



Pós-Graduação em Ciência da Computação

“A Cloud Deployed Repository for a Multi-View Component-Based Modeling CASE Tool”

Por

Breno Batista Machado

Dissertação de Mestrado



Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, Agosto/2009

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

Breno Batista Machado

A Cloud Deployed Repository for a Multi-View Component-Based Modeling CASE Tool

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

A MASTER THESIS PRESENTED TO THE FEDERAL UNIVERSITY OF PERNAMBUCO IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF M.SC. IN COMPUTER SCIENCE.

ADVISOR:

Prof. Jacques Pierre Louis Robin

RECIFE, 31 de AGOSTO de 2009

Machado, Breno Batista

**A cloud deployed repository for a multi-view
component-based modeling CASE tool / Breno Batista
Machado. - Recife: O Autor, 2009.**

xvii, 141 folhas : il., fig., tab.

**Dissertação (mestrado) – Universidade Federal de
Pernambuco. CIN. Ciência da Computação, 2009.**

Inclui bibliografia, glossário e apêndice.

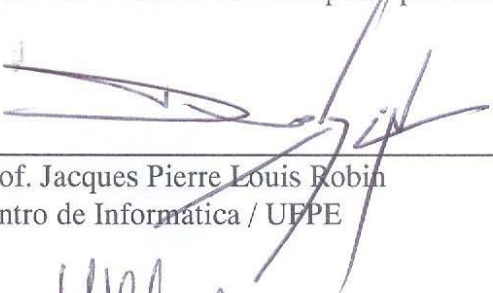
**1. Engenharia de software. 2. Arquitetura de software.
3. MDA. I. Título.**

005.1

CDD (22. ed.)

MEI2010 – 057

Dissertação de Mestrado apresentada por **Breno Batista Machado** Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**A Cloud Deployed Repository for a Multi-View Component-Based Modeling CASE Too**", orientada pelo **Prof. Jacques Pierre Louis Robin** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Jacques Pierre Louis Robin
Centro de Informática / UFPE

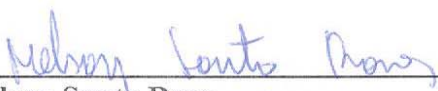


Prof. Hendrik Teixeira Ramos
Depto. de Ciência da Computação e Estatística / UFS



Profa. Ana Cristina Rouiller
Departamento de Estatística e Informática / UFRPE

Visto e permitida a impressão.
Recife, 31 de agosto de 2009.



Prof. Nelson Souto Rosa

Vice-Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Aos meus pais, meus exemplos.

ACKNOWLEDGEMENTS

Agradeço a Deus.

Agradeço também a todos, que de uma forma ou de outra, me ajudaram nesta longa jornada:

Ao meu orientador Jacques Robin, pela orientação, paciência, compreensão e por sempre acreditar em mim.

Ao Weslei que me ajudou a desenvolver este trabalho, sempre compartilhando e discutindo idéias.

À minha namorada Renata, por toda ajuda e apoio, e por seu imenso amor, sem os quais não conseguiria terminar.

Aos meus pais, Benicio e Neumann, e minha irmãzinha querida, Bruninha, que sempre acreditaram em mim e sempre me apoiaram, sempre foram meu porto-seguro.

À toda a minha família por todo apoio que me deram.

Em especial aos meus amigos Cleyton, Fábio e Geovane, que me ajudaram nos momentos finais da escrita da dissertação.

Ao pessoal da SWFactory pelo incentivo sempre.

À todos os meus amigos, da Casa do Mar, de Campo do Meio, de Lavras, de Belo Horizonte, de Recife, e de onde mais se espalharam.

Aos membros da banca, Hendrik e Ana Cristina, que se dispuseram a avaliar este trabalho, e principalmente pelas ótimas contribuições que fizeram para melhorar a qualidade deste trabalho.

À equipe do CIn/UFPE pela infraestrutura e ajudas que foram necessárias.

Ao CNPq, pelo apoio financeiro para o desenvolvimento deste trabalho.

Por final, a todos aqueles que não citei, mas sabem que me ajudaram.

RESUMO

Modelos oferecem abstrações de um sistema que possibilitam aos engenheiros raciocinarem sobre o sistema se focando apenas nos aspectos relevantes, ignorando detalhes que não são relevantes. UML se tornou um padrão de fato para análise e projeto de sistemas, mas possui algumas limitações óbvias: (1) o conjunto de elementos é muito heterogêneo e grande, e (2) o suporte de ferramentas não é satisfatório. Se faz necessário um sistema de regras que governem o processo de análise e projeto, UML é geral demais.

Desenvolvido pela UFPE em conjunto com a Universidade de Mannheim, o objetivo do método KobrA2 é resolver essas limitações através da incorporação de visões especiais de layout, navegação e comportamento de componentes da GUI, e pela introdução do conceito de engenharia de software ortográfica, na qual a construção de Modelos Independentes de Plataforma (PIM) para cada componente de software é realizado em pequenas partes através da construção ortogonal de visões específicas para cada preocupação do componente. Estas visões são então integradas dentro de um Modelo Unificado (SUM) que por sua vez verifica a conformidade com os artefatos do meta-modelo de KobrA2. Para gerar ganhos de produtividade, esta integração e verificação deve ser automaticamente implementada através da transformação de modelos interna a uma ferramenta CASE. Conseqüentemente, para ter sucesso, KobrA2 precisa de uma ferramenta que dê suporte ao seu processo de engenharia de sistemas.

Esta dissertação de mestrado é parte do projeto WAKAME (Wep App KobrA2 Modeling Environment) que tem por objetivo a construção desta ferramenta CASE. Além de ser a primeira ferramenta CASE dirigida por processo e que dá suporte a um ambiente OO, ortográfico, dirigido por modelos e baseado em componentes, para engenharia de aplicações, incluindo a construção de GUI PIMs, WAKAME também procura inovar por ser (a) distribuída em uma plataforma de cloud computing e acessível universalmente através de qualquer navegador Web, (b) ser de muito fácil aprendizagem graças a sua GUI minimalista, com poucos ícones, no estilo do Google, e (c) de uso eficiente graças ao seu cliente projetado para ser leve e com pouco uso de memória e que forneça um esquema de

navegação multidimensional, ortográfico e independente de plataforma, entre visões de baixa granularidade, específica a preocupações, e locais, de um componente.

Dentro do projeto WAKAME, esta dissertação de mestrado investiga três principais questões em aberto. A primeira é o projeto do KWAF (KobrA2 Web App Framework), um modelo de ambiente genérico e independente de plataformas para ser instanciado para o projeto específico de aplicações Web. O segundo é um PIM detalhado do WAKAME como um exemplo de instanciação do KWAF. O terceiro é a implementação e a avaliação de serviços baseados na nuvem que (a) persistentemente armazenem as visões PIM e ortográficas de componentes KobrA2, (b) integre estas visões em um SUM persistente, (c) verifique a conformidade do SUM com o meta-modelo de artefatos de KobrA2, (d) use as restrições do meta-modelo tanto para propagar mudanças em uma visão para outras relacionadas ou enviar mensagens de aviso específicas de uma visão a respeito da violação de restrições.

As principais contribuições desta dissertação são: (a) o projeto de KWAF, o primeiro estudo de caso para avaliação do processo KobrA2 para aplicações Web ricas com gráficos 2D, (b) o projeto de alto nível do WAKAME como estudo de caso para avaliação de KWAF, (c) a implementação de serviços baseados em nuvem computacional para persistência das visões e integração e validação das visões em um SUM persistente, e (d) a integração e testes destes serviços com a parte cliente do WAKAME que permite a edição ortográfica das visões.

Palavras Chave: MDA, KobrA2, Ferramenta CASE, Nuvem Computacional

ABSTRACT

Models offer abstractions of a system that allows engineers to reason about that system while ignoring details that are not relevant to focus on the most relevant aspects. The Unified Modeling Language (UML) has become a de facto standard for system analysis and design but, in fact, UML has some obvious shortcomings: (1) the set of result types is too large and heterogeneous, leading the user to ambiguous interpretation, and (2) its tool support is not satisfactory. It is necessary a system of rules which governs the analysis and design process, UML is too general.

Developed jointly by Universität Mannheim and UFPE, the Kobra2 method aims to addresses these weaknesses by incorporating special views to model the layout, navigation and behavior of GUI components and introducing the concept of orthographic software engineering in which building a Platform-Independent Model (PIM) for each software component is piecemealy carried out by constructing largely orthogonal concern-specific views of the component. These views are then integrated into a Single Unified Model (SUM) which is in turn verified for conformance to the Kobra2 artifact metamodel. To bring productivity gains, this integration and verification must be automatically implemented by model transformations built in a CASE tool. Therefore, to be successful Kobra2 needs a tool that gives support to its system engineering process.

This Master's Thesis is part of the WAKAME (Web App Kobra Modeling Environment) project that aims at constructing such CASE tool. In addition to be the first CASE tool to be process-driven and to support orthographic, model-driven, component-based OO framework and application engineering, including GUI PIM construction, WAKAME also aims to innovate by being (a) deployed on a cloud computing platform and accessible ubiquitously through any web browser, (b) being very easy to learn thanks to a minimalist, few icons, few actions, Google-style GUI and (c) efficient to use thanks to a thin client, lightweight design and a multi-dimensional navigation scheme among cognitively fine-grained, concern-specific, local, orthographic component PIM views.

Within the WAKAME project, this Master's Thesis investigates three main open questions. The first is the design of KWAF (Kobra Web App Framework), a platform-

independent model of a generic OO framework to be instantiated to design specific web applications. The second is the detailed PIM of WAKAME as an example instantiation of KWAF. The third is the implementation and evaluation of cloud-based services that (a) persistently store orthographic views of Kobra2 component PIM, (b) integrate them into a persistent SUM, (c) verify SUM conformance to the Kobra2 artifact metamodel, (d) uses the metamodel constraints to either propagate changes from one view to related ones or send view-specific warning messages about constraint violations.

The main contributions of the thesis are: (a) the design of KWAF, the first case study to validate the Kobra2 method for rich web applications with 2D graphics, (b) the high-level design of WAKAME as a case study validating KWAF, (c) the implementation of cloud-based services for view persistence, view integration into a persistent SUM and SUM verification and (d) the integration and testing of these services with WAKAME's GUI web client for orthographic view edition.

Keywords: MDA, Kobra2, CASE Tool, Cloud Computing

CONTENTS

Chapter 1 Introduction	1
1.1. Context and Motivation	1
1.2. Research Goals	5
1.3. Research Methodology	6
1.4. Master Thesis Structure	7
Chapter 2 Software Engineering Background.....	9
2.1. Model-Driven Engineering	9
2.1.1 MDE Goals, Principles and Vision	9
2.1.2 Precise Modeling and Metamodeling with OCL2.0.....	12
2.2. Component-Based MDE	12
2.2.1 Component-Based Software Engineering	12
2.2.2 The Kobra1 CBMDE Method	13
2.2.3 The Kobra2 CBMDE Method	19
2.3. Cloud Computing.....	26
2.3.1 Principles and Related Concepts	26
2.3.2 Benefits of Cloud Computing	29
2.3.3 Google App Engine	30
2.3.4 Google App Engine and others cloud computing platform.....	35
2.4. Model-Driven CASE Tools.....	36
2.4.1 MDE CASE Tools	39
2.5. Chapter Remarks	47
Chapter 3 WAKAME Project.....	49
3.1. Long-Term Goals and Principles	49
3.2. KWAF	52
3.2.1 KWAF Framework: Principles, Structure and Case Study	52
3.2.2 Structural models of the KWAF Component.....	54
3.2.3 Structural Models of the Webservice Component	55
3.2.4 Structural Models of GUIClient Component	60
3.2.5 KWAF Assessment Remarks	63
3.3. WAKAME Top-Level PIM	64
3.3.1 WAKAME as an Instance of KWAF	64
3.3.2 The WAKAME Component	66
3.4. Chapter Remarks	69
Chapter 4 The WAKAMEWebService and the Model Repository.....	70
4.1. The KWAF instance Platform Independent Model for the WAKAMEWebService and Model Repository	70
4.1.1 The Component WAKAMEWebService	70

4.1.2 The Component WAKAMEModel	72
4.1.3 The Component PIMManipulationAction	73
4.1.4 The Component ViewManipulationAction.....	75
4.1.5 The Component ModelIOAction	80
4.1.6 The Component OCLEngine	81
4.2. The WAKAMWebService and the Model Repository Implementation	82
4.2.1 Platform Definition.....	83
4.2.2 Model Repository with PyEMOF	83
4.2.3 Model Repository with EMF.....	87
4.2.4 The Kobra2 Metamodel in Ecore	88
4.2.5 Model Repository Generation.....	97
4.2.6 Server Implementation	98
4.2.7 WAKAMWebService Integration with WAKAMEGUI, Tests and Deployment	103
4.3. WAKAME Overall.....	105
Chapter 5 Assessment Experiments with Early Adopters	111
5.1. Experiment Definition	111
5.2. Execution and Analysis of the Case Study	115
5.3. Experiment Findings.....	124
5.4. Chapter Remarks	125
Chapter 6 Conclusion	126
6.1. Contributions.....	126
6.2. WAKAME Server Limitations	127
6.3. Future work on WAKAME Server	127
6.4. Limitations of WAKAME as a whole	128
6.5. Future work on WAKAME as a whole	128
6.6. Concluding Remarks.....	128
Chapter 7 Bibliography	130
Appendix A Survey: WAKAME Evaluation	137

LIST OF FIGURES

Figure 2.1. - Typical MDE Transformation scheme	11
Figure 2.2 - Locality Principle of Kobra	15
Figure 2.3. Komponent Structure.....	17
Figure 2.4 - Multiple Views with SUM.....	21
Figure 2.5 – Kobra2 Views	22
Figure 2.6 - Prototypical Structural Class Service View	23
Figure 2.7 - Prototypical Structural Class Type View.....	23
Figure 2.8 - Prototypical Specification Operational Service View	24
Figure 2.9 – Services offered by Cloud Computing. (Source: Levitt, 2009).....	27
Figure 2.10 – Layers in a computer system in the clouds. (Source: Yousef <i>et al</i> , 2008)	27
Figure 2.11 - Ecore Metamodel. (Source: Ecore, 2009)	40
Figure 2.12 - Example of GUI generated by EMF	42
Figure 2.13 - Framework for Generated Model. (Source: Backvanski and Graff, 2005)	43
Figure 2.14 - Epsilon Architecture. (Adapted from: Kolovos <i>et al</i> , 2008)	45
Figure 2.15 - ATL Module.	46
Figure 2.16 - ALTL Module Element.	46
Figure 2.17 - ALT Input and Output Patterns.	47
Figure 3.1 - The Web Application Photo Album.....	53
Figure 3.2 - Structural Model of the KWAF Component	54
Figure 3.3 - The WebService Component with its sub-components: Service Controller, MVCAction and MVCModel	56
Figure 3.4 - Platform Independent Model of the of the Photo Album application WebService Component.....	56
Figure 3.5 - Platform Specific Model of the Photo Album application WebService component.....	57
Figure 3.6 - The MVCModel Component Structure	59
Figure 3.7 - Entities and data types for the MVCModel component, representing the Model in MVC	59
Figure 3.8 - GUIClient component structure	60
Figure 3.9- PhotoAlbum GUI Navigation Model.....	61
Figure 3.10 - PhotoView Component Structural Model.....	61
Figure 3.11 - GUIController Component Structural Model	62
Figure 3.12 - PhotoAlbumGUIController Component and the Definition of one Action using OCL	63

Figure 3.13 - WAKAME Top-Level	65
Figure 3.14 – KWAF component structure.....	65
Figure 3.15 - WAKAME Specification Structural Class Service View	66
Figure 3.16 - WAKAME Realization Structural Class Service View	67
Figure 3.17 - WAKAME Realization Structural Class Type View	67
Figure 4.1 - WAKAMEWebService Specification Structural Class Service	71
Figure 4.2 - WAKAMEWebService Realization Structural Class Service	71
Figure 4.3 - WAKAMEModel Specification Structural Class Service.....	72
Figure 4.4 - WAKAMEModel Specification Structural Class Type	72
Figure 4.5 - PIMManipulationAction Specification Structural Class Service	73
Figure 4.6 - PIMManipulationAction Specification Operation Service	74
Figure 4.7 - ViewManipulationAction Specification Structural Class Service View.....	75
Figure 4.8 - ViewManipulationAction Realization Structural Class Service	76
Figure 4.9 - ViewManipulationAction Realization Operational Service	77
Figure 4.10 – Transformations Abstractions and a Expression Transformaton example.	78
Figure 4.11 - Transformations Specification Structural Class Service	79
Figure 4.12 - Transformations Specification Operational Service.....	80
Figure 4.13 - ModelIOAction Specification Structural Class Service	80
Figure 4.14 - ModelIOAction Specification Operational Service.....	81
Figure 4.15 - OCLEngine Specification Structural Class Service	82
Figure 4.16 - metaWAKAME Architecture.....	85
Figure 4.17 – metaWAKAME Prototype.....	86
Figure 4.18 - Dependencies between Kobra2 and UML	90
Figure 4.19 - Dependencies between Kobra2 and OCL	91
Figure 4.20 - Dependence between SUM and UML.....	91
Figure 4.21 - Classes Package from Kobra2 metamodel.....	92
Figure 4.22 – OCL Constraint on <i>ComponentClass</i>	92
Figure 4.23 - Kobra2 View Package Nesting.	93
Figure 4.24 - Package Dependencies between Views and SUM.	94
Figure 4.25 - Package dependencies between Views and SUM.....	94
Figure 4.26 - "Subject" component and its View-SUM relationship.	94
Figure 4.27 – Relationship between View and Subject ComponentClass.....	95
Figure 4.28 - Transformation Abstractions.	96
Figure 4.29 - Transformation Abstraction	96
Figure 4.30 – OCL Constraint on <i>View</i>	96
Figure 4.31 - Kobra2 Metamodel Packages in Ecore	97

Figure 4.32 - Process to Generate the Model Repository from Ecore File	98
Figure 4.33 - Package Structure of WAKAMWebService Implementation.	100
Figure 4.34 – Example of Transformation OCL Expression	102
Figure 4.35 - WAKAME Issue Tracker	105
Figure 4.36 - WAKAME Main Page	106
Figure 4.37 - WAKAME Create Model Page	107
Figure 4.38 - WAKAME Import Model Page	107
Figure 4.39 - WAKAME Select Model Page	108
Figure 4.40 - WAKAME Edit Model Page.....	108
Figure 4.41 - WAKAME Modeling Page	110
Figure 5.1 - Educational Level of Participants	116
Figure 5.2 - Familiarity of the Participants with the RSM and WAKAME tools.....	116
Figure 5.3 – Model Completeness Comparison.	117
Figure 5.4 –Availability Related Answers Comparison.....	118
Figure 5.5 - Performance Related Answers Comparison.	119
Figure 5.6 - Ease of Use Related Answers Comparison.....	122
Figure 5.7 - Diagram Layout Control and Legibility Related Answers Comparison.....	123
Figure 5.8 - Standard Compliance Related Answers Comparison.....	124

LIST OF TABLES

Table 2.1- Equivalences between the Ecore and EMOF concrete elements	41
Table 4.1 - Model Repository Metrics	98
Table 4.2 - Transformations Metrics	103
Table 4.3 – WAKAMWebService Implementation Metrics	103
Table 5.1 - Familiarity of Participants with the Tools RSM and WAKAME.	118
Table 5.2 – Assessment Summary.....	125

LIST OF ACRONYMS

API – Application Programming Interface

ATL – ATLAS Transformation Language

AWS – Amazon Web Service

CASE – Computer Aided Software Engineering

CBD – Component Based Development

CBE – Component Based Engineering

CIM – Computer Independent Model

CMOF – Complete MOF

CPU – Central Processing Unit

CSS - Cascading Style Sheets

DSL - Domain-specific language

DSML - Domain-Specific Modeling Languages

ECL - Epsilon Comparison Language

EJB – Enterprise Java Beans

EGL - Epsilon Generation Language

EMC - Epsilon Model Connectivity

EMF – Eclipse Modeling Framework

EML - Epsilon Merging Language

EMOF – Essential MOF

EOL - Epsilon Object Language

ETL – Epsilon Transformation Language

EVL - Epsilon Validation Language

EWL - Epsilon Wizard Language

GAE – Google App Engine

GQL – Google App Engine Query Language

GUI – Graphical User Interface

HTML - Hypertext Markup Language

HTTP – HyperText Transfer Protocol

HTTPs – HyperText Transfer Protocol Secure

IaaS – Infrastructure as a Service

IDE – Integrated Development Environment

JAR – Java Archive

JDO – Java Data Objects

JPA – Java Persistence API

JPEG – Joint Photographic Experts Group

JSP – Java Server Pages

KM3 – Kernel Meta MetaModel

KWAF – Kobra Web App Framework

MARTE – Modeling and Analysis of Real-Time and Embedded Systems

MDA – Model Driven Architecture

MDD – Model Driven Development

MDE – Model Driven Engineering

MDR - Metadata Repository

MDT - Model Development Tools

MOF – Meta Object Facility

MT – Model Transformation

MVC – Model View Controller

OCL – Object Constraint Language

OMG – Object Management Group

OO – Object Oriented

ORCAS – Ontologies and Reasoning Components for Agents and Simulations

PaaS – Platform as a Service

PIM – Platform Independent Model

PNG – Portable Network Graphics

PSM – Platform Specific Model

QVT – Query/View/Transformation

RAM – Random Access Memory

RSM – IBM Rational Software Modeler

SaaS – Software as a Service

SDK – Software Development Kit

SOA – Service Oriented Architecture

SQL – Structured Query Language

SVN - Subversion

SysML – Systems Modeling Language

SUM – Single Unified Model

UML – Unified Modeling Language

URI – Universal Resource Identifier

URL – Universal Resource Locator

VM – Virtual Machine

WAKAME – Web App Kobra Model Engineering

WAR – Web Archive

XML – Extensible Markup Language

XMI – XML Metadata Interchange

XSD – XML Schema Definition

XSL - EXtensible Stylesheet Language

CHAPTER 1

INTRODUCTION

In this chapter we present the context on which this work is present, the motivation for this work, its goals, the used methodology and its structure.

1.1. Context and Motivation

According to (Beydeda and Gruhn, 2005) it's tempting to say that software development is easy, but many issues such as the complete understanding of the problem space, large numbers of restrictions applied on the solutions being considered and many types of changes in all stages of development make it much more challenging. Software Engineering presents many methodologies that searches for addressing these issues in software development. These methodologies can be divided into two main groups (Vinekar *et al*, 2006): the traditional methodologies – like the Rational Unified Process (RUP) (Kruchten, 2003) – and the agile methodologies – where are highlighted SCRUM (Schwaber, 2004) and XP (Beck and Andres, 2004).

The traditional methodologies, like RUP, also known as “heavy weighted” methodologies, are characterized by being subdivided into large well defined phases (Pressman, 2009) and are focused in documentation. The main contributions of these methodologies are: (i) project decomposition in phases largely orthogonal to process steps; (ii) CASE tool assisted construction of digitally stored, abstract requirement, architecture and design models, prior to coding; (iii) use of visual, standard notation for these models (UML); and (iv) allow a high level of customization in their processes and steps. These methodologies are usually applied to large, distributed projects with a heterogeneous development time with a high turnover during the lifecycle. As weaknesses, we can depict the lack of practices and guidance to: software reuse; the specification of which part of the UML2 metamodel to use; the code generation from behavioral models; and how to build test models.

In contrast to the traditional methodologies we have the agile methodologies, also called “lightweight”, which are focused on deliverable artifacts development and cooperation and communication among team members. The eXtreme Programming (XP), one of the agile methodologies, present a set of practices, principles and values where we can highlight as strengths: (i) pair artifact construction; iterative requirement test construction after partial only partial requirement elicitation and before requirement implementation; (ii) focus on artifacts that directly lead to running code; and (iv) continuous integration to tested running prototype. Another agile methodology, which is more focused in management practices, is the SCRUM, from where we can highlight the informal, collective task allocation and time estimates with focus on creating team spirit and maintaining motivation. The common scope of agile methodologies are projects that present a small stable teams of developers with solid experience with application type, implementation platform(s), all software engineering roles with continuous access to known and committed users. However, agile methodologies present drawbacks, such as lack of practices and guidelines for software reuse; deployment platform fragmentation and rapid changes; team turnover, team geographical scattering, rookies; innovative applications with no experienced developers nor even know users; and high-level architectural design of large, complex software and product lines.

In an attempt to overcome the existing limitations present in these two methodology groups, the traditional and agile methodologies, the Model-Driven Engineering (MDE) (Stahl *et al*, 2006) depict as principles: separate business analysis and design refinement from platform-specific coding; build a Platform-Independent Model (PIM) refined enough to serve as input to full code generators, including behavioral code; reuse integrated consolidated comprehensive self-extensible standard modeling language family like Metamodel Object Facility (MOF2) (MOF, 2006), Object Constraint Language (OCL2), Unified Modeling Language (UML2), Diagram Interchange (DI). The MDE application scope is domains with platform heterogeneity, fragmentation and rapid evolution (e.g., mobile and desktop clients, cloud servers). In despite of this attempt to overcome previous limitations MDE is mere an engineering philosophy, no consolidated and only widely adopted with specific methods yet. Other limitations of MDE is the lack of practice and guidelines for reusing functionalities across applications, and how to define for each artifact which part of the huge UML2 metamodel to use.

At the opposite side of the use of a set of standards for MDE, we have the Domain Specific Modeling Languages (DSML) (Luoma *et al*, 2004) that creates specific languages for each kind of domain. A Domain-Specific Language (DSL), whether used for model-driven engineering is a piece of critical infrastructure that is developed during the system engineering process: DSLs show an increased correspondence of languages constructs to domain concepts when contrasted with general purpose languages. As a result, a DSL will more accurately represent domain practices and will more accurately support domain analysis (Kolovos *et al*, 2006). On the other hand, the Unified Modeling Language (UML) has become a de facto standard for system analysis and design but, in fact, UML has some obvious shortcomings (Stutz *et al*, 2002): (1) the set of result types is too large and heterogeneous, leading the user to ambiguous interpretation, and (2) tool support is not satisfactory to DSLs - It is necessary a system of rules which governs the analysis and design process, UML is too general.

In addition, UML is not really a practical modeling language; it is more a modeling language tool-kit that tries to offer many different ways of modeling many different aspects of many different software projects. Because of the redundancy, general-purpose, gigantic size of the UML plus the lack of support from UML tools to customize the interaction experience with the users, some developers adopt the use of a UML profile, not so much to extend UML, but also to restrict UML, that is to say to select few constructs and diagrams relevant for the application domain and system engineering process. Other developers use domain specific languages renouncing all together to UML, and yet others doesn't use modeling practices, renouncing all together to Platform Independent Models (PIM).

Thinking about overcome the weaknesses of the MDE and UML, Atkinson *et al* (2001) developed the Kobra method at Fraunhofer-Gesellschaft Institute in Experimental Software Engineering (IESE), Kaiserlautern, Germany. Kobra integrates MDE with Component-Based Development (CBD) and Product Line Engineering (PLE). Focused on engineering Platform Independent Models (PIM) for reuse and with reuse, this method precisely prescribes what artifacts and sub-artifacts to build at each process step, together with constraints that must hold among them. Kobra was first published in the book "*Component-Based Product Line Engineering in UML*" (Atkinson *et al*, 2001) and it was extended for components test modeling in the European Research Project Component++ project which happened between 2000 and 2003. This project was developed in academic

and industrial environments across ten countries, originating a second book, “*Component-Based Software Testing in UML*” (Gross, 2004). However the Kobra method use an informal mix of UML1.4 diagrams, ad-hoc tables and natural language in its models.

To overcome the problems of the first version of the Kobra method, the Kobra2 method was developed jointly by Universität Mannheim and UFPE, and addresses these weaknesses by integrating in synergy three software reuse approaches: Model-Driven Engineering (MDE), Component-Based Development (CBD) and Object-Oriented (OO) frameworks. Beyond this integration, Kobra2 also innovates in two other ways. First it incorporates special views to model the layout, navigation and behavior of GUI components. Second, it introduces the concept of orthographic software engineering in which building a Platform-Independent Model (PIM) for each software component is piecemeal carried out by constructing largely orthogonal concern-specific views of the component. These views are then integrated into a Single Unified Model (SUM) which is in turn verified for conformance to the Kobra2 artifact metamodel. In comparison with its first version, the Kobra2 method substitutes all informal mix of UML1.4 and natural language to natural language free semi-formal diagrams using UML2 with OCL2 expressions. In addition, Kobra2 defined the concept of Orthographic Software Engineering, defined automatic PIM constraints check mechanisms and view to SUM and SUM to view transformations.

To create Kobra2 models in a productive way, all the integration and verification of models in orthographic modeling defined in the method Kobra2 must be automatically implemented by model transformations built in a CASE tool. So, to be successful Kobra2 needs a tool that gives support to its system engineering process. Current UML and MDE CASE tools lack to support orthographic modeling (i.e., to separate concerns in separate views), and do not offer modeling with metamodel extensions nor do they offer modeling GUI customization. Another limitation of many UML tools is related with the usability problem, leading the users to reject it. Other tools cover only a tiny part of UML, generally not motivated by an underlying methodological choice, but that is unexplained and arbitrary. Yet others do not include an OCL IDE with friendly messages and hence cannot support robust and refined PIM and full behavioral code generation.

In synthesis, the current UML and MDE CASE tools: (1) can't support process-driven customization of modeling experience; (2) do not support orthographic modeling; (3) leads to DSML; (4) are very large programs requiring complex installation process and

consumes a lot of memory; and (5) prevents using them for Kobra2, a comprehensive reuse-oriented process. These problems in existing CASE tools difficult the Kobra2 method usage in a productive way and as a consequence motivated this work.

1.2. Research Goals

Based on the motivation described in Section 1.1, we defined the WAKAME Project, where WAKAME stands for Web App Kobra Model Engineering. The goal of this research project is the development of a web tool, named also WAKAME, which supports the creation of models described in the Kobra2 method. This project initially comprises the work of two masters students, being this master thesis the results of part of the project. Hence, the project scope is divided into:

- Research and development of the WAKAME Server, part of the WAKAME tool. The WAKAME Server is a repository of Kobra models, comprising components for: processing the model, importing and exporting models to XMI and integration with the GUI component; and
- the research and development of the Graphic User Interface (GUI) client for the WAKAME tool, which is the scope of this master thesis. This GUI Client will enable the user to interact with the tool for the creation and manipulation of Kobra2 models. This GUI will also be integrated with the server.

The general goal of this work is to develop and evaluate of the server component of the WAKAME tool, based on cloud computing, that allows (a) the persistence of the orthographic views of Kobra2 component PIM, (b) the integrations of these views into a persistent SUM (Single Unified Model), (c) the conformity verification of SUM in agreement with the Kobra2 artifacts defined into the metamodel, (d) the usage of defined restrictions in the metamodel to spread the changes of a specific view for the another that are related with it, and to send messages for the specific views about the violations of restrictions.

The specific goals of the development of this work are:

- Develop a framework to enable the modeling of a web applications PIM (KWAF) in collaboration with the master's student Weslei Alvim de Tarso marinho and the PhD candidate Fábio Moura Pereira;

- Validate KWAF with a case study through the modeling of a PIM of an simple web application for a Photo Album, trying to identify limitations and possible improvements in the framework;
- Definition of the principles and requirements for a modelling tool for Kobra2;
- Implement a Repository of Models for Kobra2, that is aligned with its metamodel and that executes the transformations defined in Kobra2 for the maintenance of the models consistence;
- Implement the component server of WAKAME on top of a cloud platform computing following modeled PIM and that uses the developed Repository of Models;
- Publish the tool for the Kobra2 community's use.

Through these goals we will validate the theoretical hypothesis that it is possible to design and validate a Kobra2 models repository that implements transformations and constraint check in a cloud computing environment, allowing it to be accessed from a Web GUI client. If this hypothesis demonstrates to be true, the repository will enjoy the advantages of a cloud computing application, like scalability and availability that this platform provides, having all its information processing in the cloud, instead of in the user's hardware. The repository will be published in the Web, allowing more than one GUI client can access its services. With the integration of the repository with a Web GUI client it will be possible to access the application from anywhere, regardless software installation needs. And finally, with the WAKAME tool we will have a productivity gain in Kobra2 modeling, because WAKAME will give all technology support needed for the method.

1.3. Research Methodology

The research methodology of this work is the Action Research (Thiollent, 1986; Kock,2007), which has an empirical base that is designed and carried out in close association with an action or a solution to collective problem. In the Action Research the researchers and participants of the situation or problem are involved in cooperative or participatory mode.

The action research comprises methods that fit in the concept of doing a research and to apply it in a development work, in this case, “research for the action” (Rodrigues,

2004). Here fits the methodology to be used in this work, because it will be done a research of technologies for the solution of a particular problem, in this case the lack of a CASE tool that supports Kobra2 method, and after it, to use one of those technologies in the solution of this problem.

Based on this methodology it has been defined three main phases for this work: (i) gathering of information on subjects related to our problem, (ii) the development of the tool which would solve the problem, (iii) and perform of a case study to assess the tool performance.

- The first phase focused on the finding information through bibliographic review and pragmatic studies about techniques and standards needed for this work development, which comprises MDE, cloud computing and CASE tools.
- The second phase aimed to define an architecture for platform-independent models (PIM) for web applications and validate it through a case study (the PIM of an web PhotoAlbum), develop the PIM for the WAKAME Server, implement the components according to the defined PIM, integrate the Server with the WAKAME GUI and finally perform verification through the use of both integration and system testing.

In the third and final phase, an experiment was conducted as a case study to validate our hypothesis, where the experiment participants were asked to perform a modeling session using the WAKAME tool, and do the same in another MDE tool, the IBM Rational Software Modeler (RSM). This case study comprised four steps: (i) presentation of Kobra2 method for all participants; (ii) present the KWAF framework, which would be modeled by all participants of the case study; (iii) KWAF modeling session in the WAKAME tool; and (iv) KWAF modeling session in RSM tool. With the comparison of models drawn in both tools we could have real results and thus measure the quality and feasibility of WAKAME usage.

1.4. Master Thesis Structure

This master thesis is organized in this introductory chapter and more six chapters:

In the Chapter 2 the basic concepts necessary for the accomplishment of this work related to the Model Driven Engineering (MDE) are presented. Are presented the beginnings and visions of MDE, the patterns used for modeling and MDE, the Component-

Based Engineering (CBE) concepts, and the Kobra2 method, the related concepts to Cloud Computing, and for finally it is presented an analysis and comparison among some of the most important CASE tools for MDE, especially the used for transformations of models and generation of repository of models.

In the Chapter 3 it is present the WAKAME project, exhibiting their principles and objectives. It also presents the architecture for PIM of web applications defined in this work, the KWAF, and it presents the top-level PIM of the WAKAME tool.

In the Chapter 4 the specific modeling of the PIM of the server component of WAKAME is described, how its PIM was implemented, detailing the chosen technologies, the faced problems, the integration with the client component of WAKAME and the accomplished tests.

In the Chapter 5 it is described the case study accomplished for evaluation of the tool developed in this work, as well as the results and conclusions obtained through the use of the same.

In the Chapter 6 it is presented the conclusion and a summary of the contributions of the work, as well as the limitations and future works of the general WAKAME tool, and also the server component in specific, that was the focus of this work.

At the end of this work there are the Annexes and Appendixes, in which additional information referenced along the text are presented.

CHAPTER 2

SOFTWARE ENGINEERING BACKGROUND

In this chapter we describe the key technologies of software engineering we use to develop the components of our proposed application. In particular, we present the ideas of model-driven engineering and component-based development, a set of principles and technologies that provide the structural basis of this thesis.

2.1. Model-Driven Engineering

Model-Driven Engineering (MDE) was put forward by the Object Management Group, a consortium of large companies and academic centers, under the initiative known as Model-Driven Architecture (MDA). MDE proposes essentially to raise the level of abstraction where most of the development effort is spent from source code to models, metamodels and model transformations.

2.1.1 MDE Goals, Principles and Vision

The fundamental long-term vision of MDE is that systems may be specified and realized in a completely refined way in a so called platform independent model (PIM). Then this PIM is translated to platform-specific models, which in turn are translated to source code either manually or by model transformations. The MDE initiative expects several benefits from this shift. Among them are: platform-independent business model reuse, increasing productivity and increasing deployment speed, easier applications maintenance and as a consequence of all three, economic gains in the software life-cycle as a whole.

Model-Driven Engineering pursues two related goals. The first is to minimize the cost of deploying the same functionalities on a variety of platforms, i.e. modeling once and having it deployed many times in different computational environments such as web

services, EJB, .NET etc. The second goal is to automate an ever growing part of the development process required during the life cycle of an application.

To achieve these goals, MDE switches the software engineering focus away from low-level source code towards high-level models, metamodels (i.e., models of modeling languages) and **Model Transformations (MT)** that automatically map one kind of model into another. MDE prescribes the construction of a fully refined **Platform Independent Model (PIM)** together with at least two sets of MT rules to translate the PIM into source code via an intermediate **Platform Specific Model (PSM)**.

In MDE, development starts by constructing a precise model of the application domain, which in MDE terminology is called a **Computational Independent Model (CIM)**. This first artifact defines the common, more abstract entities of the business domain in question, allowing the engineers to understand the big picture in which the future software system will be placed. The CIM must also be described using UML artifacts (such as Use-cases, statechart diagrams etc) but must not include any information about the application realization nor its behavior. A CIM should come fully equipped with traceability links to its further refinements in order to provide (semi) automated translation to the next refined artifacts.

A CIM generates a **Platform Independent Model (PIM)** of the application. This PIM should be a refinement of the software design down to the level of instructions available as built-in by the most widely used platform for a particular domain (e.g., OpenGL or Direct3D methods for computer graphics, EJB or .Net for web information systems). It constitutes the most strategic, persistent, reusable asset of the MDE process.

To ease the automation of such translating task, MDE proposes to divide it in two stages: first from the PIM to a **Platform Specific Model (PSM)** and then from such PSM to source code. The PSM is still a hybrid visual and textual notation but it incorporates concepts that are specific to one target implementation platform. The modeling languages used for the PIM and PSM must be formalized by a metamodel. The translations from PIM to PSM can then be specified as transformation rules from a source metamodel pattern to a corresponding target metamodel pattern. Pattern matching is then used to generate the PSM from the PIM and the code from the PSM.

Nowadays, a MDE CIM, PIM and PSM can be fully specified using the UML2 standard (OMG, 2009) for it incorporates (a) a platform independent component

metamodel, (b) the high level OO functional constraint language OCL2 (OMG, 2009) to fully detail, constraint and query UML2 models, and (c) the *Profile* mechanism to define, in UML2 itself, UML2 extensions with platform specific constructs for diverse PSM. MT for PIM to PIM and PIM to PSM to code translation can be specified using the rule-based, hybrid declarative-procedural **Atlas Transformation Language (ATL)** (ATL, 2009).

2.1.1.1. Automated Model Transformations

In MDE, the generation of deployment artifacts and ultimately a running application is obtained through incremental translation of these models (e.g. PIM to PSM, PSM to code). These translations are encoded as model transformations, which establish the traceability links from abstract models down to the platform-specific deployment level.

A transformation takes as input a source model accompanied with its metamodel (i.e. its specification) and generates a target model again according to a given metamodel (typically target and source metamodels are different, but not necessarily). Figure 2.1 shows this general transformation scheme. In MDE, a typical transformation pipeline would take as input a platform-independent model (and associated metamodel) and generate some PSM, according to a corresponding target metamodel. From the PSM another transformation in the pipeline would generate the final source code.

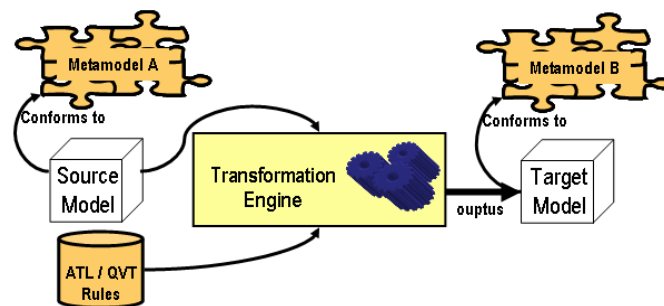


Figure 2.1. - Typical MDE Transformation scheme

A transformation tool must obviously contain a way to express a description of how a model should be transformed, i.e. a transformation specification. In general, a transformation consists of a collection of transformation rules, which are unambiguous specifications of the way that the objects of the source model can be used to create the objects of the target model. A transformation is expressed at metamodel level; in fact a transformation expresses the structural mappings between the source metamodel and the target metamodel. They are however applied and executed on the input models, not the input metamodel to produce the output model (again not the output metamodel).

