Vrije Universiteit Brussel – Belgium
Faculty of Sciences
In Collaboration with Ecole des Mines de Nantes – France
and
Universidade Nova de Lisboa – Portugal
2002

Formal Definition of Object-Oriented Design Metrics

A Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
(Thesis research conducted in the EMOOSE exchange)

By: Aline Lúcia Baroni

Promoter: Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussel)
Co-Promoter: Prof. Dr. Fernando Brito e Abreu (Universidade Nova de Lisboa)

Abstract

Software measurement has become essential to good Software Engineering. However, most published works on software engineering concentrate on the coding activity. As quality indicators and predictors of structural problems, metrics should be available as early as possible in the software life cycle, and not dependant on source code availability.

This thesis intends to assist object-oriented software measurement, improving its use among software designers. For achieving this goal, the FLAME - a Formal Library for Aiding Metrics Extraction - is built with The Object Constraint Language (OCL), a part of the Unified Modeling Language (UML) standard. Based upon OCL, object-oriented design metrics definitions are formalized on a compositional way.

The definition of each metric is done upon the UML meta-model, at different levels of abstraction, including the meta-classes Package, Model Element, Generalizable Element, Classifier, Feature, Operation and Attribute.

The combination of the UML meta-model and OCL allows unambiguous metrics definition, which in turn helps increasing tool support for object-oriented metrics extraction.

The outcome is an elegant, precise and straightforward way to define metrics that may help to overcome several current problems. Besides, it is a natural approach since we are using the object technology to define metrics on the object technology itself.

The formalization renders possible the comparisons among different sets of metrics, as well as it may be used to establish a common vocabulary among different stakeholders. As consequence, our efforts contribute to the overall quality of the Software Engineering product and process.

KEY-WORDS

Software measurement, design metrics, object-oriented metrics, meta-model, formalization.

"If knowledge can create problems, it is not through ignorance that we will solve them."

Isaac Asimov

Acknowledgments

During this hard time, I could not forget to express my gratitude to those who helped me.

First, I want to thanks God for placing so many wonderful opportunities in our lives, even with difficult moments. To God, thanks for each door that is opened in front of our paths, and to all the ups and downs that we face in this roller coaster life. Thanks for the Portuguese sun and beaches, for the French food and for all Brazilian people spread around the world!

Second, thanks to my family for supporting me since the beginning, providing me a good upbringing and sharing with me my happy and sad moments. Also, for the incentive they gave me for doing this master and even for the motivation they transmitted to me by "distance". An enormous appreciation to my dear mom and dad, to my brothers, to my sisters, and to my grandparents.

Third, thanks to all those who make EMOOSE happens, including Annya Romanczuk, Jacques Noyé, Evelyne Moreau, Theo D' Hondt and Fernando Brito e Abreu. Special remembers to my advisors, who assisted me and directed my work, showing patiently to me who to conduct a research. Thanks also to all EMOOSE teachers.

Fourth, I could not forget the teachers (and also friends) in Brazil, who trusted on me, giving me the chance to face such a change of life while telling me their own experiences. All my recognition to Carlos Alberto Maziero, Alcides Calsavara, Robert Carlisle Burnett and Edson Scalabrin.

Fifth, thanks to all my friends. To the EMOOSE ones, who fought together, even during the "thesis period". My greatest respect to Olivier Constant and his friendly family. I can not forget also the laughs, parties, foods, drinks, shouts and weeping I shared with Rodrigo, Sebastian, Eduardo, Agustina, Boriss, Emmanuel, Jessie, Michael, Laura, Nuan and the Chinese guys. Furthermore, I thanks to Zhang He, Liang Jing, and An Keqiang, for teaching me a little of the Chinese patience. To Agus, Boriss and Sebas for the nice parties during ECOOP. To Jessie and Mich for being our photographers! To Olivier and Emmanuel for showing me pleasant things in France. To Edu, for the "pasta explosion parties" he created. To Sebas, for being a kind companion in our trip to Italy. To Zhang, for the Chinese medicine when I was sick.

My gratitude to Andrés Farias and Sinagi, who helped me since I was in Brazil, answering all my doubts about France and about the master. Additionally, my gratitude to Gustavo Bobeff and Carla, to Yann-Gaël, Hervé and Olivier Motelet.

Also, to the amazing friends I got in Portugal. A big hug to João Alcântara, for guiding me through Portugal, showing me patiently the paths to the supermarket, the means of transportation some beautiful places in Portugal, and for listening me so many times. To Valéria Pequeno, Júnior and Ronaldo Filgueira for hosting me several days in their house and for all their comprehension. Thanks João, Valéria, Júnior and Ronaldo also for the Brazilian food we prepared and ate together frequently. Thanks to Sofia Bráz

and Miguel Goulão, for showing me "the night" in Lisbon and for inviting me to know some precious things in the Portuguese culture. Thanks also for the reviews Miguel did for this work and for all the dinners with Sofia. My gratitude also to Iara Carvenale, AnaBela, João Araújo, Pedro Guerreiro and Fernando Brito and Abreu (now as a friend!).

To all my friends in Brazil, that kept "talking" with me by computational means. To Mauro Augusto, to Henrique Denes, to Alcides Calsavara, to Ana Mauad, to my students, and to all the members of the "ccdiretoria".

Thanks to all those who participated directly or indirectly in this academic year, contributing even in silence for my personal success, or for the success of the overall EMOOSE program.

Last, but not least, thanks to myself, for being stronger enough to support all the dark days, and for being able to pay attention in all the wonderful things encountered during this journey.

Aline Lúcia Baroni
August 2002

Listen as your day unfolds,
challenge what your future holds
Try to keep your head up to the sky
Lovers they may cause you tears
Go ahead release your fears
Stand up and be counted,
don't be shamed to cry

You gotta be...
You gotta be bad, you gotta be bold,
you gotta be wiser
You gotta hard, you gotta be tough,
you gotta be stronger
You gotta be cool, you gotta be calm,
you gotta stay together.
All I know, all I know
Love will save the day

Des'ree – You gotta be

Dedicatory

To my mom and dad
And also
To my brothers and sister
With all my love.

If I speak with the tongues of men and of angels, but have not love, I am become sounding brass, or a clanging cymbal.

And if I have the gift of prophecy, and know all mysteries and all knowledge; and if I have all faith, so as to remove mountains, but have not love, I am nothing.

And if I bestow all my goods to feed the poor, and if I give my body to be burned, but have not love, it profiteth me nothing.

Love suffereth long, and is kind; love envieth not; love vaunteth not itself, is not puffed up, doth not behave itself unseemly, seeketh not its own, is not provoked, taketh not account of evil; rejoiceth not in unrighteousness, but rejoiceth with the truth; beareth all things, believeth all things, hopeth all things, endureth all things.

Love never faileth: but whether there be prophecies, they shall be done away; whether there be tongues, they shall cease; whether there be knowledge, it shall be done away.

For we know in part, and we prophesy in part; but when that which is perfect is come, that which is in part shall be done away.

When I was a child, I spake as a child, I felt as a child, I thought as a child: now that I am become a man, I have put away childish things.

For now we see in a mirror, darkly; but then face to face: now I know in part; but then shall I know fully even as also I was fully known. But now abideth faith, hope, love, these three; and the greatest of these is love.

I Corinthians 13: 1-13 – The Bible

Contents

List of Figures

List of Tables

1 Introduction

SINTESYS

This chapter simply puts our work into context, and presents some important related efforts that have been carried out over the world. Additionally, it mentions the main conferences in the area and shows how this document is organized.

"Once you eliminate the impossible, whatever remains, no matter how improbable, must be truth."

Sherlock Holmes (by Sir Arthur Conan Doyle, 1859 - 1930)

1.1 THE CONTEXT OF THIS WORK

This thesis is encompassed in the Software Engineering area, which studies the concepts, techniques and tools that allow the improvement of the software production activities, from the condition of handcrafts in which it is met today to a condition of real engineering. The IEEE [ANSI/IEEE729, 1990] defines the Software Engineering as being:

"The adoption of a systemic, measurable and disciplined development, operation and maintenance of software."

Software Engineering describes the collection of techniques that apply an *engineering approach* to the construction and support of software products. Software engineering activities include managing, costing, planning, modeling, analyzing, specifying, designing, implementing, testing and maintaining. By *engineering approach* we mean that each activity is understood and controlled, so that there are few surprises as the software is developed.

The importance of Software Engineering cannot be understated, since software pervades our lives. The estimation of measures mentioned in the IEEE definition is the principal objective of the Experimental Software Engineering area. This intends, by induction of experimental base, to comprehend the strong and weak points of methods, tools and techniques, and to express quantitatively the relations between the software development process and the quality of the resulting products. It is widely recognized that software quality is critical in the development of software systems, especially the large scale ones. High quality software can reduce the software maintenance or testing costs, enhance the potential reuse [Xie et al., 2000].

To evaluate the software quality more quantitatively and objectively, software metrics appears to be a powerful and effective technology to assess the software quality. DeMarco [DeMarco, 1982] succinctly state its importance saying:

"You can't control what you can't measure".

Currently software metrics are the core technology of software quality evaluation, which empirically and objectively assign a number (or symbol) to software, in order to characterize certain specific quality attributes.

The group QUASAR[1] (Quantitative Approaches in Software Engineering and Reengineering), inside the New University of Lisbon – Portugal, investigates how to achieve the goals of the Experimental Software Engineering, throughout quantitative analysis and more precisely, through software metrics.

There are basically, three types of software entities that can be measured: processes, products and resources. Most of our efforts in this thesis focus on the software product design metrics, which measure the quality of software products, by analyzing its design at the beginning of the software life cycle.

Generally speaking, there are some main reasons why software measurement becomes important in software industry [Xie et al., 2000]:

[1] For getting more information about the QUASAR group, please visit the site http://ctp.di.fct.unl.pt/QUASAR

- It can help to fully understand both the design and architecture information of the software system. It can help to comprehend the process of development by applying the process evaluation in the activity of software development.
- It can aid to discover the underlying errors in the software design at the early stage of software development life circle. It can also assist the task of software test.
- It can evaluate the quality of the software and provides the cost estimation of a software project. It becomes easier to estimate and plan new activities based on measurement, to control the progress and to improve the process, making it more cost-effective in the future.
- It can help to determine the effect of the object technology, especially reuse technology applied in the software development according to some quantitative evaluation such as productivity, quality, lead time, maintainability, etc. It also enables estimating the costs and benefits of different reuse strategies.
- Software metrics, especially reusability metrics based on object-oriented metrics, can assess the quality and reusability of software components which can assist extracting potentially useful or reusable modules or components in legacy system efficiently, so that a valuable resource can be attained to build a new system on.

These reasons serve as motivation for this work. Additionally, object-oriented technology is becoming increasingly popular in industrial software development environments. This technology offers support to provide software product with higher quality and lower maintenance costs.

However, none of these items is useful if metrics are not extracted in practice, due to the lack of standardization of metrics definitions and tools to perform this task. In order to break these limitations, this thesis proposes an approach for standardization of metrics definitions, which can consequently contribute to the increase tool support and software measurement upon object-oriented designs.

1.2 RELATED WORKS AROUND THE WORD

This section shows some of the important works that related with Experimental Software Engineering, although none of them has the same objectives.

1.2.1 European Projects

In 1986, in United Kingdom, the research project entitled *Structured-Based Software Measurement* [Elliott, 1988] started. This project was intended to build on existing research into formal modeling, analysis and measurement of software structure. It was carried out at South Bank Polytechnic's Centre for Systems and Software Engineering in London, UK.

Among others, results of this project can be found in Fenton [Fenton, 1991].

From 1989 till 1992, the Project METKIT (Metrics Educational Toolkit) [Metkit, 1993] of the European Community was created. METKIT was a collaborative project part-funded by the European Commission under the ESPRIT (European Strategic Program for Research in Information Technology) program [ESPRIT, 1990]. The aim of METKIT was to raise awareness and increase usage of software measures within European industry by producing educational material aimed to both industrial and academic audiences. The project developed an integrated set of educational materials to teach managers, software developers and academic students how to use measurement to understand, control and then improve software. An outcome of METKIT was the book of Fenton [Fenton, 1991] which gives an excellent overview of the area of software measures.

Other ESPRIT projects dealing with software engineering measurement are:

- AMI (Applications of Metrics in Industry), from 1990 to 1992;
- MUSIC (Metrics for Usability Standards in Computing), from November 1990 to November 1993;
- MUSE (Software Quality and Reliability Metrics for Selected Domains: Safety Management and Clerical Systems), from 1987 to 1990;
- PYRAMID (Promotion of Metrics), from October 1990 to October 1992;
- COSMOS (Cost Management with Metrics of Specification) from February 1989 to 1994 and
- MERMAID (Metrication and Resource Modeling Aid) from 1988 to 1992.

These projects are described on [Abreu et al., 2000] and on [Zuse and Bollmann-Sdorra, 1992].

1.2.2 American Projects

In United States (US) and Canada, software measurement begun early in the seventies. In US various software measurement groups and activities have been established since the mid-seventieth.

Many measurement programs have been established under the auspices of the Software Engineering Institute [Carnegie Mellon University] to provide a platform from which increased used of measurement within organizations can be promoted.

A number of major companies in American countries use software measures extensively, as AT&T, NASA, Motorola, Hewlett Packard, etc. In addition, the University of Maryland has a long tradition (more than 15 years) of measurement.

1.3 CONFERENCES

Software measurement has been, for many years, the focus of several conferences. We mention some of them below.

- ECOOP Quantitative Approaches on Software Engineering (6th edition on 2002);
- Brazilian Symposium on Software Engineering (16th edition on 2002);
- International Conference on Software Engineering (ICSE);
- International Workshop on Software Reliability Engineering (ISSRE);
- IEEE Software Metrics Symposium;
- International Software Engineering Standards Symposium (ISESS);
- International Conference on Software Quality (ICSQ);
- Software Quality Week;
- NASA Software Engineering Workshop;
- Annual Oregon Workshop on Software Metrics (AOWSM). Since 1989;
- International Software Engineering Standards Symposium (ISESS);
- International Software Metrics Symposium (IEEE);
- Software Metric Symposium of DECollege and *Software Metriken* of ORACLE.

1.4 DOCUMENT OVERVIEW

This document is divided in 6 chapters. Chapters 2, 3 and 4 elucidate the state-of the-art required for understanding the research carried out in this thesis. Chapters 5 and 6 show our contribution. Chapter 7 presents our conclusions and further work.

In chapter 2, we introduce the fundamentals of software measurement, including some definitions of the terms *metric*, *measure* and *measurement*. We outline the needs for software measurement and their objectives, mentioning how quantitative analysis can help us.

In chapter 3, the Object Constraint Language (OCL) - a language for improving the precision of models design - is described for being used along this thesis.

Chapter 4 depicts the UML meta-model that is, together with OCL, the basis of our research. We explain how the UML meta-model is organized, and which are its main packages and respective components. Some restrictions applied to the UML meta-model are expressed with OCL.

Chapters 5 and 6 explain the contributions of this work. To deal with software measurement, being able to apply it at the beginning of software life cycle, we created a library of auxiliary functions that is further used to formalized metrics definitions. This collection, named FLAME – Formal Library for Aiding Metrics Extraction – is formalized with OCL upon the UML meta-model, through navigations over the meta-classes of the latter.

The functions in FLAME serve as input to formalize the definitions of different metrics sets in the literature, as showed in chapter 6. Both FLAME and the metrics sets are verified and validated with the architecture explained in chapter 6.

Finally the conclusion of our work, and some future steps are given.

2

Software Measurement

SYNTHESIS

It has been three decades since the software engineering community proposed the use of software measurement to evaluate and guarantee the quality of both the software life cycle and the final product that turns out from this process. This estimation uses software complexity metrics as input. The latter are the results of quantitative analysis of software product, process and resources attributes.

The term resource refers to people, time, computers, space, tools, languages, communications facilities and others. The term product involves all the deliverables generated during the software life cycle, like requirements specifications, requisites, models, source code, test cases, user manuals and installation manuals. The term process relates to the way resources are organized to develop the product.

The certification of systems according to quality standards such as ISO9001 [ISO9001] implies the adoption of metrics in the contexts of process and product. In the product point of view, many metrics have been proposed to quantify different aspects of its complexity. This is the case of the proposals from McCabe [McCabe, 1976], Halstead [Halstead, 1977], Kafura [Kafura and Henry, 1981] and others. In the process point of view, metrics are used in the planning, organization and control of projects and software development [Paulk et al., 1993][Koch, 1993][Konrad et al., 1995].

This chapter examines software measurement, introducing its concepts, showing the existing categories to classify metrics and providing an overview through the history of the most known metrics and measures.

"A major difference between a "well developed science" such as physics and some of the less "well-developed" sciences such as psychology or sociology is the degree to which things are measured."

Fred S. Roberts, 1979

2.1 INTRODUCTION: MEASUREMENT IN EVERYDAY LIFE

Measurement lies at the heart of many systems that govern people's lives. Economic measurements determine price and pay increases. Measurements in radar systems enable the prevention of aircraft collisions when direct vision is obscured and also control the speed limits for driving. Medical system measurements enable doctors to diagnose specific illnesses, and control regular things like the blood pressure. Measurements in atmospheric systems are the basis for weather prediction.

Measurement is not solely the domain of professional technologists. Each individual uses it in everyday life. Price acts as a measure of value of an item in a shop, while people calculate the total bill to make sure the shopkeeper returns the correct change. Height and width measures are used to ensure that clothes fit properly. When traveling, distances, routes, speed and predictions involving time and probably stops to refuel can be estimated. Consequently, measurements contribute to understand the world, to interact with our surroundings and to improve human lives.

In Software Engineering, software measurement has become essential. Many of the best software developers measure characteristics of the software to get some sense of whether the requirements are consistent and complete, whether the design is of high quality, and whether the code is ready to be tested [Fenton and Pfleeger, 1997]. Effective project managers measure attributes of process and product to be able to tell when the software will be ready for delivery and whether the budget will be exceeded. Informed customers measure aspects of the final product to determine if it meets the requirements and is of sufficient quality. Maintainers must be able to assess the current product to see what should be upgraded and improved [Fenton and Pfleeger, 1997].

This chapter presents the fundamentals of measurement, some concepts, the historic overview, why measurement is important and how that measurement supports research in the Computer Science area.

2.2 SIMPLE CONCEPTS

The discussion starts by presenting some definitions and concepts allied with measurement. Although the terms *measure*, *measurement* and *metric* are often used interchangeably, it is important to note the subtle differences among them.

2.2.1 One Simple Definition of Metric

The American Heritage Dictionary [Mifflin, 2000] defines a metric as:

Definition *2.1* – **Metric** (According to the American Heritage Dictionary)

1. A standard of measurement.
2. A geometric function that describes the distances between pairs of points in a space.

The Cambridge International Dictionary of English [Press, 2000] defines a metric as:

Definition *2.2* – **Metric** (According to the Cambridge International Dictionary of English)

A system of measurement that uses meters, centimeters, liters etc.

Definitions 2.1 and 2.2 are insufficient for the purposes of software measurement, and need to be refined for better understanding of this work. Going deeper into details, the next section examines a definition based on Mathematics.

2.2.2 A Mathematical Connotation

Mathematicians define a metric more rigorously. The term applies to a real function which measures the distance between two entities [Mansfield, 1963]. It is part of the set theory that deals with sets in which any two elements have a distance from each other.

Definition *2.3* – **Metric** (According to Mansfield)

Let A be a set of objects, let R be the set of real numbers, and let μ be a one-to-one function such that $\mu{:}A \otimes A \rightarrow R$, where $\otimes$ denotes the Cartesian product of A with A. Then, μ is a metric for A if and only if:

$\forall \alpha, \beta \in A{:}\ \mu\,(\alpha, \beta) \geq 0;$ *(P1)*

$\forall \alpha, \beta \in A{:}\ \alpha = \beta \Rightarrow \mu\,(\alpha, \beta) = 0;$ *(P2)*

$\forall \alpha, \beta \in A{:}\ \mu\,(\alpha, \beta) = \mu\,(\beta, \alpha);$ and *(P3)*

$\forall\ \alpha, \beta, \gamma \in A{:}\ \mu\,(\alpha, \gamma) \leq \mu\,(\alpha, \beta) + \mu\,(\beta, \gamma).$ *(P4)*

The *Euclidean metric* (*Euclidean distance*), for example, corresponds to the shortest distance between two points α and β in a space and it satisfies the four properties above (P1 to P4).

It is possible to verify, that in the case of software, these properties are not universal.

- Consider that α and β are 2 functions and that the metric μ, for P2, is the number of times the functions α call the function β, if one of the functions is recursive then $\mu\,(\alpha, \beta) > 0$. P2 is not verified.
- The property P3 (commutability) implies that there is no direction in the relation that the μ metrics intends to quantify. In software, the relations among entities as classes, functions and variables, frequently have an associated direction. For example the number of attributes class α inherits from class β can not be the same if the inverse specialization occurs (class β inheriting from class α). Due to this, P3 is also not verified.

- Consider three functions α, β and γ, defined in a module where the state (represented by a set of attributes) is shared. If the metric μ (δ, ε) represents the number of variables shared by the functions δ and ε, the property P4 is not always true. For example, consider a module which shares the attributes x, k, z, w and n along the functions α, β and γ. Function α uses the attributes x and z. Function β uses n, w and k. Function γ uses z, k and x. According to this, $\mu(\alpha, \gamma) = 2$, $\mu(\beta, \gamma) = 1$ and $\mu(\alpha, \beta) = 0$, which breaks the property P4.

Due to these reasons, the topological connotation of the term *metric* is not, in general, used in the context of software. In spite of this, software metrics can express some distance, even if not *Euclidian*.

2.2.3 A Software Engineering Connotation

The concept of a metric as a measure of the *distance* between two items in a set *A* has very little meaning in the world of software.

There is no standard definition of measurement for software artifacts that is universally accepted [Archer and Stinson, 1995]. Anyway, two of them are reproduced here. The first one, extracted from [Abreu et al., 2000] says:

Definition *2.4* – **Measurement** (According to Abreu)

"Measurement is the experimental process in which, to precisely describe the entities or events in real world, numbers or other symbols are assigned to its attributes by using a given scale. The result of the measurement is called measure."

The second, extracted from [Fenton and Pfleeger, 1997], says:

Definition *2.5* – **Measurement** (According to Fenton)

"Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules."

Thus, measurement captures information about attributes of *entities*. An entity is an object (such as a person or a room). Entities are described by the characteristics that are important to distinguish one entity from another. An *attribute* is a feature or property of an entity.

In Software Engineering, *entities* can be:

- The products (*deliverables*) generated as outcomes from the software life cycle, as requirements specifications, documents with design, source code, tests, etc.;
- The development environment, comprised of a group of tools;

- The stakeholders[2].

The *events* correspond to phases in the software life cycle (planning, analysis, design, codification, testing, debugging, and maintenance) or to activities and incidents not associated to a specific phase, as configuration management, inspections and other nature of reviews.

The *attributes* that can be measured depend on the entity or event considered. Table 2.1 contains some examples of measurable attributes.

Entity or Event	Measurable Attributes
Requirements Specification	Words, phrases, paragraphs, verbs, adjectives
Block Diagram	Modules, coupling between modules, dependencies
States Diagram	States, transitions, points of synchronism
Source Code	Files, lines, identifiers, comments
Team of Analysts	Years of experience, age, degree of knowledge, working hours
Case Tools	Supported methods, costs for acquisition, costs for maintenance
Debugging of Programs	Duration, human and computational resources involved
Configuration Management	Produced versions, number of upgrades, number of distinct versions being currently used

Table 2.1 – *Examples of Measurable Attributes from Entities or Events*

A *metric* is a quantification of a specific characteristic from an entity in the real world, which can be inferred from a set of attributes. In the case of Software Engineering, it turns out:

Definition *2.6* – **Metric** (According to Abreu [Abreu et al., 2000])

A software metric is a combination from measures of attributes belonging to a software product, or to its development process, which shows quantitatively some of its characteristics.

To avoid confusion, this document uses the definitions of the IEEE Standard Glossary of Software Engineering Terms [ANSI/IEEE729, 1990], which is the following:

Definition *2.7* – **Measure** (According to IEEE)

A measure provides a quantitative indication of the extent, amount, dimensions, capacity or size of some attribute of a product or process.

Definition *2.8* – **Measurement** (According to IEEE)

[2] The term *stakeholder* has been used in several works related to Computer Science. It represents all kinds of people that are related with one software product, as analysts, developers, testers, designers, programmers, etc.

A measurement is the act of determining a measure.

Definition *2.9* – **Metric** (According to IEEE)

A metric is a quantitative measure of the degree to which a system, component or process possesses a given attribute.

When a single data point has been collected (e.g., the number of errors uncovered in the review of a single module), a measure has been established. Measurement occurs as the result of the collection of one or more data points (e.g., a number of module reviews are investigated to collect measures of the number of errors found during each review). A software metric relates to the individual measures in some way (e.g., the average number of errors found per review or the average numbers of errors found per person-hour expended on reviews[3]) [Pressman, 2000].

2.3 MEASUREMENT IN SOFTWARE ENGINEERING

Engineering disciplines use methods that are based on models and theories. Underpinning the scientific process is measurement. It is used to apply the theory to practice. It's difficult to imagine engineering areas such as electrical, mechanical and civil, without measurement. But this activity is still considered a luxury in Software Engineering. According to [Fenton and Pfleeger, 1997], and considering software products, most development projects:

- Fail to set measurable targets. For example, the company says the product will be user-friendly, reliable and maintainable without specifying clearly and objectively what these terms mean. The outcome is that, when the software is complete, it is not practical to say if the goals where met or not. This situation illustrates Gilb's principle of fuzzy targets [Gilb, 1988]: projects without clear goals will not achieve their goals clearly.
- Fail to understand and quantify its component costs. Most projects can not differentiate the cost of design from the cost of coding or testing. It is not possible to control costs without measuring the costs of each phase of development.
- Do not quantify or predict the quality. This makes some answers impracticable, as how reliable the product will be or how much work is necessary to make it portable.
- Allow anecdotal evidence trying revolutionary technologies, without determining if the technology is really efficient and effective.

Measurements are often done infrequently, inconsistently and incompletely [Fenton and Pfleeger, 1997]. These faults can be frustrating for those who want to make use of the results and forbid the repetition of successful measurement plans, simply because they do not exist. Thus, the lack of measurement in Software Engineering is compound by the lack of a rigorous approach.

It is clear from other engineering disciplines that measurement can be effective, if not essential, in making characteristics and relationships more visible, in assessing the magnitude of problems, and in fashioning a solution to problems. Software production involves a considerable investment of energy and money and it is time for software engineering to embrace the engineering discipline that has been so successful in other areas.

This work contributes on the proliferation of software measurement, as it makes the definition of metrics clear and applicable to object-oriented software, since its design is available.

[3] This assumes that another measure, person-hours expended, is collected for each review.

2.4 OBJECTIVES FOR SOFTWARE MEASUREMENT

Measurement is not only useful but also necessary. It is needed for verifying the status of the projects being developed and which are the resources and processes involved. How to affirm that a project is good or bad, healthy or not, with or without quality, if there are no measures of its features (goodness, health, quality, etc.)?

It is essential that good and bad characteristics of projects are kept, particularly if the company's accreditation is required by any form of certification [Eman et al., 1997; ISO9001; ISO9126; Paulk et al., 1993]. In other words, it is necessary to control projects rather than just running them.

It is not enough to think that, with measurement, control is gained. For that, the measurement objectives must be specific, tied to what managers, developers and users need to know. It is the set of goals that tell how the measurement information can be used once it is collected.

Table 2.2 exemplifies the information needed to understand and control software, separated by manager and developers perspective. It is adapted from [Fenton and Pfleeger, 1997] and [Pressman, 2000].

Perspective	Required Information
Manager	What does each process cost? What is the time and effort involved in each process?
Manager	How productive is the staff? How much time it takes to specify the system, design it, code it and test it? What was the software development productivity on past projects?
Manager	How good is the code being developed? Which is the number of faults and failures meet?
Manager	Will the user be satisfied with the product? Which of the requested requirements have actually been properly implemented? What was the quality of software that was produced?
Manager	How can we improve? Can we compare two design methods to see which one yields the higher quality code? How can past productivity and quality data be extrapolated to the present?
Engineer	Are the requirements testable?
Engineer	Have we found all the faults? Which is the number of faults found in the code? Do we need more inspections and tests?
Engineer	Have we met our product and process goals?
Engineer	What will happen in the future? Can we make some predictions based on the measures found? How can the past help us plan and estimate more accurately?

Table 2.2 – *Examples of Required Information According to Distinct Perspectives*

Table 2.2 shows that measurement is important for three basic activities. First, there are measures that help *understanding* what is happening during the development and maintenance. The current situation is estimated and baselines are established to set goals for future behavior. In this sense, measurements make aspects of process and product more visible, giving a better perception of relationships among activities and the entities they affect.

Second, measurement allows *controlling* what is happening on projects. Using the baselines, goals and understanding of relationships, prediction is likely to happen and changes in the product and/or process can help meeting the aims of stakeholders.

Third, measurement encourages *improving* processes and products. For instance, it is expected to increase the number or type of design reviews based on specification quality measures and predictions of design quality.

Measurement can be applied to software process with the intent of improving it on a continuous basis. It can be used throughout a software project to assist in estimation, quality control, productivity assessment, and project control. Finally, measurement can be used by software engineers to help assess the quality of technical work products and to assist in tactical decision making as a project proceeds [Pressman, 2000].

Metrics should be collected so that process and product indicators can be ascertained. Process indicators enable a software engineering organization to gain insight into the efficacy of existing process (i.e., the paradigm, software engineering tasks, work products, and milestones). They enable managers and practitioners to judge what works and what doesn't. Process metrics can lead to a long-term software process improvement.

Project indicators enable a software project manager to (1) assess the status of an ongoing project; (2) track potential risks; (3) uncover problem areas before the "go critical"; (4) adjust work flow or tasks; and (5) evaluate the project team's ability to control quality of software engineering work products [Pressman, 2000].

Kelvin [Kelvin, 1891-1894], once said:

"When you can measure what you are speaking about and express it in numbers, you know something about it; but when you cannot measure, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts advanced to the stage of a science."

Measurements make concepts more visible, understandable and controllable. Scientists should work to create ways of measuring the world, and improve existing measures. Of course some existing metrics are not as refined, in the sense to be made precise, as they should be. In this work we improve the specification of existing measures by making them more precise. Chapter 3 clarifies the needs for precision.

2.5 AN HISTORIC OVERVIEW

In this section some important research works are mentioned. First, the early works that lead to the measures available today are introduced. Following, a classification will divide the different kinds of metrics for further explanation of the ones interesting for this work. A brief overview of the works in the categories selected for the purposes of this document is done. Finally the history of metric for the object-oriented paradigm is outlined.

2.5.1 Software Measurement Ground Works

Estimation in the Software Engineering area has started approximately 3 decades ago. The first known reference is from Rubey [Rubey and Hartwick, 1968] and its title – *Quantitative Measurement of Program Quality* – reveals the concerns with quality.

The reasons for creating or inventing software measures are based on the knowledge that program structure and modularity are important considerations for the development of reliable software. Most software specialists agree that higher reliability is achieved when software systems are highly modularized and module structure is kept simple [Schneidewind, 1977]. Regarding modularity, Parnas [Parnas, 1975], Myers [Myers, October, 1977] and Yourdon [Yourdon, 1975] have discussed it trying to quantify it in a way or another. Parnas suggested that modules should be structured so that a module has no knowledge of the internal structure of other modules. Myers described the concept of *module strength*. These factors affect cost and quality of software, as pointed out by [Porter and Selby, 1990].

The earliest software measure is the *Lines of Code Measure* (*LOC*), which is discussed and used till today [Park, 1992]. In 1974, [Wolverton, 1974] made one of the earliest attempts to formally measure programmer productivity using LOCs. He proposed object instructions per man-month as a productivity measure and suggested what he considered to be typical code rates. The basis of the Measure LOC [Shepperd and Ince, 1993] is that program length can be used as a predictor of program characteristics such as reliability and ease of maintenance. Despite, or possibly even because of the simplicity of this metric, it suffered from severe criticism. In the sixties the *Source Lines of Code* (SLOC) were counted by the number of 80-column cards. In [Basili and Hutchens, 1983], Basili suggested that the Metric LOC should be regarded as a baseline metric enabling comparisons with other metrics. Nowadays the Measure LOC has been mentioned in more than ten thousand papers [Zuse].

In 1975, the term *Software Physics* was created by [Kolence], and in 1977 Halstead [Halstead, 1977] introduced the term *Software Science*. The idea behind these terms was to apply scientific methods to the properties and structures of computer programs. Kolence's theory connects such traditional performance measures as turnaround time, system availability, and response time with traditional management measures such as productivity, cost per unit service, and budgets. Software Physics was among the first theories to deal exclusively with computer sizing and workloads [Morris].

The most famous measures, which are still heavily discussed today [Zuse] and which were created in the middle of the seventies are the Measures of McCabe [McCabe, 1976] and of Halstead [Halstead, 1977]. McCabe derived a software complexity measure from graph theory using the definition of the *cyclomatic number*. McCabe interpreted the cyclomatic number as *the minimum number of paths* in a flowgraph. He argued that the minimum number of paths determines the complexity (*cyclomatic complexity*) of the program.

"The overall strategy will be to measure the complexity of a program by computing the number of linearly independent paths v(G), control the "size" of programs by setting an upper limit to v(G) (instead of using just physical size), and use the cyclomatic complexity as the basis for a testing methodology."

McCabe also proposed the *measure of essential complexity,* which may be the first measure which analyzes unstructuredness based on primes. In [Zuse, 1991; Zuse and Bollmann-Sdorra, 1989] the authors showed that the idea of complexity of the

Measure of McCabe can be characterized by three simple operations. The authors derived this concept from measurement theory.

The Measures of Halstead are based on the source code of programs. Halstead showed that estimated effort, or programmer time, can be expressed as a function of operator count, operand count, or usage count [Halstead et al., 1976]. Halstead's method has been used by many organizations, including IBM at its Santa Teresa Laboratory [Christensen et al., 1981], General Electric Company [Fitzsimmons and Love, 1978], and General Motors Corporation [Halstead et al., 1976], primarily in software measurement experiments. Today the most used Measures of Halstead are the *Measures Length*, *Volume*, *Difficulty* and *Effort* [Zuse].

In 1977 Laemmel and Shooman [Laemmel and Shooman] have examined *Zipf's Law*, which was developed for natural languages, and further extended to be useful in programming languages. Zipf's Law is applied to operators, operands, and the combinations of operators and operands in computer programs. The results show that Zipf's Law holds for computer languages, and complexity measures can be derived which are similar to those of Halstead.

Two other software complexity measures, namely *Interval-Derived-Sequence-Length* (IDSL) and *Loop-Connectedness* (LC), were proposed by [Hecht, 1977] and are discussed in [Zuse, 1991], but they are not well known. Also in the same year, Gilb [Gilb, 1977] published a book entitled *Tom Gilb: Software Metrics*, which is one of the first books in the area of software measures.

In 1978, another proposal of software complexity measurement was done by [McClure]. Moreover, Jones [Jones, 1978] published a paper where he discussed methods to measure programming quality and productivity.

In the next year, [Belady, 1979] proposed the *Measure BAND* that is sensitive to nesting, and [Albrecht, 1979] introduced the *Function-Point* method in order to measure the application development productivity.

In 1980, [Oviedo, 1980] developed a *Model of Program Quality.* This model treats control flow complexity and data flow complexity together. Oviedo defines the complexity of a program by the calculation of control complexity and data flow complexity with one measure. In addition, Curtis published an important paper about software measurement [Curtis, 1980]. Curtis discusses that in a less-developed science, relationships between theoretical and operationally defined constructs are not necessarily established on a formal mathematical basis, but are logically presumed to exist. He writes:

“The more rigorous our measurement techniques, the more thoroughly a theoretical model can be tested and calibrated. Thus progress in a scientific basis for software engineering depends on improved measurement of the fundamental constructs.”

In his work, Curtis refers to Jones [Jones, 1978], who is also one of the pioneers in the area of software measurement.

In 1981, [Ruston, 1981] proposed a measure which describes a program flowchart by means of a polynomial. The measure takes into account both the elements of the flowchart and its structure. Ruston's method appears to be suitable for network measurement, but has not been used as widely as McCabe's method. Furthermore, Harrison presented software complexity measures which are based on the decomposition of flowgraphs into ranges [Harrison and Magel, 1981]. Using the concept of Harrison it is possible to determine the nesting level of nodes in structured and especially unstructured flowgraphs [Zuse]. Troy [Troy and Zweben, 1981] proposed a set of 24 measures to analyze the modularity, the size, the complexity, the cohesion and the coupling of a software system. Especially cohesion and coupling are fundamental criteria for the understandability of a software system. The basic division of software (complexity) measures into inter-modular and intra-modular components and the specific conceptual measures of coupling and cohesion are based on a work of [Constantine, 1968].

In 1982, [Piwowarski] suggested a modification of the Measures of Harrison, since he found these measures have some disadvantages (e.g., unstructured flowgraph can be less complex than structured flowgraphs).

Other measures for cohesion were proposed by [Emerson, 1984]. In 1982 Weiser presented the concept of *slices* [Weiser, 1982; Weiser, 1984], which consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion. Based on the concept of slices, measures for cohesion were discussed by [Longworth et al., 1986] and by [Ott and Thuss, 1991].

In the eighties several investigations in the area of software measures were done by the Rome Air Development Center [RADC, 1984]. In this research institute the relationships of software measures and software quality attributes (usability, testability, maintainability, etc.) were investigated. The goal of these investigations was the development of a *Software Quality Framework* which quantifies both user and management-oriented techniques for quantifying software product quality.

The NASA and the SEI (Software Engineering Institute) also started very early with software measurement. NASA is one of the few institutions which has been using software measurement since more than 15 years [Nasa]. Closely connected with NASA is the work of [Basili and Zelkowitz, 1977; Basili and Reiter, 1979; Basili and Turner, 1975].

2.5.2 Classification of Software Metrics

Before continuing with the historic view of measures, metrics and measurement, it is important to classify them. Software metrics may be broadly classified as either *product metrics* or *process metrics*. Product metrics are measures of the software product at any stage of its development, from requirements to installed system. Product metrics may measure the complexity of the software design, the size of the final program (either source or object code), or the number of pages of documentation produced [Xie et al., 2000].

Process metrics, on the other hand, are measures of the software development process, such as overall development time, type of methodology used, or the average level of experience of the programming staff [Xie et al., 2000].

In some other opinions [Archer and Stinson, 1995], there are three categories of software metrics to be concerned, which include *resource metrics* besides *product metrics* and *process metrics*. Improving the previous intents of product and process metrics, resource metrics are aimed to measure the inputs to the software engineering activity, such as hardware, software, documentation, knowledge, and human resources.

Software process metrics have been increasingly focused on recently. It has been argued that the quality of software products depends heavily on the quality of the process used to design, develop, deploy and maintain them. Many practices in evaluation of process quality and maturity have been carried out. Among them, two significant ones are the Quality Improvement Paradigm (QIP) of University of Maryland [Basili and Rombach, 1988] and the Capability Maturity Model (CMM) [SEI, 1995].

