



Pós-Graduação em Ciência da Computação

“Behavioral Java Code Generation from Imperative Object Constraint Language Expressions in Platform-Independent UML Models”

By

Marcellus Antonius de Castro Tavares

M.Sc. Dissertation



Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, February/2011



Federal University of Pernambuco
Center for Informatics
Graduate in Computer Science

Marcellus Antonius de Castro Tavares

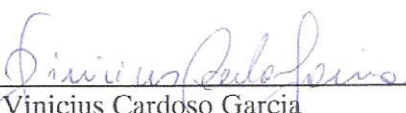
**“Behavioral Java Code Generation from Imperative
Object Constraint Language Expressions in
Platform-Independent UML Models”**

*A M.Sc. Dissertation presented to the Center for Informatics
of Federal University of Pernambuco in partial fulfillment
of the requirements for the degree of Master of Science in
Computer Science.*

Advisor: Jacques Pierre Louis Robin

RECIFE, February/2011

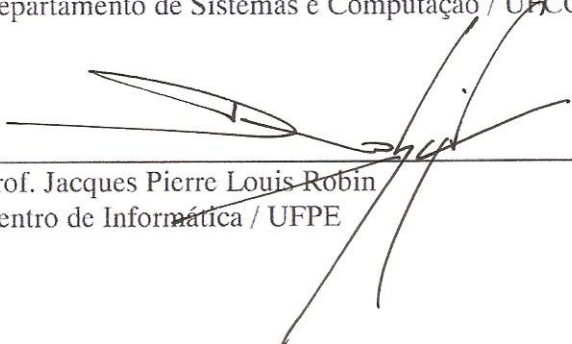
Dissertação de Mestrado apresentada por **Marcellus Antonius de Castro Tavares** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Full Behavioral Java Code Generation from Imperative Object Constraint Language Expressions in Platform-Independent UML Model**", orientada pelo **Prof. Jacques Pierre Louis Robin** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Vinicius Cardoso Garcia
Centro de Informática / UFPE

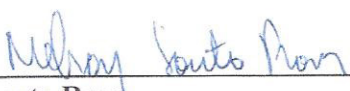


Prof. Franklin de Souza Ramalho
Departamento de Sistemas e Computação / UFCCG



Prof. Jacques Pierre Louis Robin
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 14 de fevereiro de 2011.



Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Catálogo na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Tavares, Marcellus Antonius de Castro
Behavioral java code generation from imperative
object constraint language expressions in platform-
independent UML models / Marcellus Antonius de Castro
Tavares - Recife: O Autor, 2011.
xiv, 146 p. : il., fig., tab.

Orientador: Jacques Louis Pierre Robin.
Dissertação (mestrado) Universidade Federal de
Pernambuco. Cln. Ciência da Computação, 2011.

Inclui bibliografia e apêndice.

1. Engenharia de software. 2. Transformação de modelos. 3.
Geração de código. I. Robin, Jacques Louis Pierre (orientador).
II. Título.

005.1

CDD (22. ed.)

MEI2011 – 067

*I dedicate this dissertation to my family who always gave
me all necessary support.*

Acknowledgements

Primeiramente eu gostaria de agradecer a Deus por ter me dado coragem e força para concluir mais esta etapa da minha vida.

Agradeço também aos meu pais, Fernando e Silvia, e a minha irmã Renata, que sempre acreditaram em mim e me apoiaram em todos os momentos.

A minha noiva Fabiana, por sua paciência e compreensão quando eu tive que passar diversos finais de semana me dedicando à realização deste trabalho.

Ao membros do grupo ORCAS: Ramon, Ricson, Ricardo e em especial a Thiago que me ajudou a desenvolver este trabalho, sempre compartilhando e discutindo ideias. Também um agradecimento especial a Armando que se dispôs a realizar o segundo experimento deste projeto, que foi fundamental para a conclusão do mesmo.

Ao pessoal da Liferay pelo grande incentivo.

Aos membros da banca Vinícius e Franklin que se dispuseram a avaliar a dissertação e por todas as ótimas observações que fizeram para melhorar a qualidade deste trabalho.

A meu orientador Jacques, pela orientação.

Agradeço também a todos que não foram citados aqui mas que de uma forma ou de outra, me ajudaram nesta longa jornada.

Rien ne se perd, rien ne se crée, tout se transforme.

—ANTOINE LAVOISIER

Abstract

Model-Driven Engineering (MDE) aims at improving both software productivity and quality by shifting the concerns from platform-specific programming towards platform-independent business modeling concerns.

Compared to code-driven agile methods, MDE methods involve an upfront investment in specifying detailed Platform-Independent Models (PIM). The return on such investment is clear-cut only if almost all the code is automatically generated from the PIM. However, complete code generation from a general purpose modeling language is still remarked as a challenge (Mohagheghi & Dehlen, 2008).

This master dissertation shows that this goal is reachable. It proposes an approach to overcome one of the main weakness of code generators: lack of behavioral code generation from standard modeling languages. Specifically, this dissertation proposes an innovative use of Imperative Object Constraint Language (IOCL) to completely specify operation bodies in UML (Unified Modeling Language) PIM. IOCL is a small subpart of the QVT (Query-View Transformation) standard of the Object Management Group (OMG, which also authored the UML) for model transformation specification. This work shows that the most part of the behavioral code for the most widely used implementation platform today, Java, can be automatically generated from IOCL expressions. It also shows that a behaviorally complete PIM can be efficiently specified through an experimental CASE tool that leverage a core subset of UML for structural modeling.

Within this CASE tool, the behavior code generation functionality works in synergy with other innovative functionalities resulting from other dissertations. They include view-driven edition of component-based models and its corresponding structural code generation. The combination of these functionalities, validated on the engineering of a simple information system which code was entirely generated from a PIM in UML and IOCL, contributes to materialize the most advanced software automation of the MDE vision.

Keywords: Model Driven Engineering; Behavioral Code Generation; Imperative Object Constraint Language, Unified Modeling Language.

Resumo

A Engenharia Dirigida por Modelos (MDE) visa melhorar tanto a produtividade quanto a qualidade do *software* através do deslocamento de preocupações de uma programação orientada a uma plataforma específica para uma modelagem de negócios independente de plataforma.

Comparado a métodos ágeis, os métodos MDE envolvem um investimento inicial na especificação detalhada dos Modelos Independente de Plataforma (PIM). O retorno de tal investimento é claro somente quando se consegue gerar quase todo o código automaticamente a partir do PIM. No entanto, a geração de código completo de uma linguagem de modelagem de propósito geral continua a ser considerado como um desafio (Mohagheghi & Dehlen, 2008).

Esta dissertação de mestrado mostra que tal objetivo é alcançável. Ela propõe uma abordagem para superar uma das principais fraquezas de geradores de código: a falta de geração de código de comportamental de linguagens de modelagem padrão. Especificamente, esta dissertação propõe um uso inovador da *Imperative Object Constraint Language* (IOCL) para especificar completamente o corpo das operações em UML (*Unified Modeling Language*). IOCL é uma pequena parte da especificação QVT (*Query-View Transformation*), padrão do *Object Management Group* (OMG) para a especificação de transformações de modelos. Este trabalho mostra que a maior parte do código comportamental para a plataforma mais utilizada hoje para implementação, Java, pode ser gerado automaticamente a partir de expressões IOCL. Também mostra que um PIM completo pode ser especificado de forma eficiente através de uma ferramenta CASE experimental que utiliza um subconjunto de UML para modelagem estrutural.

Dentro desta ferramenta CASE, a funcionalidade de geração de código comportamental funciona em conjunto com outras funcionalidades inovadoras resultantes de outras dissertações. Elas incluem a edição de modelos baseados em componentes e sua correspondente geração de código estrutural. A combinação destas funcionalidades, validadas a partir da engenharia de um sistema de informação simples cujo código foi inteiramente gerado a partir de um PIM especificados com UML e IOCL, contribui para materializar a mais avançada automação *software* proposta pela Engenharia Dirigida por Modelos.

Palavras-chave: MDE, UML, Imperative OCL, Transformações de Modelos, Geração de Código Comportamental.

Contents

List of Figures	xiii
------------------------	-------------

List of Tables	xvi
-----------------------	------------

1 Introduction	1
1.1 Motivation	2
1.2 Problem Statement	3
1.3 Overview of the Proposed Solution	3
1.3.1 Context	3
1.3.2 Outline of the Proposal	4
1.4 Out of the Scope	5
1.5 Statement of the Contribution	5
1.6 Organization of the Dissertation	6
2 Concepts and Terminology	7
2.1 Model-Driven Engineering	7
2.2 Model-Driven Architecture and OMG standards	8
2.2.1 Model-Driven Architecture	8
2.2.2 OMG Standards	9
UML Infrastructure	9
MOF, CMOF and EMOF	10
UML Superstructure	11
OCL	12
QVT and Imperative OCL	18
XML Metadata Interchange	26
2.2.3 Metamodeling	26
2.3 Model Transformations	27
2.3.1 Code Generation	28
Code Generation Techniques	29
2.4 Reuse-based Engineering	31
2.4.1 Component-based development	31
2.4.2 Product line engineering	32
2.5 Orthographic Engineering	32
2.6 Chapter Summary	33

3	KobrA2, KISF and WAKAME	34
3.1	KobrA2	34
3.1.1	KobrA2 goals and principles	34
	Multi-dimensional, systematic separation of concerns	35
	Top-down decomposition	35
	Standard Reuse	35
	Parsimony	36
	Uniformity	36
	Locality	36
	Encapsulation	36
3.1.2	KobrA views	36
3.2	KISF	38
3.2.1	Business Service Component	39
	Business Facade Component	39
	Business Logic Component	40
	Persistent Data Model Component	40
3.2.2	Presentation Logic Component	41
	Presentation View Component	42
	Presentation Controller Component	42
3.3	Modeling Frameworks	42
3.3.1	Eclipse Modeling Framework	43
3.3.2	MOFScript	44
3.4	WAKAME	45
3.5	Chapter Summary	47
4	WAKAME Code Generator	48
4.1	Long-Term Goals	48
4.2	Imperative OCL Compiler Architecture	50
4.2.1	Imperative OCL Input	51
4.2.2	Code Generation Strategy: PIM to Code	52
4.2.3	Adopted techniques	52
4.2.4	Dealing with pre and post conditions	53
4.3	WAKAME Architecture	54
4.3.1	The WAKAME Business Service	54
4.4	WAKAME Code Generator Top-Level PIM	55
4.5	Chapter Summary	56

5	The IOCL Compiler	57
5.1	IOCLCompiler PIM	57
5.1.1	IOCLEngine	58
	IOCLLexer and IOCLParser Component	61
	IOCLAnalyzer Component	63
5.1.2	CodeGenerator Component	65
5.2	Tests	70
5.3	Chapter Summary	72
6	IOCL Compiler Evaluations	73
6.1	Experiment Process	73
6.2	Definition	74
6.3	Hypothesis	76
6.4	First Experiment - Web Agency	76
6.4.1	Planning	77
6.4.2	Operation	78
	Web Agency - PIM	78
	Web Agency - Manual Implementation	82
	Web Agency - Automatic Generation	83
6.4.3	Analysis and Interpretation	86
6.5	Second Experiment - CHROME	88
6.5.1	Planning	89
6.5.2	Operation	90
	CHROME PIM	90
	CHROME - Manual implementation	94
	CHROME - Automatic generation	95
	Analysis and Interpretation	97
6.6	Chapter Summary	99
7	Related Work	100
7.1	Previous Studies on Behavioral Modeling	100
7.2	Works related with the use of OCL	101
7.2.1	Eclipse OCL	101
7.2.2	Dresden OCL	103
7.3	Works related with the use of IOCL	104
7.3.1	Eclipse M2M	104

7.4	Chapter Summary	106
8	Conclusion	107
8.1	Contribution	107
8.2	Limitations	108
8.3	Future Work	109
8.4	Concluding Remarks	110
	Bibliography	111
	Appendices	118
A	Imperative OCL Grammar	119
B	Web Agency - PIM	129
B.1	Structural Specification	129
B.2	Operational Specification	132
C	CHROME - PIM	137
C.1	Structural Specification	137
C.2	Operational Specification	140

List of Figures

2.1	UML Infrastructure - The <i>Core</i> Package (OMG, 2010f)	10
2.2	MOF Package Structure (OMG, 2010e)	11
2.3	Hierarchy of diagrams in UML 2.2 (Merson, 2009)	12
2.4	Flight Model (Warmer & Kleppe, 2003)	13
2.5	OCL Types package (OMG, 2010d)	16
2.6	OCL Expressions package (OMG, 2010d)	17
2.7	Relationships between QVT metamodels (OMG, 2009)	19
2.8	Imperative OCL Package - Side-effect expressions	20
2.9	Imperative OCL Package - Control and Instantiation constructs	20
2.10	Imperative OCL Package - Additional Facilities	24
2.11	Model levels (OMG, 2010f)	27
2.12	Model transformations	28
2.13	Templates and metamodel. Adapted from (Stahl & Völter, 2006).	30
2.14	API-based generators. Adapted from (Stahl & Völter, 2006).	31
2.15	Orthographic software modeling	33
3.1	KISF Top Level	38
3.2	BusinessService - Realization Structural Class Service	39
3.3	BusinessFacade - Specification Structural Class Service	40
3.4	BusinessLogic - Specification Structural Class Service	40
3.5	Persistent Data Model - Specification Structural Class Type	41
3.6	Presentation Logic - Realization Structural Class Service	41
3.7	Ecore Metamodel (EMF, 2010)	43
3.8	WAKAME Edition Screen	46
4.1	WAKAME Generator Structure	49
4.2	Imperative OCL Compiler Architecture	51
4.3	WAKAME Operational View	51
4.4	AST-based generator's mode of execution. Adapted from (Stahl & Völter, 2006)	53
4.5	WAKAME Business Service - <i>Realization Structural Class Service View</i>	55
4.6	WAKAME Generator - <i>Realization Structural Class Service View</i>	56
5.1	IOCLCompiler - Specification Structural Class Service	57
5.2	IOCLCompiler - Realization Structural Class Service	58

5.3	IOCLEngine - Realization Structural Class Service	59
5.4	IOCLEngine - Specification Structural Class Type	59
5.5	WAKAME Autocomplete	60
5.6	Imperative OCL expressions in Ecore	61
5.7	ANLTRWorks Environment	63
5.8	IOCLAnalyzer - Specification Structural Class Service	64
5.9	KobraAnalyzer - Specification Structural Class Service	64
5.10	IOCLAnalyzer - Specification Structural Class Type	65
5.11	KobraAnalyzer - Specification Structural Class Type	66
5.12	IOCLGenerator Configuration file	66
5.13	CodeGenerator - Specification Structural Class Service	66
5.14	CodeGenerator - Realization Structural Class Service	67
5.15	Visitor - Specification Structural Class Service	68
5.16	JavaHandler - Specification Structural Class Service	68
5.17	<i>Alt</i> Expression Template	70
5.18	<i>Switch</i> Template	70
5.19	Operation Call Unit Test	71
5.20	IOCLEngine Parser Test Report	72
6.1	Experiment process (Wohlin <i>et al.</i> , 2001)	74
6.2	<i>WebAgencyBusinessService</i> - Realization Structural Class Service . . .	79
6.3	<i>WebAgencyBusinessFacade</i> - Specification Operational Service View . .	79
6.4	<i>WebAgencyPersistence</i> - Specification Structural Service	80
6.5	<i>WebAgencyBusinessService</i> - Realization Structural Type	80
6.6	Appraiser - Specification Operational Service View	81
6.7	RequestLoan - Specification	84
6.8	RequestLoan - Realization	84
6.9	Appraiser - Compiled Expressions	85
6.10	AddClient - Pre condition check	86
6.11	Web Agency - <i>Instability</i> metric comparison	87
6.12	Web Agency - <i>Cyclomatic Complexity</i> metric comparison	87
6.13	Web Agency - <i>Maintainability Index</i> metric comparison	87
6.14	CHROME top-level component	90
6.15	QueryProcessor - Realization Structural Class Service	91
6.16	Entailment - Realization Operational View	92
6.17	Fired Rules - Realization Operational View	93

6.18	Fired Rules - Specification	95
6.19	Fired Rules - Realization	96
6.20	CHROME - <i>Instability</i> metric comparison	97
6.21	CHROME - <i>Cyclomatic Complexity</i> metric comparison	98
6.22	CHROME - <i>Maintainability Index</i> metric comparison	98
6.23	CHROME - <i>Maintainability Index</i> by components	99
7.1	Eclipse OCL API (MDT, 2010)	102
7.2	Dresden OCL Packages Structure (Dresden, 2011)	104
8.1	Unsupported part of OCL expressions	109
8.2	Unsupported part of IOCL expressions	109
B.1	<i>WebAgencyBusinessService</i> - Specification Structural Class Service . .	129
B.2	<i>WebAgencyBusinessService</i> - Realization Structural Class Service . . .	130
B.3	<i>WebAgencyBusinessService</i> - Realization Structural Class Type	130
B.4	<i>WebAgencyBusinessFacade</i> - Specification Structural Class Service . .	131
B.5	<i>WebAgencyBusinessLogic</i> - Specification Structural Class Service . . .	131
B.6	<i>WebAgencyBusinessPersistence</i> - Specification Structural Class Service	131
C.1	<i>CHROME</i> - Specification Structural Class Service	137
C.2	<i>CHROME</i> - Specification Structural Class Type	138
C.3	<i>CHROME</i> - Realization Structural Class Service	138
C.4	<i>QueryProcessor</i> - Realization Structural Class Service	139

List of Tables

6.1	Web Agency (Manual) - Unit Tests	83
6.2	Web Agency (Manual) - Metrics	83
6.3	Web Agency (Generated) - Unit Tests	86
6.4	Web Agency - Metrics	86
6.5	Web Agency - Unit tests comparison	88
6.6	CHROME (Manual) - Metrics	94
6.7	CHROME (Generated) - Metrics	97

Listings

2.1	Flight Constraint	14
2.2	Block Expression Syntax	21
2.3	Block Expression Syntax within Switch Expression	21
2.4	Compute Expression Syntax	21
2.5	Imperative Loop Expression Syntax	22
2.6	Instantiation Expression Syntax	22
2.7	Switch Expression Syntax	22
2.8	While Expression Syntax	23
2.9	Assignment Expression Syntax	23
2.10	Variable Initialization Expression Syntax	24
2.11	Unlink Expression Syntax	24
2.12	Unpack Expression Syntax	25
2.13	Try Expression Syntax	25
2.14	Raise Expression Syntax	25
2.15	Log Expression Syntax	26
2.16	Assert Expression Syntax	26

Acronyms

AST	Abstract Syntax Tree
CASE	Computer-aided software engineering
CWM	Common Warehouse Metamodel
EMF	Eclipse Modeling Framework
EMOF	Essential Meta Object Facility
GAE	Google App Engine
GUIPIMUF	GUI PIM UML Framework
IOCL	Imperative Object Constraint Language
KISF	KobrA Information System Framework
KWAF	KobrA Web Application Framework
MDA	Model Driven Architecture
MDE	Model Driven Engineering
MOF	Meta Object Facility
JRE	Java Runtime Environment
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Platform-Independent Model
PSM	Platform-Specific Model
QVT	Query-View-Transformation
RUP	Rational Unified Process
SUM	Single Unified Model
UML	Unified Modeling Language

WAKAME Web Application Kobra Modeling Environment

XMI XML Metadata Interchange

1

Introduction

Software development is a challenging task. It demands several subtasks such as the full understanding of the problem domain; the distillation of overlapping and contradictory requirements; and fabrication of an efficient, cost-effective implementation. All these work needs to be managed to a successful conclusion, all at the lowest possible cost in time and money (Mellor & Balcer, 2002).

One of the approaches that leads towards the rise of development productivity is the software abstraction (Stahl & Völter, 2006), which mainly refers to separation of the essential aspects of the system from the non-essential.

Models are at the next higher layer of abstraction, abstracting away the platform¹ details from the development process. Models and models transformations form the core of Model Driven Engineering (MDE). A desirable goal of MDE is to obtain the complete application source code by model transformations. However, in order to do so, two aspects of the model specification needs to be considered: the structural and the behavioral features. The focus of this dissertation is the **proposal of a tool to automatically generate the behavioral part of the model specification**, which is a necessary step to achieve model executability.

This chapter contextualizes the focus of this dissertation and starts by presenting its motivation in Section 1.1. A brief overview of the proposed solution is presented in Section 1.3, while Section 1.3 describes some related aspects that are not directly addressed by this work. Section 1.5 presents the main contributions and, finally, Section 1.6 outlines the structure of the remainder of this dissertation.

¹The underlying software and hardware that will run the program.

1.1 Motivation

In 2000, Model-Driven Engineering (MDE) was put forward by the Object Management Group (OMG, 2010c) (OMG), a consortium of companies and academic centers, under the initiative known as Model- Driven Architecture (OMG, 2010b) (MDA). MDA defines a set of roles for different models within an MDE project together with principles on how to produce these models from one another. In the MDA vision, software engineering is based on the development of (a) a *Platform-Independent Model* (PIM) which represents the application's requirements, architecture and business objects in a machine-processable notation, plus (b) one or more model transformations from the PIM to multiples *Platform-Specific Models* (PSM) and then from PSMs to code.

Over the last decade, OMG defined a coherent family of standards to support MDA. One of these standards is the Unified Modeling Language (UML). Despite all the benefits UML has to offer, such as its rich set of visual diagrams. UML's notation is not sufficiently detailed to fully determine the behavior of operations and transitions (Heitz *et al.* , 2007). This means that UML models with standard notation cannot completely specify the behavior of a software system so that model transformations cannot generate 100% of the source code.

Action Semantics (AS) are means of addressing this shortcoming. The Action Semantics proposal aims at providing modelers with a complete, software-independent specification for actions in their models. The goal is to make UML modeling executable modeling, i.e. to allow designers to test and verify early and to generate 100% of the code if desired (Sunyé *et al.* , 2001)

The concrete syntax for UML Action Language (OMG, 2010a), currently on its beta version, rely on a syntax that is incoherent with the existing UML Expression language, the Object Constraint Language (OCL) (Haustein & Pleumann, 2004).

Actually, part of the AS specification duplicates functionality that is already covered by the OCL and QVT specification and the use of two different syntaxes may be **inappropriate and confusing**.

Thus, this work investigates the possibility of using the already consolidated QVT standard, specifically the Imperative OCL package (IOCL) defined within the QVT specification to specify the behavior of software systems.

1.2 Problem Statement

Motivated by the scenario presented in the previous Section, the goal of the work described in this dissertation can be stated as:

This work defines the design and implementation of the Imperative OCL compiler tool, aiming at providing an alternative way to system modelers specify and generate the software behavior of a system using for that a standard and consolidated modeling language.

Consequently, it will provide a fundamental part of the tool support to reach the model executability, and it will, along with other related code generators improve the development productivity of the artifacts developed for the domain.

1.3 Overview of the Proposed Solution

In order to accomplish the goal of this dissertation, the IOCL (Imperative OCL) compiler is proposed. This Section presents the context where it is regarded and outlines the proposed solution.

1.3.1 Context

This dissertation is part of the ORCAS² research group, led by professor Jacques Robin, whose primary interest is the development of methodologies, CASE tools and component frameworks to speed-up the engineering of software systems.

This work is part of a long-term large project that bring together several Master and PhD. students. This project called as Web Application Kobra Modeling Environment (WAKAME) investigates open research issues involved in providing component-based, model-driven CASE services accessible via web browsers and deployed on the a computing cloud. It started in 2008 with the MSc. dissertations of Breno Machado (Machado, 2009) and Wesley Marinho (Marinho, 2009) which together investigated the challenge of providing ubiquitous, single user model edition and cloud persistence services for UML/IOCL PIM following the Kobra2 (Atkinson *et al.* , 2008) process.

The Kobra2 process, resulting from the collaboration led by Prof. Colin Atkinson at Universität Mannheim and Prof. Jacques at CIN-UFPE, proposes the first integration in synergy of three complementary software engineering reuses approaches: MDE,

²<http://www.cin.ufpe.br/orcas/>

Component-Based Development and Product Line Engineering. The process advocates orthographic (Atkinson & Stoll, 2008), component-based PIM construction. Within such an orthographic modeling approach, also called multi-view modeling or aspect-oriented modeling by some other authors, software engineers create and revise components through its views, each one focused on a single design concern. They have no direct access to the whole PIM of a component and even less of the component assembly that constitute the whole system. This whole PIM, called the Single Unified Model (SUM) gets created and updated by automatically merging the various partial views.

In an attempt to realize the MDA vision and address the automated application code generation, the Kobra metamodel merges the Imperative OCL (IOCL) package of QVT. IOCL extends the OCL language and integrates the core constructs of imperative and OO programming. In Kobra2, it is used to specify in the **bodies, pre-conditions and post-conditions** of UML operations. In all these contexts, IOCL expressions constitute a formal input to behavioral and design by contract tests (via pre and post conditions) code generation.

1.3.2 Outline of the Proposal

As mentioned, the present work is part of the WAKAME project. Currently, the development focus of WAKAME is its extension along two axes. The first is revision control allowing the secure concurrent model edition by multiple users collaborating on an MDE project. The second is the automatic generation of full web application code from UML/IOCL Kobra2 PIMs edited using WAKAME. Together these two extensions are needed to transform WAKAME from a simple visual editor of Kobra2 PIM, to a full-fledged collaborative CASE tool delivering productivity gains for real-life component-based MDE projects.

The code generation axis is divided into three sets of aspects. The first set distinguish between structural code generator and deployment code. The second set distinguish between the behavioral code generator. The third set distinguish between production code of the presentation GUI layer.

This dissertation deals with the behavioral code generator aspect as it proposes a compiler aiming at providing system modelers the ability to specify and generate the software behavior based on the QVT standard through the use of the consolidated Imperative OCL modeling language.

The proposed solution consists of modules for the front-end and back-end of the IOCL expressions compiler, including components for the syntactical and type checking

analysis and the code generator. The goal also is to create a modular implementation that makes easier further extensions to different metamodels and target languages. Also it is included in this project, studies with the purpose to evaluate the impacts of applying the tool to perform the automatic generation of method bodies.

1.4 Out of the Scope

As the proposed compiler uses Imperative OCL language as input language, which inherits all constructors and standard functions of OCL and also adds several others, a set of aspects of the OCL and IOCL specification were left out of this project scope. However this limitation do not discard future enhancements to answer more efficiently to its purpose. Meanwhile, the aspects not directly addressed by this work are listed in the following:

- **Ecore/UML Modeling Support.** Event though system specification with Ecore and UML metamodels are important to the purpose of this work. The implemented compiler only includes the support for Kobra2 models;
- **Incomplete OCL and IOCL Support.** The OCL expressions: *IfExp*, *LetExp*, *OclMessageExp*, *TupleLiteralExp* and the IOCL expressions: *AssertExp*, *CollectorExp*, *UnlinkExp*, *UnpackExp* and *TupleExp* are not supported;
- **Round trip engineering.** The automatic synchronization between different levels programming languages abstractions is an important feature for the tool adoption but currently this requirement is out of the scope of this work.

1.5 Statement of the Contribution

As a result of the work presented in this dissertation, a list of contributions may be enumerated:

- Proposal of a new usage of Imperative OCL expressions. The work proposes its utilization during the behavioral model specification at PIM level.
- Design and implementation of IOCL compiler. A tool that automatically generates platform-specific code from IOCL language.
- Two different evaluations for verifying the the tool helpfulness to a MDA process.

1.6 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

- Chapter 2 contains a comprehensive revision of the basic concepts necessities for the accomplishment of this work, with topics related to Model Driven Engineering, Model Driven Architecture and OMG standards.
- Chapter 3 presents the Kobra engineering process, KISF framework and WAKAME tool.
- Chapter 4 details the WAKAME Code Generator project, topics such the structure of its components and the integration of Code Generator project to the WAKAME tool are covered in this chapter.
- Chapter 5, the PIM of the Imperative OCL compiler is described using the Kobra methodology. This chapter also details its architecture, the implementations details and the chosen technologies.
- Chapter 6 reports the IOCL compiler's evaluations. Also in this Chapter there is a description of each experiment, the expected goals, the methodology adopted, and the findings.
- Chapter 7 focuses on the works related to this project.
- Chapter 8 concludes this dissertation by summarizing the contributions of this work, as well the limitations and proposing future enhancements to the solution, along with some concluding remarks.

2

Concepts and Terminology

In this chapter we describe the concepts of software engineering we used to develop the proposed application. We also define the terminology used in the rest of this thesis.

2.1 Model-Driven Engineering

One of the key reasons behind the success of the Object-Oriented (OO) paradigm since the mid nineties is the co-emergence of OO software modeling languages together with expressive and efficient imperative OO programming languages.

Since then, models have been used for software development in basically two distinct approaches (Stahl & Völter, 2006): the model-based approach, where models are used just as documentation and the model-driven approach, where they are used as input for automated generation of artifacts such as other models, code and documentation.

Traditional OO methods such as the Rational Unified Process (RUP) (Kruchten, 2003) are model-based, but not model-driven. In principle, the careful building of models as documentation that they advocate leads to improved software quality. It allows to separate platform-independent concerns such as business modeling, requirements elicitation and architecture design from platform-specific concerns such as programming, testing and deployment. This separation allows a professional to specialize in specific tasks. It also reduces their cognitive overload, a lead cause of bad design and software defects. However, in a model-based approach, all these artifacts need to be keep in synch manually during the software lifecycle. Under the harsh and often unrealistic deadlines that is the rule rather than the exception in the software industry, this manual synchronization maintenance incurs a generally prohibitive time and cost overhead. This overhead led model-based method to become less popular. They are progressively being replaced by two alternatives that aim to reduce this overhead following two largely opposite

approaches: agile methods and MDE. The first reduces this overhead by making models less numerous, precise, persistent, machine processable, so that synching them with code becomes superfluous. In contrast, MDE reduces this overhead by making models even more numerous, precise, persistent and machine processable so that synching them with code and among themselves can be automated.

In agile method, models become quickly drawn, disposable notes towards the first version of the code (or a major update involving some redesign). Extreme version of this approach do away with models entirely. By drastically reducing or suppressing modeling stages, agile methods gives up on the separation of concerns that these stages brought to software engineering at the first place. They have nevertheless become very popular for small scale and single platform projects that still constitute the most common ones in the software industry.

2.2 Model-Driven Architecture and OMG standards

2.2.1 Model-Driven Architecture

The Model-Driven Architecture (MDA) (OMG, 2010b), a registered trademark of the Object Management Group OMG (OMG, 2010c), refers to model-driven engineering approach based on the use of OMG's modeling standard languages.

According to OMG's directive, the two primary motivations for MDA are *interoperability* (independency from manufactures through standardization) and *portability* (platform independence) of software systems. In addition OMG postulates the system functionality specification should be separated from the implementation of its functionality. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications (Stahl & Völter, 2006).

The essence of MDA is the clear distinction of software aspects into different levels of abstractions. The fundamental long-term vision of MDA 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 (PSM), which in turn are translated to source code either manually or by model transformations. The MDA 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.

MDA 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 these goals, MDA 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.

In MDA, the software development starts with the definition of the PIM of the application. The PIM describes the system but the model remains completely independent of the later implementation on target platform such as Java or .NET. This step is generally modeled using UML, possibly adapted via profiles.

Followed by the PIM definition, the PSM is created. The PSM is usually created automatically from PIM via model transformation. In this process it incorporates concepts specifics to the target platform and making them viable to be mapped to code. This way, the source code can also be generated with another tool-supported transformation based on the PSM.

Finally, it is important to emphasize MDA solves the problem of platform fragmentation by providing a proper infrastructure for the automatic mapping of multiples PSM and latter code from each PIM definition.

2.2.2 OMG Standards

In this section we present the key OMG standards used for structuring MDA based systems. The OCL and Imperative OCL (sub package of QVT) specifications are deeply focused due their importance to this project scope.

UML Infrastructure

With UML 2.0's, one of the main goals of OMG was to align MOF, UML and OCL in a core metamodel called the UML Infra-Structure. This common metamodel contains elements designed to be reused during the definition of the UML metamodel, as well as other architecturally related metamodels, such as the Meta Object Facility (MOF) and the Common Warehouse Metamodel (CWM).

The Infrastructure consists of two packages: *Core* and *Profiles*. The Core is a complete

metamodel, particularly designed for high reusability, where other metamodels at the same metalevel either import or specialize its specified metaclasses, benefiting from the abstract syntax and semantics that have already been defined. The Profiles package defines the mechanisms used to tailor existing metamodels towards specific platforms, such as .NET or Java. Profiles have been aligned with the extension mechanism offered by MOF, but provide a more light-weight approach once they not change the related metamodel.

The figure 2.1 depicts the internal structure of the Core package. The package is divided into *PrimitiveTypes*, *Abstractions*, *Basic*, and *Constructs*. Some of these are even further divided into more fine-grained packages to make it possible to choose just the relevant parts when defining a new metamodel.

The *PrimitiveTypes* package contains a few predefined types that are commonly used when metamodeling such as Integer, Boolean, String and UnlimitedNatural types. *Abstraction* contains contains abstract metaclasses that are intended to be further specialized or that are expected to be commonly reused by many metamodels. *Constructs*, on the other hand, contains concrete metaclasses that lend themselves primarily to object-oriented modeling (with the class and namespace diagrams for example) and the *Basic* represents a few constructs that are used as the basis for UML, MOF, and other metamodels based on the Infrastructure Library (OMG, 2010f).

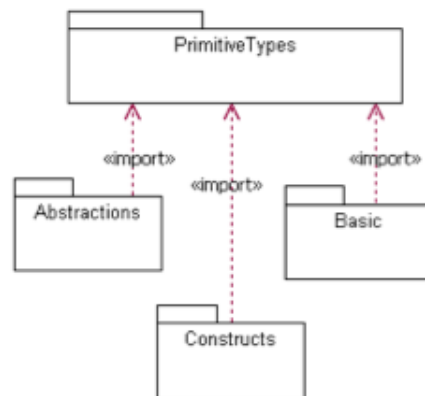


Figure 2.1: UML Infrastructure - The *Core* Package (OMG, 2010f)

MOF, CMOF and EMOF

The Meta Object Facility (MOF) 2.0 is a central key for MDA. Built on top of the UML Infrastructure, MOF provides a metamodeling framework, and a set of metamodeling services to enable the development of model driven systems.

MOF can be used to define a family of metamodels using simple class modeling concepts. The MOF itself is defined using its constructs, as well as other models and other metamodels (such as UML, CWM etc.).

The MOF model is made up of two main packages, Essential MOF (EMOF) and Complete MOF (CMOF). The MOF Model also includes additional capabilities defined in separate packages including support for, identifiers, additional primitive types, reflection, and simple extensibility through name-value pairs. The figure 2.2 illustrates the MOF package structure.

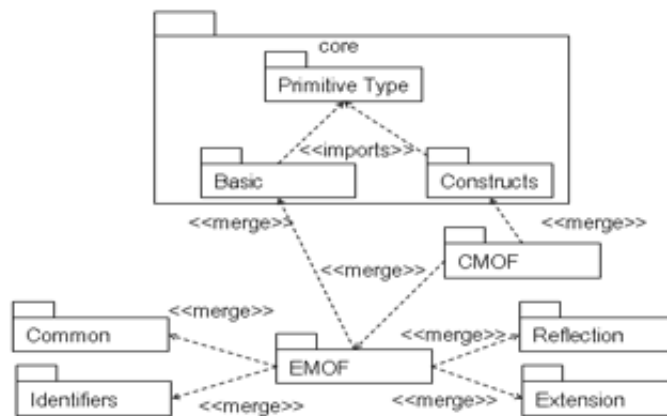


Figure 2.2: MOF Package Structure (OMG, 2010e)

The EMOF model provides the minimal set of elements required to model object-oriented systems. EMOF reuses the Basic package from UML InfrastructureLibrary as its metamodel structure without any extensions, although it does introduce some constraints.

The CMOF Model is the metamodel used to specify other metamodels such as UML2. It is built from EMOF and the *Core::Constructs* of UML 2. The CMOF Model package does not define any classes of its own. Rather, it merges packages with its extensions that together define basic metamodeling capabilities (OMG, 2010e).

UML Superstructure

The Unified Modeling Language is a language used to specify, construct, visualize, and document models of software systems (Weilkiens & Oestereich, 2006). UML is specified with MOF and OCL and is composed of several graphical diagrams which allows engineers to design different specificities of the system. These diagrams represent the structural and behavioral aspects of the system.

The figure 2.3 depicts the UML diagrams. They are divided into two main categories: *Structure* and *Behavior* diagrams. Structural diagrams emphasizes the static structure of

the system using objects, attributes, operations and relationships. The Behavior diagrams emphasizes the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects

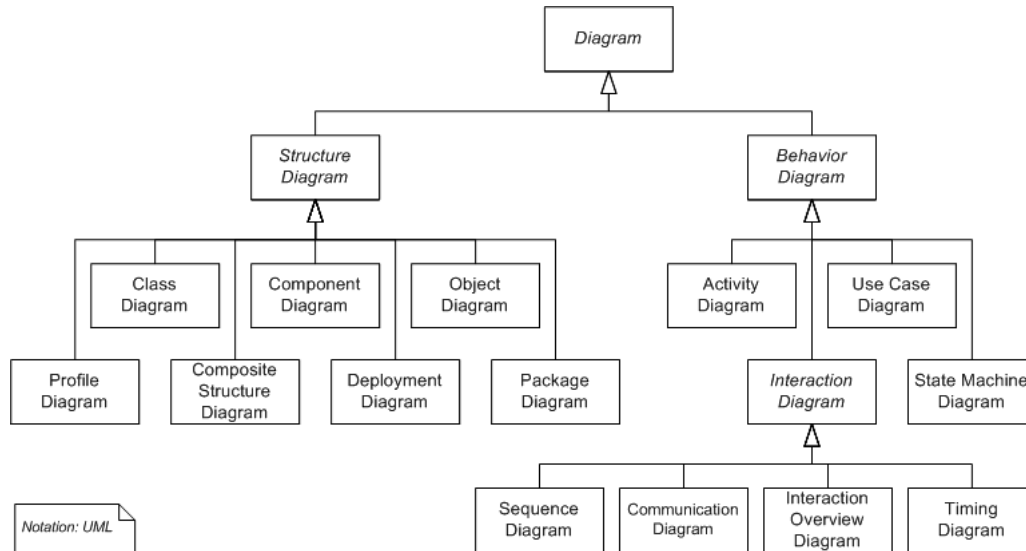


Figure 2.3: Hierarchy of diagrams in UML 2.2 (Merson, 2009)

OCL

The Object Constraint Language (OCL) is a declarative language used add vital information to models in order to make them precise enough to serve as input to automated model checking and code generation.