2.1.2 Precise Modeling and Metamodeling with OCL2.0

The Object Constraint Language (OCL) is a semi-formal specification with a hybrid functional object-oriented syntax. It is used to define constraints and queries over UML models and MOF metamodels. OCL expressions can be used to specify invariants over classes, pre- and pos-conditions over operations execution, derived attributes of a class and finally side-effect free operations as queries over a model. UML modelers can use OCL to specify:

- Arbitrary complex structural constraints among potentially distant elements of an application UML structural diagram or language metamodel; for this purpose OCL has the expressive power of first-order logic, and allows specifying class invariants and derived attributes and associations;
- Arbitrary complex algorithms that combine behavior of class operations or message passing; for this purpose is Turing-complete and allows specifying operations preconditions read-only operation bodies and read-write operations post-conditions.

2.2. Component-Based MDE

A **Component-Based** MDE process (Atkinson *et al*, 2001) structures the PIM, PSM and source code as assemblies of reusable components, each one clearly separating the services interfaces that it provides to and requires from other components from its encapsulated realization of these services (itself potentially a recursive sub-assembly).

2.2.1 Component-Based Software Engineering

While Model-driven Development fosters reuse of application models across platforms, CBD fosters reuse of functionalities across applications. A software component encapsulates a set of basic functionalities whose need recurs in diverse applications. It contains metadata that specifies how to assemble these functionalities with those encapsulated in other components to build more complex functionalities through assembly.

According to (Eriksson *et al*, 2003) “a component is a self-contained unit that encapsulates the state and behavior of a set of classifiers”. All the contents of the components, including its sub-components, are private. A component is always associated to provided and required interfaces. The key feature of CBD is the ability to promote the reuse of software components. The full encapsulation and separation of interface from

implementation enables a component to be a substitutable unit that can be replaced at design time or run time by another component that offers equivalent functionality.

In an assembly, a given component may act as both a server to some component and a client to another component. The assembly structural meta-data of a component includes provided interfaces, the operations (together with their input and output parameter types) that are available by connecting to the server ports of the component. It may also include required interfaces, the operations (together with their input and output parameter types) that the component expects to be available in the deployment environment through connections to its client ports. A component may also include assembly behavioral meta-data that describes the pre- and post-conditions of the operations provided and required at its ports in terms of its states and the states of its clients and servers in the assembly. Such meta-data allows defining a contract between a client-server component pair. Such design by contract permits black-box reuse, which is ideal for leveraging third party software and more cost-effective than the white-box reuse by inheritance in object-oriented frameworks. A component can be substituted at any time by another one that is internally different but respect the same contracts at its ports, without affecting the rest of the software.

2.2.2 The Kobra1 CBMDE Method

The Kobra method integrated Component-based development, MDE and product-line engineering (Atkinson *et al*, 2001), together with standard techniques such as top-down refinement and object-oriented development, in a coherent and comprehensive whole. It is a significant breakthrough in CBMDE because:

- It promotes reuse over the *entire range of development stages*, from requirement to modeling, implementation, testing, quality insurance and maintenance;
- In contrast to previous methods, it provides *precise guidelines for most software engineering aspects*, including a finely grained recursive process, the artifacts to specify at each step, well-formed rules for each of these artifacts as well as for the relations between them and quality metrics and control;

It is *fully platform and tool independent* by relying on the UML standard as to specify all the software artifacts¹.

A fundamental feature of Kobra is the distinction between products (software artifacts) and processes (software engineering tasks). A Kobra project defines the products independently from the processes that might be applied. Many methods mix up the description of what to do with how to do it. This makes difficult to decide what is absolutely necessary to perform and what is optional. This distinction has been adopted as an OMG standard in the Software Process Engineering Metamodel 2.0 (SPEM, 2009).

Other important concerns that Kobra distinguishes are the organization of the method in terms of three orthogonal dimensions: the first dealing with the level of abstraction; the second dealing with the level of genericity; and the third dealing with the level of composition. In a typical Kobra project development begins with a generic, abstract, black-box description of the system. To create an application from this first black-box it is necessary to: i) remove the genericity of the black-box (instantiation), ii) decompose the black-box into smaller parts (decomposition) and iii) reduce the level of abstraction to create an executable version of the system (embodiment).

In Kobra all three dimensions can be dealt with separately. The genericity dimension is tackled by the product line engineering approach; the composition dimension comes under the umbrella of component modeling, and development concerning the abstraction dimension comes under the component embodiment activity. These concerns can be tackled in various orders, and even arbitrarily interwoven (Atkinson *et al*, 2001).

2.2.2.1. Principles

A central goal of Kobra is to enable the full expressive power of the UML to be used in the modeling of components. To this end the use of the UML in Kobra is driven by four basic principles:

Uniformity: Every behavior-rich entity is treated as a Component, and every Component is treated uniformly, regardless of its granularity or location in the containment

¹ Except for code that is beyond the scope of the UML and for which Kobra puts forward the “Normal Object Format”, an abstract language that integrates the semantically common constructs of C++, Java and C# while abstracting from their syntactic differences.

tree. In particular, the system as a whole is viewed and modeled as a Component, albeit a large-grained one. This endows a Component containment tree (and the applications created from them) with the property of a fractal, since the products (and the accompanying processes) are identical at all levels of granularity. It also promotes reuse, because any Component, anywhere in a containment tree, can be made into a system if it satisfies the needs of some customer.

Encapsulation: The description of what a software unit does is separated from the description of how it does it. Encapsulating and hiding the details of how a unit works facilitates a “divide and conquer” approach in which a software unit can be developed independently from its users. This allows new versions of a unit to be interchanged with old versions provided that they do the same thing.

Locality: All descriptive artifacts represent the properties of a Component from a local perspective rather than a global perspective. This means that there are no diagrams, or other descriptive artifacts, that take a global perspective and cover all aspects of the system. Even the largest Component at the root of the containment tree only has a black-box view of its sub-Components, and thus its models only describe its local properties. By reducing the coupling between components this promotes reuse. Figure 2.2 illustrates this principle.

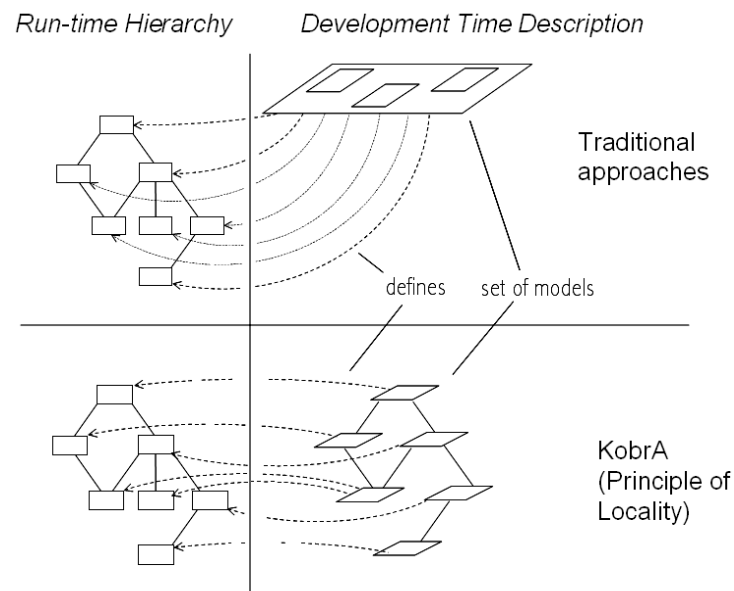


Figure 2.2 - Locality Principle of Kobra

Parsimony: Every descriptive artifact should have "just enough" information, no more and no less. This means that all models and diagrams should contain the minimum

amount of information needed to describe the necessary properties, but no more. Including more model elements than necessary simply obscures the essential information, and complicates the models.

2.2.2.2. Artifacts

In the most abstract level, Kobra prescribes a software product-line approach. A framework as defined by the methodology is a set of generic artifacts that might create several similar applications. There is a decision model associated with each artifact in order to indicate how to instantiate the framework into a specific application. This information is gathered together in the context realization, which is the initial analysis and modeling of the environment in which the systems it is to execute. The Kobra context realization produces the same artifacts as the realization of Komponenten we describe below.

The intermediate level of granularity of Kobra is the component-based paradigm. Applications are organized in terms of hierarchical composition of reusable Kobra logical components (usually referred as Komponent). Each Komponent is built using the same artifacts in a recursive fashion. At development time a containment tree that composes recursively Komponenten and sub-Komponenten until the most basic component level represents a system.

A Komponent is modeled in terms of a specification, which describes what kind of services the Komponent provides for other Komponenten that might be associated to it through a client-server or ownership relation, hence a Komponent specification is a description of the requirements that the Komponent is expected to fulfill. The Komponent Specification artifacts prescribed by Kobra are:

A Structural Model: that defines classes, operations, attributes and the other required Komponenten, and relationships of the structural part of the Komponent.

A Behavioral Model: that specifies via pre and post-conditions in OCL the behavior of each operation of the Komponent. Defines via state charts the internal states of the Komponent, the available services of the Komponent in each state and how the operations trigger the transitions to each state.

A Komponent may also have one realization associated to its specification, which describes precisely how a Komponent implements its specification (partly with the help of server Komponenten). The Komponent Realization artifacts prescribed by Kobra are:

A Structural Model: that defines the classes, operations and attributes and relationships of the structural part of the Komponent.

A Behavioral Model: that defines via OCL expressions, activity and sequence diagrams, the algorithms that realize the operations shown in the specification view.

Figure 2.3 illustrates the two views of a Komponent.

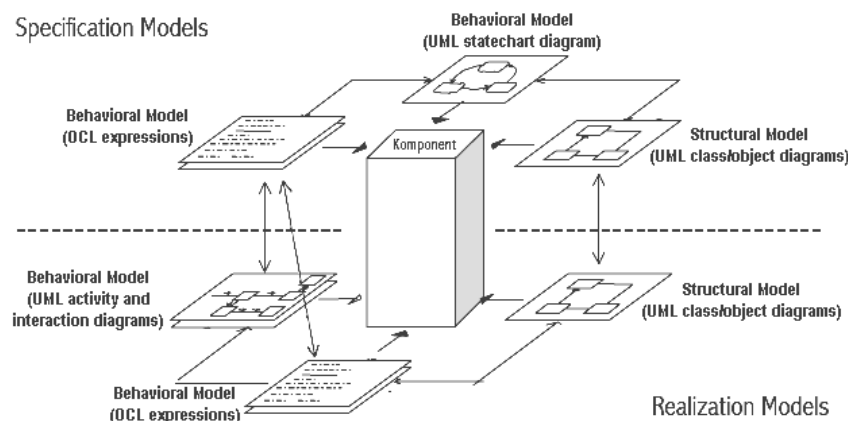


Figure 2.3. Komponent Structure.

Finally, Komponent embodiment is concerned with reducing the abstraction level of the artifacts. There are two basic strategies for achieving a deployed version of a Komponent: developing it from scratch or reuse an existing Komponent that is capable of providing conformant properties. Reuse is of course the less expensive way of realizing a Komponent implementation, but it is not always the case there is available reusable Komponenten. Developing a Komponent from scratch must deliver a tool processable form of a Komponent that can be automatically transformed in executable binary elements. In many development environments this is source code. However Kobra separates the transformation process into distinct refinement and translation steps. Refinement is concerned with describing model elements at a lower level of detail, but using the same notation, while translation is concerned with mapping model elements into program elements. Separating these concerns allows Kobra to be quite conformant to the MDA approach.

2.2.2.3. Process

The shape of the KobrA method is tightly coupled to that of the product, with many activities usually taking the same name as that artifact, which they create. At the highest level of granularity is Framework Engineering that is concerned with the construction of a framework, which captures all the properties of a product-line in a generic way by explicitly highlighting variant features and indication to which variants of the system each feature belongs. Framework engineering involves Komponent modeling and Komponent embodiment.

A sub-activity of Framework engineering is the context realization activity. Its goal is to create the set of artifacts that make up a context realization. It follows the use-case analysis strategy but accommodates genericity to include analysis of variabilities and commonalities in a product line.

The other sub-activities of Framework engineering are the Komponent Specification, which creates the artifacts for the Komponent specification from the previous realization; and the Komponent specification that is based on the same underlying approach as context realization. Data and activities are initially analyzed independently, and are then integrated by focusing on interaction modeling.

Application engineering can involve all the activities of the framework engineering activity but without the parts dealing with the variabilities and commonalities. Instead of this, it includes activities dealing with the instantiation of the framework, namely: i) the context realization instantiation, which analyses if a given generic context realization can be tailored to a specific domain or if it is better to start from scratch; ii) specification and realization instantiation which deals with recursively instantiating the Komponenten within the framework by resolving the decision models and removing the irrelevant features.

2.2.2.4. Strenghts

1. Only comprehensive full life cycle reused oriented software method.
2. Follow a few simple principles and apply them uniformly.
3. Thorough separation of concerns by integrating MDE with CDB and PLE.
4. Fully prescriptive for the UML artifact to produce at each step of the process.
5. Detailed guidelines on how to produce those artifacts.

2.2.2.5. Weaknesses

1. Based on UML 1.4, tables and natural language which prevents the most advanced model-transformation based flavor of MDE.
2. Says nothing specifically for GUI modeling.
3. PLE approach presupposes that framework model contains all artifacts of all possible product instantiation, which is unpractical in many real cases.
4. Focuses on PIM modeling, saying very little on PIM to PSM and PSM to code translation tasks.
5. Not modeled explicitly in SPEM.
6. No capability and maturity model to adopt it incrementally, which is a very serious practical barrier due to its artifact heavy nature and incorporation of three cutting edge reuse oriented techniques that are not yet wide spread.
7. Focuses on software engineering for reuse, saying very little about software engineering with reuse of legacy software.
8. Last but not least, has a very limited set of case studies that can serve as models for software engineering teams.

Except for the last one, most of these weaknesses are shared by most other software engineering methods.

2.2.3 The Kobra2 CBMDE Method

A second version of the Kobra method is currently being developed by the ORCAS Group (CIn – UFPE) with the Fakultät für Mathematik und Informatik, Universität Mannheim (FMI-UM), in Mannheim, Germany. Kobra2 expect to accommodate new trends such as support for more advanced forms of MDE and Component Based Engineering (CBE):

- By leveraging the latest OMG standards: UML 2.1 modular metamodel and better founded profiles; UML 2.1 full life cycle components; OCL 2.0 to model PIM, PSM, metamodels and UML 2.0 profiles that are fully refined and free of natural language ambiguities, thus viable input to fully automatic model transformation; MOF 2.0 and Ecore to define the Kobra2 metamodel; and

- By leveraging model transformation to weave product-specific elements onto framework components instead of or in addition to instantiating them, and to add on and take off Built-In Contract Testing (BICT).

The Specification and Realization views in Kobra2 are described via a metamodel that reuses and extends a minimal, consolidated, UML2 core, restricting model elements and enforcing consistency between views via OCL expressions. A comprehensive technical report about Kobra2 is in (Atkinson *et al*, 2009).

The Kobra2 method integrates three software reuse approaches: Model-Driven Engineering (MDE), Component-Based Development (CBD) and Object-Oriented (OO) frameworks. Beyond this integration, Kobra2 also innovates in two other ways. First it incorporates special views to model the layout, navigation and behavior of GUI components. Second, it introduces the concept of orthographic software engineering in which building a Platform-Independent Model (PIM) for each software component is piecemeal carried out by constructing largely orthogonal concern-specific views of the component. These views are then integrated into a Single Unified Model (SUM) which is in turn verified for conformance to the Kobra2 artifact metamodel.

2.2.3.1. Kobra2 Principles

The main principles of the Kobra2 method are:

Separation of Concerns - Separates functionalities of whole software concerns into reusable components. For each component separate: (a) *PIM* from *PSM* from code; (b) general product line *framework* model from specific product *application* model; (c) *execution* model from *testing* model; (d) *specification* from private, encapsulated *realization*; (e) *structural* model from *behavioral* model from *operational* model; (f) computational *service* aspects from *data* structures aspects; (g) *concept* (class) from *instance* (object) model.

Multiple Views – for each component, provide one view for each point in the multi-dimensional space of separated concerns and reconcile these views into a *Single Unified Model* (SUM). The Figure 2.4 shows the relationship between the SUM and the views.

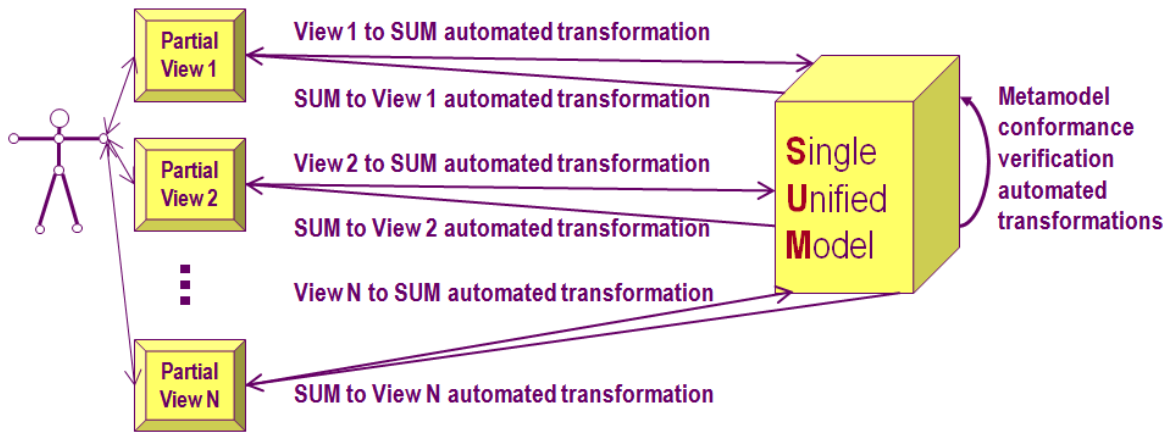


Figure 2.4 - Multiple Views with SUM.

Prescriptiveness – Kobra2 strives to precisely prescribe, as much as possible for a general purpose software engineering method, which UML2 and OCL2 model elements as basis to the development of each view of a Kobra component.

Formally Specified Rules to Automatically Conformance Checking – Kobra2 defines rules to check the conformance of the model in the following levels: (a) view-level; (b) component-level; and (c) assembly-level.

Parsimony – avoid as much as possible redundant model elements and views. To do this, Kobra2 choose a minimum model elements and diagram subsets of UML2, able to cover the key aspects/concerns of a software component.

Locality – all Kobra2 Views are *local* to a given component, and this component has the stereotype <<subject>> to specify the component owner of the View. The whole PIM of the system is derivable from the *union* of all these local views.

Uniformity – the *sole first-class modeling entity* deserving to possess its own multiple views is the Kobra2 component. All behaviorally complex entities are modeled as a Kobra2 component, including the system itself, and only behaviorally trivial system entities are modeled as Kobra2 classes.

Top-down Decomposition – the realization of any Kobra2 component *K* potentially consist of an assembly of finer-grained components, encapsulated in *K* and not directly visible outside of *K*.

2.2.3.2. Kobra2 Views

Kobra2 defines sixteen Views which can be seen in Figure 2.5 and that are described below.

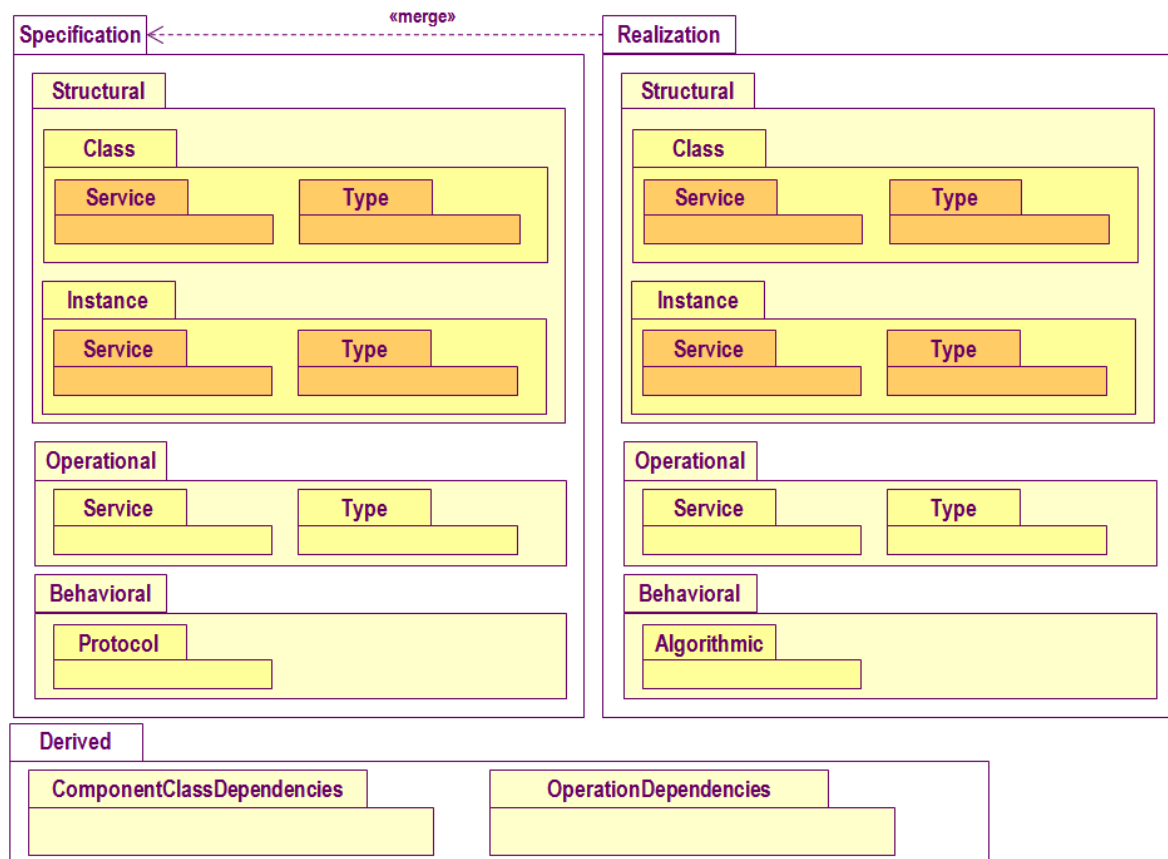


Figure 2.5 – Kobra2 Views

- *Specification Structural Class Service View* – specifies the local assembly connections of the subject component class, and its interface. Allow only public operations and attributes. The elements allowed in this kind of view are *ComponentClass*, *Class*, *Generalization*, associations stereotyped with *<<acquires>>* and *<<creates>>* and structural OCL constraints like *invariants*, *initialization*, *derivation* and *definitions*. A prototypical Specification Structural Class Service View can be seen in Figure 2.6.
- *Specification Structural Class Type View* – defines the non-primitive data types used by the subject component class in the *Specification Structural Class Service View*. The elements allowed are *Enumerations*, *Classes*, *Association Classes*, *Associations*, *Generalizations* and structural OCL constraints. The operations and attributes need to be public. A prototypical Specification Structural Class Type View is shown in Figure 2.7.

- *Specification Structural Instance Service View* – defines typical instantiation patterns of the *Specification Structural Class Service View* of the subject component class. It allows *ComponentObjects* with public slots, and *Acquires* and *Creates* associations.
- *Specification Structural Instance Type View* – defines typical instantiation patterns of the *Specification Structural Class Type View* of the subject component class. It allows *ComponentObjects* with public slots, and *Acquires* and *Creates* associations.
- *Specification Operational Service View* - declaratively specifies the behavioral contracts between the component classes of the *Specification Structural Class Service View* of the subject component class. It shows the OCL precondition, post-condition or body constraints of the operations. A prototypical *Specification Operational Service View* is shown in Figure 2.8.

```

16 context ViewPersistenceHandler::actionPerformed(
17         triggeredEvents : GUI-PIM-UF::Events::Event[*])
18 post:
19     triggeredEvents->exists(e: Event | e.oclIsTypeOf(ActionEvent)
20         and (e.source.name = 'fromCloud' or e.source.name = 'toCloud'))
21 implies
22     (e.source.name = 'fromCloud' implies self.wAKAMEGUIController^fromCloud())
23     and (e.source.name = 'toCloud' implies self.wAKAMEGUIController^toCloud())

```

Figure 2.8 - Prototypical *Specification Operational Service View*

- *Specification Operational Type View* - declaratively specifies the behavioral contracts between the component classes, (data) classes and association classes of the *Specification Structural Type View* of the subject component class. It shows the OCL precondition, post-condition or bodies constraints of the operations.
- *Specification Behavioral Protocol View* - defines the external visible state transitions of the subject component class together with the restricted subset of its public operation that is available for invocation in each of these states. Contains a simple UML2 protocol state machine. The sequence of operations on the state machine transitions represent the protocol to follow to satisfy the invocation contract of the services provided by the subject component class.
- *Realization Structural Class Service View* - defines the internal component assembly that realizes the services described in the *Specification Structural Class Service View* of the subject component. It shows the private attributes and operations signatures of

the subject component; the nested components of the subject with their public attributes and operations. It allows *ComponentClass*, *Class*, *Generalization*, stereotyped associations with <<acquires>>, <<creates>> and <<nests>>, and structural OCL constraints.

- *Realization Structural Class Type View* – it is for the *Realization Structural Class Service View* what the *Specification Structural Class Type View* is for the *Specification Structural Class Service View*. Defines the non-primitive data types used by either: (a) the private operations of subject component class; (b) the internal assembly of the subject component class; (c) but not used neither by its public operation nor by its external server components. The elements allowed are *Enumerations*, *Classes*, *Association Classes*, *Associations*, *Generalizations*, and structural OCL constraints.
- *Realization Structural Instance Service View* – it defines typical instantiation patterns of the *Realization Structural Class Service View* of the subject component class. It allows *ComponentObject* with public slots, *Acquires*, *Nests*, and *Creates*.
- *Realization Structural Instance Type View* - defines typical instantiation patterns of the *Realization Structural Class Type View* of the subject component class. It allows *ComponentObject* with public slots, *Acquires*, *Nests*, and *Creates*.
- *Realization Operational Service View* - declaratively specifies the behavioral contracts between the component classes of the *Realization Structural Class Service View* of the subject component class. It shows the OCL precondition, post-condition or body constraints of the operations.
- *Realization Operational Type View* - declaratively specifies the behavioral contracts between the component classes of the *Realization Structural Class Type View* of the subject component class. It shows the OCL precondition, post-condition or body constraints of the operations.
- *Realization Behavioral Algorithmic View* – Defines the algorithms for each operation appearing in the subject component's *Realization Structural Class Service View*. Contains $m \times n$ simple UML2 activity diagrams, where n is the average number of activity diagrams needed to fully specify the algorithms implemented by the operations.

- *Derived Operational Dependencies View* - automatically derived from the Operational and Algorithmic Views, local to each component, it shows the dependencies between the operations in the model.
- *Derived Component Class Dependencies* - automatically derived from the global operation dependency view and the local structural class views of each component, it shows the dependencies between the component classes in the model.

2.3. Cloud Computing

According to Yousef *et al* (2008) the term “Cloud Computing” is based on the collection of various old and new concepts of some areas of research as “Service-Oriented Architecture (SOA)”, “distributed and grid computing”, and also “virtualization”, although this subject is also related to other fields of research. In recent years, this term has gained much interest due to its enormous potential and the advance of diverse technologies in this area.

The term cloud is used as a metaphor based on how the Internet is represented in diagrams of computer networks, and also as an abstraction for the complex infrastructure it conceals (Scanlon and Wieners, 2009).

2.3.1 Principles and Related Concepts

There are four types of services offered by cloud computing shown in Figure 2.9, along with some suppliers of these service (Levitt, 2009):

- Internet-based Services – provides services such as storage (Data-Storage as a Service), middleware and collaboration;
- Infrastructure as a Service (IaaS) – provides a complete computing infrastructure by the Internet;
- Platform as a Service (PaaS) – provides a complete or partial development environment, that users can access and use online;
- Software as a Service (SaaS) – provides a complete application, such as complex application like CRM by Internet.

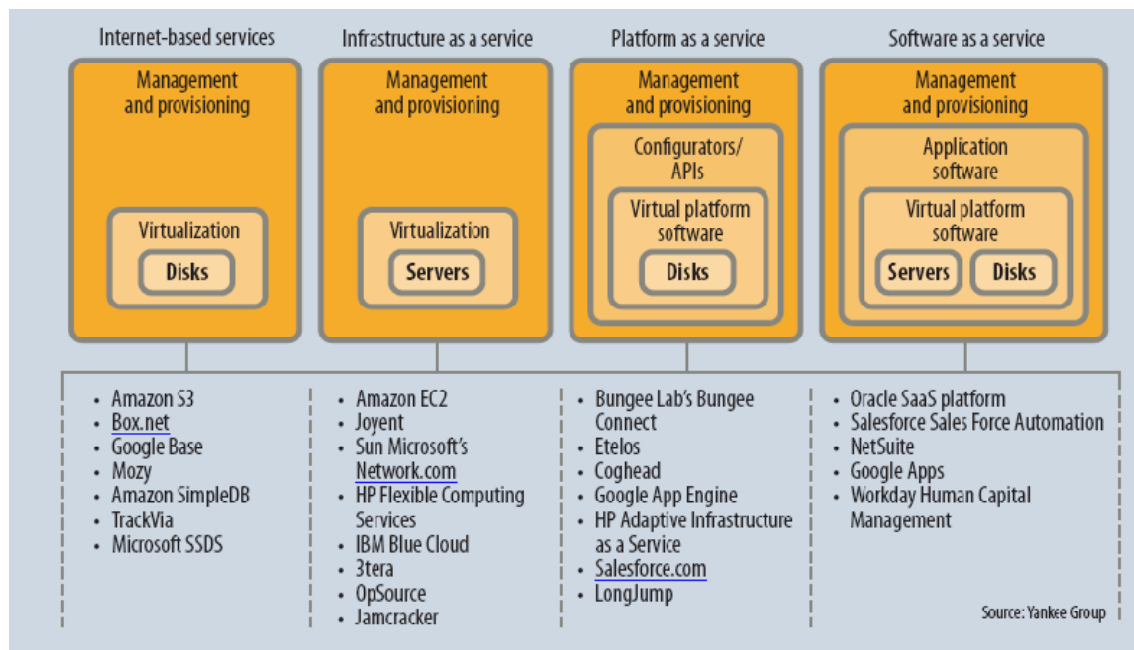


Figure 2.9 – Services offered by Cloud Computing. (Source: Levitt, 2009)

Yousef *et al* (2008) define also five layers in a computer system in clouds (Figure 2.10), being each layer responsible for one or more of the services offered by cloud computing:

- Application;
- Software Environment;
- Software Infrastructure;
- Software kernel; and
- Firmware/Hardware.

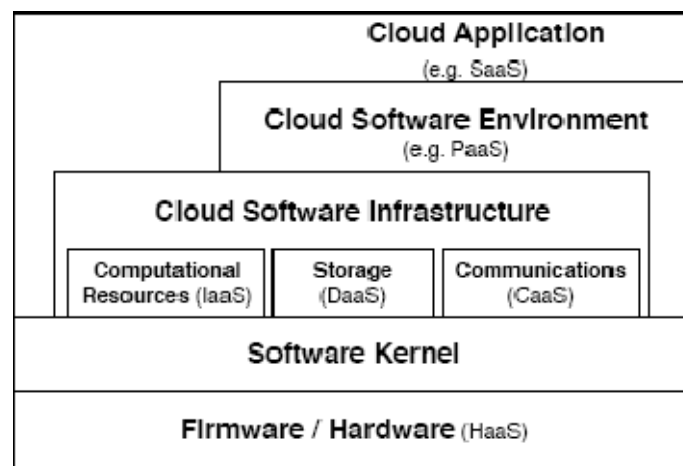


Figure 2.10 – Layers in a computer system in the clouds. (Source: Yousef *et al*, 2008)

2.3.1.1. Cloud Application Layer

This is the most visible layer for end users of the cloud. Normally, it provides services through an application for the end user. The idea of providing services may be referred as SaaS. The main idea of SaaS is that the end user will pay for the use of a server provided by a software, but he will not pay for the software itself. That is, instead of the user buy the software it needs, it will pay to use it while necessary.

For cloud computing, the idea of “to pay” is abstract, and the service provided may be paid or not. Instead of user having to download, install and configure the software it needs, it simply can access it from a browser.

Examples of applications in this layer of the cloud can be the Google Apps (GApps, 2009), the SalesForce CRM (SalesForce, 2009), PayPal (PayPal, 2009) and Yahoo! Maps (Y!Maps, 2009).

2.3.1.2. Cloud Software Environment Layer

Users of this layer are typically developers of applications for the cloud. This layer provides a development environment in the level of development with a clear set of APIs to facilitate integration between the environments and applications in cloud, as to speed up the deployment and to give support for the scalability needed of these applications. The service provided by this layer can be called PaaS.

Examples of systems in the level of this platform are the Google App Engine (GAE) (GAE, 2009), which provides a development environment for the languages Python and Java, and a set of APIs to interact with the cloud environment of Google. Another example is the Azure (Azure, 2009), the Microsoft competitor service to GAE.

2.3.1.3. Cloud Software Infrastructure Layer

This layer provides the essential resources to the subsequent layers, which can be used to build new environments of software and applications in the cloud.

The services provided by this layer may be categorized into:

- Computational Resources – Through the concepts of IaaS, this service provides virtual machines (VM) with computational resources to the cloud. These VM allow the software infrastructure customization, which will run with the best performance

and efficiency. An example of this service is the Amazon Elastic Compute Cloud (EC2) (EC2, 2009).

- Data storage – This service, known as DaaS, allows users to store their data on remote disks and access them from anywhere. This service is expected to have a high availability, reliability, performance, replication and consistency of data, so that it can allow a high scalability for application in cloud. An example of this service is the Amazon Simple Storage Service (S3) (S3, 2009).
- Communication – A vital component for an application in cloud is its network infrastructure, which provides communication between the various modules of an application in cloud. For this, it is used the concept of “Communication as a Service (CaaS)”, which says that it is necessary that the capacity of communication for systems in the cloud be service-oriented, configurable, predictable, reliable and with planning tasks.

2.3.1.4. Software Kernel Layer

In this layer, there is the software for the basic management of physical servers that make up the cloud. Typically, these software are operational systems, VM monitors and clustering middleware. At this level, the systems have as task, the abstract virtualization in grid computing, load balancing and setting up the checkpoints to migration.

2.3.1.5. Hardware / Firmware Layer

This is the base of the stack in the cloud computing, where all the physical hardware switches that form the backbone of the cloud are located. Usually large IT companies are required to provide the physical infrastructure. This layer is known as “Hardware as a Service (HaaS)”.

2.3.2 Benefits of Cloud Computing

Leavitt (2009) says that companies that run their own software platform need to buy and maintain their own software and hardware infrastructure, besides hiring a team to take care of these systems, which can be expensive and time consuming. In addition, in order to provide an environment that supports the intense use of the system, a augment of the capacity of processing and storage infrastructure is required, though in most of the time this resource becomes idle. Forrester’s Staten (Leavitt, 2009) also supports this idea,

saying “Most enterprise data centers are using less than 50 percent of the total capacity of their resources”.

The cloud computing aims to overcome this problem, offering a platform that is kept outside the user company, and is hired as needed of the user. A cloud computing system must have high availability and reliability, being executed by an infrastructure with various physical resources, and with redundancy of equipment to have a greater availability than an “in-house” infrastructure of a small or medium company.

Another advantage of computing in the cloud is the flexibility regarding the use of resources, which can grow during an intense use, or decrease during a stagnation of the use of the services. Even so, the user would pay only for the resource used in practice.

2.3.3 Google App Engine

Google App Engine (GAE, 2009) is an environment that lets anyone to run Web applications on Google’s infrastructure. GAE was launched by Google in April 2008 and is still undergoing tests. GAE proposal is to make, maintain and scale simple applications, even when the traffic of access and data storage needs increase. The aim is that GAE allows the development of a reliable application easily, even under intense use and access to large volume of data.

2.3.3.1. GAE Principles

GAE has three key aspects (GAE, 2009):

First it provides an infrastructure for running Web applications, focusing on making them easy to execute, build and to scale. The GAE is not only a solution for grid computing, it does not provides only a basic virtual machine. The GAE environment provides means to package the source code, to specify how it will respond to requests, besides providing the environment for the execution of its code.

The second aspect is that GAE environment provides a necessary infrastructure for the entire cycle of a web application, i.e., it provides means to run both the dynamics as well as the static portion of the application, supplies the database, allows the creation of

logs and methods to update to new versions. All this effort is an attempt to provide a simple alternative to the traditional *LAMP*² *stack*.

And finally, the third aspect is that the GAE provides access to Google's infrastructure, which allows creating systems with the same power of distribution, scalability and processing as the other applications of Google uses. The GAE also allows the use of the infrastructure for authentication (*Google Accounts*)(GACC, 2009), of disk storage (*GFS*) (GFS, 2009) and storage in the database (*Bigtable*) (BIGTABLE, 2009).

2.3.3.2. GAE Components

The GAE structure is divided in seven components (GAE, 2009):

- Web Server;
- Environment for Python Implementation;
- Environment for Java Implementation;
- Environment for offline development;
- Administrative Environment;
- Database;
- App Engine Services.

Web Server

One of the first components of GAE is the scalable infrastructure for publication of applications. The infrastructure implementation contains a distributed system that leverages automatically the application scalability, whenever the amount of accesses comes to grow. When uploading the application, GAE itself allocates the resources for the application, that is, the code is sent to diverse machines, and is responsible for making the connection between the user requests and the instance running the application. The GAE, automatically, provides more resources for the application, when it needs, but GAE also reduces the resources allocated, if the application is not using them.

² *LAMP stack* – refers to the solution architecture of open source software systems for configuration of web servers: *Linux*, *Apache*, *MySQL* e *PHP/Perl/Python*. It is used here meaning any set of software for web servers, representing the operating system, web server, database server and programming language.

The application runs in a secure environment (Sandbox), which leads to some limitations. For the sake of security, GAE restricts access to operational system, isolating the application and so it is independent of hardware, operational system and physical location of the web server. It is through these limitations imposed by Sandbox, GAE can distribute the requests to the web application across multiple servers.

GAE main limitations are:

- The application can only be accessed through solicitations using HTTP and HTTPS protocols.
- The application can not write to the file system and can only read files that were loaded with the application code.
- The application runs only in response to a web request, and should return the answer in a few seconds. When dealing with a request, the application can not create a sub-process or perform any operation after the response has been sent.

Environment for Python Implementation

For the development and execution of applications, one of the programming languages used by GAE is Python. This environment already includes the standard library of Python, but with some modules disabled, specifically those which directly access the operational system.

In addition to the standard library of Python, GAE includes some frameworks:

- Django 0.96.1 (DJANGO, 2009) – to support fast development, such as mapping object/relational, automatic creation of forms and templates system;
- WebOb 0.9 (WEBOB, 2009) – to deal with HTTP requests and responses;
- PyYAML 3.05 (PYYAML, 2009) – as a format for data serialization.

If the user wanted to include other frameworks, the environment can include the implementation of these codes along with the application code.

Environment for Java Implementation

Another environment of execution supported by GAE is Java. In this environment, Java applications run using the Java 6 virtual machine (JVM)(JAVA, 2009). The JVM runs in a safe “sandbox” environment that isolates the application for security reasons. The sandbox ensures that applications can only perform actions that do not interfere with the

performance and scalability of other ones. For example, it does not allow other threads to be called as well as it avoids data to be recorded on the file system location.

GAE uses the Java Servlet standard (SERVLET, 2009) for web applications, providing servlet classes, JavaServer Pages (JSPs), static files and the deployment descriptor (a *web.xml* file) in a WAR file directory structure.

The data persistence can be achieved through two standards for Java: Java Data Objects (JDO) 2.3 (JDO, 2009) and Java Persistence API (JPA) 1.0 (JPA, 2009), and the user is free to choose one.

Finally, from the standard Java library, the GAE released a White List (GAEJRE, 2009) to show which class library is allowed.