The QIP is a framework for guiding and supporting the improvement of software process and product. Goal-Question-Metrics (GQM) is one of the main components of

QIP and it is a method to guide the definition and exploitation of a goal-driven measurement program.

In addition to the distinction between product and process metrics, software metrics can be classified in other ways [Xie et al., 2000]. One may distinguish *objective* from *subjective* properties (metrics). Generally speaking, objective metrics should always result in identical values for a given metric, as measured by two or more qualified observers. For subjective metrics, even qualified observers may measure different values for a given metric, since their subjective judgment is involved in arriving at the measured value. For product metrics, the number of classes in a UML class diagram is an objective measure, for which any informed observer should obtain the same measured value for a given model. An example of a subjective product metric is the classification of the software as “organic,” “semidetached,” or “embedded,” as required in the COCOMO cost estimation model [Boehm, 1981]. Although most programs might be easy to classify, those on the borderline between categories might reasonably be classified in different ways by different knowledgeable observers. For process metrics, development time is an example of an objective measure, and level of programmer experience is likely to be a subjective measure.

Another way in which metrics can be categorized is as *primitive* metrics or *computed* metrics [Grady and Caswell, 1987]. Primitive metrics are those that can be directly observed, such as the program size (in LOC), number of defects observed in unit testing, or total development time for the project. Computed metrics are those that cannot be directly observed but are computed in some manner from other metrics. Examples of computed metrics are those commonly used for productivity, such as LOC produced per person-month (LOC/person-month), or for product quality, such as the number of defects per thousand lines of code (defects/KLOC). Computed metrics are combinations of other metric values and thus are often more valuable in understanding or evaluating the software process than are simple metrics.

Software metrics can also be classified according to the extent that it can be supported by metrics tools. Some activities can be supported automatically by CASE tools, but others can be done only by people manually or semi-manually. Therefore software metrics also can be assorted to *automatically* and *manually* implemented metrics.

Additionally, according to the software development life cycle, software metrics can also be classified as *requirement* metrics, *design* metrics, *code* metrics and *estimation* metrics. Pressman [Pressman, 2000] classifies the metrics in a different way. He divides the metrics as:

- Metrics for the analysis model
    - Function-based metrics;
    - The *Bang Metric*;
    - Metrics for specification quality;
- Metrics for the design model
    - High-level design metrics;
    - Component-level design metrics;
    - Interface design metrics;
- Metrics for source code
- Metrics for testing
- Metrics for Maintenance

According to [Pressman, 2000], metrics for the analysis model focus on function, data, and behavior of the analysis model. The function point [Albrecht, 1979] and the Bang Metric [DeMarco, 1982] provide quantitative means for evaluating the analysis model.

Metrics for design consider high-level, component-level and interface design issues. High-level design metrics consider the architectural and structural aspects of the design model. Component-level[4] design metrics provide an indication of module quality by establishing indirect measures for cohesion, coupling and complexity. Interface design metrics provide an indication of layout appropriateness for a graphical user interface (GUI). This document is particularly concerned with these kinds of metrics.

Software science provides an intriguing set of metrics at the source code level. Using the number of operators and operands present in the code, a variety of metrics can be used to assess program quality. In the remaining sections of this document, code and design metrics are presented. It is imperative to make a clear distinction between these two sorts of metrics.

Finally, considering software testing and maintenance, few metrics have been proposed. However, many other metrics can be used to guide the tests and as a mechanism for assessing the maintainability of a computer program.

This research is mainly concerned by product metrics extraction, objective, primitive and computed, automatic and manually implemented and focused on design. To avoid confusion between design and code metrics, an expanded explanation is given below, following the ideas of [Pressman, 2000].

Metrics for the design model

It is inconceivable that the design of a new aircraft, a new computer chip or a new office building would be conducted without defining design measures, determining metrics for various aspects of design quality, and using them to guide the manner in which the design evolves.

Several design metrics for software are available, but the vast majority of software engineers continue to be unaware of their existence.

High-level design metrics

High-level design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules. These metrics are black-box in the sense that they do not require any knowledge of the inner working of a particular module within the system.

One example of metric in this category was proposed in [Kafura and Henry, 1981], and it makes use of the *fan-in*[5] and *fan-out*[6]. The authors define a complexity metric of the form:

$$HKM = length(i) \times [f_{in}(i) + f_{out}(i)]^2$$

Where $length(i)$ is the number of programming language statements in module i, $f_{\text{in}}(i)$ is the fan-in of module i and $f_{\text{out}}(i)$ is the fan-out of module i. The authors extended the

[4] A component in this case refers to a module.
[5] *Fan-in* indicates how many modules directly control (invoke) a given module.
[6] *Fan-out* indicates the number of modules immediately subordinated to module *i*, that is, the number of modules that are directly controlled by (invoked by) module *i*.

definition of fan-in and fan-out to include not only the number of module control connections (module calls) but also the number of data structures from which module *i* retrieves (fan-in) or updates (fan-out) data.

In the HKM case, the design is used to estimate the number of programming language statements for module *i*. An increase in the HKM metric leads to a greater likelihood that integration and testing effort will also increase for a module.

Component-level design metrics

Component-level design metrics focus on internal characteristics of software components and include measures of module cohesion, coupling and complexity. These can help a software engineer to judge the quality of a component-level design.

The metrics presented in this section are white-box in the sense that they require knowledge of the inner working of the module under consideration.

One widely used metric in this category is the *cyclomatic complexity*, from Thomas McCabe [McCabe, 1976]. It provides a quantitative measure for testing difficulty and an indication of ultimate reliability. Experimental studies indicate a strong correlation between the McCabe metric and the number of errors existing in the source code, as well as the time required to find and correct such errors [Pressman, 2000]. Cyclomatic complexity also provides a quantitative indication of maximum module size.

Interface design metrics

Although there is a significant literature on the design of human-computer interfaces, relatively little information has been published on metrics that would provide insight into the quality and usability of the interface [Pressman, 2000].

[Sears, 1993] suggests *layout appropriateness* as a worthwhile design metric for human-computer interfaces. A typical GUI uses *layout entities* – graphic icons, text, menus, windows, and the like – to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which is used, and the "cost" of the transition from one layout entity to the next will all contribute to the appropriateness of the interface.

It is important to note that metrics of this caste can guide the construction of interfaces, but the final arbiter should be the user feedback, based on GUI prototypes.

Metrics for source code

Code metrics may be derived after the design is complete and the code is generated. The most commonly used measures of source code were developed to estimate its length. Inside this category, a well known measure is the LOC, introduce above. Regarding this measure, it is relevant to notice that some lines of code are different from others and many schemes have been proposed for counting lines [Fenton and Pfleeger, 1997].

One significant metric is the one of Halstead [Halstead, 1977]. Although his work has had a lasting impact, Halstead's software science measures provide an example of confusing and inadequate measurement. His metrics are presented in the literature as a definitive collection, with no corresponding consensus on the meaning of attributes [Fenton and Pfleeger, 1997].

Four internal attributes of code are measured, on an absolute scale: the number of distinct operators μ_1, the number of distinct operands μ_2 and the total number of respective occurrences of these, N_1 and N_2. The formula

$$N = N_1 + N_2$$

is a proposed metric of the internal program attribute *length*. N is a reasonable measure of the length of the actual code (without comments). However, the Halstead approach becomes problematic when examining some of the remaining measures, like the effort E and the time T. These variables are predicted measures of attributes of the process of implementing the program (although it is not made clear at which stage after requirements capture the process is assumed to start). There is a further serious problem with E, since the proposed measurement scale leads to contradictions involving meaningful statements about effort (the attribute being measured). For getting details of the problems with Halstead metrics refer to Fenton [Fenton and Pfleeger, 1997].

2.5.3 Software Measures in an Object-Oriented Environment

Recently object-oriented technology is becoming increasingly popular in industrial software development environments. This technology offers support to provide software product with higher quality and lower maintenance costs. Since the traditional software metrics aims at the procedure-oriented software development, they can not fulfill the requirements of the object-oriented software, and sets of new software metrics adapted to the characteristics of the object technology are indispensable. Object-oriented metrics become an essential part of object technology, as well as good software engineering [Xie et al., 1999].

At the end of the eighties software measures for the object-oriented environment (OO-Measures) started to be proposed. OO-measures are the focus of this document, and more precise information about some important OO-measures will be given later on in this document.

A very early research, dating from 1988, can be found in [Rocacher, 1988]. In 1989, Morris suggested software measures for an object-oriented application [Morris, 1989]. In 1991 Bieman discussed software measures for software reuse in an object-oriented environment [Bieman, 1991]. In 1992 Lake [Lake and Cook, 1992] presented measures for C++ applications. In 1993, Chidamber and Kemerer evaluated different Smalltalk applications [Chidamber and Kemerer, 1993b] while Li and Henry evaluated ADA-Programs [Li and Henry, 1993] and Chen and Lu evaluated OO-Measures [Chen and Lu, 1993] related to the Booch method [Booch, 1994]. Sharble [Sharble and Cohen, 1993] discussed measures for an object-oriented design and [Karner, 1993] wrote a master thesis of measurement in an object-oriented environment.

Other important papers of early nineties are [Laranjeira, 1990], [Caldiera and Basili, 1991], [Jensen and Bartley, 1991], [Rains, 1991], [Tegarden et al., 1992], [Abreu, 1993].

An interesting study of object-oriented measures is [Cook and Daniels, 1994]. The authors used factor analysis in order to figure out major factors in object-oriented programs.

Last but not least, the first book about object-oriented software metrics appeared in 1994 and it was written by Lorenz [Lorenz and Kidd, 1994].

After 1994, the research on the OO-measures field has continuously grown up achieving considerable results until today.

3

Object-Oriented Formal Specifications

SYNTHESIS

In the last years, several ways to formalize the design of object-oriented software have been proposed. Despite of the growing evidence that formal specification methods offer cost and quality benefits [Gerhart et al., 1994; Hall, 1996], effective strategies for their deployment continue to evade many organizations.

There are several reasons for the poor industrial take-up of formal specifications, including: (1) the needs of changes in process by skewing project resource requirements towards specification; (2) the inherent complexity of the formalisms, usually perceived to be hard to understand by non-experts; (3) the poor support they provide for identifying appropriate abstractions [Araújo and Sawyer, 1998].

In this chapter, efforts to break up these barriers are presented. We start discussing the needs of formal methods in Computer Science in general and later on in the object-oriented paradigm, to finally achieve the measurement field. We introduce some of the existing approaches and their limitations and we present the most recent and promising one, the Object Constraint Language (OCL), which is part of UML [OMG, 1997] standard.

OCL is the result of an effort to combine formalism soundness with usability and has its roots in the Syntropy method [Cook and Daniels, 1994]. Their creators wanted to produce a precise and unambiguous language that could be easily read and written by all practitioners of object technology and their customers.

We introduce the syntax and semantics of OCL in order to use it later, during the metrics formalization process. Some examples are presented, to make the language fully understandable.

"Whatever aptitude a man may have to exercise the power of abstraction, and to furnish himself with general ideas, he can make no considerable progress without the aid of language, spoken or written."

L. Euler (1707-1783)

3.1 INTRODUCTION: THE QUEST FOR FORMALIZATION

Many years of experience with the application of formal methods to software development indicate that the most beneficial effect of formality is, by far, a heightened degree of *precision* it introduces in specifications. Precision means clear and unambiguous statement of intent. While still imperfect, any conclusions drawn from precise specifications are more likely to be much closer to the ideal of certainty than those drawn from imprecise ones [Clark and Warmer, 2001].

Precision should not be confused with detail, although this often happens. In the case of specifications, precision implies a clear delineation between elements that are covered (included) by a specification from those that are not. Thus, a precise definition of the class of motor vehicles will allow to clearly conclude whether a particular object is a vehicle or not, while still leaving room for further detailing such as whether the vehicle is a truck, an automobile, or a moped. This means that there is no inherent conflict between precision and abstraction and that these two fundamental techniques used in design can complement and reinforce each other [Clark and Warmer, 2001].

Clearly, precise specifications are required when they are meant to be realized by a computer, since most computers do not tolerate ambiguity. They are equally necessary to accurately communicate ones' intent to other people. Consider the well-known example pointed out by [Parnas et al., 1987]:

"The level of the water in the tank shall never drop below X"

What does he mean by *water level*? Is it the instantaneous water level – which may be highly inaccurate due to the sloshing of water in the tank – or some average water level? If it is the latter, how is the average defined?

The problem is that the real word is a complex place and human rational thinking process is notoriously fallible. It is typically based on unstated assumptions, personal biases, and overextended mental shortcuts. Being precise forces people to tease out such fuzzy elements, expose them to closer scrutiny, and define the corresponding delineation boundaries.

Modern object-oriented modeling notations, such as the Unified Modeling Language (UML) [Fowler, 1997; OMG, 2001] are based on graphical notations for expressing a wide variety of concepts that are relevant for building models. While these notations are intuitive and easy to understand by users, they are not generally given a precise semantics as part of their initial definition, although a number of researchers have taken up this challenge [H. Bourdeau, 1995; J. Bicarregui, 1997; R. France, 1997]. In addition, the popular graphical notations can not express all the constraints desirable in some systems.

To remedy this, a number of authors have proposed mathematically-based textual languages, as an adjunct to the diagrams. Syntropy [Cook and Daniels, 1994] extends OMT [Rumbaugh et al., 1991] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with pre and post- conditions. Catalysis [D'Souza and Wills, 1998] does something very similar for UML. Recognizing this need, the Object Constraint Language (OCL) [OMG, 1997] was developed as a part of the UML standard, and is being used for precisely expressing constraints on a model.

Some argue that precision, either in semantics or textual annotations, is unnecessary. For software development this argument may be sustainable. Models are often discarded at the end of a development, because short-term economic weight puts pressure against them, being maintained and kept up to date only while the code is developed and tested. So, why spending a lot of time making these models precise if they are only going to be thrown away?

The software industry is now moving towards to the component-based development (CBD), and here the requirement for precision cannot be so lightly discarded [A. Hamie, 1998]. Accurate, expressive specifications are required to facilitate searching and matching of components and component assembly. Precision is essential for the automation of these processes.

In addition, the UML [OMG, 2001] is rapidly becoming a *de-facto* standard for modeling object-oriented systems. An important aspect of the language is the recognition by its authors of the need to provide a precise description of its semantics. This has resulted in a Semantics Document [Booch et al., 1997; OMG, 1997b] granting a meta-model description of the language, which forms an important part of the language's standard definition. The meta-model is presented in terms of three views: (1) the abstract syntax – expressed using a subset of UML static modeling notations; (2) well-formedness rules – expressed in the OCL, and (3) modeling element semantics, described in natural language. The UML semantics is explained in chapter 4.

A potential advantage of supplying a semantics model for UML is that many of the benefits of using a formal language such as Z [Spivey, 1992] might be transferable to UML. The major benefits of having a precise semantics for UML, according to [Kent, 1999], are:

- Clarity: the formally stated semantics can act as a point of reference to solve disagreements on interpretation and to clear up confusion over the precise meaning of a construct;
- Equivalence and consistency: a precise semantics provides an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components;
- Extendibility: the soundness of extensions to the UML can be verified;
- Refinement: the correctness of design steps in the UML can be verified and precisely documented. In particular, a properly developed semantics supports the development of design transformations, in which a more abstract model is diagrammatically transformed into an implementation model;
- Proof: justified proofs and rigorous analysis of important properties of a system described in the UML require precise semantics. Proof and rigorous analysis are not currently supported by UML;
- Tools: the tools that make use of semantics, for example a code generator or consistency checker, require semantics to be precise, whether it expressed as part of the standard or embedded in the code by the tool developer.

In fact, it is possible to generalize these benefits, applying them to any formal notation. The purpose of this chapter is to introduce the state-of-art of formality in the Software Engineering and the Object-Oriented areas.

3.2 FORMALIZATION IN OBJECT-ORIENTED SPECIFICATIONS

Since accuracy and certainty in specification have been, for many years, the aims of the branch of computer science known as *Formal Methods*, attempts have been made to combine them with object-oriented modeling. These attempts have followed four different roads [Abreu, 2000].

One road was that of extending and adapting an existing formal language with object-oriented constructs like in Object-Z [Duke et al., 1991] and VDM++, an extension of VDM [Jones, 1990]. This approach is not in line with industrial practice trends to use the simple, but powerful, graphical notations in object-oriented analysis and design. In fact, most practitioners are not at ease in using traditional formal specification languages, since they usually require a strong mathematical background.

A second road was that of complementing diagrammatic notations with some existing formal language constructs, like for instance in the case of Syntropy, mentioned above, where a subset of Z was combined with OMT. Also in this path are the ROOA [Moreira and Clark, 1996] and Metamorphosis [Araújo and Sawyer, 1998] approaches. These are respectable solutions, joining the benefits of graphical modeling with those of a formal language but still two drawbacks can be identified. First, there is a conceptual gap between the two formalisms. Second, the already mentioned difficulty of using a "traditional" formal language does not fade away. Consequently, modeling practitioners practice continued to be, during the 90's, a combination of graphical modeling with natural language descriptions to fill-in-the-blanks.

A third road was that proposed in the BON (Business Object Notation) object-oriented method [Waldén and Nerson, 1995]. There, a constraint language is used to express design by contract modeling issues, as advocated by Bertrand Meyer [Meyer, 1995]. At the time of its publication BON was, among the popular analysis and design methods, perhaps the only one to use a full-fledged assertion mechanism, allowing analysts to specify both the structure of a system and its semantics (constraints, invariants, properties of the expected results) [Meyer, 1997]. Besides graphical and tabular notations, BON uses a textual one to express assertions. This notation includes some constructs as "*delta a*" to specify that a feature can change an attribute "*a*", "*forall*" and "*exists*" to express logic formulae of first-order predicate calculus, and set operators such as "*member_of*". This notation bridges somehow the semantic gap problem previously mentioned, but still has a stumbling block – no widespread acceptance. Perchance, that was due to the fact that BON is somehow tied to the Eiffel language world. Besides, that acceptance often comes from standardization and shortly after BON was proposed, the joint initiative that would give birth to UML was already full spread ahead.

The last and more promising road to solve the problem in hand is the OCL, which will be discussed later on in this chapter.

3.3 ILL-DEFINITION OF OBJECT-ORIENTED METRICS

The lack of formalization has been felt for a long time in the object-oriented modeling area [Baroni and Abreu, 2002; Baroni et al., 2002a; Baroni et al., 2002b]. For instance, in the first well-known book [Lorenz and Kidd, 1994] on the subject of metrics for the object-oriented paradigm most proposed metrics were defined in natural language.

As an improvement, some authors have used a combination of set theory and simple algebra to express their metrics [Abreu and Carapuça, 1994; Chidamber and Kemerer, 1993a; Henderson-Sellers, 1996], but the mathematical background may not be easy to grasp. Some examples are presented by Baroni et al. [Baroni and Abreu, 2002; Baroni et al., 2002a].

Consider the metrics *Number of Times a Class is Reused* [Lorenz and Kidd, 1994] and *Count of Synchronization-based Coupled Object Types* (CSCO) [Poels and Dedene, 2001]. The former is defined as the number of references to a class. However it is not clear what references are and how the metric should be computed. Should internal and external references be counted? Should references be considered in different modules, packages or subsystem? Does the inheritance relationship count as a reference?

Poels defines

$$CSCO(P) = \#\{Q \in T - \{P\} \mid \exists e \in A: (\tau_1(e, P) = C \wedge \tau_1(e, Q) = E) \vee (\tau_1(e, P) = E \wedge \tau_1(e, Q) = C)\}.$$

Finding out the meaning of this formula, even knowing each of the components involved, is probably not an easy nut to crack, for most software designers.

Moreover, the measure of distance δ_M, is defined by Poels [Poels and Dedene, 1996] as the average distance between the object types of two different non-empty dynamic conceptual schemes. The notion behind this measure may be defined and interpreted in many ways, according to distinct viewpoints. What is a distance? Which are the conditions for measurement? Is the distance expressed by some degree of dissimilarity?

To avoid the ambiguity generated by the informal definition, Poels presents the mathematical development of the measure as follows:

$$\delta_M(M_P, M_Q) = 0 \Leftrightarrow M_P \Delta M_Q = \phi$$

$$\delta_M(M_P, M_Q) = \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}\delta\,(P_i, Q_j)}{I.J} \Leftrightarrow M_P \Delta M_Q \neq \phi$$

$$\delta_M(M_P, M_Q) = \begin{bmatrix} \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}\delta_{alph}(P_i, Q_j)}{I.J}, \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}\delta_{atr}(P_i, Q_j)}{I.J} \\ \frac{\sum_{i=1}^{I'}\sum_{j=1}^{J'}\delta_{seq}(P_i, Q_j)}{I.J}, \frac{\sum_{i=1}^{I}\sum_{j=1}^{J}\delta_{data}(P_i, Q_j)}{I.J} \end{bmatrix} \Leftrightarrow M_P \Delta M_Q \neq \phi$$

where:

M_P and M_Q are non-empty dynamic conceptual schemes;

$M_P \,\Delta\, M_Q = \varnothing \Leftrightarrow (\forall P \in M_P, \exists Q \in M_Q : \delta(P, Q) = 0) \wedge (\forall Q \in M_Q, \exists P \in M_P : \delta(Q, P) = 0)$;

cardinality$(M_P) = I$; cardinality$(M_Q) = J$;

$P_i \in M_P \quad i = 1, \ldots, I$; $P_j \in M_Q \quad j = 1, \ldots, J$.

The above definition can be used for both scalar and vector representations of the measure $\delta\,(P_i, Q_j)$ – second and third definitions respectively. Once again, inferring the meaning of this formula may be an arduous task. In other words, these examples introduce some problems, difficult to solve.

It is clear that problems can arise from the formality degree used to define metrics, namely the informal (or natural language) definition problem and mathematical formal definition problem, which leads to an ill-definition of software metrics. The former can generate diverge results, as people using metrics can interpret them in several ways. The latter requires a strong mathematical background to cope with the expressions complexity, which most of software practitioners may not have.

Without clear and precise definitions it is difficult to build adequate metric extraction tools, experiments replication is hampered, and results interpretation may be flawed.

The ill-definition problem may happen because:

i) metrics definitions are usually presented without the corresponding context, that is, without expressing which is the corresponding meta-model where the entities of interest and their interrelationships are expressed;
ii) metrics definition is done without an underlying formal specification approach that uses the former meta-model as contextual input; this formal specification should specify, among other things, under which conditions the metrics are applicable.

In this work, an approach for defining design metrics that combines understandability and formality while solving the ill-definition problem is proposed. This approach is verified and validated for sake of correction and for guaranteeing the quality of the formalizations. UML and OCL are used to build that meta-model and to express the metrics as meta-model operations. The metrics applicability limitations are defined with OCL pre-conditions.

Before presenting the approach itself, the OCL is introduced.

3.4 THE OBJECT CONSTRAINT LANGUAGE (OCL)

OCL is the most recent and promising approach to support precision and solve the previously mentioned problems; that is, it is a tool to help expressing the ideas precisely while it bridges formal methods with object-orientation. It is a formal, yet simple notation, to be used jointly with UML diagrams and whose syntax has some similarities to those of object-oriented languages such as Smalltalk, C++ or Eiffel. It is underpinned by mathematical set theory and logic, like in formal languages, but was designed for usability and is easily grasped by anybody familiar with object-oriented modeling concepts in general, and UML notation in particular.

What makes OCL unique and gives it tremendous leverage is that it is adapted to a UML context and is part of the UML standard. This creates an opportunity to introduce the benefits of precise specification to a much broader community of software developers than most other formal notations [Clark and Warmer, 2001].

In the past, one of the major impediments of many notations has been that they were not an integral part of a common development language or tool. From that perspective it is instructive to note the effect obtained when a syntactically and semantically integrated assertion construct was introduced into the C language. The net result was that a large number of software developers who had never heard of the Hoare triple used the mechanism to improve the reliability of their software [Hoare, 1973].

The term Hoare triple comes from the field of axiomatic semantics of programs. It has three parts namely a precondition *P*, a program statement or series of statements *S*, and a post-condition *Q*. It's usually written in the form

$$\{P\}\ S\ \{Q\}$$

The meaning is "if *P* is true before *S* is executed, and if the execution of *S* terminates, then *Q* is true afterwards". The triple does not assert that *S* will terminate; that requires a separate proof.

OCL provides a “programmer friendly” version of prepositional logic (that seems to repel many software practitioners). Thus, the existential and universal quantifiers (truly scary and highfalutin names) are cleverly disguised as operation names (*exists* and *forall* respectively), to hide from users that they are actually applying mathematical logic.

Due to these advantages, OCL was chosen as the formalization language for the work presented in this document and it is described now.

3.4.1 OCL Expressions

OCL allows expressing three kinds of constraints, namely *invariants*, *pre-conditions* and *post-conditions*. According to [OMG, 1999], a constraint is a semantic condition or restriction expressed in text. It is associated with a ModelElement and must be true for the model to be well formed. It indicates a restriction that must be enforced by correct design of a system. In UML, constraints are expressed as the standard stereotypes «invariant», «precondition» and «postcondition».

Invariants are constraints that represent conditions that must be met by all instances of the class, over time. Their context is therefore a class, hereafter represented using underlined fonts (in the first line), as in:

```
Customer
self.age > 18
```

The dot notation is used for attribute access. In the above example a boolean operation (comparison) is applied to the attribute *self*, which is a special implicit attribute that allows to reference the context object (the class instance).

The dot notation is also used to navigate in one class diagram through associations. This will be done later in this work, to formalize metric sets with OCL and the class diagram of the UML meta-model. If the role name of an association is identified in the UML diagram, then it is used in the navigation. Otherwise, the name of the target class is used, in lowercase letters.

Pre and post-conditions are assertions whose scope is an operation. Pre-conditions denote that the conditions of the constraint must hold for the invocation of the operation (they are constraints that must be true for an operation to be executed). They traduce the rights of the object that offers the service or the client responsibilities.

Post-conditions are constraints that must be true when the operation ends its execution (the conditions of the constraint must hold after the invocation of the operation.). They traduce the obligations to be fulfilled by the object that offers the service or the client rights.

The context of both pre and post-conditions is, therefore, an operation, as in the following extract from the *Sequence* type definition:

```
Sequence::prepend(object: T): Sequence(T)
post: result->size() = self@pre->size() +1
post: result->at(1) = object
```

Operations can have input parameters and must have a return type. In the *prepend* example, an object of type *T* is given as parameter and a sequence (also of *T* typed objects) is returned. The "**::**" sign is a scope indicator (In this case it shows that *prepend* is defined in the scope of the class *Sequence*). The "→" sign is used for applying an operation to a collection. The *result* keyword represents the object returned by the operation, whose type is identified in the operation signature (a generic type *T*, in this case). The *@pre* suffix allows using the value of the characteristic to which it is applied at the moment when the operation is called, that is, its original value before the operation is applied. Several pre and post conditions can be defined within the same operation.

OCL is a declarative typed language whose expressions are free of side effects. This means that the state of the objects does not change by the application of an OCL expression. These expressions can range from simple comparisons (e.g. an attribute having an upper limit) to complex navigations in a class diagram through their associations.

Since OCL is a typed language, it is possible to check expressions for validity during modeling[7]. Notwithstanding, OCL does not specify what happens when a constraint is broken. This problem is deferred to the implementation since the constraint and exception handling mechanisms are supported differently by available programming languages.

OCL convey a number of benefits, offering precision and better design documentation. The result is an unambiguous communication among the parts involved, such as designers, users, programmers, testers and managers.

3.4.2 OCL Types

All objects in OCL have a type, derived from *OclAny*, which determines the applicable operations. There are sets for predefined types including basic types (*Boolean*, *Integer*, *Real*, and *String*), enumeration types, and collection types (*Collection*, *Set*, *Bag* and *Sequence*). Figure 3.1 summarizes the OCL type's hierarchy.

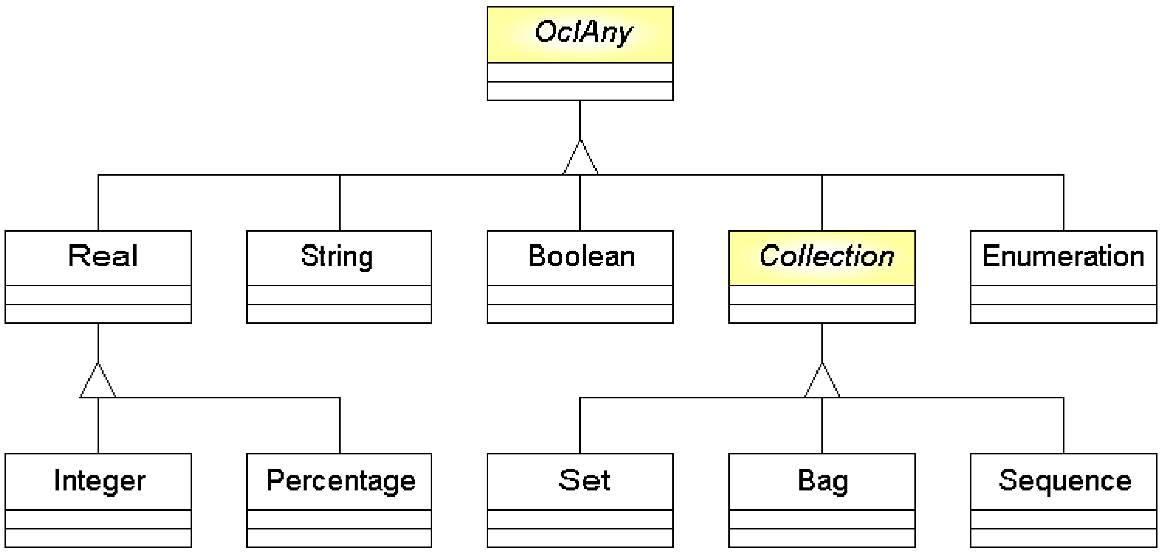

Figure 3.1 – *OCL Types*

The basic types have a number of operations defined on them, as represented in table 3.1.

[7] For this purpose a free OCL parser can be found in http://www-3.ibm.com/software/ad/library/standards/ocl.html

Type	Operations
Boolean	=, not, and, or, xor, implies, if-then-else
Real	=, +, -, *, /, abs, floor, max, min, <, >, <=, >=
Integer	=, +, -, *, /, abs, div, mod, max, min
String	=, size, toLower, toUpper, concat, substring

Table 3.1 – *Operations Defined over OCL Basic Types*

Enumeration types can be defined in a model by using:

```
enum{ value1, value2, value3 }
```

The values of the enumeration can be used within expressions. As there might be a name conflict with attribute names being equal to enumeration values, the usage of an enumeration value is expressed syntactically with an additional sharp (#) symbol prefixing the name of the value:

```
#value1
```

The type of an enumeration attribute is Enumeration, with restrictions on the values for the attribute.

Considering the collection types, *Sets* do not allow duplicates and their elements are not ordered; *Bags* allow duplicates but their elements are also not ordered; *Sequences* have an order imposed on their elements and allow duplicates.

OCL expressions are often constructed in association with a given UML diagram. For instance, the result of navigating through just one association (in a class diagram) is a *Set*[8], and through more than one association with multiplicity *many* is a *Bag*. The *Collection* class is an abstract class from which the previous three are derived. This can be expressed in OCL in the following manner:

```
Collection
Collection.allInstances->select(oclType = Collection)->isEmpty()
-- the allInstances operation returns the set of all objects of the named class and of all its
-- subclasses;
```

OCL types are open to specialization. For instance, in the Catalysis approach [D'Souza and Wills, 1998], the *Set* and *Sequence* type operations are extended. For the purposes of this document, a new type is derived from the OCL Real. The *Percentage* type is a constrained Real whose instances can only have values in the interval [0, 1]. Since Percentage is a value type [Warmer and Kleppe, 1999], its instances are values. Therefore, it is possible to write the following invariant:

```
Percentage
(self >= 0) and (self <=1)
-- 0 is 0% and 1 is 100%
```

[8] Unless the association is adorned with the {ordered} tag, in which case the result is a *Sequence*.

In expressions, it is feasible to use both operations defined over OCL types and those belonging to the UML class diagram upon which the constraints are written. However, since OCL is side-effect free, only selectors[9] are allowed.

The most frequently used operations when navigating on the class diagrams are those that manipulate collections. Figure 3.2 details the ones of OCL collection types.

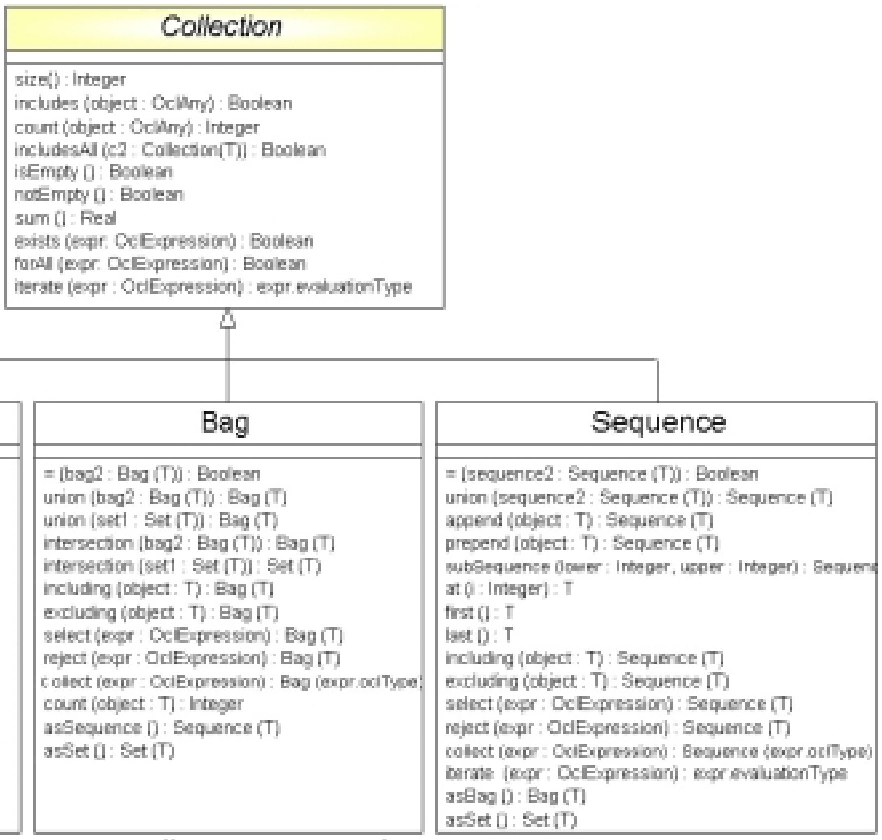
Collection
size() : Integer
includes (object : OclAny) : Boolean
count (object : OclAny) : Integer
includesAll (c2 : Collection(T)) : Boolean
isEmpty () : Boolean
notEmpty () : Boolean
sum () : Real
exists (expr: OclExpression) : Boolean
forAll (expr: OclExpression) : Boolean
iterate (expr : OclExpression) : expr.evaluationType
Set
union (set2 : Set (T)) : Set (T)
union (bag1 : Bag (T)) : Bag (T)
= (set2 : Set (T)) : Boolean
intersection (set2 : Set (T)) : Set (T)
intersection (bag1 : Bag (T)) : Bag (T)
- (set2 : Set (T)) : Set (T)
including (object : T) : Set (T)
excluding (object : T) : Set (T)
symmetricDifference (set2 : Set (T)) : Set (T)
select (expr : OclExpression) : Set (T)
reject (expr : OclExpression) : Set (T)
collect (expr : OclExpression) : Set (expr.Type)
count (object : T) : Integer
asSequence () : Sequence (T)
asBag () : Bag (T)
Bag
= (bag2 : Bag (T)) : Boolean
union (bag2 : Bag (T)) : Bag (T)
union (set1 : Set (T)) : Bag (T)
intersection (bag2 : Bag (T)) : Bag (T)
intersection (set1 : Set (T)) : Set (T)
including (object : T) : Bag (T)
excluding (object : T) : Bag (T)
select (expr : OclExpression) : Bag (T)
reject (expr : OclExpression) : Bag (T)
collect (expr : OclExpression) : Bag (expr.oclType)
count (object : T) : Integer
asSequence () : Sequence (T)
asSet () : Set (T)
Sequence
= (sequence2 : Sequence (T)) : Boolean
union (sequence2 : Sequence (T)) : Sequence (T)
append (object : T) : Sequence (T)
prepend (object : T) : Sequence (T)
subSequence (lower : Integer, upper : Integer) : Sequenc
at (i : Integer) : T
first () : T
last () : T
including (object : T) : Sequence (T)
excluding (object : T) : Sequence (T)
select (expr : OclExpression) : Sequence (T)
reject (expr : OclExpression) : Sequence (T)
collect (expr : OclExpression) : Sequence (expr.oclType)
iterate (expr : OclExpression) : expr.evaluationType
asBag () : Bag (T)
asSet () : Set (T)

Figure 3.2 – *OCL Collection, Set, Bag and Sequence Operations*

Collections of collections are conceptually difficult and are seldom used in practice. In general, it is not desirable that a collection of elements contains another collection, but contains only simple elements. The operation *flatten*[10] converts the set of collections into a set of elements, as showed below:

```
Set { Set { 1, 2 }, Set { 3, 4 }, Set { 5, 6} }
```

results in
`Set { 1, 2, 3, 4, 5, 6 }` after applying the *flatten* operation.

[9] Query operations that return a value but do not change the object state. In the UML meta-model their *isQuery* boolean attribute is true.
[10] *Flatten* is used several times in chapter 5.

3.4.3 The “Royal and Loyal” System Example

In this section a simplified example of a model that uses OCL is presented. Its original and complete version can be found in [Warmer and Kleppe, 1999].

Royal and Loyal (R&L) is a fictional company that handles loyalty programs for companies that offer their customers various kinds of bonuses. Anything a company is willing to offer can be a service rendered in a loyalty program (air miles, reduced rates, a larger car for the same price as a standard rental car, etc.). Figure 3.3 shows the UML class model for R&L.

The central class in the model is *LoyaltyProgram*. A system that administrates a single loyalty program will contain only one object of this class. A company that offers its customers a membership in a loyalty program is called *ProgramPartner*. More than one company can enter into the same program. In that case, customers who enter the loyalty program can profit from services rendered by any of the participating companies.