Next sections we present the characteristics of OCL, how it can be used during the modeling and finally we explain the abstract syntax of the OCL, which includes the types and expressions packages.

OCL characteristics

A distinguishing feature of OCL is that it is a declarative language. This means that an OCL expression simply states what should be done, but not how. This characteristic brings several advantages for the modeler. An example is that the modeler can make decisions at a high level of abstraction, without detail how something should be calculated (Warmer & Kleppe, 2003). OCL can be used to express preconditions, postconditions, invariants, results of method calls (body expressions) and they can be also be used to define a derived attribute or association.

Additionally, OCL is a strongly typed language and this means each expression has a type. This characteristic enables OCL to be checked during modeling and before the execution. Thus, simple errors such as an assignment of a value incompatible to a variable type can be removed in early stages of development.

The need for constraints in PIM

In general, UML diagrams have a natural tendency to be incomplete. The concrete visual syntax of UML only allows the expression of visibility, type and multiplicity constraints involving two model elements of meta-class classifier, property, relationship, operation or parameter. Precise models require the expression of constraints involving any number of model elements and combining several aspects (e.g., type of one element, visibility of another and multiplicity of a third).

To illustrate the need for constraints, consider the model shown in figure 2.4 (this example is part of (Warmer & Kleppe, 2003)). There is an association between class Flight and class Person, indicating that a certain group of persons are the passengers on a flight. The multiplicity (0 .. *) on the Person class indicates the number of passengers is unlimited. In real world scenario this is not true. The number of passengers on a flight is always limited to the number of seats available on the airplane associated with the flight.

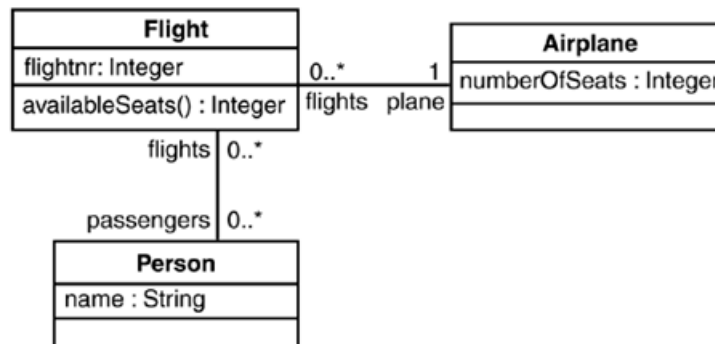


Figure 2.4: Flight Model (Warmer & Kleppe, 2003)

Such kinds of restrictions cannot be correctly specified in the diagram without the use of OCL in the model. A precise way to describe this constraint would be as listed in 2.1:

Listing 2.1: Flight Constraint

```
context Flight
inv: self.passengers->size() <= plane.numberOfSeats
```

Expressions, like the above, written in a precise language like OCL offer a number of benefits, such as: they are never ambiguous; they can be automatically checked by tools to ensure they are consistent with the model elements and perhaps, the most important is they can be automatically processed by a compiler and transformed to source code.

Constraint Definition

OCL expressions consist of several rules expressions. There are different contexts these rules can refer to. In this section we present them grouped by its respective context.

Classes

- *Invariant* - Invariants are boolean rules that must be true for each instance of the classifier at any moment of time.
- *Definition* - A definition is a construction that creates a new attribute or a new operation.

Attributes and Association Ends

- *Initialization* - Initialization expressions define the initial value for the attributes or association end it refers to. Also, the type of the expression acting as the initial value must be compatible to the type defined in the model.
- *Derivation* - Derivation rules specify the value of a derived element. An OCL expression acting as a derived value of an attribute end must conform to the type of the attribute.

Operations

- *Precondition* - A precondition is a Boolean expression that must be true whenever the operation starts executing, but only for the instance that will execute the operation.

- *Postcondition* - A postcondition is a Boolean expression that must be true at the moment the operation stops executing, but only for the instance that just executed the operation.

When defining the postconditions expressions, the modeler can make use of the “@pre” keyword; this turns possible to make reference to values at the precondition time. The variable *result* may also be used as it refers to the return value of the operation if there is any.

- *Body expression* - A body expression defines the body of an read-only operation. As OCL is a side-effect free language OCL expression cannot be used to define the body of a read/write operation. Body expression type, as occurs in others rules, must also be conformant to the type of the operation defined in the model.

OCL Abstract Syntax

The abstract syntax of OCL is divided into two different packages:

- The *Type* package describes the concepts that define the type system of OCL. It shows which types are predefined in OCL and which types are deduced from the UML models.
- The *Expressions* package describes the structure of all OCL expressions.

We will not detail each expression or type defined in the abstract syntax of OCL. Instead we will focus on the expressions currently supported by the IOCL compiler.

The Type Package

OCL is a typed language. Each OCL expression has a type which is either explicitly declared or can be statically derived. The figure 2.5 shows the defined OCL types.

PrimitiveTypes

The primitive types defined in the OCL standard library are Integer, Real, String and Boolean. They are all instance of the metaclass Primitive from the UML core package.

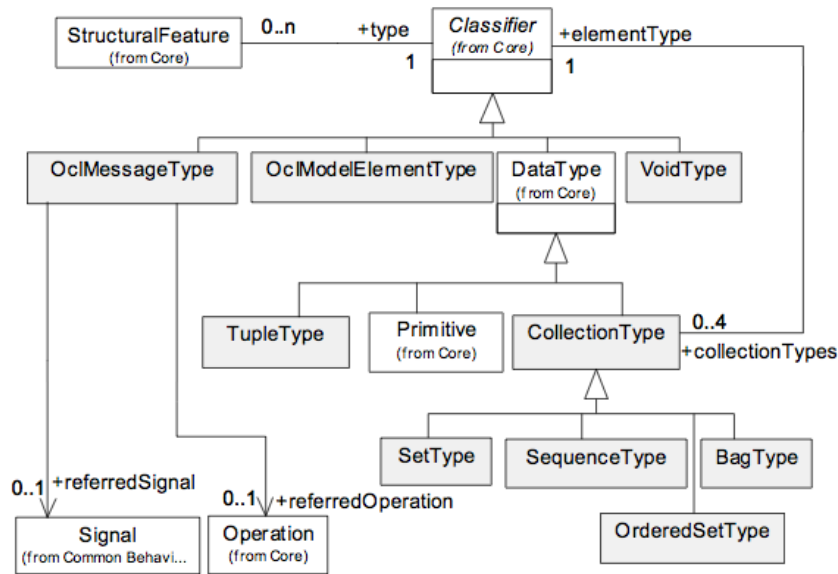


Figure 2.5: OCL Types package (OMG, 2010d)

CollectionType

Collection play an important role in OCL expressions. There are four different collection types in OCL 2.0 specification: *SetType*, *OrderedSetType*, *SequenceType*, and *BagType*. Set is equivalent to the mathematical definition of set (it does not contain duplicate elements), OrderedSet shares the same characteristics of Set by with notion of elements order. Bags differs from Set because their elements can appear more than once and Sequences are like Bags in which elements are ordered. To illustrate the use of collections, consider a user writing expressions that navigates through a simple association defined in the model. In this case, the result type of that expression would be a Set collection type. Now if the mentioned association were adorned with the ordered the result type would be an Ordered Set collection type.

Additionally, OCL defines a large number of operations to enable the modeler to manipulate collections¹. All the operations are invoked by the arrow (->) symbol. It is important to emphasize that all expressions defined using these operations never change the collections. They may even result in a collection, but rather than change the original collection, they project the result in a new one.

¹A complete list of the operations can be found in (OMG, 2010d).

The Expressions Package

The figure 2.6 depicts the Expressions package of OCL. Next, we detail the most relevant expressions accepted by the current implementation of the IOCL compiler.

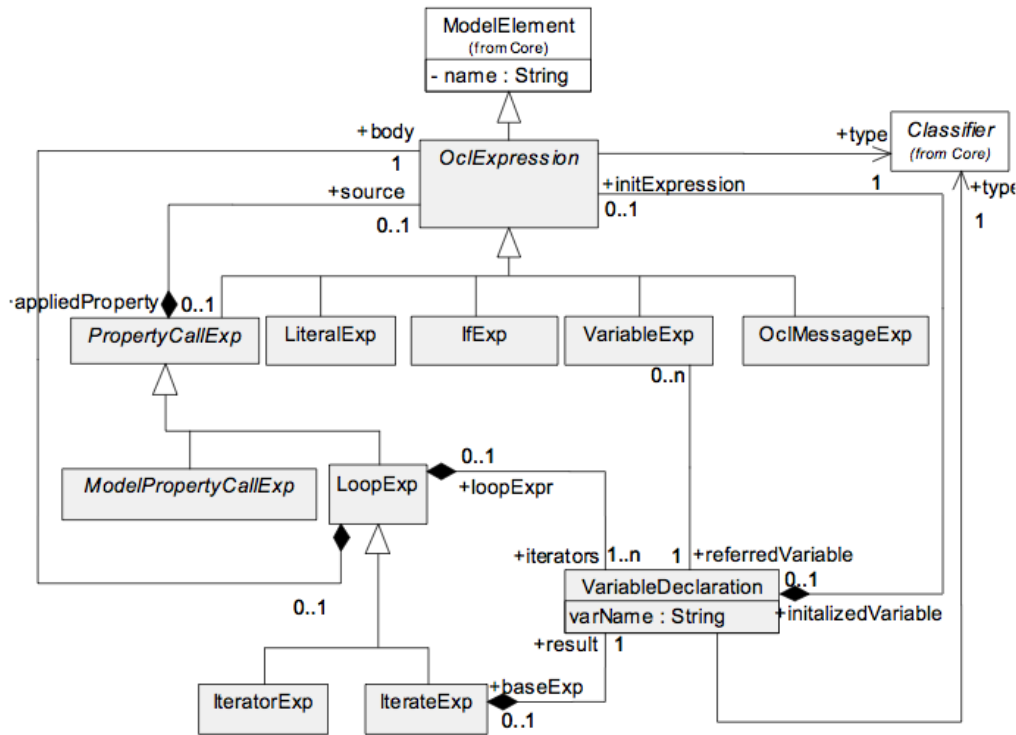


Figure 2.6: OCL Expressions package (OMG, 2010d)

LiteralExp

LiteralExp is an expression with no arguments producing a value. In general the result value is identical with the expression symbol. This includes things like the integer 1 or literal strings like ‘this is a string’.

ModelPropertyCallExp

ModelPropertyCallExp is an abstract class and it can refer to any subtype of *Feature* as defined in UML. There are three classes that specialize ModelPropertyCallExp in the OCL specification: *AttributeCallExp*, *NavigationCallExp* and *OperationCallExp*.

An `AttributeCallExp` is a reference to an `Attribute` of a `Classifier` defined in a UML model. The listing 2.1 illustrates its usage. The *plane.numberOfSeats* expression is an `AttributeCallExp` because *numberOfSeats* is an attribute of `Airplane`.

A `NavigationCallExp` is a reference to an `AssociationEnd` or an `AssociationClass` defined in a UML model. The *self.passengers* expression also listed in 2.1 is an example of a `NavigationCallExp` because *passengers* is an association of `Flight` classifier.

A `OperationCallExp` refers to an operation defined in a `Classifier`. The expression may contain a list of argument expressions if the operation is defined to have parameters. The expression *self.passengers->size()* in an example of an operation call applied to the `Set` type ².

There are several predefined operations available for OCL types. They are specified by the OCL Standard Library (OMG, 2010d). The OCL Library plays a crucial role for achieving models sufficiently precise to serve as input to code generation from PIM. This stems from the fact that substantial part of the application code are calls to operations from high-level API, frameworks or libraries and since UML has no library, the only way to be able to pursue the model executability is through the OCL.

IterateExp and IteratorExp

An `IterateExp` is an expression which evaluates its *body* expression for each element of a collection. It acts as a loop construct that iterates over the elements of its source collection and results in a value. An `iterate` expression evaluates its body expression for each element of its source collection. The `IteratorExp` are special pre defined types of `IterateExp`. The OCL standard library defines several `Iterators` depending on the type of the source collection. This includes *select*, *collect*, *reject*, *forAll*, *exists*, etc.

QVT and Imperative OCL

Query/View/Transformations (QVT) defines a standard model transformation language. QVT is a hybrid declarative/imperative language. The declarative part is structured in two packages: *Core* and *Relations*. *Relations* supports complex pattern matching and object template creation while the *Core* language, defined using minimal extensions of

²By default, navigation will result in a `Set`

EMOF and *OCL*, only supports pattern matching over a set of variables by evaluating conditions over those variables against a set of models. In spite of the different level of abstraction, both languages embody the same semantics and they are equally powerful.

In addition to the declarative languages, there are two mechanisms for invoking imperative implementations of transformations from Relations or Core: the QVT internal language, *Operational Mappings*, and the external call mechanism *Black-box* MOF operation implementations. The Operational Mappings language specify a standard way of providing imperative implementations. It defines a OCL extension with side effects, named Imperative OCL, that allow a more procedural style, and a concrete syntax that looks familiar for imperative programmers. Black box implementations is also another important part of the specification because it makes possible for integrating transformations expressed in other languages to QVT libraries. This allows transformations leverage powerful libraries in other languages than IOCL library. The figure 2.7 depicts the relationships between the metamodels.



Figure 2.7: Relationships between QVT metamodels (OMG, 2009)

Imperative OCL

As already stated, the imperative OCL language is defined inside the Operational Mappings Language, in QVT (OMG, 2009) specification. It extends the OCL language with imperative expressions and algorithm constructs such as variable assignments, block, loops, etc. It reuses OCL for OO structure navigation and functional expressions that can be nested in the imperative expressions. It also extends the type hierarchy of OCL with additional facilities such as dictionaries (hash tables), as well as the OCL standard library with new operations on both OCL types and IOCL types.

The metaclasses defined in the Imperative OCL specification are divided in two groups. The first concerns the imperative expressions and second deals with the extensions made to the OCL types

The figure 2.8 depicts all the class hierarchy defined in the package. The abstract class

ImperativeExpression specializes the *OclExpression* to serve as the base for the definition of all other side-effect expressions.

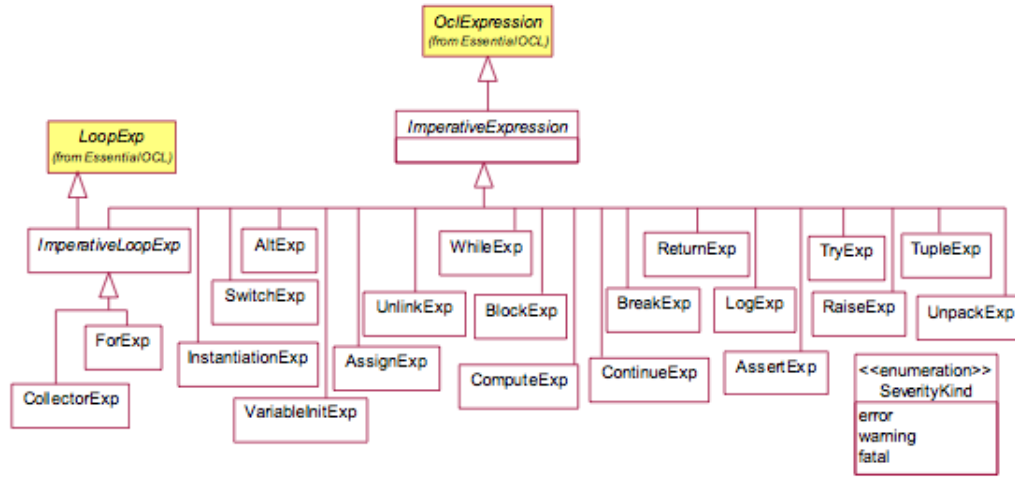


Figure 2.8: Imperative OCL Package - Side-effect expressions

The figure 2.9 details the expressions responsible for the program flow and instantiations facilities. In the following subsections we will explain each one of them.

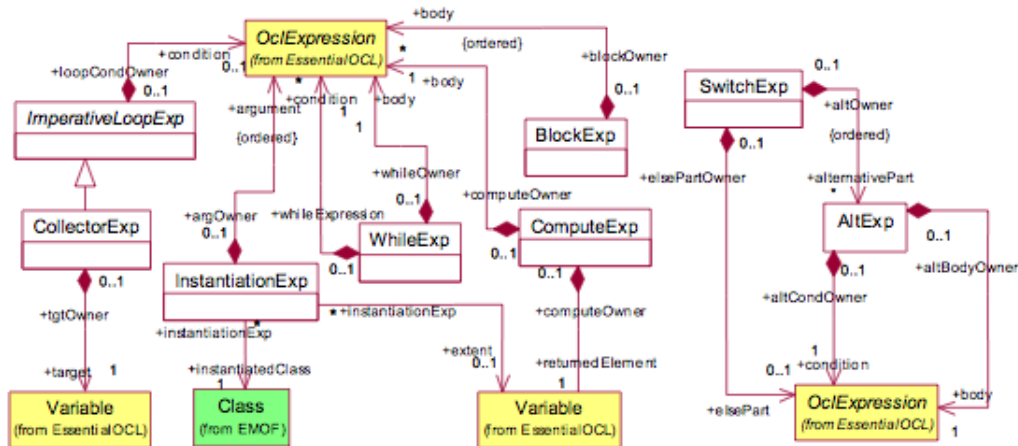


Figure 2.9: Imperative OCL Package - Control and Instantiation constructs

Block Expressions

A block expression (*BlockExp*) is a sequence of other IOCL expressions. The expressions defined in the body are executed sequentially in the order they are defined until

the end of the block. However this execution can be interrupted by a *break*, a *continue* or a *return* expression.

The block expression notation uses the *do* keyword followed by braces to delimit the sequence of expressions. The listing 2.2 shows an example.

Listing 2.2: Block Expression Syntax

```
do {  
  ...  
}
```

The *do* keyword can be sometimes skipped if the block expression is defined within the *switch*, *compute* and *for* expressions. The listing 2.3 shows an example of the *do* keyword being skipped by the use of a *switch* expression

Listing 2.3: Block Expression Syntax within Switch Expression

```
if (something) then {  
  ...  
}
```

Compute Expressions

Compute expressions are used to define a variable, possibly initializing it and a body to update its value. The result of a compute expression is the variable at the end of execution of the body.

The notation used for compute expressions is the *compute* keyword followed by the variable definition in parenthesis, followed by the body expression. The listing 2.4 shows an example of the concrete syntax.

Listing 2.4: Compute Expression Syntax

```
compute (x:String = ''){  
  ...  
}
```

Imperative Loop Expressions

IOCL defines four types of Imperative Loop expressions: *forEach*, *forOne*, *collectedSelect* and *collectedSelectedOne*.

The loop is always applied to a collection. It declares iterators, a body and a condition. The execution may also be interrupted by a *break*, *continue* or *return* expression. The

listing 2.5 shows an example of the *forEach* construct being applied to a source collection. The expression iterates through all elements of the set collection and selects the even ones.

Listing 2.5: Imperative Loop Expression Syntax

```
Set{1, 2, 3, 4}->forEach(i | i.mod(2) = 0) { ... }
```

Instantiation Expressions

Instantiation expressions create an instance of the class, by invoking its class constructor and returns the created object. The listing 2.6 shows an example of its concrete syntax. The expression invokes the `Person::Person(String, String)` constructor and returns an instance class.

Listing 2.6: Instantiation Expression Syntax

```
person := new Person('firstName', 'lastName')
```

Switch Expressions

Switch expressions are used to express alternatives (alternative expressions) that depend on conditions to evaluate. The behavior is similar to an OCL *if* expression but with three main differences: a) interrupt expressions (break, continue raise and return expressions) can be added to its alternatives b) the *else* part is not mandatory as in OCL specification c) it allows calls of operations with side-effects.

There are two available notation styles: the first may use the if-else and the latter the switch keyword. Both notations are showed in the listing 2.7.

Listing 2.7: Switch Expression Syntax

```
if (condition1) exp1
elif (condition2) exp2
else expN
endif

switch {
(cond1)? exp1;
(cond2)? exp2;
else?: expn;
}
```

While Expressions

While expressions are control expressions used to iterate on a block until a condition becomes false. *Break* and *Continue* expressions are two commands that can also be used within the block to alter the normal program flow. When a break command is executed it provokes the termination of the while expression, a continue expressions provokes the execution of the next iteration without executing the remaining instructions in the block. The listing 2.8 shows an example of the while concrete syntax.

Listing 2.8: While Expression Syntax

```
while (not node.isFinal()) {  
  ...  
}  
  
while (x:MyClass := self.getFirstItem(); x <> null) {  
  ...  
}
```

Additional Facilities

Imperative OCL also provides additional facilities to be used to express the program logic. The figure 2.10 depicts additional expressions. They provide constructs for variables initialization, assignments, exception managements and logging.

Assignment Expressions

Assignment expressions represent the assignment of a variable or a Property. There are two possible semantics of its execution depending whether the variable or property is monovalued or not. If the variable is monovalued, the effect is to reset the variable value with the new value. If the variable is multivalued, the effect is to reset the value or append it depending on the *isReset* property set for expression. This property is configured according the operator defined for the concrete syntax, the `:=` symbol set the *isReset* property to true and `+=` otherwise. The listing 2.9 show examples of its usage.

Listing 2.9: Assignment Expression Syntax

```
mymultivaluedproperty += { ... };  
mysimpleproperty := 'Hello World!';
```

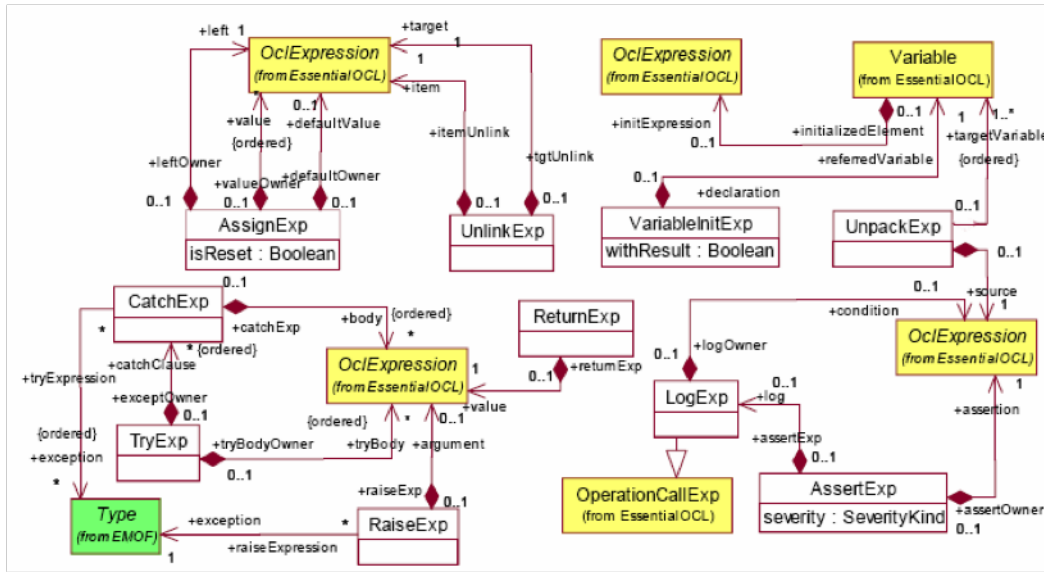


Figure 2.10: Imperative OCL Package - Additional Facilities

Variable Initialization Expressions

Variable initialization expressions represent the declaration of a variable with an optional initialization value. The variable's type can be omitted as long it can be derived from the initialization expression. A variable may also not specify the initialization value. In this case the default value is assumed (an empty collection for a collection variable, zero for numeric types, false for boolean types, empty string for string typed variables and null for all other elements). The listing 2.10 shows an example of its concrete syntax.

Listing 2.10: Variable Initialization Expression Syntax

```
var x:String := 'abracadabra';
```

Unlink and Unpack Expressions

Unlink expressions represent the explicit removal of a value from a multivalued property link. The notation used a call to the unlink operation and is exemplified in the listing 2.11.

Listing 2.11: Unlink Expression Syntax

```
feature.unlink(myattribute);
```

The unpack expression is used to unpack an ordered tuple by assigning a list of variables with the new value of the tuple elements. The listing 2.12 shows an example of the notation.

Listing 2.12: Unpack Expression Syntax

```
var (x, y, z) := self.foo();  
— assuming foo() returns a tuple of three elements  
  
var (x:X, y:Y, z:Z) := self.foo();  
— shorthand in which variables are both declared and assigned
```

Try, Catch and Raise Expressions

Try expressions are used to define exception aware blocks. Any exception that may occur during the block execution can be handled properly a list of catch clauses. Therefore, a catch expression represents the coded to be executed when a exception matching is fired during try block. The listing 2.13 exemplifies scenario where the exception1 is handled by expression2 block in the case it is fired during the expression1 execution.

Listing 2.13: Try Expression Syntax

```
try { expression1 } except (exception1) { expression2 }
```

Raises expression are used to produce an exception. The notation used the raise keyword with the exception name as body. The exceptions can also be provided as a simple string. An example is shown in the listing 2.14.

Listing 2.14: Raise Expression Syntax

```
raise 'Problem Here';
```

Log Expressions

A log expression is an expression used to print a log record to the environment. It is often used for debug. A log may only be sent when a condition holds. A log expression receives three arguments, the first is the message, the second is the model object to be printed and the third gives the level of severity of the message. Despite the three arguments, only the first is mandatory. The notation used the *log* keyword and is exemplified in the listing 2.15.

Listing 2.15: Log Expression Syntax

```
log ('property bob is null',result) when result.bob = null;
```

Assert Expressions

A assert expression is an expression that checks whether a condition holds. If the assertion fails an error message is generated - possibly accompanied with a log record. If the assertion fails with fatal severity, the execution terminates with the exception *AssertionFailed*. In all other cases the expression returns null. The notation uses the assert keyword. It may also be followed by the severity indication (warning or fatal identifiers) and the log expression introduced by the with keyword as shown in the listing 2.16.

Listing 2.16: Assert Expression Syntax

```
assert result.bob <> null with log('non null 'bob' expected', result);
```

XML Metadata Interchange

The XML Metadata Interchange (XMI) specification (OMG, 2010g) defines a bi-directional serialization/deserialization standard between models whose abstract syntax is defined by a MOF metamodel and a textual document in XML syntax. Roughly speaking, (1) each model element is mapped to a XML element in the document. That way, each XML element is tagged by the name of the element's metaclass, each property in a model element is mapped to a XML attribute in the opening tag of the XML element representing the model element and each reference from a source model element to a target model element is mapped onto an XML reference (href) element.

2.2.3 Metamodeling

Metamodels describe the possible structure of models. In an abstract way, it defines the constructs of a modeling language and their relationships, as well as constraints and modeling rules. In order to define a metamodel, an metamodel language is required that in turn is described by a meta meta model. OMG defines UML and the related standards in a four metalevels architecture as can be observed in figure 2.11.

In the OMG stack, MOF is at the M3 level and it is used to define modeling languages such as for example UML or CWM (M2 level). The UML models are located at the M1 level and the instances of this model, usually created at program runtime, are at level M0.

The idea behind this is to not tie UML as the only modeling language, but enable additional domain specific and possibly standardized languages to be defined based on MOF.

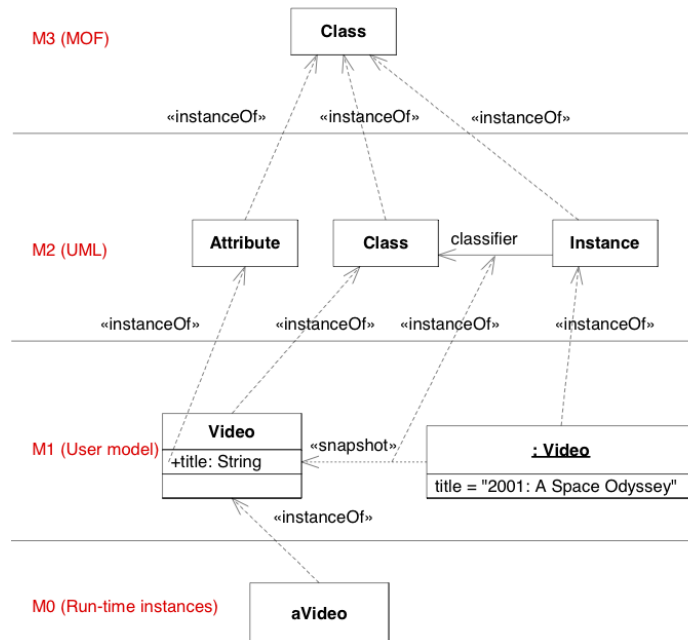


Figure 2.11: Model levels (OMG, 2010f)

2.3 Model Transformations

Models transformations map models to the respective next level representation, be it other model or even source code (Stahl & Völter, 2006). The MDA vision pursues the goal of obtaining the application source code via several subsequent transformations. Code generation from PIM is thus central to this vision.

All the productivity gain promoted by MDA is achieved thanks to model transformations. The figure 2.12 depicts one of the transformation paths between the MDA models.

In spite of the figure 2.12 details only one classification of transformation, in the case a *vertical* transformation, where level of abstraction of the model is changed, a number of other transformations are possible such as PIM to PIM, PSM to PIM, PSM to PSM, etc. According to MDA specification (OMG, 2003) it is also possible to generate code from PIM. This approach is usually the preferred approach due its simplicity (Stahl & Völter, 2006). More detailed studies about transformations can be found at (Metzger,

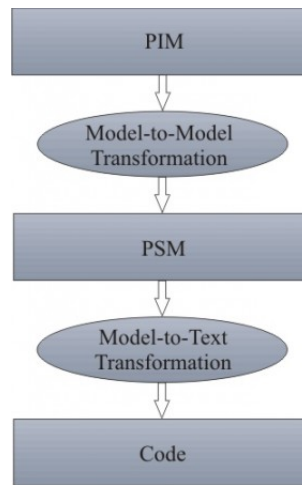


Figure 2.12: Model transformations

2005) (Czarnecki & Helsen, 2003) (Mens *et al.* , 2004).

It is the transformation that maps the concepts defined at one level to its next level of representation. During PIM to PSM transformations, for example, additional constructs and elements related to a determined platform are added to model. This kind of information is encoded into the transformations engines, which makes it possible of having multiple PSMs generated from a single PIM.

2.3.1 Code Generation

Code generation tools are used to increase the software development productivity. When generators are not bound to models, this objective is achieved through automatic generation of repetitive source code parts and generally these solutions are limited to common problems in simple domains (Herrington, 2003).

MDA pursues the goal of full application code generation from PIM, which includes all the aspects of the system, such as structural and behavioral codes, configuration files and tests.

Transformations when applied to PIM enables to capture the knowledge the specialists have on the target platform, making it possible to have high quality code generation in a standardized way (Kleppe *et al.* , 2003). Also, once the platform code is independent of the business logic, it makes easier the transitions to newer and potentially better platforms.

There are two well-known approaches for generating code from PIM and both are stated in (OMG, 2003). First approach is the multi-staged transformation. In this case there are two sequential transformations to produce the source code: PIM to PSM and

latter PSM to code. The second approach is the single stage PIM to code transformation.

The first option makes the generator more modular by dividing the transformation into three different phases (Stahl & Völter, 2006):

- First, the source model (concrete syntax) is parsed and an abstract representation of the model is created in the generator, typically in the form of an object structure, for example through instantiation of the metamodel classes;
- The parsed model is transformed into the target model, working only on the object graph representation;
- Finally, the target model is rendered into the concrete syntax of the target language.

The first step of this approach is usually achieved through the use of a model-to-model(M2M) technology. The ATLAS Transformation Language (ATL) (ATL, 2009) is one of these alternatives. ATL is a full implementation of a rule-based and declarative-procedural language used to define model transformations. ATL was proposed as a response to the OMG QVT RFP Section 2.2.2 but it was not adopted as a standard, instead it became a project of the Eclipse Foundation.

The transformation definition in ATL is based on the source and target metamodels and it is defined by a set of rules that specify how a given source model produce a target model. In order to use ATL, both models need to be in agreement to its respective metamodels, which are associated with the transformation.

The second option translates the PIM directly to source code. In this case, the transformation is more rigid because it comprises all steps in only a single one.

In 2008, the OMG released the Request for Proposal (RFP) to MOF Model to Text transformations (MOFM2T, 2010) to address the problem of how to translate a models into text artifacts. The most accepted submission to this RFP by the MDA community is the MOFScript (MOFScript, 2010), that will be detailed in 3.3.2.

Code Generation Techniques

We have identified several approaches for dealing with code generation (Stahl & Völter, 2006). In this section we will detail some of them.

Templates and Filtering

This technique describes the simplest case of code generation. It uses templates to iterate over the relevant parts of a textually represented model. One example is the use of XML (XML, 2010) models on a XSLT (XSLT, 2010) transformation. In this case, the models are used as input in XSTL engines and the generated code is based on XSLT templates.

Despite this technique is fairly straightforward, the templates soon become very complex and hard to maintain. For this reason this approach is totally unusable for larger systems, especially if the specification is based on XMI files (Stahl & Völter, 2006).

Templates and Metamodel

This technique is used to avoid the problems related to the direct code generation from XML models. This approach implements a multi-stage generator that first parses the file, then instantiates the metamodel, and finally uses it together with the templates for generation. The figure 2.13 illustrates the principle.

There are two advantages of using this technique: first is the independence from the model's concrete syntax, for example UML and its different XMI versions. Second, an imperative programming language, such as Java, can be used for performing additional model verifications.

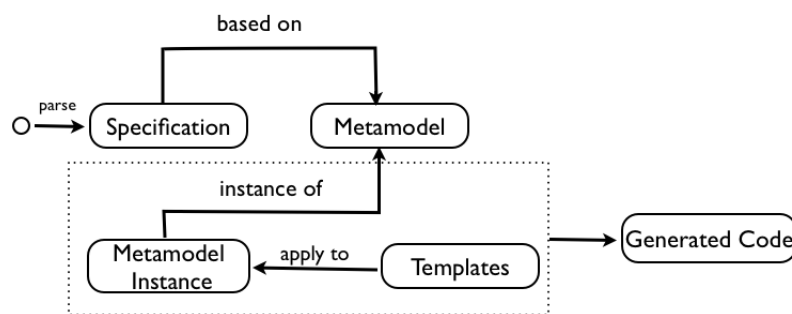


Figure 2.13: Templates and metamodel. Adapted from (Stahl & Völter, 2006).

API-based generators

API-based generators are the most popular types of code generators. Generally, this approach uses an API, based on the abstract syntax of the target language (metamodel), with which the elements of the target platform can be generated.

Therefore, they are always specific to one language. The figure 2.14 illustrates this method.

Typically the generation process begins to a set of call to the API to build an internal representation of the code (AST) and then a call to a helper function initiates the actual code generation. There are two main benefits of following this technique: a) it is very easy to use; b) only syntactically corrected codes are generated, this is guaranteed by the API in combination with the compiler of the generator code.

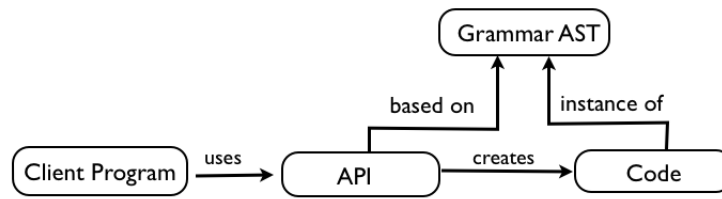


Figure 2.14: API-based generators. Adapted from (Stahl & Völter, 2006).

2.4 Reuse-based Engineering

“A program that is used in a real world environment must change or become progressively less useful in that environment.” (Lehman & Belady, 1985) The Lehman law, first postulated in 1985, could not be more valid in today’s competitive software industry. With the constant changes in the user requirements for great part of the projects, the software engineering had been trying over the last years to find a way to solve this issue.

One of the most effective methods for addressing the demand for change is software reuse. It is clear that a higher proportion of software assets that can be adapted or assembled to build a new product, leads to a greater competitive position of the company.

There are a quite number of technologies posed as solutions to the software reuse. However only a few have a significant impact in the industry. They are described in the following subsections.

2.4.1 Component-based development

The component-based development proposes a software development based on assemblies of existent executable software units (Crnkovic *et al.* , 2006). This approach can be viewed as an extension of the traditional object orientation paradigm. In this case, new applications are not only built using object orientation mechanisms such as inheritance,

polymorphism and dynamic binding but also using what is called service-orientation, where greater component or main program is composed of a set of others ready-to-assemble or ready-to-run components.

Therefore, the component paradigm promotes the reuse in lowest level of granularity, since components represents the smallest package of software that makes sense as a stand-alone and reusable entity.

2.4.2 Product line engineering

Product lines promote the reuse at the largest level of granularity. This approach is based on the fact that most softwares companies build or maintain a family of similar products rather than a single product. The development of new software is consequently based on the integration or tailoring of variables softwares parts to a common reusable core (Pohl *et al.* , 2005).

OO frameworks constitute one way of developing product line based softwares. Through class specialization and interface realization, these mechanisms represent the relationship between the domain-specific but application independent common core of a product line and its application-specific variants. The framework corresponds to the product line core and the product line variants are different instantiations of the framework.