Environment for Offline Development

The GAE provides a SDK (*Software Development Kit*) for developing the application offline, both for the Python environment, and for the Java environment. The SDK allows development, debugging and testing the application through the APIs emulation provided by the GAE. Besides allowing the local application development, the SDK also provides a tool to automatic upload an application to the GAE servers, and also allows the update of an application already available at GAE.

Administrative Environment

In order to facilitate the setting of applications, the GAE provides an administrative web environment, which gives access to the information of the applications available. This administrative interface allows checking the status of the application, the published version, access to logs of the application, the configuration of the domain name, access to the database, among other things.

It provides a panel with real-time information on the situation of the application, the requests, errors, resource allocation, all integrated in the administrative environment.

Database

The GAE provides a powerful distributed data storage service that has a *query engine* and transactions. It supports a huge data set with millions of entities, but internally it is not based on a cluster of SQL servers, but in *Bigtable*, the same system that Google uses for its massive and scalable applications. This system of data storage, unlike the traditional, is not relational, it is object/relational.

The data storage API of Python provides a modeling interface that defines the structure of the entities of storage. Thus, it is not necessary to create structures for storage in the database, which are only defined in the application code.

As previously mentioned, the data storage system does not use SQL, but for information recovery, the GAE provides a query language, based on SQL, in GQL layer, allowing the easy creation of queries.

GAE Services

The GAE has a variety of services, through APIs that are described below:

- Integration with Google Accounts - APIs for integration with the authentication system of Google, Google Accounts, are available. These enable integration with the various personal services powered by Google.
- Access by URL - This service allows recovering resources of the application through URLs, using the same infrastructure that Google uses to retrieve web pages of their products.
- Email - It allows the application to use the Google infrastructure for sending e-mails.
- Memcache - The Memcache service provides a memory space for the application of high performance that can be accessed by multiple instances of the application. This is useful when the data need not be persisted, as temporary data that need quick access.
- Manipulation of Images - The application can resize, trim, rotate and invert images in JPEG and PNG.
- Scheduling Tasks – This service allows scheduling tasks to run regularly at a set time or set intervals of time.

2.3.3.3. Strengths

The advantage that *Google App Engine* promises is to ignore from the development and publication of web applications, the concerns of scalability, configuration and maintenance of the environment of implementing the application.

The GAE, automatically and transparently, provides the scalability needed for the application, in accordance with the amount of requests it receives, making the whole

process of load balancing and distribution of the application. This is done without having to write one line of code.

Further, all the necessary concern about installation and maintenance of operational systems, web servers, database servers as well as the need for physical equipment, such as a grid computing for applications that have a large amount of access are also handled by GAE.

It is not necessary to create structures for data storage, since they are described in the application code itself and created automatically with the implementation. Better, everything is done in a transparent way for the developer.

Another strong point is the use of an infrastructure used by widely known products, Google's products, which are already established in the market. This also implies, using an infrastructure that has been validated and improved for years by one of the biggest companies in the business of web applications.

2.3.3.4. Limitations

The first limitation is that *Google App Engine* is in pre-release that is, in beta phase and is not yet fully complete. Furthermore, though the use of GAE is free, it is limited. There are limits, such as bandwidth, amount of views per month, size of storage, amount of files for applications.

But for some of these limitations, whenever necessary to use more than the limit imposed, you can buy more resource, that is, you pay for what you used beyond the pre-established limit.

Due to security restrictions addressed by GAE, other limitations were created as: the compulsory use of only HTTP and HTTPS requests, which prevents the use of sockets, and the processing is only done when a request is made, which prevents the execution of scheduled tasks.

Also, the total dependence of the application with the Google can be considered a limitation, since the GAE is not an open infrastructure. To port the application to another environment will require the recoding of the same.

2.3.4 Google App Engine and others cloud computing platform

The GAE offers services at PaaS level, providing an environment for development and implementation for applications in the clouds. Similarly, there is Azure (Azure, 2009)

which also provides similar services, but one of the main differences is about the technology used. While Azure provides a development environment based on platform .Net (DOTNET, 2009), the GAE provide environments in Python and Java. Another difference is that the GAE only allows the deployment on their own servers (unless the development environment), while a proposal of Azure is to allow companies to also use its infrastructure in internal servers. Thus, their application would be available only in the company intranet, which is a security measure for the application.

Another platform for cloud computing is provided by Amazon Web Services (AWS) (EC2, S3, 2009). The AWS provides various services at the level of IaaS. One of the services is the Amazon Elastic Compute Cloud (EC2, 2009) which provides virtual processing units, that is, virtual machines on which you can set up their infrastructure software, such as the operational system, the software that will run. Additionally, it allows, according to need, increase or decrease the processing power of your virtual drive.

Another interesting service provided by the AWS is the Amazon Simple Storage Service (S3, 2009), a robust service for persistence of data that can be accessed from anywhere. Another advantage is that there are no limits on quantity of objects to be persisted, and these objects can have from 1 B to 5 GB.

The main difference between the AWS and GAE is the kind of service that both provide. As said before, the GAE provides an environment for development and implementation of its application in the clouds, while the AWS provides services to create the environment for development and implementation. Another difference is that the use of AWS is paid, while the GAE is free, but with several limitations. The advantage of the AWS on GAE is that, as one can build its proper environment for implementing the applications, he will not have the restrictions addressed by GAE. By the other hand, unfortunately, you have to take care of the whole infrastructure of the environment of implementation, while with the GAE you will be only worried about the maintenance of the application.

2.4. Model-Driven CASE Tools

CASE is an acronym to Computer Aided Software Engineering. According to (FFIEC, 2009) CASE tools are a kind of software that automates many of the activities involved in software development life cycle phases. For example, when establishing the functional

requirements of a proposed application, prototyping tools can be used to develop graphic models of application screens to assist end users to visualize how an application will look after development. Subsequently, system designers can use automated design tools to transform the prototyped functional requirements into detailed design documents. Programmers can then use automated code generators to convert the design documents into code. Automated tools can be used collectively, as mentioned, or individually. For example, prototyping tools could be used to define application requirements that get passed to design technicians who convert the requirements into detailed designs in a traditional manner using flowcharts and narrative documents, without the assistance of automated design software.

UML CASE tool represents a subset of CASE tools that is a software application that supports some or the entire notation and semantics associated with the Unified Modeling Language (UML).

The UML tool term is used broadly to include application programs which are not exclusively focused on UML, but which support some functions of the Unified Modeling Language, either as an add-on, as a component or as a part of their overall functionality.

UML tools generally support the following kinds of functionality, such as diagramming, round-trip engineering, code generation, reverse engineering, model and diagram interchange and model transformation.

Diagramming in the context of UML CASE Tools means creating and editing UML diagrams; that is diagrams that follow the graphical notation of the Unified Modeling Language. The use of UML diagrams as a means to draw diagrams of – mostly – object-oriented software is generally agreed upon by software developers. When developers draw diagrams of object-oriented software, they usually follow the UML notation. On the other hand, it is often debated whether those diagrams are needed at all, during what stages of the software development process they should be used, and how (if at all) they should be kept up-to date.

Another functionality provided by UML CASE Tools is round-trip engineering, what refers to the ability of a UML tool to perform code generation from models, and model generation from code (a.k.a., reverse engineering), while keeping both the model and the code semantically consistent with each other. Code generation and reverse engineering are explained in more detail below.

The code generation provided by UML CASE Tools means that the user creates UML diagrams, which have some connoted model data, and the UML CASE Tool derives from the diagrams parts or all of the source code for the software system. In some tools, the user can provide a skeleton of the program source code, in the form of a source code template where predefined tokens are then replaced with program source code parts during the code generation process. There is some debate among software developers about how useful code generation as such is. It certainly depends on the specific problem domain and how far code generation should be applied.

Another useful functionality is the reverse engineering, that is, the UML CASE Tool reads program source code as input and derives model data and corresponding graphical UML diagrams from it (as opposed to the somewhat broader meaning described in the article “Reverse engineering”). Some of the challenges of reverse engineering are: The source code often has much more detailed information than one would want to see in design diagrams. This problem is addressed by software architecture reconstruction. Diagram data is normally not contained with the program source, such that the UML tool, at least in the initial step, has to create some random layout of the graphical symbols of the UML notation or use some automatic layout algorithm to place the symbols in a way that the user can understand the diagram. For example, the symbols should be placed at such locations on the drawing pane that they don’t overlap. Usually, the user of such a functionality of a UML tool has to manually edit those automatically generated diagrams to attain some meaningfulness. It also often doesn’t make sense to draw diagrams of the whole program source, as that represents just too much detail to be of interest at the level of the UML diagrams.

UML CASE tools also does models and diagram interchange, what is defined by the OMG standard: XML Metadata Interchange (XMI). Unfortunately, XMI does not yet support diagram interchange, which is a significant shortcoming for a visual modeling language. Consequently, even when you can import a UML model from one tool to another with XMI, you will likely need to redraw your diagrams.

A key concept associated with the Model-driven architecture initiative is the capacity to transform a model into another model. Some UML CASE Tools also provide this functionality. For example, one might want to transform a platform-independent domain model into a Java platform-specific model for implementation. It is also possible to refactor UML models to produce more concise and well-formed UML models. Finally, it is

possible to generate UML models from other modeling notations, such as Business Process Modeling Notation (BPMN). The standard that supports this is called QVT for Queries/Views/Transformations.

This chapter describes a MDE CASE tools analysis including their common standard architectures and deployment platforms, as well specific and common limitations.

2.4.1 MDE CASE Tools

Among the various Model-Driven Engineering (MDE) CASE tools that have arisen lately, just those focusing on the following two characteristics were taken for analysis:

- **Models Store** – tools that provide support for the creation of repositories of models for a specific metamodel. Moreover, we seek the tools that allow the instantiation of the elements of the model, in addition to functionalities to load and save them in XMI format. In this category, amongst the tools identified, we just have taken into account the following: Eclipse Modeling Framework (EMF, 2009; Budinsky *et al*, 2003) and PyMOF (PYMOF, 2009).
- **Engines for Model Transformation** – this category addresses the tools specialized in carrying transformations of models, in the context of MDE, allowing to lower the abstraction of a Platform Independent Model (PIM) to various Platform Specific Models (PSM). Amongst the various tools available in this category, we selected two to present: ATLAS Transformation Language (ATL, 2009) and Epsilon (Epsilon, 2009), in particular the Epsilon Transformation Language (ETL).

In the following subsections, we detail briefly each one, highlighting their positive features.

2.4.1.1. Eclipse Modeling Framework (EMF)

The Eclipse Modeling Framework (EMF) is an open-source project belonging to the family of projects that Eclipse (Eclipse, 2009) holds. The EMF is a modeling framework and Java code generator for creation of model repositories.

With EMF you can define a metamodel, generate a repository for it, and set up interfaces for editing the repository through the GUI of Eclipse. However, for the definition of metamodels, the EMF does not use the standard MOF supported by OMG, but it defines a metamodeling language called Ecore (Ecore, 2009), which is EMOF-based (MOF, 2009) besides being optimized for Java implementation (EMF, 2009a).

As previously mentioned, the Ecore metamodel, depicted in Figure 2.11, is quite similar to the EMOF metamodel. In a way, one can say there is a relationship one-to-one among their concrete elements. The main difference between the Ecore and EMOF emerges from the structural features: while Ecore explicitly differentiates an attribute (*EAttribute*), which is for primitive (*EDataType*) and enumeration types, and a reference (*EReference*) that are used for class (*EClass*), in EMOF both are equivalent to *Attribute*, which can be used for both primitive types (*PrimitiveType*) and class (*Class*). Furthermore, others mismatches arise from: (i) the hierarchy of abstract elements, and (ii) the name of concrete elements of the metamodels, (iii) and in terms of their properties and associations. A survey of equivalence between the concrete elements of Ecore and EMOF are listed in Table 2.1.

Figure 2.11 - Ecore Metamodel. (Source: Ecore, 2009)

formats as annotated Java interfaces, XML and XSD files and UML models. Before processing, all input formats take the form of Ecore models. Through the ECore metamodel, the model to set the configurations for the code generation, namely the *GenModel*, is defined.

Table 2.1- Equivalences between the Ecore and EMOF concrete elements

Ecore	EMOF
EModelElement	Element
EPackage	Package
EClass	Class
EDataType	PrimitiveType
EEnum	Enumeration
EEnumLiteral	EnumerationLiteral
EParameter	Parameter
EOperation	Operation
EAttribute	Attribute
EReference	Attribute
EAnnotation	Tag
EFactory	Factory

Hence, it is possible to create, based on the GenModel together with the EMF, three sorts of plug-ins for Eclipse, the repository model and a GUI example to create models generated through this repository (Backvanski and Graff, 2005):

- **EMF.model** – this plug-ins holds the complete implementation of the repository of model from a specific defined metamodel, along with the mechanisms to ensure persistence, and to import and export from/to XMI format. Usually, just the methods previously defined in the input metamodel are subject to be changed, or even implemented.
- **EMF.edit** – This plug-in allows the creation of an adapter for the plug-in interfaces EMF.model, regardless of the presentation layer. This plug-in provides an interface to access all features of the model like a layer of presentation needs. One of its responsibilities is to separate the GUI and business rules stored in the repository of models. Normally, just small changes in the generated code are necessary.
- **EMF.editor** – the GUI for Eclipse to allow access to the repository (model). The GUI is the standard interface used by the EMF, using resources as the tree of elements and the pallet of properties for editing the model. A snapshot can be seen in Figure 2.12. To create more complex interfaces, such as those using graphical diagrams, the user must code manually to replace the non-appropriate plug-in.

However, regarding the proposals of our work, we highlight the EMF.model amongst these three Eclipse plug-in, since the repository of models generated is stored within it. One positive aspect is the possibility to reuse the repository of models outside the Eclipse, in another applications and projects.

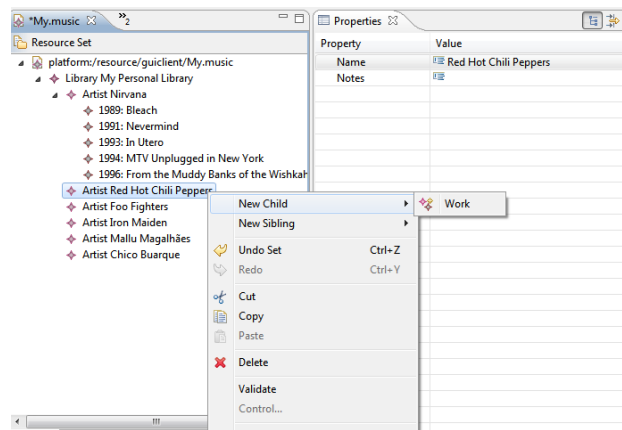


Figure 2.12 - Example of GUI generated by EMF

Each modeled class in the user metamodel leads to a pair of Java entities: an interface and a class to realize it. Additionally, the code generated implements two frameworks, one to guarantee the data persistence and a second yielding warning and all the sorts of notifications. Figure 2.13 shows an example with the structure compounds in layers. Note the interface *Artist*, regarding a user metamodel class, and its realization by the class *ArtistImpl*. All the classes defined by EMF implements the interface *EObject*, which provides the data-persistence and the API reflexive. By the transitive closure, each class implements also the *Notifier* interface, which in turn allows other elements to observe the class.

The repository defined by EMF allows to program the instantiation of the elements of the metamodel dynamically. This is possible through the interface *Factory* that provides methods for instantiate each concrete element in the repository.

2.4.1.2. PyEMOF

PyEMOF (PYEMOF, 2009) is an implementation of EMOF specification, exclusive for Python development, made by prof. PhD Raphaël Marvie from the University of Lille. As expected, the PyEMOF allows the generation of a repository of models in Python. The PyEMOF yields files in XMI format, saving or loading any model, ensuring thus its persistence. Going a bit beyond with respect the tools already discussed, it allows also as input to create the repository, text files in a notation similar to KM3.

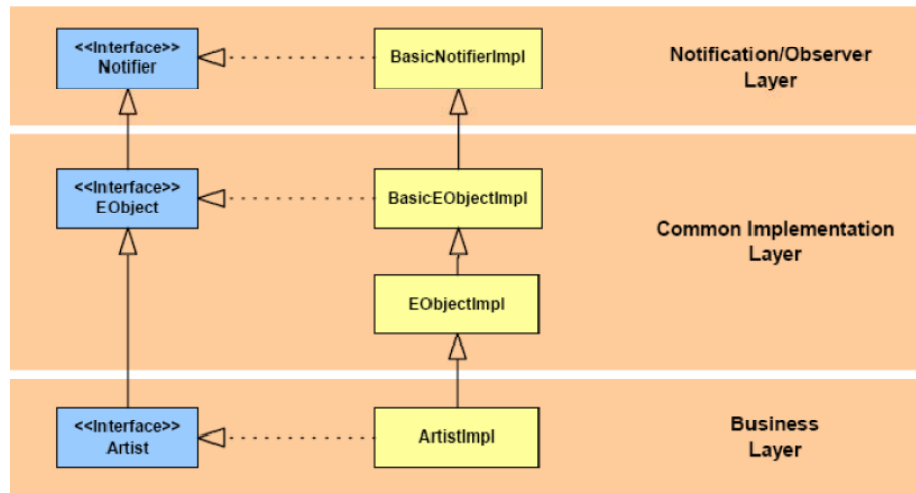


Figure 2.13 - Framework for Generated Model. (Source: Backvanski and Graff, 2005)

However the PyEMOF project is not widely used, with few references and little documentation. Furthermore, the PyEMOF is not aligned with the latest version of the EMOF specification and, moreover, it has no support for enumerators and for the reflexive API. Another gap is the lack of support for generation of repositories from metamodels structured in more than one package. Nevertheless, nowadays the PyEMOF is the only tool to an environment Python-based.

2.4.1.3. Epsilon

Epsilon (Epsilon, 2009) is an open-source project among the projects of Generative Modeling Technologies (GMT) of Eclipse. The Epsilon is a platform for building consistent and interoperable languages for specific tasks as model transformations, code generation, comparison of models, merging, and validation (Kolovos *et al*, 2008). As said before, it defines some languages for specific tasks, and for each language, provides an interpreter and IDE Eclipse-based. These languages are (Kolovos *et al*, 2008):

- **Epsilon Object Language (EOL)** – is an action language built over the navigational mechanisms of OCL, but includes additionally support for sequencing of sentences, access to multiple models, conventional programming statements, such as repetitions structures. EOL can be seen as a standalone language for managing models, but it also is reused in other languages defined in Epsilon (Kolovos *et al*, 2006).
- **Epsilon Validation Language (EVL)** – this language validates restrictions written in the models. To achieve this goal, it provides dependent restrictions, significant

errors messages and separation of critical from non-critical (alert) (Kolovos *et al*, 2006) restrictions.

- **Epsilon Transformation Language (ETL)** – is a language based on rules. An ETL module consists of a set of *transformations-rules* that can translate elements of a source model to elements of a target model. Similarly to ECL *match-rules*, each *transform-rule* has a declarative signature accompanied by an imperative body in EOL (Kolovos *et al*, 2006).
- **Epsilon Comparison Language (ECL)** – ECL is a language based on rules adapted to compare models. An ECL specification is formed by *match-rules* which can compare two elements of an input model. Declaratively, each *match-rule* is decomposed into two essential parts: the *compare* and the *conform*, to decide if the elements under investigation match and are in conformance to each other. The body of the both parts is expressed in EOL (Kolovos *et al*, 2006).
- **Epsilon Merging Language (EML)** – this language addresses the common tasks of merging models and metamodels. To this end, EML language reuses the ECL *match-rules* and the ETL *transformation-rules*, besides adding new specific rules, namely the *merge-rules*, which allows the composition of specifications of merging in a structured and comprehensive way (Kolovos *et al*, 2006).
- **Epsilon Wizard Language (EWL)** – provides an effective and appropriate support for updating transformation models of various metamodels (Kolovos *et al*, 2008).
- **Epsilon Generation Language (EGL)** – is a model-driven-template-based language for model to text code generation, built on top of the Epsilon architecture and reusing EOL. It supports content-destination decoupling, protected areas to support the automatic generation as well as the manual inclusion of code, and coordination of templates (Kolovos *et al*, 2008).

As can be seen in Figure 2.14, EOL is the basis for all the other languages defined. But for communication with the repository of models, the Epsilon sets the **Epsilon Model Connectivity (EMC)** which is the interface between EOL and the repositories of persistence and serialization of models. It permits also heterogeneous repositories, such as EMF or the Metadata Repository (MDR) (MDR, 2009) of NetBeans.

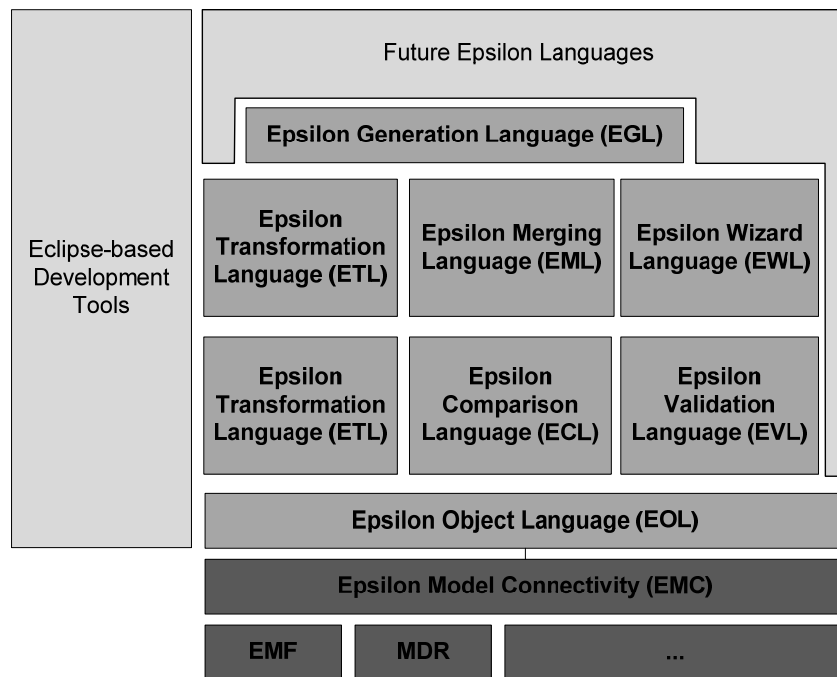


Figure 2.14 - Epsilon Architecture. (Adapted from: Kolovos *et al*, 2008)

Epsilon stands out positively by the diversity of specific languages for each MDE tasks (as specific as these languages) giving supporting to the broad set of MDE activities. However, in the main source of reference, *The Epsilon Book* (Kolovos *et al*, 2008), there are several inconsistencies in the languages metamodels, besides being incomplete. Another negative aspect is, in addition to having an IDE for each language, these IDEs does not have many resources, such as auto-complete, and also the infrastructure for debugger is too weak. Also, not so nice for the company have been the numerous complaints from the employees regarding the bugs in the tool and the definition of languages.

2.4.1.4. ATLAS Transformation Language

The ATLAS Transformation Language (ATL), developed by the research group ATLAS INRIA & LINA (ATLAS, 2009), was proposed as a response to the OMG QVT RFP. However it was not adopted as a standard by the OMG, but became a project of the Eclipse open-source. ATL is a full implementation of a language to transformation of models, rule-based, declarative-procedural hybrid that allows developers to specify how a given set of source models produce a set of target models. Obviously, the source and target models need to be in agreement to its respective metamodels, which is associated with the transformation. The ATL Development Tool (ATL-DT) (an Eclipse plug-in), is the one to provide support to the development and execution of ATL transformations.

The abstract syntax of ATL is specified as a MOF metamodel where each element has a textual concrete syntax. Figure 2.15 illustrates the basic construction of the ATL module, which encapsulates all the transformation, the components, libraries and also indicates the source and target metamodel.

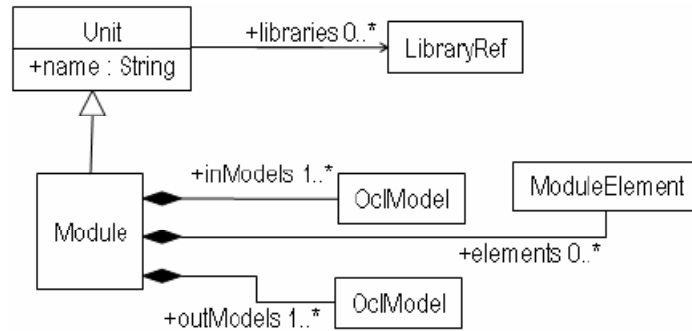


Figure 2.15 - ATL Module.

A *ModuleElement* is either a *Rule* or a *Helper* as depicted in Figure 2.16. The *Helpers* act as object-oriented methods both associated with a context of a module or a specific metaclass of the source metamodel. *Helpers* with zero arguments are attributes.

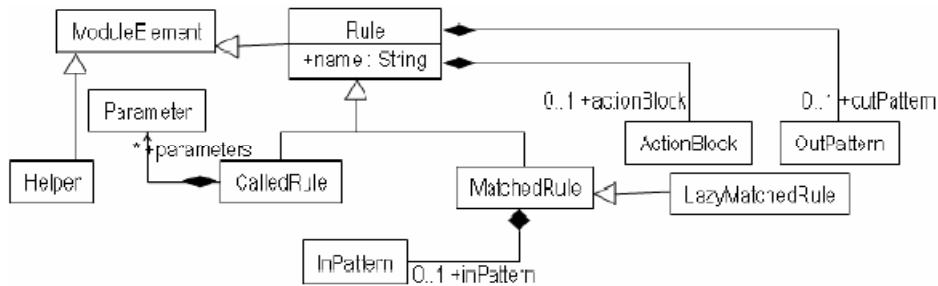


Figure 2.16 - ATL Module Element.

ATL distinguishes three types of rules:

- **Matched rules** are the purely declarative rules, fired only when matching a pattern;
- **Lazy matched rules** are possible to fire more than once;
- **Called rules** are the ones explicitly called while firing another rule; elements of the target model of this sort of rules are triggered by explicit layers within the imperative portions of code, both indirectly through pattern-matching and by lazy rules.

A rule may also contain a imperative block of action, specifying the sequence of instructions that must to be executed after the generation of the standard output.

A matching rule sets an input pattern, which specifies a set of elements of models from the input metamodel, associated with names of variables and optionally using a Boolean OCL expression. These input elements are mapped to an output pattern. Both types are OCL types. This output pattern, in turn, is a set of elements of models from the target metamodel, associated with names of variables and bindings. When the rule with the standard output is fired, the target elements are created. A binding relation specifies the value used to initialize a specific property of an instance. Figure 2.17, shows the metamodel for the input/output patterns.

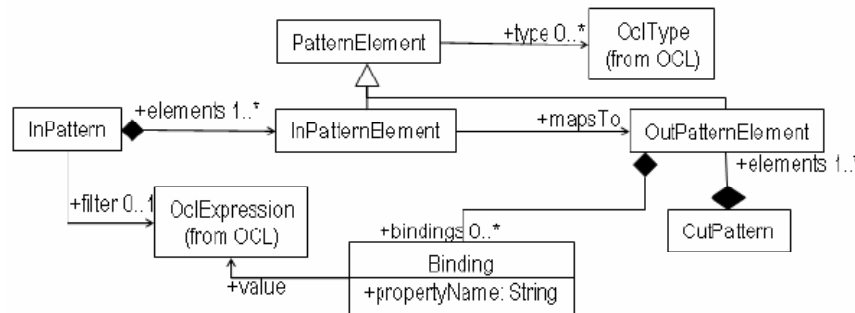


Figure 2.17 - ATL Input and Output Patterns.

A strong point of ATL is that it takes advantage of Ecore as the engine to represent the metamodels, that is, it makes reuse of the EMF, which is widely used by several tools. However, even with an IDE for development, this is not very rich in terms of resources available and, unfortunately, has a very illegible debugger. Another negative factor, especially in this work, is that the realization of transformations is made through batch jobs, and through reading/generation of input and output files based on the file system. As this work will be developed on a cloud environment, in particular in the Google App Engine (GAE), this becomes an obstacle because the GAE does not allow access to read/write directly to the file system.

2.5. Chapter Remarks

This chapter exhibited the methodological and technological background for this research. It was presented the Model Driven Engineering and its principles, standards and technologies, Component Based Engineering with the Kobra method, cloud computing principles and analyzed MDE CASE Tools. The MDE principles among CBE practices were the methodological reference for the development of this work, because the produced tool aimed to overcome the lack of standards compliant tools to support these practices.

The cloud computing principles and goals were used to plan and define the WAKAME tool architecture. The analyzed tools were used to identify its weakness and strengths in such way that the WAKAME tool could benefit from both aspects.

CHAPTER 3

WAKAME PROJECT

In this chapter we describe the foundations and requirements of the WAKAME tool, the modeling of KWAF, a framework for PIM modeling Web applications, and the modeling of a case study for the KWAF evaluation, the PhotoAlbum. After this we will show the modeling of the WAKAME'S top-level PIM as an instantiation of KWAF.

3.1. Long-Term Goals and Principles

The WAKAME project - a Web App for Kobra2 Model Engineering - has as intention the development of a CASE tool which supports the modeling with the Kobra2 method. This project is being developed by the research group ORCAS, and it is being accomplished by two master's degree students: Breno Machado (author of this master thesis) and Weslei Marinho (2009).

For the development of the WAKAME tool, the following scope separation was defined:

- Breno Machado - modeling and implementation of the component regarding the WAKAME tool server, where he should define the Repository of Models capable to execute the transformations: View to SUM, SUM to View, and SUM to SUM, defined by Kobra2;
- Weslei Marinho - modeling and implementation of the component regarding the WAKAME's client, where it should define the implementation for Views edition defined by Kobra2, creating a minimalist GUI with little icons and actions to possess a smaller learning curve.

In this research UML tools has been analyzed both standalone and the few online tools we could find. The analyzed tools are shown in Section 2.4 and we could define, by looking at them, the main features that a tool for Kobra2 method could offer, as its requirements:

- *Draw diagrams* – The tool must support easy rendering of the diagrams in the modeling language. The tool should be “intelligent” enough to understand the purpose of the diagrams and know simple semantics and rules, so it can warn the user and prohibit the inappropriate or incorrect use of the model elements.
- *Act as a repository* – The tool must support a common repository, so the collected information about the model must be stored in the same place. For instance, if the name of a class is changed in one diagram, the change must be reflected in all other diagrams in which the class is used. The integration with a configuration management system must keep the repository information consistent and synchronized.
- *Navigation support* – The tool should make it easy to navigate through the model, to trace an element from one diagram to another, or to expand the description of an element.
- *Provide multiuser support* – The tool should support multiple users and enable them to work on the same model without interfering with or disturbing each other.
- *Generate code* – An advanced tool should be able to generate code, where all the information in the model is translated into code skeletons that are used as a base for the implementation phase.
- *Reverse engineer* – An advanced tool should be able to read existing code and produce models from it. Thus, a model could be made from existing code, or a developer could iterate between working in the modeling tool and programming.
- *Web accessibility* – The tool should be able to be accessible through the web, without the user needing to install the tool at his/her computer. Therefore, the user doesn’t need to download the tool installer, to install and then execute it.
- *Lightweight* – It should be one of the primary goals for a CASE tool since it could be accessible from web. The user should not have to wait for a long time to have the tool downloaded before he/she could use it.
- *Simple User Interface* – As the tool could be used for different people, the User Interface should be simple. For instance, to find the option to set up an attribute in one element could not take a long time. The use of the property panels should be kept

to the minimum as possible, allowing the user to configure model elements directly in the diagram via its concrete syntax.

- *Low learning curve* – The tool should offer a low learning curve, in such way that the user should spend little time and energy in order to use it productively. If the tool has a high learning curve, it requires substantial study and experimentation before it actually become useful. Providing a low learning curve, the tool allows the average user to be able to pick it up and intuitively learn how to use it.
- *Integration with other tools* – A tool should be integrated with other tools, both with development environments such as an editor, compiler, and debugger, and with other enterprise tools such as configuration-management and version-control systems.
- *Cover the model at all abstraction levels* – The tool should be easy to navigate from the top-level description of the system (as a number of packages) down to the code level. For instance, to access the code for a specific operation in a class, you should be able to click the operation name in a diagram.
- *Interchange of models* – A model or individual diagrams from a model should be able to be exported from one tool and then imported into another tool, as Java code is produced in one tool and then used in another tool, the same interchange should apply to models in a well defined language.

Besides these requirements that we identified through the observation of several UML tools, to assist the principles of Kobra2, presented in the section 0, we listed other requirements that the tool should have:

- *Allow multi-views* – For each component, provide one view for each point in the multi-dimensional space of separated concerns;
- *To maintain the consistence between the SUM and the visions* – the tool must to reply, through transformations of models, all the modification that the views suffer, and to update these views in agreement with the modifications made in the SUM;
- *To allow the use and validation of OCL expressions* – for the definition of behavioral models, and to create the restrictions on the structural models;
- *To maintain the consistence among the visions* – when modifying an information in a view, the other views that share this information should be updated to maintain the conformity among them;

- *Local visions* – a vision of a component should bring only the information really necessary for the understanding of the same;
- *Navigation* - to allow navigating among the components, and through the views of each component.

These were the requirements used as the basis for the development of the WAKAME tool. In the next section, it will be described the Kobra2 Web App Framework (KWAF) - that is the architecture that WAKAME were build above. In the section 3.3 the top-level modeling of WAKAME will be exhibited, that is common for the two collaborators that are working in this project.

3.2. KWAF

The development of applications is a complex task that demands high investment of time and resources. In addition, Web applications evolve so much faster than traditional applications, not only because of changes in requirements, but also because the platforms are in constant development evolution. The KWAF (Marinho *et al*, 2009) is a framework that shapes aspects of a generic Web application, from the GUI to the Web services, through the data model. The main idea is that through the specialization of the KWAF abstract models, new models for specific web platforms could be generated, increasing productivity and reducing development costs. Main issues related in development of Web applications are presented, abstracted and mapped to the KWAF framework. The use of KWAF is illustrated by a toy example of a Web photo album application.

3.2.1 KWAF Framework: Principles, Structure and Case Study

Raising the level of abstraction for dealing with complex problems through the use of models is one of the most important principles of software engineering. The use of frameworks allows the specialization of abstract reusable models, increasing the productivity through the ceiling of choices during the model development, especially in complex systems.

The goal of the KWAF framework is to define a general architecture to Web-based systems, i.e., its basic structure, components and relationships between them. To use the framework, a developer should extend its components, making changes according the used application domain and adding specific functionality. KWAF reduces the overhead associated with the modeling and development of Web applications.

KWAF is comprised of components represented by the UML stereotype componentClass, unlike general UML models, using a specific representation for components. These components are related by two types of associations also stereotyped:

- Nests association, which indicates that a component is a sub-component of another, and;
- Acquires association, which indicates that a component use services of another component.

Other used stereotypes are defined bellow:

- subject - indicates that a diagram is related to the component marked with this stereotype, and;
- GUI - indicates that this is a graphical user interface component.

To illustrate the KWAF framework instantiation, we present the model of an online photo album (Figure 3.1 shows an example of its main screen), a toy application that allows exploit the key concepts defined by the framework. In the following sections we describe the framework in detail, illustrating it with the Photo Album application examples.

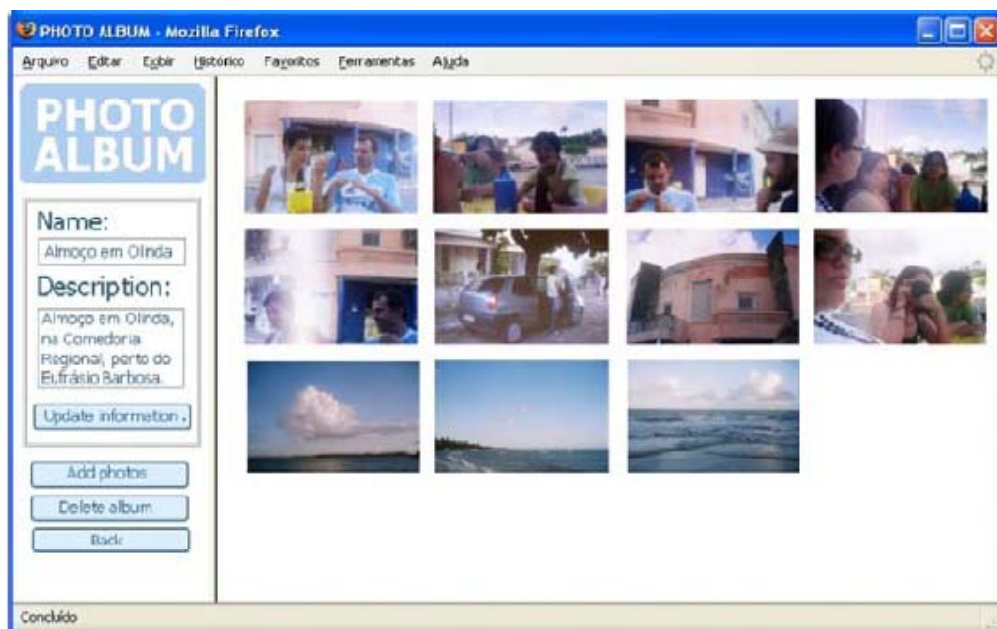


Figure 3.1 - The Web Application Photo Album

3.2.2 Structural models of the KWAF Component

This section presents a Platform Independent Model – PIM, of the KWAF components. According to (Mendes and Moses, 2006), a Web application consists of a system available through the World Wide Web that combines the characteristics of a hypermedia application with the features of a traditional application. (Shklar and Rosen, 2003) states that a Web application is a client-server application that uses a Web browser as the client program and conducts an interactive service through the connection to servers via the Internet, showing dynamic content customized based on request parameters, user behavior and security considerations. (Fowler and Stanwick, 2004) claim that a Web application client can be represented by a hybrid application, using both: a traditional application and a web browser. To give support to all these definitions and allowing flexibility in the framework use for Web applications modeling, we defined the KWAF component.

The KWAF component abstracts a Web application as a whole, do not requiring or using the services of any other external component. In order to increase flexibility and reusability, this abstraction was not set to be accessed externally, whereas the Web application would be self-contained. This representation defines a structure where the sub-components are nested. The KWAF component nests two sub-components, the first sub-component will perform the graphical user interface presentation and user interaction (*GUIClient* component), while the second sub-component handles business services, represented by Web services (*WebService* component). The organization of these components is shown in Figure 3.2.

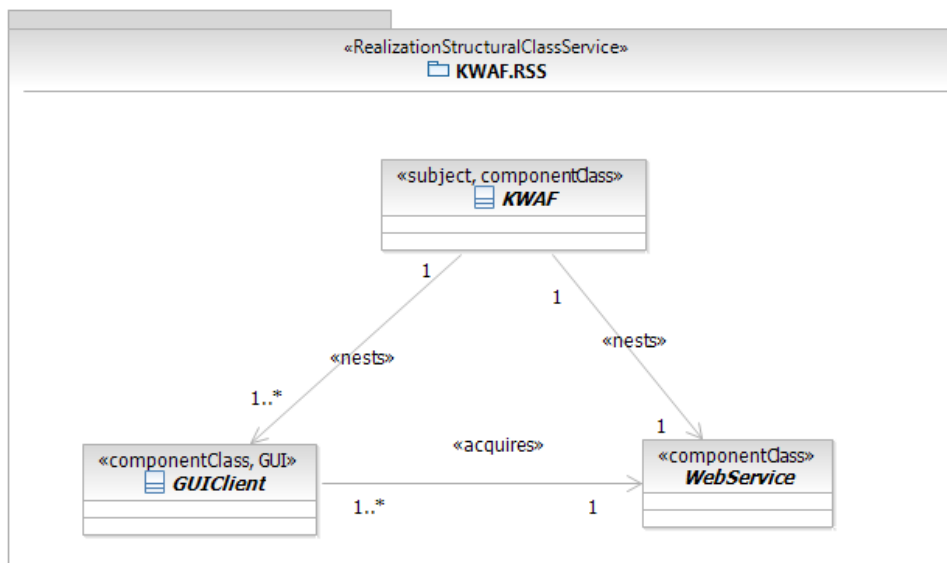


Figure 3.2 - Structural Model of the KWAF Component

The sub-component that provides Web services based on application business rules is presented in Section 3.2.3. The sub-component *GUIClient*, responsible for user interaction and information presentation via graphical interface, is detailed in Section 3.2.4.