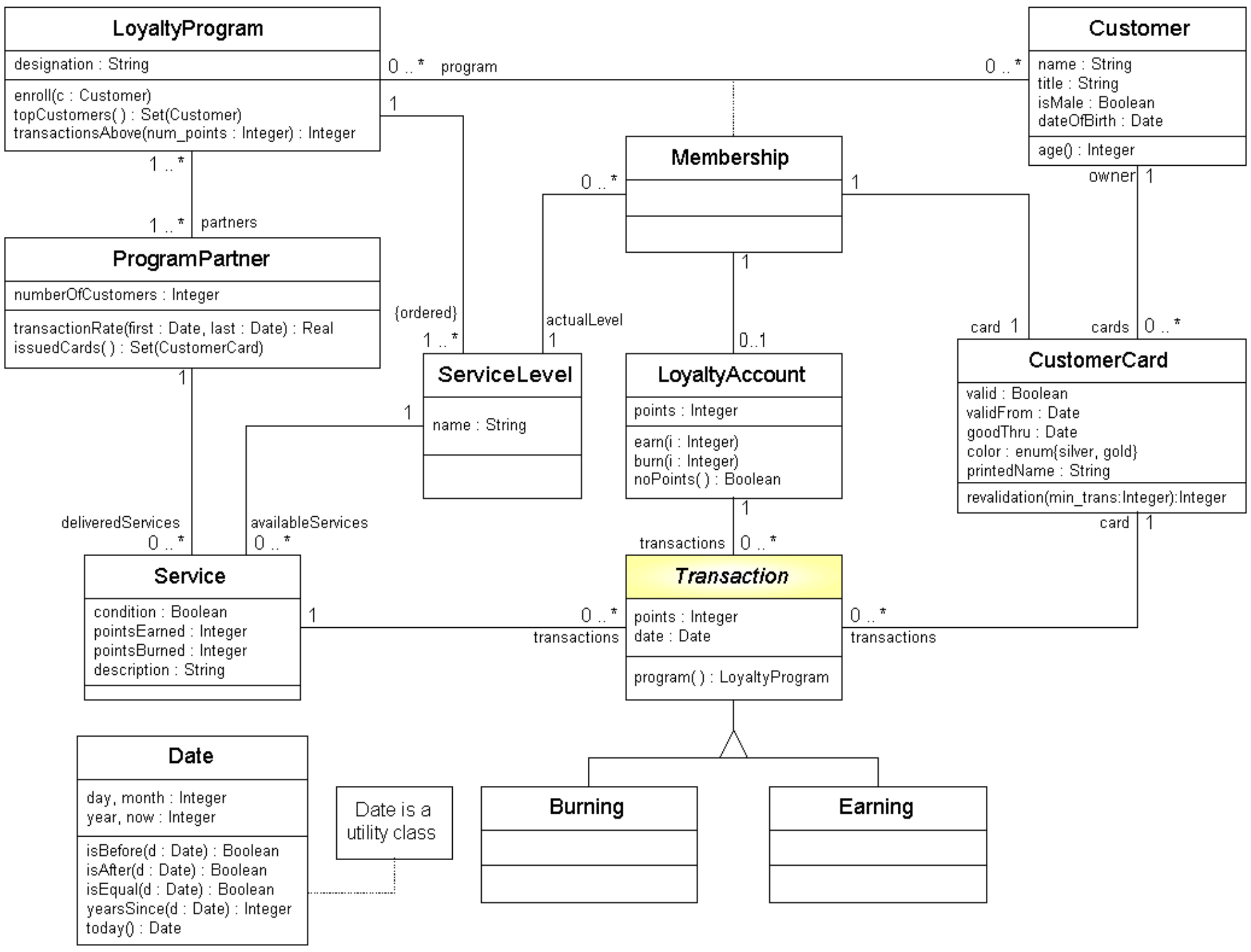

Figure 3.3 – *The Royal and Loyal Model*

Every customer of every program partner (represented by the class *Customer*) can enter the loyalty program by getting a membership card (represented by the class *CustomerCard*). Each card is issued to one person.

Most loyalty programs allow customer to save bonus points. Each individual program partner decides when and how bonus points are allotted for a certain purchase. Saved bonus points can be used to “buy” specific services form one of the program partners. To account for the bonus points that are saved by a customer, every membership can be associated with a *LoyaltyAccount*.

There are two types of transactions for each loyalty account. First, there are transactions in which the customer obtains bonus points. In the model, these transactions are represented by a subclass of *Transaction* called *Earning*. Second, there are transactions in which the customer spends bonus points. In the model, they are represented by instances of the *Burning* subclass of *Transaction*.

Customers who make extensive use of the membership are rewarded with a higher level of service (e.g., a gold card). To administer different levels of service, the class *ServiceLevel* is introduced in the model. A service level is defined by the loyalty program and used for each membership.

Each year, R&L sends a new card to all customers. When appropriate, R&L upgrades a membership card to a gold card, invalidating the old one. R&L can invalidate a membership when the customer has not used the card for a certain period.

Adding some invariants to the model

It is simple to add some invariants to a class. First, the class on which the invariant is placed is indicated. It is called the *context*[11] of the invariant. Then, a Boolean expression that states the invariant is built. All attributes of the context class may be used in this invariant.

In the R&L, a reasonable invariant for every customer card is that its data *validFrom* should be earlier than *goodThru*. In OCL this can be written as:

```
CustomerCard
validFrom.isbefore( goodThru )
```

Here the attribute *validFrom* is not of a standard type, such as *Boolean* or *Integer*, but an instance of the Date class. In this case, the operations defined for the class type can be used to write the invariant, and the operation name and parameters come after the attribute name, separated by a dot. The operation *isBefore* on the class *Date* checks whether the date in the parameter is earlier than the date object being tested, and results in a Boolean value.

It is also possible to put invariants on attributes of objects of associated classes, as in:

```
CustomerCard
printedName = customer.title.concat( customer.name )
```

This invariant means that the attribute *printedName* in every instance of *CutomerCard* must be equal to the concatenation of the *title* and *name* attributes of the associated instance of *Customer*. Notice the navigation from *CustomerCard* to *Customer*.

Another invariant on the R&L model is that the number of valid cards for every customer must be equal to the number of programs the customer participates in. This constraint can be avowed using the *select* operation on sets. The *select* takes an OCL expression as parameter and results in a subset (of the set on which it is applied)

[11] In this document the context is underlined, but this convention is not part of the UML-OCL standard.

containing all the elements from which the parameter is true. In the following example, the result of the *select* is a subset of *card*, where *card.valid* is true.

```
Customer
program -> size = cards -> select( valid = true ) -> size
```

Also relevant on the R&L model is that, when none of the services offered in a *LoyaltyProgram* credits or debits the *LoyaltyAccount* instances, these instances are useless and the account should be empty. The *forall* operation on the collections can be used to build the invariant. Like *select*, it takes an expression as parameter and its outcome is a boolean: true if the expression evaluates to true for all elements in the collection, and false otherwise.

```
LoyaltyProgram
partners.deliveredServices -> forall( pointsEarned = 0 and pointsBurned = 0 )
                                  implies membership.loyaltyAccount -> isEmpty
```

To understand the differences among collection types, consider the attribute *numberOfCustomers* of the class *ProgramPartner*. The invariant can testify that this attribute holds the number of customers who participate in one or more loyalty programs offered by this program partner. In OCL, this would be expressed as:

```
ProgramPartner
numberOfCustomers = loyaltyProgram.customer -> size
```

But there is a problem with this expression. A customer can participate in more than one loyalty program. In other words, an object of the class *Customer* could be repeated in the collection *loyaltyProgram.customer*. In the preceding expression, these customers are counted twice, and that is not what is intended.

The rule is that when navigating through more than one association with multiplicity greater than 1, the result is a bag. When navigating in an association whose multiplicity is one, the result is a set. So, to correct the previous statement, one of the operations show in figure 3.2 can be applied, as follows:

```
ProgramPartner
numberOfCustomers = loyaltyProgram.customer -> asSet( )-> size
```

In the R&L example, the program partners want to limit the number of bonus points they give away. They have set a maximum of 10,000 points to be burned for each partner. In this case, it is important to consider just the burning transactions, and this is done with the *oclIsTypeOf* operation. This way, the subclasses of *Transaction* can be considered. To retrieve from the collection all instances of the subclass *Burning*, the *select* operation is used. To obtain the set of values of burned points, the *collect* operation is employed. The elements in the collection are summed and compared with the given maximum.

```
LoyaltyProgram
partners.deliveredServices.transaction
            -> select(oclIsTypeOf( Burning ) )
                    -> collect( points ) -> sum( ) < 10,000
```

Sometimes an enumeration type is defined as an attribute type in a UML class model. The values an attribute of this type can hold are indicated in an OCL expression with a # symbol before the value name. An example can be found in the *CustomerCard* class, where the attribute *color* can have two values, either *silver* or *gold*, as shown in figure 3.3. The next invariant stress that the color of this card must match the service level of the membership.

```
Membership
actualLevel.name = ‘Silver’ implies card.color = #silver and
  actualLevel.name = ‘Gold’ implies card.color = #gold
```

Writing Pre and Post-Conditions

Pre and post-conditions specify the conditions to be met before and after the execution of one operation. They are the ways to write constraints for operations. To indicate the operation for which the condition must hold, the constraint context is extended with the name of the operation. This means that all attributes and links from the object in the context can be used, but the expressions following the context declaration must hold for the given operation only.

Two special keywords can be used to represent the working of time: *result* and *@pre*. The @ symbol followed by the *pre* keyword indicates the value of an attribute or association at the start of the execution of the operation, as in the following example.

```
LoyaltyProgram::enroll( c : Customer )
pre: not customer -> includes( c )
post: customer = customer@pre -> including ( c )
```

The pre-condition states that the customer to be enrolled is not already a member of the program. The post-condition states that the set of customers after the enroll operation is identical to the set of customers before the operation with the enrolled customer added to it. It is also possible to add a second post-condition saying that the membership for the new customer owns a loyalty account with zero points and no transactions.

```
post: membership -> select ( customer = c ) -> forall (
                                                 loyaltyAccount -> notEmpty( ) and
                                                 loyaltyAccount.points = 0 and
                                                 loyaltyAccount.transactions -> isEmpty )
```

In the R&L example, the class *LoyaltyAccount* has an operation *isEmpty*[12]. When the number of points on the account is zero, the operation returns the value true. To utter this more precisely, the operation returns the outcome of the Boolean expression points = 0. In the following constraint, the return value of the operation is indicated by the OCL keyword *result*.

```
LoyaltyAccount::isEmpty( )
pre: -- none
post: result = (points = 0)
```

[12] This operation is different from the *isEmpty* operation defined on sets.

As there is no precondition for this operation, the comment was included where the pre-condition could have been placed. For a given operation the definition of pre and post-conditions is not mandatory.

The keyword *result* indicates the return value from the operation. The type of *result* is defined by the return type of the operation. In the following example, the type of *result* is LoyaltyProgram.

```
Transaction::program( ):LoyaltyProgram
post: result = self.card.membership.program
```

In this example, the result of the program operation is the loyalty program against which the transaction was made. The *self.card* is the CustomerCard associated with the transaction, and *self.card.membership* is the membership to which this CustomerCard belongs. The *self.card.membership.program* is the loyalty program to which the membership belongs.

4

The UML Semantics Model

SYNTHESIS

This chapter specifies the semantics for some constructs used to create UML structural and behavioral object models. Structural models (also known as static models) emphasize the structure of objects in a system, including their classes, interfaces, attributes and relations. Behavioral models (also known as dynamic models) emphasize the behavior of objects in a system, including their methods, interactions, collaborations, and state histories.

The semantics for the modeling notations described in the UML Notation Guide [OMG, 2001], which includes support for a wide range of diagram techniques (class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state diagram, activity diagram, and deployment diagram), is provided through the UML Semantic Model (or UML Meta-Model). Despite of all diagram techniques covered in the UML Notation Guide, only the relevant parts for our purposes are explained. Those are the basic constructs from which metrics can be extracted at a design level.

A summary of the semantic sections that are relevant to each diagram technique can be found in [OMG, 1999]. For the complete semantics description, refer to [OMG, 2001].

"There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

4.1 INTRODUCTION: A LITTLE BIT OF STORY

Back in the decades of 70 and 80, there was a general disagreement between people that believed in functional modeling and those who believed in data modeling. At that time, the ideas of using flow diagrams or entity-relationship diagrams were generally viewed as being mutually exclusive. Towards the end of the '80s, a reconciliation took place between the two camps [Yourdon, 1989]. It was then realized that most projects could benefit from using both data models and functional models.

What followed was the birth of several object-oriented analysis and design methods. Unfortunately, each method had its own notation and used its own definitions of terms such as *objects*, *types* and *classes*. There was no consensus for an industrial standard. The number of identified modeling languages increased from less than 10 to more than 50 during the period between 1989 and 1994 [Rasmussen, 2000].

Some of the major players were Grady Booch, Jim Rumbaugh and Ivar Jacobson. They published books on their own object-oriented methodologies, *Booch* [Booch, 1994], *OMT* [Rumbaugh et al., 1991] and *OOSE* [Jacobson et al., 1992], respectively. CASE tool vendors had a particularly hard time. It was not clear which object-oriented method they should invest their efforts in.

In 1994, Grady Booch and Jim Rumbaugh announced their working union to create a *Unified Method* [Booch and Rumbaugh, 1995]. Later, Ivar Jacobson joined them. Finally, the group agreed on a common notation and defined the semantics for the basic object-oriented concepts. During 1996, the Unified Method project evolved into the *Unified Modeling Language* (UML) [OMG, 1997a]. The new name stressed that UML was a modeling language, and not a method. It concentrated on providing an expressive unified notation and defining strict semantics for the concepts involved. The unification efforts succeed in casting away elements of Booch, OMT and OOSE that had little practical value, and also, in bringing up new elements that were missing from these three methods.

UML was well received by the industry, and it slowly became a *de facto* standard in the modeling community. Most squabbles regarding the choices of modeling notations were settled when the Object Management Group (OMG) published the UML specification.

4.2 FUNDAMENTAL CONCEPTS

This section brings in some important concepts for understanding the *UML semantic model* (*meta-model*). Initially, a short concept of UML and its elements are presented. Afterwards, the 4 layers architecture for dealing with meta-models is introduced. All these concepts are used to understand the semantics of UML, expressed in section 4.3.

4.2.1 The Unified Modeling Language (UML)

According to [OMG, 1999] the Unified Modeling Language (UML) is a consistent language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling. System architects can use it to specify, visualize, construct and document designs.

The Unified Modeling Language includes:

- *model elements*, which capture the fundamental modeling concepts;
- a notation, for visual rendering of model elements;
- rules[13], which describe idioms of usage and present the semantics (see section 4.3) of model elements.

It also provides extensibility and specialization mechanisms to extend the core concepts (section 4.3.1.2). UML does not provide, define nor dictate:

- a programming language: UML has a semantics model that maps well to a family of object-oriented languages, but in itself does not require the use of a specific language;
- tools: UML neither specifies the design of CASE tools, nor specifies the use of them. However, it is natural to expect that CASE tools supporting UML follows the UML semantics closely;
- a development process: defining a standard process was not a goal of UML, and UML was intentionally created to be process independent.

4.2.2 UML Elements

All the fundamental modeling concepts in UML are described as *elements* in the *UML Specification* [OMG, 1999]. Everything from concrete language constructs such as a *class*, to abstract concepts such as a *use case,* is referred to as an *element*.

Consider you want to describe a class Car that has the relationships shown in Figure 4.1. The class Car is a specialization of the class Vehicle, and is associated with a class Wheel. For each Car object, four Wheel objects participate in the association.

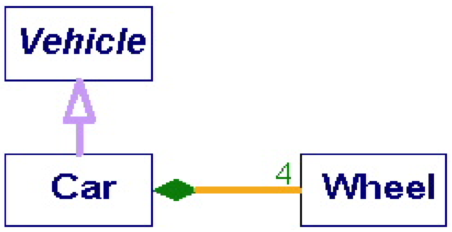

Figure 4.1 – *A Sample Design*

[13] *Rules* are also called *Constraints*.

Each class in figure 4.1 is considered an element in UML. The specialization/generalization relationship between Vehicle and Car, and the association between Car and Wheel are also considered elements. Even the association ends, where the association connects to the classes, are considered elements.

All of these elements are part of the *UML semantics model*, which gives precise meaning to concepts, such as objects and classes.

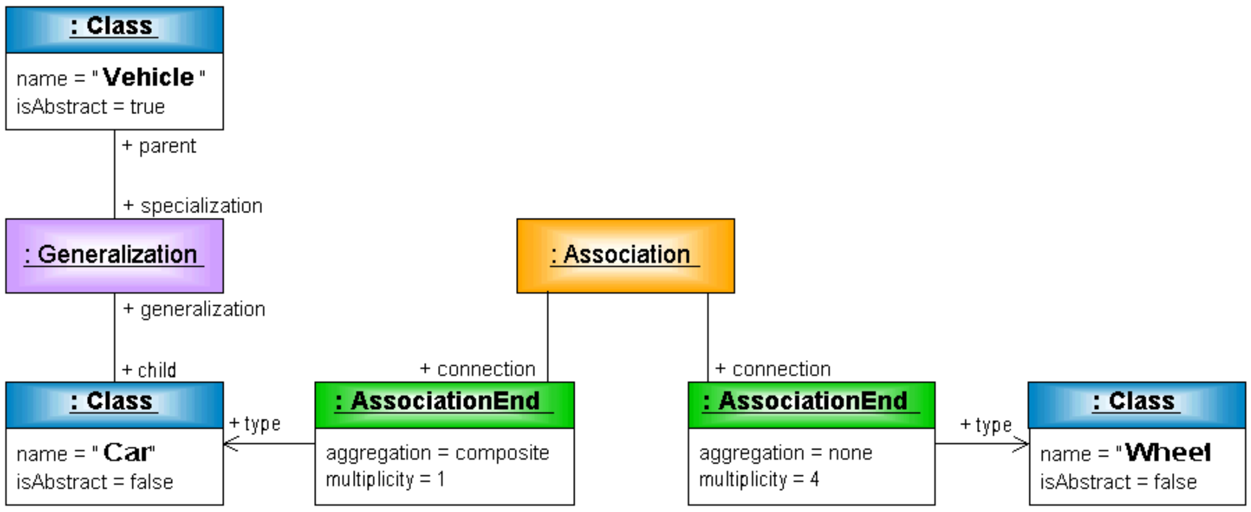

Figure 4.2 – *Sample Design Mirrored in the UML Semantics Model*

The exact semantics of each element type is described in great detail in the *UML Semantics* section of the OMG Unified Modeling Language Specification [OMG, 1999]. Figure 4.2 is an object model that depicts the elements from the UML semantic model that are used to describe the sample design of figure 4.1.

4.2.3 The UML Semantic Model (Meta-Model)

A software design model expressed in the Unified Modeling Language describes a software system. The UML semantics model is used to express the description of the modeling elements used on software design. It is therefore a model for describing models and is often referred to as the *UML meta-model*. The constituents of this model are the UML *elements* that can be used to describe designs such as the one in figure 4.1.

Each UML element is shown as an object in figure 4.2. There are several types of elements; one for each separate modeling concept. The elements describing one design have types for representing classes, generalizations, associations, association ends and so on.

Only certain arrangements of element objects have meaning. For instance, an *Association element* that has a connection to a *Generalization element* is meaningless. However, the Association object in figure 4.2 that has connections to AssociationEnd objects successfully denotes the semantics of the relationship between cars and wheels.

Element objects can have different attributes based on their type. As an example, the *AssociationEnd elements* have an attribute for denoting the multiplicity of a classifier in an association.

The UML semantic model allows representing each UML element as an object. It describes what attributes an element object can have, and which relationships can exist between element objects.

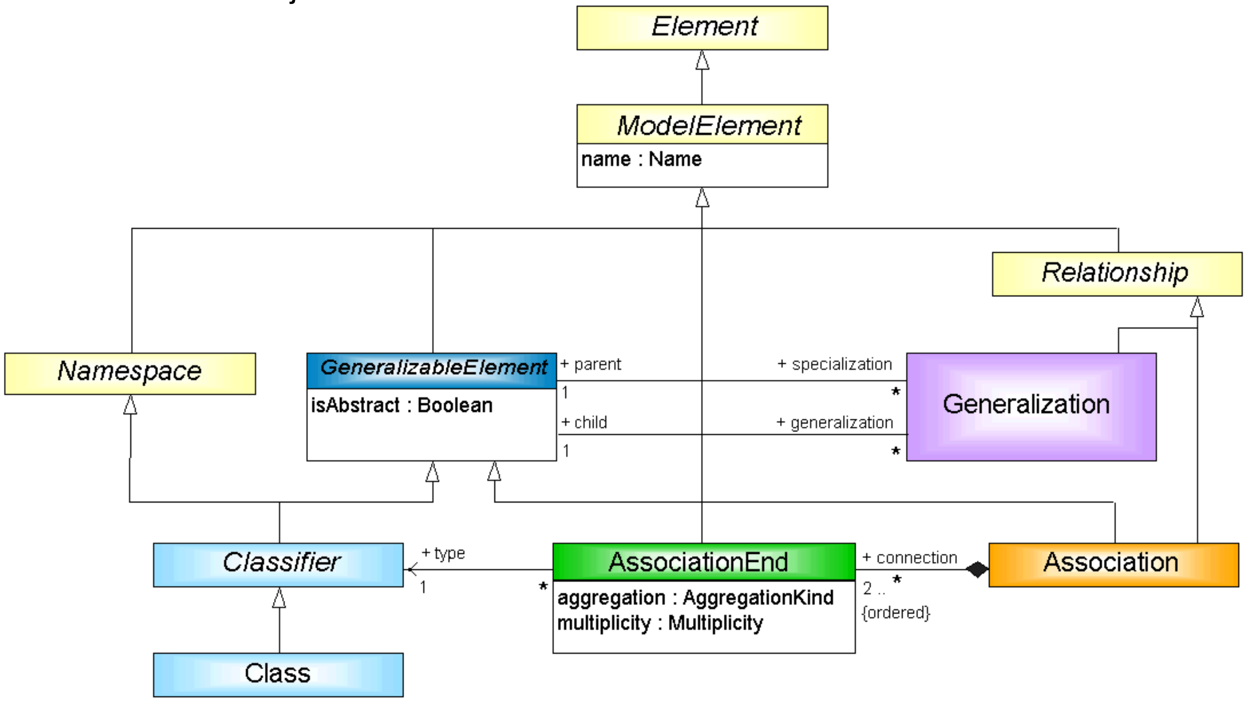

Figure 4.3 – *A Fragment of the UML Semantic Model*

Figure 4.3 shows the part of the UML meta-model corresponding to the objects in figure 4.2. Here Element, ModelElement, Namespace, GeneralizableElement, etc., are classes in the UML meta-model. Such classes are often referred to as *meta-classes* (because they belong to the meta-model). Similarly, attributes in the meta-model are called *meta-attributes*, and operations are *meta-operations*. Any design expressed in UML can be represented by instances of these classes, their properties, and the links among them.

4.2.3 Four-Layer Meta-Model Architecture

The UML architecture is based on a four-layered meta-model structure, composed of the following layers: user objects, model, meta-model, and meta-meta-model. The purpose of these layers is summarized in table 4.1.

Layer	Description	Example
Meta-Meta-Model	The infrastructure for a meta-modeling architecture. Defines the language for specifying meta-models.	`Meta-Class`, `Meta-Attribute`, `Meta-Operation`
Meta-Model	An instance of a meta-meta-model. Defines the language for specifying a model.	`Class`, `Attribute`, `Operation`, `Component`
Model	An instance of a meta-model. Defines a language to describe an information domain.	`Person`, `MasterStudent`, `getValue()`, `doPayment()`, `University`,`ChangeCountry()`
User Objects (User Data)	An instance of a model. Defines a specific information domain.	`<Person: name="Paul", height=1.83>`, `654.56`, `France`

Table 4.1 – *Four-Layer Meta-Modeling Architecture*

User objects (also known as user data) are instances of a model. The primary responsibility of the user objects layer is to describe a specific information domain. For example, ‘Aline’ is an instance of a ‘Master Student’ class in a model for describing the EMOOSE students’ domain. Other examples of objects in the user objects layer are strings, numbers, records, etc., to denote a certain entity in an specific domain, as: `<Person: name=”Paul”, height=1.83>`, `654.56`, `France`, `STN_012001` and `<Origin=“Brazil”, Destination=”Portugal”>`.

A model is a concrete representation of something, such as the design of a software system. It is also an instance of a meta-model. The primary responsibility of the model layer is to define a language that describes an information domain (this is generally done with UML diagrams as the class diagram or use case diagram). Examples of objects in the modeling layer are: `ÉcoleDesMinesDeNantes`, `University`, `Country` and `$10,000`.

A meta-model is a model representing the structure and semantics of a particular set of models. It is also an instance of a meta-meta-model. The primary responsibility of the meta-model layer is to define a language for specifying models. The meta-model describes what the set of models mean, i.e. it describes how to interpret models using the Unified Modeling Language. Examples of meta-objects in the meta-modeling layer are: `Class`, `Attribute`, `Operation`, and `Component`.

The meta-meta-modeling layer forms the foundation for the meta-modeling architecture. The primary responsibility of this layer is to define the language for specifying a meta-model. A meta-meta-model defines a model at a higher level of abstraction than that of the meta-model. It allows defining multiple meta-models, and there can be multiple meta-meta-models associated with each meta-model. Examples of meta-meta-objects in the meta-meta-modeling layer are: `meta-class`, `meta-attribute`, and `meta-operation`. Thus, each meta-model will contain (meta) classes that are instances of `meta-class` in the meta-meta-model (`Class` in the meta-model is an instance of `Meta-Class` in the meta-meta-model). The same idea can be applied to attributes and operations.

This chapter is primarily concerned with the meta-model layer. The meta-model described in figure 4.3 is UML specific but, in meta-modeling, meta-models can be constructed for other models as well (see the GOODLY meta-model [Abreu et al., 1997; Abreu et al., 1999] in appendix B). In this document, the sentences “UML meta-model”, “UML semantic model” or “meta-model” are all referred to the UML meta-model.

4.3 ORGANIZATION OF THE META-MODEL

The complexity of the UML meta-model is managed by organizing it into three logical packages, namely *Foundation*, *Behavioral Elements*, and *Model Management*. These packages are, in turn, decomposed into sub packages. For example, the Foundation package consists of the *Core*, *Extension Mechanisms* and *Data Types* sub packages, as shown in figure 4.4.

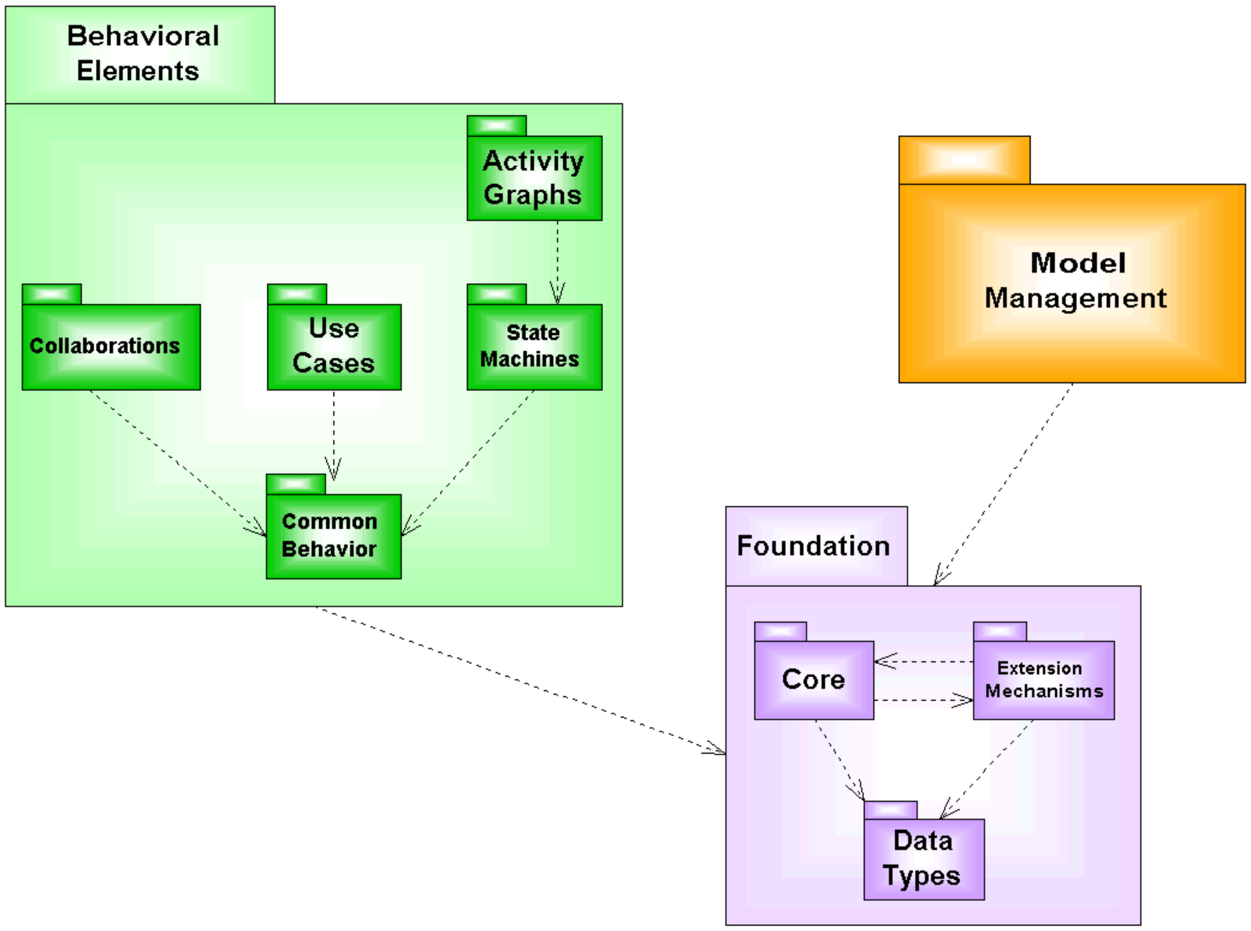

Figure 4.4 – *Meta-Model Structure*

Packages group meta-classes that show strong cohesion with each other and loose coupling with meta-classes in other packages.

For the formalization of existing sets of metrics, the most important packages are the Foundation and its sub packages. These are described in detail in the remaining part of this chapter, which uses [OMG, 1999] and [OMG, 2001] as references. Other packages are briefly explained, as they are out of the scope of the metrics formalization done in this thesis.

4.3.1 Foundation Package

It specifies the static structure of the models and is composed of the following sub packages:

- Data Types: defines the basic data structures for the language;
- Extension Mechanisms: specifies how model elements are customized and extended with new semantics;
- Core: specifies the basic concepts required for an elementary meta-model and defines an architectural backbone for attaching additional language constructs.

4.3.1.1 Data Types

The Data Types package is the sub package that specifies the different data types used to define UML. It has a simpler structure than the other packages because the semantics of the concepts inside this package are well known.

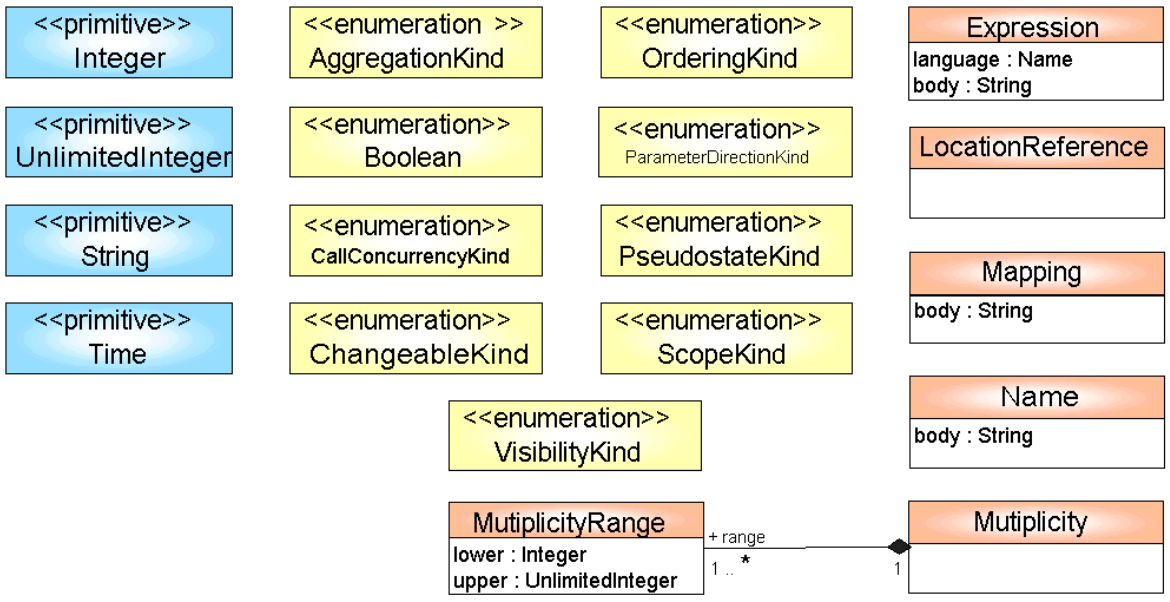
<<primitive>> Integer
<<primitive>> UnlimitedInteger
<<primitive>> String
<<primitive>> Time
<<enumeration >> AggregationKind
<<enumeration>> Boolean
<<enumeration>> CallConcurrencyKind
<<enumeration>> ChangeableKind
<<enumeration>> OrderingKind
<<enumeration>> ParameterDirectionKind
<<enumeration>> PseudostateKind
<<enumeration>> ScopeKind
<<enumeration>> VisibilityKind
Expression
language : Name
body : String
LocationReference
Mapping
body : String
Name
body : String
MutiplicityRange
lower : Integer
upper : UnlimitedInteger
+ range
1 .. *
1
Mutiplicity

Figure 4.5 – *UML Data Types*

Data types are used in the meta-model for declaring the classes' attribute types. Note that the data types are the ones used for defining UML and not the ones to be used by a user of UML (as in figure 4.1). The latter data types will be instances of the DataType meta-class defined in the meta-model. The former are described in detail in Appendix A.

4.3.1.2 Extension Mechanisms

Much of the expressiveness of UML comes from the extensibility mechanisms it provides. The designers of UML realized that they could not create a language that effectively described the semantics of all possible situations. Consequently, they created extensibility mechanisms that allow users to define custom extensions of the language, in order to accurately describe the semantics of specific information domains.

The UML extension mechanisms are useful for several purposes:

- to add new modeling elements for use in creating UML models;
- to define items that are not considered interesting or complex enough to be defined directly as UML meta-model elements;
- to define process-specific or implementation language-specific extensions;
- to attach arbitrary semantic and non-semantic information to model elements.

UML has three extensibility mechanisms:

- *Tagged values*: they can be seen as a metadata, i.e., a data to describe data [Silva and Videira, 2001]. This extensibility mechanism allows users to define new element properties, permitting arbitrary information to be attached to model elements. A tagged value is a *keyword-value pair* (<key, value>) that describes a property of a model element. The keywords are commonly referred to as *tags*. Some examples are {language=Java}, {author=Aline} and {NumberOfProcessor=3}.

- *Stereotypes*: they are meta-types, i.e., types to describe types [Silva and Videira, 2001]. They can be used to introduce additional distinctions between model elements that are not explicitly supported by the UML meta-model, allowing sub-classification of model elements. Applying a stereotype to a model element produces a *specialized* model element. This can be seen as an extensibility mechanism equivalent of inheritance since that any model element (meta-class) in the UML meta-model can be extended. Examples are <<primitive>> and <<enumeration>> in figure 4.5 and <<metaclass>>, as ModelElement.
- *Constraints*: They allow new semantic restrictions to be applied to elements. This makes it possible to specify additional constraints that should be obeyed by elements. This document uses OCL [OMG et al., 1997] as a language for specifying constraints over the model elements.

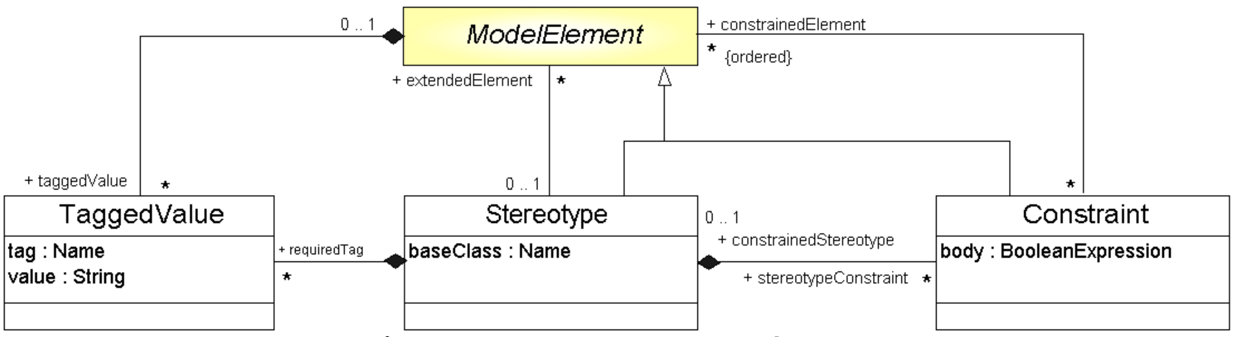

Figure 4.6 – *UML Extension Mechanisms*

4.3.1.3 Core

The Core package (figure 4.7) is the kernel of the sub packages that compose the UML Foundation package. It defines the basic abstract and concrete meta-model constructs (required for the development of object models) and forms an architectural backbone for attaching additional language constructs such as meta-classes, meta-associations, and meta-attributes.

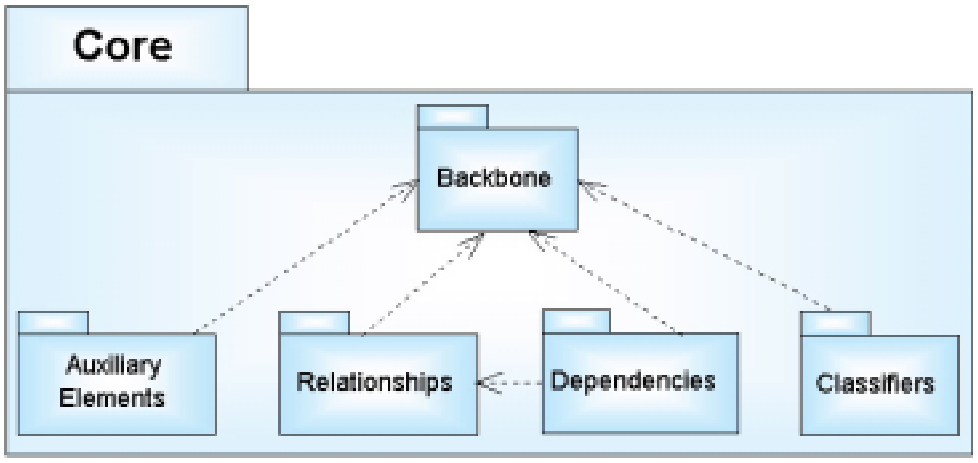

Figure 4.7 – *Core Package*

In sequence, the *Core Backbone*, *Core Relationships* and *Core Classifiers* are presented. *Core Dependencies* and *Core Auxiliary Elements* are left out, because they are not important in the context of this document. The Core Backbone is presented by building it step by step. Then, Core Relationships and Core Classifiers are introduced. For each meta-class, the corresponding attributes are explained.

Considering the Core package elements referred in this document, it is important to keep in mind that when referring to an *association end* for a binary association, the target end is the one whose properties are being discussed and the source end is the other end. Also, it is recommended to read the meta-attributes description using Appendix A as reference, to get the possible values for the data types used by the meta-classes. Furthermore, the inherited attributes are not replicated over all the meta-classes in the hierarchy (just the most important ones for the subclasses are repeated).

The information reproduced here is extracted from [OMG, 1999].

