovery. Springer, Berlin, Heidelberg, 2009.

33. Rástočný, Karol, and Andrej Mlynčár. “Automated change propagation from source code to sequence diagrams.” International Conference on Current Trends in Theory and Practice of Informatics. Edizioni della Normale, Cham, 2018.

34. Sadovykh, Andrey, et al. “On study results: round trip engineering of space systems.” European conference on model driven architecture-foundations and applications. Springer, Berlin, Heidelberg, 2009.

35. Mazón, Jose-Norberto, and Juan Trujillo. “A model driven modernization approach for automatically deriving multidimensional models in data warehouses.” International Conference on Conceptual Modeling. Springer, Berlin, Heidelberg, 2007.

A.2 ADM Search String:

***Some synonyms we added as keywords**
Digital libraries used in the SM from 2014: ACM, Engineering Village, IEEE Explorer, Scopus and Web of Science (From 2014 to 2018)*
("Abstract Syntax Tree Metamodel" OR "Abstract Syntax Tree Meta-model" OR "Architecture-Driven Modernization" OR "Architecture Driven Modernization" OR "Model Driven Modernization" OR "Model-Driven Modernization" OR "Model-driven software modernization" OR "Knowledge Discovery Metamodel" OR "KDM Metamodel" OR "KDM Meta-model" OR "KDM standard" OR "Knowledge Discovery Meta-model" OR "Knowledge-Discovery Metamodel" OR "Knowledge-Discovery Meta-model" OR "Structured Metrics Metamodel" OR "Software Metrics Metamodel" OR "Software Metrics Meta-model" OR "Structured Metrics Metamodel") OR (ADM OR OMG) AND (KDM OR ASTM OR SMM)
***New modernization standards creation**
Digital libraries included in the SM: ACM, Engineering Village, IEEE Explorer, Scopus and Web of Science (From 2014 to 2018)*
("Automated Enhancement Points" OR "Automated Function Points" OR "Structured Assurance Case Metamodel" OR "Structured Patterns Metamodel Standard") OR (ADM OR OMG) AND (SACM OR SPMS OR AEP OR AFP)
Digital libraries included in the new SM: Science Direct and Springer (From 2003 to 2018)
("Abstract Syntax Tree Metamodel") OR "Abstract Syntax Tree Meta-model" OR "Architecture-Driven Modernization" OR "Architecture Driven Modernization" OR "Model Driven Modernization" OR "Model-Driven Modernization" OR "Model-driven software modernization" OR "Knowledge Discovery Metamodel" OR "KDM Metamodel" OR "KDM Meta-model" OR "KDM standard" OR "Knowledge Discovery Meta-model" OR "Knowledge-Discovery Metamodel" OR "Knowledge-Discovery Meta-model" OR "Structured Metrics Metamodel" OR "Software Metrics Metamodel" OR "Software Metrics Meta-model" OR "Structured Metrics Metamodel" OR "Automated Enhancement Points" OR "Automated Function Points" OR "Structured Assurance Case Metamodel" OR "Structured Patterns Metamodel Standard" OR (ADM OR OMG) AND (KDM OR ASTM OR SMM OR SACM OR SPMS OR AEP OR AFP)
Since acronyms usually returns wider results we had to link them in this new and separated string.

Appendix

B

Details of Systematic Mapping on Code and Model Refactoring Tools

B.1 Final selected papers:

1. Khan, Muhammad Uzair, Muhamamd Zohaib Iqbal, and Shaukat Ali. “A heuristic-based approach to refactor crosscutting behaviors in uml state machines.” 2014 IEEE International Conference on Software Maintenance and Evolution. IEEE, 2014.

2. Sherwany, Amanj, Nosheen Zaza, and Nathaniel Nystrom. “A refactoring library for scala compiler extensions.” International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 2015.

3. Punt, Leonard, Sjoerd Visscher, and Vadim Zaytsev. “A Tool for Detecting and Refactoring the A? B* A Pattern in CSS.” 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2016.