3.2.3 Structural Models of the Webservice Component

The *WebService* component is responsible to handle services that Web application will provide. According to the definition shown by Lewandowski (1998), we can consider that this component would be the server in terms of the Client/Server architecture. The Web component provides a single interface for communication, where any client can request the available services. This interface is provided through only one method, the *process(request: Request): Response* method. In this method all the exchange of information will be done through the Request and Response class objects (WebObject, 2009), after being invoked, this method will then delegate the responsibility to fulfill that operation to the same method signature of the *ServiceController* component. The *WebService* component structure comprises three other sub-components that are nested: *ServiceController*, *MVCAction* and *MVCModel*, as is illustrated in Figure 3.3.

These three sub-components represent the model role (components *MVCModel* and *MVCAction*) and the controller role (component *ServiceController*) in the Model-View-Controller (MVC) architectural pattern (Buschmann *et al*, 1996; Gamma *et al*, 1994; Leff and Rayfield, 2001). To the generation of a Platform Specific Model (PSM), the Web component can be directly mapped to specific technologies, such as: JEE; Servlets, or the API provided by *RequestHandler* Google App Engine (GAE) in Python (GAE, 2009; Lutz, 2006).

In the KWAF framework case study (the Photo Album application PIM) we created the *PhotoAlbumWebService*, a component that extends the *WebService* component. With this, the *PhotoAlbumWebService* will obtain the specific methods and objects already defined in the *Webservice*, as shown in Figure 3.4.

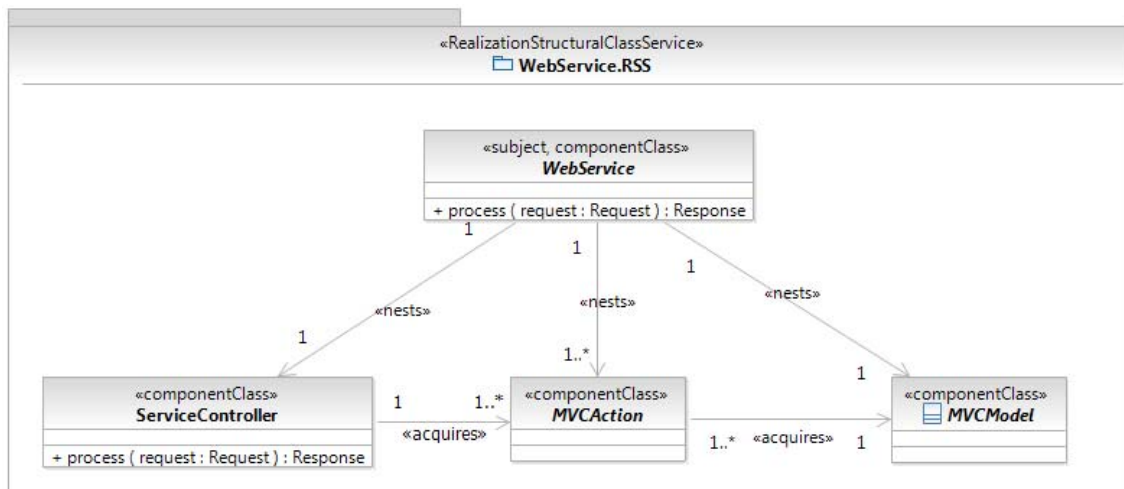


Figure 3.3 - The WebService Component with its sub-components: Service Controller, MVCAction and MVCModel

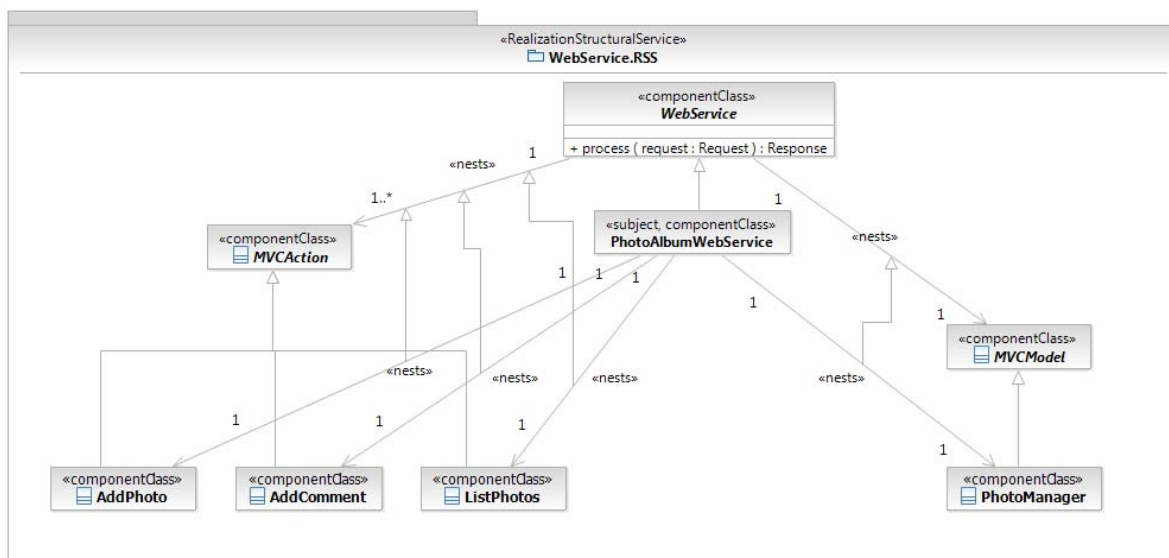


Figure 3.4 - Platform Independent Model of the of the Photo Album application WebService Component

In a typical implementation of this PIM to the Python/GAE PSM, we can map the *PhotoAlbumWebService* component directly to a Python class that extends the Python class *RequestHandler*, which belongs to the GAE application programming interface (API), as shown in Figure 3.5. This class would be required only to override the *post()* method, because it is the method that deals with *HTTP POST* requests (TheServerSide, 2009). This *post()* method accesses the objects *Request* and *Response* through the *RequestHandler* class, since they are its attributes.

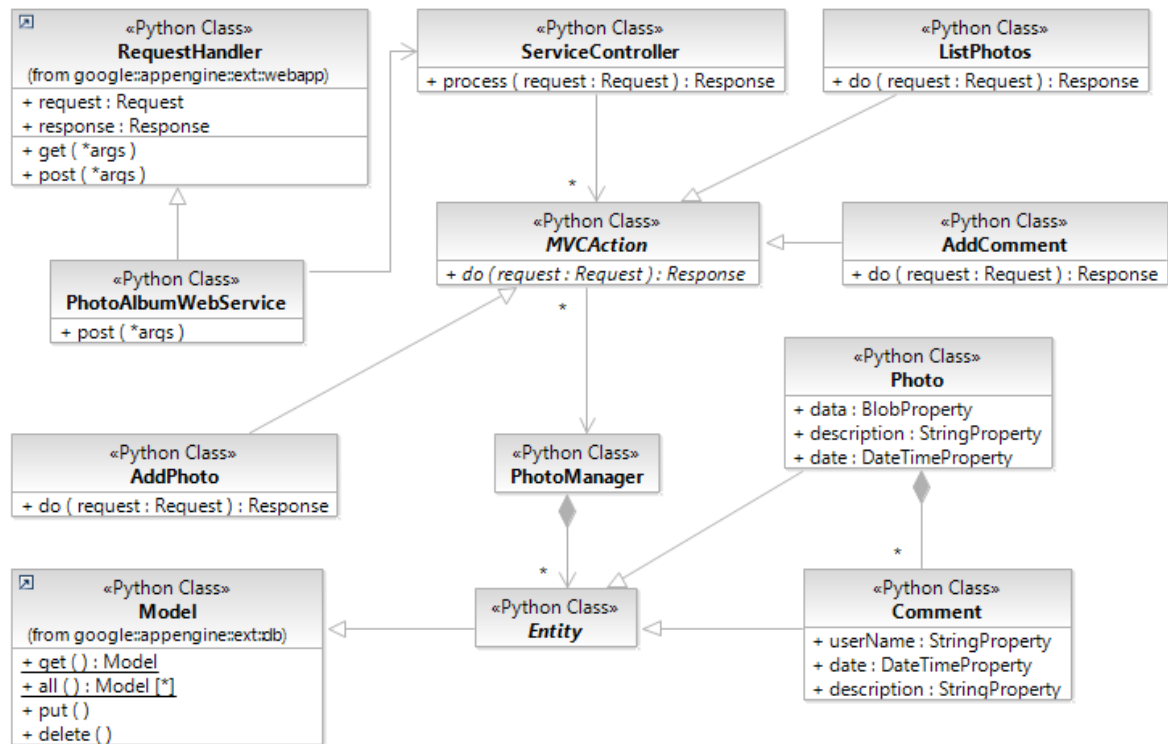


Figure 3.5 - Platform Specific Model of the Photo Album application WebService component

3.2.3.1. The ServiceController Component

The *ServiceController* component can be considered part of the controller role in the MVC architectural pattern at the server side. The *ServiceController* is provided by the same method of the *WebService* component because it has the responsibility to the services requested by the latter, as can be seen in Figure 3.3.

The task of the *ServiceController* is to provide a mapping between the client's requests and the actions requested by them. Because of this feature, this component has only one method, which is concrete, not requiring any specialization for the framework users. It will be necessary only to link the *MVCAction* components that will be defined latter in the application. The *process()* method of this component will check which is the action requested by the client, and call the *MVCAction*'s method *do()* of the corresponding action, passing the *Request* and *Response* objects received.

The mapping of the *ServiceController* component to a particular PSM can be linked directly to a class, for example, a Python or Java class, or the configuration/mapping files available in Java frameworks, such as *struts-config.xml* file for Struts Framework (Cavaness, 2004), where the mapping of the application actions is done.

In the case study for the KWAF framework, during the definition of the Photo Album application PIM it was not necessary to redefine the *ServiceController* component. For the PSM, this component was transformed into a Python class that have its *process()* method called by the *PhotoAlbumWebService* class. The corresponding action is obtained from the name of the action present in the Request object, so the relevant Action class responsible for provide that operation will have its *do(request:Request):Response* method called.

3.2.3.2. The MVCAction Component

The *MVCAction* component performs all the services (or actions) provided by the application, as shown in Figure 3.3. It represents part of the Model in the MVC architectural pattern, handling the business rules. The developer that uses the KWAF framework needs to specialize this component to every action that may be performed, so different applications using this framework will have different components that are specializations of the it. The *do(request: Request):Response* method should be specified to perform the action in question, and this is the method called by the *ServiceController*.

For the persistent data access, the *MVCAction* component manipulates the entities defined in the *MVCModel* component, for both: recovery and persistence of data. The mapping of this component to the PSM can occur, for example, in Java using MVC frameworks like Struts to its own *Action* classes, or even directly to a class, such as find in GAE/Python.

In the PhotoAlbum case study, when defining the PIM we defined three actions: add pictures, add comments and list photos. For that were created three components that extend *MVCAction*: *AddPhoto*, *AddComment* and *ListPhotos*, shown in Figure 3.4. For each of these action components was defined a Python class in the PSM, which implements the *do(request:Request):Response* method for the needed functionality.

3.2.3.3. The Component MVCModel

The *MVCModel* component is responsible for keeping the entities and data types of the application, which generally requires persistence capabilities. It represents the Model in the MVC architectural pattern. The *MVCModel* component has the class *Entity* which represents an entity of the application to be shared across all the Web application, in the server side as well on the client side. The *MVCModel* provide the functionality for persistence and data recovery, as shown in Figure 3.6.

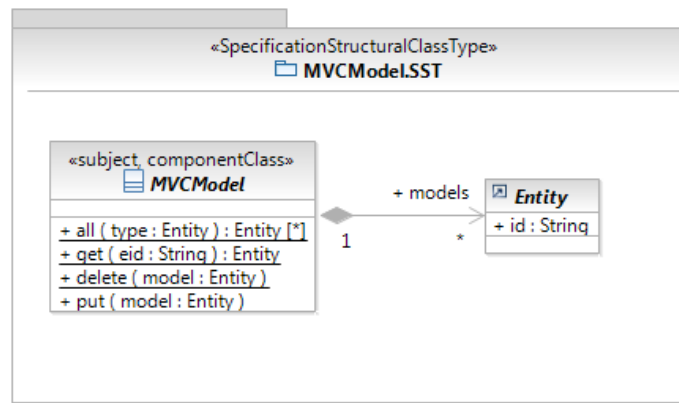


Figure 3.6 - The MVCModel Component Structure

For technologies that have a framework for object-relational data mapping, like the Java framework Hibernate (Bauer and King, 2004), the entities and the *MVCModel* should be mapped to Java Beans which Java/Hibernate annotations specifying persistence data location and constraints on its fields. The methods for recovery and persistence of *MVCModel* should be inserted into Data Access Object (DAO, 2009) classes that implement these features. Using Python for a PSM and GAE the entities can be mapped directly to classes that inherit from Expando Model, which already provides the methods for treatment of persistence, what does not need therefore to create DAO classes and does not need to map the *MVCModel*.

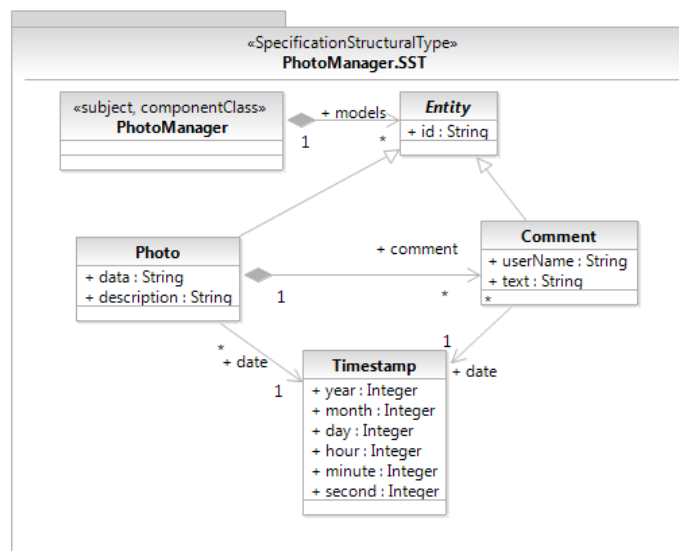


Figure 3.7 - Entities and data types for the MVCModel component, representing the Model in MVC

For the PhotoAlbum case study, we created the *PhotoManager* component, which is shown in Figure 3.4, and we created the persistent entities *Photo* and *Comment*, which are subclasses of *Entity*, shown in Figure 3.7. For the creation of the PSM, they were

mapped to Python classes that inherit from the Python/GAE Model class, thereby obtaining the functions needed for the persistence from GAE Model class, as can be seen in Figure 3.5

3.2.4 Structural Models of GUIClient Component

The *GUIClient* component, depicted in Figure 3.8, is responsible for modeling the client's side of the Web application and it is composed of the sub-components *MVCView*, corresponding to the view in the MVC architectural pattern and the *GUIController*, representing the client part of the controller role in the MVC pattern. *MVCView* is responsible for modeling the graphical user interface (GUI) and the *GUIController* is responsible for the modeling of events and treat user interaction with the application in the GUI. Note that we chose the architecture “push”, where actions that process events send data to the view layer so it can be shown to the users.

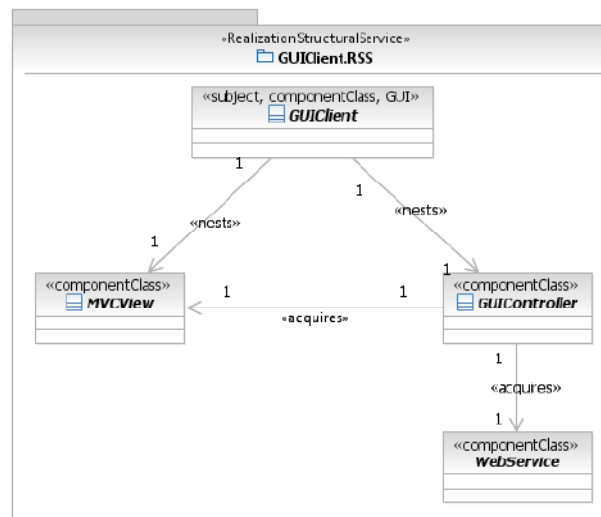


Figure 3.8 - GUIClient component structure

3.2.4.1. The Component MVCView

The *MVCView* component is composed of sub-components that represent the application windows and its navigation models. Each of these components is modeled using a framework for GUI modeling elements, called GUIPIMUF (GUI PIM Profiled UML2 Framework). The GUIPIMUF contains a number of elements for modeling the structural, navigation, and behavioral aspects of the GUI here we present the generated GUI models with its elements.

The navigation model of the PhotoAlbum Web application is shown in Figures 3.9: from the *MainWindow*, the user can navigate to the *PhotoViewer* when he/she clicks on

any picture (image element type), or open the dialog window to select a new image, using an *imageChooser*, when he/she clicks on the *addPhoto* button.

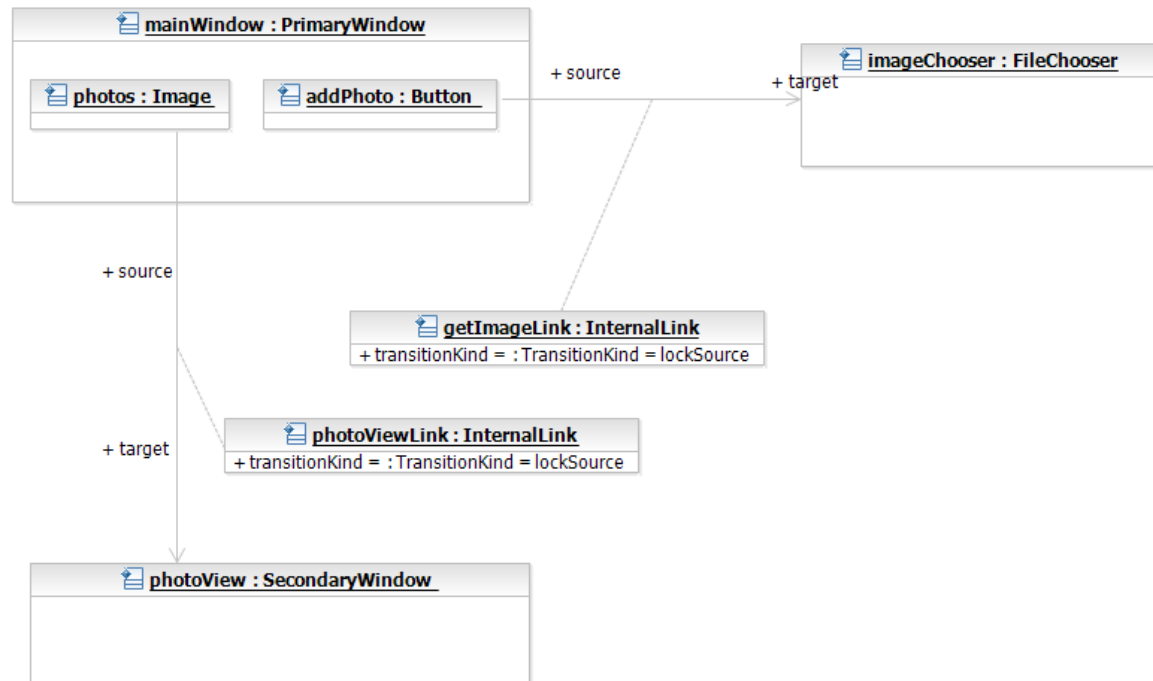


Figure 3.9- PhotoAlbum GUI Navigation Model

An example of component windows modeled with the GUIPIMUF can be seen in Figure 3.10 – the model for the secondary window PhotoViewer, which aims to present to the user a photo and its comments. Note that all models are compatible with UML2 and they were produced using the IBM Rational Software Modeler (RSM, 2009).

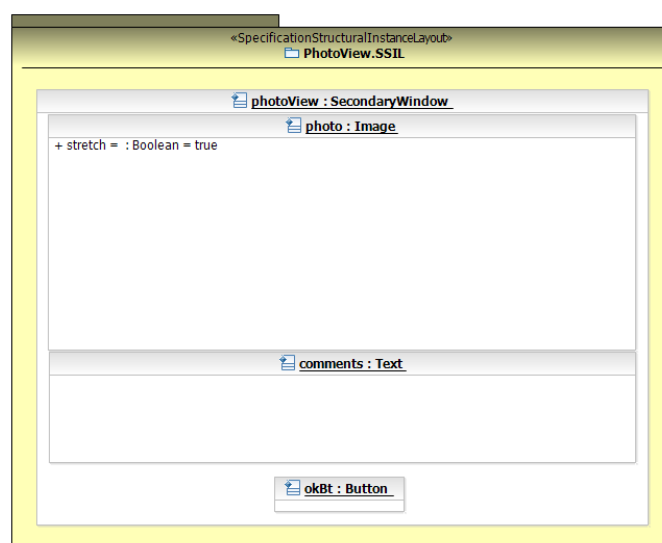


Figure 3.10 - PhotoView Component Structural Model

3.2.4.2. The Component GUIController

The *GUIController* component is responsible for making the connection between the GUI elements and the server side of the web application. It is composed by several sub-components carrying out the mapping between actions performed by users (for mouse events, keyboard, window, etc.) and calls to the server (WebService) as well the handling of presentation logic, such as control widgets behavior or appearance. Figure 3.11 shows the *GUIController* component and an example of a sub-component to control user actions. The *EventListener* controller checks if generic actions/events like mouse clicks or a keyboard inputs such as the ‘Enter’ key being pressed when a widget presented in the screen such as a menu item, buttons or text field, was focused. The behavior of these actions is defined with the use of OCL (Warmer and Kleppe, 2003).

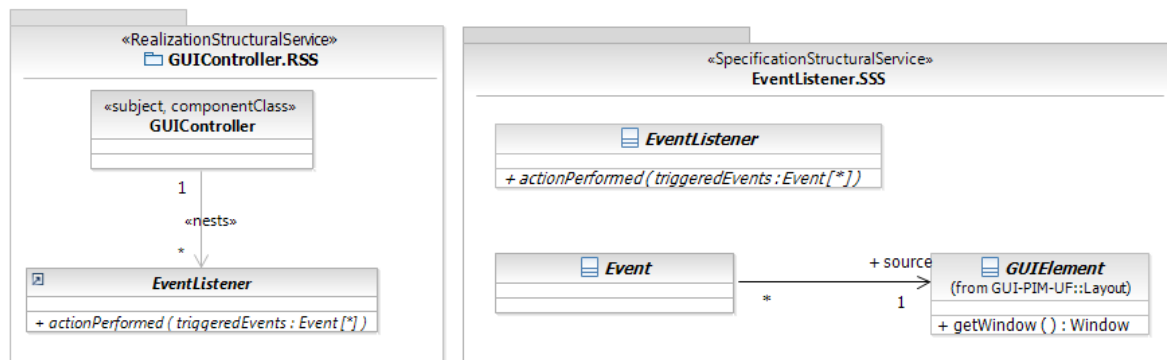


Figure 3.11 - GUIController Component Structural Model

Below (Figure 3.12) we present the structural model of the Photo Album *GUIController* application (*PhotoAlbumGUIController*). Note that the behavior of each user action mapped to the controller is specified by OCL constraints. In the example model, the *actionPerformed()* method of the component *RemoveButtonActionEvent* was specified, defining that it should create a request and calls the *process()* service of the *PhotoAlbumWebService* component.

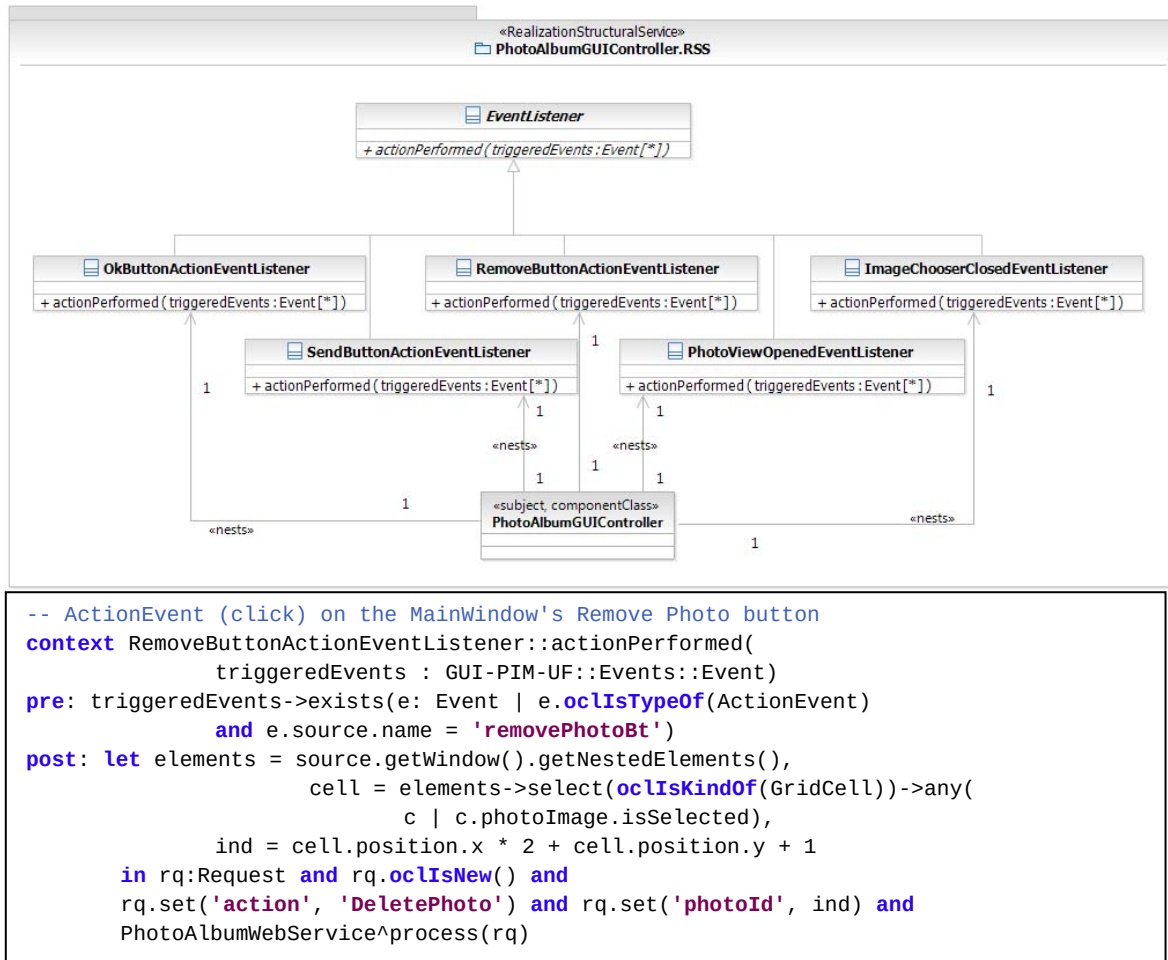


Figure 3.12 - PhotoAlbumGUIController Component and the Definition of one Action using OCL

3.2.5 KWAf Assessment Remarks

This case study aimed to define and evaluate a PIM framework called KWAf, which is a framework for modeling and development of Web applications that cover modeling aspects for both: the server and client side of a web application. This framework simplifies the development of new Web applications, reducing the specification time, increasing productivity, making the developer to be concerned only with the business aspects of the application, disregarding the Web applications features. This consequently reduces the development time of new models and simplifies the overall development process.

Another important point is the fact that specific models for a given platform can be automatically generated through the adoption of the framework, once it standardizes common aspects found in a variety of platforms. The framework can also be extended to address issues relevant to any kind of particular applications.

In this research we also noted that although the framework simplifies the development of Web applications using a model driven architecture, the lack of adequate tools to create the model acts as a barrier to this process. The current UML tools do not allow you to easily make an effective modeling, focusing on one concern at a time. For example, the user cannot to be concerned first with the structural aspects of the application and only afterwards to worry about the operational, functional and behavior aspects of the application, without having to rename multiple model elements that were supposed to be the same or propagating changes through the models.

The generation of 100% of code is another barrier found in the current tools, which completely ignore the interaction behavior models and generate only a sketch of the structural part of the application. Another problem found with the use of current tools is the difficult to reuse an existing model.

These aspects were the main motivation to define a better tool, which could minimize the effects and lack of features found on current tools. This tool is defined on Section 3.3 and as it is defined as a web application, it is also modeled as an instance of the KWAF framework.

3.3. WAKAME Top-Level PIM

In this section the Platform Independent Model of WAKAME is presented, using the Kobra2 method. We will explain how each component of the tool has been specified, showing additionally the interactions among these components.

The models were built using the *IBM Rational Software Modeler* version 7.0.5 (RSM, 2009) modeling tool, and are available for reading in the open-source repository of Google Code in <http://kobra2.googlecode.com/svn/trunk/workspace/WAKAME/>.

3.3.1 WAKAME as an Instance of KWAF

The PIM of WAKAME was defined using the *KWAF* architecture, and by so, WAKAME has been designed as a web tool, and its responsibilities were decomposed according to the MVC pattern and the client/server architecture used in KWAF. In this sense, the client would be the components of the Graphical User Interface (GUI) while the server would be the component responsible for running the services requested by the GUI component.

The strategy adopted for reusing both the architecture as the components of KWAF was to merge the contents of its UML package within the WAKAME package, as shown in Figure 3.13. Through this merge, we can redefine the components defined in KWAF for our application, adding the necessary features to them.

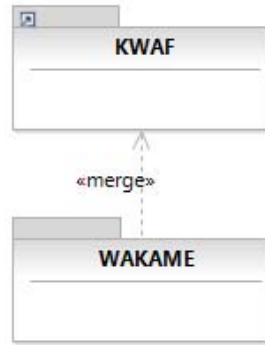


Figure 3.13 - WAKAME Top-Level

As shown before, KWAF has the structure of components presented in the Figure 3.14.

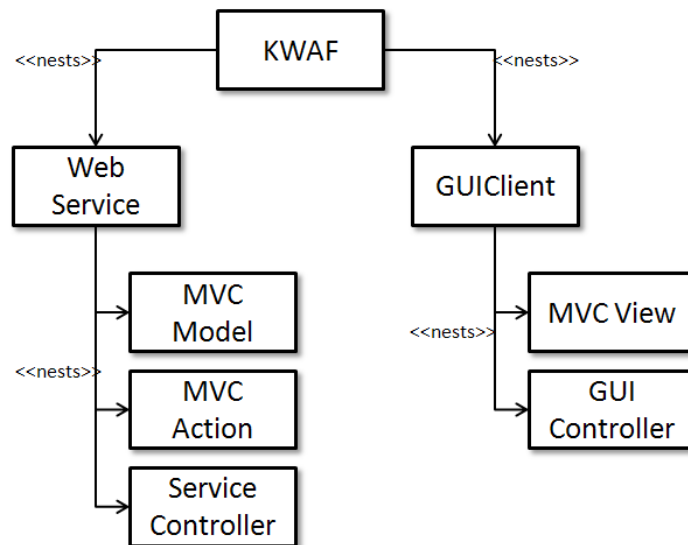


Figure 3.14 – KWAF component structure.

In the next subsections we will present how these components of KWAF were redefined to create the PIM of WAKAME, through the views that have been defined for each component, in accordance with the Kobra2 method.

3.3.2 The WAKAME Component

According to Kobra2 method, a model is composed by nesting minor sub-components, and for this specific model, the top-level component is the *WAKAME*. This component will encapsulate all other sub-components of the application. Furthermore, it specializes the abstract KWAF component, at the *Specification Structural Class Service* view, as depicted in Figure 3.15. By doing this, we elegantly define in our top-level component, *WAKAME*, all the architecture provided by KWAF, nevertheless remaining necessary the additional specification of its sub-components. It is worth to notice that *WAKAME* has no methods, because it is a self-contained application that allows only user interaction.

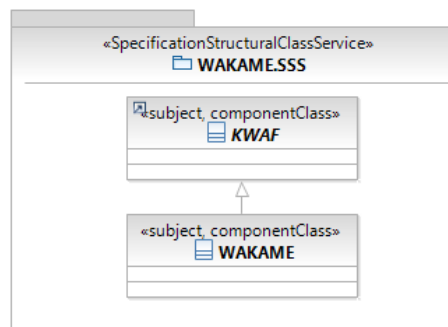


Figure 3.15 - WAKAME Specification Structural Class Service View

In the realization of the *WAKAME* component, *Realization Structural Class Service* view (Figure 3.16), the sub-components *WebService* and *GUIClient* of KWAF are redefined to *WAKAMEWebService* and *WAKAMEGUIClient*, respectively, where the former is the responsible for providing all services of the application for the second sub-component. In the latter, in turn, will be presented: (i) the user interface of the application, (ii) how the user interaction will take place, and (iii) the navigation among the windows. A third abstract sub-component not addressed by the KWAF component is the *OCLEngine*, which boils down the abstraction of an OCL engine to assist in the validation and query the components in the application. These sub-components will be explained in subsequent sections.

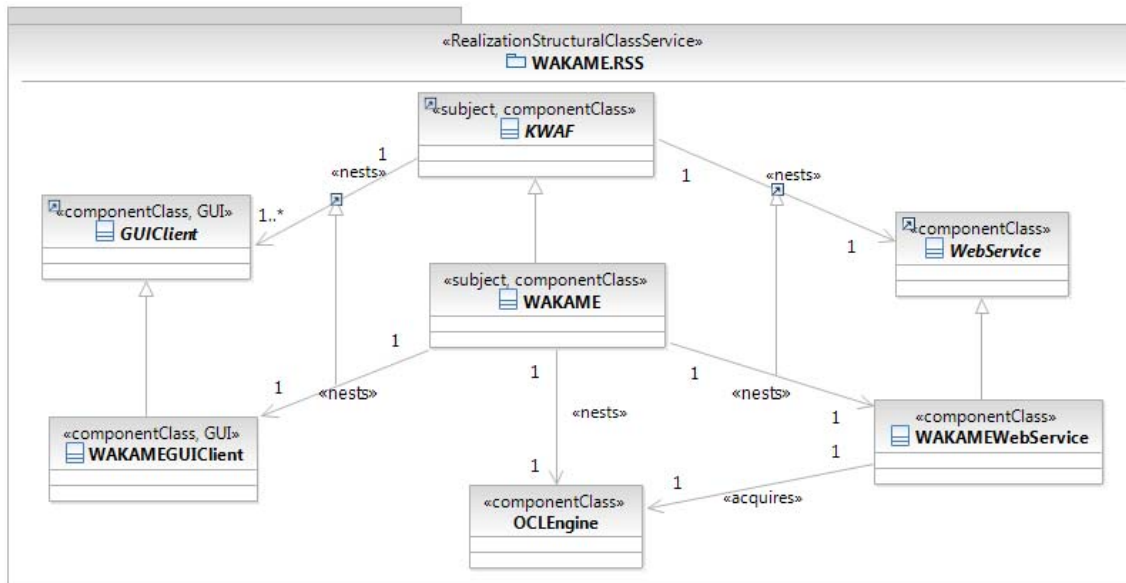


Figure 3.16 - WAKAME Realization Structural Class Service View

The entities handled by this application have been defined in *Realization Structural Class Type* view, Figure 3.17, since they will be manipulated by both sub-components, WAKAMEWebService and WAKAMEGUIClient. In this view, the package WAKAME.RST merges the content of Kobra2 package, to be able to represent a model in Kobra2, and Diagram Interchange (DI) of the OMG (DI, 2005), to allow graphical representation of each element of the model.

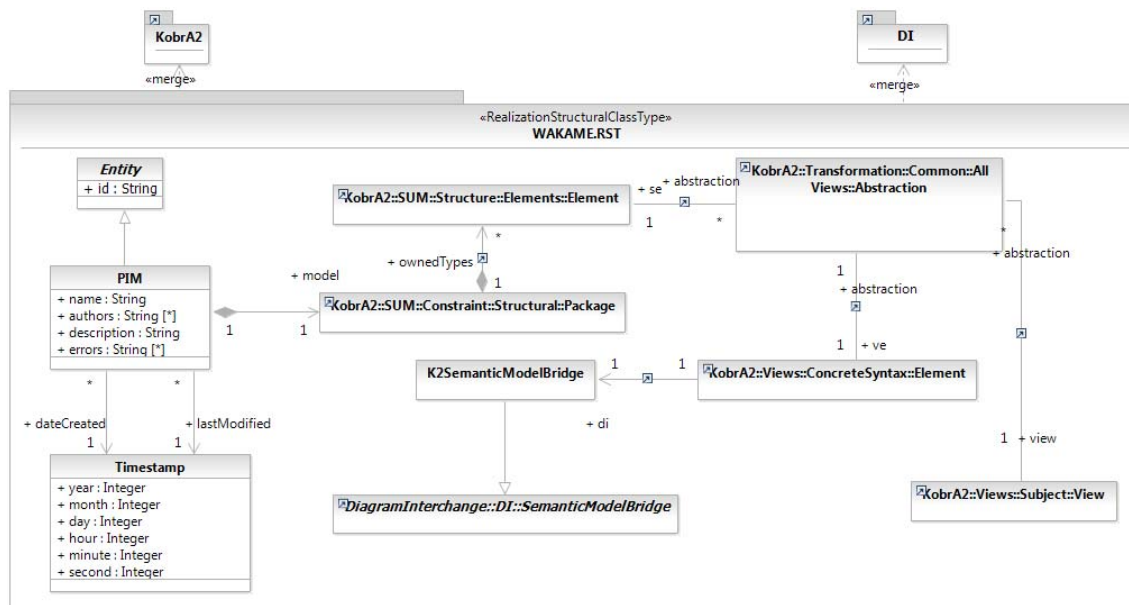


Figure 3.17 - WAKAME Realization Structural Class Type View

The main entity of this application is the PIM, defined by any arbitrary user with the tool WAKAME. It inherits the abstract class Entity of KWAF, ensuring its persistence. It possesses the following attributes:

- **name** – the PIM model name, labeled by the proper user;
- **authors** – the list of authors of the PIM model;
- **description** – a description of the model;
- **errors** – errors and warnings when validating the model;
- **dateCreated** – a time indication about when the model was built;
- **lastModified** – a time indication about when the model was modified for the last time; and
- **model** – represents the proper Kobra2 model elements under discussion.

The visual representation of the model elements will be stored in a Package (*KobrA2::Constraint::Structural::Package*) of KobrA2, through the composition *model*. This package encapsulates all these elements.

A KobrA2 model is constituted by the *Single Unified Model* (SUM), which integrates the views in a single representation for the model application. The SUM may only be modified indirectly, through changes on views. These, in turn, represent only a partial copy of the elements in SUM, showing only a particular point of view. For each component in SUM, there may be different Views (defined by KobrA2) and the coherence between the Views and SUM is ensured through transformations of models between them.

The connection between the SUM and Views is made through *Abstraction*, which connects each element of a View (*KobrA2::Views::ConcreteSyntax::Element*) with its corresponding SUM element (*KobrA2::SUM::Structure::Elements::Element*). However, as each element of the SUM may appear in different Views, it can have several *Abstractions* and, as expected, each one also has a link to the View it represents (*KobrA2::Views::Subject::View*).

In order to store the graphical representation of each element of a View, (e.g., position, size, color, among others), we use the *K2SemanticModelBridge*. This class extends the *SemanticModelBridge* class defined in the DI metamodel, avoiding the loss of visual information, which could jeopardize the readability.

The WAKAMWebService components (focus of this work) will be detailed in Section 4.1, while the specification of the WAKAMEGUI component will be accomplished by the other collaborator of this project (Marinho, 2009).

3.4. Chapter Remarks

This chapter exhibited the cornerstones of the WAKAME tool. It showed the principles and goals for the WAKAME tool, a framework for PIM modeling of Web applications, which was the base architecture for the WAKAME tool and the evaluation of the framework in a case study, which was the modeling of a Photo Album web application.

CHAPTER 4

THE

WAKAMEWEBSERVICE

AND THE MODEL

REPOSITORY

In this chapter, we explain how the WAKAMEWebService has been modeled, showing its sub-components and their interactions, and finally detailing the activities undertaken to implement it.

4.1. The KWAF instance Platform Independent Model for the WAKAMEWebService and Model Repository

The WAKAME tool was planned since its beginning to be a Web Application. Its models were conceived on top of the WebApp framework, which has been described in Chapter 3. The overview of the WAKAME tool models are also depicted in Chapter 3.

The main components of the WAKAME Server are depicted in the following sections: Section 4.2.1, Section 4.2.2 and Section 4.2.3.

4.1.1 The Component WAKAMEWebService

The *WAKAMEWebService* component is responsible for processing all the services requested by *WAKAMEGUIClient* component, as well as being responsible for the persistence of the data. Besides specializing the *WebService* component of KWAF in the *Specification Structural Class Service* view, Figure 4.1, it does not redefines the method *process(request: Request): Response*, because it is already fully specified in the framework of KWAF.