Core Backbone

Element

An element is an atomic constituent of a model. In the meta-model, an Element is the top meta-class in the meta-class hierarchy.

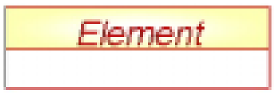

Figure 4.8 – *Core Element*

ModelElement

A model element is an abstraction drawn from the system being modeled. In the meta-model, a ModelElement is a named entity in a Model. It is the base for all modeling meta-classes in the UML. All other modeling meta-classes are either direct or indirect subclasses of ModelElement.

<u>*Attributes*</u>

- *name*: An identifier for the ModelElement within its containing Namespace.

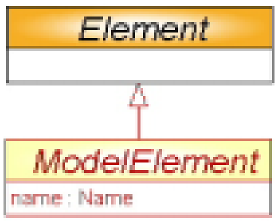

Figure 4.9 – *Core ModelElement*

GeneralizableElement

A generalizable element is a model element that may participate in a generalization relationship. In the meta-model, a GeneralizableElement can be a generalization of other GeneralizableElements (i.e., all Features defined in and all ModelElements contained in the ancestors are also present in the GeneralizableElement).

<u>*Attributes*</u>

- *isRoot*: specifies whether the GeneralizableElement is a root GeneralizableElement with ancestors or not;
- *isLeaf*: specifies whether the GeneralizableElement is a GeneralizableElement with descendents or not;
- *isAbstract*: specifies whether the GeneralizableElement may have a direct instance or not.

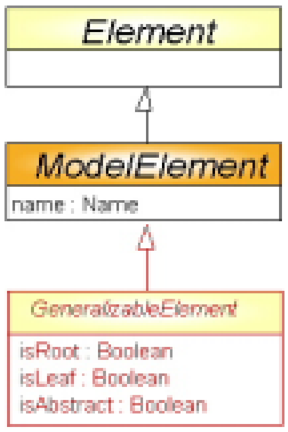

Figure 4.10 – *Core GeneralizableElement*

Namespace

A namespace is a part of a model that contains a set of ModelElements whose names designate a unique element within the namespace. In the meta-model, a Namespace is a ModelElement that can own other ModelElements, like Associations and Classifiers. The name of each owned ModelElement must be unique within the Namespace. Moreover, each contained ModelElement is owned by at most one Namespace. Explicit parts of a model element, such as the features of a Classifier, are not modeled as owned elements in a namespace. A namespace is used for unstructured contents such as the contents of a package or a class declared inside the scope of another class.

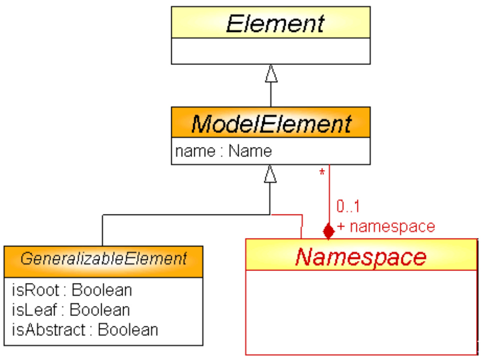

Figure 4.11 – *Core Namespace*

Classifier

A classifier is an element that describes behavioral and structural features. It is specialized in several specific forms, including class, data type, interface, component, and others that are defined in other meta-model packages. In the meta-model, a Classifier declares a collection of Features, such as Attributes, Methods, and Operations.

Classifier inherits the characteristics of GeneralizableElement and Namespace. As a GeneralizableElement, it may inherit Features and participation in Associations. As a Namespace, a Classifier may declare other Classifiers nested in its scope. It has a name, which is unique in the Namespace enclosing the Classifier.

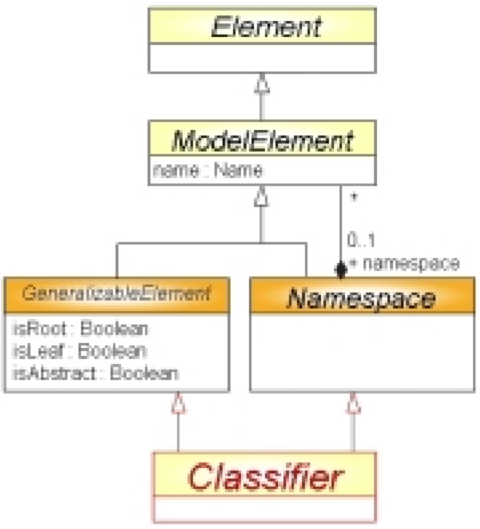

Figure 4.12 – *Core Classifier*

ElementOwnership

Element ownership defines the visibility of a ModelElement contained in a Namespace (figure 4.13). In the meta-model, ElementOwnership reifies the relationship between ModelElement and Namespace denoting the ownership of a ModelElement by a Namespace and its visibility outside the Namespace.

<u>*Attributes*</u>

- *isSpecification*: specifies whether the ownedElement is part of a specification or part of a realization for the containing namespace;
- *visibility*: specifies whether the ModelElement can be seen and referenced by other ModelElements. Its value can be one of the values defined by the VisibilityKind enumeration (see appendix A).

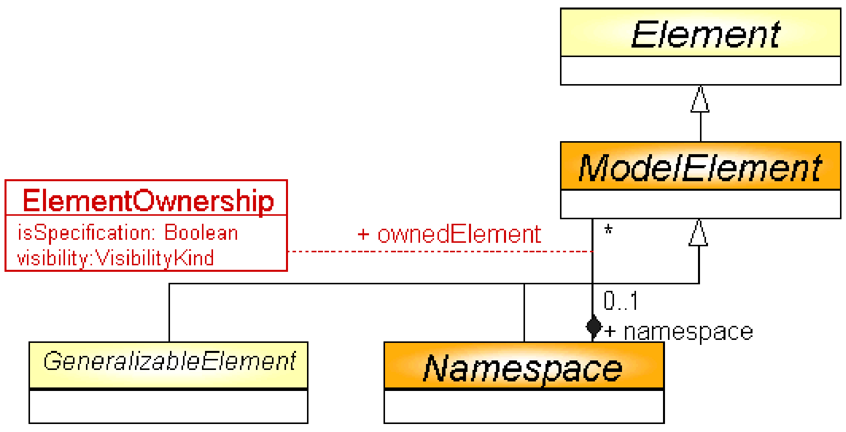

Figure 4.13 – *Core Namespace*

Feature

A feature is a property, like operation or attribute, which is encapsulated within a Classifier. In the meta-model, a Feature declares a behavioral or structural characteristic of an instance of a Classifier or of the Classifier itself.

Attributes

- *name* (Inherited from ModelElement): The name used to identify the Feature within the Classifier or instance. It must be unique across inheritance of names from ancestors including names of outgoing AssociationEnd (The meta-class AssociationEnd is explained later on in the *Core Relationships* section of this document);
- *ownerScope*: specifies whether Feature appears in each instance of the Classifier or whether there is just a single instance of the Feature for the entire Classifier. Possibilities are specified for the ScopeKind enumeration (Appendix A);
- *visibility*: specifies whether other Classifiers can use the Feature. Visibilities of nested Classifiers combine so that the most restrictive visibility is the result. Possibilities are specified for the VisibilityKind enumeration (Appendix A).

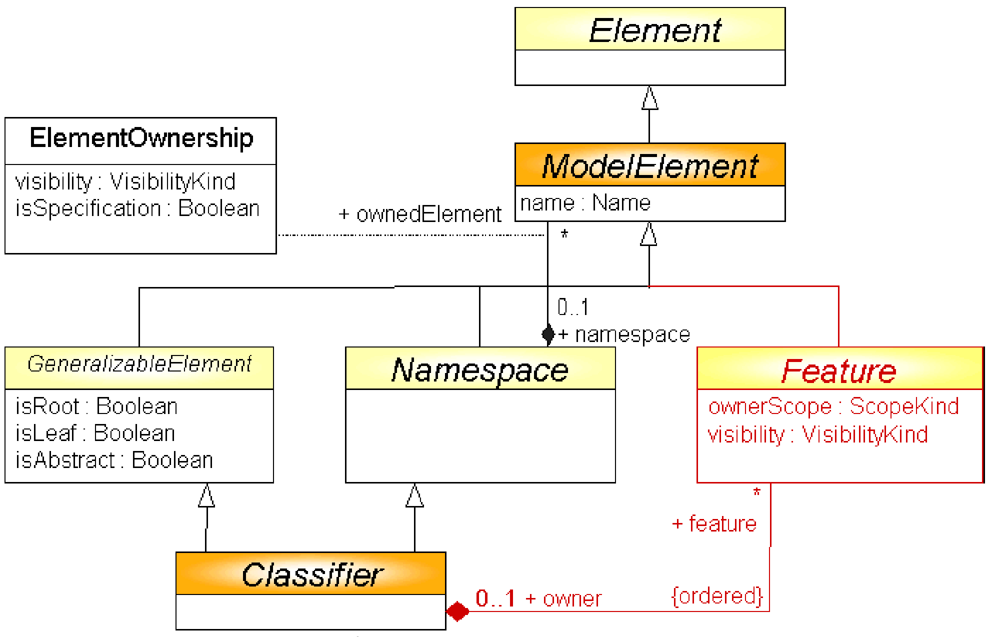

Figure 4.14 – *Core Feature*

StructuralFeature

A structural feature refers to a static feature of a model element. In the meta-model, a StructuralFeature declares a structural aspect of an instance of a Classifier, such as an Attribute. All the StructuralFeatures have a type.

Attributes

- *changeability*: specifies whether the value may be modified after the object is created. Possibilities are defined by the ChangeabilityKind enumeration (Appendix A);

- *multiplicity*: designates the possible number of data values for the attribute that may be held by an instance. The cardinality of the set of values is an implicit part of the attribute. In the most common case where the multiplicity is 1, then the attribute is a scalar (i.e., it holds exactly one value);

- *targetScope*: specifies whether the targets are instances or Classifiers. Possibilities are the ones of the ScopeKind enumeration (Appendix A).

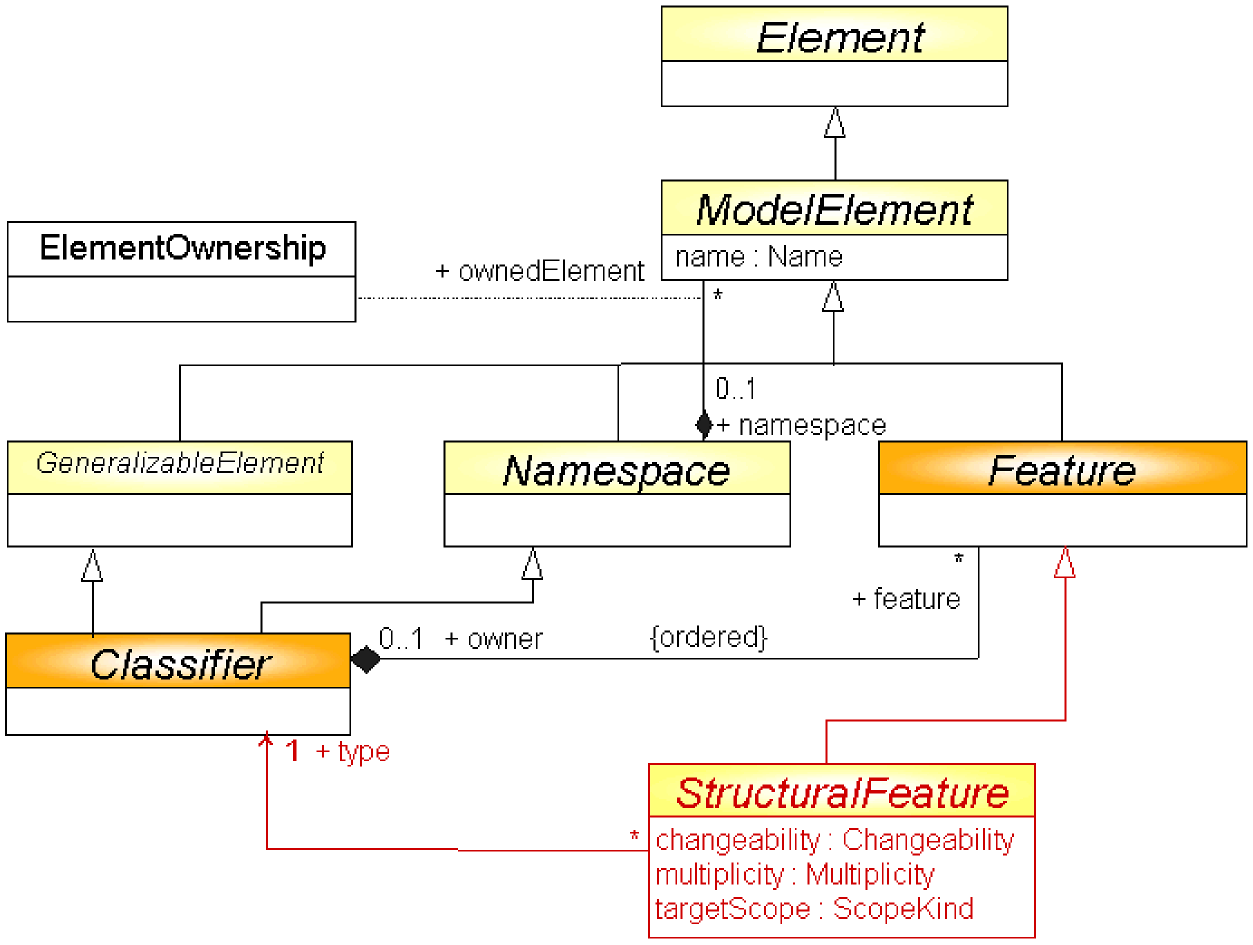

Figure 4.15 – *Core StructuralFeature*

Attribute

An attribute is a named slot within a classifier that describes a range of values that instances of the classifier may hold. In the meta-model, an Attribute defines the state of Classifier instances.

Attributes

- *name* (Inherited from ModelElement): is the name of the Attribute, which must be unique within its containing Classifier. This can be expressed in OCL as:

```
Classifier
self.feature -> select( a | a.oclIsKindOf ( Attribute ) )
                        -> forall ( p1, p2 | p1.name = p2.name implies p1 = p2)
```

- *initialValue*: An Expression specifying the value of the attribute upon initialization. It is meant to be evaluated at the time the object is initialized.

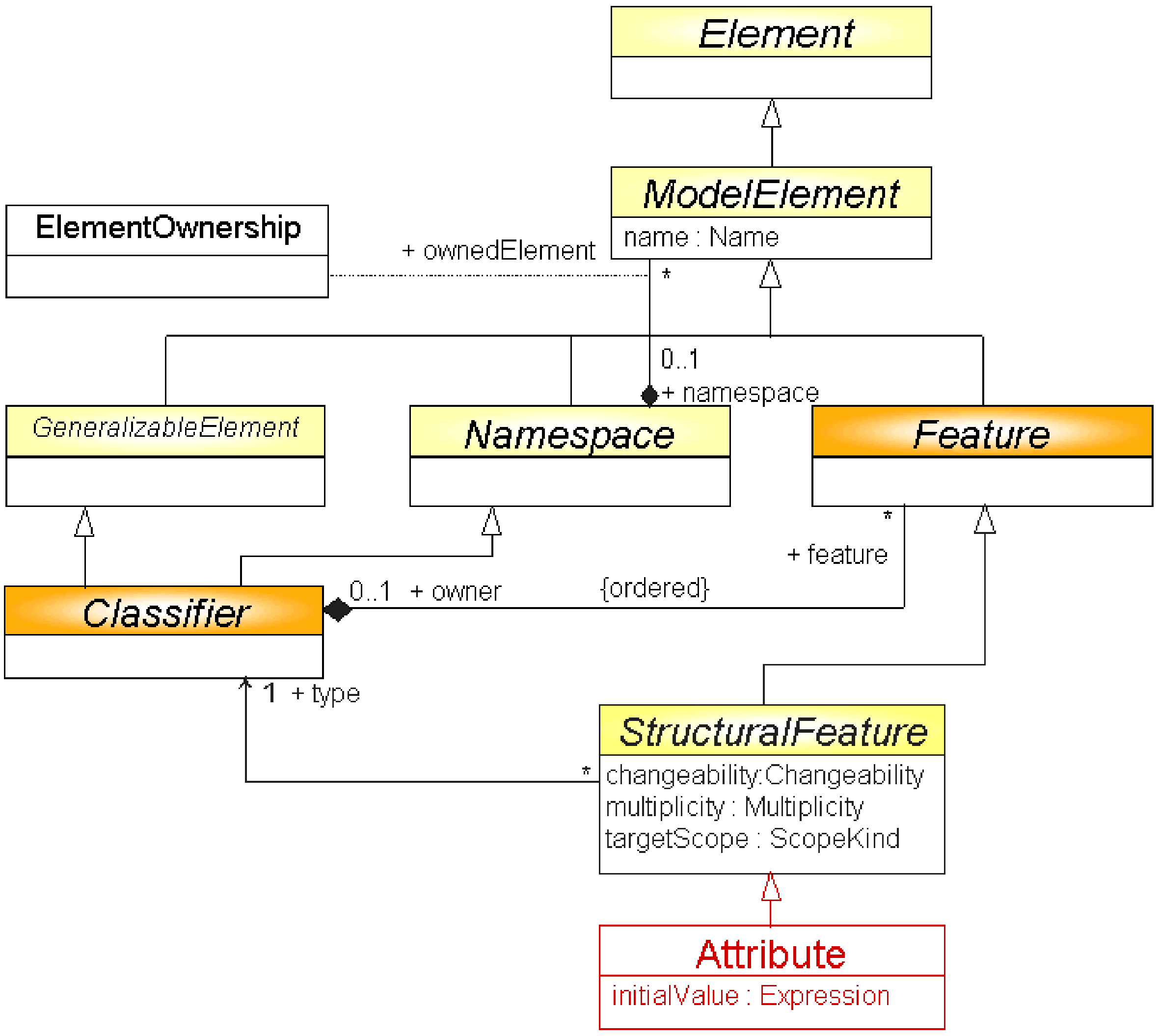

Figure 4.16 – *Core Attribute*

BehavioralFeature

A behavioral feature refers to a dynamic feature of a model element, such as an operation or method. In the meta-model, a BehavioralFeature specifies a behavioral aspect of a Classifier.

Attributes

- *name* (Inherited from ModelElement): the name of the Feature. The entire signature of the Feature must be unique within its containing Classifier. This can be expressed in OCL as:

```
Classifier
self.feature -> select( a | a.oclIsKindOf ( BehavioralFeature ) )
  -> forall ( p1, p2 | p1.name = p2.name and p1.parameter = p2.parameter implies p1 = p2)
```

- *isQuery*: specifies whether an execution of the Feature leaves the state of the system unchanged. True indicates that the state is unchanged; false indicates that side-effects do occur.

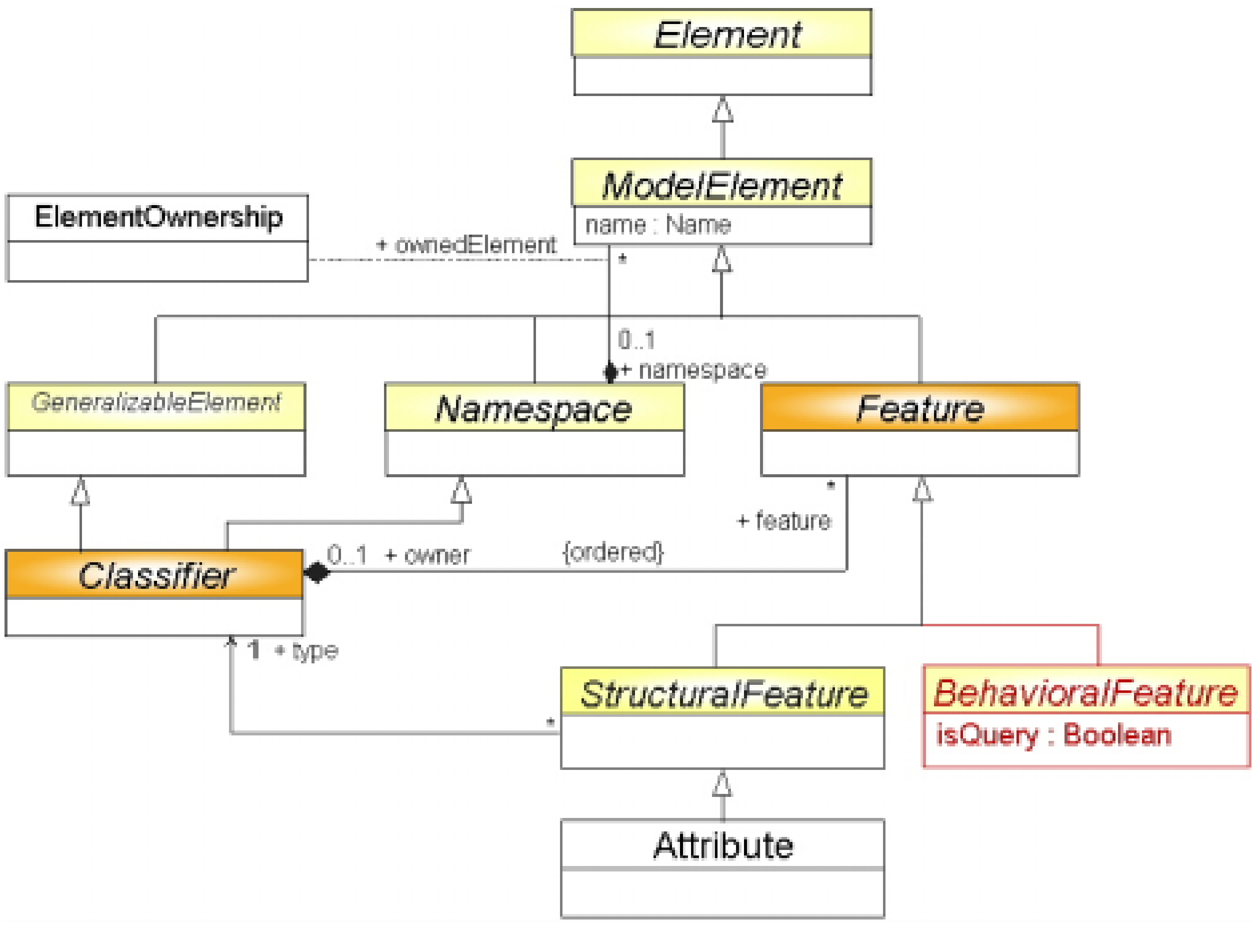

Figure 4.17 – *Core BehavioralFeature*

Parameter

A parameter is an unbound variable that can be changed, passed, or returned. A parameter may include a name, type, and direction of communication. Parameters are used in the specification of Operations, messages and events, templates, etc. In the meta-model, a Parameter is a declaration of an argument to be passed to, or returned from, an Operation or a Signal.

Attributes

- *name* (Inherited from ModelElement): The name of the Parameter, which must be unique within its containing Parameter list. This is modeled in OCL as:

```
BehavioralFeature
self.parameter -> forall( p1, p2 | p1.name = p2.name implies p1 = p2)
```

- *defaultValue*: An Expression whose evaluation yields a value to be used when no argument is supplied for the Parameter;

- *kind*: Specifies the kind of a Parameter. Possibilities are expressed by the ParameterDirectionKind enumeration.

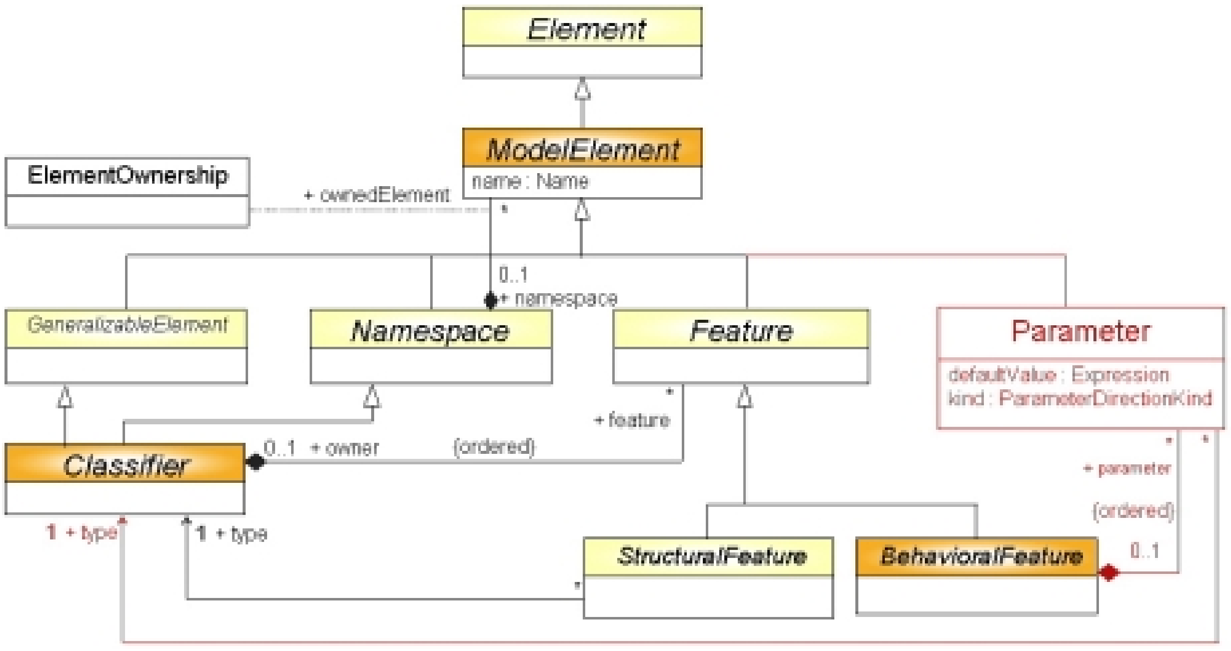

Figure 4.18 – *Core Parameter*

Operation

An operation is a service that can be requested from an object to effect behavior. It has a signature, which includes a name (inherited from ModelElement) and a list of actual parameters (from BehavioralFeature), including possible return values. In the meta-model, an Operation is a BehavioralFeature that can be applied to the instances of the Classifier that contains the Operation.

Attributes

- *concurrency*: specifies the semantics of concurrent calls to the same passive instance (i.e., an instance originating from a Classifier with isActive=false). Active instances control access to their own Operations so this property is usually (although not required in UML) set to sequential. Possibilities include the ones in CallConcurrencyKind enumeration;
- *isAbstract*: if true, then the operation does not have an implementation, and one must be supplied by a descendant. If false, the operation must have an implementation in the class or it must be inherited from an ancestor;
- *isLeaf*: if true, then the implementation of the operation may not be overridden by a descendant class. If false, then the implementation of the operation may be overridden by a descendant class;
- *isRoot*: if true, then the class must not inherit a declaration of the same operation. If false, then the class may inherit a declaration of the same operation.

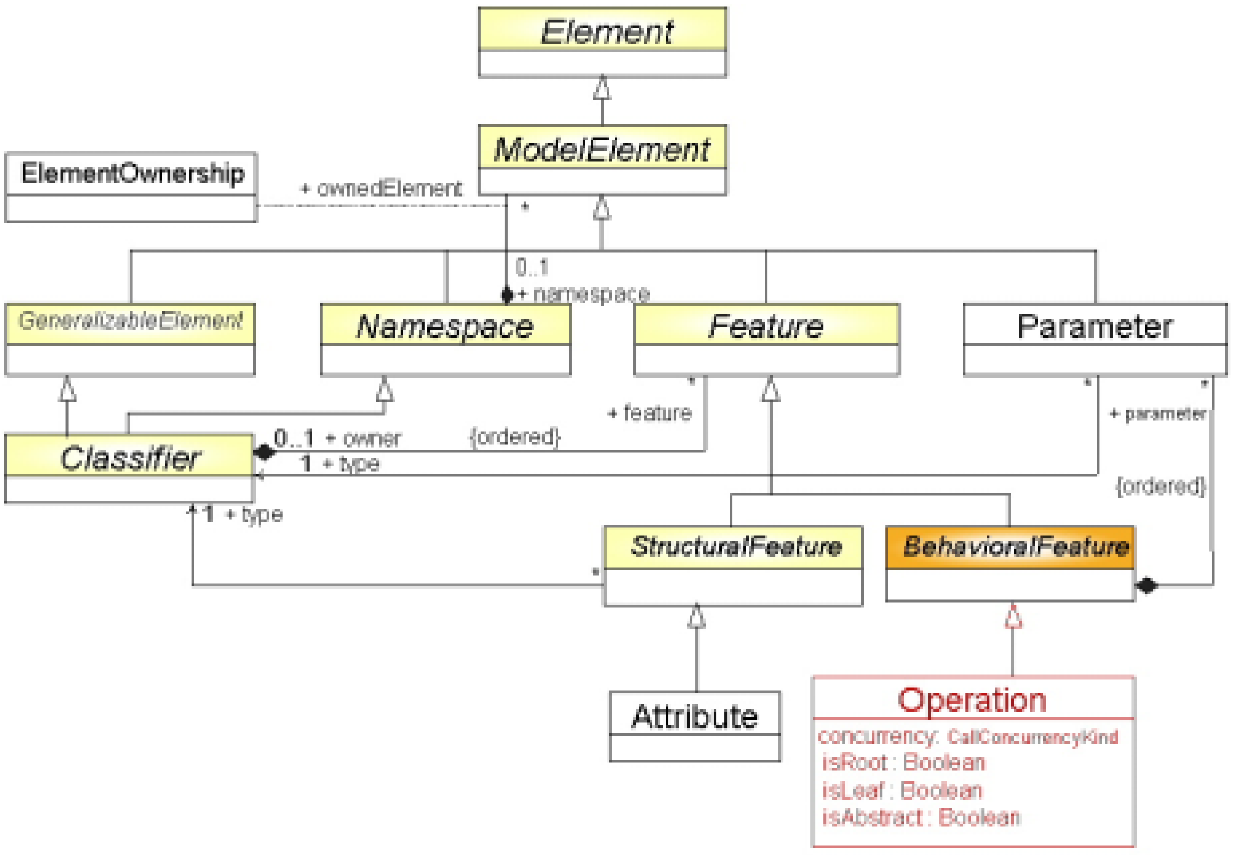

Figure 4.19 – *Core Operation*

Method

A method is the implementation of an operation. It specifies the algorithm or procedure that effects the results of an operation. In the meta-model, a Method is a declaration of a named piece of behavior in a Classifier and realizes one (directly) or a set (indirectly[14]) of Operations of the Classifier. The association named *specification* designates an Operation that the Method implements. The signature of the Operation and the Method must match. This can be expressed using the OCL, as:

```
Method
self.hasSameSignature( self.specification )
```

where *hasSameSignature* is a well-formedness rule included in the UML semantic model rules, which compares the signature.

Attributes

- *body*: the implementation of the Method as a ProcedureExpression[15].

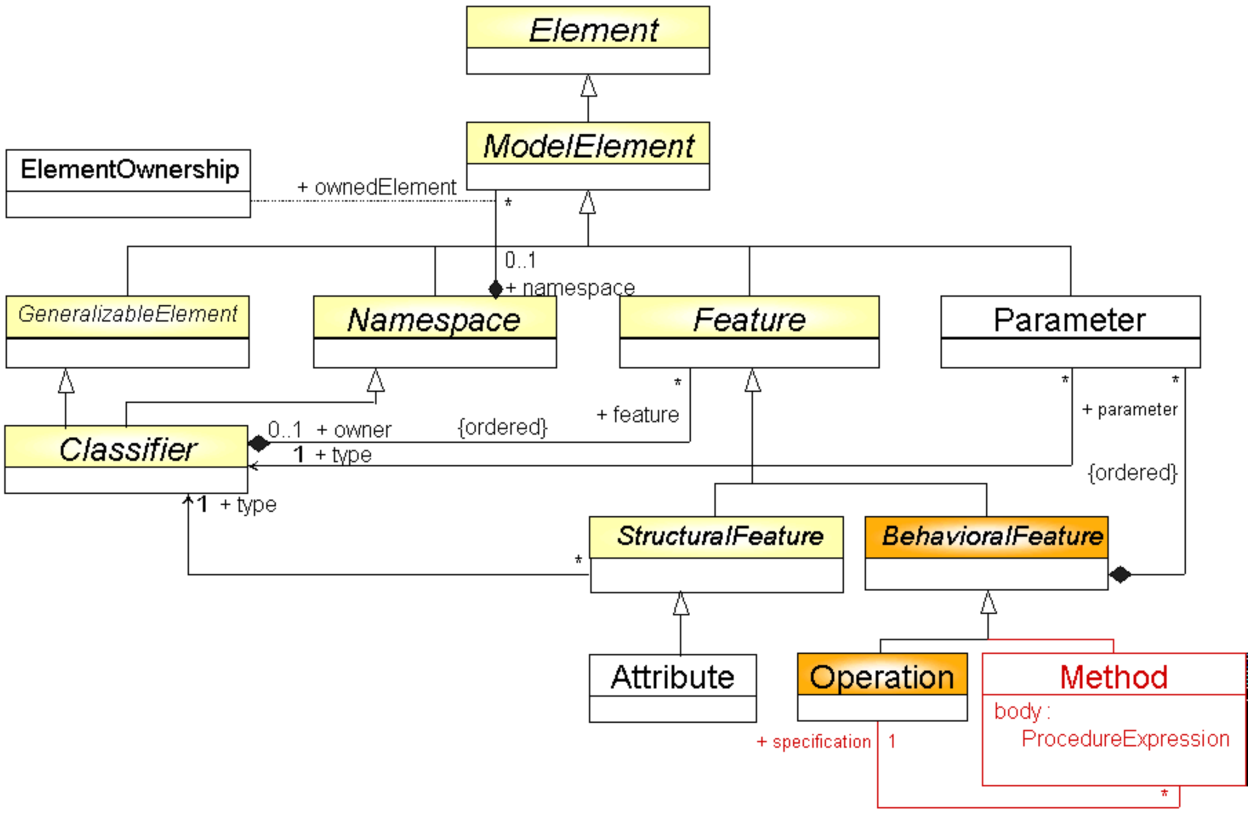

Figure 4.20 – *Core Method*

[14] Through inheritance, for example.
[15] In the meta-model *ProcedureExpression* defines a statement which will result in a change to the values of its environment when it is evaluated.

Finally, the complete Core Backbone is show in figure 4.21.

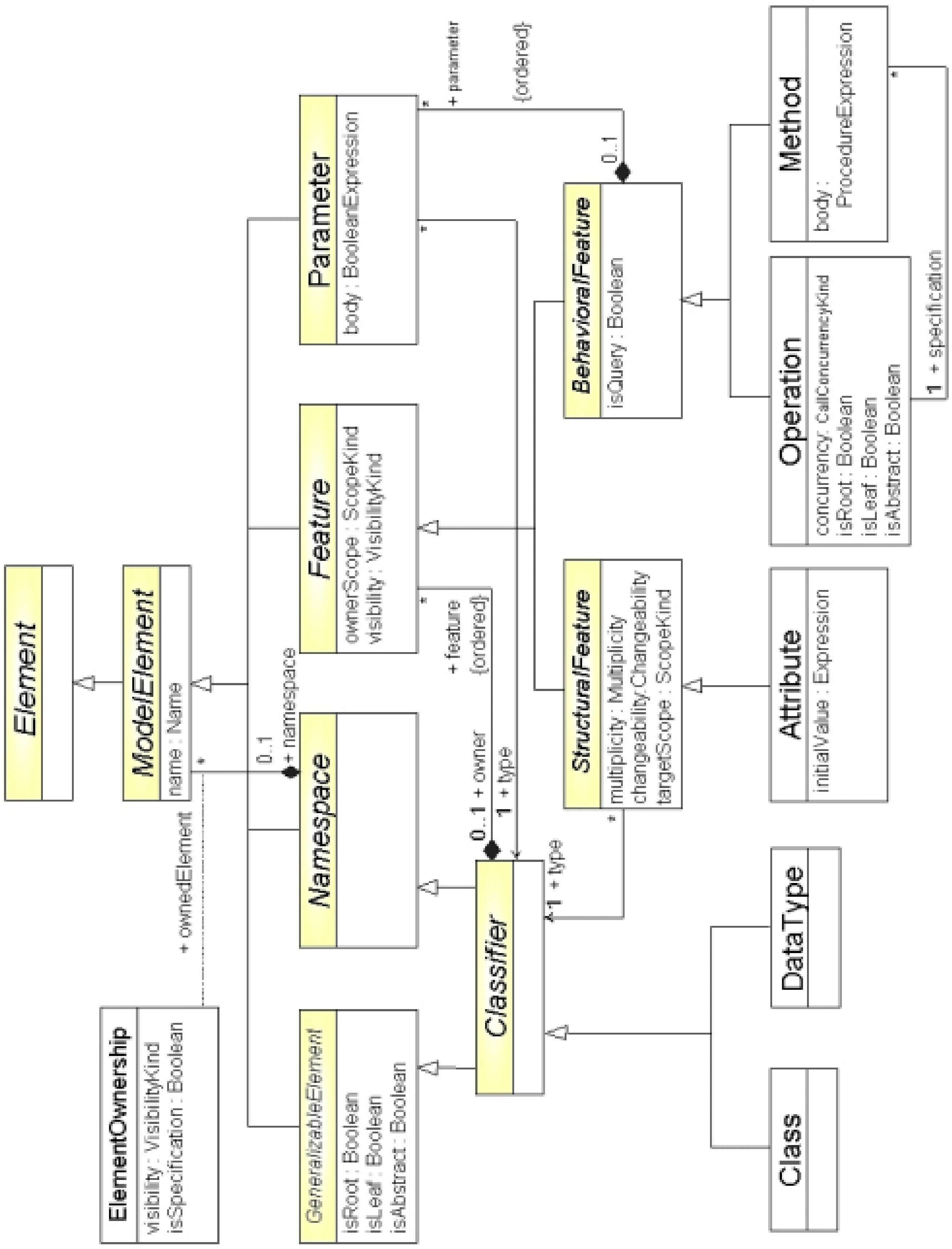

Figure 4.21 – *Core Backbone*

Core Relationships

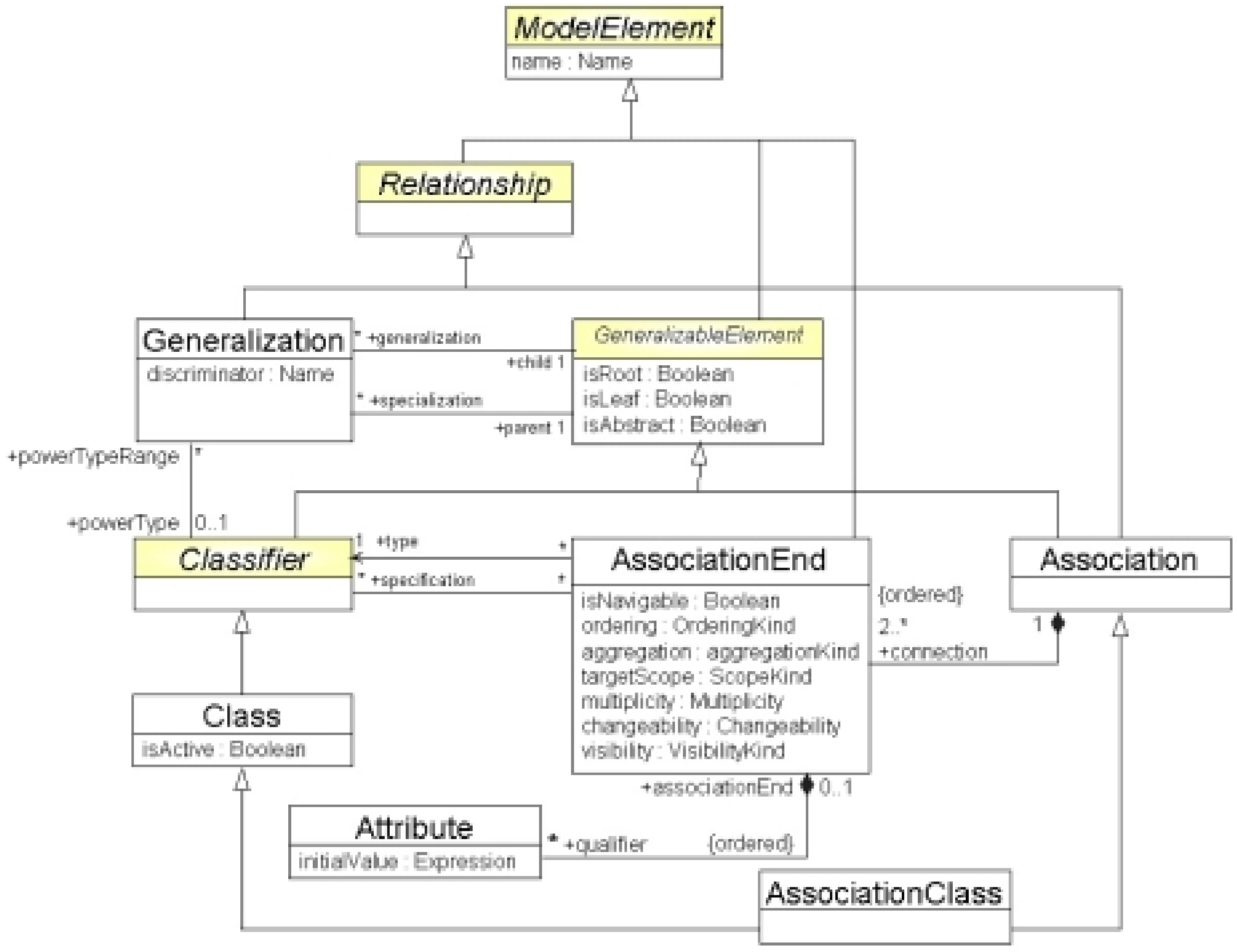

Figure 4.22 – *Core Relationships*

Association

An association defines a semantic relationship between classifiers. The instances of an association are a set of tuples relating instances of the classifiers. Each tuple value may appear at most once. In the meta-model, an Association is a declaration of a semantic relationship between Classifiers, such as Classes. An Association has at least two AssociationEnds. Each end is connected to a Classifier - the same Classifier may be connected to more than one AssociationEnd in the same Association. This allows instances of the same Classifier to be associated with each other. The Association represents a set of connections among instances of the Classifiers. An instance of an Association is a Link, which is a tuple of instances drawn from the corresponding Classifiers.