2.5 Orthographic Engineering

Different paradigms have been experimented by IT industry over the last years in an effort to accommodate the market expectation for high quality and complex application at low cost. All the models of development discussed before such as model-driven engineering, component-based development and product line engineering try to use a different combination of abstraction and decomposition techniques to break a complex system into manageable unities in an attempt to achieve this goal. Model-driven Engineering introduces views at the various levels of platform specificity together with the transformations and Product Line Engineering introduces the family wide and product specific views of systems. However, one consequence is that all of these approaches increase the number of artifacts (views) created the development process.

Multiple views approach for software development has been recognized for long time. Even the first generation object-oriented methods such as Booch (Booch *et al.* , 2007)

and later with Fusion (Coleman *et al.* , 1994a) and RUP (Kruchten, 2003) supported a number of different diagrams for capturing distinct aspects of the system.

When these methodologies are used together the number of views quickly grows in an exponential way and the lack of one view management tool to deal with these challenges may prevent the developer to take the advantages of each method.

The (C. Atkinson & Bostan, 2009) proposes the Orthographic Software Modeling (OSM) as alternative to deal with the different views of the system in a systematic and flexible way. OSM is grounded in the Single Underlying Model (SUM), that contains all informations exhibited through different views of the system. Users of OSM tools, however, never have access to the SUM, instead they are only able to manipulate the model is through the views. This approach decreases the system development complexity as whole and is illustrated in the figure 2.15.

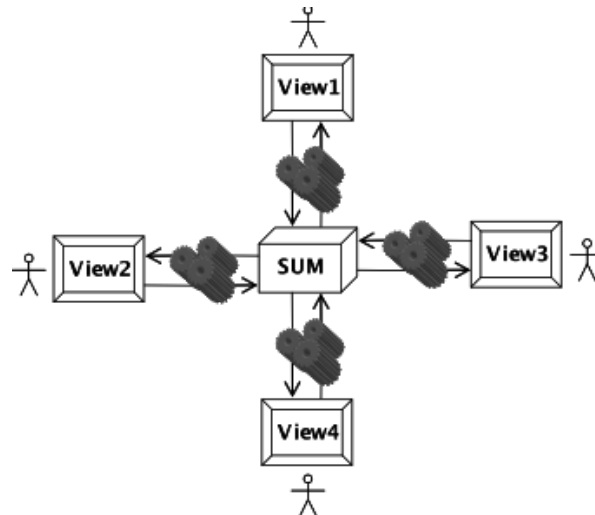


Figure 2.15: Orthographic software modeling

2.6 Chapter Summary

This chapter exhibited some of the software engineering concepts used as background for this work. It was presented the Model Driven Engineering and its principles, standards and technologies, Reuse-based Engineering, Model transformations and Code generation techniques.

All this concepts are useful to understand the Kobra methodology and the projects that are being developed by the ORCAS research group in which this research is part of.

3

KobrA2, KISF and WAKAME

3.1 KobrA2

KobrA2 is software engineering method jointly developed by Universität Mannheim and Universidade Federal de Pernambuco that aims at filling the methodological gaps of popular methods such as XP(Beck & Andres, 2004), SCRUM(Schwaber & Beedle, 2002) or RUP(Kruchten, 2003).

The three mentioned methodologies provide very little or no guidelines at all for two practices that are critical to sustain long-term software productivity. They are: the software reuse, and the software decomposition¹.

Model-based methods such as RUP require building models, however they provide no precise guidelines to define what artifact to build and which part of the huge UML2 metamodel to use during the software development process steps. This prevents the productivity gain that can be obtained through automation from models proposed by MDE based approaches.

KobrA2 leverages the modularity of the UML2 metamodel to select a general purpose yet minimal subset of constructs from the UML2 to use to construct PIMs. KobrA2 also leverages the OCL standard to build completely precise and formal models, refined enough to serve as input for full framework and tool code generation.

3.1.1 KobrA2 goals and principles

The larger goal of KobrA2 is to improve software productivity through code generation from PIM reuse. KobrA2 focuses on models constructed within the Object-Oriented

¹Separation of the software model (or code, in the case of agile methods) into independent concerns (e.g., business vs. platform concerns, structural vs. functional vs. behavioral concerns etc.) that are able to be manageable from different teams

(OO) software engineering paradigm and the family of standards defined by the Object Management Group, in particular UML2, OCL2 and MOF2.

In addition to MDE and OO, Kobra2 is based on seven other main principles:

Multi-dimensional, systematic separation of concerns

Separating concerns in distinct artifacts improves software productivity in two key ways. First, it allows artifact reuse at arbitrary granularity level. The second benefit of separation of concerns is to minimize developer cognitive overload both in terms of artifact size and heterogeneity. In this sense, Kobra2 separates:

1. The common framework functionalities that recur across applications of a given domain or product line, from the specific functionalities proper to each application;
2. The PIM of a component from its PSM and its source code;
3. The public *specification* of a component that describes the functionalities that it provides to external client components and requires from external server components, from the private *realization* of the component that describes how it internally assembles its externally provided services from its externally required and hidden services;
4. The static *structural* decomposition of a component from its dynamic behavioral decomposition and its *operational* decomposition that bridges its structural and behavioral elements;

Top-down decomposition

Each component realization can be recursively decomposed as an internally encapsulated assembly of finer-grained components not visible from the outside.

Standard Reuse

This principle fosters to reuse existing standard languages that are compatible with its chosen OO MDE orthographic paradigm. Kobra2 only reuses the most consolidated standard languages and within these the most consolidated constructs.

Parsimony

Every descriptive artifact should contain the only needed amount of information to describe the necessary properties for that artifact, but no more. 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.

Uniformity

By this principle every entity is treated as a component and every component is treated uniformly regardless its granularity or location in the component tree. In Kobra2 the whole system is viewed and modeled as a component and any component is viewed as an independent system.

Locality

By the locality principle, each component model contains only properties about itself. This means that in Kobra2, there is no model covering all system aspects. Even the root component in the component tree has only black box view of its sub components. To specify component owner of the view, Kobra used the «*subject*» stereotype.

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.

3.1.2 Kobra views

Following the Multi-dimensional principle stated in 3.1.1 and OSM 2.5, Kobra2 defines sixteen views to tackle different concerns of the software. In this subsection, we present just the views used for modeling the system proposed by this work:

- **Specification Structural Class Service View** Specifies the local assembly connections of the subject component class, and its interface. Allow only public operations and attributes.

- **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 operations and attributes need to be public.
- **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 IOCL expressions of the operations.
- **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 IOCL expressions of the operations.
- **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 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 IOCL expressions 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 IOCL expressions of the operations.

3.2 KISF

KISF, an acronym for *KobrA Information System Framework*, is a framework designed as an extension of KWAF (Marinho *et al.*, 2009), aimed to provide a generic and domain-independent infrastructure for the development of Information Systems. It shapes aspects of a generic Information System, from the GUI to the Web services, through the data model. The main idea is that through the specialization of the KISF 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 KISF framework. The use of KISF is illustrated by the case study of this work.

The idea behind KISF is to abstract any Information System. The KISF is composed by two nested two sub-components. The first is responsible for performing the graphical user interface presentation and user interaction (*PresentationLogic*), while the second sub-component handles business services (*BusinessService*). The organization of these component is shown in the figure 3.1.

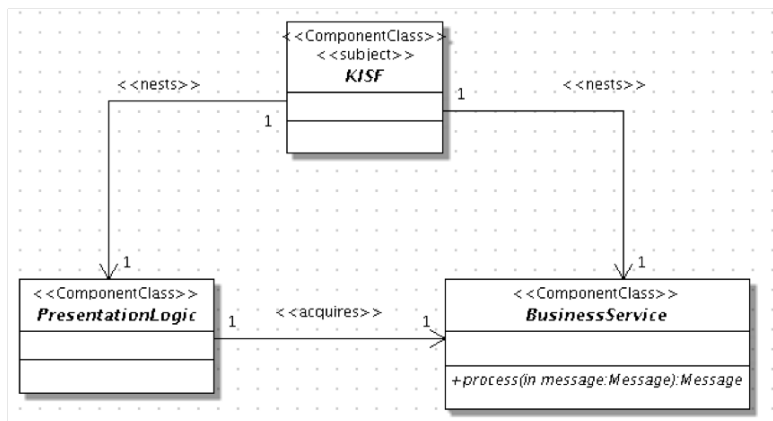


Figure 3.1: KISF Top Level

The following sections we present each one of this components and subcomponents.

3.2.1 Business Service Component

The *BusinessService* component is responsible for handling services that the system will provide. According to the definition shown by Lewandowski (Lewandowski, 1998), we can consider this component being at the server side in terms of the Client/Server architecture. The *BusinessService* 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: Message) : Message*. In this method all the exchange of information will be done through the *Message* class. A message is an object that encapsulates information about the action to be executed and a map that holds data such as the operation parameters and the operation results. After the process method is invoked, the *BusinessService* delegates the responsibility to fulfill that operation to the same method signature of the *BusinessFacade* component, which is responsible in turn for redirecting the message according to the action specified through the appropriate *BusinessLogic* component. The last one can also acquire the *PersistentDataModel* component in order to perform operations with the database. In short, KISF defines the *BusinessService* being composed of a *BusinessFacade*, several *BusinessLogics*, and one *PersistentDataModel* component. The *Realization Structural Class Service* of the *BusinessService*, figure 3.2, illustrates these relationships.

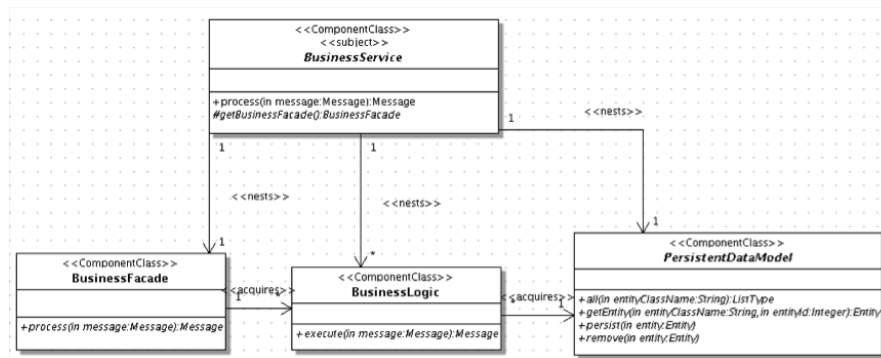


Figure 3.2: BusinessService - Realization Structural Class Service

Business Facade Component

The *BusinessFacade* can be considered part of the controller role in the MVC architectural pattern (Gamma *et al.*, 1995) at the server side. The *BusinessFacade* contains a single method with the same signature of the *BusinessService* component, and it has the responsibility of redirect the client's requests to the appropriate *BusinessLogic* that

will handle this request. In the KIF the process method is abstract and it is up to its specialization to implement the logic. The *process()* method of this component will check which is the action requested by the client, and call the BusinessLogic method *execute()* of the corresponding action, passing the Message object received. The figure 3.3 depicts the *Specification Structural Class Service* view of this component.

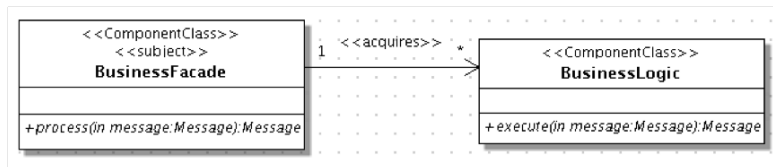


Figure 3.3: BusinessFacade - Specification Structural Class Service

Business Logic Component

The BusinessLogic component, depicted in figure 3.4, performs all the services provided by the application, it represents part of the model in the MVC architectural pattern, handling the business rules. The developer that uses the KISF 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 it. The *execute(message: Message):Message* method should be specified to perform the action in question, and this is the method called by the BusinessFacade.

For the persistent data access, the BusinessLogic component manipulates the entities defined in the *PersistentDataModel* component, for both: recovery and persistence of data.

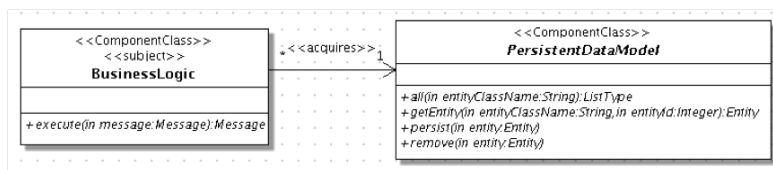


Figure 3.4: BusinessLogic - Specification Structural Class Service

Persistent Data Model Component

The PersistentDataModel component, depicted in figure 3.5, is responsible for keeping the entities and data types of the application, which requires persistence capabilities. It represents the Model in the MVC architectural pattern. The PersistentDataModel

component has the class *Entity* which represents an persistent entity of the application to be shared across all the application, in the server side as well on the client side. Every entity the KISF users needs to be persistent, they must specialize the KISF *Entity* class.

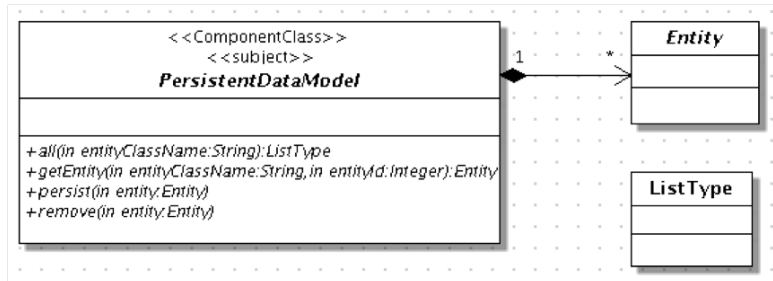


Figure 3.5: Persistent Data Model - Specification Structural Class Type

3.2.2 Presentation Logic Component

The Presentation Logic component, presented in figure 3.6, is responsible for modeling the client side of the application and it is composed by two nested subcomponents, the *PresentationView*, corresponding the view in the MVC (Gamma *et al.* , 1995) architectural pattern and the *PresentationController*, representing the client part of the controller role in the MVC pattern. *PresentationView* is responsible for modeling the graphical user interface and the *PresentationController* is responsible for modeling of events and treat user interaction with the application in the GUI.

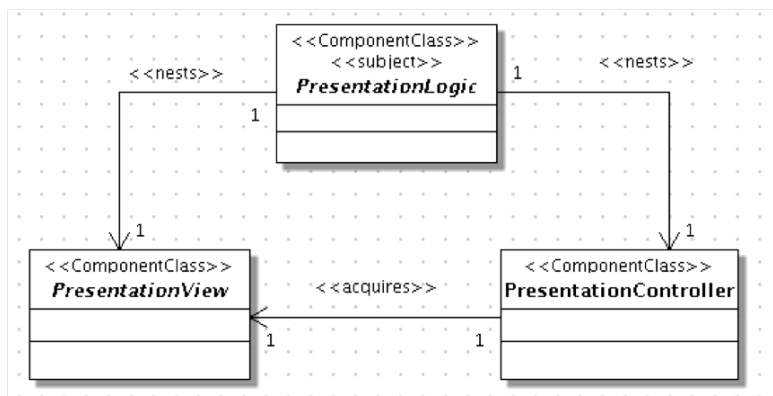


Figure 3.6: Presentation Logic - Realization Structural Class Service

Presentation View Component

The PresentationView 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. More information about this framework can be found in (Lacerda, 2007)

Presentation Controller Component

The PresentationController 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 (BusinessService) as well the handling of presentation logic, such as control widgets behavior or appearance. The behavior of all these actions is defined with the use of Imperative OCL. More information about the sub-components that make part of the PresentationLogic can also be found in (Lacerda, 2007).

3.3 Modeling Frameworks

Among the various Model-Driven Engineering CASE tools existing, just those focusing on the following characteristic were taken for analysis:

- **Model 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 we just take into account the Eclipse Modeling Framework (EMF) (EMF, 2010), the model store framework used by WAKAME.
- **Engines for Model Transformation** - this category addresses the tools specialized in carrying transformations of models, specifically related to text generation from Platform Independent Model. Amongst the various tools available in this category, we selected to present the MOFScript (MOFScript, 2010) tool.

3.3.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) (EMF, 2010) is one of many open-source projects being developed by Eclipse Foundation. Simply put, the EMF is a modeling framework and code generator facility for building models repositories.

With EMF, the user is able to define models and generate a repository from it. The EMF API makes possible to serialize (using the XMI standard), store and retrieve models.

The model used to represent models in EMF is called Ecore. Ecore is itself an EMF model, and thus is its own metamodel. The Ecore metamodel is depicted in figure 3.7.

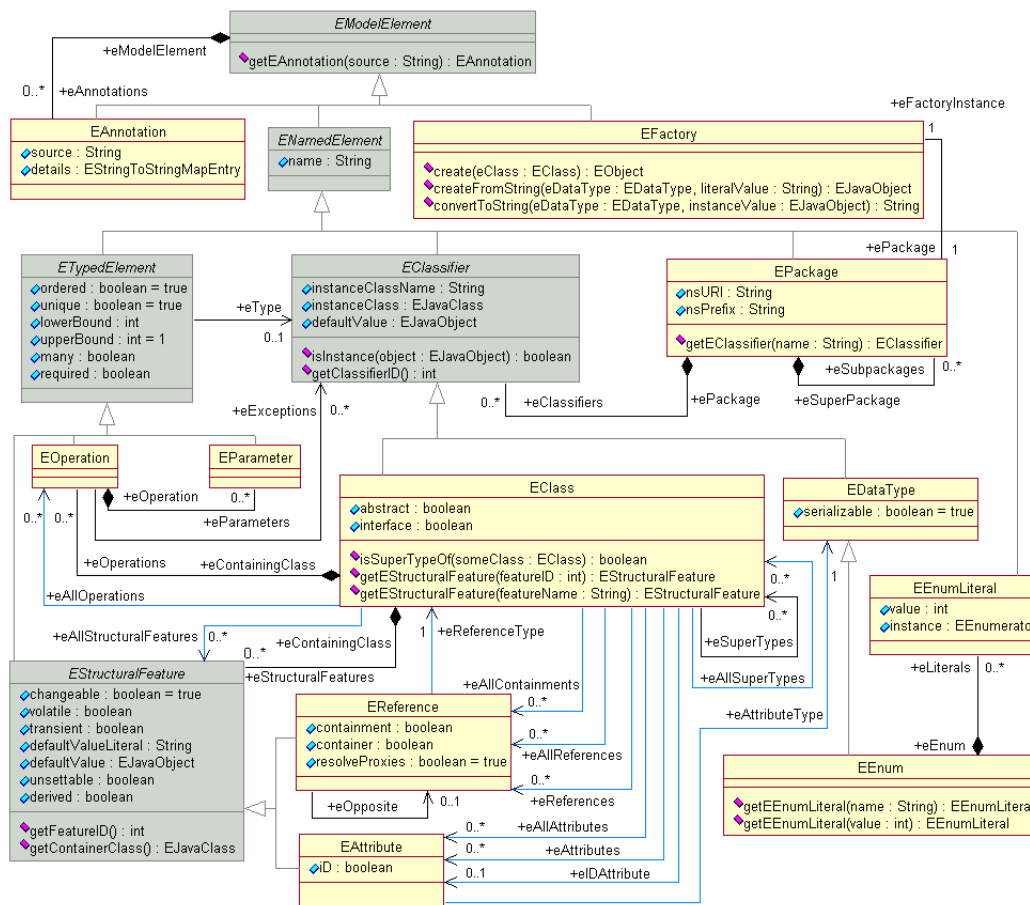


Figure 3.7: Ecore Metamodel (EMF, 2010)

Ecore metamodel is very similar to EMOF metamodel, from MOF specification. The main difference stems from the fact Ecore explicitly differentiates an attribute (EAttribute) from a reference (EReference) while in EMOF both are equivalent to Attribute.

Perhaps one of the most important benefits of EMF is the java code generation capability. The metamodels created serves as input to the EMF engine that generates all

the code for the repository. This step specifically involves another model type of model called GenModel. The GenModel is an Ecore model specifically used for code generation. The GenModel adds the the missing information that is not stored in the Ecore model such as where to put the generated code what prefix to use for the generate factory and package class names.

For each modeled class, EMF generates one interface and a class to realize it. Other thing to notice about the generated interface is that it extends directly or indirectly from the base *EObject* interface. The EObject interface is the basis of all modeled object.

Additionally, the EObject extends another interface, the *Notifier*. The Notifier interface introduces the model change characteristic, as in the Observer design pattern (Freeman *et al.* , 2004) for every modeled object. So, like the object persistence, notification is an important feature of an EMF object.

There are two other important classes generated for a model: a factory and a package. The generated factory introduces the create method for each class in the model and the generated package provides convenient accessors for all the Ecore metadata for the model.

3.3.2 MOFScript

MOFScript is a complete tool for model to text transformations. Its language responded to the OMG's RFP process to define a MOF Model to Text Transformation Language (MOFM2T, 2010). This RFP intended to standardize the transformation of models into a textual representation. Several mandatory requirements were listed in the request specification, such as:

- Alignment to existing OMG standards (e.g. QVT)
- Generation of text from MOF-based models
- Transformations should be defined at the metalevel of the source model
- String manipulation (i.e. The ability to manipulate string values)

The MOFScript tool is composed by two main architectural parts: the tool components and services component. The tool components are end user tools that provide the editing capabilities and interaction with the services. The services provide capabilities for parsing, checking, and executing the transformation language. The language is represented by a model (the MOFScript model), an Eclipse Modeling Framework (EMF) model populated by the parser and this model is the basis for semantic checking and execution (MOFScript, 2010).

3.4 WAKAME

All the idea of Kobra and dynamic view management would not be possible without a CASE (Wikipedia, 2011a) tool to assist users during the PIM specification. Originally developed during two master's degree students, mainly due the nonexistence of a tool with support for the Kobra methodology, the Web App for Kobra Model Engineering (WAKAME) project, came to fill this gap.

Thus, with intention to cover the Kobra principles, the WAKAME application has the following features (Machado, 2009) (Marinho, 2009):

- **Draw diagrams** The tool supports easy rendering of the diagrams in the modeling language. The tool is “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.
- **Multi-views support** For each component, WAKAME provides one view for each point in the multi -dimensional space of separate concerns;
- **Consistency between SUM and view** The tool maintains this consistency through transformations of models;
- **To maintain consistency among the views** Due the fact the views are dynamically generated, when the user switched between view the information shared between them are consistent;
- **Local visions** Each view of a component in WAKAME only brings the necessary information for understating the same;
- **Navigation** WAKAME tool also allows the navigation between component, and through views of each component;
- **Store models** The tool must support storing models to the database. This feature was specially implemented thanks to Eclipse Modeling Framework project (EMF, 2010).

WAKAME is a Google App Engine application written in Java, deployed entirely on Google's infrastructure and it is available through the link <http://wakametool.appspot.com>. This decision of choosing Google App Engine was made to take advantage of the scalability offered for applications developed on top of this environment.

Google launched this project in April 2008 with the proposal of making easy to build an application that runs reliably, even under high intense with larges amount of data access. This is achieved by replicating the application to other nodes when needed and by the use of the Bigtable storage(Chang *et al.* , 2006), a high performance and extremely large-scale database system.

The figure 3.8 shows a screen shot taken from WAKAME during the modeling of a component. On the left menu, the user is able to select the appropriate view to work on. The *Component Navigation Tree* on the top right shows the nesting of components and it is used to select the *subject* component. The *Element Selection Tree* shows all the elements available for selection (i.e. classes, enumerations, components) that could be reused, during the specification of views of other components, for example. On the bottom of the screen there is a console that logs all server side operations. Finally, the *fromCloud* and *toCloud* buttons are used for retrieve and save the model to the Google's cloud respectively.

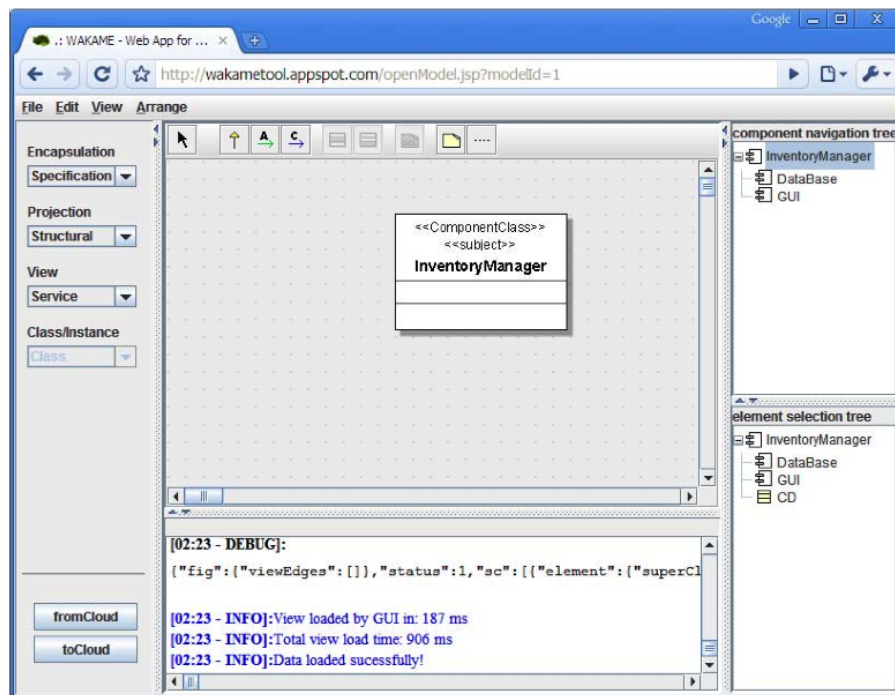


Figure 3.8: WAKAME Edition Screen

3.5 Chapter Summary

This Chapter presented a brief review of this research context. On Section 3.1 the Kobra2 methodology was discussed. We showed its principles and views that were observed in practice on the Section 3.2 that discuss a generic framework for the development of Information Systems (KISF).

At the end of the chapter we showed the WAKAME tool, project of the ORCAS group, developed to fill the lack of a case tool that supported the Kobra process. However, WAKAME still needs a code generation tool to completely generate a ready-to-deploy application.

It is in this context that WAKAME Code Generator project emerged and its goals and architecture will be further be presented in Chapter 4.

4

WAKAME Code Generator

In this chapter we describe the foundations and requirements of the WAKAME Code Generator tool together with the code generation techniques used to accomplish the project objective. In addition, the last section shows the Kobra top-level PIM of the WAKAME Code Generator component.

4.1 Long-Term Goals

The goal of the WAKAME generator component is to generate automatically the complete code of an application from a Kobra PIM SUM. The SUM model results from editing Kobra PIM views using the WAKAME view editor component and then merging them together into a single model (SUM) using the WAKAME view merging component. It consists of a UML2 components and class assembly where the operation constraints, preconditions, postconditions and bodies, are specified in IOCL.

The target code platform is the restricted version of the Java Runtime Environment that can be transparently deployed on Google's cloud through the Google App Engine (GAE) (Google, 2011). The code to generate include:

1. the application GUI;
2. the structural code of the Java classes and interfaces that implement the UML2 component;
3. the behavioral code of the Java methods that implement the IOCL bodies of the UML2 operations;
4. the calls to the GAE persistence operations implementing the CRUD operations on the UML2 classes that represent the persistent data of the application at the PIM

level;

5. deployment artifacts (i.e configurations files);
6. units tests and application documentation.

In order to accomplish these goals, the WAKAME Code Generator was divided into three main components:

1. generator of the structural and deployment code of an information system from Kobra2-UML2 components and classes specializing the business services component of KISF framework;
2. generator of the behavioral code of an information system from IOCL expressions representing in platform-independent fashion the algorithms of the operations contained by these Kobra2-UML2 components and classes;
3. generator of the GUI layout and navigation code of an information system specializing the presentation layout component of the KISF framework.

The figure 4.1 illustrates the general architecture of the WAKAME generator component.

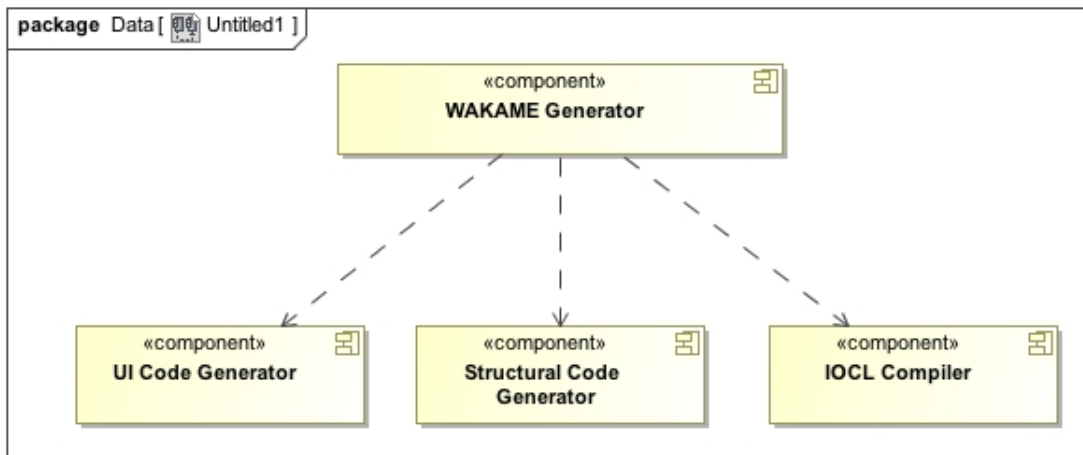


Figure 4.1: WAKAME Generator Structure

Based on (Calic *et al.* , 2008) and on the code generator tools analyzed, we have outlined a set of requirements the WAKAME Code Generator tool should have:

1. To be a Kobra2 driven method;
-

2. To generate the entire application code and project infrastructure required for running the application.
3. To support automatic generation of the API project documentation (Javadoc ¹);
4. To have a good performance even when generating code from a big Kobra2 structure graph, which could require a fairly amount of memory footprint.

My MSc. dissertation contributes to this second component. The MSc. dissertation of Thiago Oliveira (Oliveira, 2011) contributes to the first component. The issues raised by the third component will be tackled in a future work.

4.2 Imperative OCL Compiler Architecture

The Imperative OCL Compiler was divided into the Imperative OCL Engine and the Code Generator components. The idea is to create an independent component for performing IOCL related operations but not tied to any code generation technologies. The first component is responsible for *parsing* and performing *type checking analysis* of IOCL expressions. It also has operations for performing completion to partially input expressions, in the same manner as modern IDEs. This feature is important for WAKAME users because it provides not only suggestions about the available operations or navigations the user could write but it also performs real time syntax and semantic checking for the pre-entered expressions and therefore, allowing early error detection and increasing the productivity in a general way. The second component consists of a code generator core, which performs the code generation flow and a number of languages catridges, which allows the compiler to generate code to different target languages.

The figure 4.2 illustrates the separation of compiler into two independent components.

Although they are independents, the CodeGenerator requires the use of the IOCL Engine to work properly. The reason is because the CodeGenerator receives as input a parse tree of the IOCL engine. All the details will be formally specified when the Kobra PIM of both components is presented in the chapter 5.

¹<http://en.wikipedia.org/wiki/Javadoc>

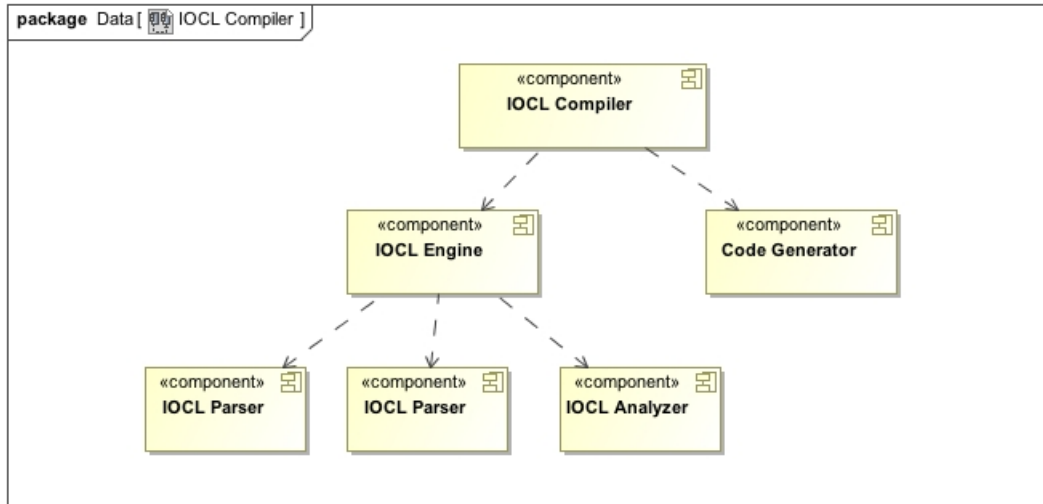


Figure 4.2: Imperative OCL Compiler Architecture

4.2.1 Imperative OCL Input

The declaration of the Imperative OCL expressions in the WAKAME tool is done through the *Operational Views* 3.1.2. Such view contain all three types of UML operation constraints: **pre and post conditions and body expressions**. This can be seen in the figure 4.3.

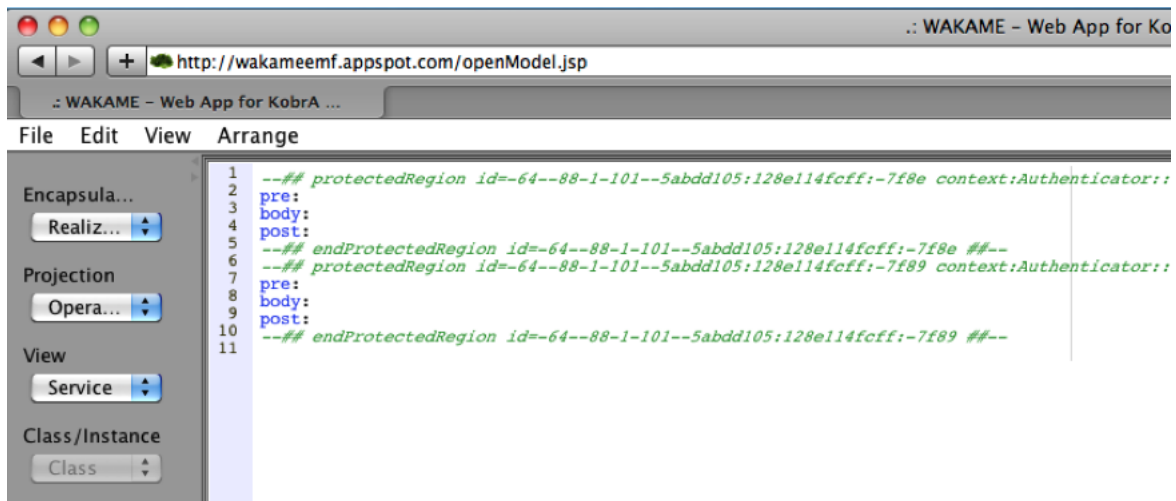


Figure 4.3: WAKAME Operational View

The image 4.3 illustrates the tree function types available for completion. After saving this view, the IOCL expressions will be stored in the Kobra models as *String*, to be further used for code generations purposes.

4.2.2 Code Generation Strategy: PIM to Code

The adoption of the multi-staged transformation described in Section 2.3.1 was evaluated during the early phases of this work. Our first prototype involved the usage of ATL for performing the firsts steps of the compiler and utilization of MOFScript (MOFScript, 2010) to execute the model-to-text transformation step.

However the need of several metamodel definitions contributed to rejection of this alternative. The PSM metamodel definition involved several internal discussion among the ORCAS group members, mainly related to the necessities PSM elements that should be present at the target language metamodel and the necessities APIs that needed to be further specified. Even before the conclusion of this step this approach ended up to be discredited.

The lack of maturity of the IDEs to work with this kind of transformation along with the high level of experience and Java skills programming by the author when compared with M2M technologies also influenced this decision. ATL and also the Epsilon (Epsilon, 2011) tool, which was analyzed during this work, both sinned by the deficiency of good editors and debuggers, fundamental requirement during the development stages.

After taking above reasons into consideration, we decided to go with the single-stage or PIM to code approach. The decision is also adopted by several others MDA code generators tools (Stahl & Völter, 2006).

Our option although does not discard the the multi-staged transformation at all. As a future work, the IOCL compiler could be extended to support it.

4.2.3 Adopted techniques

After deciding by the PIM to code approach, we had to choose among the code generation techniques presented in 2.3.1 to see the best one applicable in our case.

Since WAKAME stores the IOCL expressions as *String* in its models, we had two necessary steps in the development: the **parsing** and **semantic analysis** of the expressions, which typically involves *context* evaluation. After that, we can transform the IOCL expression into either generate a XMI representation and reuse any model-to-text technologies such as MOFScript (MOFScript, 2010) or transform it directly into source code using consolidated patterns such as the Visitor (Gamma *et al.* , 1995).

The first option was evaluated but it involved an extra step which would not bring any real benefit to the work. Also, M2T technologies such as MOFScript still lack of a good development environment for writing transformation and debugging them. This was the

key reasons why we do not adopted M2T.

However, during the evaluation of the M2T tools, it was observed that the use of templates² for text generation purposes simplifies the transformation and we also chose this approach.

This led us to adopt the *Template and Metamodel* technique presented in 2.3.1 for the development of the IOCL Compiler. After the model instantiation, we use the Visitor pattern to traverse the expression tree and templates to help with code generation. The Figure 4.4 depicts the adopted transformation flow.