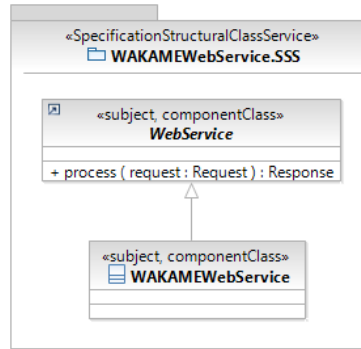


Figure 4.1 - WAKAMEWebService Specification Structural Class Service

In the realization of the component *WAKAMEWebService*, namely the *Realization Structural Class Service* view shown in Figure 4.2, the sub-component responsible for the persistence of data, the *WAKAMEModel*, implements the abstract *MVCModel* component. Additionally, the *MVCAction KWAF* componentClass is a generalization for the following *WAKAMEWebService* sub-components: the *PIMManipulationAction*, *ViewManipulationAction* and *ModelIOAction*, all of them responsible for processing the available services of the tool.

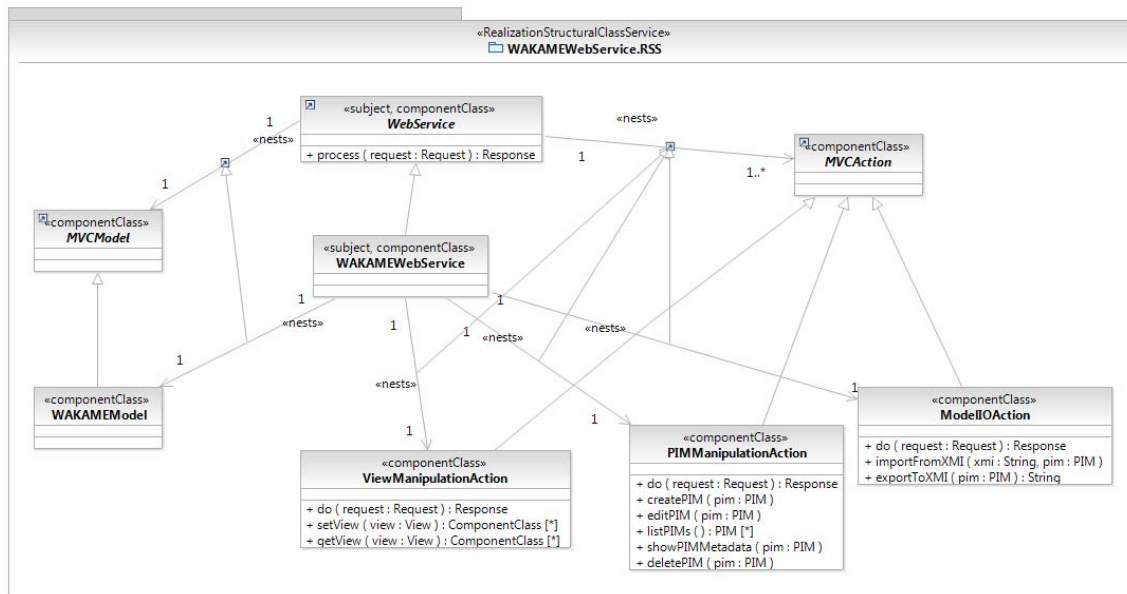


Figure 4.2 - WAKAMEWebService Realization Structural Class Service

It appears that the specification of some sub-components of *WAKAMEWebService* does not need be displayed. Actually, it happens these specifications were already done in the *KWAF*, as the *ServiceController* component, responsible for mapping requests in the corresponding actions.

4.1.2 The Component WAKAMEModel

The *WAKAMEModel* component, as told before, is the one responsible for the data-persistence. Through it, an instance of PIM can be saved or recovered. By specializing *MVCModel* KWAF component, the *Specification Structural Class Service* view (Figure 4.3), all necessary methods to ensure the persistence are inherited directly, and therefore need not be redefined.

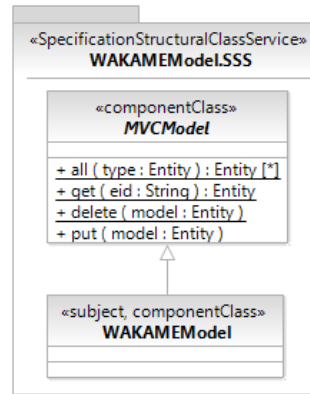


Figure 4.3 - WAKAMEModel Specification Structural Class Service

The model at *Specification Structural Class Type* view of the *WAKAMEModel* Component, Figure 4.4, restricts this componentClass to hold only PIM entities. Graphically, the composition which associates the *MVCModel* to Entity is specialized into the one relating *WAKAMEModel* to, exclusively, PIM entities. This guarantees that the *WAKAMEModel* component will go only to deal with entities of type PIM.

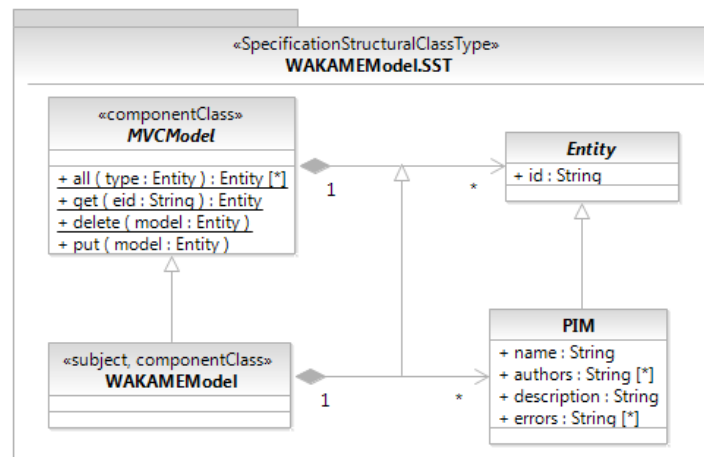


Figure 4.4 - WAKAMEModel Specification Structural Class Type

Before moving to the next component, the reader may question the lack of realization views for this component. However, as the *WAKAMEModel* only (1) inherits

the methods of *MVCModel*, (2) specializes the composition from this componentClass to Entity, at the KWAF level, and above all, (3) has no sub-components, these views simply do not exist.

4.1.3 The Component *PIMManipulationAction*

All activities related to the manipulation of PIM model instances defined in the tool are represented by the component *PIMManipulationAction*. As it is a component that will provide services (or actions), it specializes *MVCAction* component of KWAF, as shown in the *Specification Structural Class Service* view depicted in Figure 4.5. The communication between the solicitant of the action (the *WAKAMEGUIClient*) and the action itself (some specialization of *MVCAction*) is done by the method *do(request:Request):Response*, invoked by the ServiceController. As input parameter (request), the method takes the action to be done.

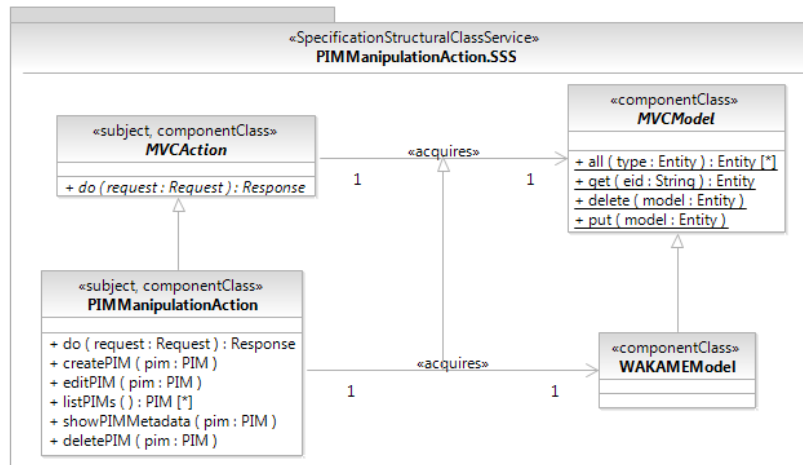


Figure 4.5 - *PIMManipulationAction* Specification Structural Class Service

However the *PIMManipulationAction* component will be responsible for providing more than one service, therefore another necessary methods was defined. In this case, the method *do()* will be responsible for redirecting the action to the solicitant, indicated by the parameter *subaction* given through the parameter request, and as response, it will also be responsible for defining an object of *Response* type. One pre-condition addressed by this method ensures the request parameter be a *subaction*.

The operational behavior pre/post conditions attached to each one of these methods can be found below, in the *Specification Operational Service* view (Figure 4.6).

```

context PIMManipulationAction::do(request: Request): Response
pre: not request.get('subaction').oclIsUndefined()
post:
    result.oclIsKindOf(Response) and result.oclIsNew() and
    request.get('subaction') = 'createPIM' implies
        result^write('ok') and self^createPIM(request.get('model'))
    and request^get('subaction') = 'editPIM' implies
        result^write('ok') and self^editPIM(request.get('model'))
    and request^get('subaction') = 'listPIMs' implies
        result^write(self.listPIMs())
    and request^get('subaction') = 'showPIMMetadata' implies
        result^write(self.showPIMMetadata(request.get('model')))
    and request^get('subaction') = 'deletePIM' implies
        result^write('ok') and self^deletePIM(request.get('model'))

context PIMManipulationAction::createPIM(pim: PIM)
pre: pim.id.oclIsUndefined()
post: self.mvcmodel^put(pim)

context PIMManipulationAction::editPIM(pim: PIM)
pre: not self.mvcmodel.get(pim.id).oclIsUndefined()
post: self.mvcmodel^put(pim)

context PIMManipulationAction::listPIMs() : Set(PIM)
post: result = self.mvcmodel.all()

context PIMManipulationAction::showPIMMetadata(pim: PIM): PIM
pre: not self.mvcmodel.get(pim.id).oclIsUndefined()
post: result = self.mvcmodel.get(pim.id)

context PIMManipulationAction::deletePIM(pim: PIM)
pre: not self.mvcmodel.get(pim.id).oclIsUndefined()
post: self.mvcmodel.delete(pim)

```

Figure 4.6 - PIMManipulationAction Specification Operation Service

With regard yet to *PIMManipulationAction* component, it must acquires the services of *WAKAMEModel*, because it will call the methods responsible for the entity persistence. Hence, this component will be responsible for keeping the data from the model, such as name, authors and others. Handling the model elements is the liability addressed by the component *ViewManipulationAction*, which will be explained in more detail throughout this master thesis. The methods in *PIMManipulationAction* were:

- ***createPIM(pim: PIM)*** – responsible for creating a new model (PIM). It takes as parameter an instance of the PIM entity with the initial valuation for the attributes, and the object, then, turns persistent. To register a PIM entity, its identifier must be unique, that is, none object have been created with the same id.
- ***editPIM(pim: PIM)*** – responsible for editing an existing model. The data that this method changes are: name, authors, description and last modified date. As input parameter, the method takes an instance of PIM entity together with the values to be modified. The pre-condition is that there is some model with the id parameterized.

- ***listPIMs()*** – this method will find all instances stored in a PIM and return them in a list, if any.
- ***showPIMMetadata(pim: PIM): PIM*** – this method will fetch data from a given model. While as input, it takes an identified PIM instance, as output it returns another PIM instance, but with filled data. The soundness of this method lies in the identified model input.
- ***deletePIM(pim: PIM)*** – allow (only) identified PIM instance to be deleted from the diagram.

4.1.4 The Component ViewManipulationAction

This component will be responsible for handling the elements defined by the user in the modeling tool. As for the methodology *KobrA2*, the model is accessed and modified from the Views, two actions have been identified for this component: one to retrieve a View to be displayed to the user through the component *WAKAMEGUIClient*, and another to transmit the changes carried through in any view to the SUM.

The *Specification Structural Class Service* view (Figure 4.7) of the *ViewManipulationAction* shows that this component specializes *MVCAction*. Besides redefining the *do()* method, it redirects the execution of the action to the appropriate method according to the parameter *subaction*. In response, this method will send the (i) return of the methods called, (ii) a list of possible inconsistencies of the model and also (iii) a list of components, so that the GUI can build the navigation tree between them.

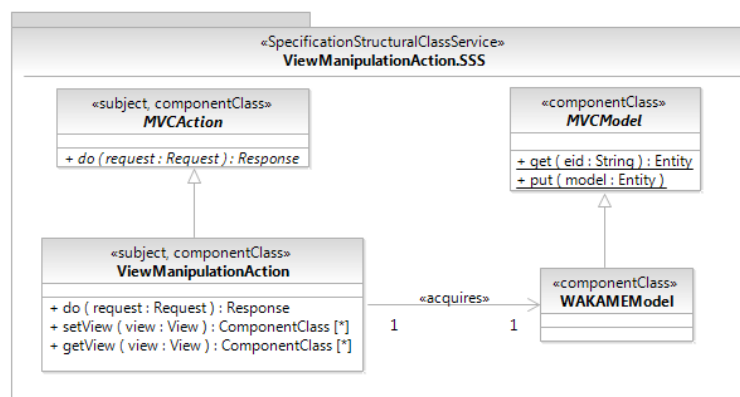


Figure 4.7 - ViewManipulationAction Specification Structural Class Service View

In the *Realization Structural Class Service* view (Figure 4.8), the *ViewManipulationAction* nests a sub-component *Transformations*, responsible for aligning

the changes between the SUM and the Views. More specifically, it yields how the SUM derives the required View and also how to update it whenever a change is made in any View. In order to access the instances of the model already saved and save them again after the changes, the component *ViewManipulationAction* acquires the services of *WAKAMEModel* component. Moreover, the actions that this component performs should return a list of *ComponentesClass* that constitutes the navigation tree among components of the tool, because potentially these updates could have modified the tree.

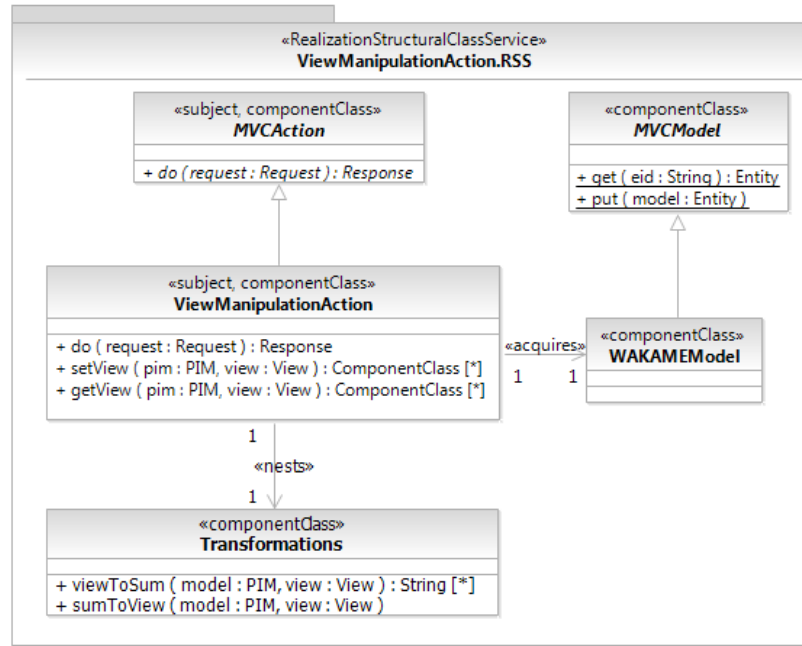


Figure 4.8 - ViewManipulationAction Realization Structural Class Service

Following, the *Realization Operational Service* view (Figure 4.9) highlights the conditions regarding each method defined in the *ViewManipulationAction* component.

The method *setView(pim: PIM, view: View): Sequence (ComponentClass)* is responsible to bring up to date the SUM from a View modified by the user. This method receives as parameter an object containing the PIM model to be modified, which is recovered from the id, and a View object containing the changes made by the user. To update the SUM, the method *setView()* delegates the changes to the method *viewToSum()* of the *Transformation* sub-component . Also the method *setView ()* is responsible for saving the model changed with the updates made to the SUM, and finally returns a sequence of *ComponentClass*, which will compose the tree of components.

```

context ViewManipulationAction::do(request: Request): Response
pre: not request.get('subaction').oclIsUndefined()
    and not self.mvcmodel.get(request.get('model').id).oclIsUndefined()
post: let pim: PIM = self.mvcmodel.get(request.get('model').id),
      view: View = request.get('view') in
      result.oclIsKindOf(Response) and result.oclIsNew() and
      request.get('subaction') = 'getView' implies
          result^write(self.getView(pim, view)) and result^write(view)
      and request.get('subaction') = 'setView' implies
          result^write(self.setView(pim, view))

context ViewManipulationAction::setView(pim: PIM, view: View): Sequence(ComponentClass)
post: self.transformations.viewToSum(pim, view) and self.mvcmodel.put(pim) and
      result = pim.model.ownedType->collect(
          cc:ComponentClass | cc.oclIsTypeOf(ComponentClass))

context ViewManipulationAction::getView(pim: PIM, [inout] view: View):
                                          Sequence(ComponentClass)
post: self.transformations.sumToView(pim, view) and
      result = pim.model.ownedType->collect(
          cc:ComponentClass | cc.oclIsTypeOf(ComponentClass))

```

Figure 4.9 - ViewManipulationAction Realization Operational Service

The method *getView(pim: PIM, [inout] view: View):Sequence (ComponentClass)* searches the SUM seeking a specific View. For this, it takes as input parameter the model (PIM) and a View object with the necessary information to determine what type of View is requested, and what component this View belongs to. To avoid redundancy, the View parameter is typed with input/output modifier, in such a way it will have the necessary data for the search of the View, so as it will serve to return the version updated. Internally, to search one View, the method *sumToView ()* of *Transformations* sub-component will carry out the necessary changes required to meet the request. Just as the other, the method *getView()* returns the sequence of *ComponentClass*.

4.1.4.1. The Component Transformation

In Kobra2, changes made in Views are transmitted to the SUM through several transformations of models that were specified in the metamodel Kobra2 by OCL expressions. Not only the replication of changes from the Views to SUM is given by transformations, but also the election of which information of SUM will appear in one View. It is worth to notice that the integrity of the SUM is ensured by these transformations too.

These transformations, defined in the Kobra2, can be categorized in three types:

- **SUM to View** – the election of which elements of SUM and what information regarding these elements will be shown in a View for the component under discussion.
- **View to SUM** – it yields the transformations to align the SUM with changes done in a view. It specifies which elements need be created, updated or removed from the SUM as well as the information about these elements subject to change.
- **SUM to SUM** – the transformations used to validate the SUM integrity as a whole. After a normal View to SUM transformation, some inconsistencies can be inserted. It is then up to this transformation to detect the failures of integrity, so that messages and warning can help the user to fix the problems find out.

In the Kobra2 metamodel, the rules of transformations are defined in terms of OCL expressions. The element *Abstraction* can be composed of a *TransformationExpression*, as shown in Figure 4.10. This latter element stores the expression of transformations for each entity within the SUM. Again, the same figure shows some Ocl invariants highlighting an example of transformation for the *ComponentClass* concerning the *Specification Service View*.

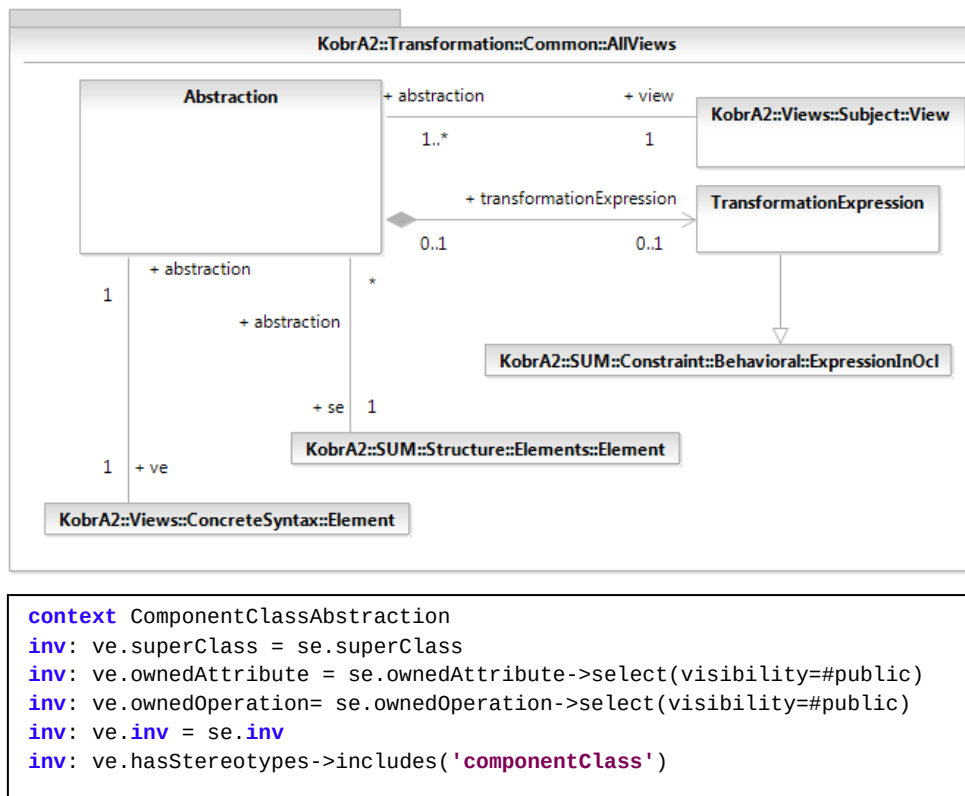


Figure 4.10 – Transformations Abstractions and a Expression Transformaton example.

In this sense, the component *Transformation* had been defined within the component *ViewManipulationAction* to hold these transformations. To make this possible, the component acquires the *OCLEngine* componentClass for evaluation of the OCL expressions of the transformations defined in the Kobra2 metamodel, as shown in the *Specification Structural Class Service* view (Figure 4.11).

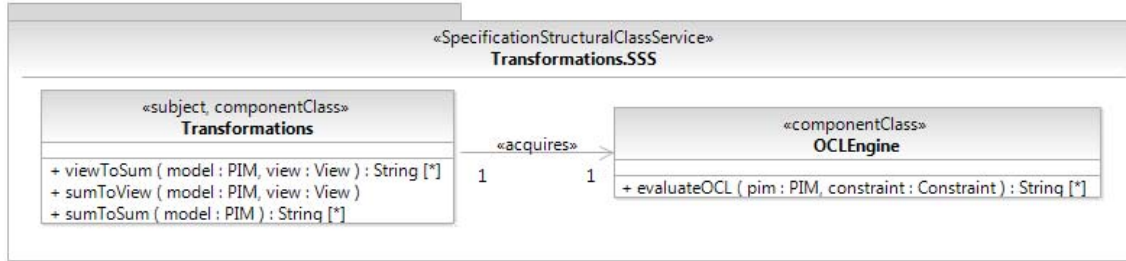


Figure 4.11 - Transformations Specification Structural Class Service

For each transformation, the *Specification Operational Service* view (Figure 4.12) addresses one correspondent method. The first method, *viewToSum([inout] pim: PIM, view: View): Set (String)* is responsible for making updates to the SUM in accordance with the changes in the View. Through this method, for each element of View, the correspondent entity in the SUM is taken (if no entity exists one will be created), based on the abstraction association that connects the two elements. Following, this abstraction fills the *evaluateOcl()* parameters with the expression of transformation (*transformationExpression*) and the corresponding model to be modified. This method will evaluate the OCL expression, and will further perform the transformation, taking into account all the necessary changes at SUM. After updating, SUM enters in the integrity consistency phase through the call to method *sumToSum()*. The argument of return, the inconsistency messages, is used as the output of method *viewToSum()*. This later takes as input parameters the PIM model (also an output parameter with the modifications made to the SUM) and a View instance with the changes that will be used.

The method *sumToView(pim: PIM, [inout] view: View)* is responsible for carrying out the change recovering a View of a component. As input parameter, this method takes the model and an instance of the View with the information to identify the requested one. This parameter also assumes an output direction, with the View from the SUM. For each element in the subject component of the View, the method *evaluateOCL()* of *OclEngine* will be invoked based on the transformation expression attached to the Abstraction of the view.

```

context Transformations::viewToSum([inout] pim: PIM, view: View): Set(String)
post: view.ownedElement->forAll(
    ve:ViewElement | pim.model.ownedType->any(
        se:SUMElement | se.abstraction->any(a | a.ve = ve)) and
        oclengine^evaluateOCL(pim, ve.abstraction.transformationExpression))
and result = self.sumToSum(pim)

context Transformations::sumToView(pim: PIM, [inout] view: View)
post: view.subject.packagedElement->forAll(
    se: SUMElement | oclengine^evaluateOCL(pim, se.abstraction->any(
        a: Abstraction | a.view = view).transformationExpression))

context Transformations::sumToSum(pim: PIM) : Set(String)
post: result.oclIsKindOf(Set(String)) and result.oclIsNew() and
    pim.model.ownedType->forAll(
        se: SUMElement | result->include(oclengine.evaluateOCL(
            pim, se.abstraction->any().transformationExpression)))

```

Figure 4.12 - Transformations Specification Operational Service

Finally the method *sumToSum (pim: PIM): Set (String)* implements the rules to ensure the entire consistency of the SUM. It can investigate every part, assessing the *transformationExpression* by the *evaluateOCL()* method, detecting and returning all the inconsistencies found.

4.1.5 The Component ModelIOAction

For services like export and import the models defined in the tool, there is the *ModelIOAction* component. The standard used for exchange of models is *the XML Metadata Interchange (XMI)* (2007), which is the already used standard for exchanging models between most modeling tools. Being a component that will provide services to the component *WAKAMEGUIClient*, it specializes *MVCAction* in the Specification Structural Class Service view (Figure 4.13). Again, it is through the method *do()* that each processing action will be guided for the appropriate method by the parameter *subaction*.

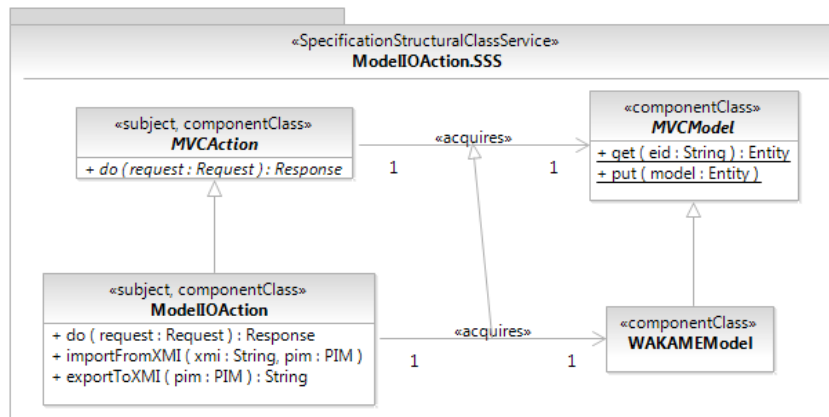


Figure 4.13 - ModelIOAction Specification Structural Class Service

We have defined the following methods for the actions that this component will provide, which are fully specified in *Specification Operational Service* view (Figure 4.14):

- ***importFromXMI(xmi: String, pim: PIM)*** – this method takes a model in XMI format to import the new model into the tool. As can be noted in the signature of the operation, two parameters are needed: a string in XMI format, which will have the information of the model to be imported, and an instance of PIM containing information such as name of the model, authors, and others to create it.
- ***exportToXMI(pim: PIM): String*** – this method is responsible for exporting the information from a model in XMI format. It takes as input parameter an instance of PIM with the id of the model to be exported. This method returns a string in XMI format.

```
context ModelIOAction::do(request: Request): Response
pre: not request.get('subaction').oclIsUndefined()
post: result.oclIsKindOf(Response) and result.oclIsNew() and
      request.get('subaction') = 'importFromXMI' implies
      result^write('ok') and self^importFromXMI(request.get('xmi'),
      request.get('model'))
      and request.get('subaction') = 'exportToXMI' implies
      result^write(self^exportToXMI(request.get('model')))

context ModelIOAction::importFromXMI(xmi: String, pim: PIM)
pre: pim.id.oclIsUndefined()
post: pim.model = xmi.toModel() and self.mvcmodel^put(pim)

context ModelIOAction::exportToXMI(pim: PIM): String
pre: not self.mvcmodel.get(pim.id).oclIsUndefined()
post: result = self.mvcmodel.get(pim.id).toString()
```

Figure 4.14 - ModelIOAction Specification Operational Service

4.1.6 The Component OCLEngine

In order to implement the changes described in the Kobra2 metamodel, we use the abstraction of a component for validation and evaluation of OCL expressions, the OCLEngine. The idea is to abstract the features existing in the various OCL engines (Dresden, 2009; OCLMDT, 2009; OCLE, 2009; Octopus, 2009) to meet the needs of this application.

For this, we have define a single method on the component *OCLEngine*, the *evaluateOCL* (*[inout] pim: PIM, constraint: Constraint*): *Set (String)* (Figure 4.15), since it takes the model under validation, and a constraint (defined in the metamodel Kobra2)

with some extra information, as the element that contains the restriction, the OCL expression, among others.

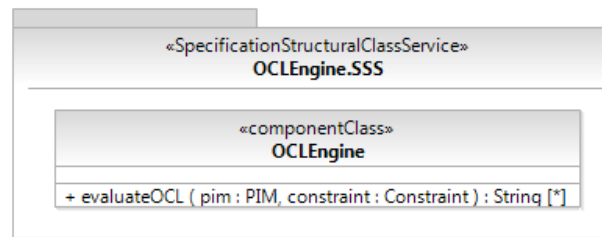


Figure 4.15 - OCLEngine Specification Structural Class Service

For the sake of expediency, this abstract method would perform the evaluations of OCL expressions, and based on the results, it would return the validations messages, but also implement corrective measures from the validation. Just the *Specification Structural Class Service* view (Figure 4.15) was considered here, because this component is an interface representing a possible OCL engine available.

4.2. The WAKAMEWebService and the Model Repository Implementation

To implement the model repository and the WAKAMEWebService, some planning tasks was need:

- Setting the platform for implementation of WAKAMEWebService;
- Evaluate and choose among the tools supporting the creation of repositories of models, the one most appropriate to the requirements of WAKAME and the platform chosen;
- Development of the Model Repository;
- Deployment of WAKAMEWebService over the platform and the repository of models defined;
- Evaluate and choose among the tools and technologies to transformation of models, the one with the best adaption to the needs of WAKAME to assist the development of transformations defined in the KobrA2 metamodel;
- Implement the changes defined in KobrA2, using the technology chosen in the above item;

- Perform unit testing of WAKAMWebService;
- Perform the integration with WAKAMEGUI followed by the appropriate tests, and,
- Provide a web tool with wide access by the public.

The next sections will describe the activities mentioned above.

4.2.1 Platform Definition

One requirement in the creation of the tool was the availability on the web, so we have decided to develop it on top of a cloud computing platform, now available in the market. As seen in Section 2.3.4, we have evaluated some existing platforms, and after this analysis we chose to use Google App Engine (GAE). Using the platform offered by GAE, the programmer can abandon several worries already addressed by GAE, for example, a scalable architecture, allowing the load balancing and multiple machines processing (e.g. grid computing).

Another positive aspect is that GAE already provides a ready environment for the implementation of the application. This avoids additional worries as what is the operational system running, the application server, the server database, which compiler/interpreter for the language chosen. These advantages allow the developers to keep the focus only on the business rules application to be developed. Additionally, the free services offered by GAE (but with some limitations) were another factor of choice, when compared to other cloud computing platforms available.

When the platform analysis was carried out, the applications in GAE could only be done based on Python. However, along the development of this work, the GAE started also to support the Java programming language, thus opening more opportunities for reuse of existing tools and frameworks, since Java has a higher popularity than Python.

4.2.2 Model Repository with PyEMOF

Once chosen the platform, a research regarding MDE tools was performed, as detailed in Section 2.4.1, in particular, tools allowing the development of model repositories. As the single programming language supported by GAE was Python in the beginning of our application, we had chosen to reuse PyEMOF (2.4.1.2) as the tool for developing the model repository, because it was the only tool available for programming in Python.

Unfortunately, PyEMOF has some shortcomings, so that was necessary to adapt it for use in this work. This was possible because the PyEMOF is an open-source tool with source code available for modification. The changes made in PyEMOF to correct its shortcomings were:

- Adjustment of EMOF metamodel within the PyEMOF to follow the latest specification available, released by OMG. The first change was to add the EMOF elements not supported by PyEMOF: *Comment* and *UnlimitedNatural*. After that, we made the modification of the following elements: *Element* (to support the property *ownedComment*), *Type* (to support the property *package*), *Datatype* (for removal of properties not in the specification of EMOF). Finally, the mechanism to generate identifiers for instances, namely the *id*, has been restructured;
- Adjustment of Class *Factory*, responsible for define each element of the metamodel to support the new elements added. It was done a restructuring of the *Repository* class, which is responsible for the storage of instantiated elements of the metamodel, in order to modify the way as the element instances are stored and also to create methods to search for specific instances;
- Modification of the classes responsible to export/import the PyEMOF instance models to XMI format. These changes were made to reflect the changes made in the metamodel, besides being in accordance with the XMI standard used by EMF, since there are many available tools that support this standard;
- Restructuring the engine for automatic generation of code to create model repositories based on a model defined in EMOF. In addition to the adjustments made to include the changes to the metamodel, the engine was redesigned to allow the generation of codes for various models organized into packages, which was not supported by the tool, so far.

These changes in PyEMOF are available in a repository of source code powered by Google Code that allows any public access. This repository can be accessed via the following web address: <http://code.google.com/p/pymof>.

Once the improvements made to PyEMOF were realized, we have checked the compatibility on the platform offered by GAE. For this validation, was developed a simple web application that allowed the creation of metamodels in EMOF through a screen based

on a tree navigation mechanism and palettes of properties. This application has two components, as depicted in Figure 4.16:

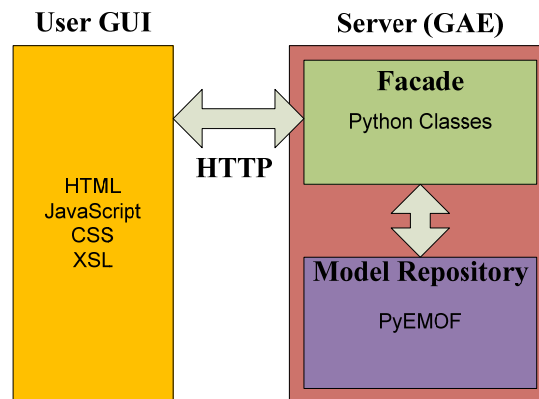


Figure 4.16 - metaWAKAME Architecture

- **User GUI** – is the component that interacts with the user. For the development were used the following web technologies: HTML, JavaScript, CSS, and XSL. It allows the user create/edit/import/export metamodels. On the main screen for edition of models (Figure 4.17) we can note on the left, the tree of elements presented. By selecting any of these elements, its properties will appear on the right side, allowing the user to just consult or even change these properties. The User GUI also contains links to the creation of new elements. Like any arbitrary web application, the User GUI communicates with the Server through HTTP protocol;
- **Server (GAE)** – is the component in which was implemented all the functionality of the application. The Server is deployed on the GAE platform. This component is subdivided into two minor components:
 - **Facade** – in this component was realized a facade of features that can access the model repository. It was implemented through Python classes. We can list as its most important purposes: (i) define what are the possible actions in the application, and (ii) implement the necessary manipulation of the data sent by the user, in order to use the applications of the model repository, and therefore, allowing the edition/creation of such models. Architecturally, this component is arranged between the user interface (GUI) and the Model Repository.
 - **Model Repository** – this component is the EMOF model repository, but generated by PyEMOF. All activities related to management of models, such as

creating, editing, export or import, and even the data persistence is under the responsibility of this component.

The development of an example of a real application led to a better implementation of the tests on the developed metamodel, because it allowed, in a way easier than programmatically, the definition of models and for import and export these to XMI notation, where a set of format validation was carried out. Another objective achieved with the implementation of this tool was to check the compatibility of using the repository created on PyEMOF, under the GAE platform.

As already mentioned, this application was developed and deployed in GAE and has a content of public access through the link <http://emof.appspot.com>.



Figure 4.17 – metaWAKAME Prototype

The development of an example of a real application led to a better implementation of the tests on the developed metamodel, because it allowed, in a way easier than programmatically, the definition of models and for import and export these to XMI notation, where a set of format validation was carried out. Another objective achieved with the implementation of this tool was to check the compatibility of using the repository created on PyEMOF, under the GAE platform.

As already mentioned, this application was developed and deployed in GAE and has a content of public access through the link <http://emof.appspot.com>.

4.2.3 Model Repository with EMF

Along the development of this work, yet during the definition of the model repository, the GAE was assigned with an additional language for development: the Java programming language. This opened the area of production to use other MDE tools which gives support to generate models repository, which were even more mature, since the PyEMOF is not yet widely used, and so, as usually happens in the large area of information systems, is subject to faults and bugs.

At this epoch, it was carried out a research concerning the tools to support repositories of existing models for Java, and among those found, stood out the *Eclipse Modeling Framework* (EMF) (2009) and *Metadata Repository* (MDR) (2009). However, we instantly, discard the MDR because it is based on the specification of MOF 1.3, which is an old specification. Another negative aspect relies on the project has not changed since 2004, which says that either the project was abandoned or, at least, there is no more interest in its continuity by the researches.

Consequently, we decided to use the EMF (consult Section 2.4.1.1 for further references). The advantage of using the EMF on PyEMOF is that the former began to be developed for some years and there are many people who use it, showing thus, stability. Another point is that EMF is reused by several modeling tools such as *IBM Rational Software Modeler* (RSM, 2009), *Papyrus* (Papyrus, 2009), *Epsilon* (Epsilon, 2009), *Kermeta* (Kermeta, 2009) among others. Finally, the EMF provides an environment for development of models integrated with Eclipse and has several libraries to assist the manipulation of these models, such as the *MDT-OCL* (OCLMDT, 2009) for evaluation of OCL expressions, and the *EMF Validation Framework* (EMFV, 2009) for validation of models.

4.2.4 The Kobra2 Metamodel in Ecore

With the redesign of the tool (EMF) to create the repository of models, the next step taken was to define the Kobra2 metamodel regarding the specific format of this tool. In this case, the Kobra2 metamodel should be defined in Ecore in terms of restrictions on the EMF.

For this work, due to the size of the Kobra2 metamodel, it was necessary to choose carefully which part of the Kobra2 metamodel would be implemented in WAKAME. Currently, the Kobra2 method has 16 types of Views:

- Specification Structural Class Service;
- Specification Structural Class Type;
- Specification Structural Instance Service;
- Specification Structural Instance Type;
- Specification Operational Service;
- Specification Operational Type;
- Specification Behavioral Protocol;
- Realization Structural Class Service;
- Realization Structural Class Type;
- Realization Structural Instance Service;
- Realization Structural Instance Type;
- Realization Operational Service;
- Realization Operational Type;
- Realization Behavioral Algorithm;
- Derived ComponentClassDependencies; e
- Derived OperationDependencies.

Amongst these views, we choose only the ones most important while modeling an arbitrary system. Hence, the eight views chosen are summarized below:

- *Specification Structural Class Service* – addressing the component specification;

- *Specification Structural Class Type* – holding the types used at the component specification;
- *Specification Operational Service* – to define the pre/post conditions as well as the body of the methods listed in the component specification;
- *Specification Operational Type* – the same of the last view, but regarding the methods of the types associated to the component specification;
- *Realization Structural Class Service* – addressing the component realization;
- *Realization Structural Class Type* – holding the types used at the component realization;
- *Realization Operational Service* – to define the pre/post conditions as well as the body of the methods listed in the component realization, and;
- *Realization Operational Type* – the same of the last view, but regarding the methods of the types associated to the component realization.

With this defined scope for the implementation of the model repository, the tool WAKAME will not be able to model neither the views of instances of a particular situation nor complex models in which activities and protocol state machines diagram are necessary. However, the WAKAME is able to model various systems, including itself, since its PIM does not use any of the views that were not covered in this work.