Attributes

- *name (inherited from ModelElement)*: The name of the Association that, in combination with its associated Classifiers, must be unique within the enclosing namespace (usually a Package).

AssociationClass

An association class is an association that is also a class. It not only connects a set of classifiers but also defines a set of features that belong to the relationship itself instead of to any of the participating classifiers. In the meta-model, an AssociationClass is a declaration of a semantic relationship between Classifiers, which has a set of features of its own. AssociationClass is a subclass of both Association and Class (i.e., each AssociationClass is both an Association and a Class); therefore, an AssociationClass has both AssociationEnds and Features.

AssociationEnd

An association end is an endpoint of an association, which connects the association to a classifier. Each association end is an ordered part of one association. In the meta-model, an AssociationEnd is part of an Association and specifies the connection of an Association to a Classifier. It has a name and defines a set of properties of the connection (e.g., which Classifier the instances must conform to, their multiplicity, and if they may be reached from the hooked instance via this connection).

Attributes

- *name* (Inherited from ModelElement): the role name of the end. When placed on a target end, provides a name for traversing from a source instance across the association to the target instance or set of target instances;
- *isNavigable*: when placed on a target end, specifies whether traversal from a source instance to its associated target instances is possible[16]. Specification of each direction across the Association is independent;
- *ordering*: when placed on a target end, specifies whether the set of links from the source instance to the target instance is ordered. The ordering must be determined and maintained by Operations that add links. Possibilities are the ones in the OrderingKind enumeration (Appendix A);
- *aggregation*: when placed on a target end, specifies whether the target end is an aggregation with respect to the source end. Only one end can be an aggregation. This rule can be expressed as:

```
Association
self.allConnections -> select( aggregation <> #none ) -> size <= 1
-- {At most one AssociationEnd may be an aggregation or composition}
Association::allConnections:Set( AssociationEnd )
= self.connection
```

  Possibilities are of AggregationKind type;
- *targetScope*: specifies whether the target value is an instance or a classifier. Its value is one of the ScopeKind enumeration;

[16] Remember that the target end is the one whose properties are being discussed and the source end is the other end.

- *multiplicity*: when placed on a target end, specifies the number of target instances that may be associated with a single source instance across the given Association;
- *changeability*: when placed on a target end, specifies whether an instance of the Association may be modified from the source end. Possibilities are of ChangeableKind enumeration type;
- *visibility*: specifies the visibility of the association end from the viewpoint of the classifier on the other end. Possibilities are the ones of the VisibilityKind enumeration.

Class

A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics. A class may use a set of interfaces to specify collections of operations it provides to its environment. In the meta-model, a Class describes a set of Objects sharing a collection of Features, including Operations, Attributes and Methods, that are common to the set of Objects.

Attributes

- *isActive*: specifies whether an Object of the Class maintains its own thread of control. If true, then an Object has its own thread of control and may run concurrently with other active Objects. The corresponding class is informally called an *active clas*s. If false, then Operations run in the address space and under the control of the active Object that controls the caller. The corresponding class is informally called a *passive clas*s.

Generalization

A generalization is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information. In the meta-model, a Generalization is a directed inheritance relationship, between a GeneralizableElement and a more general GeneralizableElement in a hierarchy. Generalization is a sub typing relationship (i.e., an instance of the more general GeneralizableElement may be substituted by an instance of the more specific GeneralizableElement).

Attributes

- *discriminator*: designates the partition to which the Generalization link belongs[17]. All of the Generalization links that share a given parent GeneralizableElement are divided into disjoint sets (partitions) by their discriminator names.

[17] In other words, the discriminator represents the name of the parent element in the inheritance relationship.

Core Classifiers

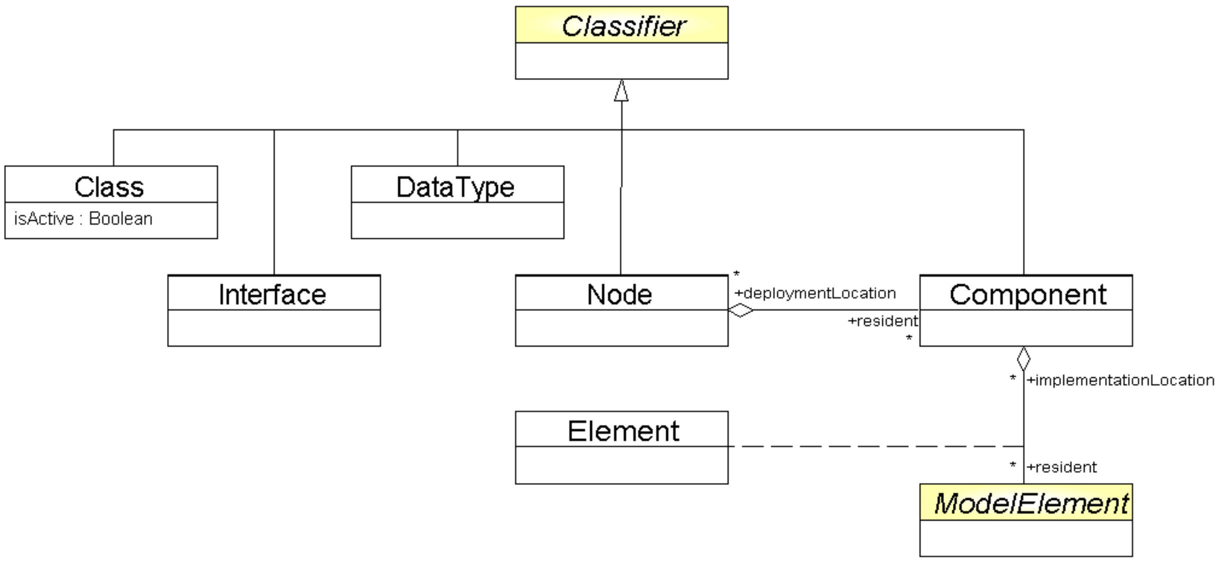

Figure 4.23 – *Core Classifiers*

Most of the subclasses of Classifier are not used in this document, but they are described here for the sake of meta-model completeness.

Component

A component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces, which represent services implemented by the elements resident in the component. These services define behavior offered by instances of the Component as a whole to other client Component instances. In the meta-model, a Component is a child of Classifier. It provides the physical packaging of its associated specification elements. As a Classifier, it may also have its own Features, such as Attributes and Operations, and realize Interfaces.

DataType

A data type is a type whose values have no identity (i.e., they are pure values). Data types include primitive built-in types (such as integer and string) as well as definable enumeration types (such as the predefined enumeration type boolean whose literals are false and true). In the meta-model, a DataType defines a special kind of Classifier in which Operations are all pure functions (i.e., they can return DataValues but they cannot change DataValues, because they have no identity). For example, an “add” operation on a number with another number as an argument yields a third number as a result; the target and argument are unchanged.

Interface

An interface is a named set of operations that characterize the behavior of an element. In the meta-model, an Interface contains a set of Operations that together define a service offered by a Classifier realizing the Interface. A Classifier may offer several services, which means that it may realize several Interfaces, and several Classifiers may realize the same Interface. Interfaces are GeneralizableElements. Interfaces may not have Attributes, Associations, or Methods. In other words, the interface can only contain Operations and Receptions[18]. This is expressed in OLC as:

```
Interface
self.allFeatures[19] -> forall( f | f.oclIsKindOf( Operation ) or f.oclIsKindOf( Reception ) )
```

Node

A node is a run-time physical object that represents a computational resource, generally having at least a memory and often processing capability as well, and upon which components may be deployed. In the meta-model, a Node is a subclass of Classifier. It is associated with a set of Components residing on the Node.

Core Dependencies

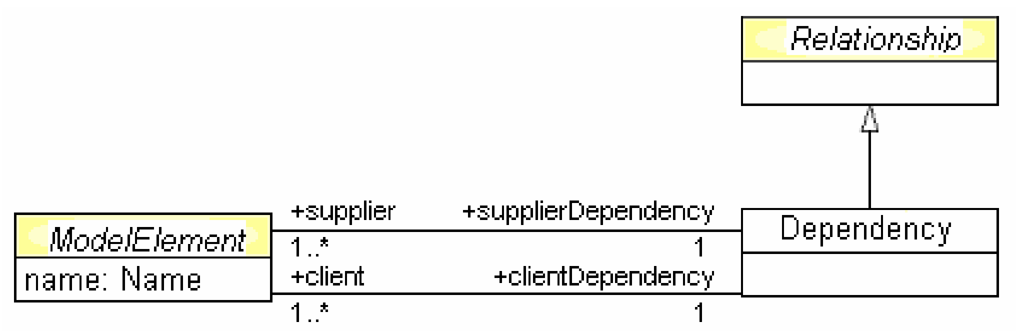

Figure 4.24 – *Core Dependencies*

Dependency

A dependency states that the implementation or functioning of one or more elements requires the presence of one or more other elements. In the meta-model, a Dependency is a directed relationship from a client[20] (or clients) to a supplier (or suppliers) stating that the client is dependent on the supplier[21] (i.e., the client element requires the presence and knowledge of the supplier element).

[18] *Reception* is a meta-class in the Common Behavior Package (see section 4.3.3). It is a declaration stating that a Classifier is prepared to react to the receipt of a Signal (Signal is a GeneralizableElement representing an asynchronous stimulus communicated between intances). The reception designates a Signal and specifies the expected behavior by the reception feature. It is used in state machines and, as such, it is out of the scope of this document.
[19] The operation *allFeatures( )* is described in the next chapter.
[20] *Client* is the element that is affect by the supplier element.
[21] Designates the element that is unaffected by a change.

4.3.2 Model Management

The Model Management package (dependent on the Foundation package – see picture 2.4) specifies how model elements are organized into models, packages, and subsystems, defining the classes Model, Package, and Subsystem respectively, which all serve mainly as grouping units for other ModelElements.

Packages are used within a Model to group ModelElements. A Subsystem is a special kind of Package that represents a behavioral unit in the physical system, and hence in the Model. In this section it is necessary to distinguish between the *physical system* being modeled (i.e., the subject of the model) and the *subsystem elements* that represent the physical system in the model.

An example of a physical system is a credit card service, which includes software, hardware and people. The UML model for this physical system might consist of a top-level subsystem called CreditCardService which is decomposed into subsystems for Authorization, Credit, and Billing.

Doing an analogy with the construction of houses, the house corresponds to the physical system, the blueprint corresponds to a model, and an element used in a blue print corresponds to a model element. There could be subsystems for the electricity and hydraulic systems.

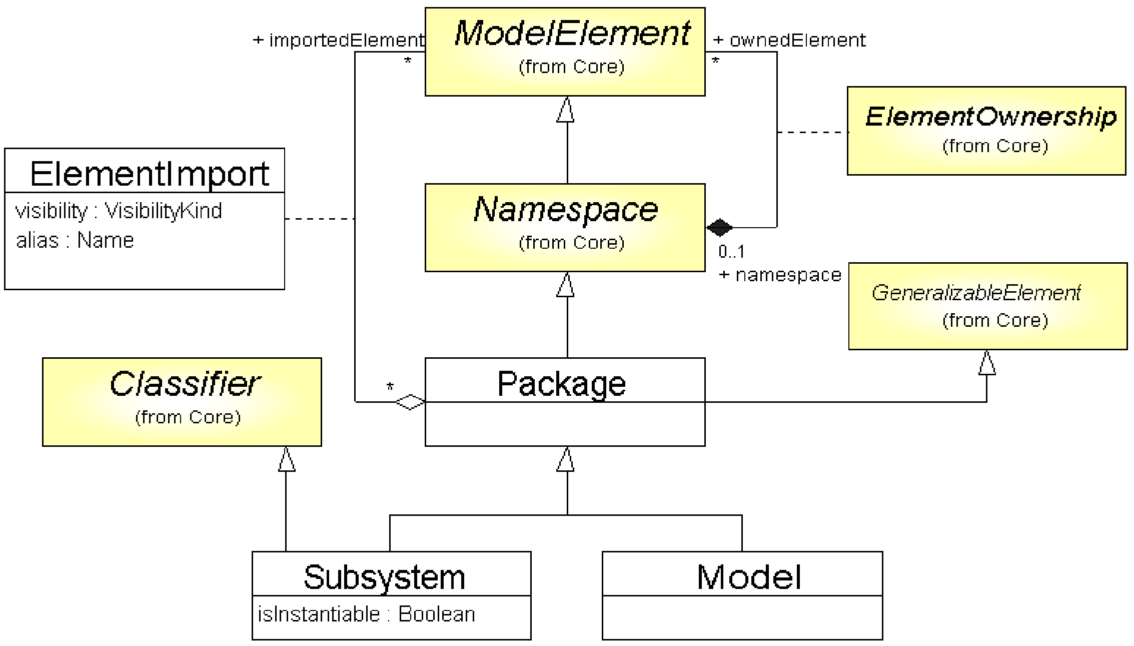

Figure 4.25 – *Model Management*

Package

A package is a grouping of model elements. In the meta-model, Package is a subclass of Namespace and GeneralizableElement and may contain ModelElements like other Packages, Classifiers, and Associations. A Package may also contain Constraints and Dependencies between ModelElements of the Package.

Each ModelElement of a Package has a visibility relative to the Package stating if the ModelElement is available to ModelElements in other Packages with a Permission («access» or «import») or Generalization relationship to the Package. An «access» or «import» permission from one Package to another allows public ModelElements in the target Package to be referenced by ModelElements in the source Package. They differ in that all public ModelElements in imported Packages are added to the Namespace of the importing Package, whereas the Namespace of an accessing Package is not affected at all.

The ModelElements available in a Package are those in the contents of the Namespace of the Package, which consists of owned and imported ModelElements, together with public ModelElements in accessed Packages.

Model

A model is an abstraction of a physical system, with a certain purpose. This purpose determines what is to be included in the model and what is irrelevant. Thus the model completely describes those aspects of the physical system that are relevant to the purpose of the model, at the level of detail that is given by the purpose.

In the meta-model, Model is a subclass of Package and, as such, it has a containment hierarchy of ModelElements describing the physical system. A Model also contains a set of ModelElements which represents the environment of the system together with their interrelationships, such as Dependencies, Generalizations, and Constraints.

Different Models can be defined for the same physical system, specifying it from different viewpoints. They may be nested, i.e. a Model may contain other Models.

Subsystem

A subsystem is a grouping of model elements that represents a behavioral unit in a physical system. A subsystem offers interfaces and has operations. In addition, the model elements of a subsystem can be partitioned into specification and realization elements, where the former, together with the operations of the subsystem, are realized by, i.e. implemented by, the latter.

In the meta-model, Subsystem is a subclass of both Package and Classifier. As such, it may have a set of Features, which are constrained to be Operations and Receptions. As presented above, this restriction can be stated as:

```
Subsystem
self.allFeatures -> forall( f | f.oclIsKindOf( Operation ) or f.oclIsKindOf( Reception ) )
```

The contents of a Subsystem are divided into two subsets: specification elements and realization elements. The former subset provides, together with the Operations of the Subsystem, a specification of the behavior contained in the Subsystem, while the ModelElements in the latter subset jointly provide a realization of the specification. Any kind of ModelElement can be a specification element or a realization element.

Still with the previous example of vehicles, figure 4.26 shows a sample design using packages. Elements of the types Class, Generalization and Association are owned by a surrounding namespace. The namespace is represented by an element that is an instance of Namespace.

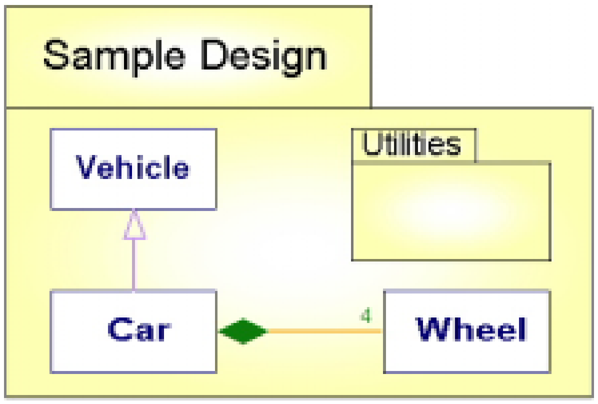

Figure 4.26 – *A Sample Design Using Packages*

The package element named “Utilities” is owned by the element named “Sample Design”. All namespace elements are owned by enclosing namespace elements, except for a top-level namespace that is the root of the data structure.

Figure 4.27 shows an object diagram corresponding to the subset of the UML meta-model that includes the meta information on the sample design presented in figure 4.26.

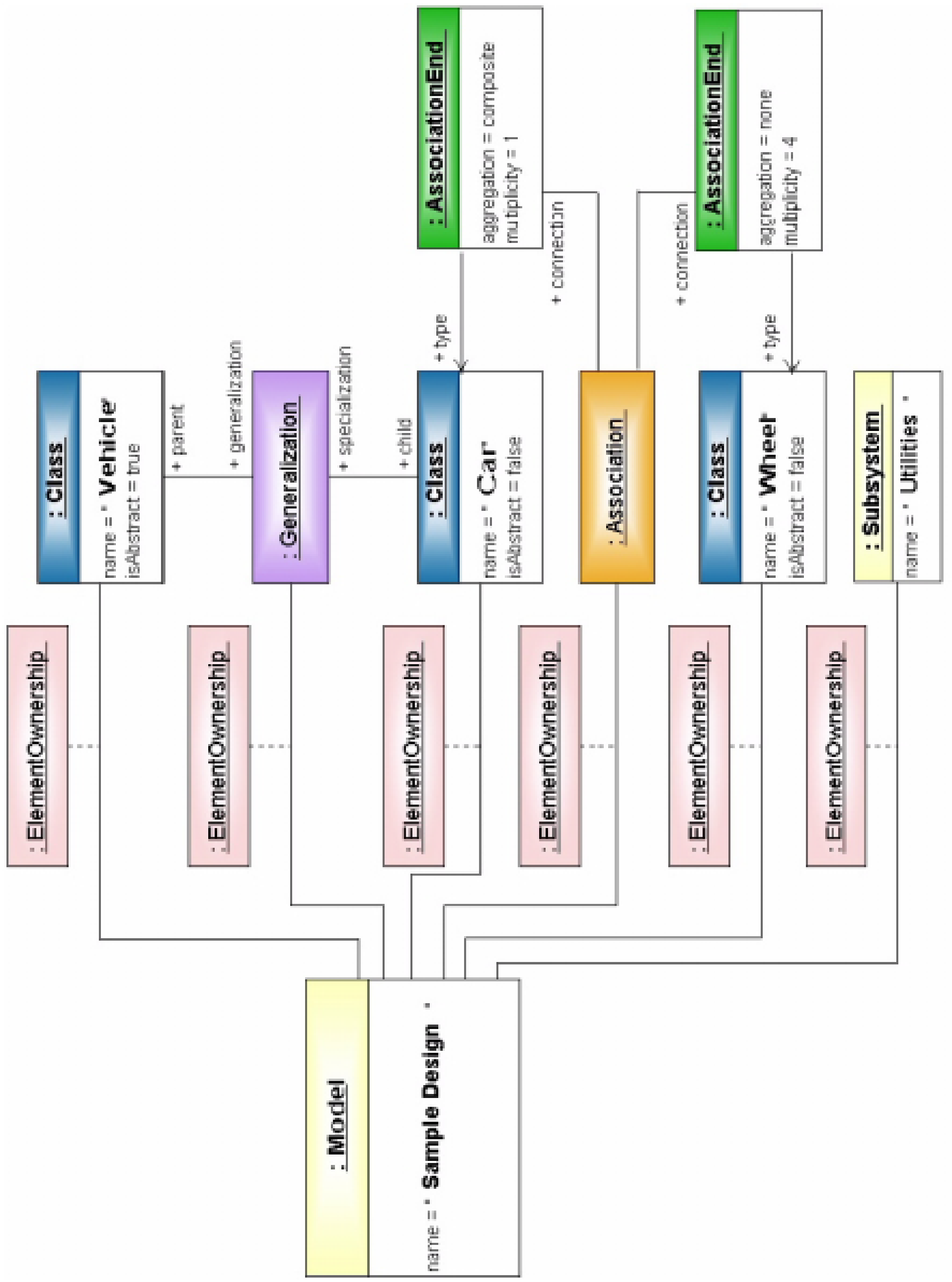

Figure 4.27 – *Meta-Model Objects for a Sample Design*

4.3.3 Behavioral Elements

The Behavioral Elements package (figure 4.28) is the one that specifies the dynamic behavior of models[22]. It is decomposed into the following sub packages: *Common Behavior*, *Collaborations*, *Use Cases*, *State Machines*, and *Activity Graphs*.

Common Behavior specifies the core concepts required for behavioral elements. The Collaborations package specifies a behavioral context for using model elements to accomplish a particular task. The Use Case package specifies behavior using actors and use cases. The State Machines package defines behavior using finite-state transition systems. The Activity Graphs package defines a special case of a state machine that is used to model processes.

The Behavioral Elements package is not the focus of this document, and is introduced here just to mention the entire meta-model. For a complete reference of this package, refer to [OMG, 1999].

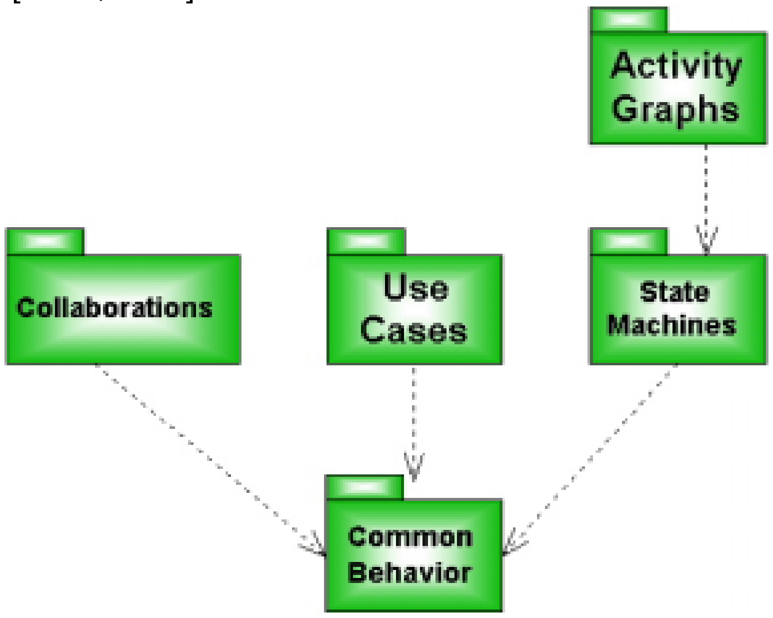

Figure 4.28 – *Behavioral Elements Package*

Common Behavior

The Common Behavior package is the most fundamental of the sub packages that compose the Behavioral Elements package. It specifies the core concepts required for dynamic elements and provides the infrastructure to support Collaborations, State Machines and Use Cases.

Collaborations

The Collaborations package specifies the concepts needed to express how different elements of a model interact with each other from a structural point of view. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

A Collaboration defines a specific way to use ModelElements in a Model. It describes how different kinds of Classifiers and their Associations are to be used in accomplishing a particular task. The Collaboration defines a restriction of, or a

[22] The Behavioral Elements package is not used in this document. It is included here just to present a complete overview of the UML meta-model.

projection of, a collection of Classifiers, i.e. what properties instances of the participating Classifiers must have when performing a particular collaboration.

A Collaboration is a GeneralizableElement. This implies that one Collaboration may specify a task which is a specialization of another Collaboration's task. A Collaboration may be presented in a diagram, either showing the restricted views of the participating Classifiers and Associations, or by showing instances and Links conforming to the restricted views.

Use Cases

The Use Cases package specifies the concepts used for definition of the functionality of an entity like a system. The package uses constructs defined in the Foundation package of UML as well as in the Common Behavior package.

The elements in the Use Cases package are primarily used to define the behavior of an entity, like a system or a subsystem, without specifying its internal structure. The key elements in this package are UseCase and Actor. Instances of use cases and instances of actors interact when the services of the entity are used. How a use case is realized in terms of cooperating objects, defined by classes inside the entity, can be specified with a Collaboration. A use case of an entity may be refined by a set of use cases of the elements contained in the entity. How these subordinate use cases interact can also be expressed in a Collaboration.

State Machines

The State Machine package specifies a set of concepts that can be used for modeling discrete behavior through finite state-transition systems. These concepts are based on concepts defined in the Foundation package as well as concepts defined in the Common Behavior package. This enables integration with the other sub packages in Behavioral Elements.

State machines can be used to specify behavior of various elements that are being modeled. For example, they can be used to model the behavior of individual entities (e.g., class instances) or to define the interactions (e.g., collaborations) among entities.

In addition, the state machine formalism provides the semantic foundation for activity graphs, which will be seen in the next section.

Activity Graphs

An activity graph is a special case of a state machine that is used to model processes involving one or more classifiers. Its primary focus is on the sequence and conditions for the actions that are taken, rather than on which classifiers perform those actions. Most of the states in such a graph are action states that represent atomic actions (i.e., states that invoke actions and then wait for their completion). Transitions into action states are triggered by events, which can be:

- The completion of a previous action state (completion events);
- The availability of an object in a certain state;
- The occurrence of a signal, or
- The satisfaction of some condition.

Activity graphs define an extended view of the State Machine package. Both state machines and activity graphs are essentially state transition models, and share many meta-model elements.

5

FLAME: Formal Library for Aiding Metrics Extraction

SYNTHESIS

In this chapter we present the first contribution of our work: a library of measures, named FLAME, which is mainly used to formalize metrics. The library is itself formalized with the Object Constraint Language (see chapter 3) upon the UML meta-model (see chapter 4).

FLAME was validated within an OCL tool, and the results of the formalization where compared with the ones generated by the MOODlib (appendix C) – another library for the MOOD [Abreu, 1993; Abreu, 1998] metrics formalization, which gave birth to FLAME. Since some new functions in FLAME could not be compared (because there were no equivalent in the MOODlib), their expected result was calculated manually for a set of test cases and then compared, successfully, with the corresponding computed values.

FLAME is further used in this document, in the formalization of the metrics definitions presented in chapter 6. Its creation was intended for metrics formalization but, in spite of this, it can be used for other purposes outside the scope of this document.

"Each problem that I solved became a rule which served afterwards to solve other problems."

Rene Descartes (1596-1650) – Discours de la Methode

5.1 FLAME: FORMAL LIBRARY FOR AIDING METRICS EXTRACTION

The functions herein presented are used to calculate the metrics formalized in chapter 6. Each of them is classified upon one of the following meta-classes in the UML meta-model: Attribute, Classifier, Feature, GeneralizableElement, ModelElement, Namespace, Operation and Package.

The idea to create the functions is based on the MOODlib (see appendix C), a library of functions used to calculate the MOOD [Abreu, 1995a] and MOOD2 [Abreu and Cuche, 1998] metrics. However, as the MOODlib is based on the GOODLY meta-model (see appendix B), the functions introduced in this section are completely new. Thus, this set of functions is useful to formalize not only the MOOD metrics, but all the metrics discussed in this document. Therefore, it is designated *FLAME* (Formal Library for Aiding Metrics Extraction).

A few functions are included in the UML meta-model (version 1.3) itself [OMG, 1999], and they were copied or altered here, to make this library as complete as possible. Notwithstanding, all the formalization, done with OCL, is part of the work performed for the completion of this thesis.

The tables in section 5.1.1 summarize the functions, which are classified according to their context in the UML meta-model. In sequence, the functions are outlined in section 5.1.2, in the alphabetical order of their contexts[23].

5.1.1 Existing Functions in FLAME

This section shows the names, the acronyms and the return types of all the functions created to compose FLAME and to assist the extraction of the metrics formalized on chapter 6.

It is possible to observe the distribution of the functions in categories, according to the context they are applied upon the UML meta-model.

Some of the function names were copied from the MOODlib, while others were adapted or simply created. However, most of the names are new, and they follow some conventions. For example, note that all the names that finish with the word "*Number*" have an integer return type. In addition, the functions that return sets have no acronyms, because it would be complicated to create short acronyms for all the functions avoiding repetition.

Acronym	Name	Return Type
AUN	Attribute Use Number	`Integer`

Table 5.1 – *Functions at Attribute Context*

[23] This means that some functions are used before being presented and that is necessary to move along this chapter looking for some definitions. The alphabetical order was chosen because it was difficult to arrange the functions in a good order.

Acronym	Name	Return Type
FCV	Feature to Classifier Visibility	`Boolean`
—	Coupled Classes	`Set( Classifier )`
—	Feature To Attribute Set	`Set( Attribute )`
—	Feature To Operation Set	`Set( Operation )`
—	New Features	`Set ( Features )`
—	Defined Features	`Set ( Features )`
—	Directly Inherited Features	`Set ( Features )`
—	All Inherited Features	`Set ( Features )`
—	Overridden Features	`Set ( Features )`
—	All Features	`Set ( Features )`
—	New Attributes	`Set( Attribute )`
—	Defined Attributes	`Set( Attribute )`
—	Directly Inherited Attributes	`Set( Attribute )`
—	All Inherited Attributes	`Set( Attribute )`
—	Overridden Attributes	`Set( Attribute )`
—	All Attributes	`Set( Attribute )`
—	New Operations	`Set( Operation )`
—	Defined Operations	`Set( Operation )`
—	Directly Inherited Operations	`Set( Operation )`
—	All Inherited Operations	`Set( Operation )`
—	Overridden Operations	`Set( Operation )`
—	All Operations	`Set( Operation )`
—	All Contents	`Set( ModelElement )`
—	Associations	`Set( Association )`
—	All Opposite Association Ends	`Set( AssociationEnd )`
—	Opposite Association Ends	`Set( AssociationEnd )`
NAN	New Attributes Number	`Integer`
DAN	Defined Attributes Number	`Integer`
IAN	Inherited Attributes Number	`Integer`
OAN	Overridden Attributes Number	`Integer`
AAN	Available Attributes Number	`Integer`
NON	New Operations Number	`Integer`
DON	Defined Operations Number	`Integer`
ION	Inherited Operations Number	`Integer`
OON	Overridden Operations Number	`Integer`
AON	Available Operations Number	`Integer`

PRIAN	Private Attributes Number	`Integer`
PROAN	Protected Attributes Number	`Integer`
PUBAN	Public Attributes Number	`Integer`
PRION	Private Operations Number	`Integer`
PROON	Protected Operations Number	`Integer`
PUBON	Public Operations Number	`Integer`

Table 5.2 – *Functions at Classifier Context*

Acronym	Name	Return Type
FUN	Feature Use Number	`Integer`

Table 5.3 – *Functions at Feature Context*

Acronym	Name	Return Type
—	Is Root	`Boolean`
—	Is Leaf	`Boolean`
—	Children	`Set( GeneralizableElement )`
—	Descendants	`Set( GeneralizableElement )`
—	Parents	`Set( GeneralizableElement )`
—	Ascendants	`Set( GeneralizableElement )`
CHIN	Children Number	`Integer`
DESN	Descendants Number	`Integer`
PARN	Parents Number	`Integer`
ASCN	Ascendants Number	`Integer`

Table 5.4 – *Functions at GeneralizableElement Context*

Acronym	Name	Return Type
—	Client	`Set( ModelElement )`
—	All Clients	`Set( ModelElement )`

Table 5.5 – *Functions at ModelElement Context*

Acronym	Name	Return Type
—	Contents	`Set( ModelElement )`

Table 5.6 – *Functions at Namespace Context*

Acronym	Name	Return Type
OUN	Operation Use Number	`Integer`

Table 5.7 – *Functions at Operation Context*

Acronym	Name	Return Type
—	Is Internal	`Boolean`
—	All Classes	`Set( Class )`
—	Internal Base Classes	`Set( Classifier )`
—	Base Classes	`Set( Classifier )`
—	Base Classes in Packages	`Set( Classifier )`
—	Internal Supplier Classes	`Set( Classifier )`
—	Supplier Classes	`Set( Classifier )`
—	Supplier Classes in Packages	`Set( Classifier )`
—	Related Classes	`Set( Classifier )`
CN	Classes Number	`Integer`
PNAN	Package New Attributes Number	`Integer`
PDAN	Package Defined Attributes Number	`Integer`
PIAN	Package Inherited Attributes Number	`Integer`
POAN	Package Overridden Attributes Number	`Integer`
PAAN	Package Available Attributes Number	`Integer`
PNON	Package New Operations Number	`Integer`
PDON	Package Defined Operations Number	`Integer`
PION	Package Inherited Operations Number	`Integer`
POON	Package Overridden Operations Number	`Integer`
PAON	Package Available Operations Number	`Integer`
EILN	External Inheritance Links Number	`Integer`
IILN	Internal Inheritance Links Number	`Integer`
PILN	Packages Inheritance Links Number	`Integer`
ECLN	External Coupling Links Number	`Integer`
ICLN	Internal Coupling Links Number	`Integer`
PCLN	Packages Coupling Links Number	`Integer`
AVN	Attribute Visibility Number	`Integer`
OVN	Operation Visibility Number	`Integer`
FVN	Feature Visibility Number	`Integer`
APV	Attribute to Package Visibility	`Percentage`
OPV	Operation to Package Visibility	`Percentage`
FPV	Feature to Package Visibility	`Percentage`

Table 5.8 – *Functions at Package Context*

5.1.2 Formal Description of the functions in FLAME

This section presents the formal definition of the functions that compose FLAME, using OCL and the UML meta-model as background. In spite of the division of the functions according to their context in the UML meta-model, the functions are further classified as *general*, *set*, *percentage* or *counting* functions.

General functions are those that return booleans. Set functions return set of elements, which can have the type of any meta-class in the UML meta-model. Percentage functions return a value representing a percentage and, finally, counting functions return integers.

Some of them have interesting properties that can generate some doubts, which consequently could lead to a different formalization of the function. In order to show also these doubtful points in FLAME, when such properties arise, they are placed after the function, in the item "*Discussion*". The consequence of our choices to solve the doubts can affect the results of other functions and/or metrics[24]. When this happens, the affected functions are mentioned over the item "*Consequences*".

The functions are displayed in the same order than presented in the tables above.

Functions at Attribute Context

Attribute Counting Functions

Name	**AUN – Feature Use Number**
Informal Definition	*Number of ModelElements that use the Attribute (excludes the ModelElement where the Attribute is declared).*
Formal Definition	***Attribute:: AUN( ): Integer*** *= self.FUN( )*
Comments	

[24] All the metrics mentioned in the "*Consequences*" section are explained and formalized in chapter 6.

Functions at Classifier Context

Classifier General Functions

Name	***FCV – Feature to Classifier Visibility***
Informal Definition	*Indicates if a Classifier can access the Feature.*
Formal Definition	***Classifier:: FCV(f: Feature): Boolean*** *= self.allFeatures( ) -> exists(* *(f.owner = self) or* *(f.visibility = #public) or* *((f.visibility = #protected) and* *(self.ascendants( ).oclAsType(Classifier).allFeatures( ) -> includes(f)))) or* *((self.allFeatures( ) = oclEmpty(Set(Feature))) and (f.visibility = #public))*
Comments	*One Classifier can access the feature “f” when:* - *It has features to access “f” and* - *It is the owner of the Feature;* - *The Feature is public;* - *The Feature is protected and belongs to one ascendant of the current Classifier or* - *It has no Features, but “f” is public.*

DISCUSSION

Should an empty class (a class without features) exist in the model? In general this does not happen but it is not possible to forbid their use in the models. So, should this empty class access a public feature (even if it has no ways to access it (because it is empty)?

One example of empty class is Membership, in the Royal and Loyal example (see chapter 3). Suppose one of the features in the Royal and Loyal example is public and, as it happens, all the classes are in the same package. For counting purposes, it is expected that all the 12 classes can access this feature. That is the reason why the clause

((self.allFeatures() = oclEmpty(Set(Feature))) and (f.visibility = #public))

was included in the formalization. Besides, we observed that the result of applying FCV to Membership, with a public feature should result in true, according to the results of metrics extraction.

CONSEQUENCES

The inclusion of empty classes in the function results affects the functions FVN and FPV, as well as the metrics MFA and MAA (from QMOOD), PRF and ARF (from MOOD and MOOD2).