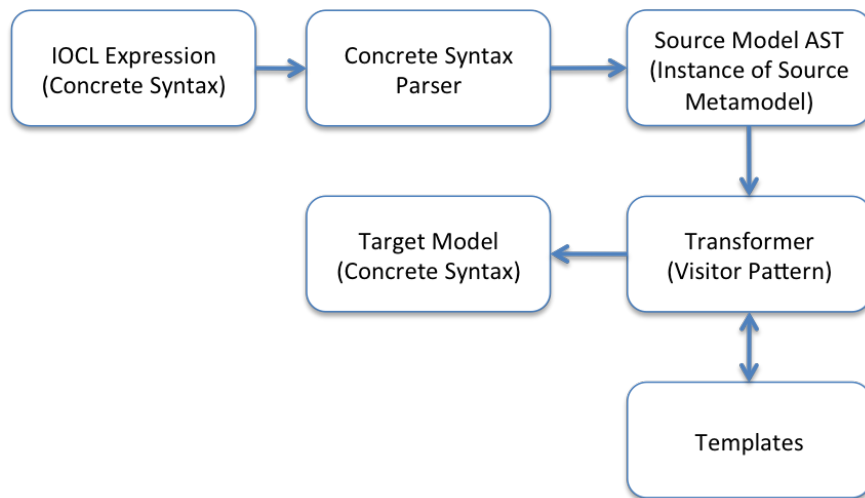


Figure 4.4: AST-based generator’s mode of execution. Adapted from (Stahl & Völter, 2006)

4.2.4 Dealing with pre and post conditions

Imperative OCL expressions are used to specify constraints in Kobra models. As we said in 4.2.1, pre and post conditions are supported by WAKAME. They are a form of specifying runtime constraint checking of the WAKAME generated applications. However for these checks to be effective, they must satisfy some requirements including transparency and modularity (Yoonsik Cheon, 2009).

Transparency states that the execution of the checking code should not change the behavior of the program unless the constraint is violated. Modularity is related to code checking organization and it means that the constraint checking code should be organized in modules separated from the main program to improve maintainability by enforcing the separation of concerns.

²Technique employed by many of the M2T tools such as MOFScript, JET(JET, 2011), etc.

The WAKAME Generator approach to deal with runtime constraint checking is based on Aspect Oriented Programming (AOP) (Kiczales *et al.* , 1997). Specifically, by translating OCL expressions to AspectJ³. AspectJ is an aspect-oriented extension developed for the Java programming language.

Others approaches are also studied in Carmen Avila (2010) for dealing with constraint checking and in spite of the aspect solution is not being considered best in terms of performance, this approach offers the transparency and modularity characteristics to the software and therefore it is rated as a good solution in terms of software quality.

There are three types of advices available in AspectJ: *before*, *after* and *around*. Each one is related to the order of execution of the advice. The before advice is executed is executed just before the joint point is executed. After advice is executed right after the joint point executed. The around advice is executed as the program flow reaches the joint point, but unlike the before and after advice, the around takes control of the joint point execution, deciding whether it will be executed or not.

For the WAKAME generator tool, we took the advantage provided by the before and after advice over method calls for implementing the pre and post runtime checking respectively.

4.3 WAKAME Architecture

The PIM of WAKAME was defined using the KISF framework 3.2. For the realization of the WAKAME component, the sub-components *BusinessService* and *PresentationLogic* of KISF were redefined to *WAKAMEBusinessService* and *WAKAMEPresentationLogic* respectively.

In this section we detail the server side of WAKAME where the WAKAME Code generator is located.

4.3.1 The WAKAME Business Service

The *WAKAMEBusinessService* component is responsible for processing the services requested by *WAKAMEPresentationController* component and for the persistence of the data. Figure 4.5 shows its integration with the *WAKAMEGenerator* component. It is done by a new specialization of the *BusinessLogic* component of the KISF called *GenerateApplicationAction*. This component calls the specific methods on the WAKAMEGerator,

³<http://www.eclipse.org/aspectj/>

based on attributes provided by the Message object.

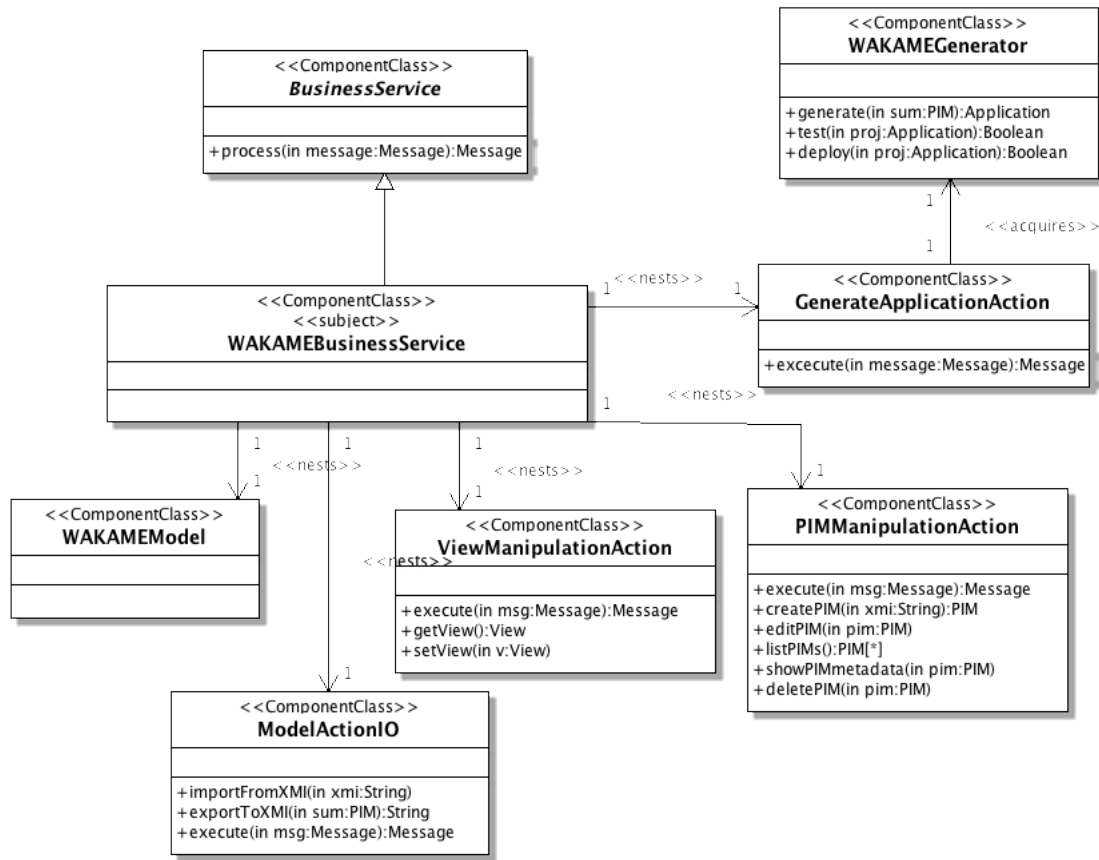


Figure 4.5: WAKAME Business Service - *Realization Structural Class Service View*

4.4 WAKAME Code Generator Top-Level PIM

Figure 4.6 shows the *Realizations Structural Class Service 3.1.2* of the *WAKAMEGenerator* component. The *WAKAMEGenerator* is a top-level component and it acts as an interface, redirecting all the operations to the *StructuralGenerator* component. Thus, *StrucutalGenerator* implement its three main operations, each one related to a specific development cycle. They are: *generate*, *test* and *deploy*. The *generate()* receives a KobrA model as input and outputs the structural Java code together with deployment artifacts (such as XML or property files). It works by performing a top-down traversal of the model element containment tree. When it encounters an IOCL expression node in this tree, it calls the *IOCLCompiler*. The *test()* method checks whether the generated persistent entities are being stored or removed accordingly in the cloud data store or not.

The *deploy()* operation packs all the generated code into a *war* (Web Archive) file ⁴ and sends it to the Google's cloud in order to the application be deployed.

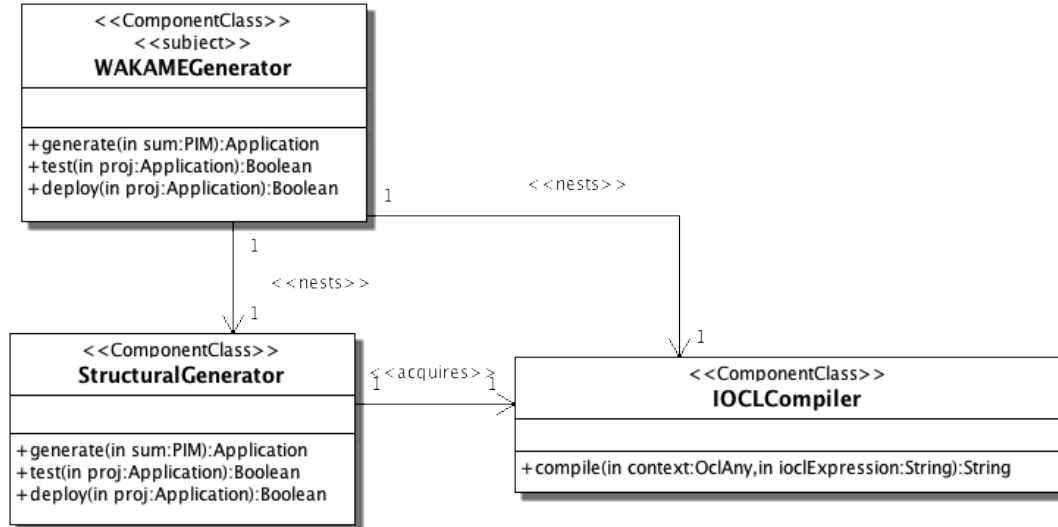


Figure 4.6: WAKAME Generator - *Realization Structural Class Service View*

4.5 Chapter Summary

In this chapter we exhibited the long terms goals and the submodules of the WAKAME Code Generator project, which this work is part of. After that, we showed the top level architecture of the IOCL Compiler and the techniques employed for the implementation. Finally, it was illustrated the integration of WAKAME Generator in the WAKAME tool. Next chapter the IOCL compiler will be more detailed.

⁴Standard format for Java web applications

5

The IOCL Compiler

In this chapter, we specify the `IOCLCompiler` component using Kobra methodology, showing its sub-components and the underlying decisions made to implement it. The component implementation is open source and the code repository could be accessed through the link <https://github.com/marcellustavares/imperative-ocl>.

5.1 IOCLCompiler PIM

According to Kobra method, a model is composed by nesting minor sub-components, and for this specific model, the top-level component is the *IOCLCompiler*. This component will encapsulate all other sub-components related to the expression parsing, type checking, code suggestions and code generation. The figure 5.1 shows its *Specification Structural Class Service* view. As stated in 3.1.2, this view specifies the public interface of the component, i.e. all the component's externalized operations.

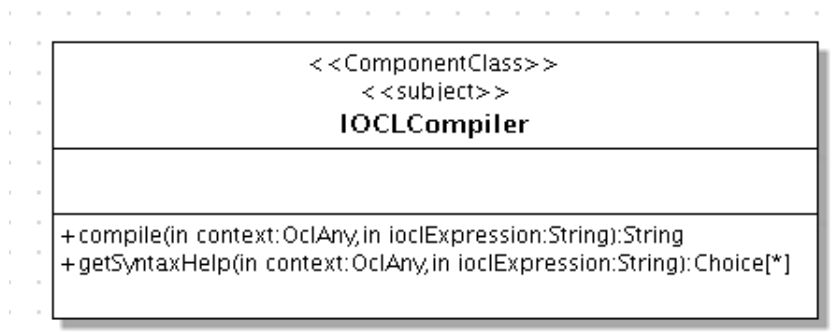


Figure 5.1: `IOCLCompiler` - Specification Structural Class Service

The first defined operation is *compile()*. This is the main operation of the component and it is responsible for transforming the IOCL expression into code that is dependent of

platform. The second operation, *getSyntaxhelp()*, is the one responsible for retrieving the available alternatives (represented as the Choice class in the model) for the incomplete typed expression.

In the realization of the *IOCLCompiler* component, namely the Realization Structural Class Service view, shown in figure 5.2, we detail its two nested components: *IOCLEngine* and the *CodeGenerator*.

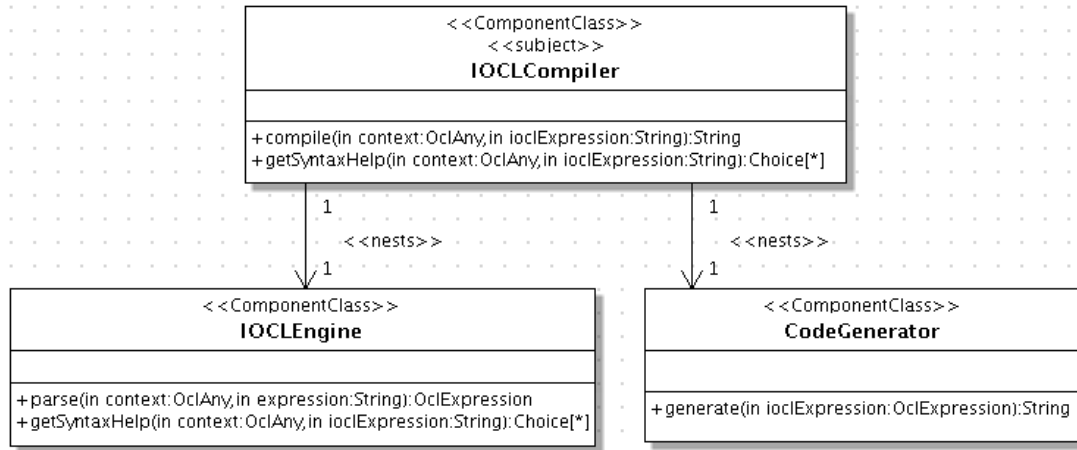


Figure 5.2: IOCLCompiler - Realization Structural Class Service

Both components will be further detailed in the next subsections.

5.1.1 IOCLEngine

The IOCLEngine component is responsible for parsing and performing the type checking of the IOCL expressions input in the WAKAME's operational views. Also, it has a helper function to help users to write their IOCL expressions. This is operation, called as *getSyntaxHelp()*, generates suggestions for the incomplete expressions.

In the realization of the *IOCLEngine* component, depicted in figure 5.3, shows its three nested components: *IOCLLexer*, *IOCLParser* and *IOCLAnalyzer*. Each component plays a different role inside the compilation process. IOCLLexer is responsible for converting the string source into a sequence of tokens and the purpose of the IOCLParser is to get the tokens generate by the lexer and create the abstract syntax tree representation of the Imperative OCL expression. Finally, IOCLAnalyzer deals with the semantic analysis of the AST.

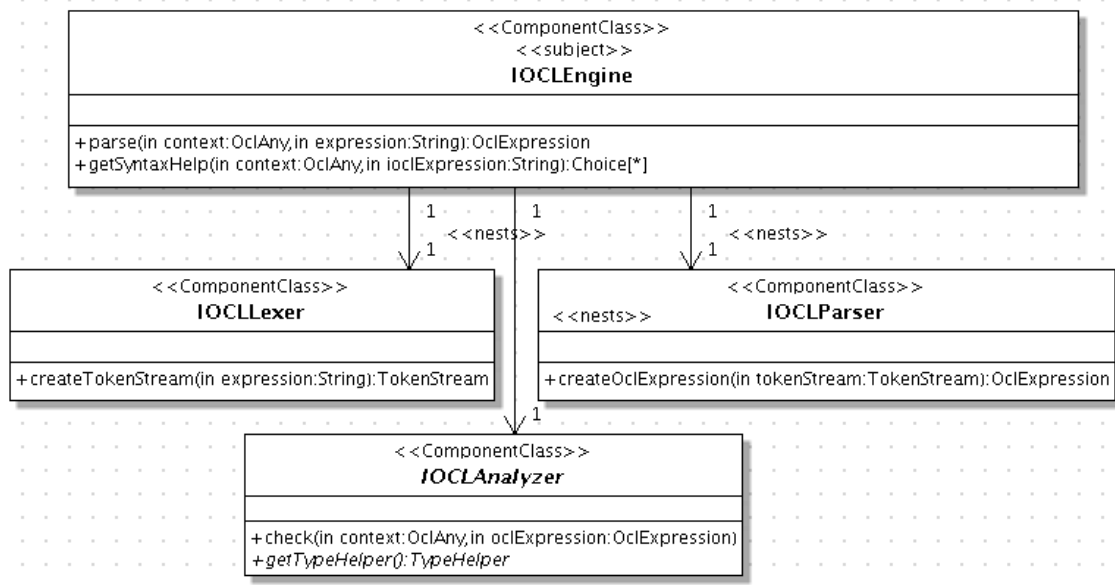


Figure 5.3: IOCLCompiler - Realization Structural Class Service

The *Specification Structural Service Type* view, depicted in 5.4, shows the non-primitive data types used by the IOCLCompiler.

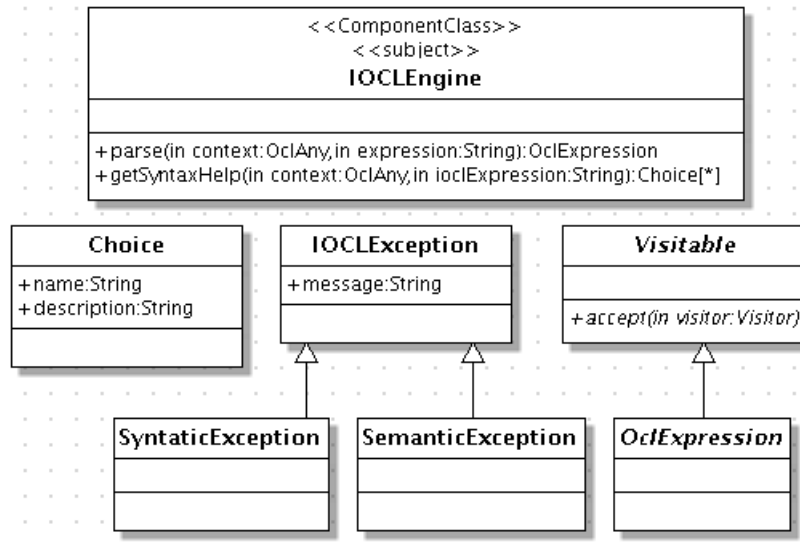


Figure 5.4: IOCLCompiler - Specification Structural Class Type

The *Choice* class, as already mentioned, represents one alternative presented to the user to complete expression and *getSyntaxHelp()* is the operation that retrieves this information. Internally, the *getSyntaxHelp()* uses the *IOCLAnalyzer* to deal with this context sensitive query. The details of the *IOCLAnalyzer* will be explained in section

5.1.1. The figure 5.5 simulates the end result where the choices are displayed to the user through a pop-up window in the WAKAME Operational views.

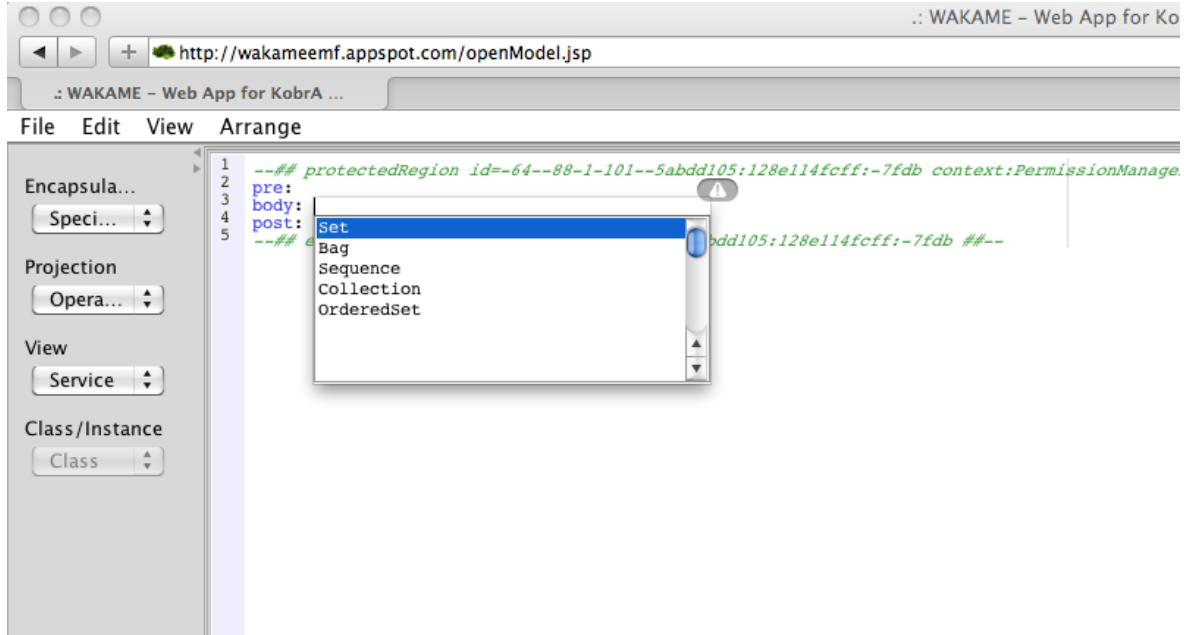


Figure 5.5: WAKAME Autocomplete

The *IOCLEException* represents the problems that may arise during the IOCL Engine operations. *SyntacticException* represents any syntactic error encountered for the expression passed as argument, while the other constitutes a semantic error. An example of semantic error would be an assignment expression for two different types of operands.

Finally the *OCLEExpression* is the top-level abstraction for all OCL and Imperative OCL expressions, exactly as specified in Section 2.2.2. However, two details are important to mention:

1. All OCL expressions and Imperative OCL expressions were modeled in Ecore, as depicted in figure 5.6. This way we could take advantage of two built-in features provided by the EMF framework: the serialization, which is needed to store the parsed expression into the GAE data store, together with the other parts of the model, and the code generator tool, responsible for generate all the expressions Java classes
2. *OCLEExpression* realizes the *Visitable* interface ¹ as defined in the picture 5.4. The *Visitable* interface is part of the Visitor pattern (Gamma *et al.* , 1995) used for the

¹In spite of *Visitable* being defined as an interface in UML, we represented it as an abstract class because interfaces are not allowed in Kobra models, according to Kobra metamodel (Robin, 2009).

code generations of the expressions and will be discussed in the IOCLGenerator component.

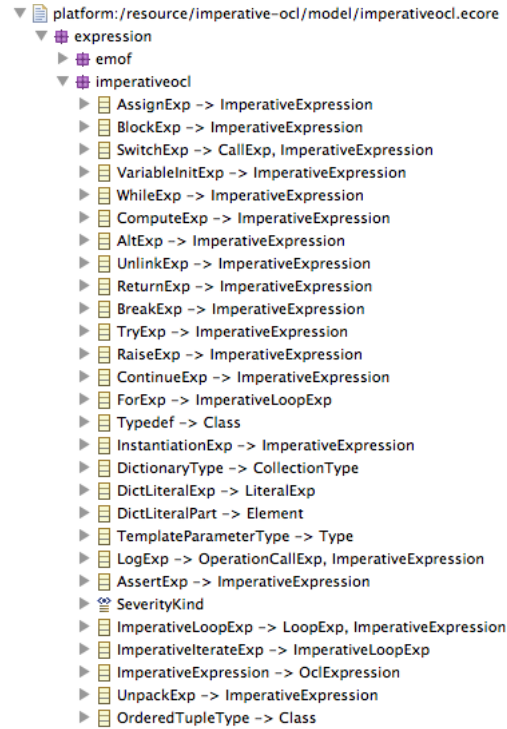


Figure 5.6: Imperative OCL expressions in Ecore

IOCLLexer and IOCLParser Component

The IOCLLexer is the component responsible for breaking up the input stream into a sequence of tokens. These tokens, called in our model as *TokenStream*, are then used to feed the IOCLParser component that in turn tries to recognize the expression structure. If the recognition is completed successfully, the IOCLParser is able to return the abstract syntax tree representation of the IOCL expression through the *createOclExpression()* operation, otherwise a syntactic error message is thrown to the user.

For the development of both components, two parsers generators were evaluated to be used in the project: ANTLR (Parr, 2007) and JavaCC (JavaCC, 2011). The criteria used for choice resulted from the analysis of three factors: *user documentation*, *development IDE* and *ease of use*.

- *User documentation* - The documentation found for ANLTR was more comprehensive than JavaCC. The ANTLR's site display several grammars examples, discussion lists and articles while JavaCC documentation is restricted to books.

- *Development IDE* - ANTLR offers a better development support when compared to JavaCC. The tooling support for specifying JavaCC grammars is based on Eclipse plugins. The plug-in includes features such as grammar formatting, syntax coloring, an outline view for JavaCC source, etc. ANTLR support is based on the ANTLRWorks project, a GUI development environment that helps edition, navigation, and debugging of grammars. Most importantly, ANTLRWorks helps the resolution of grammar analysis errors. The tool includes features such as: Grammar-aware editor, language-agnostic debugger for isolating grammar errors, Nondeterministic path highlighter for the syntax diagram view, Refactoring patterns for many common operations such as “remove left-recursion” and “in-line rule”, etc. The figure 5.7 illustrates the environment.
- *Ease of Use* - The ease of use is much related of a good development IDE availability and the simplicity to define the grammars. Both tools supports Extended BNF (EBNF) notation that allows optional and repeated elements. However, the parsing strategy used by ANTLR makes easier to write grammars when compared to JavaCC. ANTLR uses LL(*) parsing and JavaCC uses LL(k). which is more powerful than traditional LL(k)-based parsers. The latter is limited limited to a finite amount of lookahead (k), while LL(*) allows the lookahead to roam arbitrarily far ahead, relieving the programmer of the responsibility of specifying k. LL(*) does not alter the recursive-descent parsing strategy itself, it just enhances an LL decision’s predictive capabilities. Additionally, ANTLR is able to generate a default AST through a simpler notation (ANLTR’s tree description language) than JavaCC, that have this job done through the JJTree preprocessor.

All the factors described above revealed ANTLR to be a less risky tool solution to be adopted for defining the IOCL grammar. Our choice, however, does not implies that the tool or even the parser generated by ANTLR is better than JavaCC. The analysis of the alternatives were based on own requirements in order to select the tool that best fit in our development process.

After the technological choice and the subsequent grammar definition, ANLTR automatically generates a lexer and parser Java file. Both files were used without changes in the *IOCLLexer* and *IOCLParser* respectively. However, an additional operation had to be implemented for the last component. This operation coverts the ANLTR AST tree to our proper IOCL tree structure and is achieved by traversing the ANTLR tree and transforming its proprietary object nodes into IOCL objects defined in EMF as depicted in Figure 5.6.

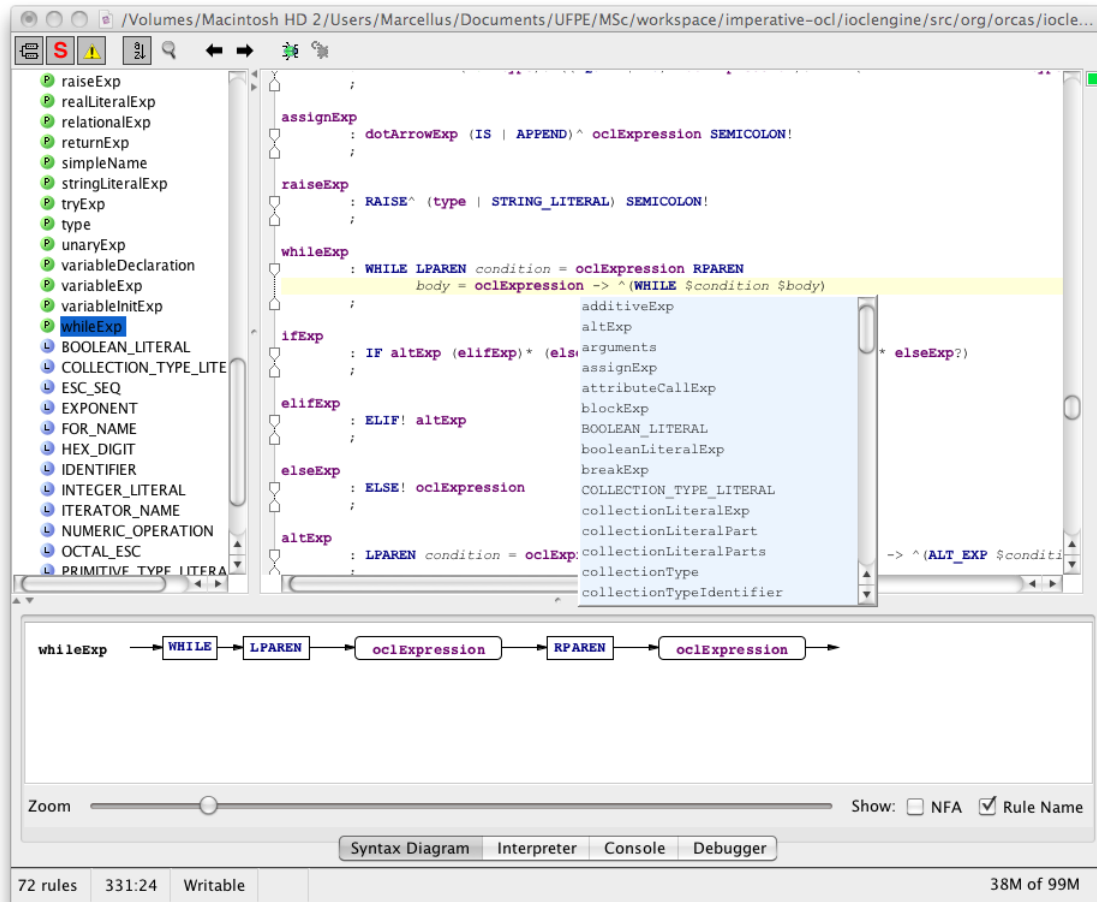


Figure 5.7: ANLTRWorks Environment

IOCLAnalyzer Component

All semantic analysis for the successfully parsed expressions is performed by the *IOCLAnalyzer* component. *IOCLAnalyzer* is an abstract component and its *Specification Structural Class Service* is illustrated in the figure 5.8.

Besides the *check()* method, which contains the abstract logic for transversing the AST representation and search for semantic incompatibilities, the component also contains the *getChoices()* operation. The latter deals with the generation of possible alternatives presented to users in order to facilitate the IOCL expression coding.

It is important to highlight what we mean for abstract logic. We put this way because the *IOCLAnalyzer* is an abstract component and it is not tied to any metamodel. All method implementations defined in this class have a common logic independently whether we change the underlying metamodel or not. An example is the type checking for an assignment expression. If we use the Kobra metamodel, we will be checking Kobra

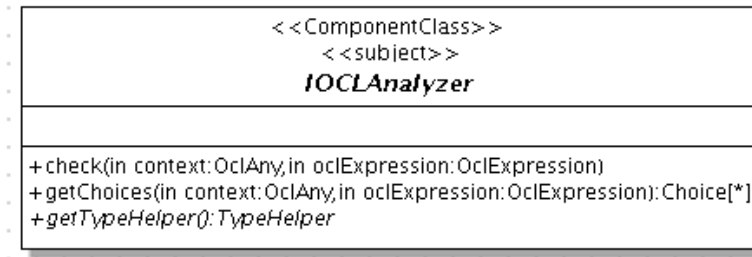


Figure 5.8: IOCLAnalyzer - Specification Structural Class Service

types. If we switch the metamodel for UML, we will be checking UML types. However both operations have same logic.

The metamodel configuration, i.e. the concrete *IOCLAnalyzer*, is done through a property file defined in the IOCLCompiler component. Inside this file we specify the *IOCLAnalyzer* subclass, which will handle with the specificities brought by the custom metamodel. The figure 5.9 illustrates the *Specification Structural Class Service* view of the *KobraAnalyzer*.

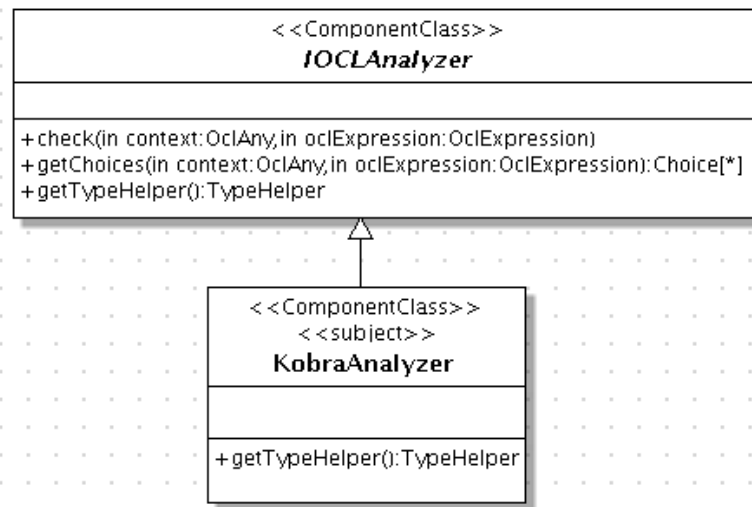


Figure 5.9: KobraAnalyzer - Specification Structural Class Service

The *TypeHelper* class presented in this view represents an interface with operations for dealing with the symbol table and for resolving types, operations and properties based on the passed context. This class is detailed in the *Specification Structural Class Type*, figure 5.10, of the component.

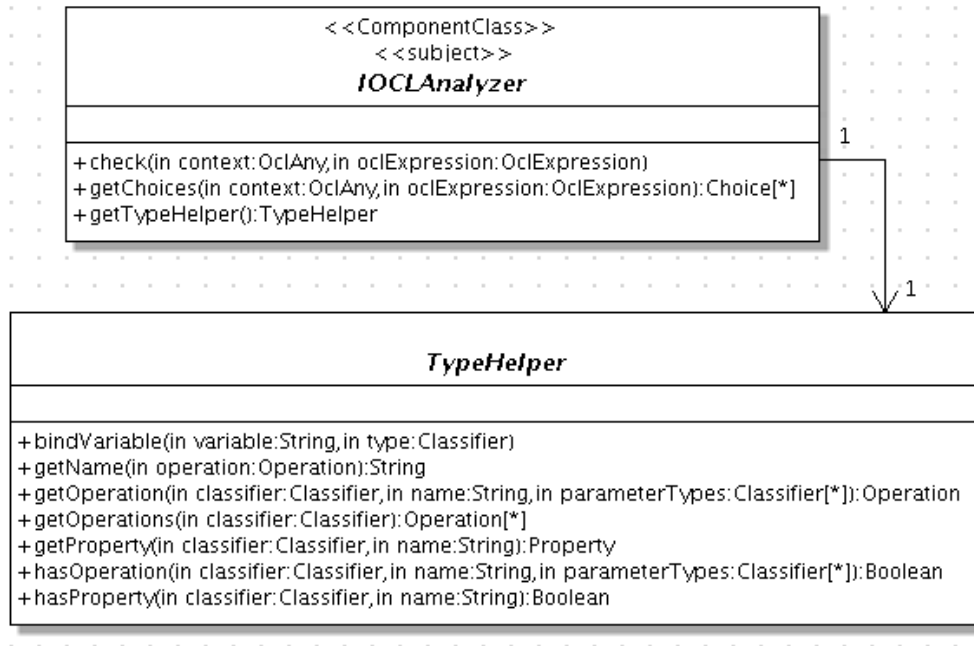


Figure 5.10: IOCLAnalyzer - Specification Structural Class Type

Finally, the *KobraTypeHelper* specializes all the abstract operations defined by the *TypeHelper*. This class is tied to the KobraA metamodel and it is responsible for performing the analysis of the KobraA classifiers passed as context of the *IOCL*Engine *parse()* operation. This is depicted in 5.9.

5.1.2 CodeGenerator Component

The *CodeGenerator* component is responsible for generating code for a specific target platform. The code it generates is determined by class that specializes the *Handler* abstract class. The *Handler* class defines several methods whose purpose is to generate target code for each type of expression. These methods are called by a *Visitor* class implementation, which traverses the entire expression tree structure and then call the *handle()* methods when appropriate. The *Handler* class implementation is configurable through a properties file called *ioclgenerator.properties*. The figure 5.12 shows the content of the property file.

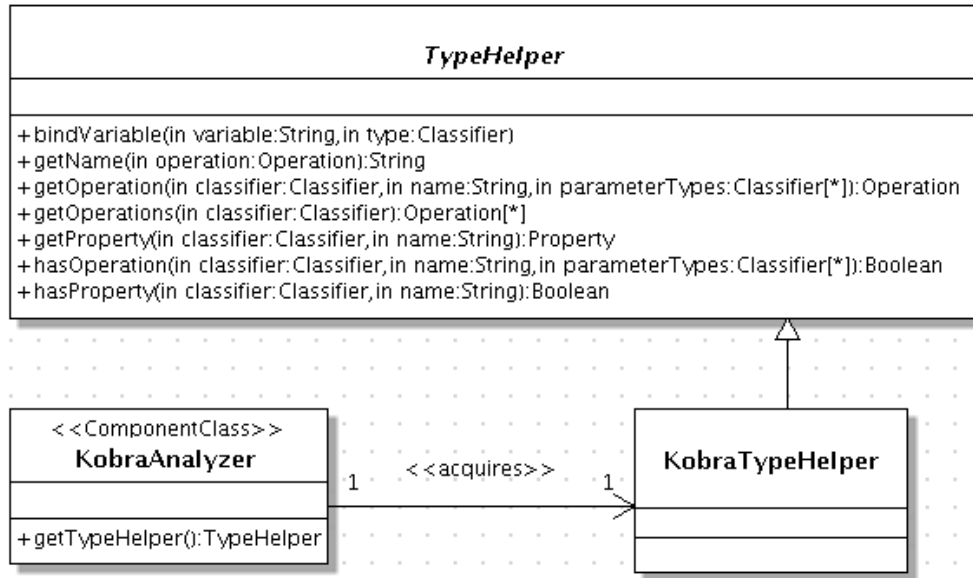


Figure 5.11: KobraAnalyzer - Specification Structural Class Type

```

#
# Code generator class
#
iocl.handler.class=org.orcas.ioclgenerator.java.JavaHandler

```

Figure 5.12: IOCLGenerator Configuration file

The *Specification Structural Class Service* 5.13 of the CodeGenerator shows its public method `generate()`. This method receives the `OCLExpression` AST properly parsed and checked by the `IOCLEngine` component and generates appropriate target code based on the handler class implementation as described before.