After the election of views that would be implemented, the next step was to disregard the unnecessary packages of the Kobra2 metamodel, since some packages define the elements of Views out of the scope. The following Kobra2 metamodel packages, are not supported::

- *Kobra2::SUM::Behavior* and the sub-packages;
- *Kobra2::Views::Specification::Structural::Instance* and the sub-packages;
- *Kobra2::Views::Specification::Behavioral::Protocol*;
- *Kobra2::Views::Realization::Structural::Instance* and the sub-packages;
- *Kobra2::Views::Realization::Behavioral::Algorithm*;
- *Kobra2::Views::Derived* and the sub-packages;
- *Kobra2::Views::Derived::OperationDependencies*;

- *KobrA2::Transformation::Specification::Structural::Instance;*
- *KobrA2::Transformation::Specification::Behavioral::Protocol;*
- *KobrA2::Transformation::Realization::Structural::Instance;*
- *KobrA2::Transformation::Realization::Behavioral::Algorithm;* and
- *KobrA2::Transformation::Derived* and the sub-packages.

To create the KobrA2 metamodel to generate the model repository was used the RSM tool, because there was a full KobrA2 metamodel (without the parties dependent on the UML metamodel and OCL) modeled in UML in this tool.

With the withdrawal of the above mentioned packages, it was also essential to remove the remaining dependencies on other packages. As the KobrA2 makes use of subparts of the UML 2 metamodel, these portions were added into the metamodel under development. For this, we used the metamodel of UML 2 in the old format of Rational Rose (which is compatible with RSM) and is available on the website of the OMG (OMG, 2009a). And finally, it was necessary to model the metamodel for OCL 2 to be added into the KobrA2 metamodel. Figure 4.18 shows KobrA2 and UML 2 metamodel dependencies and, Figure 4.19 depicts the dependencies between KobrA2 and OCL 2.

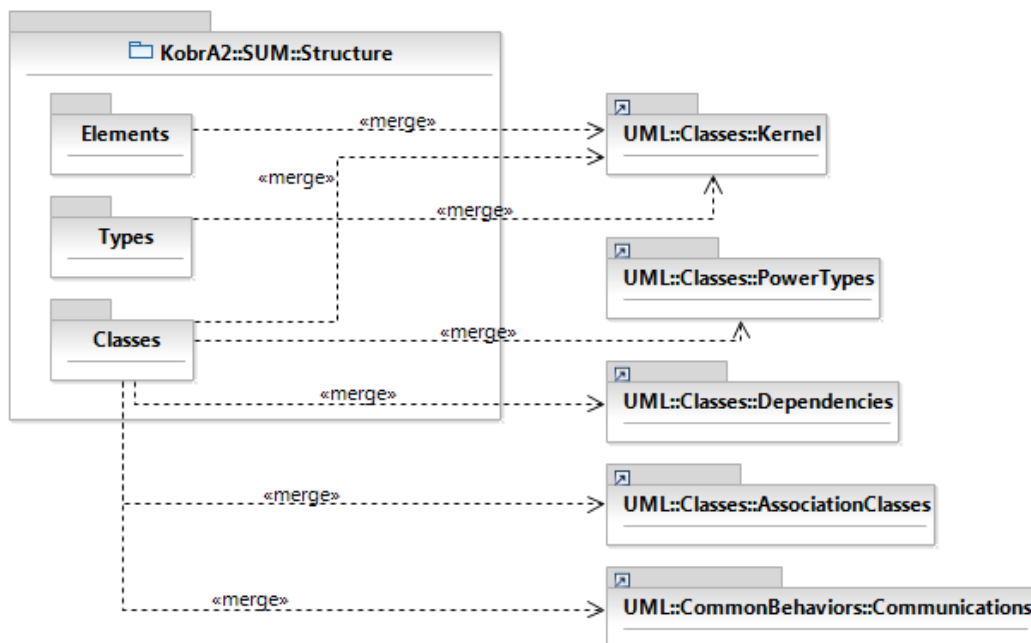


Figure 4.18 - Dependencies between KobrA2 and UML

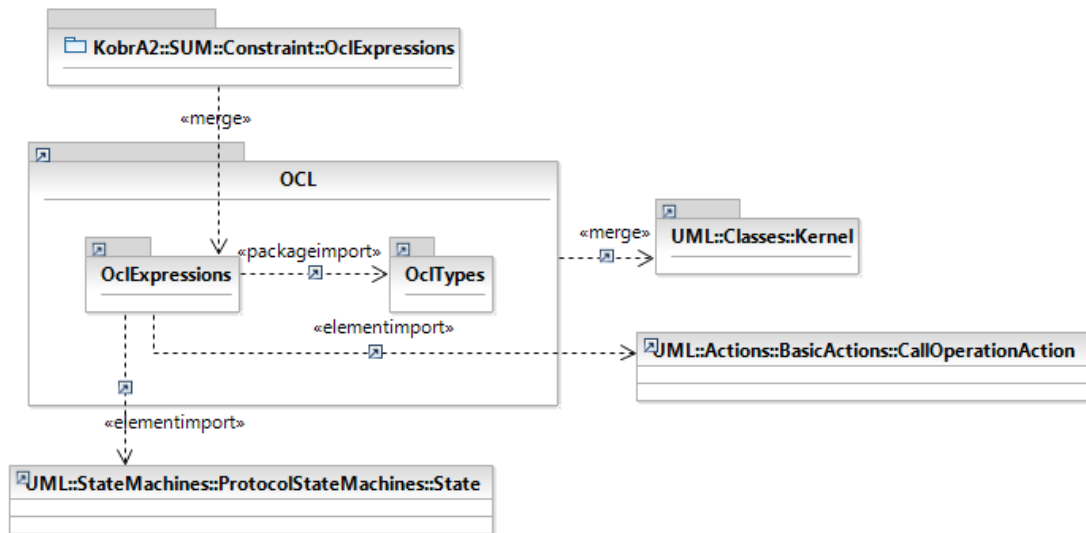


Figure 4.19 - Dependencies between KobraA2 and OCL

Along this phase, we noticed some inconsistencies in the original KobraA2 metamodel, defined by its authors that had to be corrected to enable the creation of the repository of models. They were:

9. The dependencies between the sub-packages of *KobraA2::SUM::Structure* of KobraA2 with the UML packages, where only the package *Structure* merges the content of the *Kernel* package, as outlined in Figure 4.20. However, it was essential to extend the merge to all the sub-packages, instead of the *Structure* itself. The result of this change is shown in Figure 4.18.

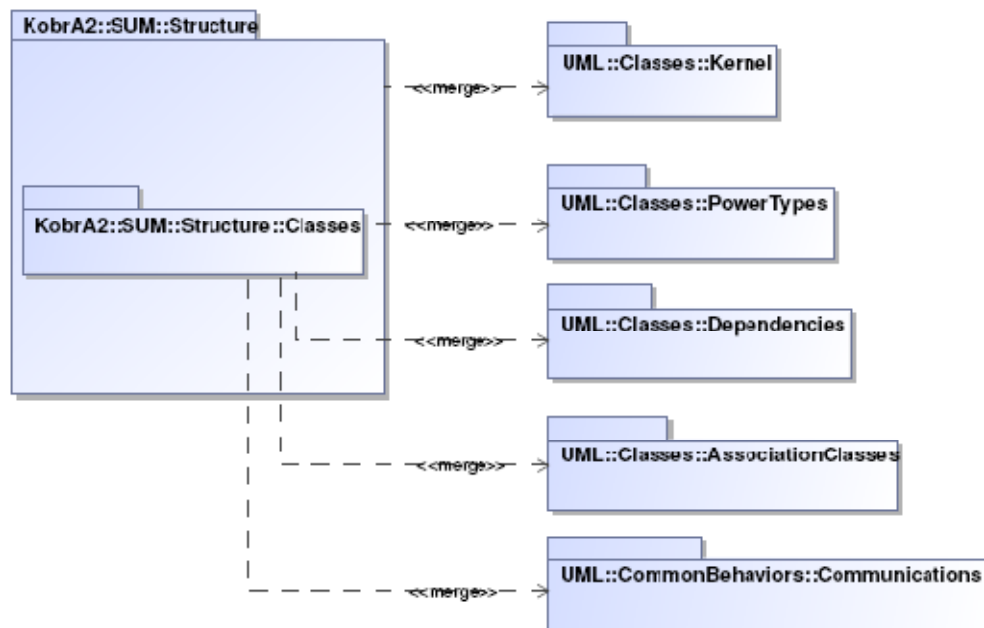


Figure 4.20 - Dependence between SUM and UML.

10. The need to create an OCL restriction in the *ComponentClass* shown in Figure 4.21, to express the behavior of the association *Nests*. When an association *Nests* is realized between two *ComponentClass*, it means the target *ComponentClass* is included "inside" the source, by the *packagedElement* composition. This OCL restriction can be visualized Figure 4.22.

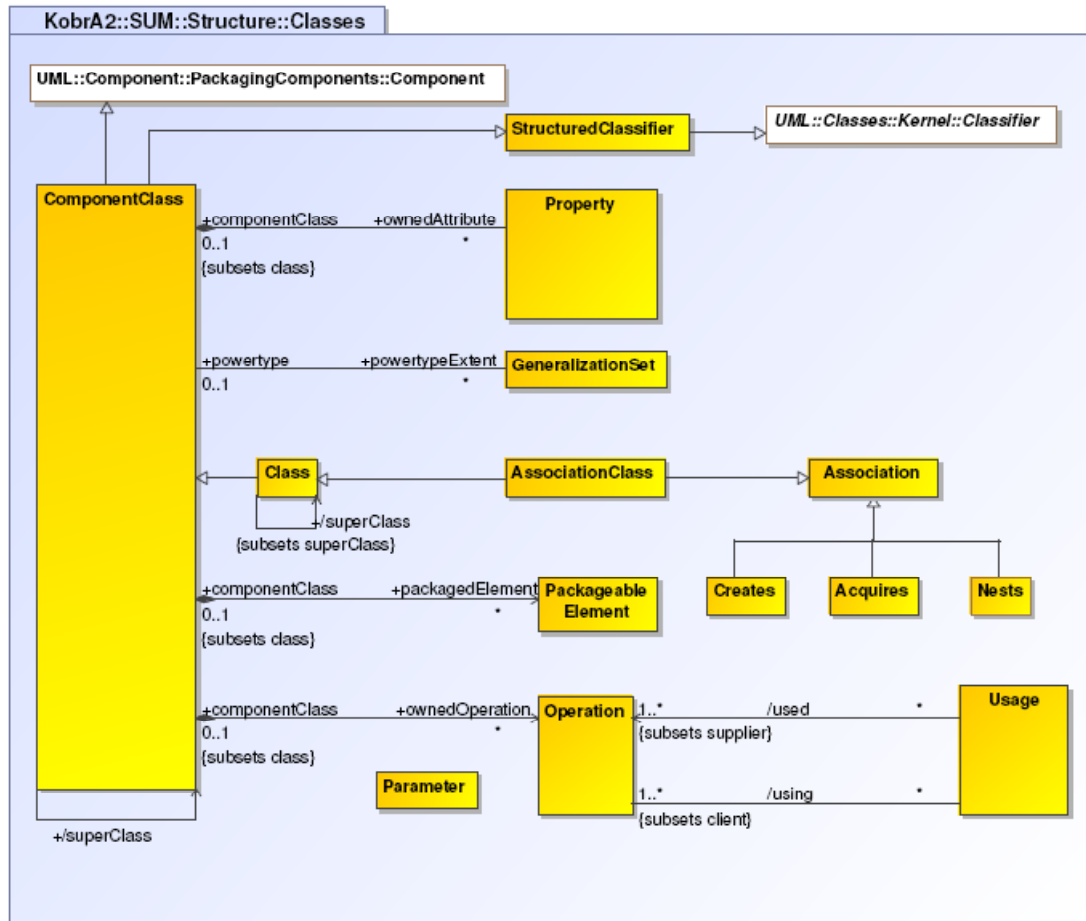


Figure 4.21 - Classes Package from Kobra2 metamodel

```
context ComponentClass:
inv: let nest:Set(Nests) = ownedAttribute.association->select(a | a.ocIsKindOf(Nests)),
      nestComp: Set(ComponentClass) =
        nest.navigableOwnedEnd->select(c | c.ocIsKindOf(ComponentClass))
in nestedComp->forall(c | self.packagedElement->includes(c))
```

Figure 4.22 – OCL Constraint on *ComponentClass*

11. The merge relationship between the packages *Kobra2::Views::Realization* and *Kobra2::Views::Specification*, shown in Figure 4.23, which semantically wants to represent a realization Kobra2 model, merging its current specification. However, this has been incorrectly defined, as this merge is being held at the metamodel, and it

does not address the desired semantics. This merge should exist at the model level, which was programmatically done in our repository. Hence, the merge at metamodel level has been removed.

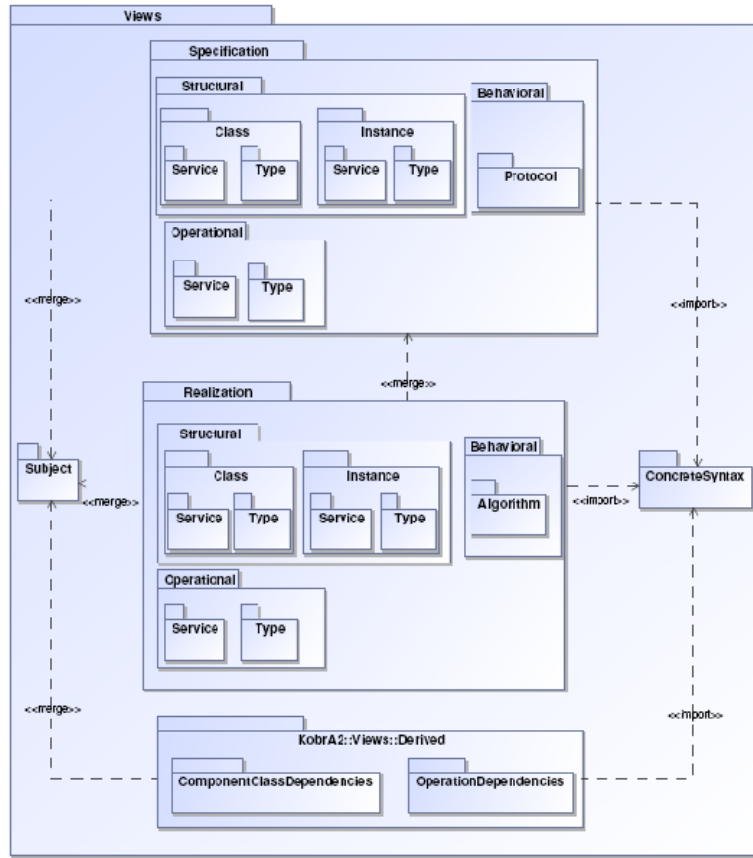


Figure 4.23 - Kobra2 View Package Nesting.

12. The dependencies between the sub-packages of *Kobra2::Views* with the sub-packages of *Kobra2::SUM*, (Figure 4.24), since the package *Kobra2::Views::Specification::Structural::Class::Type* merges erroneously the package *Kobra2::SUM::Constraint::Structural* as would be the correct the package *Kobra2::Views::Realization::Structural::Class::Type* realize such association. It was also necessary that the package *Kobra2::Views::Specification::Structural::Class::Service* would merge the SUM package above mentioned. These corrections were made in our metamodel, in Figure 4.25.

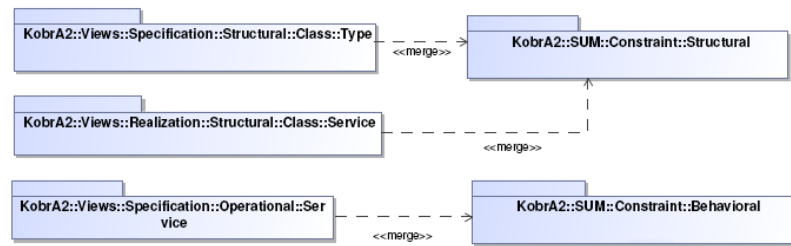


Figure 4.24 - Package Dependencies between Views and SUM.



Figure 4.25 - Package dependencies between Views and SUM

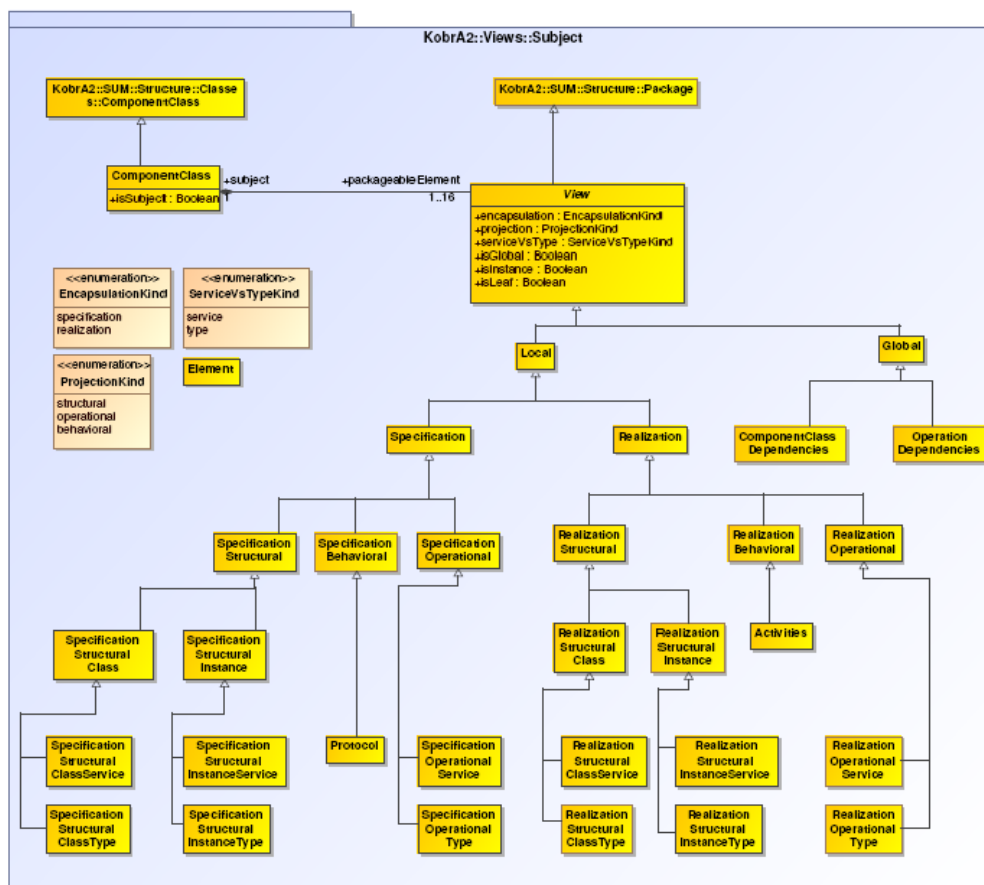


Figure 4.26 - "Subject" component and its View-SUM relationship.

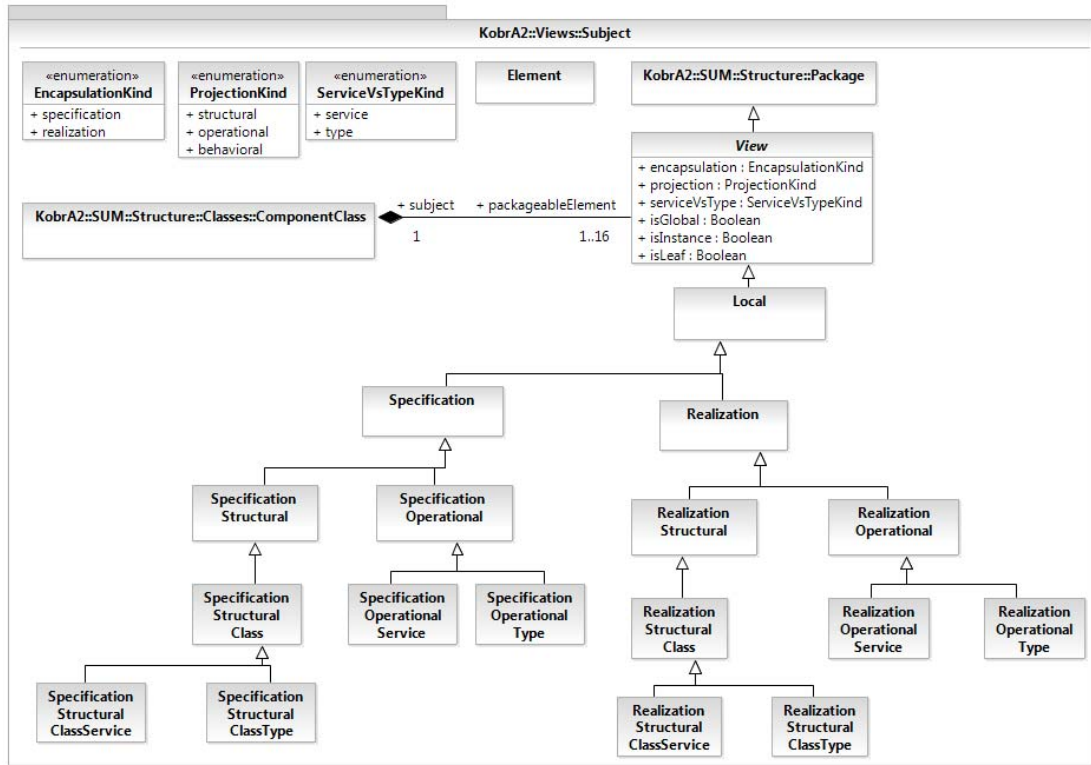


Figure 4.27 – Relationship between View and Subject ComponentClass

13. The *ComponentClass* class, which inherits from *KobrA2::SUM::Structure::Classes::ComponentClass*, (Figure 4.26), need not exist, so this class was removed and the relationship between it and the element *View* has to be made by the super-class, as shown in Figure 4.27.
14. The association between the elements of the SUM with the elements *View* and *Abstraction*, shown in Figure 4.28, was wrongly defined. This association wants to say that each element of the SUM is associated with one or more *Abstraction*, and this, in turn, is associated with a single element of *View*. Thus, the *SumElement* elements, which inherits from *Element* of SUM, and, *ViewElement*, which specializes *Element* of the *View*, it would not allow entity of the SUM, which inherits from *Element*, to be related to *Abstraction* by *SumElement* because in this case, they were "siblings" and not "father-son". The same thing applies to the element *ViewElement*. To correct this problem was essential to associate *Abstract* directly with the *Element* of SUM and the *Element* of *View*, as depicted in Figure 4.29.
15. The need of an OCL constraint in the OCL *View*, Figure 4.29, to ensure that the views elements associated with an *Abstraction* are the same view elements belonging to *View* entity associated to the *Abstraction*. Figure 4.30 shows the OCL constraint.

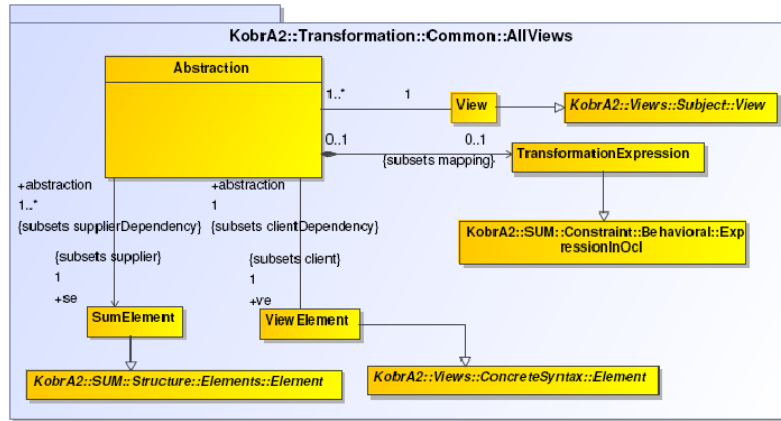


Figure 4.28 - Transformation Abstractions.

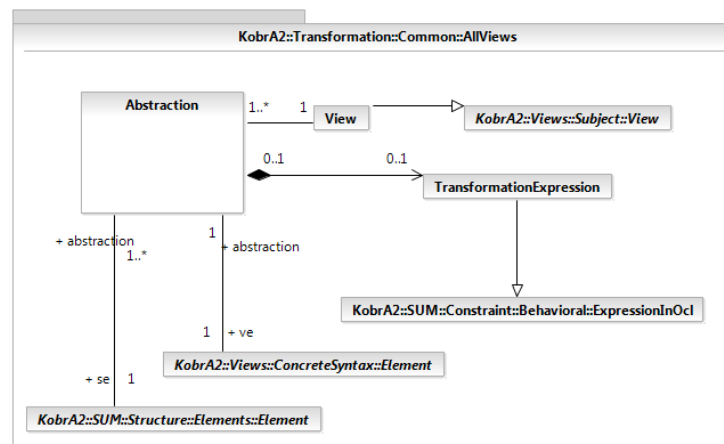


Figure 4.29 - Transformation Abstraction

```
context View:
inv: self.abstraction->forAll(a | self.ownedElement->includes(a.ve))
```

Figure 4.30 – OCL Constraint on View

After making the necessary changes in the KobrA2 metamodel, together with the inclusion of the dependencies of UML and OCL, we have reached the first, stable UML Model in RSM. Then, the model was exported to an Ecore notation, one service also supported by RSM. Thus the model repository could be created through the EMF. However, the conversion cannot preserve the merge associations between the packages, since these are not supported by Ecore.

The way to avoid this problem was the implementation of a routine for carrying out the merge. This routine takes the source and the target packages. Furthermore, it was necessary to perform the routine for each merge in the UML model regarding the correct order of dependencies; however we had to make the modification of the references in the

packages manually. When this step was completed, we included the UML and OCL portions used in the Kobra2 packages, besides removing that which were no longer needed.

Nowadays, the Ecore Kobra2 metamodel is constituted by forty-nine *EPackages* and two thousand one hundred and sixty-five *EClass* defined. Figure 4.31 displays the package structure in the final metamodel.

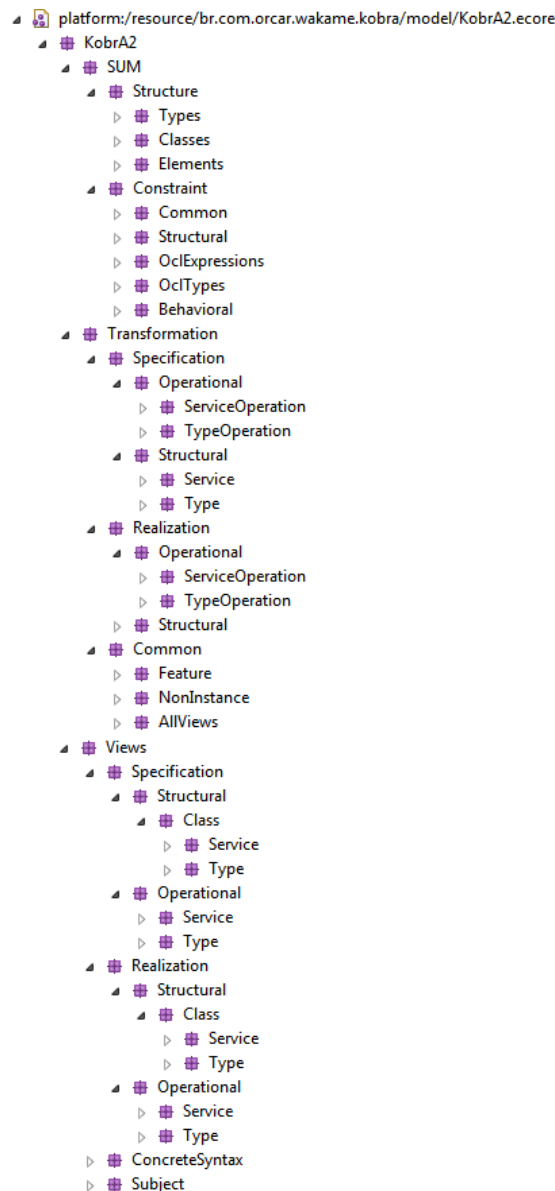


Figure 4.31 - Kobra2 Metamodel Packages in Ecore

4.2.5 Model Repository Generation

With the completion of the Ecore Kobra2 metamodel, the process for generating the code of the model repository (Figure 4.32) was finally carried out. Initially, the Ecore leads to

Genmodel file, in which the information need to generate the code are inserted. Next, from the Genmodel, it is possible to generate the three plug-ins by EMF: EMF.model, EMF.edit and EMF.editor, as explained in Section 2.4.1.1. However, concerning this work, we adjust the process for a single plug-in creation, the EMF.model, because the model repository is stored within it.

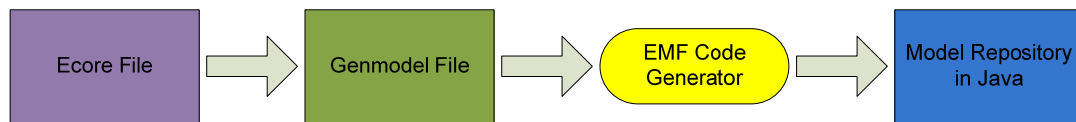


Figure 4.32 - Process to Generate the Model Repository from Ecore File

At the end of this process, the model repository for the Kobra2 metamodel in the Java programming language was finally completed. For each *EClass* in the metamodel is created an interface and a Java class that implements it. Table 4.1 summarizes some metrics about the model repository.

Table 4.1 - Model Repository Metrics

Statistic	Value
Packages	90
Classes	3.493
Total of Lines of Code	467.465
Size of Implementation	38,3 MB
Size of JAR	6,27 MB

4.2.6 Server Implementation

As stated previously, the platform chosen for developing the tool was Google App Engine (GAE). The GAE provides environments for implementation in Python and Java languages, but as we have used EMF for generating the model repository, the language chosen for the server component development was Java.

For the development of the server, it was conjectured the following development environment:

- *Google App Engine SDK for Java 1.2.2* (GAE, 2009a) – local development environment for GAE;
 - *Eclipse 3.4* (Eclipse, 2009) *With Google plug-in* (GoogleP, 2009) – Eclipse was the chosen IDE for Java development, because it possess the plug-in Google that supports the implementation and deployment of GAE applications;
 - *Subclipse* (Subclipse, 2009) – SVN plug-in for Eclipse to version control of code;
- and,

- *EMF 2.4* (EMF, 2009) – plug-in for Eclipse for the repository of models. This one addresses the dependencies for the creation and serialization of models via the repository. However, not all plug-ins that come with the EMF have been taken to use. Those selected to conduct our application (and without which, such application would not be possible to be performed) were: *org.eclipse.emf.ecore*, *org.eclipse.emf.common* and *org.eclipse.emf.ecore.xmi*.

The server implementation was done according to the architecture defined in PIM, detailed in Section 4.1. However, for the encoding based on PIM, some platform-specific projects decisions were necessary:

- Structure of packages for code organization and definition of responsibilities;
- Communication with the client; and,
- Technologies, framework and/or tools to perform the transformations defined in Kobra2.

According to the PIM set in the server, the *WAKAMWebService* is divided into three types of components: the *ServiceController*, responsible for the management of request and the redirection to the specific entity; *MVCAction* specializations, which are responsible to realize the requests made by the client, that is, it is the place where all the business rules are implemented, and, finally, the *WAKAMModel* responsible for data persistence. In accordance with these components, the structure of package is depicted out in Figure 4.33.

O pacote *br.com.orcas.wakame.webservice.service* contém as classes que implementam o *ServiceController*. Como este componente tem a responsabilidade de redirecionar as requisições para as ações específicas, ele foi implementado utilizando Java Servlets[19] para poder manipular as requisições dos clientes.

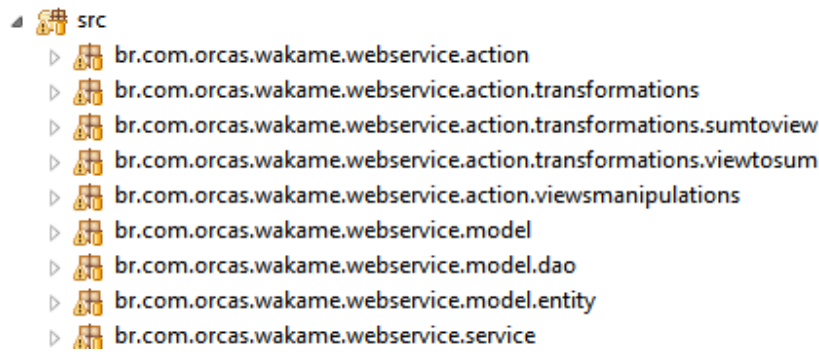


Figure 4.33 - Package Structure of WAKAMEWebService Implementation.

The package *br.com.orcas.wakame.webservice.service* contains the classes that implement the *ServiceController*. As this component has the responsibility to redirect requests for specific actions, it was implemented using *Java Servlets* (SERVLET, 2009) to be able to handle the requests of clients.

Following, the package *br.com.orcas.wakame.webservice.model* and its sub-packages implement the *WAKAMEModel* component, responsible for data persistence. The GAE has a mechanism of data persistence transparent to the programmer, since it is only necessary to define which entities are persistent and the properties for the persistence of the attributes of each entity. In our case, the *PIM* class defined in the *WAKAME* modeling is persistent. Another responsibility of this package is, through the repository of models, to save and/or load the models created, and additionally, to import/export the model repository to XMI format.

The package *br.com.orcas.wakame.webservice.action* and its sub-packages implements the actions defined in the modeling of *WAKAMEWebService*. Within this package, there is a minor one, *transformations*, which is the implementation of the component *Transformations*, which lies within the existing component *ViewManipulationAction*. Continuing, within the action package there is yet the *viewsmanipulations* package, with the necessary mechanism for serialization of model elements in a pattern that will be used for communication with the client. We chose to implement a proper serialization, because this avoids sending data that are not relevant to the model, with information about the class of the Java object. Thus, the amount of data that will travel in this communication decreases. Another advantage of not using the existing serialization in Java, is due the client is not tied to one specific technology. Hence, the client may be implemented in any technology.

To establish the communication between the server with the GUI, we carried out a survey of existing technologies for communication over the HTTP protocol. Currently, the most used is XML (XML, 2009), however, it defines a verbose markup language, generating an overhead due the huge amount of data to be transferred between the GUI and the server. Another researched pattern was JSON (JSON, 2009), which is similar to XML when talking about the transfer of plain texts over HTTP. However, JSON has the advantage of defining a more concise format compared to XML, thus generating a lower overhead in the amount of data transferred during communication. For this reason, JSON was used as standard for communication between GUI and server.

The biggest challenge in implementing the *WAKAMEWebService* were the transformations defined in the *KobrA2* metamodel. Being transformations of model, we searched for specific tools and technologies that could support this activity. The tools that stood out were *ATL* and *Epsilon*, as detailed in the section 2.4.1.3. Although these tools gave support for most of the requirements, some deficiencies, exclusively for our application, could be verified. The first negative aspect was the difficulty to debug the code developed in these two tools, because the IDE available in both, sinned by lack of readability. The second, and most critical, is the transformations of both being done in batch, through reading/writing files, which is not allowed in GAE. Finally, it was also taken into account the high level of experience and Java skills programming by the author, when compared with the tools mentioned above. Thus, the Java language was chosen for the implementation of the transformation, because it has a great IDE for development and debug.

Therefore, with the package *br.com.orcas.wakame.webservice.action.transformations* were implemented the transformations defined in the *KobrA2* metamodel. The code was organized into sub-packages with the transformations of View to SUM in *viewtosum*, and, in opposite direction, the transformations from SUM to View in *sumtoview*.

The transcript of the transformations was made following the OCL constraints defined in the package *KobrA2::Transformation*. However, these restrictions are bi-directional, that is, they are defined once and refer to both the changes to View to SUM and SUM to View. Thus, for each OCL constraint were defined two transformation rules in Java, one for each type of transformation.

These OCL expressions consist of several invariants defined on each element type of the metamodel. As an example, Figure 4.34 shows the transformation rules between the *ComponentClass* of *Specification Structural Class Service View* with the *ComponentClass* of SUM.

```
context ComponentClassAbstraction
inv: ve.superClass = se.superClass
inv: ve.ownedAttribute = se.ownedAttribute->select(visibility=#public)
inv: ve.ownedOperation = se.ownedOperation->select(visibility=#public)
inv: ve.inv = se.inv
inv: ve.hasStereotypes->includes('componentClass')
```

Figure 4.34 – Example of Transformation OCL Expression

The expression in Figure 4.34 shows that the *ComponentClass* of *Specification Structural Class Service View* possess all super classes, invariants, public attributes and methods of the *ComponentClass* of SUM. And finally, the *ComponentClass* of the view is stereotyped with *componentClass*. The rules for the properties of *ComponentClass* are defined in other context, of the elements specialized by *ComponentClass*, which are *Classifier/NamedElement/Element*.

By codifying the expressions for the transformation from View to SUM, for each constrained element, there are two methods: one to check whether it already exists in SUM (*update*) and update the basic attributes, that does not reference other elements, and; another method (*populate*) to fill the references to other elements. For each existing element in the View, to be processed, will run the *update* method for the specific element, for the creation of elements not yet existing in the SUM. After that, will run the *populate* method to update the references. The update method will ensure that when you run the *populate* action, all references have already been created.

In the same sense, for the transformation from SUM to View, it was created only one method for each constrained element of the metamodel. This method will update, create or delete the elements of View, according to the modifications that occurred in SUM.

Because Java is not a proper language for transformations of models, for certain transformation rules were needed many lines of codes, especially when it came to properties that are collections of elements, such as the, *ownedAttribute* and *ownedOperation* of *ComponentClass*.

As an example of comparison regarding the code size, we can cite the methods implemented for transformations View to SUM and SUM to View of the *ComponentClass*. These methods had two hundred and twenty lines of code to perform the six lines of the OCL restriction shown in Figure 4.34. The Table 4.2 displays the metrics of the codification of the transformations rules of the Kobra2 metamodel portion taken as study in this work.

Table 4.2 - Transformations Metrics

Statistic	Value
Packages	3
Classes	9
Total of Lines of Code	7.151
Size of Implementation	304 KB

A Table 4.3 shows the metrics related to general implementation of WAKAMWebService, including the transformation metrics, but not taking into account the metrics of the model repository. All the WAKAMWebService code and transformations were implemented manually, while the code of the model repository, presented in Section 4.2.5 was automatically generated by EMF, requiring few manual corrections.

Table 4.3 – WAKAMWebService Implementation Metrics

Statistic	Value
Packages	11
Classes	57
Total of Lines of Code	23.528
Size of Implementation	931 KB

4.2.7 WAKAMWebService Integration with WAKAMEGUI, Tests and Deployment

Throughout the implementation phase were performed unit tests of classes and methods, as they were developed. In these unit tests, several bugs were found and fixed by the author, thus ensuring a minimum quality of the code.

For the integration of WAKAMWebService with the implementation of WAKAMEGUI, an environment has been created on Google App Engine to store both implementations, Server and GUI. This environment created to perform integration testing is similar to the production environment in which the tool WAKAME is published. This is excellent for testing, because in this way, we avoid (unpleasant) "surprises" when it is published in a production environment, which is likely to occur when dealing with

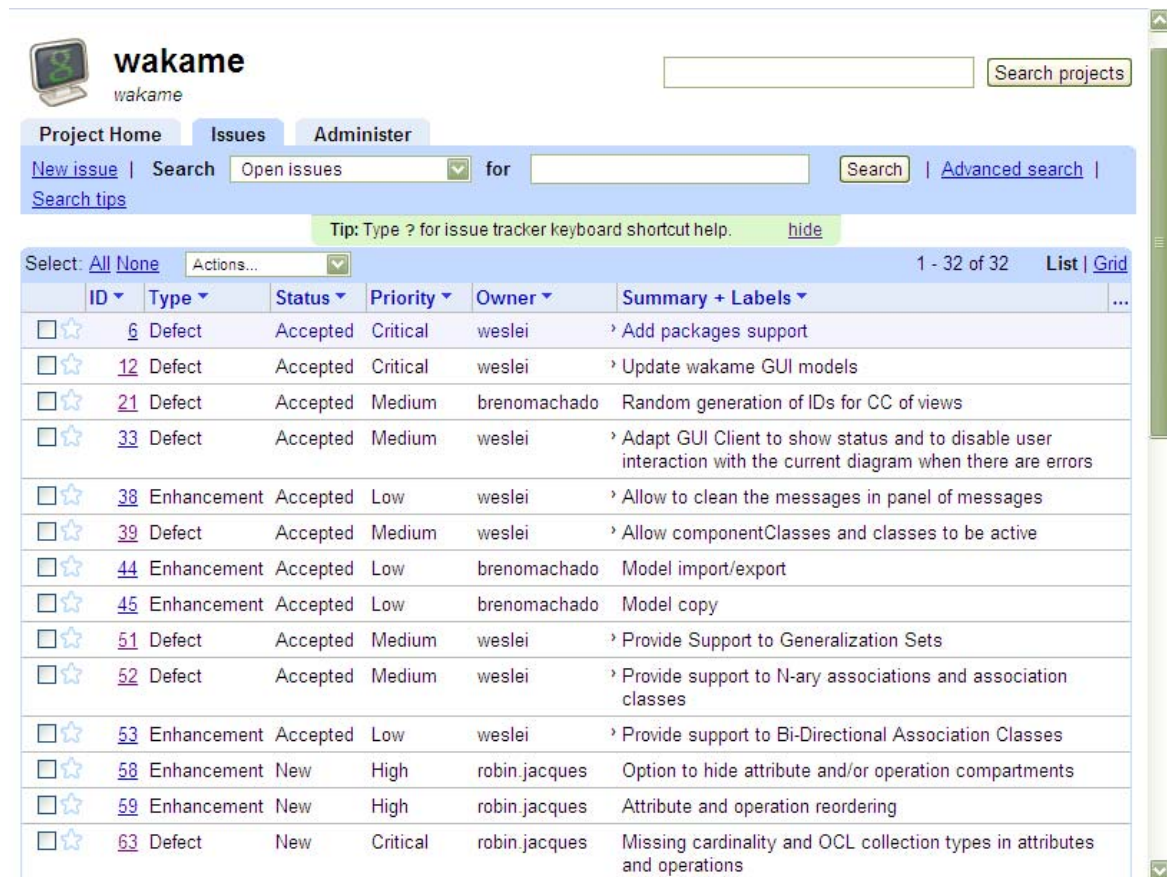
differences in environments. The test environment can be accessed through the address <http://wakameemf.appspot.com>.