Classifier Set Functions

Name	***coupledClasses***
Informal Definition	*Set of Classifiers to which the current Classifier is coupled (excluding inheritance).*
Formal Definition	***Classifier:: coupledClasses( ): Set(Classifier)*** *= self.allOppositeAssociationEnds( ).type* *-> union(self.allAttributes( ).type* *-> union(self.allOperations( ).parameter.type* *-> union(self.allOperations( ).type)))* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) -> asSet( )*
Comments	*This function includes the coupled classes corresponding to:* - *Classes directly associated with the current one;* - *Class Attributes;* - *Class Operations Parameters type;* - *Class Operations return type.* *The function does not include:* - *Method local Attributes;* - *Attributes from other Classes used by the Classifier's Methods;* - *Receivers from messages sent by the Classifier's Methods and* - *The Data Types.*

DISCUSSION

Should the date types included in the programming language be rejected? We think this rejection results in more flexibility. The data types should be considered in the function only when the user model is related to data types directly, for example when the user is extending the language data types system. In this ways, the data types of language will be the real entities of the user model, and as such they will be classes (and not anymore data types). Like this the results will contain the data types as coupled classes. The opposite solution (which is to include all the data types of the language) implies that there is no choice for the user in counting the data types, because they will always be there, even when they do not make sense. For example, in the Royal and Loyal system, we considered to be nonsense to say that the class Transaction is coupled to the Integer class, because Integer is a data type in most of the programming languages. However if the user wants to create a new data types system, and wants to sub classify the Integer class, as the current user's model includes the class Integer, it will be considered a class and not a data type. This is what happens with the Date class in the Royal and Loyal example. It is considered a class and not a data type, although some languages can use Date as a data type.

Should a classifier coupled to itself be counted? According to the function *coupledClasses* in the MOODlib, couplings involving only one classifier (a classifier associated with itself) do not count. We preserved this choice here.

How to go beyond the UML meta-model restrictions to include in the coupling the features used by methods? As the UML meta-model does not specify how the method bodies are composed, and it is not possible to know what a method body has, the function does not consider the local attributes declared inside the methods, as well as the attributes from other classes used in the method body and receivers from messages sent in the body of the class methods.

CONSEQUENCES

These choices affect the functions *internalSupplierClasses* and *ICLN*, and the metrics *CBO* (from MOOSE), *DAC* (from EMOOSE), *ICF* and *ECF* (from MOOD and MOOD 2).

Name	***feature2AttributeSet***
Informal Definition	*Subset of Attributes (from one set of Features) belonging to the current Classifier.*
Formal Definition	***Classifier:: feature2AttributeSet(s: Set(Feature)): Set(Attribute)*** *= s -> select(f: Feature | f.oclIsKindOf(Attribute))* *-> collect(f | f.oclAsType(Attribute)) -> asSet*
Comments	

Name	***feature2OperationSet***
Informal Definition	*Subset of Operations (from one set of Features) belonging to the current Classifier.*
Formal Definition	***Classifier:: feature2OperationSet( ): Set(Operation)*** *= s -> select(f: Feature | f.oclIsKindOf(Operation))* *-> collect(f | f.oclAsType(Operation)) -> asSet*
Comments	

Name	***newFeatures***
Informal Definition	*Set of Features declared in the current Classifier. This definition excludes inherited Features (and consequently, it excludes overridden Features).*
Formal Definition	***Classifier:: newFeatures(): Set(Feature)*** *= definedFeatures( ) – allInheritedFeatures( )*
Comments	

Name	***definedFeatures***
Informal Definition	*Set of Features declared in the Classifier, including overridden Operations.*
Formal Definition	***Classifier:: definedFeatures( ): Set(Feature)*** *= self.feature -> asSet*
Comments	

Name	***directlyInheritedFeatures***
Informal Definition	*Set of directly inherited Features.*
Formal Definition	***Classifier:: directlyInheritedFeatures( ): Set(Feature)*** *= self.parents( )* *-> iterate(elem: GeneralizableElement; acc: Set(Feature) = oclEmpty(Set(Feature))* *| acc -> union (elem.oclAsType(Classifier).definedFeatures( )))*
Comments	

Name	***allInheritedFeatures***
Informal Definition	*Set containing all Features of the Classifier itself and all its inherited Features (both directly and indirectly).*
Formal Definition	***Classifier:: allInheritedFeatures( ): Set(Feature)*** *= self.directlyInheritedFeatures( )* *-> union(self.parents( )* *-> collect(p | p.oclAsType(Classifier).allInheritedFeatures( ))* *-> flatten -> asSet)*
Comments	

DISCUSSION

What happens with inheritance if two or more parents have used the same name for different features?[25] For example, consider a class for vehicles that can be conducted only on the streets (*earth vehicles*), a class the ones that can be driven on the water (*aquatic vehicles*), and a class for *amphibious vehicles* (those that can be used both on the water and on the earth). The class of amphibious inherits both from the earth and water vehicles. However, both earth and water vehicles have an attribute called *name*. What happens with the class amphibious? Should it inherit both names?

In an approach emphasizing construction-box-like combination of modules from several sources, we may expect attempts to combine separately developed classes that contain identically named features.

For solving the name clashes problem, we adopt the solution presented on [Meyer, 1997]. The attributes are renamed in order to be correctly inherited. For example, it is possible to add a prefix or a suffix to the attribute that suffers from the conflict, as this can be the name of the class whose attribute belongs to. In the previous example, the names of the conflicting attributes could be replaced for *earthVehicle_name* and *aquaticVehicle_name*. This way, the objects in the amphibious class would have both names. This is enough (assuming there is no other clash) to remove the clash.

In our work, the name clashes problem is avoided automatically when converting the UML class diagram to a textual representation, as explained in section 6.1. The converter adds automatically a prefix (with the name of the owner class) to all the feature's names.

CONSEQUENCES

The implementation affects the functions that use inheritance (IAN, OAN, ION, AON, OAN, OON, PIAN, POAN, PION, PAON, POAN, POON, EILN, IILN and TILN), as well as the metrics AIF, OIF, IIF and EIF (from MOOD and MOOD2).

Name	***overriddenFeatures***
Informal Definition	*Set of redefined Features in the Classifier.*
Formal Definition	***Classifier:: overriddenFeatures( ): Set(Feature)*** *= definedFeatures( ) -> intersection (allInheritedFeatures( ))*
Comments	

[25] This problem is known in the literature as *name clash*.

Name	***allFeatures***
Informal Definition	*Set containing all Features of the Classifier itself and all its inherited Features.*
Formal Definition	***Classifier:: allFeatures( ): Set(Feature)*** *= self.feature -> union(self.parents( )* *-> collect(g | g.oclAsType(Classifier).allFeatures( ))* *-> flatten -> asSet)*
Comments	*Previously defined in the UML meta-model, but rewritten here.* *It can be alternatively defined as:* *= newFeatures( ) -> union(allInheritedFeatures( ))*

Name	***newAttributes***
Informal Definition	*Set of Attributes declared in the current Classifier.*
Formal Definition	***Classifier:: newAttributes( ): Set(Attribute)*** *= definedAttributes( ) - allInheritedAttributes( )*
Comments	*The definition excludes inherited Attributes (and consequently, it excludes overridden Attributes).*

Name	***definedAttributes***
Informal Definition	*Set of Attributes declared in the Classifier, including overridden Attributes.*
Formal Definition	***Classifier:: definedAttributes( ): Set(Attribute)*** *= feature2AttributeSet(self.definedFeatures( ))*
Comments	

Name	***directlyInheritedAttributes***
Informal Definition	*Set of directly inherited Attributes.*
Formal Definition	***Classifier:: directlyInheritedAttributes( ): Set(Attribute)*** *= feature2AttributeSet(self.directlyInheritedFeatures( ))*
Comments	

Name	***allInheritedAttributes***
Informal Definition	*Set of all inherited Attributes (both directly and indirectly).*
Formal Definition	***Classifier:: allInheritedAttributes( ): Set(Attribute)*** *= feature2AttributeSet(self.allInheritedFeatures( ))*
Comments	

Name	***overriddenAttributes***
Informal Definition	*Set of redefined Attributes in the Classifier.*
Formal Definition	***Classifier:: overriddenAttributes( ): Set(Attribute)*** *= definedAttributes( ) -> intersection (allInheritedAttributes( ))*
Comments	

Name	***allAttributes***
Informal Definition	*Set containing all Attributes of the Classifier itself and all its inherited Attributes (both directly and indirectly).*
Formal Definition	***Classifier:: allAttributes( ): Set(Attribute)*** *= feature2AttributeSet(self.allFeatures( ))*
Comments	*Previously defined in the UML meta-model, but redefined here. It can be alternatively defined as:* *= newAttributes( ) -> union(allInheritedAttributes( )) or* *= self.allFeatures( )* *-> select(f | f.oclIsKindOf(Attribute))* *-> collect(f | f.oclAsType(Attribute)) -> asSet*

Name	***newOperations***
Informal Definition	*Set of Operations declared in the current Classifier.*
Formal Definition	***Classifier:: newOperations( ): Set(Operation)*** *= definedOperations( ) - allInheritedOperations( )*
Comments	*The definition excludes inherited Operations (and consequently, it excludes overridden Operations).*

Name	***definedOperations***
Informal Definition	*Set of Operations declared in the Classifier, including overridden Operations.*
Formal Definition	***Classifier:: definedOperations( ): Set(Operation)*** *= feature2OperationSet(self.definedFeatures( ))*
Comments	

Name	***directlyInheritedOperations***
Informal Definition	*Set of directly inherited Operations.*
Formal Definition	***Classifier:: directlyInheritedOperations( ): Set(Operation)*** *= feature2OperationSet(self.directlyInheritedFeatures( ))*
Comments	

Name	***allInheritedOperations***
Informal Definition	*Set containing all Operations of the Classifier itself and all its inherited Operations (both directly and indirectly).*
Formal Definition	***Classifier:: allInheritedOperations( ): Set(Operation)*** *= feature2OperationSet(self.allInheritedFeatures( ))*
Comments	

Name	***overriddenOperations***
Informal Definition	*Set of redefined Operations in the Classifier.*
Formal Definition	***Classifier:: overriddenOperations( ): Set(Operation)*** *= definedOperations( ) -> intersection (allInheritedOperations( ))*
Comments	

Name	***allOperations***
Informal Definition	*Set containing all Operations of the Classifier itself and all its inherited Operations.*
Formal Definition	***Classifier:: allOperations( ): Set(Operation)*** *= feature2OperationSet(self.allFeatures( ))*
Comments	*Previously defined in the UML meta-model, but redefined here. It can be alternatively defined as:* *= newOperations( ) -> union(allInheritedOperations( )) or* *= self.allFeatures( )* *-> select(f | f.oclIsKindOf(Operation))* *-> collect(f | f.oclAsType(Operation)) -> asSet*

Name	***allContents***
Informal Definition	*Set containing all ModelElements contained in the Classifier together with the contents inherited from its parents.*
Formal Definition	***Classifier:: allContents( ): Set(ModelElement)*** *= self.contents( ) -> union(self.parents( )* *-> collect(g | g.oclAsType(Classifier).allContents( ))* *-> flatten -> asset)*
Comments	*Previously defined in the UML meta-model, but rewritten here.*

Name	***associations***
Informal Definition	*Set containing all Associations of the Classifier itself.*
Formal Definition	***Classifier:: associations( ): Set(Association)*** *= self.associationEnd* *-> collect(ae : AssociationEnd | ae.association) -> asSet*
Comments	*Previously defined in the UML meta-model.*

Name	***allOppositeAssociationEnds***
Informal Definition	*Set of all AssociationEnds, including the inherited ones, that are opposite to the Classifier.*
Formal Definition	***Classifier:: allOppositeAssociationEnds( ): Set(AssociationEnd)*** *= self.oppositeAssociationEnds( )* *-> union(self.parents( )* *-> collect(g |g.oclAsType(Classifier).allOppositeAssociationEnds( ))* *-> flatten -> asSet)*
Comments	*Previously defined in the UML meta-model, but rewritten here.*

Name	***oppositeAssociationEnds***
Informal Definition	*Set of all AssociationEnds that are opposite to the Classifier.*
Formal Definition	***Classifier:: oppositeAssociationEnds( ): Set(AssociationEnd)*** *= self.associations( ) -> select (a | a.connection* *-> select (ae | ae.type = self) -> size = 1)* *-> collect (a | a.connection -> select (ae | ae.type <> self))* *-> union (self.associations( ) -> select (a | a.connection* *-> select (ae |ae.type = self) -> size > 1)* *-> collect (a | a.connection)) -> flatten -> asSet*
Comments	*Previously defined in the UML meta-model.*

Classifier Counting Functions

Name	***NAN – New Attributes Number***
Informal Definition	*Number of new Attributes belonging to the Classifier.*
Formal Definition	***Classifier:: NAN( ): Integer*** *= newAttributes( ) -> size( )*
Comments	*This function was called AN (Attributes New) in the MOODLIB.*

Name	***DAN – Defined Attributes Number***
Informal Definition	*Number of defined Attributes in the Classifier.*
Formal Definition	***Classifier:: DAN( ): Integer*** *= definedAttributes( ) -> size( )*
Comments	*This function was called AD (Attributes Defined) in the MOODLIB.*

Name	***IAN – Inherited Attributes Number***
Informal Definition	*Number of inherited Attributes in the Classifier.*
Formal Definition	***Classifier:: IAN( ): Integer*** *= allInheritedAttributes( ) -> size( )*
Comments	*This function was called AI (Attributes Inherited) in the MOODLIB.*

Name	***OAN – Overridden Attributes Number***
Informal Definition	*Number of overridden Attributes in the Classifier.*
Formal Definition	***Classifier:: OAN( ): Integer*** *= overriddenAttributes( ) -> size( )*
Comments	*This function was called AO (Attributes Overridden) in the MOODLIB.*

Name	***AAN – Available Attributes Number***
Informal Definition	*Number of Attributes in the Classifier.*
Formal Definition	***Classifier:: AAN( ): Integer*** *= allAttributes( ) -> size( )*
Comments	*This function was called AA (Attributes Available) in the MOODLIB.*

Name	***NON – New Operations Number***
Informal Definition	*Number of new Operations in the Classifier.*
Formal Definition	***Classifier:: NON( ): Integer*** *= newOperations( ) -> size( )*
Comments	*This function was called ON (Operations New) in the MOODLIB.*

Name	***DON – Defined Operations Number***
Informal Definition	*Number of defined Operations in the Classifier.*
Formal Definition	***Classifier:: DON( ): Integer*** *= definedOperations( ) -> size( )*
Comments	*This function was called OD (Operations Defined) in the MOODLIB.*

Name	***ION – Inherited Operations Number***
Informal Definition	*Number of inherited Operations in the Classifier.*
Formal Definition	***Classifier:: ION( ): Integer*** *= allInheritedOperations( ) -> size( )*
Comments	*This function was called OI (Operations Inherited) in the MOODLIB.*

Name	***OON – Overridden Operations Number***
Informal Definition	*Number of overridden Operations in the Classifier.*
Formal Definition	***Classifier:: OON( ): Integer*** *= overriddenOperations( ) -> size( )*
Comments	*This function was called OO (Operations Overridden) in the MOODLIB.*

Name	***AON – Available Operations Number***
Informal Definition	*Number of Operations in the Classifier.*
Formal Definition	***Classifier:: AON( ): Integer*** *= allOperations( ) -> size( )*
Comments	*This function was called OA (Operations Available) in the MOODLIB.*

Name	***PRIAN – PRIvate Attributes Number***
Informal Definition	*Number of private Attributes in the Classifier.*
Formal Definition	***Classifier:: PRIAN( ): Integer*** *= self.allAttributes( ) -> select(a | a.visibility = #private) -> size( )*
Comments	

Name	***PROAN – PROtected Attributes Number***
Informal Definition	*Number of protected Attributes in the Classifier.*
Formal Definition	***Classifier:: PROAN( ): Integer*** *= self.allAttributes( ) -> select(a | a.visibility = #protected) -> size( )*
Comments	

Name	***PUBAN – PUBlic Attributes Number***
Informal Definition	*Number of public Attributes in the Classifier.*
Formal Definition	***Classifier:: PUBAN( ): Integer*** *= self.allAttributes( ) -> select(a | a.visibility = #public) -> size( )*
Comments	

Name	***PRION – PRIvate Operations Number***
Informal Definition	*Number of private Operations in the Classifier.*
Formal Definition	***Classifier:: PRION( ): Integer*** *= self.allOperations( ) -> select(o | o.visibility = #private) -> size( )*
Comments	

Name	***PROON – PROtected Operations Number***
Informal Definition	*Number of protected Operations in the Classifier.*
Formal Definition	***Classifier:: PROON( ): Integer*** *= self.allOperations( ) -> select(o | o.visibility = #protected) -> size( )*
Comments	

Name	***PUBON – PUBlic Operations Number***
Informal Definition	*Number of public Operations in the Classifier.*
Formal Definition	***Classifier:: PUBON( ): Integer*** *= self.allOperations( ) -> select(o | o.visibility = #public) -> size( )*
Comments	

Functions at Feature Context

Feature Counting Functions

Name	***FUN – Feature Use Number***
Informal Definition	*Number of ModelElements that use the Feature (excludes the ModelElement where the Feature is declared).*
Formal Definition	***Feature:: FUN( ): Integer*** *= self.allClients( ) -> size( )*
Comments	

Functions at GeneralizableElement Context

GeneralizableElement General Functions

Name	***isRoot***
Informal Definition	*Indicates whether the GeneralizableElement has ascendants or not. A true value indicates it has no ascendants and a false value indicates it has ascendants.*
Formal Definition	***GeneralizableElement:: isRoot( ): Boolean*** *= self.isRoot*
Comments	*As an alternative, the result could be: = parents( ) -> isEmpty( ) or* *= PARN( ) = 0*

Name	***isLeaf***
Informal Definition	*Indicates whether the GeneralizableElement has descendants or not. A true value indicates it has no descendants and a false value indicates it has descendants.*
Formal Definition	***GeneralizableElement:: isLeaf( ): Boolean*** *= self.isLeaf*
Comments	*As an alternative, the result could be: = children( ) -> isEmpty( ) or* *= CHIN( ) = 0*

GeneralizableElement Set Functions

Name	***children***
Informal Definition	*Set of directly derived Classes of the current GeneralizableElement.*
Formal Definition	***GeneralizableElement:: children( ): Set(GeneralizableElement)*** *= self.generalization -> collect(g | g.parent) -> excluding(self) -> asSet*
Comments	

Name	***descendants***
Informal Definition	*Set of all derived Classes (both directly and indirectly).*
Formal Definition	***GeneralizableElement:: descendants( ): Set(GeneralizableElement)*** *= children( )* *-> iterate(elem: GeneralizableElement; acc: Set(GeneralizableElement) = children( ) |* *acc -> union (elem.descendants( )))*
Comments	

Name	***parents***
Informal Definition	*Set of Classes from which the current GeneralizableElement derives directly.*
Formal Definition	***GeneralizableElement:: parents( ): Set(GeneralizableElement)*** *= self.specialization -> collect(c | c.child) -> asSet( ) -> excluding(self)*
Comments	

Name	***ascendants***
Informal Definition	*Set of all classes from which the current GeneralizableElement derives (both directly and indirectly).*
Formal Definition	***GeneralizableElement:: ascendants( ): Set(GeneralizableElement)*** *= parents( )* *-> iterate(elem: GeneralizableElement; acc: Set(GeneralizableElement) = parents( ) |* *acc -> union(elem.ascendants( )))*
Comments	

GeneralizableElement Counting Functions

Name	***CHIN – Children Number***
Informal Definition	*Number of directly derived Classes.*
Formal Definition	***GeneralizableElement:: CHIN( ): Integer*** *= children( ) -> size( )*
Comments	*This function was called CC (Children Count) in the MOODLIB.* *If CHIN( ) = 0 then the class is a leaf class.*

Name	***DESN – Descendants Number***
Informal Definition	*Number of all derived Classes (both directly and indirectly).*
Formal Definition	***GeneralizableElement:: DESN( ): Integer*** *= descendants( ) -> size( )*
Comments	*This function was called DC (Descendants Count) in the MOODLIB.*

Name	***PARN – Parents Number***
Informal Definition	*Number of Classes from which the current GeneralizableElement derives directly.*
Formal Definition	***GeneralizableElement:: PARN( ): Integer*** *= parents( ) -> size( )*
Comments	*This function was called PC (Parents Count) in the MOODLIB.* *If PARN( ) = 0 then the class is a base class; if PARN( ) > 1 multiple inheritance happens.*

Name	***ASCN – Ascendants Number***
Informal Definition	*Number of Classes from which the current GeneralizableElement derives (both directly and indirectly).*
Formal Definition	***GeneralizableElement:: ASCN( ): Integer*** *= ascendants( ) -> size( )*
Comments	*This function was called AC (Ascendants Count) in the MOODLIB.*

Functions at ModelElement Context

ModelElement Set Functions

Name	***client***
Informal Definition	*Set containing all direct clients of the ModelElement.*
Formal Definition	***ModelElement:: client( ): Set(ModelElement)*** *= self.supplierDependency -> collect(d : Dependency | d.client) -> flatten -> asSet*
Comments	

Name	***allClients***
Informal Definition	*Set containing all the ModelElements that are clients of this ModelElement, including the clients of these ModelElements. This is the transitive closure.*
Formal Definition	***ModelElement:: allClients( ): Set(ModelElement)*** *= self.client( ) -> union(self.client( ) -> collect(m : ModelElementImpl | m.allClients( )) -> flatten) -> asSet*
Comments	

Functions at Namespace Context

Namespace Set Functions

Name	***contents***
Informal Definition	*Set containing all ModelElements contained by the Namespace.*
Formal Definition	***Namespace:: contents( ): Set(ModelElement)*** *= self.ownedElement*
Comments	*This function belongs to the UML meta-model.*

Functions at Operation Context

Operation Counting Functions

Name	***OUN – Operation Use Number***
Informal Definition	*Number of ModelElements that use the Operation (excludes the ModelElement where the Feature is declared).*
Formal Definition	***Operation:: OUN( ): Integer*** *= self.FUN( )*
Comments	

Functions at Package Context

Package General Functions

Name	***isInternal***
Informal Definition	*Indicates if the Class received as parameter belongs to the considered Package.*
Formal Definition	***Package:: isInternal(c: Class): Boolean*** *= self.contents( ) -> includes(c)*
Comments	

Package Set Functions

Name	***allClasses***
Informal Definition	*Set of all Classes belonging to the current Package.*

Formal Definition	***Package:: allClasses( ): Set(Class)***
	= self.contents( )
	    -> iterate(elem: ModelElement; acc:Set(Class) = oclEmpty(Set (Class)) |
	        if elem.oclIsTypeOf(Class) then
	            acc -> union(acc -> including(elem.oclAsType(Class)))
	        else
	          acc
	        endif)

Comments	

Name	***internalBaseClasses***
Informal Definition	*Set of base Classes in the current Package, converted to Classifiers.*
Formal Definition	***Package:: internalBaseClasses( ): Set(Classifier)*** *= allClasses( )* *-> iterate(elem: Classifier; acc:Set(Classifier) = oclEmpty(Set(Classifier)) |* *acc -> union(elem.parents( ).oclAsType(Classifier)) -> asSet( ))*
Comments	

Name	***baseClasses***
Informal Definition	*Set of base Classes (from the current Package) that belong to the Package "p" , converted to Classifiers.*
Formal Definition	***Package:: baseClasses(p: Package): Set(Classifier)*** *= self.internalBaseClasses( )* *-> select(c: Classifier | p.isInternal (c.oclAsType(Class)))*
Comments	*A base Class is a Class that has at least one child.*

Name	***baseClassesInPackages***
Informal Definition	*Set of base Classes in both the current Package and the one bound to the parameter.*
Formal Definition	***Package:: baseClassesInPackages(p: Package): Set(Classifier)*** *= self.internalBaseClasses( ) -> union(p.internalBaseClasses( ))*
Comments	

Name	***internalSupplierClasses***
Informal Definition	*Set of supplier Classes in the current Package.*
Formal Definition	***Package:: internalSupplierClasses( ): Set(Classifier)*** *= supplierClasses(self)*
Comments	

Name	***supplierClasses***
Informal Definition	*Set of supplier Classes (from the current Package) that belong to the Package "p"(excludes inheritance).*
Formal Definition	***Package:: supplierClasses(p: Package): Set(Classifier)*** *= self. allClasses( )* *-> iterate(elem: Classifier; acc: Set(Classifier) = oclEmpty(Set(Classifier)) |* *acc -> union(elem.coupledClasses( )))* *-> select(c: Classifier | p.isInternal (c.oclAsType(Class)))*
Comments	

Name	***supplierClassesInPackages***
Informal Definition	*Set of supplier Classes in both the current Package and the one bound to the parameter.*
Formal Definition	***Package:: supplierClassesInPackages(p: Package): Set(Classifier)*** *= self.internalSupplierClasses( ) -> union(p.internalSupplierClasses( ))*
Comments	

Name	***relatedClasses***
Informal Definition	*Set of Classes from the "p" Package that are either base or supplier Classes.*
Formal Definition	***Package:: relatedClasses(p: Package): Set(Classifier)*** *= baseClasses(p) -> union(supplierClasses(p))*
Comments	

Package Counting Functions

Name	***CN – Classes Number***
Informal Definition	*Number of Classes in the Package.*
Formal Definition	***Package:: CN( ): Integer*** *= allClasses( ) -> size( )*
Comments	*This function was called TC (Total Classes) in the MOODLIB.*

Name	***PNAN – Package New Attributes Number***
Informal Definition	*Number of new Attributes in the Package.*
Formal Definition	***Package:: PNAN( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.NAN( ))*
Comments	*This function was called TAN (Total Attributes New) in the MOODLIB.*

Name	***PDAN – Package Defined Attributes Number***
Informal Definition	*Number of defined Attributes in the Package.*
Formal Definition	***Package:: PDAN( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.DAN( ))*
Comments	*This function was called TAD (Total Attributes Defined) in the MOODLIB.*

Name	***PIAN – Package Inherited Attributes Number***
Informal Definition	*Number of Attributes inherited in the Package.*
Formal Definition	***Package:: PIAN( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.IAN( ))*
Comments	*This function was called TAI (Total Attributes Inherited) in the MOODLIB.*

Name	***POAN – Package Overridden Attributes Number***
Informal Definition	*Number of overridden Attributes in the Package.*
Formal Definition	***Package:: POAN( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.OAN( ))*
Comments	*This function was called TAO (Total Attributes Overridden) in the MOODLIB.*

Name	***PAAN – Package Available Attributes Number***
Informal Definition	*Number of available Attributes in the Package.*
Formal Definition	***Package:: PAAN( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.AAN( ))*
Comments	*This function was called TAA (Total Attributes Available) in the MOODLIB.*

Name	***PNON – Package New Operations Number***
Informal Definition	*Number of new Operations in the Package.*
Formal Definition	***Package:: PNON( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.NON( ))*
Comments	*This function was called TON (Total Operations New) in the MOODLIB.*

Name	***PDON - Package Defined Operations Number***
Informal Definition	*Number of defined Operations in the Package.*
Formal Definition	***Package:: PDON( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |acc + elem.DON( ))*
Comments	*This function was called TOD (Total Operations Defined) in the MOODLIB.*

Name	***PION – Package Inherited Operations Number***
Informal Definition	*Number of inherited Operations in the Package.*
Formal Definition	***Package:: PION( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 | acc + elem.ION( ))*
Comments	*This function was called TOI (Total Operations Inherited) in the MOODLIB.*

Name	***POON – Package Overridden Operations Number***
Informal Definition	*Number of overridden Operations in the Package.*
Formal Definition	***Package:: POON( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |acc + elem.OON( ))*
Comments	*This function was called TOO (Total Operations Overridden) in the MOODLIB.*

Name	***PAON – Package Available Operations Number***
Informal Definition	*Number of available Operations in the Package.*
Formal Definition	***Package:: PAON( ): Integer*** *= allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |acc + elem.AON( ))*
Comments	*This function was called TOA (Total Operations Available) in the MOODLIB.*

Name	***EILN – External Inheritance Links Number***
Informal Definition	*Number of inheritance relations where the derived Classes belong to the current Package and the base ones belong to the Package “p” given as parameter.*
Formal Definition	***<u>Package:: EILN(p:Package): Integer</u>*** *= allClasses( ).parents( )* *-> select(g: GeneralizableElement | p.isInternal (g.oclAsType(Class))) -> size( )*
Comments	*This function was called IL (Inheritance Links) in the MOODLIB.* *EILN(p) <= IILN( )*

Name	***IILN – Internal Inheritance Links Number***
Informal Definition	*Number of inheritance relations where the base and derived Classes belong to the current Package.*
Formal Definition	***<u>Package:: IILN( ): Integer</u>*** *= allClasses( ).parents( )* *-> select(g: GeneralizableElement | self.isInternal(g.oclAsType(Class))) -> size( )*
Comments	*This function was called TIL (Total Inheritance Links) in the MOODLIB. It can be alternatively defined as* *= EILN(self)* *but this could lead to some name confusion in metrics definition (see IIF e EIF on chapter 6).*

Name	***PILN – Packages Inheritance Links Number***
Informal Definition	*Number of inheritance relations between the current package and the one received as parameter.*
Formal Definition	***<u>Package:: PILN(p: Package): Integer</u>*** *= self.IILN( ) + self.EILN(p)*
Comments	

Name	***ECLN – External Coupling Links Number***
Informal Definition	*Number of coupling relations where the client Class belongs to the current Package and the supplier Class belongs to the Package “p” (excludes inheritance).*
Formal Definition	***<u>Package:: ECLN(p: Package): Integer</u>*** *= self.supplierClassesInPackage(p) -> size( )*
Comments	*This function was called CL (Coupling Links) in the MOODLIB.*

Name	***ICLN – Internal Coupling Links Number***
Informal Definition	*Number of distinct coupling relations where both the client and the supplier Classes belong to the current Package (excludes inheritance).*
Formal Definition	***Package:: ICLN( ): Integer*** *= self.internalSupplierClasses( ) -> size( ))*
Comments	

Name	***PCLN – Packages Coupling Links Number***
Informal Definition	*Number of coupling relations between the current package and the one received as parameter.*
Formal Definition	***Package:: PCLN(p: Package): Integer*** *= self.ICLN( ) + self.ECLN(p)*
Comments	

Name	***AVN – Attribute Visibility Number***
Informal Definition	*Number of Classes in the considered Package where the Attribute can be accessed.*
Formal Definition	***Package:: AVN(a: Attribute): Integer*** *= self.FVN(a)*
Comments	*This function could be omitted if the MOOD2 set of metrics had the proposed polymorphic metric FHEF, instead of AHEF. It was preserved here to keep the original set of metrics.*

Name	***OVN – Operation Visibility Number***
Informal Definition	*Number of Classes in the considered Package where the Operation can be accessed.*
Formal Definition	***Package:: OVN(o: Operation): Integer*** *= self.FVN(o)*
Comments	*This function could be omitted if the MOOD2 set of metrics had the proposed polymorphic metric FHEF, instead of OHEF. It was preserved here to keep the original set of metrics.*

Name	***FVN – FeatureVisibility Number***
Informal Definition	*Number of Classes in the considered Package where the Feature can be accessed.*
Formal Definition	***Package:: FVN(f: Feature): Integer*** *= self.allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |* *if elem.FCV(f) then* *acc + 1* *else* *acc* *endif)*
Comments	

Package Percentage Functions

Name	***APV – Attribute to Package Visibility***
Informal Definition	*Percentage of Classes in the considered Package where the Attribute can be accessed (excludes the Classifier where the Attribute is declared).*
Formal Definition	***Package:: APV(a: Attribute): Percentage*** *= (self.AVN(a) – 1) / (self.CN( ) – 1)* *pre: self.CN( ) > 1*
Comments	*This function was called ASV (Attribute to Specification Visibility) in the MOODLIB.* *The pre-condition states that at least another class besides the one where the attribute is defined must exist.*

Name	***OPV – Operation to Package Visibility***
Informal Definition	*Percentage of Classes in the considered Package where the Operation can be accessed (excludes the class where the Operation is declared).*
Formal Definition	***Package:: OPV(o: Operation): Percentage*** *= (self.OVN (o) – 1) / (self.CN( ) – 1)* *pre: self.CN( ) > 1*
Comments	*This function was called OSV (Operation to Specification Visibility) in the MOODLIB.* *The pre-condition states that at least another class besides the one where the attribute is defined must exist. This would hardly be a true restriction.*

Name	***FPV – Feature to Package Visibility***
Informal Definition	*Percentage of Classes in the considered Package where the Feature can be accessed (excludes the Classifier where the attribute is declared).*
Formal Definition	***Package:: FPV(f: Feature): Percentage*** *= (self.FVN(f) -1) / (self.CN( ) -1)* *pre: self.CN( ) > 1*
Comments	

6

Metrics for Object-Oriented Design

SYNTHESIS

Design metrics are useful means for improving the quality of software. A number of object-oriented metrics have been suggested as being helpful for identifying fault-prone classes, for predicting required maintenance efforts, for assessing productivity and for estimating rework efforts [Tang and Chen, 2002].

To obtain the design metrics of the software under development, most existing approaches measure the metrics by parsing the source code of the software. Such approaches can only be performed in a late phase of software development, thus limiting the usefulness of the design metrics in the early phases of the development life cycle. Another problem is that such metrics are informally described, limiting the tool support.

In this chapter, we present a methodology that compiles UML specifications to obtain design information and allows computing the design metrics at an early stage of software development. Our approach eliminates the mentioned problems.

"Not everything that can be counted counts and not everything that counts can be counted."

Albert Einstein (1879-1955)

6.1 AN ARCHITECTURE FOR METRICS EXTRACTION

The current state of the art for adding precision to object-oriented modeling by the use of OCL is depicted in figure 6.1. Commercial UML modeling tools – as Rational Rose [Rational Software Corporation, 2001], Objectory [Rational Software Corporation, 1998], JDeveloper [Oracle Technology Network, 2002], QuickUML [Excel Software], PowerDesigner [Sybase Inc.], etc. – provide some graphic diagram editors that allow building models of systems. The models – represented by X, Y and Z in the picture – are stored in the tool repository.

Nowadays, modeling tools do not offer facilities for the evaluation of OCL expressions over the models in the repository. Nevertheless, several tools – like Use [University of Bremen], Cybernetic Parser [Cybernetic Intelligence], Elixer [Elixer Technology], ModelRun [BoldSoft], etc – are emerging from undergoing research projects and can be used to formalize the models, provided that they can be exported with an appropriate input format to the OCL tools. Typically, a textual file representing the model is generated by a translator (XML can be used as an example to represent the model). One example of translation, which can be understood by the USE tool is shown in Appendix D.

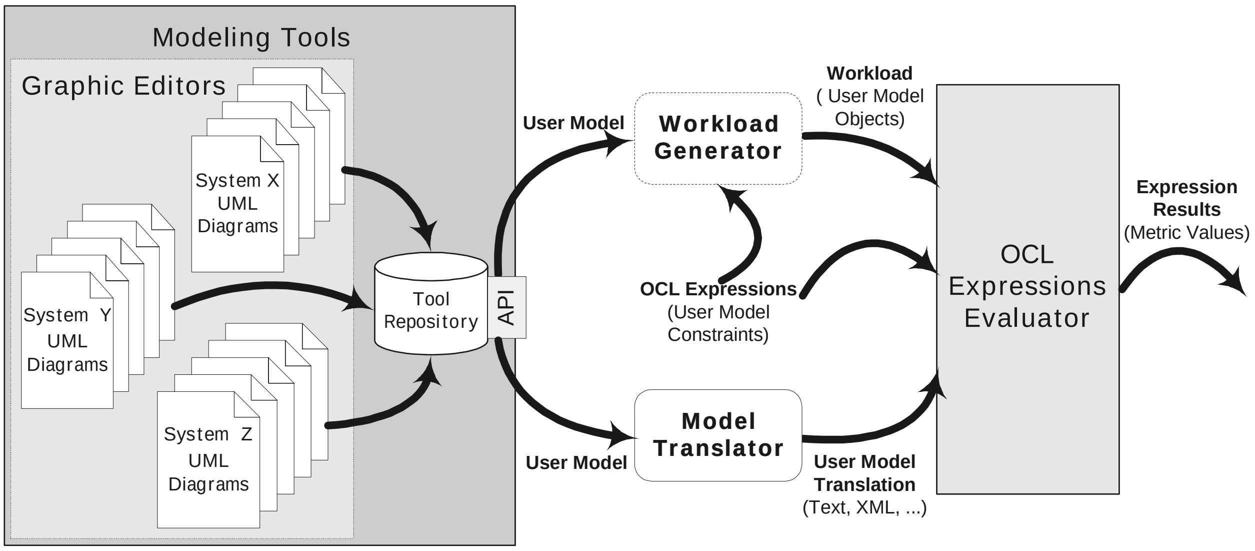
Modeling Tools
Graphic Editors
System X UML Diagrams
System Y UML Diagrams
System Z UML Diagrams
Tool Repository
API
User Model
Workload Generator
Workload (User Model Objects)
OCL Expressions (User Model Constraints)
User Model
Model Translator
User Model Translation (Text, XML, ...)
OCL Expressions Evaluator
Expression Results (Metric Values)

Figure 6.1 – *Model Level Architecture*

After the file conversion of the model (to a representation that can be understood by OCL tools), the real instances of the entities in the diagram are created and the model is populated (i.e., a plenty of objects, corresponding to the entities in the model, are created). These instances are the base of the assertions that are constructed with OCL. For this process one workload generator tool (see figure 6.1) would be of great help because, frequently, the UML model instances are done “by hand”.

The diagrams that compose the models and their respective objects serve as input to an OCL evaluation tool, which takes the converted representation of the diagram, the added OCL constraints and the instances of the model, and evaluate each of the constraints, showing the results. The OCL evaluator should be capable of verifying if the constraints are broken or not, for a given workload of user model instances. Moreover, it should evaluate each assertion separately and to provide feedback on which are the design test cases that meet or break the constraints.

While the architecture depicted in figure 6.1 corresponds to a model level evaluation, the one depicted in figure 6.2 is related to a meta-model level evaluation. In the meta-model level architecture, all the functionalities of the model-level are preserved. Notwithstanding, there are two main additions: one is the introduction of the class diagrams corresponding to the UML meta-model. Another is the introduction of an automatic instance generator[26], which will take the meta-model and automatically generate all the instances to populate it. Using these features (meta-model and corresponding instances) we formalize and test several design sets of metrics that can be found in the literature, expressed as OCL expressions upon the UML meta-model, as shown in the next section. Some examples of tests are presented in Appendix D.