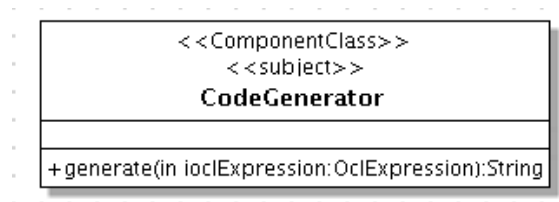


Figure 5.13: CodeGenerator - Specification Structural Class Service

The *Realization Structural Class Service* 5.14, shows the nesting of the CodeGenerator component with the Visitor component.

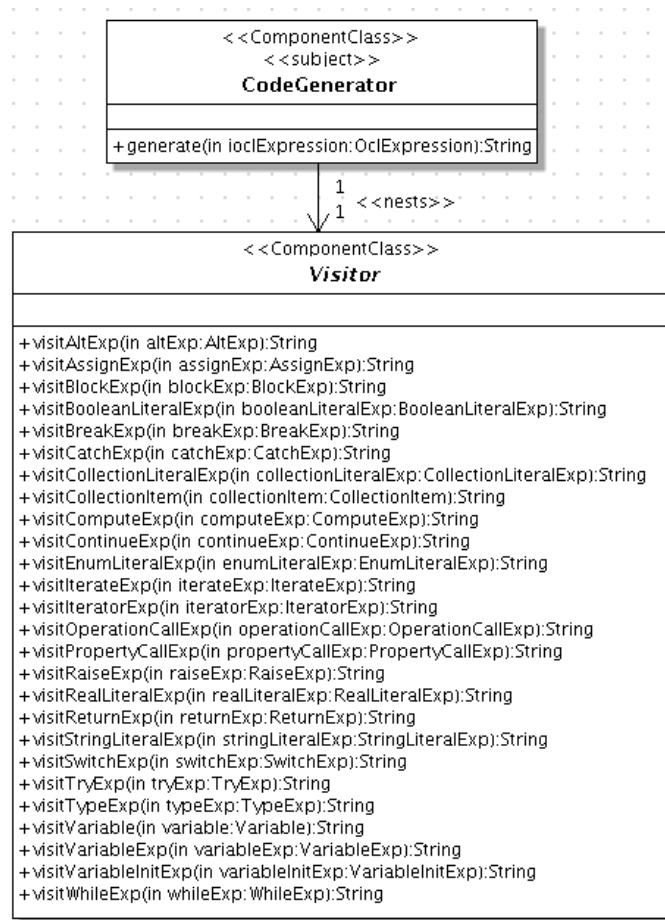


Figure 5.14: CodeGenerator - Realization Structural Class Service

As we described before, there is a clear separation in the API between the methods responsible for transversing the AST, which are all implemented in the Visitor component and the ones dealing with code generation, which are implemented by a the Handler component. The figure 5.15 illustrates the *Specification Structural Class Service* of the Visitor component.

We took this approach in order to isolate the logic involved in transversing the AST from the code generation capabilities. This way the customization to other target generator is easier to maintain.

The specialization of the *Handler* is illustrated in the Specification Structural Class Service of the *JavaHandler* component in the figure 5.16.

In addition, the *JavaHandler* component acquires the *TemplateEngine* component. All the *handle* methods presented in *JavaHandler* have direct calls to the *TemplateEngine* component. This last component deals with the templates defined for the Java platform to make even easier the code generation.

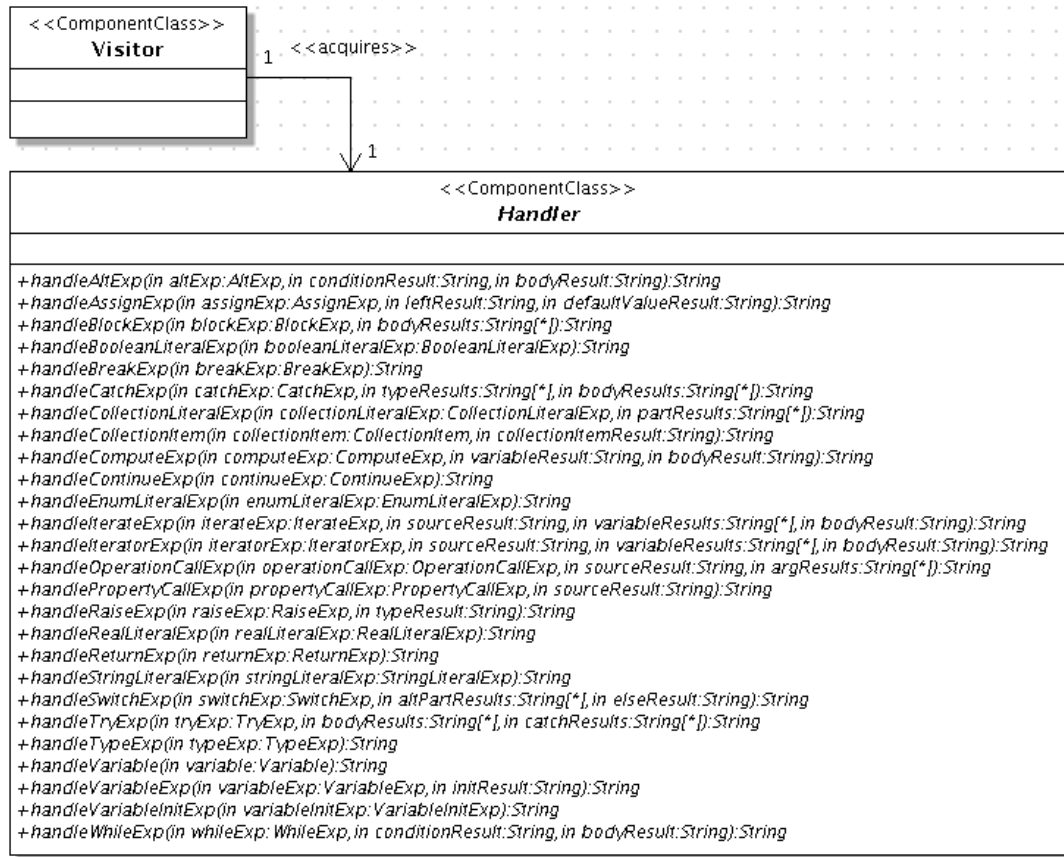


Figure 5.15: Visitor - Specification Structural Class Service

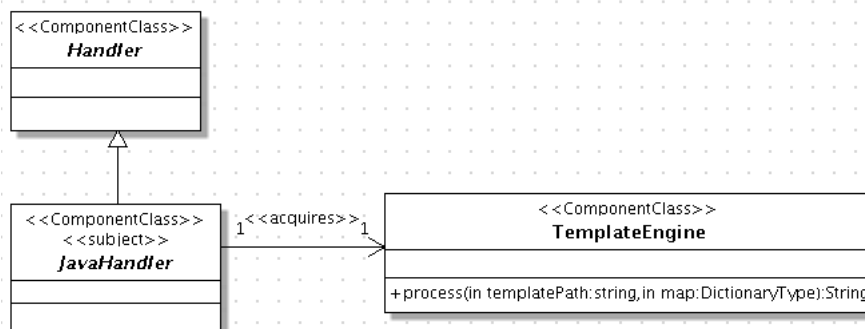


Figure 5.16: JavaHandler - Specification Structural Class Service

During the development of this component two templates engines for Java were analyzed: Velocity (Apache, 2011) and FreeMarker ² (FreeMarker, 2011). Both tools analyzed are general-purpose template engines with very similar characteristics. Thus, the selection of one of them was based on the evaluation of two factors: *expressivity*

²Velocity and FreeMarker were selected because they are the most used templates engines in Java projects.

power and ease of use.

- *Expressivity Power* - Both tool are similar in terms of expressivity. In FreeMarker although, the macros functions can be called recursively, which is not possible in Velocity. In general FreeMarker has more built-in functionalities when compared to the other, such as: possibility of *break* statements out of loops, detect if the current iteration is the last in the loop, several String and Lists manipulation utilities, etc. FreeMarker also allows using multiple namespace for variables, which is useful for structuring libraries a of macros. The negative factor is that the learning curve for FreeMarker is a little steeper than the one for Velocity.
- *Easy of use* - Several factors contribute to a tool be easy to use: ease of install, consistency of the API, and the quality of the documentation. The installation process of both tools are easy. The process involves only the addition of the engine dependency to the classpath. The difference of between the two is that velocity requires additional libraries that should also be present at the classpath in order to run properly. API documentation of both Velocity and FreeMarker is available in API doc format and they are equally informative. Both contains an overview of the languages and examples of use. In this point both tools are equivalent.

In sum, both tools very similar in terms of functionality. Velocity enjoys a larger use base, but FreeMarker is more sophisticated and faster ³, specially when parsing larger templates.

After taking this into consideration together with my previous experience with FreeMarker, the last was chosen to serve as a underlying technology of the *TemplateEngine* component.

For the development of this class we used the open source FreeMarker (FreeMarker, 2011) project. FreeMarker is a Java-based template engine used for any kind of text generation.

The use of templates during this work added a good level of isolation of the generator program from the syntax specificity of the target language. That way, target language constructs, such as the *if*, *while* or a variable definition, resides on templates. The only point of the parameterization is done by the DictionaryType object, from the Imperative OCL package, passed as argument for the TemplateEngine *process()* method.

An example of the use of templates is been shown in the figures 5.17 and 5.18. The first template generates the alternatives (i.e. boolean conditions) that will be used in order

³<http://www.javaworld.com/javaworld/jw-11-2007/jw-11-java-template-engines.html>

to generate the compiled switch expression. The switch expression template receives as context the list of the possible alternatives (*altPartResults*), that will be printed as *if* and *else if* in the Java target language, and the *elseResult* if there is one. The *compress* directive is a FreeMarker command and it is useful for removing superfluous white-space of the template.

```
<#compress>
({conditionResult}) {
    ${bodyResult}
}
</#compress>
```

Figure 5.17: *Alt* Expression Template

```
<#compress>
<#list altPartResults as altPartResult>
    <#if altPartResult_index == 0>
        if ${altPartResult}
    <#else>
        else if ${altPartResult}
    </#if>
</#list>
<#if elseResult?has_content>
    else {
        ${elseResult}
    }
</#if>
</#compress>
```

Figure 5.18: *Switch* Template

5.2 Tests

In order to ensure the correct functioning of the IOCL Engine, there were defined several unit tests for checking the parsing and semantic analysis operations. For the development of these unit test we used the JUnit Framework (JUnit, 2011).

JUnit provides a complete infrastructure that facilitate tests execution and results visualization. The figure 5.19 details one of the implemented tests used to check operation calls. The purpose of this test is to verify whether the Semantic Exceptions is being thrown when an expression refers to a nonexistent operation on a classifier or not.

We begin the test creating a component named *Test* that owns the *sayHello()* operation. After, we define the “self.sayHello();” expression and then the call the *parse()* method of the IOCL Engine component is executed. The method should run without exceptions

because the expression defines a call of an operation that exists, while the second execution of the *parse()* should raise an exception because the call is on an operation that does not exist.

All the implemented tests are also available at <https://github.com/marcellustavares/imperative-ocl>.

```
// imports omitted

public class OperationCallTest extends TestCase {

    public void testNonexistentOpCall() {
        ComponentClass component =
            getFactory().createComponentClass();

        component.setName("HelloWorld");

        Operation process = createOperation(
            "sayHello", VisibilityKind.PUBLIC, null, false);

        List<Operation> operations = businessFacade.getOwnedOperation();
        operations.add(process);

        String expression = "self.sayHello()";

        try {
            IOCLEngine.parse(businessFacade, expression);
        }
        catch (Exception e) {
            fail("Exception_should_not_be_raised.");
        }

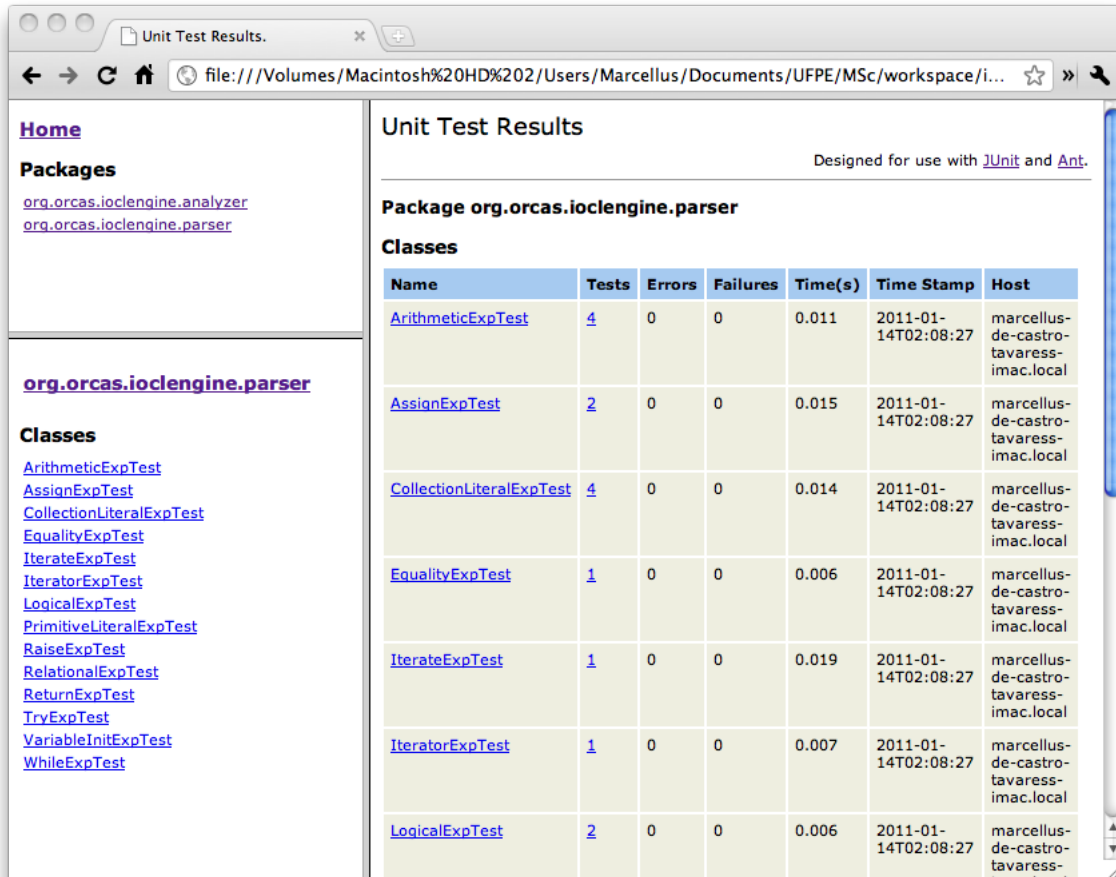
        expression = "self.nonExistentOperation()";

        try {
            IOCLEngine.parse(businessFacade, expression);

            fail("Semantic_exception_not_raised.");
        }
        catch (Exception e) {
            assertTrue(e instanceof SemanticException);
        }
    }
}
```

Figure 5.19: Operation Call Unit Test

The figure 5.20 exemplifies one of the reports generated by JUnit for the IOCLCompiler parser component.



Unit Test Results

Designed for use with [JUnit](#) and [Ant](#).

Package org.orcas.ioclengine.parser

Classes

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
ArithmeticExpTest	4	0	0	0.011	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
AssignExpTest	2	0	0	0.015	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
CollectionLiteralExpTest	4	0	0	0.014	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
EqualityExpTest	1	0	0	0.006	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
IterateExpTest	1	0	0	0.019	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
IteratorExpTest	1	0	0	0.007	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local
LogicalExpTest	2	0	0	0.006	2011-01-14T02:08:27	marcellus-de-castro-tavaress-imac.local

Figure 5.20: IOCLEngine Parser Test Report

5.3 Chapter Summary

This Chapter presented the main aspects of IOCLCompiler component. The architecture and the set of technologies employed during its construction were discussed. Next Chapter presents the evaluations performed to verify the tool's correctness and its helpfulness to the process of generating behavioral implementations.

6

IOCL Compiler Evaluations

In this chapter, the experiments performed are described in order to evaluate the IOCL Compiler tool. The experiments aim at validating the thesis that the code generated by the IOCL Compiler tool have same or better characteristics when compared to the handwritten code with respect to quality of the code.

This Chapter is organized as follows: Section 6.1 describes the process used in order to execute the experiments described in this chapter. Section 6.2 defines the goals, metrics and threats of the experiments and latter on this chapter, specifically on Section 6.4 and Section 6.5, the two experiments are discussed. Its subsections describe the experiments and the results obtained. Finally Section 6.6 draws the chapter summary.

6.1 Experiment Process

In order to perform an experiment, several tasks have to be done in certain order. In this Chapter we followed the process defined by (Wohlin *et al.* , 2001), which can be divided into the main activities: **Definition**, where the experiment is defined in terms of problem, objective and goals. **Planning**, where the design of the experiment is determined, the instrumentation is considered and the threats to the experiment are evaluated. **Operation**, in which measurements are collected, analyzed and evaluated in the analysis and interpretation. Finally, the results are presented and packaged in the **presentation and package**. The process is depicted in Figure 6.1

Next Section presents the definition step common to both of the experiments.

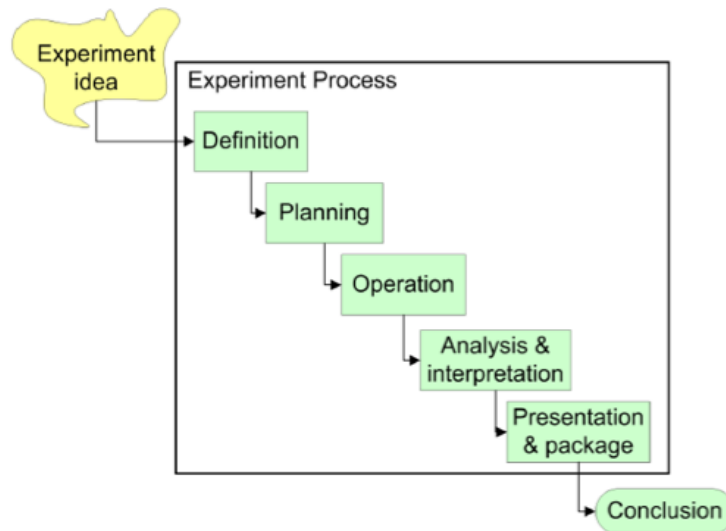


Figure 6.1: Experiment process (Wohlin *et al.* , 2001)

6.2 Definition

In the definition step, it is defined the objectives and goals of the experiment. To achieve it, (Wohlin *et al.* , 2001) follow the GQM (Goal-Question Metric) approach (Basili *et al.* , 1994). The GQM is based upon the assumption that for an organization to measure in a purposeful way, it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provides a framework for interpreting the data with respect to the stated goals.

Thus, following the GQM template, one goal was defined for this evaluation, further detailed and described by its respective questions and metrics.

Goal. The goal of both experiments is to to *analyze* the *IOCL Compiler tool* with the goal of *determine* if the tool promotes an *improvement of quality* of code when compared to a handwritten code from the point of view of *system analysts*.

Question. To achieve this goal, the following question were defined:

- Q₁. Is the quality of the generated artifacts better than to the manual code?

Metrics. After have the questions, we look for metrics that could provide evidences about the formulated questions. Thus for both experiments we have defined the following metrics for the question related to the purpose of the evaluation:

- *M₁. Instability* - According to Martin (Martin, 2002), what makes software to be unstable and hard to be reused is the dependencies among its modules. A change in one module, for example, would cause changes in many other different modules. The instability metric (*I*) pursues to estimate this characteristics of software. The instability is defined as: $I = Ce / (Ca + Ce)$, where *Ce* means *Afferent Couplings* or the number of classes outside this package that depend upon classes within this package and *Ca* means *Efferent Couplings* or the number of classes inside this package that depend upon classes outside this package. The instability metric has the range between 0 and 1. *I* = 0 indicates a maximally stable package and *I* = 1 indicates a maximally unstable package;
- *M₂. Cyclomatic complexity* - This metric, introduced by (McCabe, 1976) measures the structural complexity of a method. It is calculated from a graph that represents the execution flow of the program by the following rule: $M = E - N + P$ where *E* is the number of edges of the graph, *N* is the number of nodes of the graph and *P* is the number of connected components. For this metric, values ranging from 1 to 10 represents a low risk method, 11 to 20 a moderate risk, 21 to 50 a high risk and values greater than 50 represents highly unstable methods;
- *M₃. Maintainability Index* - It is a compound metric designed with primary aim to determine how easy it will be to maintain a particular body of code. The Maintainability index is defined as (Coleman *et al.* , 1994b): $MI = 171 - 5.2 \ln(aveV) - 0.23aveV(g') - 16.2 \ln(aveLOC) + (50 * \sin(\sqrt{2.46 * perCM}))$, where *aveV* is the average Halstead volume per module, *aveV(g')* is the average extended cyclomatic complexity per module, *aveLOC* is the average numbers of lines of code per module and *perCM* is the average percent of comments of comment lines per module. According to this metric modules with MI less than 65 are harder to be maintained, modules ranging between 65 to 85 have moderate maintainability and values greater than 85 and more: good maintainability;
- *M₄. % of successful unit tests* - This metrics calculates the number of successful unit tests divided by the total tests defined for the application. Generally unit tests are defined for each operation existing in the application. Therefore, this metric evidences the overall correctness of the software.

6.3 Hypothesis

The practice of science involves formulating and testing hypotheses, assertions that are falsifiable using a test of observed data (Wikipedia, 2011b).

Null hypotheses typically corresponds to assumptions that the researcher wants to reject. In this research, we used the concept of null hypothesis in order to validate the thesis set this work. Thus, the null hypotheses for this evaluation were based on the questions and metrics defined earlier and they were defined as:

- H_{0a} : $Instability_{(generated)} \geq Instability_{(manual)}$
- H_{0b} : $Cyclomatic\ complexity_{(generated)} \geq Cyclomatic\ complexity_{(manual)}$
- H_{0c} : $Maintainability\ Index_{(generated)} \leq Maintainability\ Index_{(manual)}$
- H_{0d} : $\% \text{ of successful unit tests}_{(generated)} \leq \% \text{ of successful unit tests}_{(manual)}$

Alternative Hypothesis is the hypothesis in favor of which the null hypothesis is rejected. In this study, the alternative hypothesis determines that the use of the tool produces benefits that justify its use. They were:

- H_{1a} : $Instability_{(generated)} \leq Instability_{(manual)}$
- H_{1b} : $Cyclomatic\ complexity_{(generated)} \leq Cyclomatic\ complexity_{(manual)}$
- H_{1c} : $Maintainability\ Index_{(generated)} \geq Maintainability\ Index_{(manual)}$
- H_{1d} : $\% \text{ of successful unit tests}_{(generated)} \geq \% \text{ of successful unit tests}_{(manual)}$

6.4 First Experiment - Web Agency

The first experiment was conducted at the university lab from October, 2010 to December, 2010. However, further minors refinements were done in the model until April, 2011. The domain of this experiment involves a typical Information System with business rules and persistence requirements. Specifically, the project was developed to simulate an on-line credit agency (Web Agency). We considered this domain representative because it contains requirements present on the majority of web applications and also because it covers several variety of IOCL expressions.

The Web Agency is a illustrative example of an application that simulates an online credit agency. It was also conceived to be used as experiment to the related research to this work (Oliveira, 2011). The Web Agency possesses the the following requirements: a) **Add, list and remove clients**; b) **Request Loans** based on client requests and profiles; c) **Add and remove loans**; d) **List loans** taken by a specific client.

Most of the requirements involved in the Web Agency are related to persistence operations, one of the exceptions is the *request loan* method. In this method, the client requests a loan value and the number of parcels he wishes to pay the debt. The application processes this request and responds to it according the following business rule: the loan request is accepted if the value of this loan plus the value of the previous client's loans does not exceed a certain limit. The limit is derived by the salary and the social class of the client (calculated having as base the salary and the number of dependents of the client). If the loans request is according to this rule the system responds to the request by allowing the client borrow that sum, otherwise the system proposes an alternative loan value that is in accordance to the previous rule.

6.4.1 Planning

The experiment was planned to be accomplished at the university lab by the Thiago Oliveira (Oliveira, 2011) and the author of this dissertation. Initially, a complete PIM specification of the Web Agency was defined with the purpose of being used as input to the WAKAME Generator and also to serve as reference to the manual implementation.

Based on the generated applications, the data extraction was conducted, followed by the interpretation of the results from the numbers obtained from the manual and automatic implementation.

The collection of metrics M_1 , M_2 and M_3 was performed with the aid of different tools. Because the artifacts were written in Java, we used the Eclipse Metrics¹ (cyclomatic complexity and instability) and JHawk² (Maintainability Index). Both tools work with source code in Java.

The test reports used to calculate the metric M_4 were obtained by the testing framework JUnit³. The tests simulate possible scenarios of the application services and although they are unitary the fact they were applied on the top level component gives them characteristics of a system test.

¹<http://metrics2.sourceforge.net/>

²<http://www.virtualmachinery.com/>

³<http://junit.org/>

Also it should be mentioned that the participation of the author of the dissertation in the experiment, may have influenced the results, being a threat to the validity. Yet, it was considered that the results are valid, since many of them were obtained from objective metrics, which do not suffer the influence of the researcher.

6.4.2 Operation

The experiment ran during the part of M.Sc. course at Federal University of Pernambuco and the execution consisted into three parts:

1. PIM specification of the Web Agency;
2. Manual implementation of the application from PIM;
3. Automatic code generation from PIM.

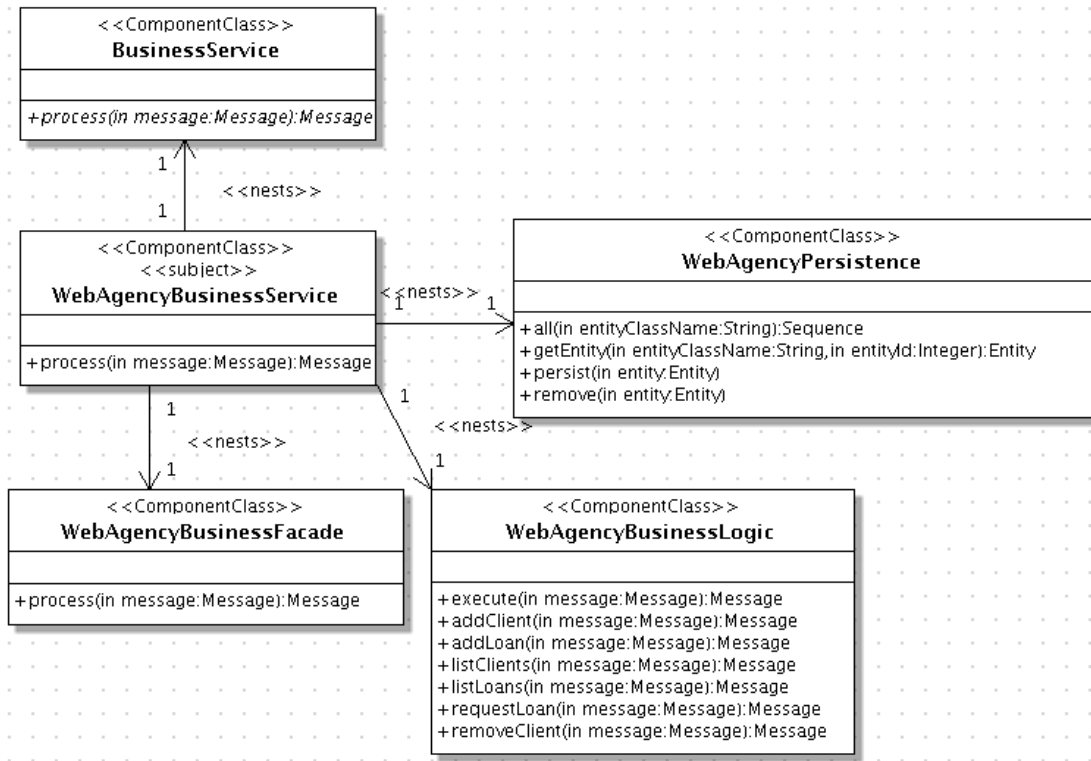
Web Agency - PIM

In this section we detail the most relevant elements of the Web Agency PIM. The complete PIM of this project is described in Appendix B and it is also available at the WAKAME under a project with the same name and it can be accessed via the following web address: <http://wakametool.appspot.com/>.

The Web Agency is an instance of KISF, and by so, it is decomposed according the MVC pattern and the client/server architecture used in KISF. 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 Figure 6.2, depicts the *Realization Structural Service View* of the *WebAgency-BusinessService* component. In the realization, we can observe the projects requirements being modeled as operations of the *WebAgencyBusinessLogic*, component that specializes the KISF *BusinessLogic* component. Two more components complete this view: *WebAgencyBusinessFacade* and *WebAgencyPersistence*.

The first one, as already discussed in Figure 3.2.1 has the responsibility of redirecting the client's requests to the appropriate *BusinessLogic* classifier. The Figure 6.3 details the specification of the *process()* method in Imperative OCL.

Figure 6.2: *WebAgencyBusinessService* - Realization Structural Class Service

```

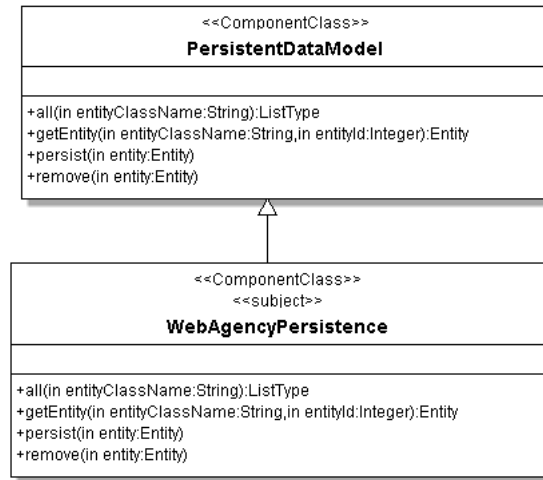
context : WebAgencyBusinessFacade :: process (message : Message) : Message
body:
do {
    var classifier:String := message.getClassifier();

    try {
        if (classifier = 'webAgencyBusinessLogic') {
            return self.webAgencyBusinessLogic.execute(message);
        }
    }
    except (Exception) {
        raise 'Action execution raised an error.';
    }
}

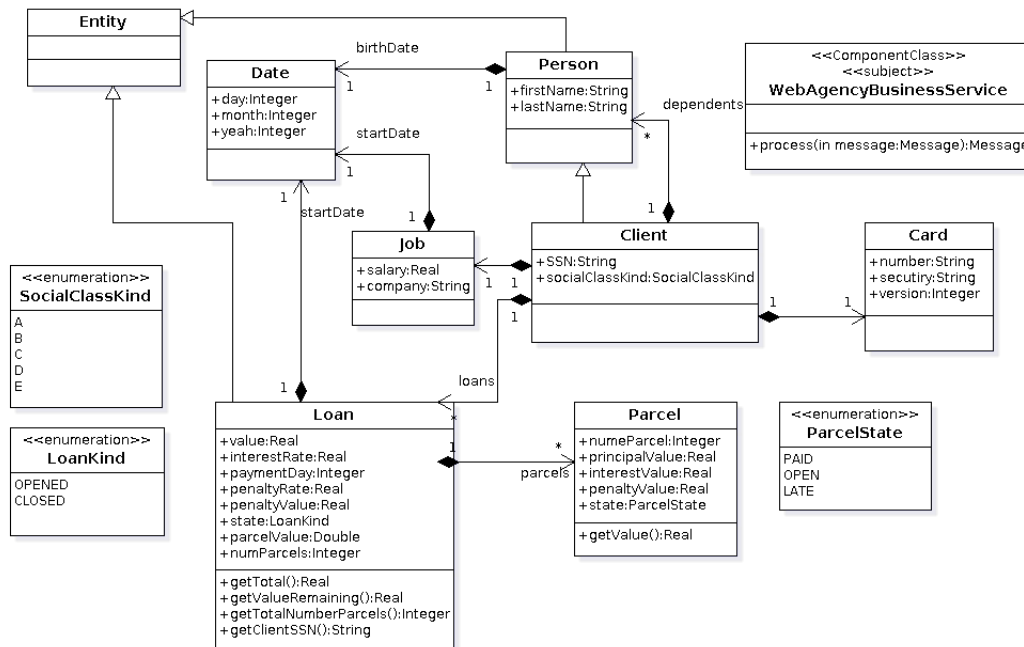
```

Figure 6.3: *WebAgencyBusinessFacade* - Specification Operational Service View

The *WebAgencyPersistence* component in particular, illustrated in the Figure 6.4 is acquired by the *WebAgencyBusinessLogic* because some of its operations requires persistence capabilities, i.e. AddClient, AddLoan. This is because the *WebAgencyPersistence* is a specialization of the KISF *PersistentDataModel* component, and by so, it inherits all datastore operations that will be used by the *BusinessLogics* components.

Figure 6.4: *WebAgencyPersistence* - Specification Structural Service

All the classes and entities handled by the *WebAgencyBusinessService* as its sub-components are depicted in Figure 6.5. The figure shows domain concepts existing the credit agency context such as *Client* and *Loan*. Classes in the model that specializes the *Entity* class of KISF will have persistent capabilities granted automatically. That way, the modeler can pass one of the classes to be stored in the database through the *WebAgencyPersistence*.

Figure 6.5: *WebAgencyBusinessService* - Realization Structural Type

In order to illustrate the business rules of the WebAgency, we show in the figure Figure 6.6 the specification of the *Appraiser* operations. In sum, the code shows the method *proposeLoan* definition. The operation response follows the requirement described in Section 6.4. The *calculateMaxParcelValue()* and *getMaxBorrowingBySocialClass* are auxiliary operations that makes part of the algorithm.

```
context: Appraiser :: calculateMaxParcelValue(c: Client): Real
body: do {

    var max: Real := client.getSalary() *
        self.getMaxBorrowingBySocialClass(c.getSocialClass());

    var totalPending: Real = 0.0;

    (client.loans)->forEach(loan: Loan) {
        totalPending := totalPending + loan.parcelValue;
    }

    return (max - totalPending);
}
context: Appraiser :: proposeLoan(c: Client, numParcels: Integer, value: Real): Loan
body: do {
    var maxParcelValue: Real := self.calculateMaxParcelValue(c);

    var requestedParcelValue := value / numParcels;
    var loan: Loan = new Loan()

    if (requestedParcelValue <= maxParcelValue) {
        loan.setParcelValue(requestedParcelValue);
        loan.setNumParcels(numParcels);
        return loan;
    }
    else {
        loan.setParcelValue(maxParcelValue);
        loan.setNumParcels(numParcels);
        return loan;
    }
}
context: Appraiser :: getMaxBorrowingBySocialClass(socialClass: SocialClassKind): Real
body: do {
    if (socialClass = SocialClassKind::A) {
        return 0.5;
    }
    elif (socialClass = SocialClassKind::B) {
        return 0.4;
    }
    elif (socialClass = SocialClassKind::C) {
        return 0.3;
    }
    elif (socialClass = SocialClassKind::D) {
        return 0.2;
    }
    else {
        return 0.1;
    }
}
```

Figure 6.6: Appraiser - Specification Operational Service View

The *calculateMaxParcelValue()* returns the maximum parcel value of the loan the client is allowed to have. In this case, the parcel cannot exceed a maximum value that is calculated based on the client's salary and its social class, represented in the model as *SocialClassKind*. Subjects that are classified in the "A" social class, for example, can only compromise 50% of its salary with debts.

The remaining views and operations are specified in the Appendix B.

Web Agency - Manual Implementation

After the previous step and taking the Web Agency PIM as reference, we proceeded with the manual implementation. This task also was executed by the Thiago Oliveira and the author of this dissertation.

For this phase, we have used the same target language (Java) for implementation and we also used the same technologies (JPA) for implementing the persistence layer. It is also valid to mention the application GUI was also implemented during this phase.

Several unit tests were also manually written in order to test the correctness of the Web Agency. Each test checks whether the operation is valid or not. The criteria used to evaluate the validity on a non-persistence operation was based on the PIM specification.

For the operations related to the persistence of the entities of the model, the unit tests were based on the following assumption:

- **Add operation** - For a specific entity, the size of its stored list after the execution of the operation is equal to the value the list size before the operation was executed **plus** one.
- **Remove operation** - For a specific entity, the size of its stored list after the execution of the operation is equal to the value the list size before the operation was executed **minus** one.
- **List operation** - For a specific entity, the size of list returned by the datastore is greater if an **add operation** is executed.

With that in mind, the Table 6.1 summarize tests results:

Operation	Success	Failure
Add Client	x	
Add Loan	x	
List Clients	x	
List Loans	x	
Request Loan	x	
Remove Client	x	

Table 6.1: Web Agency (Manual) - Unit Tests

Also, several quality metrics were collected after the implementation, the Table 6.2 lists the results for the metrics described in Section 6.1.

Metric	Value
Instability	0.37
Cyclomatic Complexity	1.22
Maintainability Index	145.26

Table 6.2: Web Agency (Manual) - Metrics

Web Agency - Automatic Generation

After the PIM specification, the SUM model, serialized as a XMI file, was the passed as input to the WAKAME Code Generator for the automatic generation of the application. Within the WAKAME Code Generator, the Structural Generator (Oliveira, 2011) works is parallel with the IOCL Compiler in order to generate all parts of the applications.

For each component, a Java interface and an interface realization is created by the Structural Generator. The figures 6.7 and 6.8 details the specification and realization code of the RequestLoan component respectively. All the structural features such as class definition, attributes definition and operation definition are generated by the Structural Generator. The IOCL Generator is used in the meantime to generate the **operation implementations** and the **body of the aspects**, which are created when a pre or post condition is present in the model.