4. Dotzler, Georg, Ronald Veldema, and Michael Philippsen. “Annotation support for generic patches.” 2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE). IEEE, 2012.

5. Moghadam, Iman Hemati, and Mel Ó. Cinnéide. "Automated refactoring using design differencing." 2012 16th European Conference on Software Maintenance and Reengineering. IEEE, 2012.

6. Tanhaei, Mohammad, Jafar Habibi, and Seyed-Hassan Mirian-Hosseinabadi. "Automating feature model refactoring: A model transformation approach." Information and Software Technology 80 (2016): 138-157.

7. Pretschner, Alexander, and Wolfgang Prenninger. "Computing refactorings of behavior models." International Conference on Model Driven Engineering Languages and Systems. Springer, Berlin, Heidelberg, 2005.

8. Fernández-Ropero, María, Ricardo Pérez-Castillo, and Mario Piattini. "Improving Business Process Model after Reverse Engineering." International Conference on Evaluation of Novel Approaches to Software Engineering. Springer, Berlin, Heidelberg, 2013.

9. Ghaith, Shadi, and Mel Ó Cinnéide. "Improving software security using search-based refactoring." International Symposium on Search Based Software Engineering. Springer, Berlin, Heidelberg, 2012.

10. Chen, Nicholas, and Ralph E. Johnson. "JFlow: Practical refactorings for flow-based parallelism." 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013.

11. Cruz, Luis, Rui Abreu, and Jean-Noël Rouvignac. "Leafactor: Improving energy efficiency of android apps via automatic refactoring." 2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, 2017.

12. Schuster, Christopher, Tim Disney, and Cormac Flanagan. "Macrofication: Refactoring by reverse macro expansion." European Symposium on Programming. Springer, Berlin, Heidelberg, 2016.

13. Ghannem, Adnane, Ghizlane El Boussaidi, and Marouane Kessentini. "Model refactoring using interactive genetic algorithm." International Symposium on Search Based Software Engineering. Springer, Berlin, Heidelberg, 2013.

14. Winetzhammer, Sabine, and Bernhard Westfechtel. "Model refactorings for and with graph transformation rules." International Conference on Software Technologies. Springer, Cham, 2014.

15. König, Harald, Michael Löwe, and Christoph Schulz. "Model transformation and induced instance migration: a universal framework." Brazilian Symposium on Formal Methods. Springer, Berlin, Heidelberg, 2011.

16. Hamioud, Sohaib, and Fadila Atil. "Model-driven java code refactoring." Computer Science and Information Systems 12.2 (2015): 375-403.

17. Kimura, Shuhei, et al. "Move code refactoring with dynamic analysis." 2012 28th IEEE International Conference on Software Maintenance (ICSM). IEEE, 2012.

18. Winetzhammer, Sabine, and Bernhard Westfechtel. "Propagating model refactorings to graph transformation rules." 2014 9th International Conference on Software Paradigm Trends (ICSOFT-PT). IEEE, 2014.

19. Zeng, Kaiping, and Sorin A. Huss. "RAMS: a VHDL-AMS code refactoring tool supporting high level analog synthesis." IEEE Computer Society Annual Symposium on VLSI: New Frontiers in VLSI Design (ISVLSI'05). IEEE, 2005.

20. Zeiss, Benjamin, et al. "Refactoring and metrics for TTCN-3 test suites." International Workshop on System Analysis and Modeling. Springer, Berlin, Heidelberg, 2006.

21. Garrido, Alejandra, and Ralph Johnson. "Refactoring C with conditional compilation." 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.. IEEE, 2003.

22. Marković, Slaviša, and Thomas Baar. "Refactoring OCL annotated UML class diagrams." Software & Systems Modeling 7.1 (2008): 25-47.