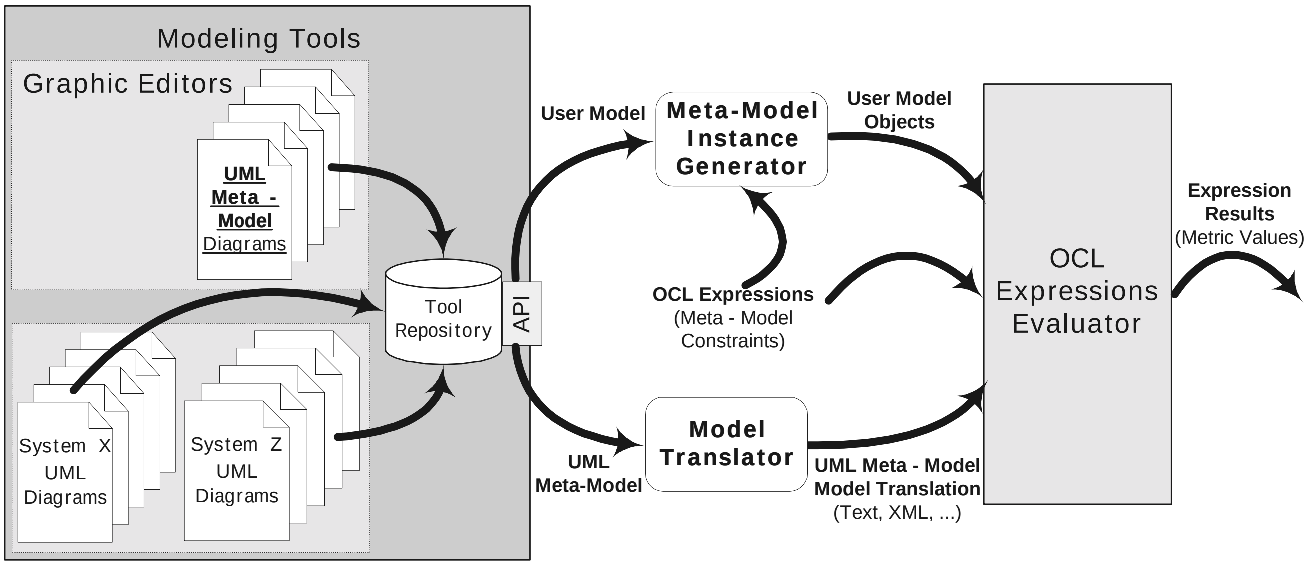
Modeling Tools
Graphic Editors
UML Meta - Model Diagrams
Tool Repository
API
System X UML Diagrams
System Z UML Diagrams
User Model
Meta-Model Instance Generator
User Model Objects
OCL Expressions (Meta - Model Constraints)
OCL Expressions Evaluator
Expression Results (Metric Values)
UML Meta-Model
Model Translator
UML Meta - Model Model Translation (Text, XML, ...)

Figure 6.2 – *Meta-Model Level Architecture*

A textual version (in XML format for example) of the UML meta-model can be obtained from a UML meta-model class diagram, using the architecture represented in figure 6.2. The members of the QUASAR group developed a meta-model instance generator to instantiate the objects corresponding to the meta-classes.

Our work helps improving quality in the models, since it helps developers (at an initial stage of the software production) to estimate important characteristics of the system while the quality factors are being evaluated and adapted for getting a better product.

6.2 METRICS FOR OBJECT-ORIENTED DESIGN

Most software developments face the risks of schedule slips and/or cost overruns. Effective resource allocation, reduction of design complexity, and adoption of effective software engineering techniques are thus the keys for resolving or reducing such risks. Design metrics, which are quantitative measures of the complexity of the

[26] The automatic instance generator was created by the QUASAR team and it can be download at http://ctp.di.fct.unl.pt/QUASAR

software or design, have been suggested as useful means to assist in achieving these goals [Tang and Chen, 2002].

A number of design metrics have been studied and demonstrated as useful in several aspects, such as in understanding, assessing and evaluating the complexity of designs [Henry and Selig, 1990], estimating the complexity of software based on its design [Harrison, 1988], identifying fault-prone software units and plausible types of faults [Basili et al., 1996; Tang et al., 1999b], and estimating required maintenance efforts [Li and Henry, 1993; Rombach, 1990; Abreu and Melo, 1996].

Nevertheless, despite research studies, design metrics have not been widely utilized, as expected, in the software industry. One of the major problems that has limited their use is the lack of available tools to measure the metrics, which in turn can be a consequence of their imprecise specification.

This chapter acquaint with both the informal and formal definitions of four well-known sets of design metrics, namely the MOOD and MOOD2 – Metrics for Object-Oriented Design [Abreu, 1993; Abreu, 1998], MOOSE – Metrics for Object-Oriented Software Engineering [Chidamber and Kemerer, 1993a; Chidamber and Kemerer, 1993b], EMOOSE – Extended MOOSE [Li et al., 1995] and QMOOD – Quality Model for Object-Oriented Design [Bansiya and Davis, 1997a; Bansiya and Davis, 1997b] – metric sets.

Most of the metrics on these sets were originally defined informally, using natural language, and the major contribution of this work remains on bringing up precision, through their formal definitions. As in the previous chapter, the formalization is done with OCL (explained in chapter 3) and the UML meta-model (introduced in chapter 4). Moreover, the library of measures FLAME (presented in chapter 5) serves as input for the metrics formalization. All the metrics were tested with the architecture described on section 6.1, and with real examples of the QUASAR laboratory.

Instead of using the meta-class Class context when necessarily mentioned on the informal definitions of the metrics, the Classifier context was adopted. Such a change makes the metrics formalization more flexible because the definitions can be applied in the subclasses of Classifier (including mainly the meta-classes Class and DataTypes). For instance, consider a case when the user wants to define a new structure of data types in his model, including classes as Date and Time, or a Percentage class as a subtype of the Real class. With the formalization upon the Classifier context, the user's new Data Types structure can be estimated. This would not be possible if the definitions were restricted to the context of Class.

In some cases, the metric formalization is not possible. When this occurs, the area corresponding to the field "*Formal Definition*" is left only with the signature of the metric. For each case, the reasons that forbid the formalization are explained. Furthermore, some suggestions or problems are described in the section "Comments".

6.2.1 MOOD and MOOD2 Metrics

The MOOD metric set (Metrics for Object-Oriented Design) was first introduced in [Abreu, 1993]. Its use and validation was presented in several occasions such as in [Abreu, 1995b; Abreu and Melo, 1996; Harrison et al., 1998].

After some experiments, it became evident that some important aspects of the design were not being measured by the MOOD metrics, namely the existence of polymorphism and the amount of reuse. In addition, the MOOD set only considered metrics calculated within a given specification and many executable systems (applications) are usually composed upon several specifications[27].

These led to the birth of the MOOD2 set, whose metrics were divided on two groups: intra-specification metrics and inter-specification metrics. The first group includes those metrics that refer to the context specification only and whose definition relies upon information contained solely on it. Therefore, they are parameter less. The second group includes those metrics that the definition relates to the relationship between the context specification and the one that is passed as an argument. In this way, some metrics (inheritance and coupling ones) reflect the internal (within the specification) design aspects, while others reflect the external (among distinct specifications) ones.

Table 6.1 and 6.2 present the metric sets MOOD and MOOD2. The new metrics (on MOOD2) are marked with a star. A few of the original MOOD metrics were renamed for naming consistency.

Acronym	Designation	
AIF	Attribute Inheritance Factor	
OIF	Operations Inheritance Factor [28]	
IIF	Internal Inheritance Factor	*
AHF	Attribute Hiding Factor	
OHF	Operations Hiding Factor [29]	
AHEF	Attributes Hiding Effectiveness Factor	*
OHEF	Operations Hiding Effectiveness Factor	*
BPF	Behavioral Polymorphism Factor[30]	
PPF	Parametric Polymorphism Factor	*
CCF	Class Coupling Factor [31]	
ICF	Internal Coupling Factor	

Table 6.1 – *Intra-Specification Metrics*

Acronym	Designation	
EIF (S)	External Inheritance Factor	*
ECF (S)	External Coupling Factor	*
PRF(S)	Potential Reuse Factor	*
ARF(S)	Actual Reuse Factor	*
REF(S)	Reuse Efficiency Factor	*

Table 6.2 – *Inter-Specification Metrics*

[27] Specification, in this context, stands for a description of a system. It is equivalent to the concept of *Package* in UML.
[28] Originally called MIF - Methods Inheritance Factor.
[29] Originally called MHF - Methods Hiding Factor.
[30] Originally called POF – POlymorphism Factor.
[31] Originally called COF – COupling Factor.

The MOOD2 metrics retain the main characteristics of the original set. All of them are defined as quotients where the numerator represents the actual value of the design characteristic being measured, while the denominator represents its theoretical maximum value. As a result, they take values in a percentual scale (real numbers in the interval [0, 1]).

Another improvement in the MOOD2 set is that their definition was made on a compositional way, based upon a set of auxiliary functions (the MOODlib, described on Appendix C), at different levels of abstraction, namely *Attribute*, *Operation*, *Class* and *Specification*. Each of these levels corresponds to one metaclass in the GOODLY meta-model [Abreu et al., 1997; Abreu et al., 1999; Abreu et al., 2001]. Appendix B reproduces the GOODLY meta-model and the original functions created to define the MOOD2 metrics.

Instead of using the GOODLY meta-model and the MOODlib, this document employs the UML meta-model and the FLAME library. As the UML is becoming a well-known standard, these replacements will improve tool support and the extraction of the metrics.

Intra-Specification Level Metrics

Name	***AIF – Attributes Inheritance Factor***
Informal Definition	*Quotient between the number of inherited Attributes in all Classes of the Package and the number of available Attributes (locally defined plus inherited) for all Classes of the current Package.*
Formal Definition	***Package:: AIF( ): Percentage*** *= self.PIAN( ) / self.PAAN( )* *pre: self.PAAN( ) > 0*
Comments	*The pre-condition states that the package must have available Attributes.* *AIF( ) = 0 means that there is no effective Attribute inheritance (either there are no inheritance hierarchies or all inherited Attributes are redefined).*

Name	***OIF – Operations Inheritance Factor***
Informal Definition	*Quotient between the number of inherited Operations in all Classes of the Package and the number of available Operations (locally defined plus inherited) for all Classes of the current Package.*
Formal Definition	***Package:: OIF( ): Percentage*** *= self.PION( ) / self.PAON( )* *pre: self.PAON( ) > 0*
Comments	*The pre-condition states that the package must have available Operations.* *OIF( ) = 0 means that there is no effective Operation inheritance (either there are no inheritance hierarchies or all inherited Operations are redefined).* *It is possible, in a future work, to try to define a polymorphic function FIF – Features Inheritance Factor – instead of using AIF and OIF.*

Name	***IIF – Internal Inheritance Factor***
Informal Definition	*Quotient between the number of inheritance links where both the base and derived Classes belong to the current Package and the total number of inheritance links originating in the current Package.*
Formal Definition	***<u>Package:: IIF( ): Percentage</u>*** *= self.IILN( ) / self.PILN(self)* *pre: self.IILN( ) > 0*
Comments	*The pre-condition states that the package must have some inheritance links defined on it.* *Inheritance links originating in the current package are those where the derived Class belongs to it. The inheritance link is directed from the derived Class to the base one.*

Name	***AHF – Attributes Hiding Factor***
Informal Definition	*Quotient between the sum of the invisibilities of all Attributes defined in all Classes in the current Package and the total number of Attributes defined in the Package.*
Formal Definition	***<u>Package:: AHF( ): Percentage</u>*** *= allClasses( ).allAttributes( ) -> asSet( )* *-> iterate(elem:Attribute; acc:Real = 0 | acc + 1 - self.APV(elem)) /* *self.PDAN( )* *pre: self.CN( ) > 1* *pre: self.PDAN( ) > 0*
Comments	*The invisibility of an Attribute is the percentage of the Classes in the package from which this Attributes is not visible and is given by 1-APV(self), where self is the current Package.* *If all Attributes are private, the numerator is 0 and as such, AHF( ) = 0. If all Attributes are public, the numerator is 0 and as such, AHF( ) = 1.* *The pre-condition regarding the number of Classes is required for calculating the Attributes visibility (the package must have some Classes).The second pre-condition means that Attributes are necessary for calculating the metric.*

Name	***OHF – Operations Hiding Factor***
Informal Definition	*Quotient between the sum of the invisibilities of all Operations defined in all Classes in the current Package and the total number of Operations defined in the Package.*
Formal Definition	***<u>Package:: OHF( ): Percentage</u>*** *= allClasses( ).allOperations( ) -> asSet( )* *-> iterate(elem: Operation; acc: Real = 0 | acc + 1 - self.OPV(elem)) /* *self.PDON( )* *pre: self.CN( ) > 1* *pre: self.PDON( ) > 0*

Comments	*The invisibility of an Operation is the percentage of the total Classes in the package from which this Operation is not visible and is given by 1-OPV(self), where self is the current package.* *If all Operations are public the numerator equals the denominator and then OHF( ) = 1. Otherwise, if all Operations are private, OHF( ) = 0.* *The pre-condition regarding the number of Classes is required for calculating the Operations visibility (the package must have some Classes).The second pre-condition means that Operations are necessary for calculating the metric.* *It is possible, in a future work, to try to define a polymorphic function FHF – Features Hiding Factor – instead of using AHF and OHF.*

Name	***AHEF – Attributes Hiding Effectiveness Factor***
Informal Definition	*Quotient between the cumulative number of the Package Classes that do access the Package Attributes and the cumulative number of the Package Classes that can access the Package Attributes.*
Formal Definition	***Package:: AHEF( ): Percentage*** *= allClasses( ).allAttributes( ) -> asSet( )* *-> iterate(elem: Attribute; acc: Integer = 0 | acc + elem.AUN( ))* */ allClasses( ).allAttributes( ) -> asSet( )* *-> iterate(elem: Attribute; acc: Integer = 0 | acc + self.AVN(elem))* *pre: allClasses( ).allAttributes( ) ->* *iterate(elem: Attribute; acc: Integer = 0 | acc + elem.AVN(self)) > 0*
Comments	

Name	***OHEF – Operations Hiding Effectiveness Factor***
Informal Definition	*Quotient between the cumulative number of the Package Classes that do access the Package Operations and the cumulative number of the Package Classes that can access the Package Operations.*
Formal Definition	***Package:: OHEF( ): Percentage*** *= allClasses( ).allOperations( ) -> asSet( )* *-> iterate(elem: Operation; acc: Integer = 0 | acc + elem.OUN( ))* */ allClasses( ).allOperations( ) -> asSet( )* *-> iterate(elem: Operation; acc: Integer = 0 | acc + self.OVN(elem))* *pre: allClasses( ).allOperations( ) ->* *iterate(elem: Operation; acc: Integer = 0 | acc + elem.OVN(self)) > 0*
Comments	*It is possible, in a future work, to try to define a polymorphic function FHEF – Features Hiding Effectiveness Factor – instead of using AHEF and OHEF.*

Name	**BPF – Behavioral Polymorphism Factor**
Informal Definition	*Quotient between the actual number of possible different polymorphic situations within the current Package and the maximum number of possible distinct polymorphic situations (due to inheritance).*
Formal Definition	***Package:: BPF( ): Percentage*** *= self.POON( ) / self.PAON( )* *pre: PAON( ) > 0*
Comments	*An Operation in a Class C_i can have as many shapes ("morphos" in ancient Greek) as the number of times it is overridden (in C_i descendants). This represents the actual number of, possible different, polymorphic situations for that Class (For this metric, the overriding of Operations is considered only when defined in the current package).* *The maximum number of possible distinct polymorphic situations for Class C_i occurs if all new Operations defined in it are overridden in all of their derived Classes.*

Name	**PPF – Parametric Polymorphism Factor**
Informal Definition	*Percentage of the Package Classes that are parameterized.*
Formal Definition	***Package:: PPF( ): Percentage*** *= allClasses( ) -> select(templateParameter -> notEmpty( )) -> size( ) / CN( )* *pre: self.CN( ) > 0*
Comments	

Name	**CCF – Class Coupling Factor**
Informal Definition	*Quotient between the actual number of coupled class-pairs within the Package and the maximum possible number of class-pair couplings in the Package. This coupling is the one not imputable to inheritance.*
Formal Definition	***Package:: CCF( ): Percentage*** *= sqrt(self.ICLN( ) / ((self.CN( ) * self.CN( )) - self.CN( )))* *pre: self.CN( ) > 1*
Comments	*In a coupled class-pair one Class is the client and the other is the supplier. These client-supplier relations can have several shapes; see the function Classifier::CoupledClasses for details.* *The pre-condition states that, with only one Class, there are no couplings within the package.* *The square root counteracts for the fact that the couplings grow quadratically with the number of Classes.*

Name	***ICF – Internal Coupling Factor***
Informal Definition	*Quotient between the number of coupling links where both the client and supplier Classes belong to the current Package and the total number of coupling links originating in the current Package.*
Formal Definition	***Package:: ICF( ): Percentage*** *= self.ICLN(self) / self.PCLN(self)* *pre: self.ICLN( ) > 0*
Comments	*Coupling links originating in the current Package are those where the client Class belongs to it. The coupling link is directed from the client Class to the supplier one.*

Inter-Specification Level Metrics

Name	***EIF – External Inheritance Factor***
Informal Definition	*Quotient between the number of external inheritance links to Package "p" and the total number of inheritance links originating in the current Package.*
Formal Definition	***Package:: EIF(p: Package): Percentage*** *= self.EILN(p) / self.PILN(self)* *pre: self.importedElement -> includes(p)* *pre: self.IILN( ) > 0*
Comments	*External inheritance links are those originating in the current Package, but where the base Class lies outside of it. By other words, they correspond to local derivations of external Classes (defined in external package "p").*

Name	***ECF – External Coupling Factor***
Informal Definition	*Quotient between the number of external coupling links to Package "p" and the total number of coupling links originating in the current Package.*
Formal Definition	***Package:: ECF(p: Package): Percentage*** *= self.ECLN(p) / self.PCLN(self)* *pre: self.importedElement -> includes(p)* *pre: self.ICLN( ) > 0*
Comments	*External coupling links are those originating in the current package, but where the supplier Class is defined outside of it (in external package "p").*

Name	***PRF – Potential Reuse Factor***
Informal Definition	*Percentage of the available Operations in the current Package that were imported from the "p" Package.*
Formal Definition	***<u>Package:: PRF(p: Package): Percentage</u>*** *= relatedClasses(p).allOperations( ) -> asSet( ) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem))* */ (allClasses( ).allOperations( )* *-> union (relatedClasses(p).allOperations( ))) -> asSet( ) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem))* *pre: self.importedElement -> includes(p)* *pre: (allClasses( ).allOperations( ) union* *relatedClasses(s).allOperations( )) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem)) > 0*
Comments	*The Operations imported from the external Package "p" correspond to those inherited from the Classes from which the current Package Classes derive, plus the ones from "p" which are coupled to internal Classes.*

Name	***ARF – Actual Reuse Factor***
Informal Definition	*Percentage of the available Operations in the current Package that corresponds to effectively used Operations imported from the "p" Package.*
Formal Definition	***<u>Package:: ARF(p: Package): Percentage</u>*** *= relatedClasses(p).allOperations( )* *-> select(o: Operation | o.FUN( ) > 0) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem))* */ (allClasses( ).allOperations( )* *-> union (relatedClasses(p).allOperations( ))) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem))* *pre: self.importedElement -> includes(p)* *pre: (allClasses( ).allOperations( ) union* *relatedClasses(s).allOperations( )) ->* *iterate(elem: Operation; acc: Real = 0 | acc + self.FPV(elem)) > 0*
Comments	

Name	***REF – Reuse Efficiency Factor***
Informal Definition	*Percentage of the imported Operations (from the "p" Package) that are effectively used.*
Formal Definition	***Package:: REF(p: Package): Percentage*** *= self.ARF(p) / self.PRF(p)* *pre: self.importedElement -> includes(p)* *pre: self.PRF( ) > 0*
Comments	

It is possible to see that all the metrics in MOOD and MOOD 2 were successfully formalized. Some results may be different than when applied considering the source code together with the design, as it can happen with the metrics that use the function *coupledClasses*.

As one improvement, we suggest a review of the set, especially of the metrics that are similar for Attributes and Operations, in order to try to make them polimorphic and more generic. Such metrics could deal with Features instead of Attributes and Operations.

6.2.2 MOOSE Metrics

One of the most referenced metric suites is the MOOSE set, proposed by Chidamber and Kemerer [Chidamber and Kemerer, 1993b].

Although this set is widespread, not all the metrics are design ones, and some of them cannot be formalized upon the UML Model. Anyway, several studies have been conducted to validate the MOOSE metrics and have shown that they are useful quality indicators for predicting fault-prone classes [Basili et al., 1996; Tang et al., 1999a] and maintenance effort [Li and Henry, 1993], as well as being significant economic variable indicators [Chidamber et al., 1998].

For the complete description of the metrics, their corresponding usefulness and evaluation, refer to [Chidamber and Kemerer, 1991; Chidamber and Kemerer, 1993a].

Name	***WMC – Weighted Methods per Class***
Informal Definition	*The sum of complexities of the Methods in the current Class. If all method complexities are considered to be unique, WMC is equal to the number of Methods.*
Formal Definition	***Classifier:: WMC( ): Integer*** *= self.allOperations( ) -> size( )*
Comments	*The authors do not propose any algorithm for calculating the complexities of methods. As such, in the formalization above, the complexities were considered unitary.*

Name	***DIT – Depth of Inheritance Tree***
Informal Definition	*The length of the longest path of inheritance from the current Class to the root of the tree.*
Formal Definition	***Classifier:: DIT( ): Integer*** *= if self.isRoot( ) then 0 else if PARN( ) = 1 then 1 + self.parents( ) -> iterate(elem: GeneralizableElement; acc: Integer = 0 | acc + elem.oclAsType(Class).DIT( )) else self.parents( ) -> iterate(elem: GeneralizableElement; acc: Integer = 0 | acc + elem.oclAsType(Class).DIT( )) endif endif*
Comments	

Name	***NOC – Number of Children***
Informal Definition	*The number of classes that inherit directly from the current Class.*
Formal Definition	***Classifier:: NOC( ): Integer*** *= self.CHIN( )*
Comments	

Name	***CBO – Coupling Between Objects***
Informal Definition	*The number of other Classes that are coupled to the current one. Two Classes are coupled when methods declared in one Class use Methods or instance variables defined by the other Class.*
Formal Definition	***Classifier:: CBO( ): Integer*** *= self.coupledClasses( ) -> size( )*
Comments	

Name	***RFC – Response for a Class***
Informal Definition	*The number of Methods in the current Class that might respond to a message received by its object, including Methods both inside and outside of this Class.*
Formal Definition	***Classifier:: RFC( ): Integer*** *= (self.allOperations( ) -> union(self.allOperations( ).method.allClients( ).oclAsType(Operation))) -> asSet( ) -> size( )*
Comments	$RFC = \{M\} \cup_{all\ i} \{R_i\}$ *where* $\{R_i\}$ *= set of Methods called by Method i and* $\{M\}$ *= set of all Methods in a Class. Ri is dependent on the implementation of the Method i.*

Name	***LCOM – Lack of Cohesion in Methods***
Informal Definition	*The degree of similarity of Methods in the current Class (by counting instance variables sets used by all possible Method pairs).*
Formal Definition	***Classifier:: LCOM( ): Integer*** *=*
Comments	*The set of instance variables used by one Method is known only after completing the implementation of the Method. So this metric is code dependent, and can not be extracted in the design phase.*

The MOOSE set of metrics is not completely related with design. This restricts the formalization. For instance, the metric LCOM depends on the source code. Moreover, it is necessary to note one limitation considering this set. The metric WMC has, in this document, provides only the simplest implementation regarding complexities (it considers all the methods' complexities as unitary). However, different implementations could be offered. As the authors of the set do not define the algorithms for calculating the complexities, we considered the simplest case.

6.2.3 EMOOSE Metrics

This set was conceived as an extension of the MOOSE metrics. The EMOOSE (Extended MOOSE) metrics were created by Wei Li, Sallie Henry et. al. [Li et al., 1995].

The EMOOSE metrics contains the ones defined in the MOOSE set, plus the ones illustrated below. The set has a restrict granularity because the metrics are applied only to the Class (Classifier) context.

Name	***MPC – Message Pass Coupling***
Informal Definition	*Number of messages sent by the Class' Operations. It is based on the calls presented in the implementation of all Operations.*
Formal Definition	***Classifier:: MPC( ): Integer*** *=*
Comments	*As LCOM above, this metric is code dependent.*

Name	***DAC – Data Abstraction Coupling***
Informal Definition	*Number of Classes aggregated to the current Class.*
Formal Definition	***Classifier:: DAC( ): Integer*** *= self.allAttributes( ).type -> size( )* *pre: self.namespaceImpl.oclAsType(Package).CN( ) > 1* *pre: self.namespaceImpl.oclAsType(Package).ICLN( ) > 0*
Comments	*The word "aggregated" is different from the word "coupled". The first considered only the links throughout the Class Attributes. The latter includes not only the Attributes, but also Methods, and Parameters (see the function coupledClasses in FLAME).* *The pre-conditions state that the system must have some Classes with coupling among them.*

Name	***NOM – Number of Methods***
Informal Definition	*Number of Operations that are local to the Class, i.e., that can only be accessed by other Class Operations (and not in other Classes).*
Formal Definition	***<u>Classifier:: NOM( ): Integer</u>*** *= self.allOperations( ) -> select(o: Operation | o.visibility = #private) -> size( )* *pre: self.ON( ) > 0*
Comments	*The Class must have some Operations.*

Name	***SIZE 1***
Informal Definition	*Number of lines of codes.*
Formal Definition	***<u>Classifier:: SIZE1( ): Integer</u>*** *=*
Comments	*The number of lines (LOC) is a measure that can be extracted only when the source code is available.*

Name	***SIZE 2***
Informal Definition	*Number of local Attributes and Operations defined in the Class.*
Formal Definition	***<u>Classifier:: SIZE2( ): Integer</u>*** *= self.DON( ) + self.DAN( )*
Comments	

As the MOOSE set, the EMOOSE is not purely related with design. This implies that some metrics can not be formalized using our approach (OCL upon the UML meta-model), as it happens with MPC and SIZE 1.

Note that the function NOM will produce the same result than WMC in the MOOSE group. This means that the function NOM in the extended group (EMOOSE) makes sense when the WMC calculates the complexity of the methods in a different way, otherwise the metric results will be duplicated.

6.2.4 QMOOD Metrics

QMOOD is a quality model for assessing high-level external quality attributes such as reusability, functionality and flexibility of object-oriented designs based on the internal properties of C++ design components.

It defines a set of metrics that can be applied both on the contexts of system or classes (Package or Classifier, when using the UML meta-model). The complete description of QMOOD can be found in Bansyia's PhD thesis.

We consider that this set of metrics has several problems, mainly because it is language dependant and because some definitions are the same, even having different names. Additionally these metrics present a great disparity in the scales used in the results, mixing Integers, Reals and Percentage values.

Finally, their definition in natural language can give different interpretations to the results, which made our work more difficult. When this happens, we present the alternative interpretations in the "*Comments*" field. It is still necessary to discuss which of the interpretations is the right one.

System Measures

Name	***DSC – Design Size in Classes***
Informal Definition	*Count of the total number of Classes in the design.*
Formal Definition	***Package:: DSC( ): Integer*** *= self.CN( )*
Comments	

Name	***NOH – Number of Hierarchies***
Informal Definition	*Count of the number of Class hierarchies in the design.*
Formal Definition	***Package:: NOH( ): Integer*** *= self.allClasses( ).children( ) -> size( )*
Comments	*In [Abreu et al., 2000], this metric has another interpretation. There, hierarchies are not the number of inheritance relations but the number of inheritance trees. In this case, this metric is always equal to 1 for systems developed in languages that have a common super class (like Object in Smalltalk and Java). This case shows that the metric is centered in the language, which is considered as a potential problem (Remember QMOOD metrics were created based upon C++).*

Name	***NIC – Number of Independent Classes***
Informal Definition	*Count of the number of Classes that are not inherited by any Class in the design.*
Formal Definition	***Package:: NIC( ): Integer*** *= self.allClasses( ) -> select(isLeaf) -> size( )*
Comments	

Name	***NSI – Number of Single Inheritance***
Informal Definition	*Number of Classes (sub classes) that use inheritance in the design.*
Formal Definition	***Package:: NSI( ): Integer*** *= self.allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |* *if elem.PARN( ) = 1 then* *acc + 1* *else*

	acc *endif)*
Comments	*According to [Abreu et al., 2000], for the same reason than in NOH, this metric would be always equal to DSC (if the Classes belonging to the development environment were considered) or to DSC – 1 (in the opposite case). This is another signal of language dependency.*

Name	***NMI – Number of Multiple Inheritance***
Informal Definition	*Count of the number of instances of multiple inheritance in the design.*
Formal Definition	***<u>Package:: NMI( ): Integer</u>*** *= self.allClasses( ) -> iterate(elem: Class; acc: Integer = 0 |* *if elem.PARN() > 1 then* *acc + 1* *else* *acc* *endif)*
Comments	*[Abreu et al., 2000] say that for languages as Smalltalk or Delphi, which do not support multiple inheritance, this metric is always equal to zero. This interpretation is a clear signal of ill definition of the metric.*

Name	***NNC – Number of Internal Classes***
Informal Definition	*Count of the number of internal Classes defined for creating generalization-specialization structures in Class hierarchies of the design.*
Formal Definition	***<u>Package:: NNC( ): Integer</u>*** *= NOH( )*
Comments	*In this case, why to have the same value than NOH? Has this metric another interpretation? [Abreu et al., 2000] say that this metric is always equal to DSC or DSC - 1, by the reasons pointed out in NOH and NSI.*

Name	***NAC – Number of Abstract Classes***
Informal Definition	*Count of the number of Classes that have been defined purely for organizing information in the design.*
Formal Definition	***<u>Package:: NAC( ): Integer</u>*** *= self.allClasses( ) -> select(isAbstract) -> size( )*
Comments	

Name	***NLC – Number of Leaf Classes***
Informal Definition	*Count of the number of leaf Classes in the hierarchies of the design.*
Formal Definition	***Package:: NLC( ): Integer*** *= self.allClasses( ) -> select(isLeaf) -> size( )*
Comments	*This metric is the same than NIC, defined above.*

Name	***ADI – Average Depth of Inheritance***
Informal Definition	*The average depth of inheritance of Classes in the design. It is computed by dividing the summation of maximum path lengths to all Classes by the number of Classes. The path length for a Class is the number of edges from the root to the Class in an inheritance tree representation.*
Formal Definition	***Package:: ADI( ): Real*** *= self.allClasses( ) -> iterate(elem: Class; acc: Real = 0 |* *(acc + elem.DOI( )) / CN( ))*
Comments	

Name	***AWI – Average Width of Inheritance***
Informal Definition	*The average number of children per Class in the design. The metric is computed by dividing the summation of the number of children over all Classes by the number of Classes in the design.*
Formal Definition	***Package:: AWI( ): Real*** *= self.allClasses( ) -> iterate(elem: Class; acc: Real = 0 |* *(acc + elem.CHIN( )) / CN( ))*
Comments	*Accordingly to a different view point, [Abreu et al., 2000] say this metric is always equal to DSC or DSC - 1, when multiple inheritance is not supported by the language, as in Smalltalk, Eiffel or Java. The same happens with NOH, NSI and NNC.*

Name	***ANA – Average Number of Ancestors***
Informal Definition	*The average number of Classes from which a class inherits information.*
Formal Definition	***Package:: ANA( ): Real*** *= self.internalBaseClasses( ) -> size( ) / CN( )*
Comments	*This metric is similar to the ADI measure and differs only when there are instances of multiple inheritance in the design.*

Class Measures

Name	**MFM – Measure of Functional Modularity**
Informal Definition	Computes modularity based on the deviation of the number of Methods in a Class from the average number of Methods per Class in the design.
Formal Definition	**Classifier:: MFM( ): Integer** = (self.allOperations( ) -> size( ) – (self.namespaceImpl.oclAsType(Package).PAON( ) / self.namespaceImpl.oclAsType(Package).CN( ))) / (self.namespaceImpl.oclAsType(Package).PAON( ) / self.namespaceImpl.oclAsType(Package).CN( ))
Comments	A value closer than zero is preferred for this metric. A lower value indicates a smaller deviation among Classes in the number of services provided.

Name	**MFA – Measure of Functional Abstraction**
Informal Definition	The ratio of the number of Methods inherited by a class to the total number of Methods accessible by members in the Class.
Formal Definition	**Classifier:: MFA( ): Real** = self.ION( ) / self.allOperations( ) -> iterate(elem: Operation; acc:Integer = 0 | if self.FCV(elem) then acc + 1 else acc endif) pre: self.AON( ) > 0
Comments	In order to calculate MFA, the number of Operations must be greater than zero.

Name	**MAA – Measure of Attribute Abstraction**
Informal Definition	The ratio of the number of Attributes inherited by a Class to the total number of Attributes in the Class.
Formal Definition	**Classifier:: MAA( ): Real** = self.IAN( ) / self.allAttributes( ) -> iterate(elem: Attribute; acc: Integer = 0 | if self.FCV(elem) then acc + 1 else acc endif) pre: self.AAN( ) > 0
Comments	In order to calculate MAA, number of attributes must be greater than zero.

Name	***MAT – Measure of Abstraction***
Informal Definition	*The average of functional and attribute abstraction measures.*
Formal Definition	***Classifier:: MAT( ): Real*** *= (self.MFA( ) + self.MAA( )) / 2*
Comments	

Name	***MOA – Measure of Aggregation***
Informal Definition	*Count of the number of data declarations whose types are user defined Classes.*

Formal Definition	***Classifier:: MOA( ): Integer***
	= self.allAttributes( )
	    -> iterate(elem: Attribute; acc: Integer = 0 |
	        if self.namespaceImpl.oclAsType(Package).allClasses( )
	          -> includes(elem.type.oclAsType(Class)) then
	            acc + 1
	        else
	            acc
	        endif)

Comments	

Name	***MOS – Measure of Association***
Informal Definition	*Measure of the number of direct relationships a Class has to objects of other Classes. The metric value is the same as the DCC measure.*
Formal Definition	***Classifier:: MOS( ): Integer*** *= self.DCC( )*
Comments	

Name	***MRM – Modeled Relationship Measure***
Informal Definition	*Measure of the total number of Attribute and Parameter based relationships in a Class.*
Formal Definition	***Classifier:: MRM( ): Integer*** *= self.MOS( ) + self.NAD( )*
Comments	

Name	***DAM – Data Access Metric***
Informal Definition	*The ratio of the number of private Attributes to the total number of Attributes declared in a Class.*
Formal Definition	***<u>Classifier:: DAM( ): Real</u>*** *= self.PRIAN( ) / self.AAN( )* *pre: self.AAN() > 0*
Comments	*A high value of DAM is desired. The pre-condition states that the Class must have Attributes.*

Name	***OAM – Operation Access Metric***
Informal Definition	*The ratio of the number of public Methods to the total number of Methods declared in the Class.*
Formal Definition	***<u>Classifier:: OAM( ): Real</u>*** *= self.PUBON( ) / self.AON( )* *pre: self.AON() > 0*
Comments	*A high value for OAM is desired. The pre-condition states that the Class must have Operations.*

Name	***MAM – Member Access Metric***
Informal Definition	*This metric computes the access to all the members (Attributes and Methods) of a Class.*
Formal Definition	***<u>Classifier:: MAM( ): Real</u>*** *= ((1 - self.DAM( )) + self.OAM( )) / 2*
Comments	*A high value for MAM is desired.*

Name	***DOI – Depth of Inheritance***
Informal Definition	*The length of the inheritance path from the root to the current Class.*

Formal Definition	***<u>Classifier:: DOI( ): Integer</u>***
	= if self.isRoot( ) then 0 else
	    if PARN( ) = 1 then
	      1 + self.parents( ) -> iterate(elem: GeneralizableElement; acc: Integer = 0 |
	                        acc + elem.oclAsType(Class).DOI( ))
	    else
	      self.parents( ) -> iterate(elem: GeneralizableElement; acc:Integer = 0 |
	                    acc + elem.oclAsType(Class).DOI( ))
	    endif
	  endif

Comments	

Name	***NOC – Number of Children***
Informal Definition	*Count of the number of immediate children (sub classes) of the Class.*
Formal Definition	***Classifier:: NOC( ): Integer*** *= self.CHIN( )*
Comments	

Name	***NOA – Number of Ancestors***
Informal Definition	*Counts the number of distinct Classes which a Class inherits.*
Formal Definition	***Classifier:: NOA( ): Integer*** *= self.ASCN( )*
Comments	

Name	***NOM – Number of Methods***
Informal Definition	*Count of all the Methods defined in a Class.*
Formal Definition	***Classifier:: NOM( ): Integer*** *= self.AON( )*
Comments	

Name	***CIS – Class Interface Size***
Informal Definition	*Number of public Methods in a Class.*
Formal Definition	***Classifier:: CIS( ): Integer*** *= self.PUBON( )*
Comments	

Name	***NOI – Number of Inline Methods***
Informal Definition	*Number of Methods that are inline, such as Methods that access and get/set Attributes. These methods are marked as inline in C++.*
Formal Definition	***Classifier:: NOI( ): Integer*** *=*
Comments	*This metric is language dependent and can not be formalized upon the UML meta-model. This is because no Attribute is provided by the UML meta-model to specify whether a Method is inline or not.*

Name	***NOP – Number of Polymorphic Methods***
Informal Definition	*Count of the Methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.*
Formal Definition	***Classifier:: NOP( ): Integer*** *=*
Comments	*As in the case of NOI, the UML meta-model has no support to say whether a Method is virtual or not. A solution using stereotypes could solve the problems of NOI and NOP definitions, but in this case it would be hard-coded, which is not desirable.*

Name	***NOO – Number of Overloaded Operators***
Informal Definition	*Count of the overloaded operator methods (C++) defined in the Class.*

Formal Definition	***Classifier:: NOO( ): Integer***
	`= self.overriddenOperations( )`
	`    -> iterate( elem: Operation; acc:Integer = 0 |`
	`            if ( self.ascendants( ).oclAsType( Classifier ).allOperations( ) -> asSet( )`
	`                -> collect( name ) -> includes( elem.name ) and`
	`                  ( self.ascendants( ).oclAsType( Classifier ).allOperations( )`
	`                  -> asSet( ) -> select( o: Operation | o.name = elem.name)`
	`                  -> iterate( elem2: Operation; acc2: Integer = 0|`
	`                        if not ( elem2.parameter.type = elem.parameter.type ) then`
	`                         acc2 + 1 else acc2 endif ) ) > 0 )`
	`            then`
	`              acc + 1`
	`            else`
	`              acc`
	`            endif )`

Comments	*This metric compares the signatures (names and parameters) of the overridden Operations defined in the current Classifier with the ones defined on its ancestors in order to look for overloaded operators.*

Name	***NPT – Number of Unique Parameter Types***
Informal Definition	*Number of different Parameter types used in the Methods of the Class.*
Formal Definition	***Classifier:: NPT( ): Integer*** *= self.allOperations( ).parameter.type -> asSet( ) -> size( )*
Comments	

Name	***NPM – Number of Parameters per Method***
Informal Definition	*Average of the number of Parameters per Method in the Class. Computed by summing the Parameters of all Methods and dividing by the number of Methods in the Class.*
Formal Definition	***Classifier:: NPM( ): Real*** *= self.allOperations( ).parameter -> size( ) / AON( )* *pre: self.AON() > 0*
Comments	*The pre-condition states the Class must have some Operations.*