In this model, the IOCL compiler was used to generate various types of Java expressions. In this section we illustrate compiled codes of expressions showed in Section 6.4.2. The Figure 6.9 for example, illustrates the compiled expressions of the Appraiser component.

```
package webagencybusinessservice.nests.requestloan;

public interface RequestLoan extends KComponent,
webagencybusinessservice.nests.businessservice.nests.businesslogic.BusinessLogic {
    public webagencybusinessservice.nests.businessservice.Message execute(
        webagencybusinessservice.nests.businessservice.Message message)
        throws Exception;
}
```

Figure 6.7: RequestLoan - Specification

```
// imports omitted
@Component
public class RequestLoanImpl extends
webagencybusinessservice.nests.businessservice.nests.businesslogic.impl.BusinessLogicImpl
    implements webagencybusinessservice.nests.requestloan.RequestLoan {
    @Inject
    @acquires
    public webagencybusinessservice.nests.webagencypersistence.WebAgencyPersistence
webAgencyPersistence;
    @Inject
    @nests
    public webagencybusinessservice.nests.requestloan.nests.appraiser.Appraiser
appraiser;

    public webagencybusinessservice.nests.businessservice.Message execute(
        webagencybusinessservice.nests.businessservice.Message message)
        throws Exception {

        webagencybusinessservice.Client client =
            (webagencybusinessservice.Client)message.getAttribute(
                "client");
        Integer numParcels = (Integer)message.getAttribute("numParcels");
        Double value = (Double)message.getAttribute("value");
        webagencybusinessservice.Loan loan = this.appraiser.proposeLoan(
            client, numParcels, value);
        message.setMessageKind(
            webagencybusinessservice.nests.businessservice.MessageKind.SUCCESS);
        message.setAttribute("result", loan);

        return message;
    }
}
```

Figure 6.8: RequestLoan - Realization

Runtime constraint checks are also generated from pre and post condition expressions. The Figure 6.10 shows an AspectJ code created from a pre condition applied to the *calculateSocialClass()* operation in order to guarantee non negative integers for the *dependents* argument.


```

public java.lang.Double calculateMaxParcelValue(
    webagencybusinessservice.Client c) {

    Double max = (c.getJob().getSalary() *
    this.getMaxBorrowingBySocialClass(c.getSocialClass()));
    Double totalPending = 0.0;
    webagencybusinessservice.Loan loan = null;

    for (java.util.Iterator<webagencybusinessservice.Loan> it = c.loans.iterator();
        it.hasNext();) {
        loan = it.next();

        totalPending = (totalPending + loan.parcelValue);
    }

    return (max - totalPending);
}

public webagencybusinessservice.Loan proposeLoan(
    webagencybusinessservice.Client c, java.lang.Integer numParcels,
    java.lang.Double value) {

    Double maxParcelValue = this.calculateMaxParcelValue(c);
    Double requestedParcelValue = (value / numParcels);

    webagencybusinessservice.Loan loan = new webagencybusinessservice.Loan();

    if ((requestedParcelValue <= maxParcelValue)) {
        loan.setParcelValue(requestedParcelValue);
        loan.setNumParcels(numParcels);
        return loan;
    }
    else {
        loan.setParcelValue(maxParcelValue);
        loan.setNumParcels(numParcels);
        return loan;
    }
}

public java.lang.Double getMaxBorrowingBySocialClass(
    webagencybusinessservice.SocialClassKind socialClass) {

    if ((socialClass.equals(webagencybusinessservice.SocialClassKind.A))) {
        return 0.5;
    }
    else if ((socialClass.equals(webagencybusinessservice.SocialClassKind.B))) {
        return 0.4;
    }
    else if ((socialClass.equals(webagencybusinessservice.SocialClassKind.C))) {
        return 0.3;
    }
    else if ((socialClass.equals(webagencybusinessservice.SocialClassKind.D))) {
        return 0.2;
    }
    else {
        return 0.1;
    }
}

```

Figure 6.9: Appraiser - Compiled Expressions

After generation, the same unit tests implemented during the manual step were also applied for testing the generated artifacts. The Table 6.3 lists the results:

```

package aspects.constraints.webagencybusinessservice.nests.addclient;

// imports omitted

public privileged aspect AddClientCheck{

pointcut calculateSocialClass():call(
    public webagencybusinessservice.SocialClassKind
webagencybusinessservice.nests.addclient.impl.AddClientImpl.calculateSocialClass(
    java.lang.Integer, java.lang.Double));
before(webagencybusinessservice.nests.addclient.impl.AddClientImpl addClient,
    java.lang.Integer dependents, java.lang.Double salary)
: target(addClient) && calculateSocialClass() && args(dependents, salary){
    Boolean hasViolations = !(((Integer)message.getAttribute('dependents')) > 0);

    if(hasViolations) throw new ViolationPreException();
}

// other methods omitted
}

```

Figure 6.10: AddClient - Pre condition check

Operation	Success	Failure
Add Client	x	
Add Loan	x	
List Clients	x	
List Loans	x	
Request Loan	x	
Remove Client	x	

Table 6.3: Web Agency (Generated) - Unit Tests

The metrics were also collected after the automatic generation, the Table 6.4 summarizes the results obtained.

Metric	Value
Instability	0.35
Cyclomatic Complexity	1.17
Maintainability Index	169.3

Table 6.4: Web Agency - Metrics

6.4.3 Analysis and Interpretation

The Figures 6.11, 6.12 and 6.13 depicts the data obtained from Eclipse Metrics and JHawk categorized by the nature of the source code. It is shown the values of metrics related to the quality of the system, such as: instability, cyclomatic complexity and maintainability

index obtained from the artifacts built *with* and *without* automatic generation.

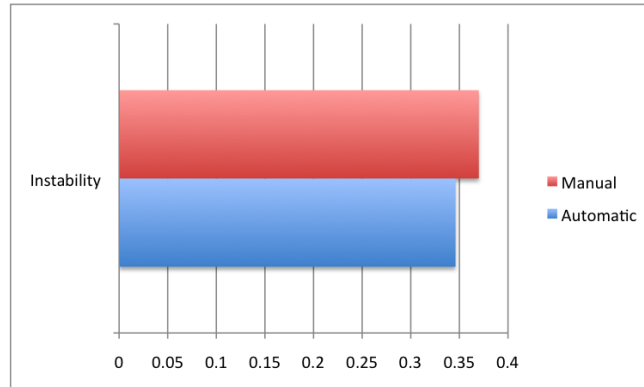


Figure 6.11: Web Agency - *Instability* metric comparison

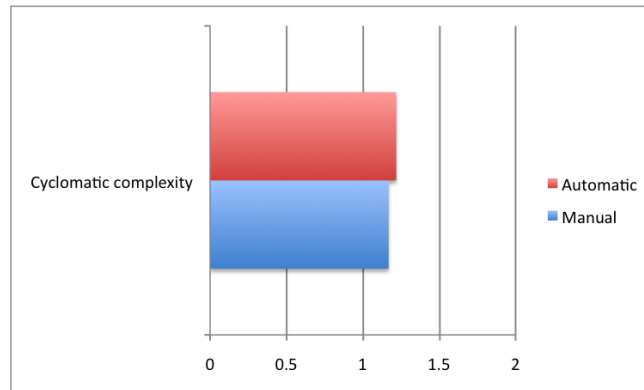


Figure 6.12: Web Agency - *Cyclomatic Complexity* metric comparison

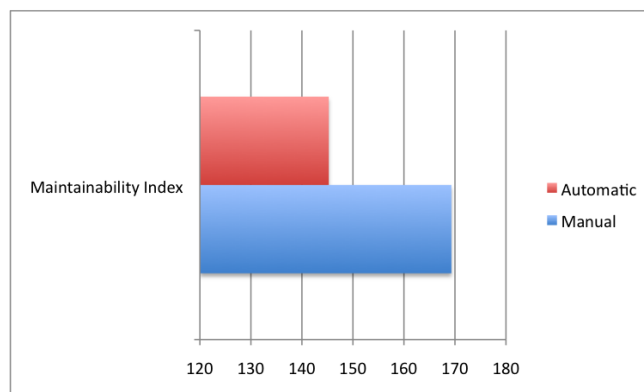


Figure 6.13: Web Agency - *Maintainability Index* metric comparison

It was verified that there was no significant changes on metrics of instability and cyclomatic complexity. However they appear slightly lower when the automatic genera-

tion is used. Similarly, the maintainability index number are close, with the index being a little greater when automatic generation is used instead of manual implementation.

The lack of significative changes in the numbers can be explained by two reasons: 1) the manual implementation of the web agency was strictly written according the PIM specification; 2) the same knowledge used to build the IOCL Compiler tool was also used to write the manual code. Even so, the numbers **rejects** the null hypothesis that the tool degrades the quality of the system.

The difference of the numbers though, can be explained by the good programming practices contained into the code the generator such as: patterns, source formatting and the use utilities libraries to perform similar tasks. The values, however, pointed another benefit brought by the generator: the **quality** of the code is **consistent** through all the code base.

Finally, the Table 6.5 lists the comparison of the tests results of both applications. It is showed that the two application implementations passed in all tests, which was expected due the same level of experience of the tool implementors and the experiment executors.

Operation	Manual Impl.	Automatic Impl.
Add Client	Success	Success
Add Loan	Success	Success
List Clients	Success	Success
List Loans	Success	Success
Request Loan	Success	Success
Remove Client	Success	Success

Table 6.5: Web Agency - Unit tests comparison

The Web Agency experiment **confirmed** our thesis that the **IOCL compiler maintains the same or improves the quality characteristics when compared to the hand-written code**. However, it was experienced that for simple system may not be worthwhile the extra work of building the complete PIM specification but if the modeling is a mandatory step, then the compiler could be used without compromising the quality of the system.

6.5 Second Experiment - CHROME

The second experiment was executed from March, 2011 to April, 2011. The experiment involved the automatic generation of the CHROME (Constraint Handling Rule Online Model-driven Engine) inference engine (Vitorino, 2009). CHROME is a CHR^V engine

written in Java with a complete search algorithm (e.g. the conflict-directed back jumping algorithm) and it is the first rule-based engine that integrates production rules, rewrite rules and Constraint Logic Programming (CLP).

Constraint Handling Rules (CHR) is a high-level programming language based on multi-headed rewrite rules. CHR was originally designed by (Frühwirth, 1994) for the special purpose of adding user-defined constraint solvers to a host-language.

That said, the CHROME, in the context of this dissertation, has the purpose of evaluate IOCL Compiler tool with respect to the quality of the generated artifacts when it is used for generating complex algorithms.

Also, it is worth mentioning that this experiment is not an instance of KISF and a developed version of this project already exists in Java. Another important characteristic for its choice as experiment is that all of its code is self-contained, i.e. there is no dependencies to third-party libraries.

6.5.1 Planning

The experiment was planned to be conducted by Armando Gonçalves, M.Sc. student in Software Engineering from Federal University of Pernambuco, also a member of the ORCAS group. In his master's project, he plans to extend the CHROME project. As group member, he has familiarity with WAKAME tool, OCL and Kobra2 method. To run this experiment, we also provided some training about Imperative OCL aiming Gonçalves have a better understanding on language and its constructs.

After this step, the activities assigned to Gonçalves were:

1. Definition of the CHROME PIM in the WAKAME tool;
2. Execution of the WAKAME Code Generator;
3. Execution of manual tests, both in the manual and the generated version of CHROME;
4. Collect the metrics defined in Section 6.2 of both application versions using Eclipse Metrics and JHawk tool.

Some threats to validity and execution of this experiment were identified, such as: 1) Difficulty in performing tests in CHROME, threat identified by Gonçalves; 2) Lack of knowledge of the executor of the experiment in IOCL; 3) Time restrictions to complete the experiment.

6.5.2 Operation

The first step for achieving this second experiment were the training given to Gonçalves about topics including WAKAME, IOCL and WAKAME Code Generator tool. After that, all necessary tools for the experiment execution were provided to the executor. The first activity performed was the creation of CHROME PIM, that will be briefly described in the Section 6.5.2.

CHROME PIM

The CHROME PIM is fully described in (Vitorino, 2009) thesis and it was used as reference to specify the CHROME in the WAKAME⁴ tool. Despite the existence of the PIM there was no possible way of reuse the existing one in WAKAME due incompatibility of the files formats. Thus the PIM definition had to be created from scratch.

The Figure 6.14, depicts the *Realization Structural Services View* of the top-level component.

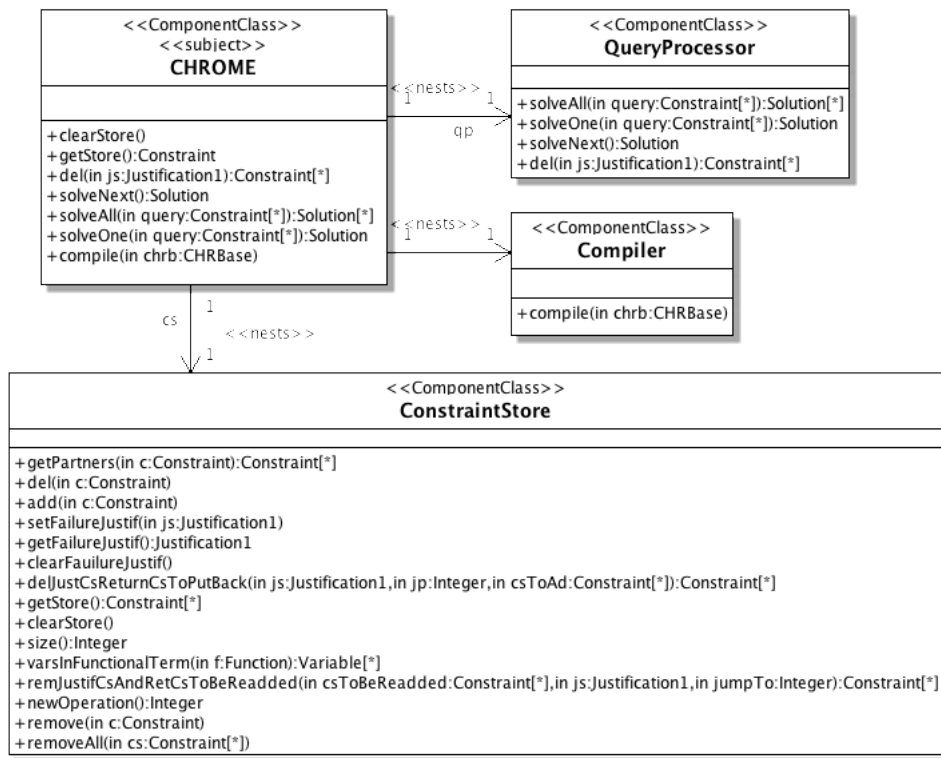


Figure 6.14: CHROME top-level component

⁴<http://wakametool.appspot.com>

The top-level CHROME component encapsulates all sub-components that compose the CHROME environment. Its interface provides methods to compile a rule base, to solve a query (displaying one or more solutions for such query), to adapt a solution when a given set of justified constraints is deleted and to clear the constraint store for processing a new query.

The main component for the CHROME run-time is the QueryProcessor. Its implements several operations defined in CHROME top-level, such as: solve query and clear the constraint store.

The *Compiler* component is a pipe-line of 4 ATL(ATL, 2009) transformations. The Specification of the component compiler provides only one operation, namely *compile()*, that writes a Java file operationally equivalent to the CHR^V base.

Finally, the ConstraintStore component provides the data structures to store constraints either processed or generated by rules.

The QueryProcessor component is composed of four nested subcomponents, depicted in Figure 6.14: Entailment, FiredRules, PropagationHistory and CDBJSearch.

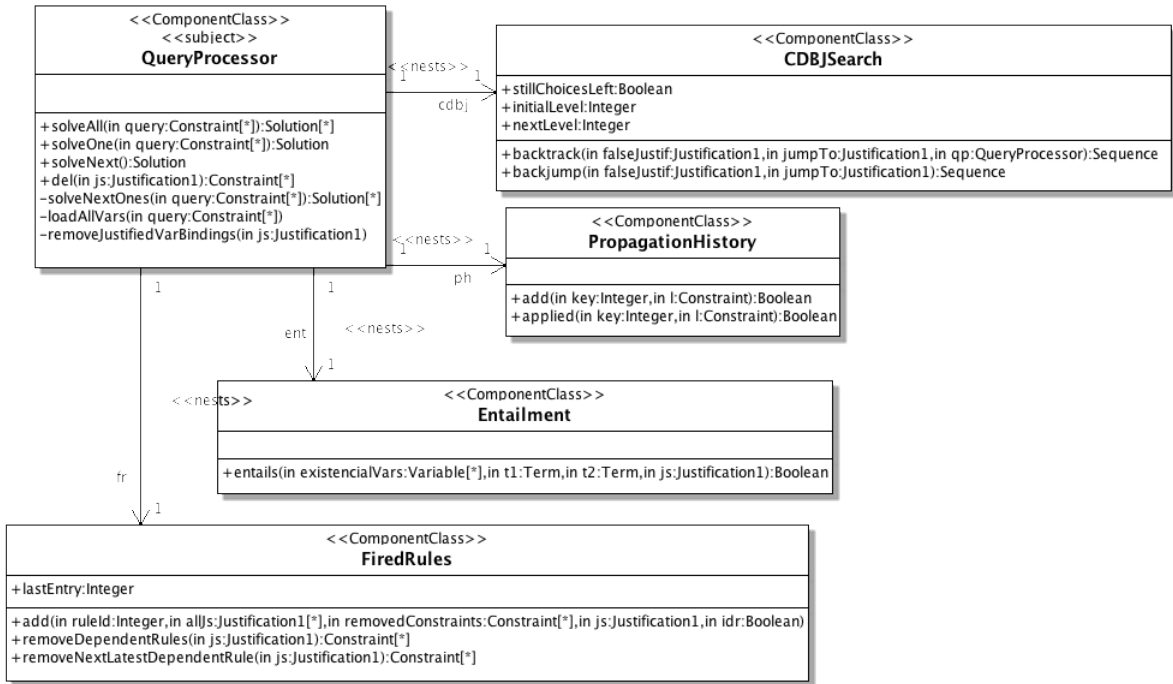


Figure 6.15: QueryProcessor - Realization Structural Class Service

The *Entailment* component encapsulates entailment services that is used every time a guard must be checked. It takes as parameters a set of existential variables (the ones that may be bound during the procedure) plus two terms instances (Abdennadher, 2001). The

goal of this component is to check if these two terms are syntactically equivalent given the current state of all variable bindings in the constraint store. The Figure 6.16 depicts its main operation.

```

context Entailment::entails(
localVars:Sequence(Variable), t1:Term, t2:Term, js:Justification):Boolean
body:
do {
  var j1:Justification := new Justification();
  var j2:Justification := new Justification();
  var d1:Term := self.deref(t1, j1);
  var d2:Term := self.deref(t2, j2);
  js.mergeJustification(j1);
  js.mergeJustification(j2);
  if ((d1 = d2) and (not (d1.ocIsKindOf(Function)
    and d2.ocIsKindOf(Function)))) {
    return true;
  }
  if (d1.ocIsKindOf(Variable)) {
    if ((self.isVarLocal(t1.ocAsType(Variable), localVars)) and
      (not (d1.ocAsType(Variable).value.ocIsUndefined())))) {
      self.bind(d1.ocAsType(Variable), t2, js);
      return true;
    }
  }
  if (d2.ocIsKindOf(Variable)) {
    if ((self.isVarLocal(t2.ocAsType(Variable), localVars)) and
      (not (d2.ocAsType(Variable).value.ocIsUndefined())))) {
      self.bind(d2.ocAsType(Variable), t1, js);
      return true;
    }
  }
  if ((d1.ocIsKindOf(Function)) and (d2.ocIsKindOf(Function))) {
    if ( (d1.ocAsType(Function).args->size() =
      d2.ocAsType(Function).args->size()) and (d1.name = d2.name)) {
      var i:Integer := 0;
      var r:Boolean := true;
      while ((i < d1.ocAsType(Function).args->size()) and r) {
        var localTerm1 := d1.ocAsType(Function).args->at(i);
        var localTerm2 := d2.ocAsType(Function).args->at(i);
        r := r and self.entails(
          localVars, localTerm1, localTerm2, js);
        i := i + 1;
      }
      return r;
    }
    else {
      return false;
    }
  }
}

```

Figure 6.16: Entailment - Realization Operational View

The main purpose of *FiredRules* component is to store the fired rules history in order to undo any rules in case of a backtracking event. The Component keeps internally a collection of entries, each of which stores the rule Id from a given fired rule, the corresponding removed constraints, if any, that must be re-added in case of backtracking,

the justification of the constraints used to fire the rule and whether the rule is either disjunctive or composed only by conjunctions. The Figure 6.17 depicts its operational view.

```

context FiredRules::add(ruleId:Integer , allJs:Justification ,
removedConstraints:Sequence(Constraint), idr:Boolean)
body:
do {
  var element:FiredRulesEntry := new FiredRulesEntry(
    ruleId , allJs , removedConstraints , idr);
  contents := contents->including(element);
  self.lastEntry := self.lastEntry + 1;
}
context FiredRules::removeNextLatestDependentRule(
justification:Justification):FiredRulesEntry
body:
do {
  var i:Integer := self.lastEntry;
  while ( i > 0 ) {
    var element:FiredRulesEntry := contents->at(i);
    if (element.js.isJustifiedBy(js)) {
      contents := contents->excluding(element);
      self.lastEntry := self.lastEntry -1;
      return element;
    }
    i := i-1;
  }
  return null;
}

context FiredRules::removeDependentRules(justification:Justification):FiredRulesEntry
body:
do {
  var i:Integer := self.lastEntry;
  var element:FiredRulesEntry = new FiredRulesEntry();
  var lastElement:FiredRulesEntry := new FiredRulesEntry();
  var result:Sequence(Constraint) := Sequence{};
  while ( i > 0 ) {
    element := contents->at(i);
    if (element.js.isJustifiedBy(js)) {
      element := contents->excluding(element);
      self.lastEntry := self.lastEntry - 1;
      result := result->union(element.constraintsToBeReadded);
      lastElement := element;
    }
    i := i -1;
  }
  if (lastElement.isADisjunctiveRule) {
    contents := contents->including(lastElement);
    (lastElement.removedConstraints)->forEach(c:Constraint) {
      result := result->excluding(c);
    }
    self.lastEntry := self.lastEntry +1;
  }
  return result;
}

```

Figure 6.17: Fired Rules - Realization Operational View

Every time a combination of keep constraints fires a rule they have to be stored in order to assure that future applications of the same rule will not be used the exact same

combination of keep constraints. The reason for that is to avoid trivial non-termination of propagation rules.

The *PropagationHistory* component tracks each combination of keep constraints that were used to fire a given rule.

Finally, the *CDBJSearch* component provides the functionality to perform either chronological backtracking or backjumping to handle disjunctive bodies.

The remaining views and operations are specified on the Appendix C.

CHROME - Manual implementation

As stated, the CHROME already had an implemented version developed by (Vitorino, 2009). Currently CHROME is being extended to its second version by Gonçalves, executor of this experiment. The first version however, has two important characteristic that needs to be hightailed.

The first stems from the fact that despite the built PIM follows the Kobra2 method, the first implementation version of CHROME did not followed completely recommended implementation patterns for dealing with components (Fowler, 2002). There is no distinction at code level between the component specification (interfaces) and component realizations, for example. The second distinction is the lack of test for the implemented application and the difficulty of the experimenter to create a set of tests that were useful to cover the main features of CHROME.

In spite of that, Gonçalves could extract several metrics related to code quality. The result, grouped by components, is listed in Table 6.6.

Constraint Store	Instability	0.5
	Cyclomatic Complexity	3.77
	Maintainability Index	121.25
CDBJ Search	Instability	0.33
	Cyclomatic Complexity	5.0
	Maintainability Index	37.22
Propag. History	Instability	0.5
	Cyclomatic Complexity	3.5
	Maintainability Index	141.09
Entailment	Instability	0.25
	Cyclomatic Complexity	5.75
	Maintainability Index	81.27
Fired Rules	Instability	0.25
	Cyclomatic Complexity	2.67
	Maintainability Index	150.21

Table 6.6: CHROME (Manual) - Metrics

CHROME - Automatic generation

During this phase of the experiment, Gonçalves has used an incomplete version of the CHROME PIM. The reason for that is the confirmation of the second and third threat described in Section 6.5.1. The version used as input to the WAKAME Code Generator tool contains all the structural specification describes but it lacks of some operational views specifications. Nevertheless, the experimenter could specify **50% components completely**: *Propagation History*, *Fired Rules*, *Constraint Store* and *Entailment*. Thus this section describes some artifacts generated from these components as well the quality metrics obtained from them.

The Figure 6.18 depicts the interface of the Fired Rules component generated by the Structural Generator. Its implementation, Figure 6.19, shows the method body implementations generated by the IOCL Compiler tool.

```
// imports omitted

public interface FiredRules extends KComponent {
    public void add(
        java.lang.Integer ruleId, chrome4.Justification allJs,
        chrome4.nests.constraintstore.Constraint removedConstraints,
        chrome4.Justification js, java.lang.Boolean idr)
        throws Exception;

    public List<chrome4.nests.constraintstore.Constraint> removeDependentRules(
        chrome4.Justification js)
        throws Exception;

    public chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry
        removeNextLatestDependentRule(chrome4.Justification js) throws Exception;
}
```

Figure 6.18: Fired Rules - Specification

During the execution of this phase, Gonçalves reported some issues involving the IOCLCompiler, all of them were completely resolved and they were registered in the issue tracker of the IOCL Compiler tool. 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 <https://github.com/marcellustavares/imperative-ocl/issues>.

```

// imports omitted
@Component
public class FiredRulesImpl
implements chrome4.nests.queryprocessor.nests.firedrules.FiredRules {
    private java.lang.Integer lastEntry;
    public java.util.List<
chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry> contents;

    public void add(java.lang.Integer ruleId, chrome4.Justification allJs,
chrome4.nests.constraintstore.Constraint removedConstraints,
chrome4.Justification js, java.lang.Boolean idr)
        throws Exception {

        chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry element =
new chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry();
        element.ruleId = ruleId;
        element.js = allJs;
        element.isADisjunctiveRule = idr;
        this.contents =
org.orcas.commons.collections.CollectionUtils.including(this.contents, element);
        this.lastEntry = (this.lastEntry + 1);
    }

    public List<chrome4.nests.constraintstore.Constraint> removeDependentRules(
chrome4.Justification js) throws Exception {

        Integer i = lastEntry;
        chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry element =
new chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry();
        chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry lastElement =
new chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry();
        java.util.List<chrome4.nests.constraintstore.Constraint> result =
new org.orcas.commons.collections.list.TreeList(java.util.Arrays.asList());

        while ((i > 0)) {
            element = org.orcas.commons.collections.CollectionUtils.at(contents, i);
            if (element.js.isJustifiedBy(js)) {
                contents = org.orcas.commons.collections.CollectionUtils.excluding(
                    contents, element);
                lastEntry = (lastEntry - 1);
                result = org.orcas.commons.collections.CollectionUtils.union(
                    result, element.removedConstraints);
                lastElement = element;
            }
            i = (i - 1);
        }

        if (lastElement.isADisjunctiveRule) {
            contents = org.orcas.commons.collections.CollectionUtils.including(
                contents, lastElement);
            chrome4.nests.constraintstore.Constraint c = null;
            for (java.util.Iterator<chrome4.nests.constraintstore.Constraint> it =
                lastElement.removedConstraints.iterator(); it.hasNext(); ) {
                c = it.next();
                result = org.orcas.commons.collections.CollectionUtils.excluding(
                    result, c);
            }
            this.lastEntry = (this.lastEntry + 1);
        }
        return result;
    }

    public chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry
removeNextLatestDependentRule(
        chrome4.Justification js) throws Exception {

        Integer i = this.lastEntry;

        while ((i > 0)) {
            chrome4.nests.queryprocessor.nests.firedrules.FiredRulesEntry element =
org.orcas.commons.collections.CollectionUtils.at(contents, i);
            if (element.js.isJustifiedBy(js)) {
                contents = org.orcas.commons.collections.CollectionUtils.excluding(
                    contents, element);
                this.lastEntry = (this.lastEntry - 1);
                return element;
            }
            i = (i - 1);
        }
        return null;
    }
}

```

Figure 6.19: Fired Rules - Realization

Finally the Table 6.7 summarizes the quality metrics obtained from these four components.

Constraint Store	Instability	0.25
	Cyclomatic Complexity	1.41
	Maintainability Index	122.01
Propag. History	Instability	0.25
	Cyclomatic Complexity	2.25
	Maintainability Index	156.88
Entailment	Instability	0.33
	Cyclomatic Complexity	4.17
	Maintainability Index	102.94
Fired Rules	Instability	0.25
	Cyclomatic Complexity	2.2
	Maintainability Index	103.24

Table 6.7: CHROME (Generated) - Metrics

Analysis and Interpretation

The analysis of the data of CHROME just considered the numbers (average) of the four QueryProcessor subcomponents. With that in mind, the Figure 6.20, 6.21 and 6.22 depicts the values of the metrics obtained from the manual and automatic implementation of these components.

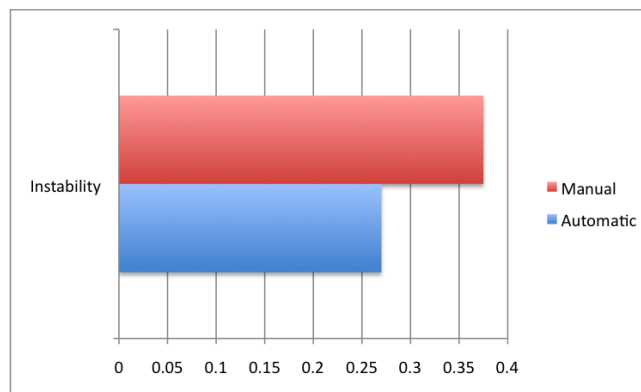
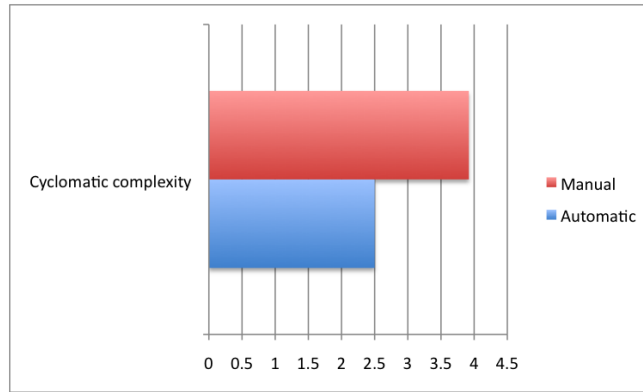
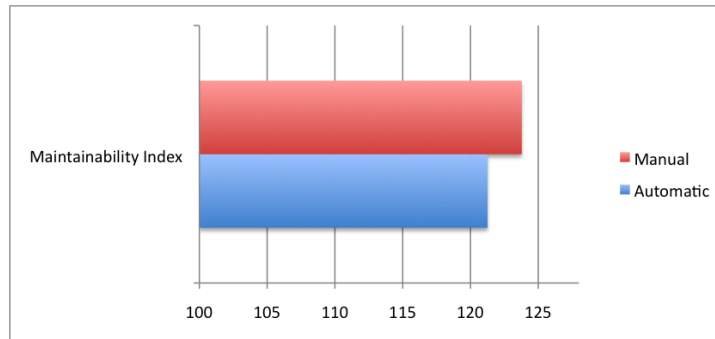


Figure 6.20: CHROME - *Instability* metric comparison

Similarly to the Web Agency experiment, the numbers reveal that there is no real discrepancy related to these metrics in analysis. Both the *Instability* and *Cyclomatic Complexity* indexes are slightly better in the automatic generation. Both values **rejects** the H_{0a} and H_{0b} respectively.

Figure 6.21: CHROME - *Cyclomatic Complexity* metric comparisonFigure 6.22: CHROME - *Maintainability Index* metric comparison

However, one of the metrics showed different result from what was obtained during the Web Agency experiment. The *Maintainability index* dropped from 123.785 in the manual implementation to 121.26 in the automatic version. The number in absolute value **confirms** the null hypothesis H_{0c} , but considering the scale of the index this represents a very small variation and both values still are in the range considered as modules with **good maintainability**.

A detailed analysis of the *Maintainability index*, depicted in Figure 6.23 shows the index values by component. The values shows an improvement of the index in **75%** of the components, being lower only in the Fired Rules Component.

A further investigation in the source code showed us that the manual implementation of Fired Rules component is not 100% compatible with the specification described in the PIM and this reason helps to explain the root cause of the difference. All other components were implemented according to the specification. The lack of a complete PIM, however, prevented the calculus of an average the metrics of all components and a final evaluation of the experiment, but the values collect until this phase of the experiment still provides us **evidences** that IOCL Compiler **maintained the same quality characteristics** when

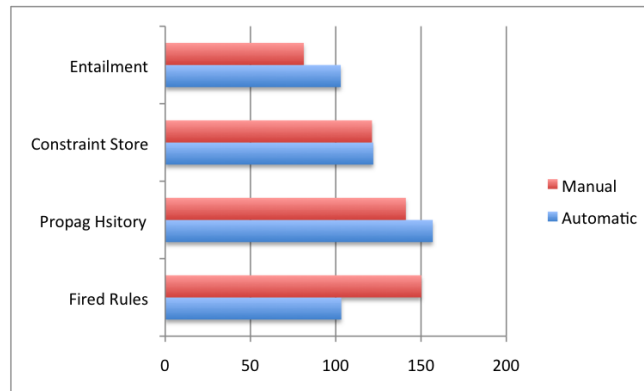


Figure 6.23: CHROME - *Maintainability Index* by components

compared to the handwritten code.

The non conclusion of the complete PIM and also the non existence of unit tests prevented its execution in both versions of the application. Thus we could not prove the correctness of the versions and a further research in this area is still needed.

6.6 Chapter Summary

In this Chapter was presented all the evaluations performed for IOCL Compiler. All experiments brought valid consideration about the tool, the second experiments although, could not be fulfilled because one of its threats (time restrictions to complete the experiment) could not be resolved.

Even though, the quality metrics collected provided good evidences about the benefits of the tool. It was observed a quality improvement in most of components evaluated and the second experiment in particular provided us the opportunity to resolve some issues found by the experimenter when using the tool. It is worthwhile mentioning that additional experiments can be also be done in the future, mainly to provide us data about the tool helpfulness in the process execution and to evaluate the advantages/disadvantages of the usage of Imperative OCL to describe the system behavior.

Next Chapter presents the works found in the literature that are related to this dissertation.

7

Related Work

During the development of this work, several works and tools were identified as related to the context of this research. In this chapter we present the selected ones. Section 7.1 presents the works related to the behavioral modeling. Section 7.2 describes the most adopted OCL tools and Section 7.3 details the Eclipse M2M project, which includes an implementation the QVT specification. Finally, Section 7.4 draws the chapter summary.

7.1 Previous Studies on Behavioral Modeling

Several previous studies discussed approaches for behavioral modeling in platform-independent models. The main approach advocated by the OMG group is the use of the *Action Language* (OMG, 2010a) that conforms to the UML *Action Semantics*.

However, criticisms to action languages are raised since 2004 by (Haustein & Pleumann, 2004). The authors propose a surface language that is based on and aligned with OCL. They claim that all semantic language rely on a syntax that is incoherent with the existing OMG standard such as the Object Constraint Language (OCL). Thus, in his article he discusses a proposal of an action surface language by embedding OCL expressions in new syntax constructs for actions.

(Jiang *et al.* , 2008) also propose an OCL-based executable UML (OxUML). He suggests that that OCL can be partly used for ASL, and the capability of model execution can be provided by extending OCL. Thus he defines an new language called OCL for Execution (OCL4X) by extending OCL with the expressions present in the AS specification but not in OCL.

(Kelsen *et al.* , 2008) follows the same direction the PIM executability by using OCL. It is proposed a declarative language for describing the behavior of platform-independent models based on a hybrid notation that uses graphical elements as well as textual elements

in the form of OCL code snippets.

Different approaches like (Riccobene & Scandurra, 2009) seek to use an ASM-based extension of the UML and its Action Semantics to define a high level behavioral formalism for the construction of executable PIMs. The executability in this case is achieved by weaving behavioral aspects expressed in terms of ASM elements into the UML metamodel.

All the techniques described above seeks to reuse existing standards to achieve PIM executability. However all the approaches found pursues this goal by either creating concrete syntax for AS or extending behavior diagrams and OCL languages with action constructors.

In our approach, we propose the usage of Imperative OCL Language to specify the system behavior. As discussed along this dissertation, IOCL is a consolidated standard and it also has all necessary requirements to achieve the model executability at PIM level. Through the combination of IOCL and Kobra/UML we also attempt to ease the adoption of the developed solution.

Although we were not able to find any downloadable tool capable of compile IOCL expressions to target languages, a tool was proposed by (Vajk & Levendovszky, 2006). However, in this work, IOCL is used to specify model transformations and the compiler is designed to run inside an environment called Visual Modeling and Transformation System (VMTS) (VTMS, 2011), features that make this project out of the scope of this work.

7.2 Works related with the use of OCL

In this section we present the most adopted OCL tools by the MDA community. Its usage as the base of the development of the IOCL Compiler was considered during the early phases of this work but due the lack of documentation related to the modules extension and specially because a high level intrusion was required in the project's source code to extend the OCL to IOCL in these tools led us to just adopt them as reference to the implementation of our work.

7.2.1 Eclipse OCL

Nowadays, Eclipse (Eclipse, 2010a) is much more than Integrated Development Environments (IDEs). The Eclipse foundation also supports several open-source projects and some of them related do Model Driven Engineering.

Eclipse OCL (MDT, 2010) is one of the many projects comprised in the Eclipse Modeling Project (EMP). This project focuses on the evolution and promotion of model-based development technologies within the Eclipse community. Also, it unites projects failing into this classification to bring holistic model-based capabilities to Java community. It is a strong effort of the Eclipse foundation in realizing the MDA mission.