Both code, WAKAMWebService and WAKAMEGUI, were placed on version control through SVN, so that both authors could have access to the modifications occurred in the components during the integration phase.

This integration phase consisted primarily of providing the services of WAKAMWebService through URLs, so that the WAKAMEGUI could access. The communication was set to both through HTTP requests in JSON format, as stated in Section 4.2.6. Therefore, at this stage, we tested the formats of communication defined in the implementation phase, and the bugs have been fixed as they were identified.

After the tests of integration between WAKAMWebService and WAKAMEGUI, some general WAKAME tests were taken. For these, we have used the same environment of the integration tests. It was created also a project called WAKAME on Google Project Hosting to manager the coordination of the tests. This project provides an *issue tracker*, so that each issue found in testing, could be registered to be discussed, besides taking the necessary corrections. The issue tracker was used to manage the bugs found, but also to allow requests for improvements in the tool. This tracker can be accessed through the address <http://code.google.com/p/wakame/issues/list>. Additionally, the main types of problems were categorized, if, for example, they are related to Component UI, Logic or Persistence. As the tool WAKAME was released in a web environment for testing, anyone could access it, and if any problem or suggestion is found, it could register it in the issue tracker. Figure 4.35 shows the screen of the issue tracker that displays the list of existing issues.

The tests were performed and reported by the authors and several collaborators from the research group, in which the tool was performed. Until this date, were opened one hundred and twenty four issues in the project and sixty-seven have been corrected, thirteen issues were invalid, and seven was duplicate, remaining forty-one issues open. Nevertheless, thirty issues are related to improvements.



ID	Type	Status	Priority	Owner	Summary + Labels
6	Defect	Accepted	Critical	weslei	› Add packages support
12	Defect	Accepted	Critical	weslei	› Update wakame GUI models
21	Defect	Accepted	Medium	brenomachado	Random generation of IDs for CC of views
33	Defect	Accepted	Medium	weslei	› Adapt GUI Client to show status and to disable user interaction with the current diagram when there are errors
38	Enhancement	Accepted	Low	weslei	› Allow to clean the messages in panel of messages
39	Defect	Accepted	Medium	weslei	› Allow componentClasses and classes to be active
44	Enhancement	Accepted	Low	brenomachado	Model import/export
45	Enhancement	Accepted	Low	brenomachado	Model copy
51	Defect	Accepted	Medium	weslei	› Provide Support to Generalization Sets
52	Defect	Accepted	Medium	weslei	› Provide support to N-ary associations and association classes
53	Enhancement	Accepted	Low	weslei	› Provide support to Bi-Directional Association Classes
58	Enhancement	New	High	robin.jacques	Option to hide attribute and/or operation compartments
59	Enhancement	New	High	robin.jacques	Attribute and operation reordering
63	Defect	New	Critical	robin.jacques	Missing cardinality and OCL collection types in attributes and operations

Figure 4.35 - WAKAME Issue Tracker

To perform the case study, the tool was published in a production environment, available at <http://wakametool.appspot.com>. However, the test environment was not removed because the improvements and bug fixes are published first on it and only after validation; these changes are published in a production environment.

4.3. WAKAME Overall

Figure 4.36 displays the main page of WAKAME, in which there is a brief description of the project and participants, and from there, one can navigate to the model selection page, create a new model, importing an existing model or go to issue tracker page.

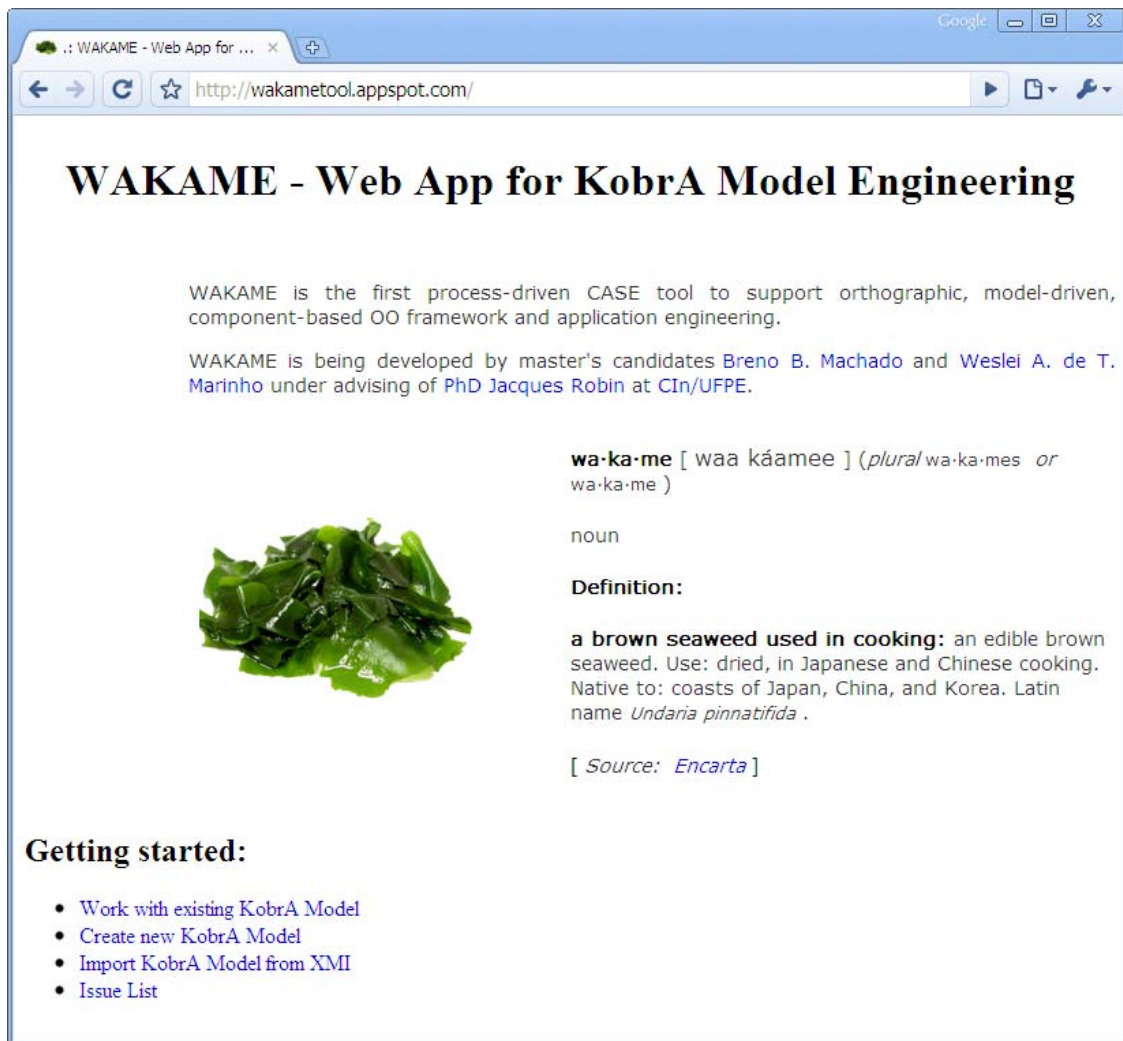
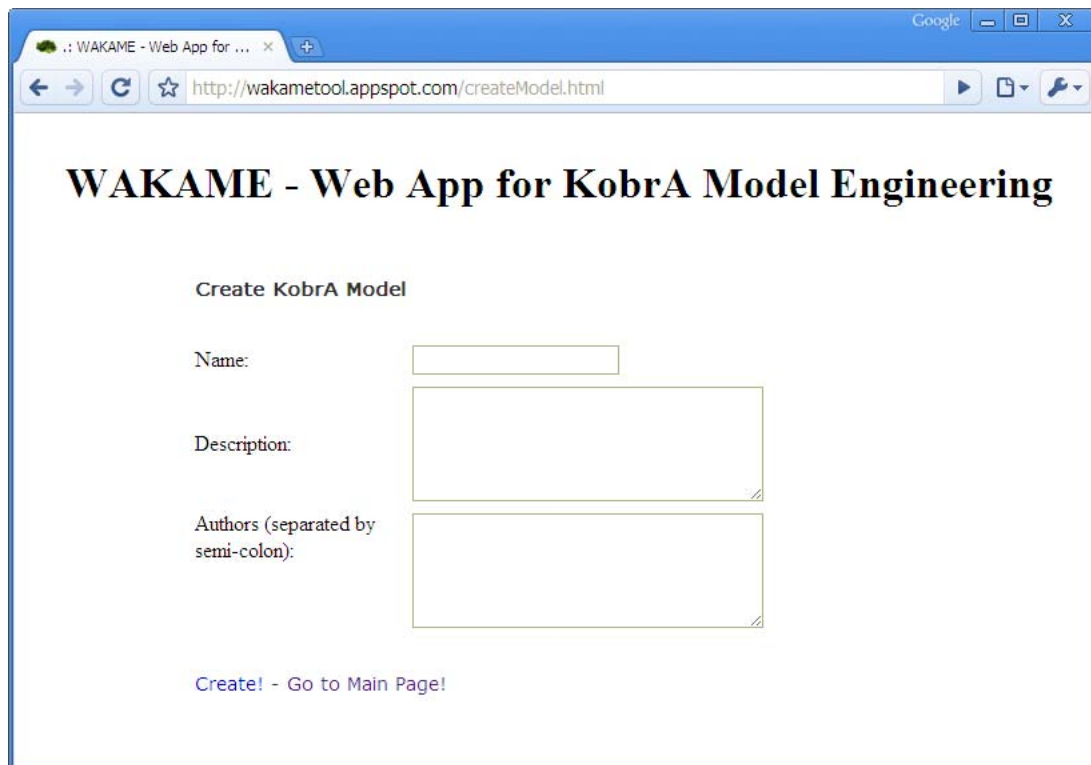


Figure 4.36 - WAKAME Main Page

Figure 4.37 displays the page to create a new model in the tool. Firstly, the user sets the model name, a description and the authors. The page to import a model from a XMI file, Figure 4.38, is similar to the registration page, with only one extra field, which is the one to upload the XMI file. This feature allows only importing KobraA2 models in XMI format; WAKAME does not support importing other types of models, such as UML, MOF or Ecore.



The screenshot shows a web browser window with the title "WAKAME - Web App for Kobra Model Engineering". The address bar displays "http://wakametool.appspot.com/createModel.html". The main heading is "WAKAME - Web App for Kobra Model Engineering". Below it, the section is titled "Create Kobra Model". There are three input fields: "Name:" (a single-line text box), "Description:" (a multi-line text box), and "Authors (separated by semi-colon):" (a multi-line text box). At the bottom, there is a link that says "Create! - Go to Main Page!".

WAKAME - Web App for Kobra Model Engineering

Create Kobra Model

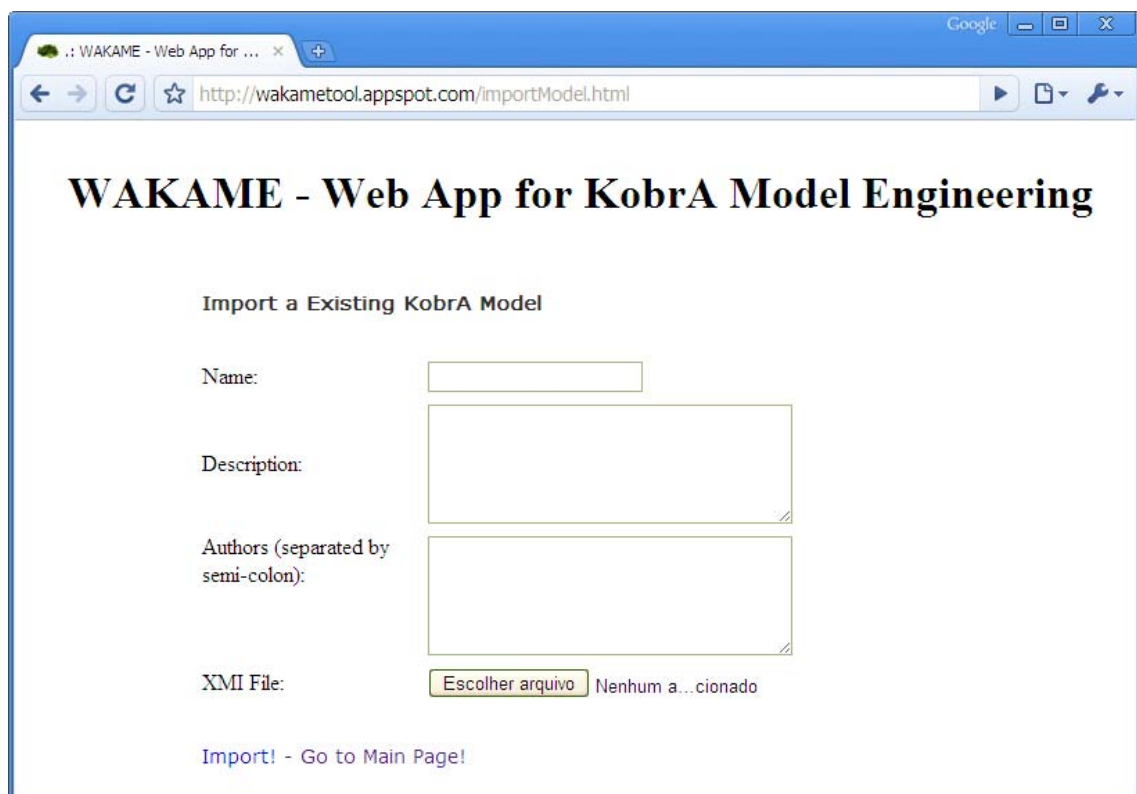
Name:

Description:

Authors (separated by semi-colon):

[Create! - Go to Main Page!](#)

Figure 4.37 - WAKAME Create Model Page



The screenshot shows a web browser window with the title "WAKAME - Web App for Kobra Model Engineering". The address bar displays "http://wakametool.appspot.com/importModel.html". The main heading is "WAKAME - Web App for Kobra Model Engineering". Below it, the section is titled "Import a Existing Kobra Model". There are four input fields: "Name:" (a single-line text box), "Description:" (a multi-line text box), "Authors (separated by semi-colon):" (a multi-line text box), and "XMI File:" (a file selection button labeled "Escolher arquivo" and a text box showing "Nenhum a...cionado"). At the bottom, there is a link that says "Import! - Go to Main Page!".

WAKAME - Web App for Kobra Model Engineering

Import a Existing Kobra Model

Name:

Description:

Authors (separated by semi-colon):

XMI File: Nenhum a...cionado

[Import! - Go to Main Page!](#)

Figure 4.38 - WAKAME Import Model Page

Figure 4.39 displays the page for selection of models. The list of model on the server is available through a combobox, and by selecting a model, the user can navigate to the modeling page, edit/view the details of the model, to export to XMI format or delete this model. The screen to edit/view the details of the model, Figure 4.40, displays and lets you modify the data that were registered, and also displays, but read-only, the date of creation of the model and date of last change. By requesting to export the model to the XMI format, will be downloaded the XMI file to the computer user.



Figure 4.39 - WAKAME Select Model Page

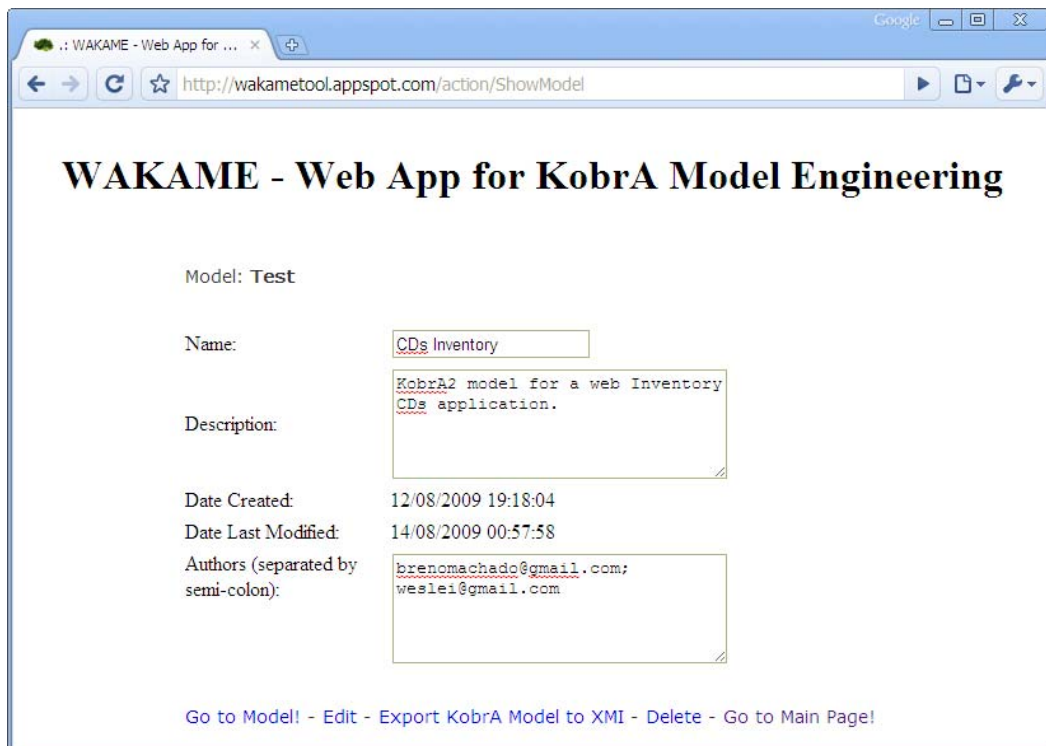


Figure 4.40 - WAKAME Edit Model Page

Once you select the link "Go to Model", for a specific model, the user will be redirected to the modeling page, Figure 4.41. This is the main project WAKAME page, which allows modeling any system. On the left side, are displayed the combo-boxes for selecting the view, according to the possible different perspectives:

- **Encapsulation** – *Specification* to select the views of the component specification, or, *Realization*, which lets you select the realization views of the component;
- **Projection** – *Structural*, for the views of structural modeling of the component, or, *Operational*, for the views of modeling, in OCL, of the operations declared in structural view; and,
- **View** – allows the options: *Service*, for the views of services that the component will provide, or *Type*, the views of types used in the component.

Below the combo-boxes for selecting the appropriate view, there are the buttons "*fromCloud*" to refresh the selected view with the new view on the server, and "*toCloud*" to persist the view on the server.

On the right side, we have the "*component navigation tree*", which allows navigation of the views for the component. Just below there is the "*element selection tree*", which lets you drag to the current view, an existing element to the current view. The bottom of the tool has a panel for displaying the messages, which can be notifications, error or debug messages.

The center of the tool is the free area for modeling and has at its top, the tool bar with the possible elements for modeling the view in question. The content of this tool bar will vary for each type of view, since each one allows a set of different elements.

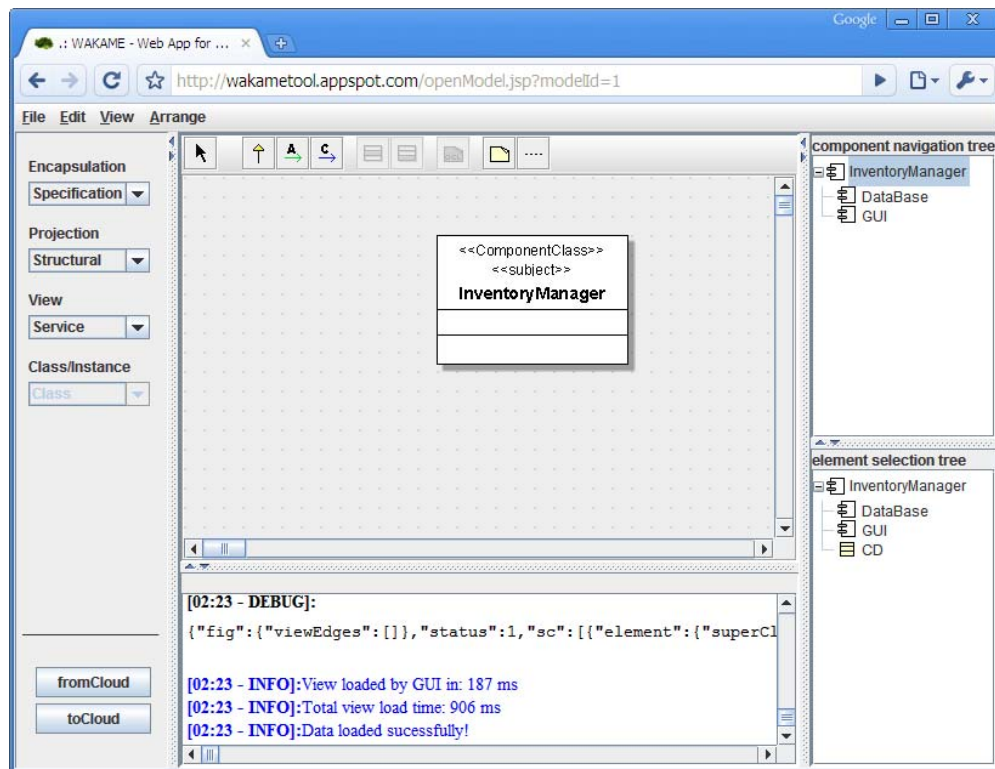


Figure 4.41 - WAKAME Modeling Page

CHAPTER 5

ASSESSMENT EXPERIMENTS WITH EARLY ADOPTERS

This chapter exhibits the evaluation of the WAKAME tool against the IBM Rational Software Modeler tool, the obtained results and our findings.

5.1. Experiment Definition

A productive way to assess whether a case tool fits its purpose is to determine the business objectives that the tool should met and then compare the tool against these objectives. After the analysis of several CASE tools, we states that the common features that a CASE tool should provide are mainly classified in terms of:

- Availability – This criteria relates to the degree to which a system, subsystem, or equipment is operable and in a committable state at the start of a mission, when the mission is called for at an unknown, i.e., a random, time. Simply put, availability is the proportion of time a system is in a functioning condition. Availability also relates to the ratio of (a) the total time a functional unit is capable of being used during a given interval to, and (b) the length of the interval.
- Performance – Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used. It may be defined in terms of short response time for a given piece of work, high throughput (rate of processing work), low utilization of computing resource(s), high availability of the computing system or application, fast (or highly compact) data compression and decompression, and high bandwidth / short data transmission time.
- Ease of use – The term ease of use is used to denote the how ease a people can employ a particular tool or other human-made object in order to achieve a particular goal.

- Diagram layout control and legibility – This aspect relates to the amount of control the user has over the working diagram and how legible is the diagram for the user.
- Standard compliance – This criteria takes in consideration aspects related to whether the tool is adherent to its standards and constrains the models created by the user to fit these standards.

However it is very difficult to obtain objective measurement data to determine whether the tool could fit these goals, or even to state the level of its agreement, since they have high subjectivity. Therefore we decided to compare the performance of the WAKAME tool with another tool. This comparison had to happen according to the identified business objectives, and the modeling performance would be measured considering the same business objectives in both tools. For accomplish this, we needed to choose a CASE tool where the user could create Kobra2 models and that meets the following constraints:

- UML 2 compliance - For being compliant to the UML 2, a tool should present both: concrete and abstract syntax compliance. Concrete syntax compliance, relates to whether users can continue to use a notation they are familiar with across different tools and have their communication easier, because the elements could have the same meaning across different tools. Abstract syntax compliance means that users can move models across different tools, even if they use different notations, what is essential when the language is used as a basis for model-driven development. The user might want to use tool A for creating the model, tool B for validating the model, tool C for somehow transforming/enhancing the model and tool D for generating code from it, this is only possible if all the tools are compliant to the UML 2 abstract syntax.
- OCL 2 support – The tool should be compliant with the OCL metamodel and to support constraints as well.
- Availability – The tool should be available for use, so if the tool A is proprietary and we have no license for its use it could not be used in the comparison.
- Industry acceptance – The tool had to be widely accepted by the software industry around the globe. By satisfying this constraint, the tool would be considered one of the most important tools in its application area.

- Portability – The tool should be able to operate under different platforms, so its user would not be restricted to use tool under one specific environment.
- Multi user support – The tool should allow different users to shared and work on the same model.

For comparison, the tool we found that met these constraints was the IBM Rational Software Modeler (RSM). The RSM could offer both, high and concrete syntax compliance to the UML 2 metamodel and support to OCL 2 expressions. Regarding its availability, we had academic licenses of this tool at UFPE, place where the case study would be executed, and the RSM tool has high industry acceptance as an UML and model driven CASE tool. The IBM RSM is also developed in Java technology what means that it can work in different operational systems and hardware platforms, and it enables multiple developers working with the same model through the use of an external version control system.

Once defined the experiment, its goals and the tool that was going to be used, we had to choose a Kobra2 model for the experiment. We decided to use a representative Kobra2 model, in terms of number of components, number of views and model applicability. The chosen model was the Kobra Web App Framework, which comprises 8 components with a total of 18 views and represents a real world model, being also part of this work (Section 3.2). The KWAF model was also consolidated, so the experiment participants wouldn't have to take design choices, or to create the model from scratch. Instead, they would follow the existing model, and focus their attention only on the model creation.

To the user's evaluation, we developed a survey, which would take in account aspects related to availability, performance, ease of use, diagram layout control and legibility and standard compliance, since these were our main goals to meet. These criteria and the survey coverage to them are explained bellow:

- Availability – At the survey, the availability questions were related to time and ease of installation of the tool and the ease to setup one environment for multiple users collaborate in the same model.
- Performance – This criteria has been broken down into the survey in questions regarding startup time; upload and download time from and to a persistent medium; and resource consumption (RAM, CPU, network bandwidth).

- Ease of use – In the survey, the ease of use was taken in account by considering the number of menus needed to go through to start drawing models, the number of actions (clicks, go to pane, fill up fields, etc.) needed to add a model element or property, the number of features on the screen that you don't understand and need, ease of navigation between various diagrams of the same model, ease of previously created model elements for inclusion in current diagram, ease of drag and drop elements between diagrams and overall modeling speed.
- Diagram layout control and legibility – In the survey, this aspect was covered by questions related to the ease to change diagram layout (boxes and links), legibility of automatically re-laid-out functionality (after change), maximum size of diagramming area which is based in the ability of hide unwanted elements in the screen to maximize the working area and ease of undoing changes.
- Standard compliance – This aspect was broken down in the survey questioning metamodel non-conformance detection, helpfulness of warning messages, enforced coherence between several diagrams of the same model and ease of introducing process-specific constraints on models.

The user could fill in the survey questions regarding these criteria about IBM Rational Software Modeler and the same questions about WAKAME tool. The survey also contained questions regarding the user's education level and both tools familiarity.

For participating on the experiment, we invited the ORCAS Research Group members, which comprises researchers of different scholarship degree, starting from undergraduate students to PhD holders. They also have different levels of knowledge about Kobra2 and experience in IBM RSM tool.

After the definition of the experiment goal, the tool that we were going to compare WAKAME with it, the comparison criteria and the experiment participants, we created a planning for the case study execution. Therefore, a schedule for the experiment was defined containing the following activities:

1. Presentation of the Kobra2 method to the participants, its principles, the main views and sample views (**Error! Reference source not found.**);
2. Presentation of the KWAF model, which was going to be modeled by the participants in the two CASE tools;

3. Distribution of the case study;
4. Installation of the IBM RSM at the computers that don't have it;
5. Half of the participants model for 50 minutes using the IBM RSM, while the others model with the WAKAME tool also for 50 minutes;
6. After 50 minutes, who were modeling with the IBM RSM starts to model with the WAKAME tool, while the other group would take the IBM RSM;
7. After finishing the specified time the participants were surveyed about their experience in modeling with both tools (Appendix A).

5.2. Execution and Analysis of the Case Study

For the execution of the case study, a room was reserved for accomplishment of the presentation and to make possible that all the participants were in the same atmosphere to not suffer influences of different external factors.

Among the 26 members of the ORCAS research group invited to participate, only fourteen attended. The execution happened according to the planned stages, being: 30 minutes for the presentation of the Kobra2 method; 20 minutes for the presentation of the KWAF model; 40 minutes for the installation of the IBM RSM tool; and, 1 hour and 50 minutes for the accomplishment of the models by the participants (50 minutes in one tool, 10 minutes for interval, and 50 minutes in another tool). After these stages, all the participants answered the survey.

Among the participants, eight didn't possess the tool RSM installed in their computers and they had to accomplish this installation. The IBM RSM application has approximately 1.2 GB of size in disk and after the installation it occupies 980 Mb approximately. The installation of RSM in the participants' computers happened without problems, and it took on average 15 minutes each installation.

Now we will present the results obtained in the research. The Figure 5.1 displays a graph with the educational level of the case study participants. It can be observed that the educational level among the participants was diversified, tends, from undergraduate student to a person PhD. holder. Already the participants' experience with the RSM and WAKAME tools can be seen in the Figure 5.2, where it can be observed that most never

used none of the both tools. However, we see a larger number of people that already had some contact with RSM than with WAKAME.

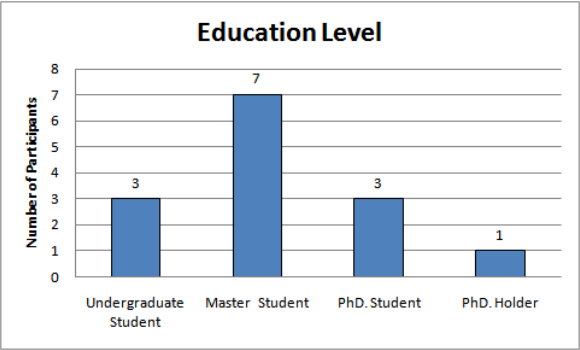


Figure 5.1 - Educational Level of Participants

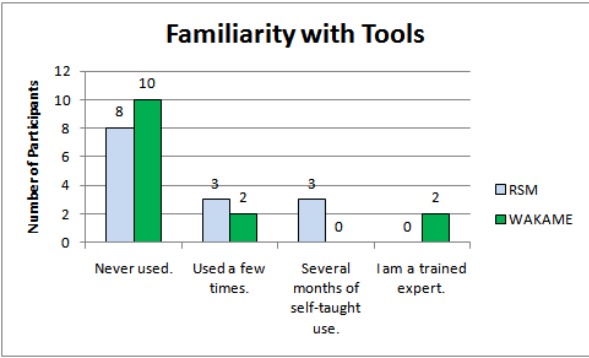


Figure 5.2 - Familiarity of the Participants with the RSM and WAKAME tools.

The first item verified in the case study was the percentage of the models accomplished in each tool by the participants in the given time (50 minutes). To calculate this measured, the weight of each view of the model used for the case study was defined through the amount of existent elements: ComponentClasses, Classes, Enumerations, Operations, Attributes, Parameters, Associations and Stereotypes (since the same is not generated automatically in RSM as it is in WAKAME). For the operational views, that are just textual for the OCL constraints writing, the weight ONE was defined. The Figure 5.3 exhibits the percentage of the model done in each tool, for each participant, where each line represents a participant (P1 to P14). The left side of the graph represents the percentage done in WAKAME and the right side the percentage done in RSM. The graph bellow exhibits the general average, of all the participants, of the percentage done in each tool. Observing these results, we can say that the time spent for modeling in RSM is approximately the double of the time spent in WAKAME.

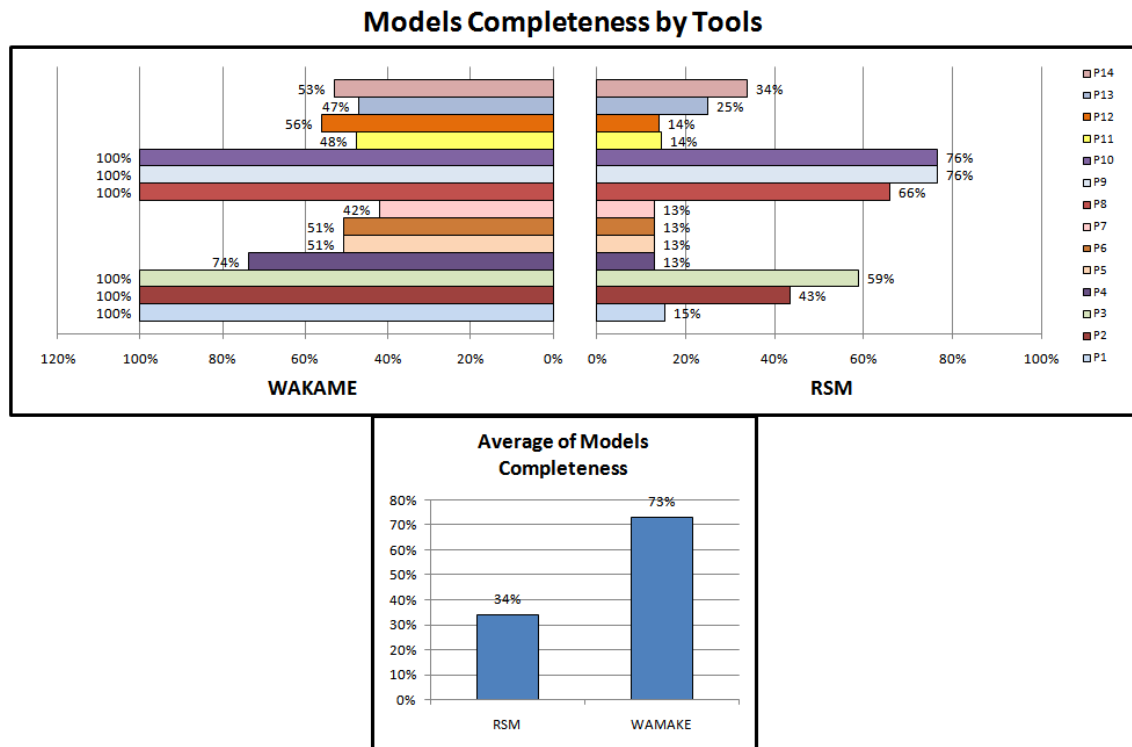


Figure 5.3 – Model Completeness Comparison.

Looking more closely, in comparison with Table 5.1, we noticed that only the participants that already had some contact with the RSM tool got to model at least 40% of the proposed model, differently of the result obtained by WAKAME, where all did more than 40% of the model. It was also observed that the participants P2 and P3, that never had contact with the WAKAME tool, got to complete the modeling. However, it is also observed that these two participants, and the others that got finish the model in the WAKAME tool, P1, P8, P9 and P10, already had at least some contact with the RSM tool.

Of these observations, we can say that the WAKAME tool possesses a better performance for the accomplishment of Kobra2 models, on average the double of the performance in relation to RSM, and for users that already had some contact with the RSM, the learning curve is very low, as in the participants' case P2 and P3. Finally, it is noticed that the participants that never had provided with none of the two tools had the lowest performance.

After having done the analysis of the percentage of the models accomplishment, the analysis of the answers from the participants to the form began, we compile these answers by criteria.

Table 5.1 - Familiarity of Participants with the Tools RSM and WAKAME.

How familiar are you with the Tool?		
	WAKAME	RSM
P1	Used a few times	Used a few times
P2	Never used	Used a few times
P3	Never used	Several months of self-taught use
P4	Never used	Never used
P5	Never used	Never used
P6	Never used	Never used
P7	Never used	Never used
P8	Used a few times	Used a few times
P9	I am a trained expert	Several months of self- taught use
P10	I am a trained expert	Several months of self- taught use
P11	Never used	Never used
P12	Never used	Never used
P13	Never used	Never used
1P4	Never used	Never used

The first criterion to be analyzed is Availability (Figure 5.4). It can be observed that the RSM tool had in larger part the answers *Sub-par* and *Dismal* for both questions done, while WAKAME had most of the answers *Excellent*. A possible reason for this fact is that RSM possess a relatively slow installation process and the same has a difficult download process. Another fact is that RSM doesn't possess any preinstalled tool for aid in the teamwork development, being necessary the plug-in installation. On the other hand, the WAKAME tool, for being Web, is easily accessible and for its execution it is necessary just to download the necessary JARs file, which the browser automatically does. The size of these necessary files is of approximately 1.5 Mb, almost a thousand times minor than the RSM installation file.

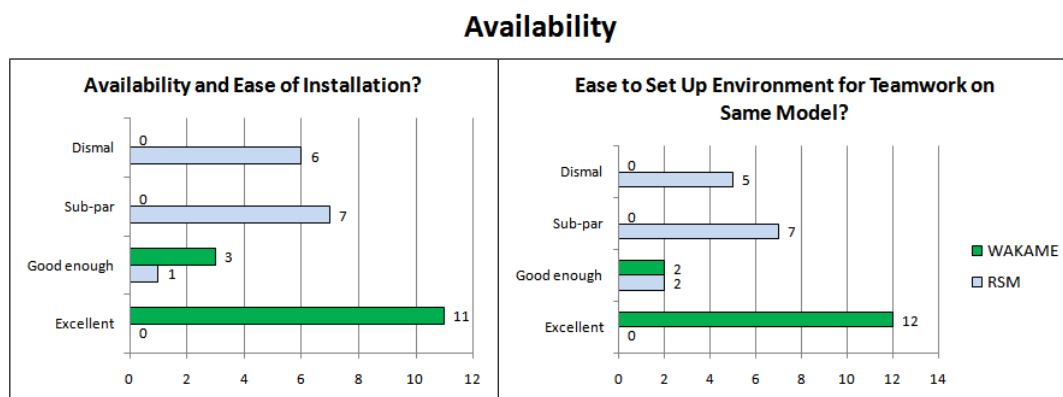


Figure 5.4 –Availability Related Answers Comparison.

In the criterion Performance (Figure 5.5), the first question refers to the necessary time to start the tool, for it to be ready for the use. In this point, in understanding of the

participants, the WAKAME tool had a better performance than the RSM tool, being most of the answers *Good enough* for WAKAME against *Sub-par* for RSM. In this case, the time spends of start-up of WAKAME is practically the time spend for the download of the necessary files, because, due to its reduced size, the download is fast. Yet RSM, for being a quite robust tool, and also for using the Eclipse's platform its time of load to the memory is large, though it is not necessary to download any file.