23. Romanovsky, Konstantin, Dmitry Koznov, and Leonid Minchin. "Refactoring the documentation of software product lines." IFIP Central and East European Conference on Software Engineering Techniques. Springer, Berlin, Heidelberg, 2008.

24. Rüegg, Michael, and Peter Sommerlad. "Refactoring towards seams in c++." 2012 7th International Workshop on Automation of Software Test (AST). IEEE, 2012.

25. Rodríguez-Gracia, Diego, et al. "Runtime adaptation of architectural models: an approach for adapting user interfaces." International Conference on Model and Data Engineering. Springer, Berlin, Heidelberg, 2012.

26. Luecke, Kenn R., et al. "Software code base conversions." 2007 IEEE/AIAA 26th Digital Avionics Systems Conference. IEEE, 2007.

27. Straeten, Ragnhild Van Der, Viviane Jonckers, and Tom Mens. “Supporting model refactorings through behaviour inheritance consistencies.” International Conference on the Unified Modeling Language. Springer, Berlin, Heidelberg, 2004.

28. Tamrawi, Ahmed, et al. “SYMake: a build code analysis and refactoring tool for makefiles.” 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2012.

29. Wimmer, Manuel, Nathalie Moreno, and Antonio Vallecillo. “Systematic evolution of WebML models by coupled transformations.” International Conference on Web Engineering. Springer, Berlin, Heidelberg, 2012.

30. Giese, Holger, and Leen Lambers. “Towards automatic verification of behavior preservation for model transformation via invariant checking.” International Conference on Graph Transformation. Springer, Berlin, Heidelberg, 2012.

Appendix

C

Taxonomy's Evaluation Strategy

Taxonomy's Evaluation Strategy

Bruno Santos

February 24, 2022

1 Methodology

We intent to perform the focus group evaluation presented in Figure 1. The following step is to perform the delphi method evaluation presented in Figure 2.

The focus group are going to provide a qualitative evaluation to the taxonomy and ,in summary, the process will consists in a set of questions about the taxonomy to a subject group composed by industry software engineers and graduate students.

The discussion in this focus group intends to refine the technical report in order to prepare it to be evaluated by the delphi method.

The delphi method, in summary, intends to evaluate the taxonomy using a questionnaire containing qualitative and quantitative questions that are going to be evaluated by industry software engineers, graduate students and professors of software engineer/software architecture.

The next section presents a draft of the methodology we intend to apply. We followed the guidelines proposed in [1] (Full paper included as an attachment in the e-mail).

2 Focus Group – Methodology Draft

Figure 1 presents a five stage methodology to perform a focus group validation that is a qualitative approach. Stage 1 is responsible for defining the purpose of the validation. Stage 2 is responsible for defining the methodology to be applied in stage 3, which is the stage where the subjects along with the focus group team (team responsible for the conduction of the study)

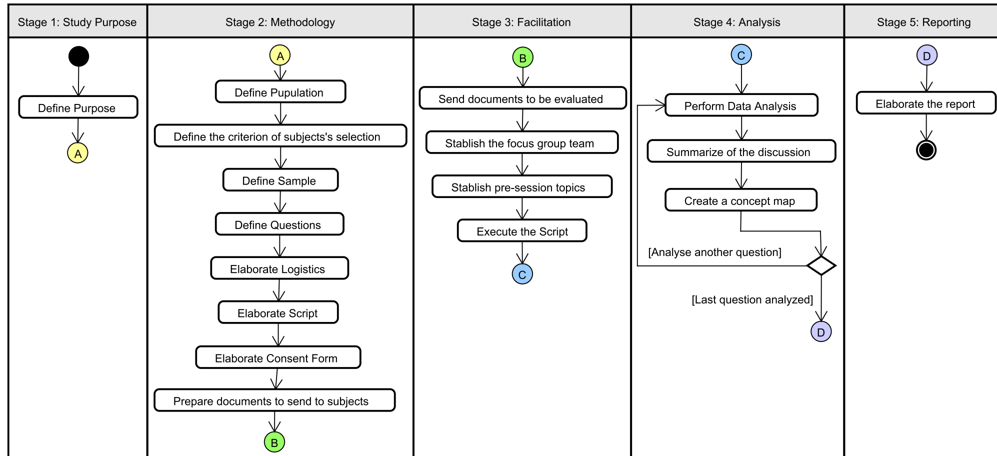

Figure 1: Qualitative Evaluation - Focus Group.

discuss about the taxonomy. Stage 4 is where the focus group team perform the analysis of the data collected in stage 3 in order to elaborate a report in stage 5.