The EMP project is subdivided in several areas, which in turn comprises other several subprojects, involving fields such Abstract Syntax Development, Concrete Syntax Development and Model Transformations technologies. The Eclipse OCL is one of them.

The Eclipse OCL is an implementation of the OCL OMG standard for EMF-based models. It provides the entire infrastructure for parsing, validating and evaluating OCL expressions. It also provides a Visitor API for transforming the AST model of the input OCL expression. The Figure 7.1 shows the basic API for handling OCL expression in Eclipse OCL.

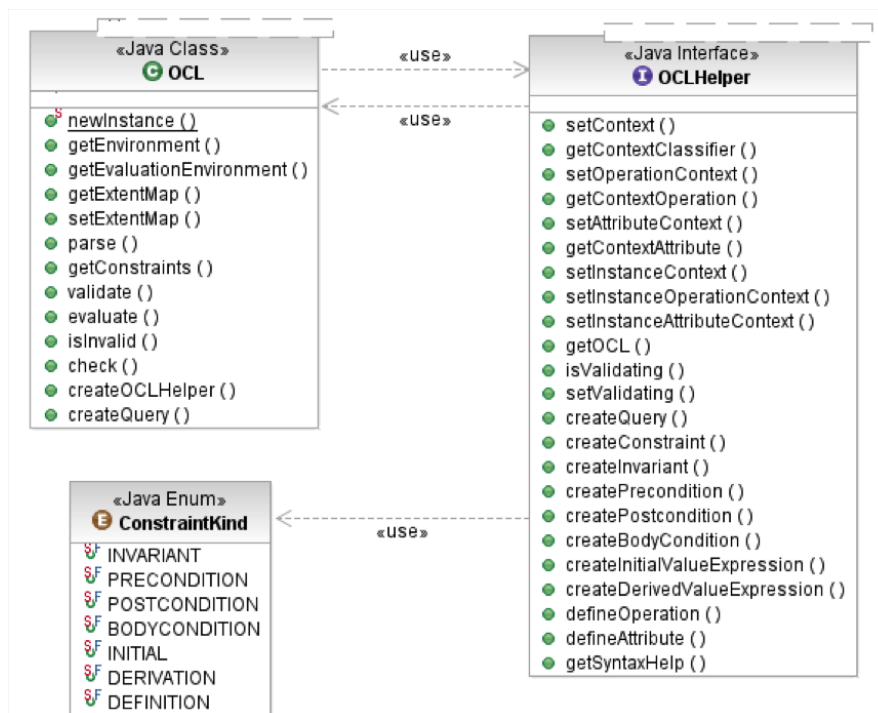


Figure 7.1: Eclipse OCL API (MDT, 2010)

In this figure we can see the OCL class serves as the main entrypoint into the parsing API. From the OCL instance the OCLHelper object can be created to parse constraints and query expressions. Each type of constraint is handled by its specific method in the *OCLHelper* interface.

An important aspect is that different kind of constraints requires different contexts

definitions. The *setContext()*, *setOperationContext()*, and *setAttributeContext()* methods are responsible for configuring the appropriate environment to the further analysis of the parsed expressions. During the parsing and evaluation of the expressions, the *ParserException* can be raised. There are two possible reasons for that (Eclipse, 2010b): syntactic or contextual problems. The former detects errors in providing a proper OCL syntax, such as closing parenthesis, wrong operation names, etc. The latter detects errors such as wrong operands types, invalid context navigation, etc.

After the OCL expression is parsed, it is transformed to an AST using EMF. Once the AST is built, the evaluation uses the Visitor pattern (Gamma *et al.*, 1995) to visit all expression elements and do the evaluation task of the given element. The interface *EvaluationVisitor* is responsible for doing this job. The Visitor interface provided by Eclipse OCL can be easily extended to perform other kinds of functions such as code generation.

The Eclipse OCL tool does not provide any built-in facility to compile OCL expressions to platform-specific code, but other tools were found in literature aiming filling this gap, such as: (Garcia & Shidqie, 2007) and (Shidqie, 2007).

7.2.2 Dresden OCL

The Dresden OCL toolkit (Dresden, 2011) has been developed at Technische Universität Dresden since 1999. The latest version is called Dresden OCL2 for Eclipse and includes the OCL parser, interpreter and code generator. Its architecture is shown in figure 7.2.

The back-end represents the meta-model used to load models and constraints into the toolkit. There is a total independence of model repositories used with the framework due the implementation of a *pivot model*, the pivot model responsible for making the toolkit independent of the meta-model used. This architecture makes the repository easily exchanged for example between EMF and the UML repository from Eclipse Model Development Tools Project (Eclipse, 2011b).

The toolkit basis layer represents the project kernel. It is divided into three modules: the Pivot Model, Essential OCL and the Model Bus. This layer provides core functionalities to load models and constraints into the toolkit. The Pivot Model use was mentioned before; the Essential OCL extends the Pivot Model to define a meta-model, based on the OCL 2.0 standard library for OCL constraints. Several types of expressions are supported by the toolkit, such as invariants, pre and postconditions, definitions and body expressions. Finally, the Model Bus provides access to models or model instances that have been loaded into the toolkit.

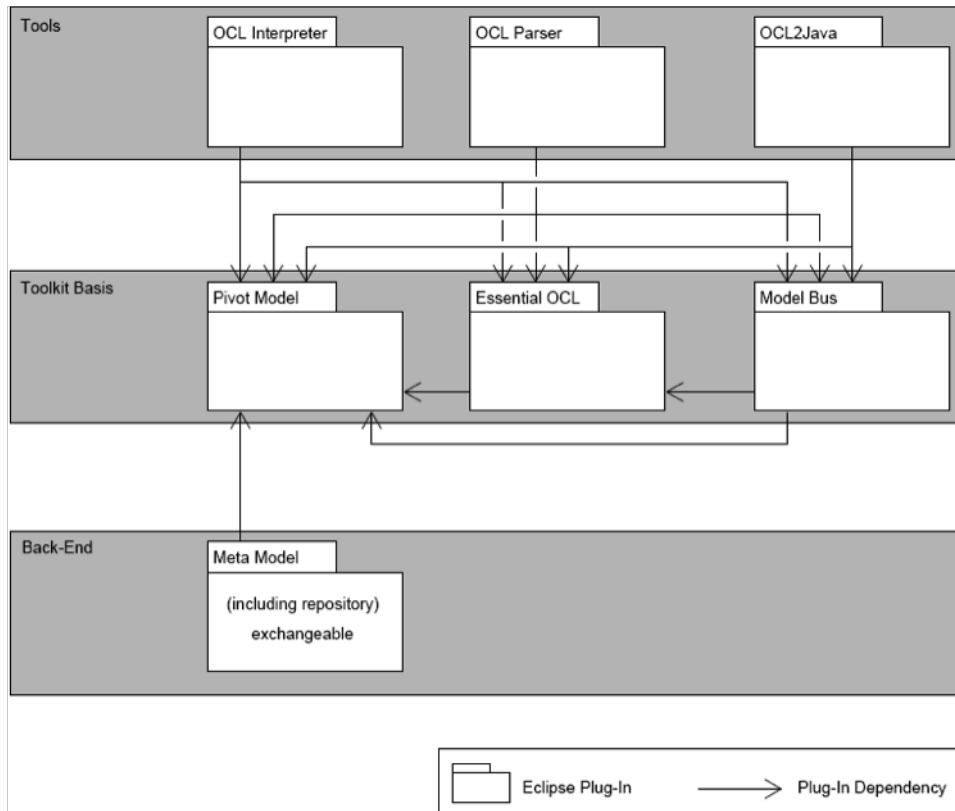


Figure 7.2: Dresden OCL Packages Structure (Dresden, 2011)

The top layer contains the built-in tools provided by the toolkit. The last stable version of the project is 2.2 and it provides three tools: OCL2 Parser, OCL2 Interpreter and OCL2 java code generator. The parser and the interpreter are used to load, interpret and verify OCL constraints, the OCL2Java is the code generator module, and it is used to generate Java code for the successful parsed OCL constraints. The OCL2Java uses aspect-oriented language AspectJ to instrument the Java code generation for the imported model.

7.3 Works related with the use of IOCL

7.3.1 Eclipse M2M

The Eclipse M2M project (Eclipse, 2011a), as Eclipse OCL, is one subproject of Eclipse Modeling Project (EMP). The main purpose of this project is to develop a framework for model-to-model transformation languages. The project is composed of a core transformation infrastructure and several pluggable transformation engines that will execute the transformations. There are three transformation engines developed in this context:

ATL(ATL, 2009), Procedural QVT and Declarative QVT.

The ATL component of the M2M project aims at providing a set of model-to-model transformation tools. These include some sample ATL transformations, an ATL transformation engine, and an IDE for ATL.

The procedural QVT (QVTO) component is an implementation of the Operational part of the QVT specification. The QVTO operates with EMF models and it is composed of three packages:

- **QVT Operational package** General structuring elements and top-level constructions;
- **Imperative OCL** Package extension to OCL expressions and type system;
- **Standard Library** Several utility functions used to specify the transformations.

The QVT Operational package tackles the parsing and execution of the QVT operational transformations. Operational transformations contains several of imperative operations (mappings, helpers, queries, constructors) that makes easier the specification of the transformation. There are five types of imperative operations inside the QVTO: Entry Operation, Mapping, helpers, Queries and Constructors

An entry operation is the entry point for the execution of a transformation. Typically refers to model parameters and invokes top-level mappings. A helper is an operation that performs a computation on one or more source objects and provides a result. A query is a “read-only” helper which is not allowed to create or update any objects. A constructor is an operation that defines how to create and populate the properties of an instance of a given class. Mapping is a special constructor that defines a A mapping between one or more source model elements into one or more target model elements.

Imperative OCL packages handles the IOCL expressions. The IOCL expressions are meant to be used for the definition of the operational M2M transformations. The package also includes a parser for all the expressions presented in Section 2.2.2. However, the project does not provide any external access to this module. The rigid implementation and the lack of user documentation prevented its usage and extension in this project.

in our project to reuse this package to implement the IOCL Compiler.

The QVT Declarative (QVTd) component aims to provide a complete Eclipse based IDE for the Core and Relations Languages defined by the OMG QVT Relations language. The goal includes all development components necessary for development of QVT core and QVT relation programs and APIs to facilitate extension and reuse

The main difference between QVTO and QVTR is that QVTO contains fully-described detailed execution instructions. On the contrary, QVTR omits these explicit steps and relies on individual correspondence between elements of the input and output domain. Consequently, QVTO scripts require a complete algorithm (i.e. how to produce an output model given the whole input model) while QVTR contains routines for element-to-element mapping.

7.4 Chapter Summary

All two OCL tools evaluated during this work have similar features that includes parsing and evaluation of OCL expression. However, the Dresden toolkit and the Object Constraint Language Environment also have the ability to generate Java code from the expressions.

Compared with the Imperative OCL compiler, both Eclipse OCL and Dresden toolkit show a higher level of stability and a greater number of OCL expressions supported. However, none of them are able to parse and generate code from Imperative OCL expressions.

In spite of the fact that Eclipse M2M supports IOCL for writing transformations, M2M is scoped to model-to-model transformation, the usage of IOCL is treated differently from what this work proposes, which advocates its usage for describing the system behavior at PIM level.

Next Chapter concludes this dissertation by summarizing the work performed, reviewing some related works, pointing directions for future enhancements to the environment and presenting some final considerations.

8

Conclusion

This chapter presents the contributions and limitations of the Imperative OCL Engine and Imperative OCL Generator components, the future work of these modules and concluding remarks.

8.1 Contribution

This work presents contributions that were organized and classified as contributions to the WAKAME CASE tool, to MDE and Kobra.

The identified contributions to WAKAME CASE tool were:

- Proposal of a innovative usage of Imperative OCL expressions. The work proposes its utilization during the behavioral model specification at PIM level;
- Design and implementation of IOCL compiler. A tool that automatically generates platform-specific code from IOCL language.
- Two different evaluations for verifying the the tool helpfulness to a MDA process.

Identified contributions to MDE and Kobra:

- Creation and publication of the compiler component from IOCL expressions. IOCL expressions was proved to be a interesting way to simplify the specification of behavioral algorithms. The difficulty of draw behavioral diagrams and generate platform-specific code from them is one of the cause the lack of the tools capable of generate the full code of the application.
- Update of Kobra metamodel with Imperative OCL expressions.

- Validation of Kobra method through the accomplished of the models elaborated in this work, KISF, Web Agency and the IOCLEngine and IOCLGenerator components

8.2 Limitations

There are some known limitations of this initial version of the IOCLCompiler. The reasons for these limitations were not technical but resulting of time restrictions for the development. They are:

- The IOCLEngine current implementation does not support all Imperative OCL expressions, with all the syntax variations as presented in QVT specification. The figures 8.1 and 8.2 shows which expressions are not currently supported by this compiler. The red marks show the expressions not yet covered by the compiler and the orange marks identify expressions that are partially implemented. By partially we mean: a) some expressions provide different types of syntax flavors, such the Switch expression in the Imperative OCL package and for this cases and only one flavor was contemplated; b) complex iterator expressions are not fully supported at this stage of development.
- The IOCLAnalyzer subcomponent, which deals with the semantics checking, is only available for Kobra models.
- The *CodeGenerator* component does not map all operations available in QVT operational mappings standard library to Java standard library calls.

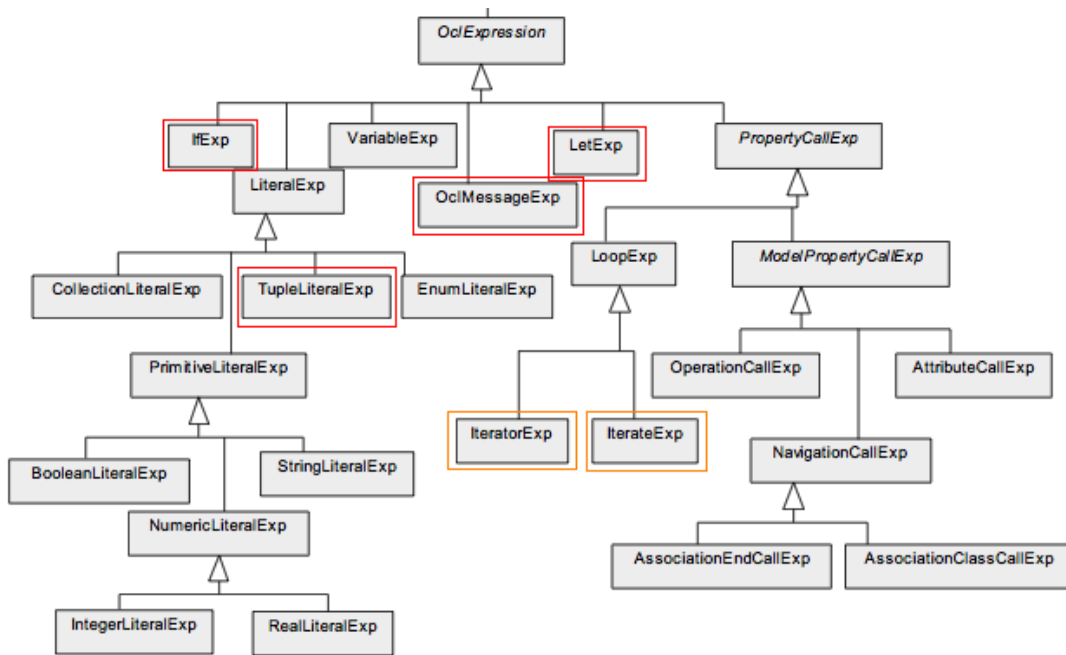


Figure 8.1: Unsupported part of OCL expressions

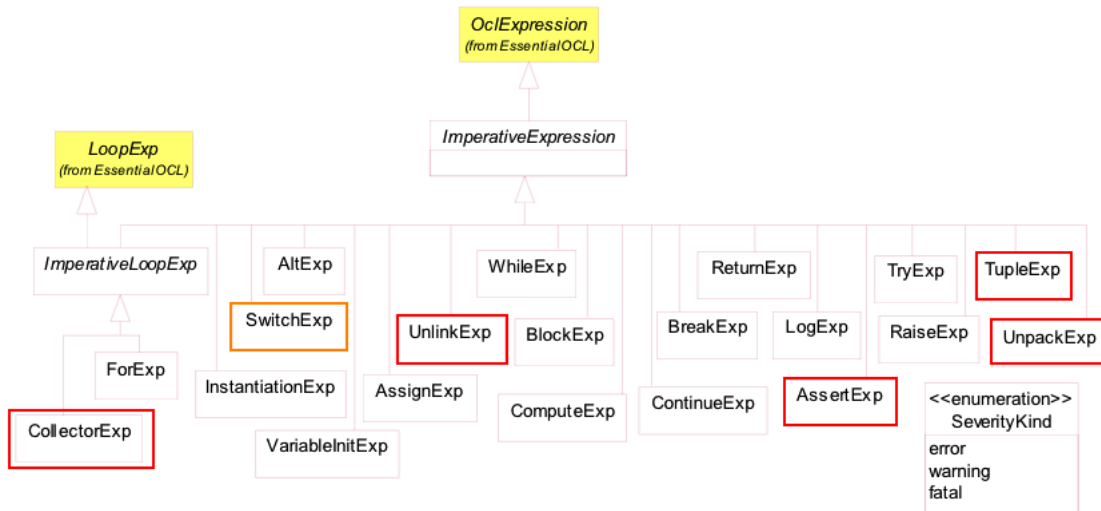


Figure 8.2: Unsupported part of IOCL expressions

8.3 Future Work

In spite of the commitment to develop the tool, some improvements are visualized. They were not included in the initial tool development because some of them were considered out of the scope of the master degree, but are important to a full industrial tool, and others

were comments/feedbacks from users. In this fashion, some important aspects that were left out of this version are enumerated:

- Extend the IOCLAnalyzer implementation to other metamodels such as UML or EMF.
- Extend the IOCL grammar to include the missing expressions and syntax variations not yet covered
- Complete the CodeGenerator to handle the missing expressions, depicted in figures 8.1 and 8.2.
- Complete the remaining operations mapping from QVT standard library to Java
- Integrate the WAKAMECodeGenerator in the WAKAME tool.

8.4 Concluding Remarks

The development of the WAKAME Code Generator tool involved the collaboration of two master's degree students. The focus of this work, which is part of this project, was the development of IOCL components able to handle the parsing, semantic analysis and code generation of imperative OCL expressions into Java code.

We divided this work into two components: the first one responsible for the parsing, semantic analysis and syntax suggestions of the input expressions and the second that provides a reusable infrastructure for code generation together with a built-in IOCL to Java generator.

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 both components described in chapter 5. And the third and last stage was accomplished a case study for the evaluation of the implemented applications, and of this case study it was obtained positive results in relation to what we proposed.

The main contributions at the end of this work were the development of the first Imperative OCL to Java generator, which was proved to be a good way to simplify the behavioral specification of the application to finally achieve the full code generation proposed by MDA.

Bibliography

- Abdennadher, Slim. 2001. *Rule-Based Constraint Programming: Theory and Practice*.
- Apache. 2011. *Velocity*. <http://velocity.apache.org/>. Last Access: 07/05/2011.
- Atkinson, Colin, & Stoll, Dietmar. 2008. Orthographic Modeling Environment. *Pages 93–96 of: Fiadeiro, José Luiz, & Inverardi, Paola (eds), FASE. Lecture Notes in Computer Science*, vol. 4961. Springer.
- Atkinson, Colin, Bostan, Philipp, Brenner, Daniel, Falcone, Giovanni, Gutheil, Matthias, Hummel, Oliver, Juhasz, Monika, & Stoll, Dietmar. 2008. Modeling Components and Component-Based Systems in Kobra. *Pages 54–84 of: Rausch, Andreas, Reussner, Ralf, Mirandola, Raffaella, & Plasil, Frantisek (eds), The Common Component Modeling Example. Lecture Notes in Computer Science*, vol. 5153. Springer Berlin / Heidelberg.
- ATL, Eclipse. 2009. *ATL - Atlas Transformation Language*.
- Basili, V.R., Caldiera, G., & Rombach, H.D. 1994. The Goal Question Metric Approach. *Encyclopedia of Software Engineering*, **1**, 528–532.
- Beck, Kent, & Andres, Cynthia. 2004. *Extreme Programming Explained : Embrace Change (2nd Edition)*. Addison-Wesley Professional.
- Booch, Grady, Maksimchuk, Robert A., Engel, Michael W., Young, Bobbi J., Conallen, Jim, & Houston, Kelli A. 2007. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. 3 edn. Addison-Wesley Professional.
- C. Atkinson, D. Stoll, & Bostan, P. 2009 (5). *Supporting View-Based Development through Orthographic Software Modeling*. Proceedings of 4th International Conference on Evaluation on Novel Approaches to Software Engineering.
- Calic, Tihomir, Dascalu, Sergiu, & Egbert, Dwight. 2008. Tools for MDA Software Development: Evaluation Criteria and Set of Desirable Features. *Information Technology: New Generations, Third International Conference on*, **0**, 44–50.
- Carmen Avila, Amritam Sarcar, Yoonsik Cheon. 2010. Runtime Constraint Checking Approaches for OCL, A Critical Comparison. *22nd International Conference on Software Engineering and Knowledge Engineering*, **22**, 393–398.

- Chang, Fay, Dean, Jeffrey, Ghemawat, Sanjay, Hsieh, Wilson C., Wallach, Deborah A., Burrows, Mike, Chandra, Tushar, Fikes, Andrew, & Gruber, Robert E. 2006. Bigtable: A distributed storage system for structured data. *In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*.
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., & Jeremaes, P. 1994a. *Object-Oriented Development: The FUSION Method*. Englewood Cliffs, New Jersey, USA: Prentice Hall International.
- Coleman, D.M., Ash, D., Lowther, B., & Oman, P.W. 1994b. Using metrics to evaluate software system maintainability. *IEEE Computer*, **27**(8), 44–49.
- Crnkovic, Ivica, Chaudron, Michel R. V., & Larsson, Stig. 2006. Component-Based Development Process and Component Lifecycle. *Page 44 of: ICSEA*. IEEE Computer Society.
- Czarnecki, Krzysztof, & Helsen, Simon. 2003. Classification of Model Transformation Approaches. *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*.
- Dresden. 2011 (01). *Dresden OCL*. <http://dresden-ocl.sourceforge.net>. Last Access: 01/29/2011.
- Eclipse. 2010a. *Eclipse Foundation*. <http://www.eclipse.org/org/>. Last Access: 01/29/2011.
- Eclipse. 2010b. *Tutorial: Working with OCL*. <http://publib.boulder.ibm.com/infocenter/rsahelp/v7r0m0/index.jsp?topic=/org.eclipse.emf.ocl.doc/tutorials/oclInterpreterTutorial.html>. Last Access: 01/29/2011.
- Eclipse. 2011a. *Eclipse M2M Project*.
- Eclipse. 2011b. *Eclipse Modeling Development Tools Project*. <http://www.eclipse.org/modeling/mdt/>. Last Access: 01/29/2011.
- EMF, Eclipse Modeling Framework Project. 2010. *Eclipse Modeling Framework Project*.
- Epsilon. 2011.
-

- Fowler, Martin. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, Amsterdam.
- Freeman, Eric, Freeman, Elisabeth, Bates, Bert, & Sierra, Kathy. 2004. *Head First Design Patterns*. 1 edn. O'Reilly Media.
- FreeMarker. 2011. *FreeMarker*. <http://freemarker.sourceforge.net/>. Last Access: 01/29/2011.
- Frühwirth, Thom. 1994. Theory and Practice of Constraint Handling Rules. *The Journal of Logic Programming*, **37**(1-3), 95–138.
- Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1995. *Design Patterns*. Boston, MA: Addison-Wesley.
- Garcia, Miguel, & Shidqie, A. Jibran. 2007. OCL Compiler for EMF. *Eclipse Modeling Symposium at Eclipse Summit Europe 2007, Stuttgart, Gemany, 2007*.
- Google. 2011. *Google App Engine*. <http://code.google.com/appengine/>. Last Access: 12/18/2010.
- Haustein, Stefan, & Pleumann, Jörg. 2004. OCL as Expression Language in an Action Semantics Surface Language. *Pages 99–113 of: Patrascoiu, Octavian (ed), OCL and Model Driven Engineering, UML 2004 Conference*. University of Kent.
- Heitz, Claudius, Thiemann, Peter, & Wölflé, Thomas. 2007. Integration of an Action Language Via UML Action Semantics. *Pages 172–186 of: Draheim, Dirk, & Weber, Gerald (eds), Trends in Enterprise Application Architecture*. Lecture Notes in Computer Science, vol. 4473. Springer Berlin / Heidelberg. 10.1007/978-3-540-75912-6_13.
- Herrington, Jack. 2003. *Code Generation in Action*. Greenwich, CT, USA: Manning Publications Co.
- JavaCC. 2011. *Java CC*. <http://java.net/projects/javacc/>. Last Access: 01/13/2011.
- JET. 2011.
- Jiang, Ke, Zhang, Lei, & Miyake, Shigeru. 2008. Using OCL in Executable UML. *ECEASST*, **9**.
- JUnit. 2011. *Junit Test Framework*. <http://www.junit.org/>. Last Access: 01/13/2011.
-

- Kelsen, Pierre, Pulvermueller, Elke, & Glodt, Christian. 2008. Specifying Executable Platform-Independent Models using OCL. *ECEASST*, **9**.
- Kiczales, Gregor, Lamping, John, Menhdhekar, Anurag, Maeda, Chris, Lopes, Cristina, Loingtier, Jean-Marc, & Irwin, John. 1997. Aspect-Oriented Programming. *Pages 220–242 of: Akşit, Mehmet, & Matsuoka, Satoshi (eds), Proceedings European Conference on Object-Oriented Programming*, vol. 1241. Berlin, Heidelberg, and New York: Springer-Verlag.
- Kleppe, Anneke G., Warmer, Jos, & Bast, Wim. 2003. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
- Kruchten, Philippe. 2003. *The Rational Unified Process: An Introduction*. Third edn. Boston, MA: Addison-Wesley.
- Lacerda, Luiz Francisco Buarque. 2007. *Um profile UML 2 e um processo de modelagem para engenharia de interfaces gráficas dirigida a modelos e baseada em componentes*. M.Phil. thesis, UNIVERSIDADE FEDERAL DE PERNAMBUCO, CENTRO DE INFORMÁTICA.
- Lehman, M. M., & Belady, L. A. 1985. *Program evolution: processes of software change*. San Diego, CA, USA: Academic Press Professional, Inc.
- Lewandowski, Scott M. 1998. Frameworks for Component-Based Client/Server Computing. *ACM Comput. Surv.*, **30**(1), 3–27.
- Machado, Breno Batista. 2009. *A Cloud Deployed Repository for a Multi-View Component-Based Modeling CASE Tool*. M.Phil. thesis, Universidade Federal de Pernambuco.
- Marinho, Machado, B. B., Robin, & P., J. L. 2009. *Um Framework UML2 para Modelagem de Aplicações Web dentro de um Processo de Engenharia de Software Dirigido por Modelos e Baseado em Componentes*. Anais do III Workshop de Desenvolvimento Rápido de Aplicações.
- Marinho, Weslei Alvim Tarso. 2009. *A Web GUI for a Multi-View Component-Based Modeling CASE Tool*. M.Phil. thesis, Universidade Federal de Pernambuco.
- Martin, Robert C. 2002. *Agile Software Development, Principles, Patterns, and Practices*. Prentice-Hall, Inc.
-

- McCabe, Thomas J. 1976. A Software Complexity Measure. *IEEE Transactions on Software Engineering*, **2**(4), 308–320.
- MDT, Model Development Tools. 2010. *Model Development Tools - OCL Project*. Last Access: 12/21/2010.
- Mellor, Stephen J., & Balcer, Marc. 2002. *Executable UML: A Foundation for Model-Driven Architectures*. Boston, MA, USA: Addison-Wesley.
- Mens, Tom, Czarnecki, Krzysztof, & Gorp, Pieter Van. 2004. 04101 Discussion - A Taxonomy of Model Transformations. In: Bézivin, Jean, & Heckel, Reiko (eds), *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings, vol. 04101. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.
- Merson, Paulo. 2009. *Hierarchy of diagrams in UML 2.2*.
- Metzger, A. 2005. *A Systematic Look at Model Transformations*. in S. Beydeda, V. Gruhn (Eds.) *Model-driven Software Development - Volume II of Research and Practice in Software Engineering*. Heidelberg: Springer-Verlag. 2005.
- MOFM2T, OMG. 2010. *MOF Model to Text Transformation Language*. <http://www.omg.org/spec/MOFM2T/>. Last Access: 01/29/2011.
- MOFScript. 2010. *MOFScript*. <http://www.eclipse.org/gmt/mofscript/>. Last Access: 01/29/2011.
- Mohagheghi, Parastoo, & Dehlen, Vegard. 2008. Where Is the Proof? - A Review of Experiences from Applying MDE in Industry.
- Oliveira, Thiago Araujo Silva. 2011. *Um Gerador de Código Estrutural Java Implantável em Nuvens a partir de modelos de componentes assembly UML que seguem o método Kobra*. M.Phil. thesis, Universidade Federal de Pernambuco.
- OMG. 2003 (June). *MDA Guide Version 1.0.1*. Object Management Group, Framingham, Massachusetts.
- OMG. 2009. *Query/View/Transformation (QVT)*. Tech. rept. Object Management Group.
- OMG. 2010a. *Action Language For Foundational UML (ALF)*. <http://www.omg.org/spec/ALF/>.
-

- OMG. 2010b. *Model Driven Architecture*. <http://www.omg.org/mda/>. Last Access: 12/18/2010.
- OMG. 2010c. *Object Management Group*. <http://www.omg.org>. Last Access: 12/18/2010.
- OMG. 2010d. *OMG Object Constraint Language (OMG OCL) Version 2.2*. Tech. rept. formal/2010-02-01. Object Management Group.
- OMG. 2010e. *OMG's MetaObject Facility (MOF)*. <http://www.omg.org/mof/>. Last Access: 01/29/2011.
- OMG. 2010f (05). *UML 2.3 Infrastructure*. <http://www.omg.org/spec/UML/2.3>. Last Access: 01/29/2011.
- OMG. 2010g. *XML Metadata Interchange (XMI)*. <http://www.omg.org/spec/XMI/>.
- Parr, Terence. 2007. *The Definitive ANTLR Reference Guide: Building Domain-specific Languages*. Raleigh, NC: Pragmatic Bookshelf.
- Pohl, Klaus, Böckle, Günter, & van der Linden, Frank. 2005. *Software product line engineering: foundations, principles, and techniques*. Birkhäuser.
- Riccobene, Elvinia, & Scandurra, Patrizia. 2009. Weaving executability into UML class models at PIM level. *First European Workshop on Behaviour Modelling in Model Driven Architecture*.
- Robin, Jacques Louis Pierre. 2009 (09). *KobrA2 Metamodel*. <http://code.google.com/p/kobra2/>. Last Access: 01/29/2011.
- Schwaber, Ken, & Beedle, Mike. 2002. *Agile Software Development with Scrum*. Upper Saddle River, NJ: Prentice Hall.
- Shidqie, Ayatullah Jibran. 2007. *Compilation of OCL into Java for the Eclipse OCL Implementation*. M.Phil. thesis, Information and Media Technologies.
- Stahl, Thomas, & Völter, Markus. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. Chichester, UK: Wiley.
- Sunyé, Gerson, Pennaneach, François, Ho, Wai-Ming, Guennec, Alain Le, & Jééquel, Jean-Marc. 2001. Using UML Action Semantics for Executable Modeling and Beyond. *Pages 433–447 of: Dittrich, Klaus R., Geppert, Andreas, & Norrie, Moira C. (eds), CAiSE. Lecture Notes in Computer Science, vol. 2068*. Springer.
-

- Vajk, Tamás, & Levendovszky, Tihamér. 2006. *Magyar Kutatók 7. Nemzetközi Szimpóziuma 7 th International Symposium of Hungarian Researchers on Computational Intelligence Imperative OCL Compiler Support for Model Transformations*.
- Vitorino, Jairson. 2009. *Model-Driven Engineering a Versatile, Extensible, Scalable Rule Engine through Component Assembly and Model Transformations*. Ph.D. thesis, Universidade Federal de Pernambuco.
- VTMS. 2011. *Visual Modeling and Transformation System*. Last Access: 04/14/2011.
- Warner, J., & Kleppe, A. 2003. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc.
- Weilkiens, Tim, & Oestereich, Bernd. 2006. *UML 2 Certification Guide: Fundamental & Intermediate Exams (The OMG Press)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Wikipedia. 2011a. *Computer Aided Software Engineering*. http://en.wikipedia.org/wiki/Computer-aided_software_engineering. Last Access: 12/19/2010.
- Wikipedia. 2011b. *Null hypothesis*. http://en.wikipedia.org/wiki/Null_hypothesis. Last Access: 02/05/2011.
- Wohlin, C., Runeson, P., Höst, M., M. C. Ohlsson, B. Regnell, & Wesslén, Anders. 2001. Book Review: Experimentation in Software Engineering: An Introduction. By Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell and Anders Wesslén. Kluwer Academic Publishers, 1999, ISBN 0-7923-8682-5. *Softw. Test., Verif. Reliab.*, **11**(3), 198–199.
- XML, W3C. 2010. *XML*. <http://www.w3.org/XML/>. Last Access: 01/29/2011.
- XSLT, W3C. 2010. *XSLT*. <http://www.w3.org/TR/xslt>. Last Access: 01/29/2011.
- Yoonsik Cheon, Carmen Avila, Steve Roach. 2009. Checking Design Constraints at Runtime Using OCL and AspectJ. *International Journal of Software Engineering*, **2**(3), 5–28.