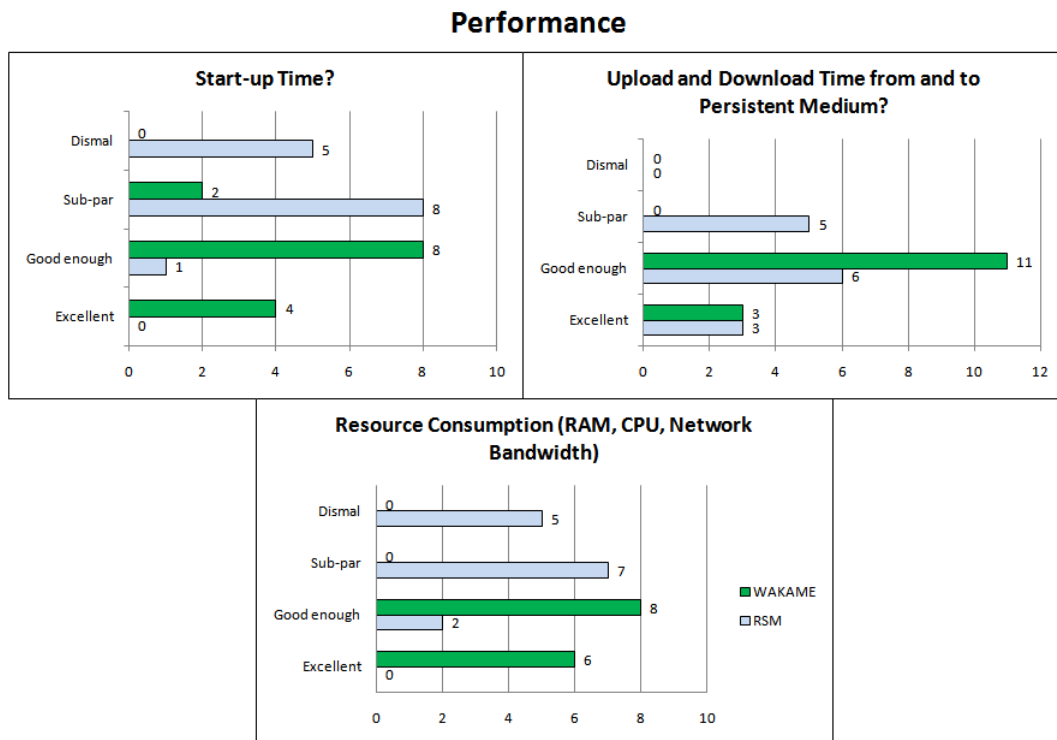


Figure 5.5 - Performance Related Answers Comparison.

The second question, of the Performance criterion, refers to the necessary time to load and to save a model in a persistent medium. The RSM had a better evaluation in relation to the other questions of this criterion. It occurs because the persistent medium used by RSM is the local disk of the computer, which possesses a fast access, but the processing necessary to do it still little costly. As the medium of persistence used by WAKAME is in the cloud, the result of this subject is directly tied to the user's band width. In this case, as the case study was accomplished at CIn/UFPE, and it has a satisfactory access to the internet, WAKAME obtained a good result.

The last evaluated question of the Performance criterion was related to the consumption of computational resources. Again, WAKAME had the results between *Good enough* and *Excellent*, because its client component is light and it demands little processing

and memory, but it already makes use of a considerable band width. Yet the RSM had the majority of the answers *Sub-par* and *Dismal*, due to its size, consuming a lot of processing resource and memory. To execute the RSM satisfactorily it is necessary a machine with a good hardware configuration, different from WAKAME, that doesn't have this need.

The next criterion evaluated, *Ease of Uses*, that can be considered one of the most important, because if a tool is complicated to use its learning curve is high and, consequently it is necessary a high investment for its adoption, what usually discourages the tool adoption. In general, WAKAME obtained very good evaluations in contrast with the terrible evaluations of RSM, what can be perceived in the Figure 5.6. This happened because RSM is a tool with several objectives and with support to several functionalities, what generates an enormous overhead of information and options for work. Differently, WAKAME was projected to assist only the needs of the KobrA2 method, leaving its interface simple with only the necessary information and options.

The first question of the *Ease of Use* criterion has the intention of evaluating the necessary "effort" to begin modeling, in other words, how many screens, how many choice options, how many buttons are necessary to click for the user begin to writhe the model. WAKAME has the option of accessing a model already existent or to begin a new one. To access an existent model the user should access the page of models listing, to choose the model wanted in the combo-box and, finally, to access the modeling screen. For the creation of a new model, the user should access the page of model register, in which is necessary to fill out few information and after this to make the process described to access an existent model. That is contemplated in the result of the evaluation with most of the answers *Excellent*, due to the easiness and intuitive access/creation of models.

To access a model in RSM, so a new model, as a model already existent, it is necessary, in the beginning of the tool execution, to choose the wanted workspace, as the whole tool that is based on Eclipse. After this, it is necessary to choose the creation of a new modeling project, among the several other different project options. Then it is necessary to choose, among several options of modeling templates, which will be used for modeling, to finally to begin modeling. To access an existent model it can be done directly, if the same is already in the work place, otherwise, it is necessary to do the whole procedure of project import provided by the Eclipse. In the case study, it was noticed certain difficulty among the participants that never had contact with RSM to begin a new

model. This occurred because if the chosen modeling template is not the most specific, some necessary modeling elements for KobrA2 are hidden.

The next question of the *Ease of Use* criterion has as objective to measure the usability in agreement with the amount of necessary clicks to include a new element in the model, or a new property of an existing element. In this case, WAKAME was evaluated, in the major part, as *Excellent* and, the RSM as *Dismal*. As dictated previously, WAKAME only makes available what is necessary for the view in subject, facilitating like this, the insertion of a new element. To add properties of an element, in WAKAME it happens through the inclusion of its concrete syntax, without the need to access the properties palette, as in RSM. In the question regarding the number of available features for the user, the evaluation was similar to the previous, this happened because of the same reason already explained before – WAKAME only exhibits the necessary and RSM, because of its general purpose, exhibits more information and options than the user usually needs.

The next three questions of the *Ease of Use* criterion refers to the navigation and the easiness to include items in diagrams. In the first one, that it is the navigation easiness, WAKAME had a good evaluation, in contrast with RSM. That is due to the fact of the navigation in WAKAME to be guided through component and the orthographically options of the views. Yet in RSM, depending on the nesting of the packages, to navigate through the diagrams becomes very complex. However, in the other two questions, regarding the inclusion of existent elements in different diagrams both tools had a similar result, where WAKAME was between *Good enough* and *Excellent*, and RSM among *Dismal*, *Sub-par* and *Good enough*.

The last question of *Ease of Use* refers to the speed for the creation of models. This question depends directly of the previous evaluations, because, a tool with a good usability usually possesses a good speed for development due to the learning curve, simplicity and intuition of its interface. Due to the simple and objective interface for the KobrA2 method, the time spent to model KobrA2 in WAKAME is much smaller than in RSM.

Ease of Use

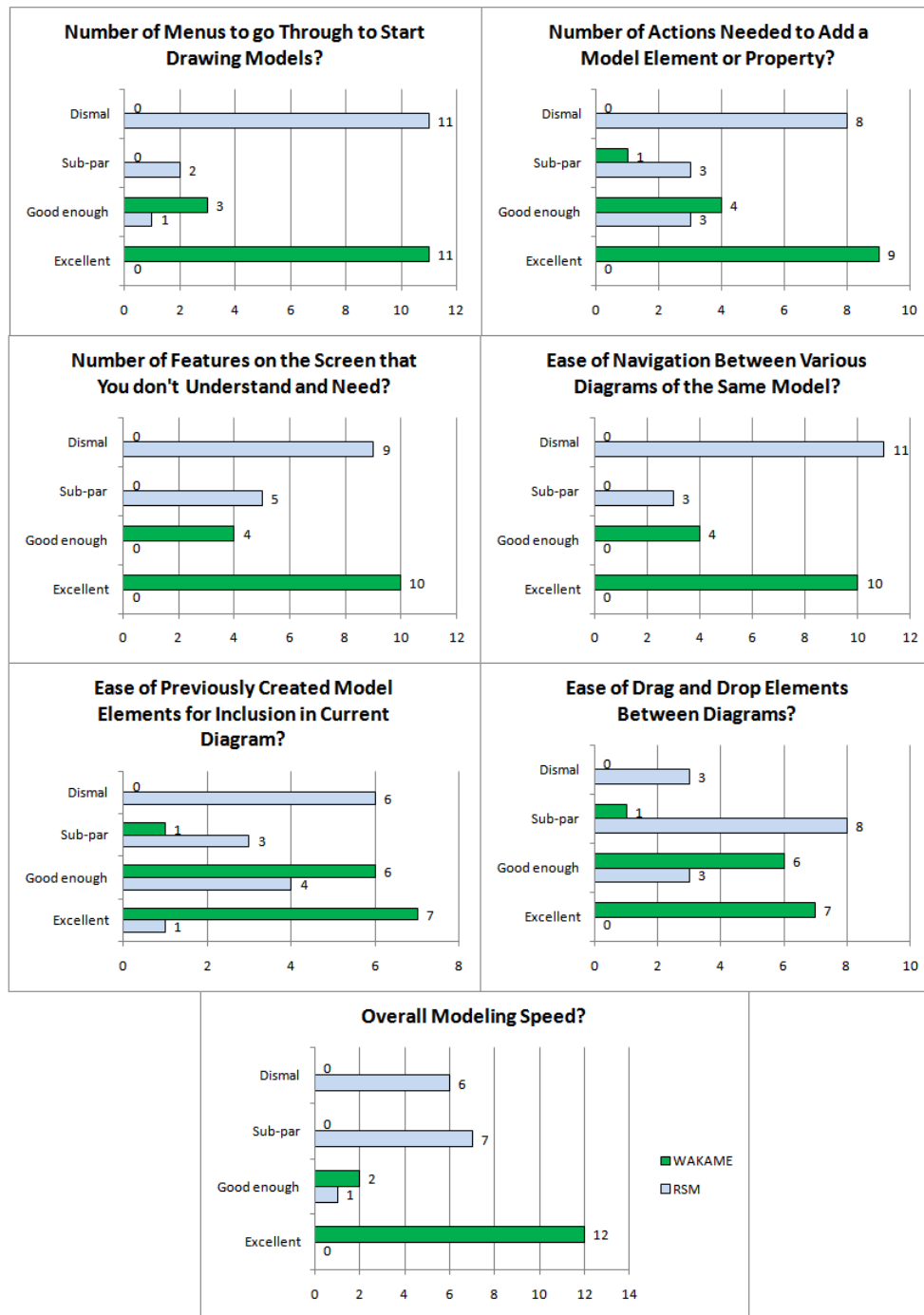


Figure 5.6 - Ease of Use Related Answers Comparison.

Another evaluated criterion was *Diagram Layout Control and Legibility*, where the result can be seen in the Figure 5.7. In the first two questions WAKAME had a good evaluation, because its interface allows easily manipulation of the size and position of the elements and mainly it allows total control above the disposition of the links in the model. Yet in RSM, its low rate in this criterion, pointed by the participants, is due because of its

low usability, mainly for the manipulation and links formatting that the tool possesses. In the question regarding the maximum area of the diagram, both tools had a good appraised performance, because both possess resources to hide the undesirable options and to exhibit only the modeling area. However, WAKAME was gotten a larger useful area than RSM.

The last question of the criterion mentioned above, concerns to allow to undo modifications, WAKAME had an inferior evaluation then RSM. This occurs because WAKAME still doesn't possess the functionality to undo actions. However, WAKAME allows not to save the modifications done (before sending to cloud) and to bring the information from the cloud and overwrite the current information. Yet RSM obtained the evaluation *Good enough*, because it provides the action *Undo*, even this not work for a certain subset of actions.

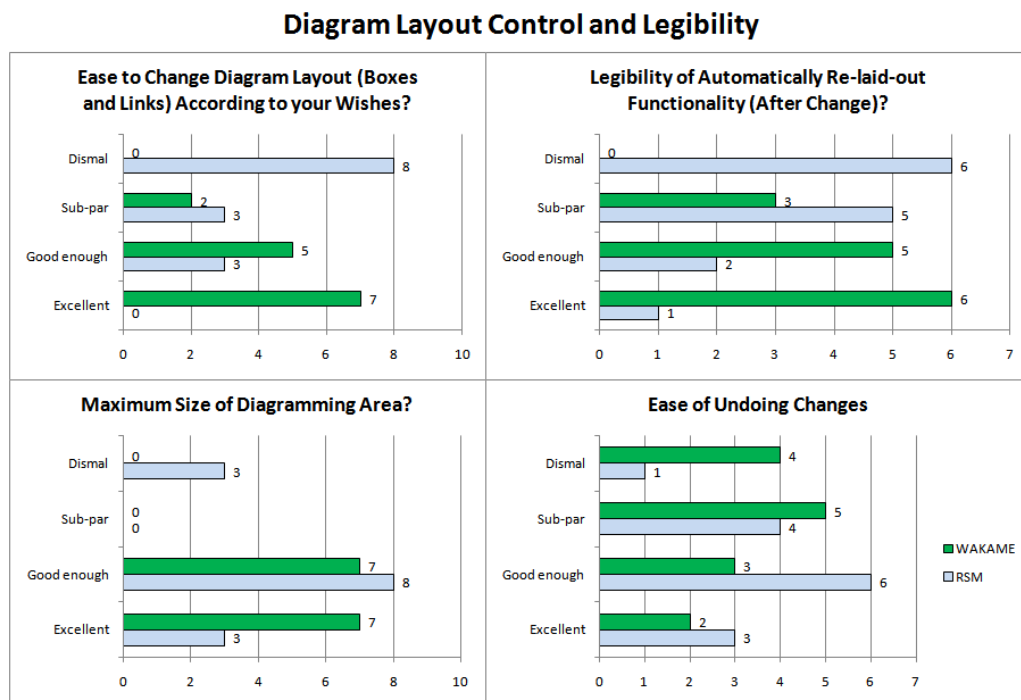


Figure 5.7 - Diagram Layout Control and Legibility Related Answers Comparison.

The last appraised criterion was *Standard Compliance*. Were appraised questions as: the detection of no conformities with the metamodel, the capacity to express the non conformities through warning messages, the capacity to maintain the coherences of elements that appear in several diagrams and the capacity of inserting specific restrictions of process in the model.

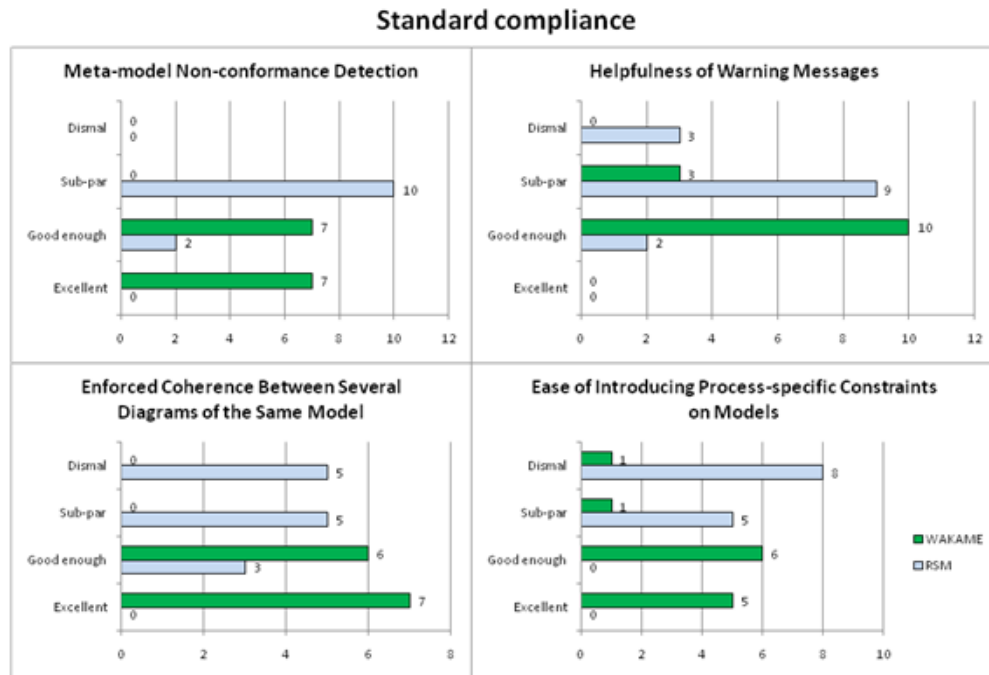


Figure 5.8 - Standard Compliance Related Answers Comparison.

The Figure 5.8 shows the results of this criterion, where WAKAME stood out in the capacity to verify no conformities in the model, in the capacity to maintain the coherence between the diagrams (views) of a model (SUM) and in the capacity to include specific restrictions of process in the model. However, in the subject regarding warning messages, WAKAME received *Good enough* and *Sub-par*. Yet RSM, had an evaluation close to *Dismal* and *Sub-par*, except for the question regarding the checking of no conformities, because in this case RSM made these verifications in the UML model level.

5.3. Experiment Findings

During the execution of the case study, some new issues of the WAKAME tool were discovered, and properly reported in the issue list of the project.

Another factor observed, that was not in the survey, is that the participants that didn't know none of the tools, didn't also have knowledge, or had little knowledge, on Kobra2. Differently, the participants that already knew one of the tools, already possessed a reasonable knowledge about Kobra2. The factor of a participant to have knowledge in Kobra2 might have influenced his performance in WAKAME.

Also this case study helped to show improvement points in the tool, as example, the need of the functionality *Undo*, also the need to define a way to reduce the traffic of data in the net in way to improve the warning messages.

It is important to emphasize that WAKAME is a specialist tool, in the sense of assisting only the Kobra2 method, differently of RSM that is of general purpose. This makes possible that WAKAME has a simple interface and a low learning curve in relation to RSM, for models done in the Kobra2 method.

In general WAKAME had a better performance in relation to RSM, in almost all points evaluated. Making a weighted average of the answers of the questions related to amount of answers (considering *Dismal* - 1, *Sub-par* - 2, *Good enough* - 3, and *Excellent* - 4) we have in Table 5.2 the summary of the evaluation.

Table 5.2 – Assessment Summary

Assessment Summary		
	WAKAME	RSM
Availability	<i>Excellent (4)</i>	<i>Sub-par (2)</i>
Performance	<i>Good enough(3)</i>	<i>Sub-par(2)</i>
Ease of use	<i>Excellent(4)</i>	<i>Sub-par(2)</i>
Diagram layout control and legibility	<i>Good enough(3)</i>	<i>Sub-par(2)</i>
Standard compliance	<i>Good enough(3)</i>	<i>Sub-par(2)</i>

With the data of Table 5.2 we noticed that the strong points of the WAKAME tool, in general, are its availability, for being web, and its use easiness, consequence of being totally specific for Kobra2.

5.4. Chapter Remarks

This chapter exhibited the evaluation study of the tool WAKAME. It showed the experiment definition, its execution, the obtained data and the identified findings, which demonstrated good acceptance and promising results for the WAKAME tool.

CHAPTER 6

CONCLUSION

This Chapter presents the contributions and limitations of the WAKAME Server, future work on WAKAME Server, limitations of WAKAME tool, future work of WAKAME tool and concluding remarks.

6.1. Contributions

This work presents contributions that were organized and classified as contributions to the CASE Tool Engineering; to MDE, Kobra2 and CBE.

The identified Contributions to CASE Tool Engineering were:

- Definition of an architecture to model and implement Web CASE tools for diagramming elements related to one specific metamodel;
- Production of a CASE tool that: (a) verifies conformance of diagram elements to the metamodel definition; (b) create models based on UML2 with OCL support; (c) is lightweight, cross platform, Web, easy to use and to extend; and (d) is the first tool to support Kobra2.

Identified contributions to MDE, Kobra2 and CBE:

- The creation of KWAF, a framework for modeling web applications, presented in the Section 3.2. Also the modeling of a web application of photo album was accomplished to the first evaluation of KWAF;
- The implementation of a repository of models aligned with the metamodel of Kobra2, presented in the Section 4.2.4 and 4.2.5;
- Implementation of transformation rules to spread the modifications done in a view of the SUM and from the SUM to the related views related with the modification, exhibiting warning messages when found some restriction violation, presented in the Section 4.2.6;

- Implementation of the WAKAMWebService component in the GAE platform in agreement with the developed PIM and its integration with the WAKAMEGUIClient component , as shown in the Section 4.2.6;
- Creation and publication of a CASE tool for MDE, CBE and Kobra2 that supports the separation of concerns and orthographic modeling - the WAKAME as a completely;
- Validation of the method and of Kobra2 through the accomplishment of the models elaborated in this work, KWAF, Photo Album and WAKAME tool, accomplished in the sections 3.2, 3.3 and 4.1;
- Revision and correction of the Kobra2 metamodel during the implementation of the repository of models. The identified corrections are striped in the section 4.2.4;
- The evaluation of the Kobra2 metamodel through the WAKAME tool, because with the tool it is possible to create models totally compatible with the specification of the Kobra2 metamodel.

6.2. WAKAME Server Limitations

The base of the WAKAME tool is completed but at the time of this writing, there are limitations regarding the WAKAME Server, which are:

- WAKAME Server current implementation doesn't support all the Kobra2 metamodel elements, as can be seen in section 4.2.4;
- WAKAME Server does not handle concurrency for a real time multi user collaborative model edition, and;
- It does not allow from one model to refer elements from another models defined in the tool.

In addition these limitations, there are still bugs and enhancement opportunities registered at the project issue tracker at the address <http://code.google.com/p/wakame/issues/list>.

6.3. Future work on WAKAME Server

The identified possibilities of future works for WAKAME Server are:

- The resolution of the limitations exhibited in the Section 6.2;
- Modifications in the repository of models so that the personalization is allowed, by the user, of the used metamodel, and;
- Refactoring of the structure of packages of the metamodel and elimination of the elements that are not used and updating of the metamodel to the last available version of Kobra2.

6.4. Limitations of WAKAME as a whole

We can enumerate the following limitation to the WAKAME tool:

- The tool doesn't support all Kobra2 metamodel;
- There is still no support to create GUI models (GUIPIMUF), and;
- It does not handle real time multi user collaborative model edition.

6.5. Future work on WAKAME as a whole

The identified possibilities of future works for WAKAME as a whole are:

- The resolution of the limitations exhibited in the Section 6.4;
- Allow the user to define your metamodel and use it in the WAKAME tool, and;
- Executable code generation from models defined in the tool.

6.6. Concluding Remarks

With the realization of this work, we could confirm the theoretical hypothesis that it is possible to define and validate a Kobra2 model repository that implements View-Sum-View transformations and does automatic constraint verification in these models. It was exhibited that this repository allows its integration with a Web GUI client, composing as a whole one CASE tool that supports Kobra2 method. As it was described, this work has justified itself because, the modeling process using the Kobra2 method using the existent modeling CASE tools is a complicated and costly task, because none of these tools possesses support for the multi-view modeling with the rules of transformations defined by Kobra2 for consistencies among these views.

The development of the WAKAME tool took place through the collaboration among two master's degree students (being one of these the author). The focus of this work was the development of the server component of the tool, responsible for the persistence of the model, and creation of the transformations rules for the propagation of the changes among View-SUM-View.

The work was accomplished in three stages. The first consisted of the bibliographical rising of the related areas to the subject of the research. The second stage was the accomplishment of the modeling and implementation of the WAKAME Server. And the third and last stage was accomplished a case study for the evaluation of the WAKAME tool, and of this case study it was obtained positive results in relation to usability, performance and availability.

The main contributions at the end of this work were: (a) the first case study to validate the KobrA2 method for rich web applications with 2D graphics, (b) the high-level design of WAKAME as a case study validating KWAF, (c) the implementation of cloud-based services for view persistence, view integration into a persistent SUM and SUM verification and (d) the integration and testing of these services with WAKAME's GUI web client for orthographic view edition.

CHAPTER 7

BIBLIOGRAPHY

- Atkinson *et al*, 2001 Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laguna, R., Muthig, D., Paech, B., Wüst, J. and Zettel, J. **“Component-Based Product Line Engineering with UML”**. The Component Software Series. Addison-Wesley. 2001.
- Atkinson *et al*, 2009 Atkinson, C., Robin, J. P. L. and Stoll, D. **“Kobra2 Technical Report”**, 2009.
- ATL, 2009 ATLAS Transformation Language. 2009. Available at: <<http://www.eclipse.org/m2m/atl/>>. Last access: 03/01/2009.
- Azure, 2009 Microsoft Azure. 2009. Available at: <<http://www.microsoft.com/azure/default.aspx>>. Last access: 05/03/2009.
- Backvanski and Graff, 2005 Backvanski, V. and Graff, P. **“Mastering Eclipse Model Framework”**. EclipseCon2005. 2005.
- Bauer and King, 2004 Bauer, C. e King, G. **“Hibernate in Action”**. Manning Publications. 2004.
- Beck and Andres, 2004 Beck, K., Andres, C. **“Extreme Programming Explained: Embrace Change”**. Addison-Wesley Professional. 2 ed. 2004.
- Beydeda and Gruhn, 2005 Beydeda, S. and Gruhn V. **“Model-Driven Software Development”**. Springer-Verlag New York, Inc. Secaucus. NJ. 2005.
- BIGTABLE, 2009 Google BigTable. 2009. Available at: <<http://labs.google.com/papers/bigtable-osdi06.pdf>>. Last access: 02/01/2009.
- Budinsky *et al*, 2003 Budinsky, F., Steinberg, D., Merks, E., Ellersick, R. and Grose, T., J. **“Eclipse Modeling Framework: A Developer's Guide”**. Addison Wesley. 2009.
- Buschmann *et al*, 1996 Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal M. **“Pattern-Oriented Software Architecture: A System of Patterns”**. John Wiley and Sons. 123- 168. 1996.
- Cavaness, 2004 Cavaness, C. **“Programming Jakarta Struts, 2nd Edition”**. O'Reilly. 2 ed. 2004.

DAO, 2009	Data Access Object Pattern. 2009. Available at: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html . Last access: 21/02/2009.
DI, 2005	Object Management Group. “ UML Diagram Interchange (DI) Specification ”. 2005. Available at: http://www.omg.org/cgi-bin/doc?ptc/2005-06-07 .
DJANGO, 2009	DJango. 2009. Available at: http://www.djangoproject.com . Last access: 07/01/2009.
DOTNET, 2009	Microsoft .Net. 2009. Available at: http://www.microsoft.com/NET/ . Last access: 29/05/2009.
Dresden, 2009	Dresden OCL Toolkit. 2009. Available at: http://dresden-ocl.sourceforge.net/ . Last access: 04/01/2009.
EC2, 2009	Amazon Elastic Compute Cloud (Amazon EC2). 2009. Available at: http://aws.amazon.com/ec2/ . Last access: 05/03/2009.
Eclipse, 2009	Eclipse Project. 2009. Available at: http://www.eclipse.org/ . Last access: 05/01/2009.
Ecore, 2009	Ecore. 2009. Available at: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.emf.doc/references/javadoc/org.eclipse/emf/ecore/package-summary.html . Last access: 25/04/2009.
EMF, 2009	Eclipse Modeling Framework. 2009. Available at: http://www.eclipse.org/modeling/emf/ . Last access: 07/08/2009.
EMF, 2009a	The Eclipse Modeling Framework (EMF) Overview. Available at: http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html . Last access: 20/03/2009.
EMFV, 2009	The Eclipse Modeling Framework (EMF) Validation Framework Overview. 2009. Available at: http://help.eclipse.org/ganymede/topic/org.eclipse.emf.doc/references/overview/EMF.Validation.html . Last access: 04/01/2009.
Epsilon, 2009	Epsilon. 2009. Available at: http://www.eclipse.org/gmt/epsilon/ . Last access: 14/03/2009.
Eriksson <i>et al</i> , 2003	Eriksson, H. E., Penker, M., Lyons, B. And Fado, D. “ UML 2 Toolkit ”. Wiley Computer Publishing. 2003.
FFIEC, 2009	“ Software Development Techniques ”. 2009. Available at: http://www.ffiec.gov/ffiecinfobase/booklets/d_a/10.html . Last access: 31/06/2009.

Fowler and Stanwick, 2004	Fowler, S. e Stanwick, V. “ Web Application Design Handbook: Best Practices for Web-Based Software ”. Morgan Kaufmann Publishers. 2004.
Frankel and Parodi, 2002	Fowler, S. e Stanwick, V. “ Web Application Design Handbook: Best Practices for Web-Based Software ”. Morgan Kaufmann Publishers. 2004.
GACC, 2009	Google Account API. 2009. Available at: < http://code.google.com/apis/accounts/ >. Last access: 02/01/2009.
GAE, 2009	Google App Engine. 2009. Available at: < http://code.google.com/appengine/ >. Last access: 05/03/2009.
GAE, 2009a	Google App Engine SDK for Java v.1.2.2. 2009. Available at: < http://code.google.com/appengine/downloads.html >. Last access: 13/07/2009.
GAEJRE, 2009	The GAE JRE Class White List. Available at: < http://code.google.com/intl/en/appengine/docs/java/jrewhitelist.html >. Last access: 29/05/2009.
Gamma <i>et al</i> , 1994	Gamma, E., Helm, R., Johnson, R. and Vlissides, J. M. “ Design Patterns: Elements of Reusable Object-Oriented Software ”. Addison-Wesley Professional. 1994.
GApps, 2009	Google App. 2009. Available at: < http://www.google.com/apps/intl/en/business/index.html >. Last access: 03/01/2009.
GFS, 2009	Google File System. 2009. Available at: < http://code.google.com/p/google-gears/wiki/FileSystemAPI >. Last access: 02/01/2009.
GoogleP, 2009	Google Plug-in for Eclipse. 2009. Available at: < http://code.google.com/eclipse/ >. Last access: 13/07/2009.
Gross, 2004	Gross, H. G. “ Component-Based Software Testing with UML ”. Springer. 1 ed. 2004.
JAVA, 2009	Java. 2009. Available at: < http://www.java.com >. Last access: 21/05/2009.
JDO, 2009	Java Data Objects (JDO). 2009. Available at: < http://java.sun.com/jdo/ >. Last access: 21/05/2009.
JPA, 2009	Java Persistence API (JPA). Available at: < http://java.sun.com/javaee/technologies/persistence.jsp >. Last access: 21/05/2009.
JSON, 2009	JavaScript Object Notation (JSON). 2009. Available at: < http://www.json.org/ >. Last access: 04/01/2009.
Kermeta, 2009	Kermeta. 2009. Available at: < http://www.kermeta.org/ >. Last access: 04/01/2009.

- Kock, 2007 Kock, N (ed.). **“Information Systems Action Research”**. Integrated Series in Information Systems, Springer, New York. 2007.
- Kolovos *et al*, 2006 Kolovos, D. S., Paige, R. F. and Polack, F. A. C. **“Eclipse Development Tools for Epsilon”**. Eclipse Summit Europe. 2006.
- Kolovos *et al*, 2008 Kolovos, D., Paige, R., Rose, L. and Polack, P. **“The Epsilon Book”**. 2008. Available at: <http://www.eclipse.org/gmt/epsilon/doc/book/>.
- Kruchten, 2003 Kruchten, P. **“The Rational Unified Process: An Introduction”**. Addison-Wesley Professional. 3 ed. 2003.
- Leff and Rayfield, 2001 Leff, A. e Rayfield, J. T. **“Web-Application Development Using the Model/View/Controller Design Patter”**. Proc. IEEE International Enterprise Distributed Object Computing Conference. IEEE. 2001.
- Levitt, 2009 Leavitt, N. **“Is Cloud Computing Really Ready for Prime Time”**. Innovative Technology for Computer Professionals. 2009
- Lewandowski, 1998 Lewandowski S. C. **“Frameworks for Component-Based Client/Server Computing”**. ACM Computing Surveys. 1998.
- Luoma *et al*, 2004 Luoma, J., Kelly, S., and Tolvanen, J.-P. **“Defining Domain-Specific Modeling Languages - Collected Experiences”**. 4th Object-Oriented Programming Systems, Languages, and Applications Workshop on Domain-Specific Modeling (OOPSLA 2004). 2004.
- Lutz, 2006 Lutz, M. **“Programming Python”**. O’Reilly. 3 ed. 2006.
- Marinho *et al*, 2009 Marinho, W. A. T., Machado, B. B., Pereira, F. M. and Robin, J. P. L. **“Um Framework UML2 para Modelagem de Aplicações Web dentro de um Processo de Engenharia de Software Dirigido por Modelos e Baseado em Componentes”**. III Workshop de Desenvolvimento Rápido de Aplicações. 2009.
- Marinho, 2009 Marinho, W. A. T. **“A Web GUI for a Multi-View Component-Based Modeling CASE Tool”**. Master Thesis (Master Degree) – Federal University of Pernambuco, Recife, 2009.
- MDR, 2009 Metadata Repository. 2009. Available at: <http://mdr.netbeans.org/>. Last access: 01/03/2009..
- Mendes and Moses, 2006 Mendes, E. and Mosley, N. **“Web Engineering”**. Springer. 2006.
- MOF, 2006 Object Management Group. **“Meta-Object Facility (MOF) 2.0 Specification”**. 2006. Available at: <http://www.omg.org/spec/MOF/2.0/>.

OCLE, 2009	OCLE 2.0 - Object Constraint Language Environment. 2009. Available at: < http://lci.cs.ubbcluj.ro/ocle/index.htm >. Last access: 04/01/2009.
OCLMDT, 2009	Eclipse OCL MDT. 2009. Available at: < http://www.eclipse.org/modeling/mdt/downloads/?project=ocl >. Last access: 04/01/2009.
Octopus, 2009	Octopus: OCL Tool for Precise Uml Specifications. 2009. Available at: < http://octopus.sourceforge.net/ >. Last access: 04/01/2009.
OMG, 2009	Object Management Group. 2009. Available at: < http://www.omg.org/ >. Last access: 03/01/2009.
OMG, 2009a	Object Management Group. “UML 2.0 Metamodel in Rose” . 2009. Available at: < http://www.omg.org/spec/UML/2.1.2/ >. Last access: 04/01/2009.
Papyrus, 2009	Papyrus UML. 2009. Available at: < http://www.papyrusuml.org/ >. Last access: 09/06/2009.
PayPal, 2009	PayPal. 2009. Available at: < http://www.paypal.com/ >. Last access: 04/01/2009.
Pressman, 2009	Pressman, R. “Software Engineering: A Practitioner’s Approach” . McGraw-Hill. 7 ed. 2009.
PYMOF, 2009	PyEMOF. 2009. Available at: < http://www2.lifl.fr/~marvie/software/pyemof.html >. Last access: 15/04/2009.
PYYAML, 2009	PyYAML. 2009. Available at: < http://pyyaml.org >. Last access: 08/01/2009.
Rodrigues, 2004	Rodrigues, M. G. C. “Metodologia da Pesquisa Científica” . Rio de Janeiro, 2004.
RSM, 2009	IBM Rational Software Modeler. 2009. Available at: < http://www-01.ibm.com/software/awdtools/modeler/swmodeler/ >. Last access: 07/08/2009.
S3, 2009	Amazon Simple Storage Service (Amazon S3). 2009. Available at: < http://aws.amazon.com/s3/ >. Last access: 05/03/2009.
SalesForce, 2009	SalesForce. 2009. Available at: < http://www.salesforce.com/ >. Last access: 04/01/2009.
Scanlon and Wieners, 2009	Scanlon, J., H. and Wieners, B. “The Cloud Computing” . Available at: < http://www.thestandard.com/article/0,1902,5466,00.html >. Last access: 05/03/2009.

- Schwaber, 2004 Schwaber, K. **“Agile Project Management with Scrum”**. Microsoft Press. 1 ed. 2004
- SERVLET, 2009 Java Servlet Technology. 2009. <<http://java.sun.com/products/servlet/>>. Last access: 21/05/2009.
- Shklar and Rosen, 2003 Shklar, L. and Rosen, R. **“Web Application Architecture: Principles, Protocols and Practice”**. John Wiley & Sons, Ltda. 2003.
- SPEM, 2009 Object Management Group. **“Software Process Engineering Meta-Model, Version 2.0”**. 2009. Available at: <<http://www.omg.org/technology/documents/formal/spem.htm>>. Last access: 03/01/2009.
- Stahl *et al*, 2006 Stahl, T., Voelter, M., Czarnecki, K. **“Model-Driven Software Development: Technology, Engineering, Management”**. Wiley. 1 ed. 2006.
- Stutz *et al*, 2002 Stutz, C., Siedersleben, J., Kretschmer, D., Krug, **“W. Analysis Beyond UML”**. In Proceedings of the IEEE Joint International Conference on Requirements Engineering, p.215-222, 2002.
- Subclipse, 2009 Subclipse. 2009. Available at: <<http://subclipse.tigris.org/>>. Last access: 04/01/2009.
- TheServerSide, 2009 **“Post/Redirect/Get pattern for web applications”**. 2008. Available at: <http://www.theserverside.com/patterns/thread.tss?thread_id=20936>. Last access 02/03/2009.
- Thiollent, 1986 Thiollent, M. **“Metodologia da pesquisa-ação”**. São Paulo, Cortez, 1986.
- Vinekar *et al*, 2006 Vinekar, V., Slinkman, C. W., Nerur, S. **“Can Agile and Traditional Systems Development Approaches Coexist? An Ambidextrous View”**. Information System Management, Volume 3, Issue 3. 2006.
- Warmer and Kleppe, 2003 Warmer, J., Kleppe, A. **“The Object Constraint Language: Getting Your Models Ready for MDA”**. Addison Wesley. 2 ed. 2003.
- WEBOB, 2009 WebOb. 2009. Available at: <<http://pythonpaste.org/webob>>. Last access: 08/01/2009.
- WebObject, 2009 **“WebObjects Web Applications Programming Guide: How Web Applications Work”**. 2009. Available at: <http://developer.apple.com/documentation/WebObjects/Web_Applications/Articles/1_Architecture.html>. Last access: 03/03/2009.

XMI, 2007	Object Management Group. “ XML Metadata Interchange (XMI), v2.1.1 ”. 2007. Available at: < http://www.omg.org/technology/documents/formal/xmi.htm >.
XML, 2009	Extensible Markup Language (XML). 2009. Available at: < http://www.w3.org/XML/ >. Last access: 04/01/2009.
Y!Maps, 2009	Yahoo! Local Maps. 2009. Available at: < http://maps.yahoo.com/ >. Last access: 04/01/2009.
Yousef <i>et al</i> , 2008	Youseff, L., Butrico, M. and Silva, D. “ Toward a Unified Ontology of Cloud Computing ”. Grid Computing Environments Workshop. 2008.

APPENDIX A

SURVEY: WAKAME EVALUATION

1) What is your education level?

- ☐ Undergraduate Student
- ☐ Master Student
- ☐ PhD. Student
- ☐ PhD. Holder

2) How familiar are you with IBM RSM (Rational Software Modeler)?

- ☐ 1. Never used.
- ☐ 2. Used a few times.
- ☐ 3. Several months of self-taught use.
- ☐ 4. I am a trained expert.

3) How familiar are you with WAKAME?

- ☐ 1. Never used.
- ☐ 2. Used a few times.
- ☐ 3. Several months of self-taught use.
- ☐ 4. I am a trained expert

4) How would you assess **RSM** in terms of:

A: Availability:

Availability and ease of installation?

- ☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease to set up environment for teamwork on same model?

- ☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

B: Performance:

Start-up time?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Upload and download time from and to persistent medium?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Resource consumption (RAM, CPU, network bandwidth)

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

C: Ease of use:

Number of menus to go through to start drawing models?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Number of actions (clicks, go to pane, fill up fields, etc.) needed to add a model element or property?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Number of features on the screen that you don't understand and need?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of navigation between various diagrams of the same model?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of previously created model elements for inclusion in current diagram?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of drag and drop elements between diagrams?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Overall modeling speed?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

D: Diagram layout control and legibility:

Ease to change diagram layout (boxes and links) according to your wishes?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Legibility of automatically re-laid-out functionality (after change)?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Maximum size of diagramming area?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of undoing changes:

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

E: Standard compliance:

Metamodel non-conformance detection?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Helpfulness of warning messages?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Enforced coherence between several diagrams of the same model?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of introducing process-specific constraints on models?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

5) How would you assess **WAKAME** in terms of:

A: Availability:

Availability and ease of installation?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease to set up environment for teamwork on same model?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

B: Performance:

Start-up time?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Upload and download time from and to persistent medium?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Resource consumption (RAM, CPU, network bandwidth)

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

C: Ease of use:

Number of menus to go through to start drawing models?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Number of actions (clicks, go to pane, fill up fields, etc.) needed to add a model element or property?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Number of features on the screen that you don't understand and need?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of navigation between various diagrams of the same model?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of previously created model elements for inclusion in current diagram?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of drag and drop elements between diagrams?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Overall modeling speed?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

D: Diagram layout control and legibility:

Ease to change diagram layout (boxes and links) according to your wishes?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Legibility of automatically re-laid-out functionality (after change)?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Maximum size of diagramming area?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of undoing changes:

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

E: Standard compliance:

Metamodel non-conformance detection?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Helpfulness of warning messages?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Enforced coherence between several diagrams of the same model?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.

Ease of introducing process-specific constraints on models?

☐ a. Excellent, ☐ b. Good enough, ☐ c. Sub-par, ☐ d. Dismal.