The following sections presents additional information of the stages presented in Figure 1.

2.1 Study Purpose

The purpose of this is study is to perform an **evaluation** of a taxonomy for classifying modernization tools.

2.2 Methodology

2.2.1 Population:

- Software engineers that could use the taxonomy to classify modernization tools. These professionals could also use the taxonomy to communicate while talking about modernization tools. Another usage is to design in a clearer way the modernization tools architecture.
- Graduation Researchers that could use the taxonomy to contextualize, to understand and to develop new modernization tools.

2.2.2 Subject's criterion selection:

- Knowledge in Model-Driven Architecture;
- Knowledge in Reengineering;
- Minimum with master's degree or at least one year of course.

2.2.3 Sample:

- 3 Software engineers
- 3 Graduate students
- Is is possible to have a group in spain?

2.2.4 Questions

- What do you think of a taxonomy for classifying modernization tools ?
- Do you think the way the taxonomy is presented (as a class diagram) assist in its use ?
- What would you change in the proposed taxonomy?
- Do you think that the concepts presented in the taxonomy are enough for representing modernization tools? Why?
- What were the main challenges while reading the taxonomy's technical report?
- ???

2.2.5 Elaborate Logistics

To be described

2.2.6 Elaborate the Script

To be described

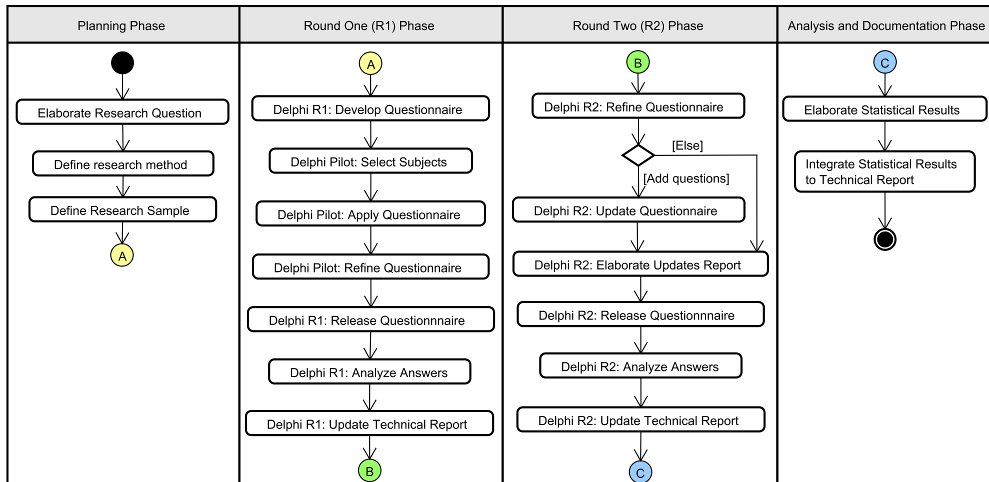

Figure 2: Quantitative and Qualitative Evaluation - Two Round Delphi Method.

2.2.7 Elaborate Consent Form

To be described

2.3 Facilitation

To be described

2.4 Analysis

To be described

2.5 Reporting

To be described.

References

[1] Barry Nagle and Nichelle Williams. Methodology brief: Introduction to focus groups. *Center for Assessment, Planning and Accountability*, (1-12), 2013.