Appendices



Imperative OCL Grammar

```
grammar IoCl;  
  
options {  
  backtrack=true;  
  memoize=true;  
  output=AST;  
}  
  
tokens {  
  ALT_EXP;  
  AND = 'and';  
  APPEND = '+=';  
  ARROW = '→';  
  ATTRIBUTE_CALL;  
  BLOCK;  
  BREAK = 'break';  
  COLLECTION_LITERAL;  
  COLLECTION_TYPE;  
  COLON = ':';  
  COMPUTE = 'compute';  
  CONTINUE = 'continue';  
  DICT = 'Dict';  
  DICT_LITERAL_EXP;  
  DIV = '/';  
  DO = 'do';  
  DOT = '.';  
  ENDIF = 'endif';  
  ENUM_LITERAL;  
  ELIF = 'elif';  
  ELSE = 'else';  
  EQUAL = '=';  
  EXCEPT = 'except';  
  FOR = 'for';  
  GT = '>';  
  GTE = '>=';  
  IF = 'if';  
  IMPERATIVE_OPERATION_CALL;
```

```

ITERATE = 'iterate';
ITERATOR;
IS = ':=';
LOG = 'log';
LCURLY = '{';
LPAREN = '(';
LT = '<';
LTE = '<=';
MINUS = '-';
NEW = 'new';
NOT = 'not';
NOT_EQUAL = '<>';
MULT = '*';
OPERATION_CALL;
OR = 'or';
PATH_NAME;
PLUS = '+';
RAISE = 'raise';
RCURLY = '}';
RETURN = 'return';
RPAREN = ')';
SCOPE = '::';
SELF = 'self';
SEMICOLON = ';';
TRY = 'try';
VAR = 'var';
VARIABLE;
WHILE = 'while';
XOR = 'xor';
}

@lexer::header{
package org.orcas.ioclengine.parserantlr;
}

@header {
package org.orcas.ioclengine.parserantlr;
}

@members {
protected void mismatch(IntStream input, int ttype, BitSet follow)
    throws RecognitionException
{
    throw new MismatchedTokenException(ttype, input);
}

public Object recoverFromMismatchedSet(IntStream input, RecognitionException e, BitSet follow)
    throws RecognitionException
{
    throw e;
}
}

@rulecatch {

```

```

catch (RecognitionException e) {
    throw e;
}

oclExpression
    : imperativeExp
    | logicalExp
    ;

logicalExp
    : equalityExp ((AND | OR | XOR)^ equalityExp)*
    ;

equalityExp
    : relationalExp ((EQUAL | NOT_EQUAL)^ relationalExp)*
    ;

relationalExp
    : additiveExp ((LT | LTE | GT | GTE)^ additiveExp)*
    ;

additiveExp
    : multiplicativeExp ((PLUS | MINUS)^ multiplicativeExp)*
    ;

multiplicativeExp
    : unaryExp ((MULT | DIV)^ unaryExp)*
    ;

unaryExp
    : (MINUS | NOT)^ unaryExp
    | instantiationExp
    | dictLiteralExp
    | dotArrowExp
    ;

instantiationExp
    : NEW^ pathName '(! arguments? )'!
    ;

dictLiteralExp
    : DICT LCURLY RCURLY -> ^(DICT_LITERAL_EXP)
    ;

dotArrowExp
    : oclExp propertyCallExp^+
    | oclExp
    ;

propertyCallExp
    : (DOT | ARROW)! modelPropertyCallExp
    | ARROW! loopExp
    ;

```

```

modelPropertyCallExp
    : operationCallExp
    | attributeCallExp
    ;

operationCallExp
    : NUMERIC_OPERATION '(' arguments? ')' -> ^(NUMERIC_OPERATION arguments?)
    | simpleName '(' arguments? ')' -> ^(OPERATION_CALL simpleName arguments?)
    ;

attributeCallExp
    : simpleName -> ^(ATTRIBUTE_CALL simpleName)
    ;

oclExp
    : literalExp
    | variableExp
    | type
    | '(' oclExpression ')' -> oclExpression
    ;

variableExp
    : simpleName -> ^(VARIABLE simpleName)
    ;

literalExp
    : enumerationLiteralExp
    | collectionLiteralExp
    | primitiveLiteralExp
    | nullLiteralExp
    ;

collectionLiteralExp
    : collectionTypeIdentifier '{' collectionLiteralParts? '}' -> ^(COLLECTION_LITERAL collectionTypeIdentifier collectionLiteralParts)
    ;

collectionTypeIdentifier
    : COLLECTION_TYPE_LITERAL
    ;

collectionLiteralParts
    : collectionLiteralPart (','! collectionLiteralParts)*
    ;

collectionLiteralPart
    : oclExpression
    ;

primitiveLiteralExp
    : numericLiteralExp
    | stringLiteralExp
    | booleanLiteralExp
    ;

```

```

nullLiteralExp
    : NULL_LITERAL
    ;

numericLiteralExp
    : integerLiteralExp
    | realLiteralExp
    ;

stringLiteralExp
    : STRING_LITERAL
    ;

booleanLiteralExp
    : BOOLEAN_LITERAL
    ;

integerLiteralExp
    : INTEGER_LITERAL
    ;

realLiteralExp
    : REAL_LITERAL
    ;

enumerationLiteralExp
    : IDENTIFIER (SCOPE IDENTIFIER)+ -> ^(ENUM_LITERAL IDENTIFIER IDENTIFIER+)
    ;

loopExp
    : iteratorExp
    | iterateExp
    ;

iteratorExp
    : ITERATOR_NAME LPAREN ((v1 = variableDeclaration ',')? v2 = variableDeclaration 'l')? oclExpression RPAREN
    -> ^(ITERATOR ITERATOR_NAME $v1? $v2? oclExpression)
    ;

iterateExp
    : ITERATE LPAREN (v1 = variableDeclaration SEMICOLON)? v2 = variableDeclaration 'l' oclExpression RPAREN
    -> ^(ITERATE $v1? $v2 oclExpression)
    ;

variableDeclaration
    : IDENTIFIER (':' type)? ('=' oclExpression)? -> ^(VARIABLE IDENTIFIER type? oclExpression?)
    ;

arguments
    : oclExpression ('!' oclExpression)*
    ;

simpleName
    : SELF

```

```

    | IDENTIFIER
    ;

primitiveType
    : PRIMITIVE_TYPE_LITERAL
    ;

collectionType
    : collectionTypeIdentifier (LPAREN type RPAREN)? -> ^(COLLECTION_TYPE collectionTypeIdentifier type?)
    ;

type
    : primitiveType
    | collectionType
    | DICT (LPAREN keyType=type ',' valueType=type RPAREN)? -> ^(DICT $keyType? $valueType?)
    | pathName
    ;

pathName
    : IDENTIFIER (SCOPE IDENTIFIER)* -> ^(PATH_NAME IDENTIFIER IDENTIFIER*)
    ;

// Imperative Expressions

imperativeExp
    : blockExp
    | breakExp
    | computeExp
    | continueExp
    | returnExp
    | variableInitExp
    | assignExp
    | raiseExp
    | whileExp
    | ifExp
    | tryExp
    | forExp
    | logExp
    | imperativeOperationCallExp
    ;

blockExp
    : DO? LCURLY imperativeExp* RCURLY -> ^(BLOCK imperativeExp*)
    ;

breakExp
    : BREAK^ SEMICOLON
    ;

computeExp
    : COMPUTE LPAREN variableDeclaration RPAREN oclExpression -> ^(COMPUTE variableDeclaration oclExpression)
    ;

continueExp

```

```

: CONTINUE^ SEMICOLON
;

returnExp
: RETURN logicalExp? SEMICOLON -> ^(RETURN logicalExp?)
;

variableInitExp
: VAR^ imperativeVarDeclarations SEMICOLON!
;

imperativeVarDeclarations
: imperativeVarDeclaration (','! imperativeVarDeclaration)*
;

imperativeVarDeclaration
: IDENTIFIER (':' type)? ((EQUAL | IS) logicalExp)? -> ^(VARIABLE IDENTIFIER type? logicalExp?)
;

assignExp
: dotArrowExp (IS | APPEND)^ logicalExp SEMICOLON!
;

raiseExp
: RAISE^ (type | STRING_LITERAL) SEMICOLON!
;

whileExp
: WHILE LPAREN condition = logicalExp RPAREN
  body = imperativeExp -> ^(WHILE $condition $body)
;

ifExp
: IF altExp (elifExp)* (elseExp)? ENDIF? -> ^(IF altExp elifExp* elseExp?)
;

elifExp
: ELIF! altExp
;

elseExp
: ELSE! oclExpression
;

altExp
: LPAREN condition = logicalExp RPAREN body = imperativeExp -> ^(ALT_EXP $condition $body)
;

tryExp
: TRY LCURLY imperativeExp* RCURLY except -> ^(TRY imperativeExp* except)
;

except
: EXCEPT LPAREN type RPAREN LCURLY imperativeExp* RCURLY -> ^(EXCEPT type imperativeExp*)

```

```

;

forExp
: oclExp ARROW FOR_NAME LPAREN iteratorList (',' condition = oclExpression)? RPAREN body = oclExpression
  -> ^(FOR FOR_NAME oclExp iteratorList $condition? $body)
;

iteratorList
: variableDeclaration (','! variableDeclaration)*
;

logExp
: LOG LPAREN STRING_LITERAL (',' IDENTIFIER)? (',' INTEGER_LITERAL)? RPAREN SEMICOLON -> ^(LOG STRING_LITERAL
;

imperativeOperationCallExp
: dotArrowExp SEMICOLON -> ^(IMPERATIVE_OPERATION_CALL dotArrowExp)
;

BOOLEAN_LITERAL
: 'true '
| 'false '
;

COLLECTION_TYPE_LITERAL
: 'Bag'
| 'Collection'
| 'OrderedSet'
| 'Sequence'
| 'Set'
;

PRIMITIVE_TYPE_LITERAL
: 'Integer'
| 'String'
| 'Real'
| 'Boolean'
| 'OclAny'
;

INTEGER_LITERAL
: '0'..'9'+
;

REAL_LITERAL
: ('0'..'9')+ '.' ('0'..'9')+ EXPONENT?
| '.' ('0'..'9')+ EXPONENT?
| ('0'..'9')+ EXPONENT
;

STRING_LITERAL
: '\\' ( ESC_SEQ | ~('\\'|'\'' ) ) * '\\'
;

```

```

: 'null'
;

```

```
: 'any'
| 'closure'
| 'collect'
| 'collectNested'
| 'exists'
| 'forAll'
| 'isUnique'
| 'one'
| 'select'
| 'sortedBy'
| 'reject'
;
```

```

: 'forEach'
| 'forOne'
;

```

$$\vdash ('a' .. 'z' | 'A' .. 'Z' | '_') ('a' .. 'z' | 'A' .. 'Z' | '0' .. '9' | '_')^*$$

```
: INTEGER_LITERAL '.' IDENTIFIER
;
```

```

: ( ' '
    | '\t'
    | '\r'
    | '\n'
  ) { $channel=HIDDEN; }
;

```

```
: ('e'|'E') ('+'|'-'?)? ('0'..'9')+
;
```

```
: ('0'..'9'|'a'..'f'|'A'..'F') ;
```

```
: '\\\ ' ('b'|'t'|'n'|'f'|'r'|'\\"'|'\ ' '|'\ ')|
| UNICODE_ESC
| OCTAL_ESC
:
;
```

```
fragment
OCTAL_ESC
    : '\\' ( '0'..'3' ) ( '0'..'7' ) ( '0'..'7' )
    | '\\' ( '0'..'7' ) ( '0'..'7' )
    | '\\' ( '0'..'7' )
    ;

fragment
UNICODE_ESC
    : '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
    ;
```

B

Web Agency - PIM

B.1 Structural Specification

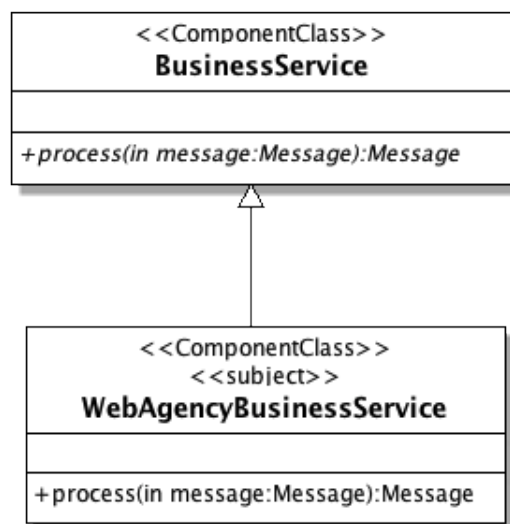


Figure B.1: *WebAgencyBusinessService* - Specification Structural Class Service

B.1. STRUCTURAL SPECIFICATION

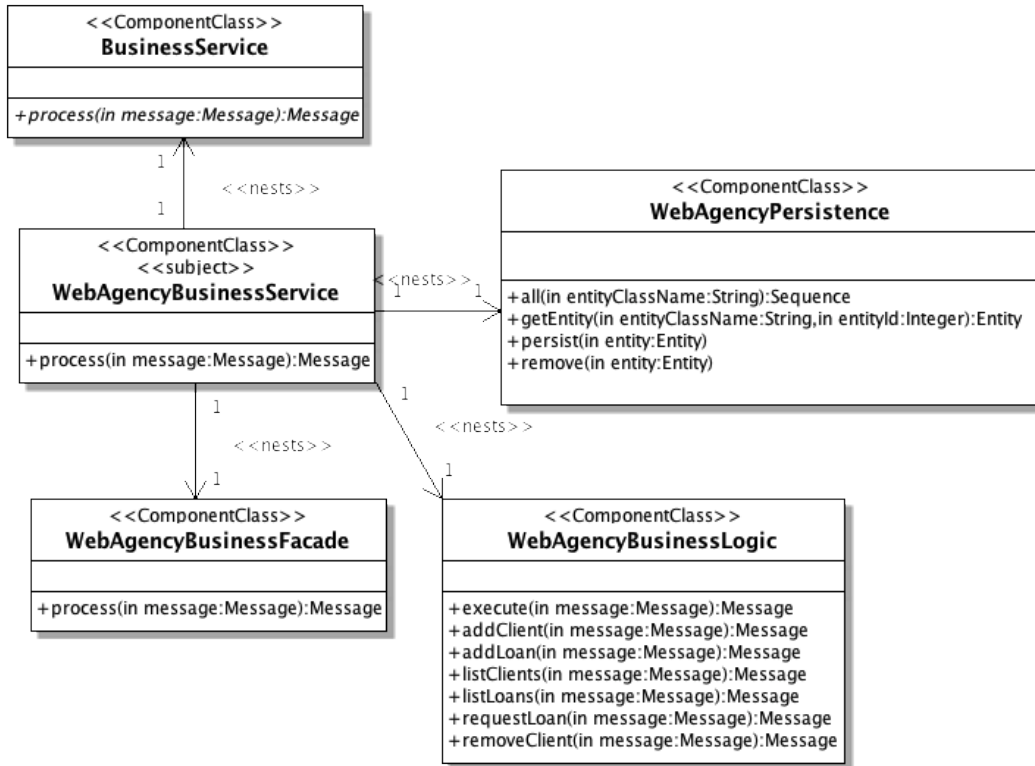


Figure B.2: *WebAgencyBusinessService* - Realization Structural Class Service

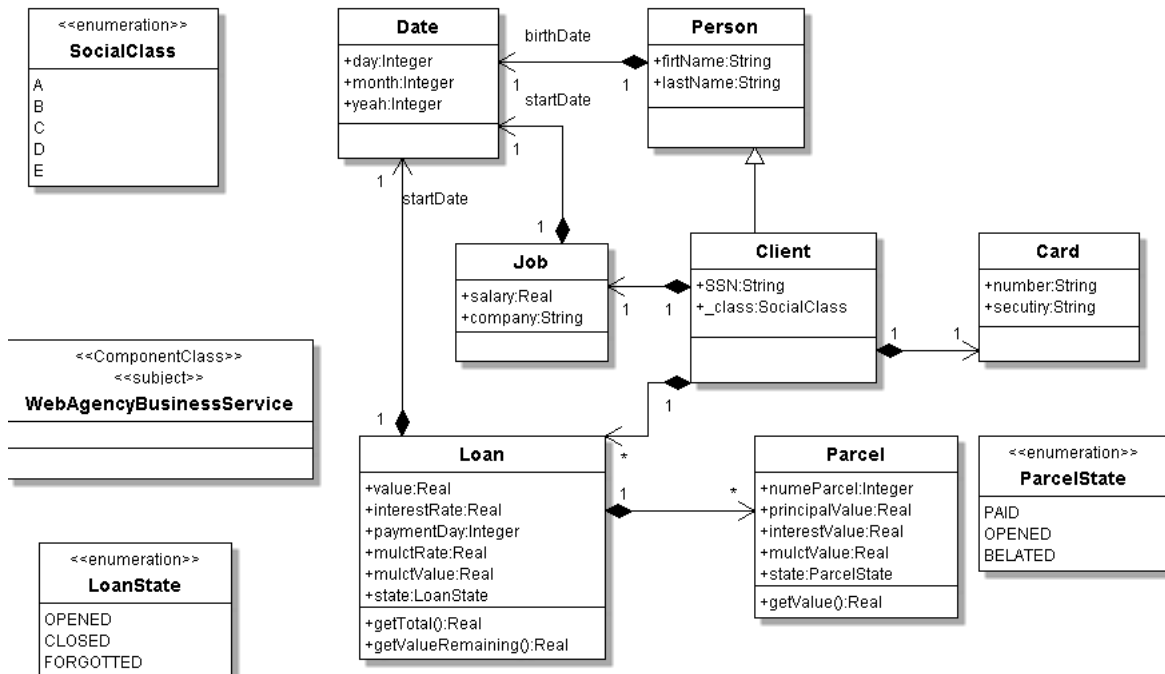


Figure B.3: *WebAgencyBusinessService* - Realization Structural Class Type

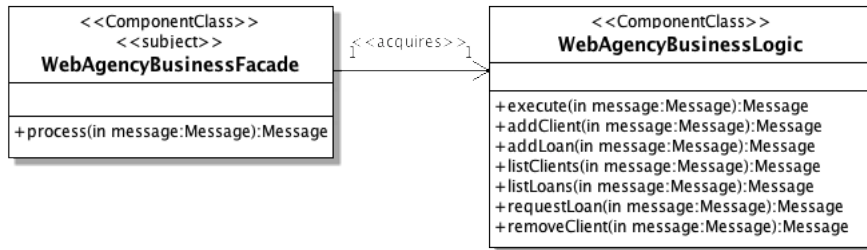


Figure B.4: *WebAgencyBusinessFacade* - Specification Structural Class Service

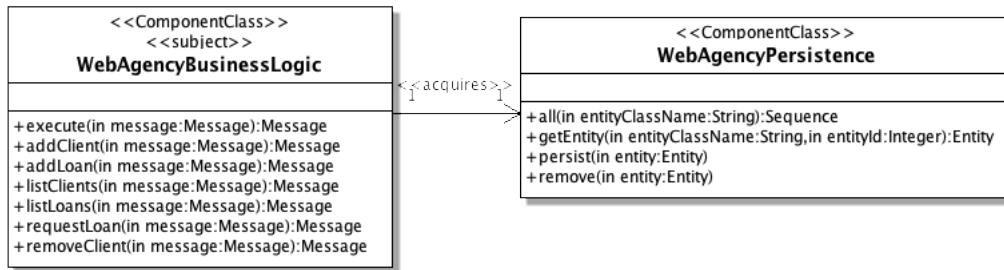


Figure B.5: *WebAgencyBusinessLogic* - Specification Structural Class Service

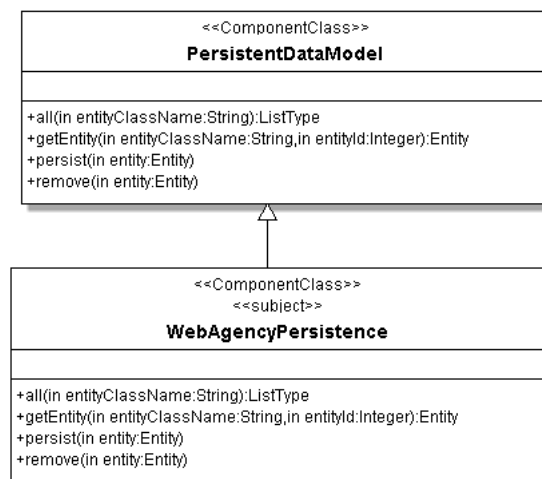


Figure B.6: *WebAgencyBusinessPersistence* - Specification Structural Class Service

B.2 Operational Specification

```
context: WebAgencyBusinessService :: process (message : Message) : Message
body: return self.webAgencyBusinessFacade.process (message);

context: WebAgencyBusinessFacade :: process (message : Message) : Message
body: do {
    var classifier : String := message.getClassifier();

    try {
        if (classifier = 'webAgencyBusinessLogic') {
            return self.webAgencyBusinessLogic.process (message);
        }
    }
    except (Exception) {
        raise 'Action_execution_raised_an_error.';
    }
}

context: WebAgencyBusinessLogic :: execute (message : Message) : Message
body: do {
    var operation : String := message.getOperation();

    if (operation = 'addClient') {
        return self.addClient (message);
    }
    elif (operation = 'addLoan') {
        return self.addLoan (message);
    }
    elif (operation = 'listClients') {
        return self.listClients (message);
    }
    elif (operation = 'listLoans') {
        return self.listLoans (message);
    }
    elif (operation = 'requestLoan') {
        return self.requestLoan (message);
    }
    elif (operation = 'removeClient') {
        return self.removeClient (message);
    }

    raise 'Operation_not_found!';
}

context: WebAgencyBusinessLogic :: addClient (message : Message) : Message
pre: message.getAttribute ('dependents').oclAsType (Integer) > 0
body: do {
    var firstName : String := message.getAttribute ('firstName').oclAsType (String);
    var lastName : String := message.getAttribute ('lastName').oclAsType (String);
    var SSN : String := message.getAttribute ('SSN').oclAsType (String);

    var company : String := message.getAttribute ('company').oclAsType (String);
```



```
var salary:String := message.getAttribute('salary').oclAsType(String);
var dependents:Integer := message.getAttribute('dependents').oclAsType(Integer);

var client:Client := new Client();

client.setFirstName(firstName);
client.setLastName(lastName);
client.setSSN(SSN);

client.setSocialClassKind( self.calculateSocialClassKind(dependents, salary) );

var job:Job := new Job();

job.setCompany(company);
job.setSalary(salary);

client.setJob(job);

self.webAgencyPersistence.persist(client);

message.setMessageKind(MessageKind::SUCCESS);
message.setAttribute('result', client);

return message;
}

context: WebAgencyBusinessLogic::addLoan(message:Message):Message
body: do {
    var loan:Loan := message.getAttribute('loan').oclAsType(Loan);

    self.webAgencyPersistence.persist(loan);

    message.setMessageKind(MessageKind::SUCCESS);
    message.setAttribute('result', loan);

    return message;
}

context: WebAgencyBusinessLogic::listClient(message:Message):Message
body: do {
    var clients:Sequence := self.webAgencyPersistence.all(
        'webagencybusinessservice.Client');

    message.setMessageKind(MessageKind::SUCCESS);
    message.setAttribute('result', clients);

    return message;
}

context: WebAgencyBusinessLogic::listLoans(message:Message):Message
body: do {
    var loans:Sequence := self.webAgencyPersistence.all(
        'webagencybusinessservice.Loan');
```

```
var SSN:String := message.getAttribute('SSN').oclAsType(String);

var selectedLoans:Sequence := loans->select(loan | loan.getClientSSN() = SSN);

message.setMessageKind(MessageKind::SUCCESS);
message.setAttribute('result', selectedLoans);

return message
}

context: WebAgencyBusinessLogic::requestLoan(message:Message): Message
body: do {
    var client:Client := message.getAttribute('client').oclAsType(Client);
    var numParcels:Integer := message.getAttribute('numParcels').oclAsType(Integer);
    var value:Real := message.getAttribute('value').oclAsType(Real);

    var loan:Loan := self.appraiser.proposeLoan(client, numParcels, value);

    message.setMessageKind(MessageKind::SUCCESS);
    message.setAttribute('result', loan);

    return message;
}

context: WebAgencyBusinessLogic::removeClient(message:Message): Message
body: do {
    var SSN:String := message.getAttribute('SSN').oclAsType(String);

    var client := self.webAgencyPersistence.getEntity(
        'webagencybusinessservice.Client', SSN.oclAsType(Integer));

    self.webAgencyPersistence.remove(client);

    message.setMessageKind(MessageKind::SUCCESS);

    return message
}

context: WebAgencyBusinessLogic::calculateSocialClassKind(message:Message): Message
body: do {
    var value:Real := 1 + (dependents / 5) + (5000 / salary);
    var indicator:Real := 10 * (2 / value);

    if (indicator >= 8) {
        return SocialClassKind::A;
    }
    elif (indicator >= 6 and indicator < 8) {
        return SocialClassKind::B;
    }
    elif (indicator >= 4 and indicator < 6) {
        return SocialClassKind::C;
    }
    elif (indicator >= 2 and indicator < 4) {
        return SocialClassKind::D;
    }
}
```

```
    }
    else {
        return SocialClassKind::E;
    }
}

context: Appraiser::calculateMaxParcelValue(client: Client): Real
body: do {
    var max: Real := client.getJob().getSalary() *
    self.getMaxBorrowingValueBySocialClass( client.getSocialClassKind() );

    var totalPending: Real := 0.0;

    (client.loans)->forEach(loan: Loan) {
        totalPending = totalPending + loan.getParcelValue();
    }

    return (max - totalPending);
}

context: Appraiser::proposeLoan(client: Client, numParcels: Integer, value: Real): Loan
body: do {
    var maxParcelValue: Real := self.calculateMaxParcelValue(xlient);

    var requestedParcelValue := value / numParcels;
    var loan: Loan = new Loan();

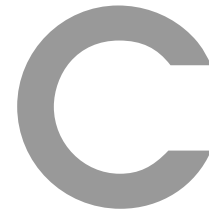
    if (requestedParcelValue <= maxParcelValue) {
        loan.setParcelValue(requestedParcelValue);
        loan.setNumParcels(numParcels);

        return loan;
    }
    else {
        loan.setParcelValue(maxParcelValue);
        loan.setNumParcels(numParcels);

        return loan;
    }
}

context: Appraiser::getMaxBorrowingValueBySocialClass(
    socialClass: SocialClassKind): Real
body: do {
    if (socialClass = SocialClassKind::A) {
        return 0.5;
    }
    elif (socialClass = SocialClassKind::B) {
        return 0.4;
    }
    elif (socialClass = SocialClassKind::C) {
        return 0.3;
    }
    elif (socialClass = SocialClassKind::D) {
```

```
        return 0.2;
    }
    else {
        return 0.1;
    }
}
```



CHROME - PIM

C.1 Structural Specification

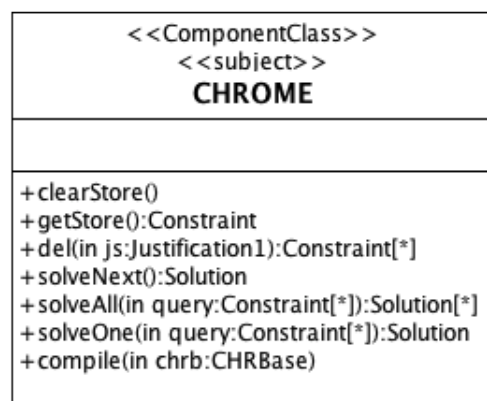


Figure C.1: *CHROME* - Specification Structural Class Service

C.1. STRUCTURAL SPECIFICATION

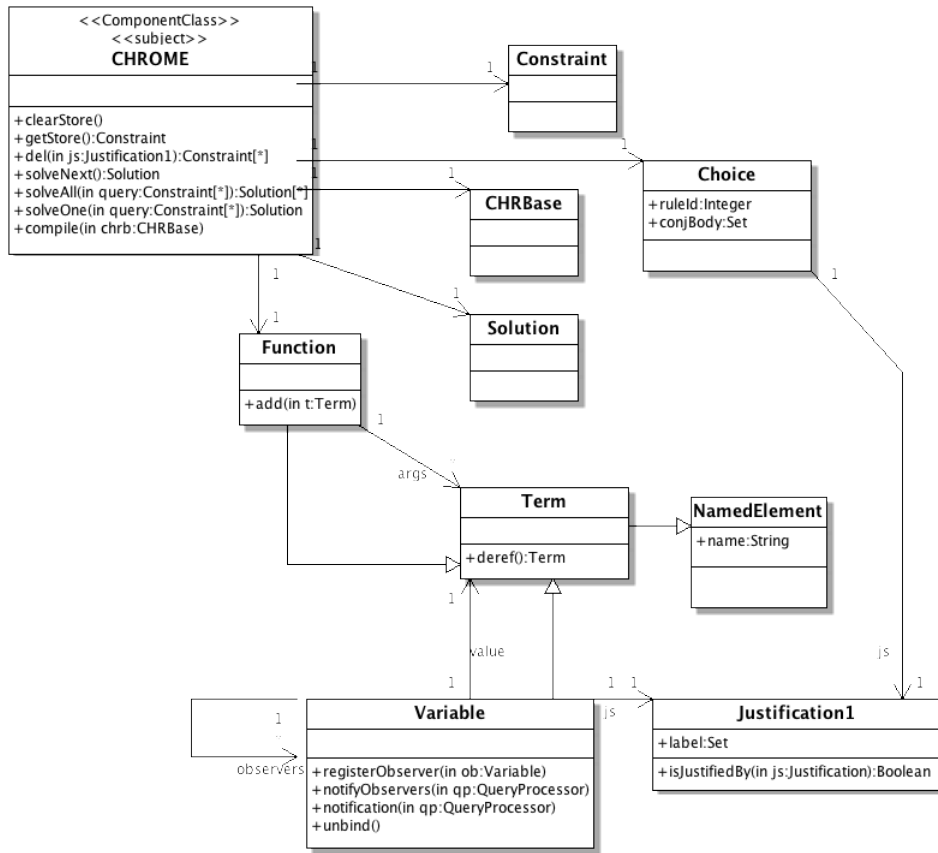


Figure C.2: *CHROME* - Specification Structural Class Type

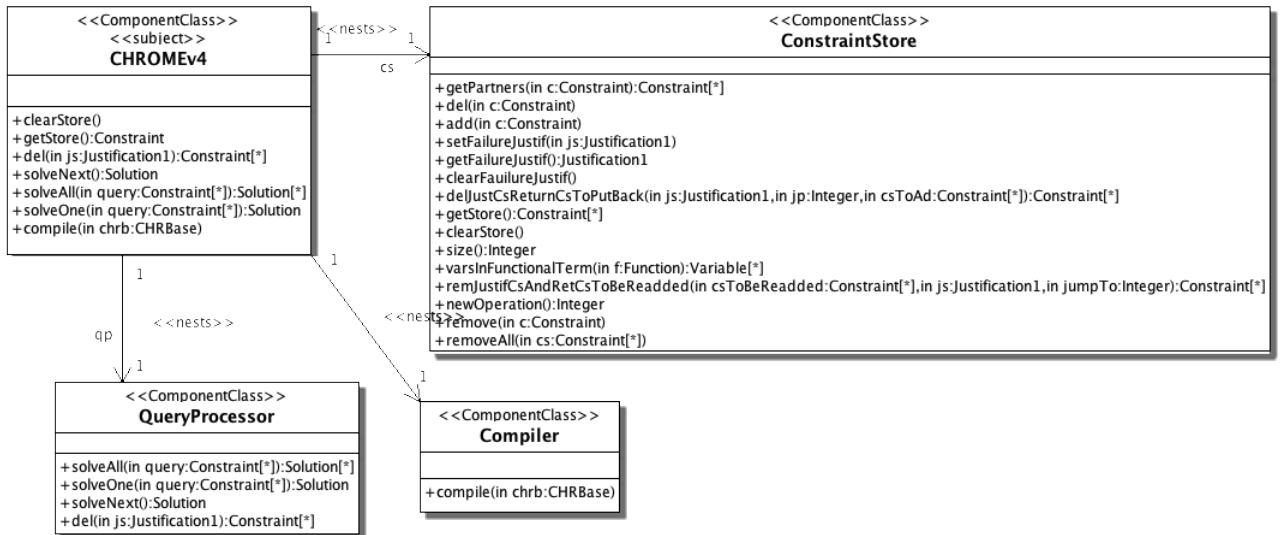


Figure C.3: *CHROME* - Realization Structural Class Service

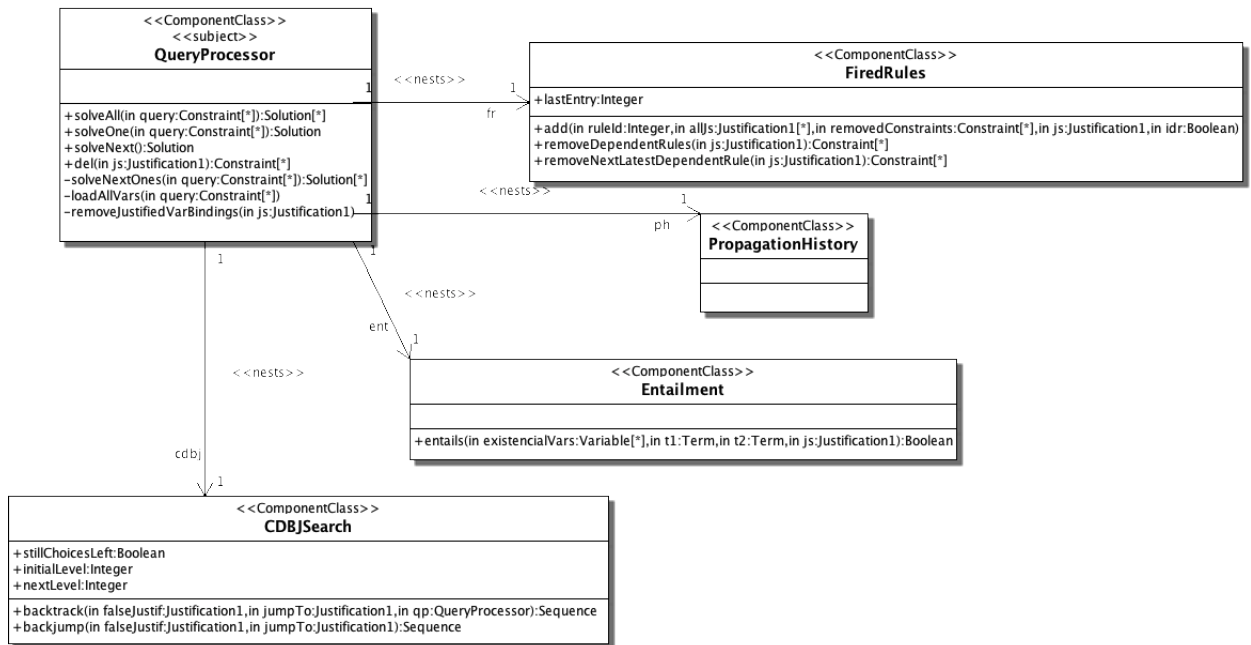


Figure C.4: *QueryProcessor* - Realization Structural Class Service

C.2 Operational Specification

```
context ConstraintStore :: add(constraint : Constraint)
body:
do {
  if (constraintTable2.containsKey(c.getName())) {
    var ht:Dict := self.constraintTable2.get( c.getName() );

    ht.put(c.hashCode(), c );
  }
  else {
    var ht:Dict := new DictionaryType();

    ht.put(c.hashCode(), c);

    constraintTable2.put(c.getName(), ht);
  }
}

context ConstraintStore :: getPartners(
  constraint : Constraint , constraintArity : Integer ): Sequence( Constraint )
body:
do {
  var result:Sequence(Constraint) = Sequence{};

  if (self.constraintTable2.containsKey(c.getName())) {
    var ht:Dict := self.constraintTable2.get( c.getName() );

    var constraints:Sequence(Constraint) := ht.values();

    constraints->forEach(constraint : Constraint) {
      if ((not constraint.removed) and (constraint.args.size() = constraintArity)) {
        result := result->including(constraint);
      }
    }
  }

  return result;
}

context ConstraintStore :: del(constraint : Constraint)
body:
do {
  if (self.constraintTable2.containsKey(c.getName())) {
    var ht:Dict := self.constraintTable2.get( c.getName() );

    if (ht.containsKey(c.hashCode())) {
      ht.remove(constraint);
    }
  }
}
```

```
context ConstraintStore :: getStore () : Constraint
body:
do {
  var result : Sequence (Constraint) = Sequence {};

  var constraintMaps : Sequence (DictionaryType) := self.constraintTable2.values ();

  constraintMaps ->forEach (constraintMap : DictionaryType) {
    var constraints : Sequence (Constraint) := constraintMap.values ()

    constraints ->forEach (constraint : Constraint) {
      result := result ->including (constraint);
    }
  }

  return result;
}

context FiredRules :: add (ruleId : Integer , allJs : Justification ,
removedConstraints : Sequence (Constraint) , idr : Boolean)
body:
do {
  var element : FiredRulesEntry := new FiredRulesEntry (
    ruleId , allJs , removedConstraints , idr);
  contents := contents ->including (element);
  self.lastEntry := self.lastEntry + 1;
}

context FiredRules :: removeNextLatestDependentRule (
justification : Justification) : FiredRulesEntry
body:
do {
  var i : Integer := self.lastEntry;
  while ( i > 0 ) {
    var element : FiredRulesEntry := contents ->at (i);
    if (element.js.isJustifiedBy (js)) {
      contents := contents ->excluding (element);
      self.lastEntry := self.lastEntry - 1;
      return element;
    }
    i := i - 1;
  }
  return null;
}

context FiredRules :: removeDependentRules (justification : Justification) : FiredRulesEntry
body:
do {
  var i : Integer := self.lastEntry;
  var element : FiredRulesEntry = new FiredRulesEntry ();
  var lastElement : FiredRulesEntry := new FiredRulesEntry ();
  var result : Sequence (Constraint) := Sequence {};
  while ( i > 0 ) {
    element := contents ->at (i);
```

```
    if (element.js.isJustifiedBy(js)) {
      element := contents>excluding(element);
      self.lastEntry := self.lastEntry - 1;
      result := result->union(element.constraintsToBeReadded);
      lastElement := element;
    }
    i := i - 1;
  }
  if (lastElement.isADisjunctiveRule) {
    contents := contents->including(lastElement);
    (lastElement.removedConstraints)->forEach(c: Constraint) {
      result := result->excluding(c);
    }
    self.lastEntry := self.lastEntry + 1;
  }
  return result;
}

context Entailment::deref(t:Term, js:Justification):Term
body:
do {
  if (t.ocIsKindOf(Variable) and t.ocAsType(Variable) <> null) {
    if ((t.ocAsType(Variable).value).ocIsKindOf(Variable)) {
      return self.deref(t.ocAsType(Variable).value, js);
    }
    else {
      js.mergeJustification(t.ocAsType(Variable).js);

      return t.ocAsType(Variable).value;
    }
  }
  else {
    return t;
  }
}

context Entailment::isVarLocal(v:Variable, localVars:Sequence(Variable)):Boolean
body: return self.localVars->includes(v);

context Entailment::bind(v:Variable, t:Term, js:Justification)
body:
do {
  v.value = t;
  v.js.mergeJustification(js);
}

context Entailment::entails(
localVars:Sequence(Variable), t1:Term, t2:Term, js:Justification):Boolean
body:
do {
  var j1:Justification := new Justification();
  var j2:Justification := new Justification();
  var d1:Term := self.deref(t1, j1);
  var d2:Term := self.deref(t2, j2);
```

```
js.mergeJustification(j1);
js.mergeJustification(j2);
if ((d1 = d2) and (not (d1.ocIsKindOf(Function)
    and d2.ocIsKindOf(Function)))) {
    return true;
}
if (d1.ocIsKindOf(Variable)) {
    if ((self.isVarLocal(t1.ocAsType(Variable), localVars)) and
        (not (d1.ocAsType(Variable).value.ocIsUndefined())))) {
        self.bind(d1.ocAsType(Variable), t2, js);
        return true;
    }
}
if (d2.ocIsKindOf(Variable)) {
    if ((self.isVarLocal(t2.ocAsType(Variable), localVars)) and
        (not (d2.ocAsType(Variable).value.ocIsUndefined())))) {
        self.bind(d2.ocAsType(Variable), t1, js);
        return true;
    }
}
if ((d1.ocIsKindOf(Function)) and (d2.ocIsKindOf(Function))) {
    if ( (d1.ocAsType(Function).args->size() =
        d2.ocAsType(Function).args->size()) and (d1.name = d2.name)) {
        var i:Integer := 0;
        var r:Boolean := true;
        while ((i < d1.ocAsType(Function).args->size()) and r) {
            var localTerm1 := d1.ocAsType(Function).args->at(i);
            var localTerm2 := d2.ocAsType(Function).args->at(i);
            r := r and self.entails(
                localVars, localTerm1, localTerm2, js);
            i := i + 1;
        }
        return r;
    }
    else {
        return false;
    }
}
}
```

```
context PropagHist::add(key:Integer, l:Sequence(Constraint))
body:
do {
    var ht:DictionaryType := self.propagationTable.get(key);

    if (not ht.ocIsUndefined()){
        ht.put(l.hashCode(), l);
    } else {
        ht := Dict{};

        ht.put(l.hashCode(), l);

        self.propagationTable.put(key, ht);
    }
}
```

```
}

context PropagHist::applied(key: Integer , l: Sequence(Constraint)): Boolean
body:
do {
    var ht: Dict := self.propagationTable.get(key);

    if (ht.ocIsUndefined()) {
        return false;
    }
    else {
        var clist: Sequence(Sequence(Constraint)) := ht.values();

        clist->forEach(storedConstraints: Sequence(Constraint)) {
            if ((storedConstraints->includesAll(l) && (storedConstraints->size() = l.size()))) {
                return true;
            }
        }
    }
}
```