Name	***NOD – Number of Attributes***
Informal Definition	*Number of Attributes in the Class.*
Formal Definition	***Classifier:: NOD( ): Integer*** *= self.AAN( )*
Comments	

Name	***NAD – Number of Abstract Data Types***
Informal Definition	*Number of user defined objects (ADTs) used as Attributes in the Class and which are necessary to instantiate an object instance of the (aggregate) Class.*
Formal Definition	***Classifier:: NAD( ): Integer*** *= self.allAttributes( ).type.oclAsType(Classifier)* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) ->asSet( ) -> size( )*
Comments	

Name	***NRA – Number of Reference Attributes***
Informal Definition	*Number of pointers and references used as Attributes in the Class.*
Formal Definition	***Classifier:: NRA( ): Integer*** *=*
Comments	*Pointers and references are language specific, and they are not part of the UML Data Types. However, the types system could be extended to support pointers and references. This way, the metric could be similar to* *= self.allAttributes( ).type.oclAsType(Classifier)* *-> select(c: Classifier | c.oclIsKindOf(Pointer)) ->asSet( ) -> size( )*

Name	***NPA – Number of Public Attributes***
Informal Definition	*Number of Attributes that are declared as public in the Class.*
Formal Definition	***Classifier:: NPA( ): Integer*** *= self.PUBAN( )*
Comments	

Name	***CSB – Class Size in Bytes***
Informal Definition	*The size of objects in bytes that will be created from the Class declaration. The size is computed by summing the size of all Attributes declared in the Class.*
Formal Definition	***Classifier:: CSB( ): Integer*** *=*
Comments	*The size of a type is language and platform dependent. As this metric was developed based upon the C++ language, the solution could be precise. However, if we do not consider the language and platform dependency, it would be necessary to build a "table" with all the possible combinations of Attributes size and further select the appropriated ones, summing up all the objects size. As is can evolve since new languages and architectures are created, such a "table" would be a hard-coded solution. This is a deficiency of the metric.*

Name	***CSM – Class Size Metric***
Informal Definition	*Sum of the number of Methods and Attributes in the Class.*
Formal Definition	***Classifier:: CSM( ): Integer*** *= self.AAN( ) + self.AON( )*
Comments	

Name	***CAM – Cohesion Among Methods of Class***
Informal Definition	*Computes the relatedness among Methods of the Class based upon the Parameter list of the Methods. The metrics is computed using the summation of the intersection of Parameters of a Method with the maximum independent set of all Parameter types in the Class.*
Formal Definition	***Classifier:: CAM( ): Real*** *= (self.allOperations( )* *-> iterate(elem: Operation; acc: Integer = 0 |acc + elem.parameter.type -> asSet( )* *-> size( )))* */ (AON( ) * self.allOperations( ).parameter.type -> asSet( ) -> size( ))*
Comments	

Name	***DCC – Direct Class Coupling***
Informal Definition	*Count of the different number of Classes that a Class is directly related to. The metric includes Classes that are directly related by Attribute declarations and message passing (Parameters) in Methods.*
Formal Definition	***Classifier:: DCC( ): Integer*** *= self.allAttributes( ).type* *-> union(self.allOperations( ).parameter.type).oclAsType(Classifier)* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) -> asSet( ) -> size( )*
Comments	

Name	***MCC – Maximum Class Coupling***
Informal Definition	*This metric not only includes Classes that are directly related to a Class by Attributes and Methods, but also Classes that are indirectly related through the directly related Classes .*
Formal Definition	***Classifier:: MCC( ): Integer*** *= self.allAttributes( ).type* *-> union(self.allOperations( ).parameter.type).oclAsType(Classifier)* *-> iterate(elem: Classifier; acc: Bag(Classifier) = oclEmpty(Bag(Classifier)) |* *acc -> union(elem.allAttributes( ).type* *-> union(elem.allOperations( ).parameter.type).oclAsType(Classifier)))* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) -> asSet( ) -> size( )*
Comments	

Name	***DAC – Direct Attribute Based Coupling***
Informal Definition	*This metric is a direct count of the number of different Class types that are declared as Attribute references inside a Class.*
Formal Definition	***Classifier:: DAC( ): Integer*** *= self.allAttributes( ).type.oclAsType(Classifier)* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) -> asSet( ) -> size( )*
Comments	

Name	***MAC – Maximum Attribute Based Coupling***
Informal Definition	*Number of different Class Types that are declared as Attribute references directly and indirectly inside the Class.*
Formal Definition	***Classifier:: MAC( ): Integer*** *=*
Comments	*The UML meta-model, as mentioned in NRA, has not support for identifying references.*

Name	***DPC – Direct Parameter Based Coupling***
Informal Definition	*Number of Class object types that are required directly for a message passing (Parameters) to Methods in the Class.*
Formal Definition	***Classifier:: DPC( ): Integer*** *= self.allOperations( ).parameter.type.oclAsType(Classifier)* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) -> asSet( ) -> size( )*
Comments	

Name	***MPC – Maximum Parameter Based Coupling***
Informal Definition	*Number of Class object types that are required directly and indirectly for message passing (Parameters) in the Class.*
Formal Definition	***Classifier:: MPC( ): Integer*** *= self.allOperations( ).parameter.type.oclAsType(Classifier)* *-> iterate(elem: Classifier; acc: Bag(Classifier) = oclEmpty(Bag(Classifier)) |* *acc -> union(elem.allOperations( ).parameter.type).oclAsType(Classifier))* *-> reject(c: Classifier | c.oclIsKindOf(DataType)) ->asSet( ) -> size( )*
Comments	

Name	***VOM – Virtuality of Methods***
Informal Definition	*Number of virtual Methods in a Class. Overridden virtual Methods are counted only once.*
Formal Definition	***Classifier:: VOM( ): Integer*** *=*
Comments	*The UML meta-model has no support to identify virtual Methods. This shows that this metric is language dependent.*

Name	***CCN – Class Complexity Based on Nodes in AST***
Informal Definition	*Measures the complexity of the Class based on the number of nodes it takes to construct the definition of the Class in an AST representation.*
Formal Definition	***Classifier:: CCN( ): Integer*** *=*
Comments	*The metric does not specify how to build the AST, and how to count the nodes.*

Name	***CEC – Class Entropy Complexity***
Informal Definition	*Computes the complexity of the Class based upon the information content of the Class. The information content of the Class is measured by counting the occurrences of different name strings in a Class definition.*
Formal Definition	***Classifier:: CEC( ): Integer*** *=*
Comments	*The information content of a Class requires the source code to be available.*

Name	***CCD – Class Complexity Based on Data***
Informal Definition	*Computes complexity based upon the number of components (Attributes) that are defined in the Class. All component declarations are resolved to the basic primitives (integers, doubles and characters). The metric value is a count of the number of primitives.*
Formal Definition	***Classifier:: CCD( ): Integer*** *= self.AAN( ) + (self.allAttributes( ).type.oclAsType(Classifier)* *-> iterate(elem: Classifier; acc: Integer = 0 | acc + elem.AAN( )))*
Comments	

Name	***CCP – Class Complexity Based on Method Parameters***
Informal Definition	*Estimates complexity based upon the number of Parameters required to call Methods of the Class. Inherited Method Parameters are also included in the computation of the metric value.*
Formal Definition	***Classifier:: CCP( ): Integer*** *= self.allOperations( ).parameter -> size( )*
Comments	

Name	***CCM – Class Complexity Based on Members***
Informal Definition	*This metric is an aggregate of the data and method Parameter complexities.*
Formal Definition	***Classifier:: CCM( ): Integer*** *= self.CCD( ) + self.CCP( )*
Comments	

The QMOOD set of metrics suffers from some deficiencies. First, it mixes design with code (as MOOSE and QMOOD) and as such some metrics can not be formalized. Second, it was created considering only C++ models, and it uses some concepts that are not supported or are not the standard in other languages, as *inline* and *virtual* methods. Third, some metrics can have more than one interpretation, which makes them difficult to be standardized and tested. Fourth, some metrics seem to be the same, even having different names.

Although it was possible to formalize most metrics in the set, in some cases where different interpretations arise, some metrics can still be wrongly defined.

7 Conclusions and Further Work

This chapter simply outlines our conclusions and shows some directions for future work, presenting the different extensions that could improve, according to us, the current contribution.

"Success usually comes to those that are too busy to be looking for it."

Henry David Thoreau (1817-1862)

7.1 CONCLUSIONS

Measurement plays an important role in everyday discipline and there is no question that it is an important method in order to get higher quality of software [Zuse]. Measurement enables engineers to obtain quantitative measures of attributes in entities and also serves as a baseline for classification, comparison, and analysis of these attributes. Dieter Rombach [Rombach, 1990], who worked with the Software Engineering Laboratory (SEL) in the USA, said at the Eurometrics 1991 in Paris:

"We should no longer ask if we should measure, the question today is how."

Software measurement contributes to software quality from various aspects, such as understandability, complexity, reliability, testability and maintainability, as well as performance and productivity of software projects [Tang et al., 2002]. With the pervasive popularity and adaptation of object-oriented programming languages and methodologies in software development, software metrics tailored to object-oriented characteristics are essential to improve the object-oriented process and products.

Although in the past much research has been done in this area, there are still many open questions. Firstly, there is a lack of maturity in software measurement. Secondly, there is no standardization of software measures. Many of the proposed software measures are not widely accepted. Validation of software measures in order to predict an external variable is still a research topic for the future. Calculations of correlations and regression analysis require a discussion of measurement scales.

In this work, we try to solve these problems by the formalization of several metrics definitions. We used the OCL, a part of the UML standard, to define object-oriented design metrics in a very natural and understandable way. The precision granted by the formality of OCL comes at a much lower cost, for both practitioners and tool builders, than when using other formal specification constructs. Since UML became a *de facto* standard, both in academia and industry, more and more people are expected to use OCL in their designs and, as such, to understand its syntax and semantics.

We believe the time has come for object-oriented metrics research community to standardize the way we define the metrics, as it happened with the object-oriented analysis and design notations. Although we are strong believers that diversity and innovation should not be constrained, we are indebted that standardization efforts to those that are our final users – the design practitioners and those that support and train them, such as tool manufacturers, consultants, professional trainers or academic teachers – can bring several benefits. We think that such a standardization effort will not reach widespread acceptance if it is not integrated with the current state-of-the-practice object-oriented design technology. We hope to have shown here that this is possible, promoting the utilization of the design metrics on UML designs.

We expect that our efforts can indeed contribute to Software Engineering practical aspects, providing better tool support for metrics and also to emphasize the importance of quantitative approaches on industry and academic applications. We will be happy if this document generates some discussion and feedback around this topic.

7.2 FUTURE WORKS

This section presents the possibilities of new studies in the area.

7.2.1 Formalization of the Metrics Sets upon Different Meta-Models

Besides formalizing some metrics sets using the UML meta-model, it is possible to make a similar effort based upon the OML (*OPEN Modeling Language*) meta-model. OML emerged from the OPEN – Object-Oriented Process, Environment and Notation – consortium [Henderson-Sellers and Edward, 1994; Graham, 1995; Odell, 1995; Firesmith, 2000]. The latter is supported by a large group of well-known methodologists such as Brian Henderson-Sellers (author of the *MOSES* method [Henderson-Sellers, 1991; Henderson-Sellers and Edward, 1994]), Ian Graham (author of *SOMA – Semantic Object Modeling Approach* [Graham, 1995]), Donald Firesmith [Firesmith, 2000] and Jim Odell [Odell, 1995].

7.2.2 Creation of a Framework for Measuring Metrics Characteristics (The Meta-Metrics Framework)

It is feasible to abstract the common characteristics of all the formalized metrics in order to build a high level model. The latter would be a framework for describing, classifying and accessing existing metric sets, as well as a basis for the production of new ones. Our idea is to introduce a quality model for metrics, which will consequently facilitate the creation of the meta-metrics – metrics that measure metrics characteristics.

Some examples of those characteristics, still illustrated in an informal way, are:

- Understandability: the effort required to understand the metric. It is inversely proportional to the weighted sum of the meta-model classes and associations involved in the metric definition.
- Efficiency: number of resources necessary to compute the metric. It is inversely proportional to the computational complexity of the metric calculation algorithm.

These characteristics could be expressed with the OCL.

7.2.3 Formalization of Other Metrics Sets

We have formalized the definitions of the most accepted sets of metrics for object-oriented models. Notwithstanding, a plenty of sets still exist, and it is possible to apply the same approach presented here on these ones.

7.2.4 Use of Other UML Diagrams as Input

In this work we were mainly concerned about design metrics extracted from UML class diagrams. We plan to investigate which are relevant metrics for dealing not only with the static structure of models, but also with its behavior. For such, we can investigate the possibilities of metrics formalization (and even creation) based on other UML diagrams.

7.2.5 Metrics for Prototype-Based Environments

As far as we know, there are no metrics for prototype based languages. It is possible not only to use or to build a meta-model to serve as background for prototype metrics definitions, but also to create new metrics sets for prototype technology.

7.2.6 Metrics for Human-Computer Interaction

Pressman [Pressman, 2000] points out the needs of new metric sets for dealing with human-computer interaction. We plan to extend the UML semantics (possibly using stereotypes) in order to create and formalize some metrics that can collect characteristics of good graphical user interfaces. The LabIUtil in Brazil[32] studies which are these good characteristics, and we can try to model them extending the approach presented in this work wit the UML extension mechanisms.

7.2.7 Adaptation to the UML Semantic Model Version 1.4

In the beginning of this work, the UML semantics model in its current latest version (1.4) was not available for use. Thus, in this work, the version 1.3 of the UML meta-model was used. However, it is probably easy to adapt the contribution presented here to the new version of this meta-model.

[32] The site in Portuguese is http://www.labiutil.inf.ufsc.br

APPENDIX A – DATA TYPES IN UML

This appendix describes the data types used for defining UML. There are three kinds of data types: *primitive*, *enumeration* and *classes*. They are presented below. Not all of these were used in this document.

In this document when referring to an *association end* for a binary association, the target end is the one whose properties are being discussed and the source end is the other.

Primitive Types

Integer

In the meta-model an Integer is an element in the (infinite) set of integers (…, -2, -1, 0, 1, 2…).

UnlimitedInteger

In the meta-model UnlimitedInteger defines a data type whose range is the nonnegative integers augmented by the special value “unlimited”. It is used for the upper bound of multiplicities.

String

In the meta-model a String defines a stream of text.

Time

In the meta-model a Time defines a value representing an absolute or relative moment in time and space. A Time has a corresponding string representation.

Enumeration Types

AggregationKind

An enumeration that denotes what kind of aggregation an Association is. When placed on a target end, specifies the relationship of the target end to the source end. AggregationKind defines an enumeration whose values are:

- None: The end is not an aggregate.
- Aggregate: The end is an aggregate. Therefore, the other end is a part and must have the aggregation value of none. The part may be contained in other aggregates.
- Composite: The end is a composite. Therefore, the other end is a part and must have the aggregation value of none. The part is strongly owned by the composite and may not be part of any other composite.

Boolean

In the meta-model, Boolean defines an enumeration that denotes a logical condition. Its values are:

- True: The Boolean condition is satisfied.
- False: The Boolean condition is not satisfied.

CallConcurrencyKind

An enumeration that denotes the semantics of multiple concurrent calls to the same passive instance (i.e., an Instance originating from a Classifier with isActive=false). It is an enumeration with the values:

- Sequential: Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.
- Guarded: Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete.
- Concurrent: Multiple calls from concurrent threads may occur simultaneously to one Instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics.

ChangeableKind

In the meta-model ChangeableKind defines an enumeration that denotes how an AttributeLink or LinkEnd may be modified. Its values are:

- Changeable: No restrictions on modification.
- Frozen: The value may not be changed from the source end after the creation and initialization of the source object. Operations on the other end may change a value.
- AddOnly: If the multiplicity is not fixed, values may be added at any time from the source object, but once created a value may not be removed from the source end. Operations on the other end may change a value.

OrderingKind

Defines an enumeration that specifies how the elements of a collection are arranged. It is used in conjunction with elements that have a multiplicity, in cases where the multiplicity value is greater than one. The ordering must be determined and maintained by operations that modify the set. Values are:

- Unordered: The elements of the collection have no inherent ordering.
- Ordered: The elements of the collection have a sequential ordering.

Other possibilities (such as sorted) may be defined by declaring additional keywords.

ParameterDirectionKind

In the meta-model ParameterDirectionKind defines an enumeration that denotes if a Parameter is used for supplying an argument and/or for returning a value. The enumeration values are:

- In: An input Parameter (may not be modified).
- Out: An output Parameter (may be modified to communicate information to the caller).
- Inout: An input Parameter that may be modified.
- Return: A return value of a call.

PseudostateKind

In the meta-model, PseudostateKind[33] defines an enumeration that discriminates the kind of Pseudostate. The enumeration values are:

[33] Not used in this document.

- Choice: Splits an incoming transition into several disjoint outgoing transitions. Each outgoing transition has a guard condition that is evaluated after prior actions on the incoming path have been completed.
- DeepHistory: When reached as the target of a transition, restores the full state configuration that was active just before the enclosing composite state was last exited.
- Fork: Splits an incoming transition into several concurrent outgoing transitions. All of the transitions fire together.
- Initial: The default target of a transition to the enclosing composite state.
- Join: Merges transitions from concurrent regions into a single outgoing transition. All the transitions fire together.
- Junction: Chains together transitions into a single run-to-completion path. May have multiple input and/or output transitions. Each complete path involving a junction is logically independent and only one such path fires at one time.
- ShallowHistory: When reached as the target of a transition, restores the state within the enclosing composite state that was active just before the enclosing state was last exited. Does not restore any sub states of the last active state.

ScopeKind

In the meta-model ScopeKind defines an enumeration that denotes whether a feature belongs to individual instances or to an entire classifier. Its values are:

- Instance: The feature pertains to Instances of a Classifier. For example, it is a distinct Attribute in each Instance or an Operation that works on an Instance.
- Classifier: The feature pertains to an entire Classifier. For example, it is an Attribute shared by the entire Classifier or an Operation that works on the Classifier, such as a creation operation.

VisibilityKind

In the meta-model VisibilityKind defines an enumeration that denotes how the element to which it refers is seen outside the enclosing name space. Its values are:

- Public: Other elements may see and use the target element.
- Protected: Descendants of the source element may see and use the target element.
- Private: Only the source element may see and use the target element.

Classes

Expression

In the meta-model an Expression defines a statement that will evaluate to a (possibly empty) set of instances when executed in a context. An Expression does not modify the environment in which it is evaluated. An expression contains an expression string and the name of an interpretation language with which to evaluate the string.

Attributes

- Language: Names the language in which the expression body is represented. The interpretation of the expression depends on the language.
- Body: The text of the expression expressed in the given language.

Mapping

In the meta-model a Mapping[34] is an expression that is used for mapping ModelElements. For exchange purposes, it should be represented as a String.

Attributes

- Body: A string describing the mapping. The format of the mapping is currently unspecified in UML.

Name

In the meta-model a Name defines a token that is used for naming ModelElements. Each Name has a corresponding String representation. For purposes of exchange a name should be represented as a String.

Attributes

- Body: The name string.

LocationReference

It[35] designates a position within a behavior sequence for the insertion of an extension use case. It may be a line or range of lines in code, or a state or set of states in a state machine, or some other means in a different kind of specification.

Multiplicity

In the meta-model a Multiplicity defines a non-empty set of non-negative integers. A set which contains only zero ({0}) is not considered a valid Multiplicity. Every Multiplicity has at least one corresponding String representation.

MultiplicityRange

In the meta-model a MultiplicityRange[36] defines a range of integers. The upper bound of the range cannot be below the lower bound. The lower bound must be a nonnegative integer. The upper bound must be a nonnegative integer or the special value *unlimite*d, which indicates there is no upper bound on the range.

[34] Not used in this document.
[35] Not used in this document.
[36] Not used in this document.

APPENDIX B – THE GOODLY META-MODEL

The GOODLY[37] language [Abreu, 1997; Abreu, 2000] allows the textual representation of object-oriented design information such as modules, classes and templates, inheritance hierarchies, attributes, operations and their parameters and message exchanges.

Since it is a design language, GOODLY is not computationally complete: neither algorithmic capabilities, nor control flow structures, are present. It is used mainly in [Abreu, 2000] as a common intermediate formalism that allows the extraction of quantitative data [Abreu1998b]. In this document, the GOODLY meta-model is introduced to present the MOODlib, a library of auxiliary functions to calculate metrics. The meta-model constructs are briefly explained below, while the MOODlib is discussed in appendix C.

Specification

The structural unit at the highest abstraction level is the Specification. It is an identified package formed by a set of interrelated design parts. A Specification is produced by a named person, team or company, and is made available as a whole and not only partially.

A Specification may “use” other Specifications. This means that in order to provide the services for which they were conceived, the components in a Specification (the “using” one) may depend on the collaboration of components in others (the “used” ones).

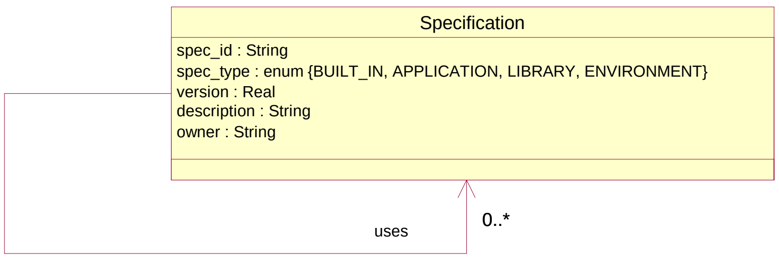

Figure B.1 – *Specification in the GOODLY Meta-Model*

Each Specification mentions which others it must use directly, so that the origin of all used symbols is known. By other words, if the specification A uses symbols of specifications B1 and B2, and B1 uses symbols defined in specification C11 and C12, then this “indirect” use in A of symbols defined in C11 and C12 is not enlisted in A. It does not make sense to explicit that a specification uses itself, since that is implicit.

There are four specifications types, which can be seen in figure B.1. They are explained in [Abreu, 2000].

Module

A Specification is organized as a set of Modules (figure B.2). A Module is a set of classes (types) grouped by a given aggregation criterion. The Specification and Module abstraction levels correspond, in the UML meta-model, to two nested packaging levels.

[37] It stands for a Generic Object Oriented Design Language? Yes!

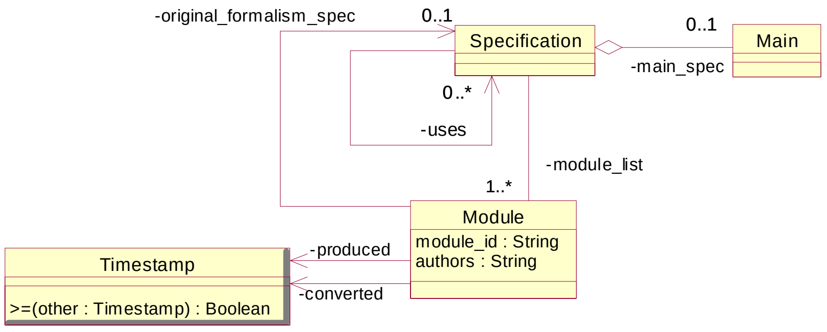

Figure B.2 – *Modules in the GOODLY Meta-Model*

TimeStamp

The utility Timestamp class has the relational operators for manipulating dates (for example to compare the current timestamp with another one), with the following interface:

```
Timestamp::=(other: Timestamp): Boolean
Timestamp::<>(other: Timestamp): Boolean
Timestamp::>(other: Timestamp): Boolean
Timestamp::>=(other: Timestamp): Boolean
Timestamp::<(other: Timestamp): Boolean
Timestamp::<=(other: Timestamp): Boolean
```

Class

The basic component of a Module is the Class. Each Class must have a unique identifier within each Module. Each class has both a set of Attributes (comprising both instance variables and class variables), which characterize the object or class state and a set of Operations that characterize the object behavior.

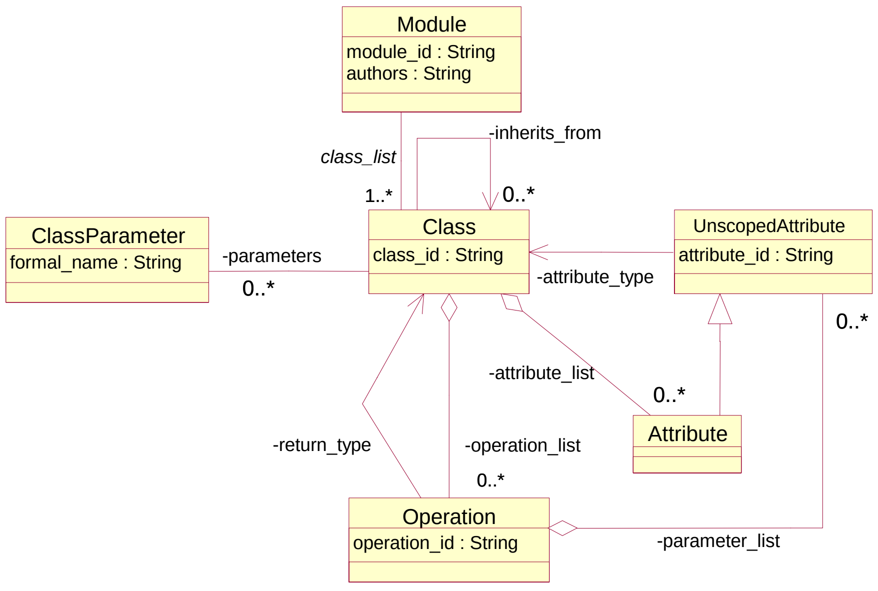

Figure B.3 – *Classes and its Features in the GOODLY Meta-Model*

Attribute

Each Attribute has an identifier, a type (class) and a scope. Within the same Class, attribute identifiers should always be unique:

Operations

Each Operation has an interface and a body or implementation. The interface includes its identifier, the formal parameter list and corresponding type(s), the returning type, its scope and traceability information. Within the same Class, Operations' signature (identifier plus the parameter list) should always be unique:

Scope

The Scope is characterized by the visibility that components (classes) have on the Attribute or Operation. Invisibility implies inability to use. The following scope options are *private*, *protected*, *public*, *discriminated*, *class_tree* and *module*. They are not detailed here. The Scope of an Attribute or Operation always includes the own Class where it is defined. Therefore it is useless to include it explicitly in the Scope clause.

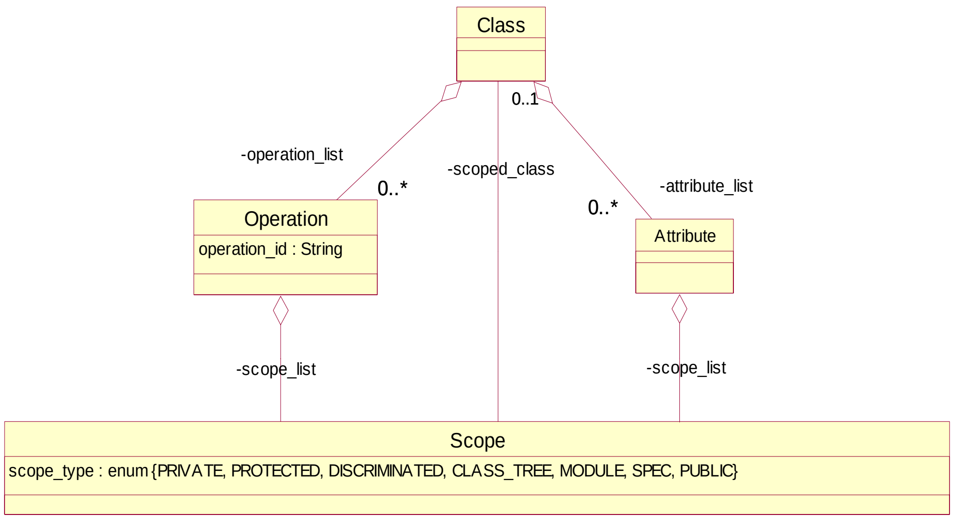

Figure B.4 – *Scope of Attributes and Operations in the GOODLY Meta-Model*

Implementation Body

Both the Main section of a Specification and every Operation have an Implementation Body. The latter may have local Attributes defined on it, may employ Attributes from named Classes and can issue requests (send messages) to instances of the same or of other Classes.

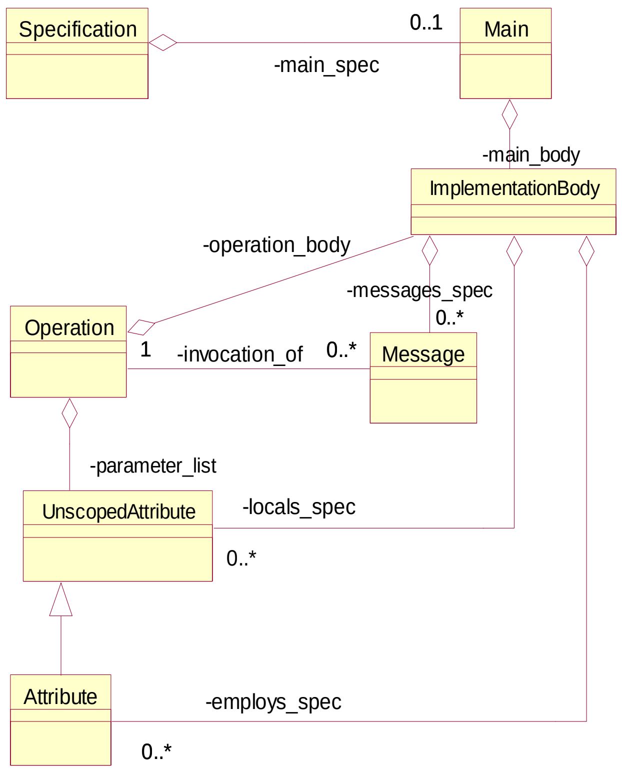

Figure B.5 – *The Operations Implementation Body in the GOODLY Meta-Model*

With the previous information, it is possible to present the GOODLY model in its totality. Some classes were not explained because they are not used in the MOODlib. Notice that this meta-model is much simpler than the UML one.

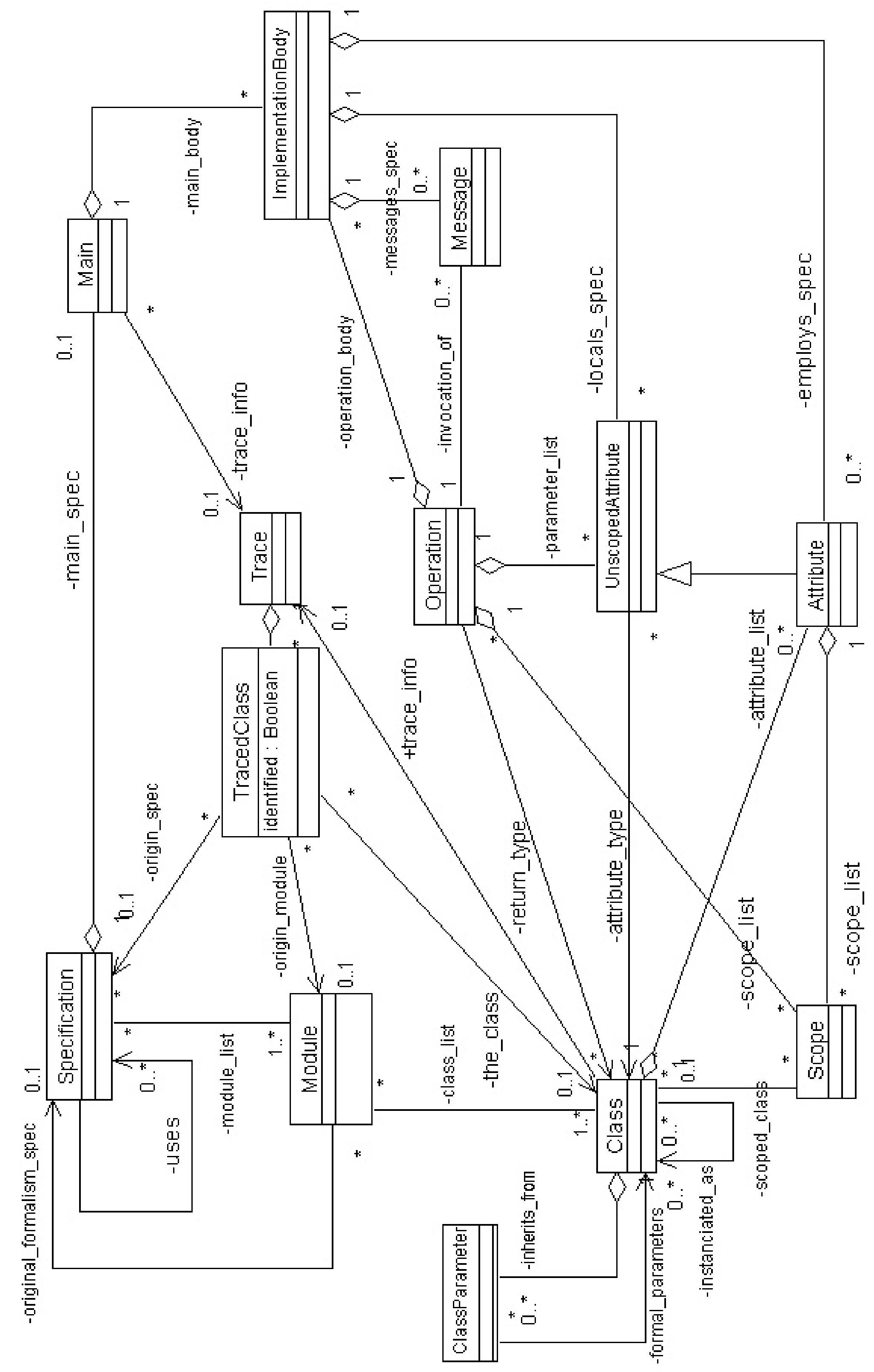

Figure B.6 – The *Full Version of the GOODLY Meta-Model*

APPENDIX C – THE MOODLIB

This section presents the functions that belong to the MOODlib [Abreu, 2001], defined over the GOODLY meta-model. Each of them is classified over one of the following meta-classes: Attribute, Class, Operation and Specification. The following tables show the categories of the functions, which are presented in sequence.

C.1 Functions Designation

Acronym	Name	Type
ACV(c)	Attribute to Class Visibility	Boolean
ASV(s)	Attribute to Specification Visibility	Percentage
AUN(s)	Attribute Use Number	Integer
AVN(s)	Attribute Visibility Number	Integer

Table C.1 – *Attribute-Level Functions*

Acronym	Name	Type
OCV(c)	Operation to Class Visibility	Boolean
OSV(s)	Operation to Specification Visibility	Percentage
OUN(s)	Operation Use Number	Integer
OVN(s)	Operation Visibility Number	Integer

Table C.2 – *Operation-Level Functions*

Acronym	Name	Type
IsInternal(s)	Internal class predicate	Boolean
IsRoot	Root class predicate	Boolean
IsLeaf	Leaf class predicate	Boolean

Table C.3 – *Class-Level Predicate Functions*

Acronym	Name	Type
Children( )	Set of children classes	Set(Class)
Descendants( )	Set of descendant classes	Set(Class)
Parents( )	Set of parent classes	Set(Class)
Ascendants( )	Set of ascendant classes	Set(Class)
CoupledClasses	Set of coupled classes	Set(Class)
NewOperations( )	Set of class's new operations	Set(Operation)
InheritedOperations( )	Set of class's inherited operations	Set(Operation)
OverriddenOperations( )	Set of class's overridden operations	Set(Operation)
DefinedOperations( )	Set of class's defined operations	Set(Operation)
AvailableOperations( )	Set of class's available operations	Set(Operation)
NewAttributes( )	Set of class's new attributes	Set(Attribute)
InheritedAttributes( )	Set of class's inherited attributes	Set(Attribute)
OverriddenAttributes( )	Set of class's overridden attributes	Set(Attribute)
DefinedAttributes( )	Set of class's defined attributes	Set(Attribute)
AvailableAttributes( )	Set of class's available attributes	Set(Attribute)

Table C.4 – *Class-Level Set Functions*

Acronym	*Name*	*Type*
CC	Children Count	Integer
DC	Descendants Count	Integer
PC	Parents Count	Integer
AC	Ascendants Count	Integer
ON	Operations New	Integer
OI	Operations Inherited	Integer
OO	Operations Overridden	Integer
OD	Operations Defined	Integer
OA	Operations Available	Integer
AN	Attributes New	Integer
AI	Attributes Inherited	Integer
AO	Attributes Overridden	Integer

AD	Attributes Defined	Integer
AA	Attributes Available	Integer

Table C.5 – *Class-Level Counting Functions*

Acronym	Name	Type
AllClasses	Set of all classes	Set(Class)
BaseClasses(s)	Set of base classes	Set(Class)
SupplierClasses(s)	Set of supplier classes	Set(Class)
RelatedClasses(s)	Set of related classes	Set(Class)

Table C.6 – *Specification-Level Set Functions*

Acronym	Name	Type
TC	Total number of Classes	Integer
TON	Total Operations New	Integer
TOO	Total Operations Overridden	Integer
TOD	Total Operations Defined	Integer
TOI	Total Operations Inherited	Integer
TOA	Total Operations Available	Integer
TAN	Total Attributes New	Integer
TAO	Total Attributes Overridden	Integer
TAD	Total Attributes Defined	Integer
TAI	Total Attributes Inherited	Integer
TAA	Total Attributes Available	Integer
IL(s)	Inheritance Links	Integer
TIL	Total Inheritance Links	Integer
CL(s)	Coupling Links	Integer
TCL	Total Coupling Links	Integer

Table C.7 – *Specification-Level Counting Functions*

C.2 Functions Definition

The following functions just reproduce the one of MOODlib, as they are defined on [Abreu, 2001].

Attribute - Level functions

Name	***ACV – Attribute to Class Visibility***
Informal definition	*Predicate that indicates if a given Class can access the Attribute.*
Formal definition	***Attribute::ACV(c: Class): Boolean*** *post: result = scope_list->exists(* *(class = c) or* *(scope_type = #PUBLIC) or* *(scope_type=#SPEC) and (class.module.specification=c.module.specification) or* *(scope_type = #MODULE) and (class.module = c.module) or* *(scope_type = #CLASS_TREE) and scoped_class.Descendants()->includes(c) or* *(scope_type = #PROTECTED) and class.Descendants()->includes(c) or* *(scope_type = #DISCRIMINATED) and (scoped_class = c))*
Comments	

Name	***AVN – Attribute Visibility Number***
Informal definition	*Number of Classes in the considered Specification where the Attribute can be accessed.*
Formal definition	***Attribute::AVN(s: Specification): Integer*** *post: result = s.AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *if self.ACV(elem) then* *acc + 1* *else* *acc* *endif)*
Comments	

Name	***ASV – Attribute to Specification Visibility***
Informal definition	*Percentage of Classes in the considered Specification where the Attribute can be accessed (excludes the Class where the Attribute is declared).*
Formal definition	***Attribute::ASV(s: Specification): Percentage*** *pre: s.TC() > 1* *post: result = (AVN(s) -1) / (s.TC() -1)*
Comments	

Name	***AUN – Attribute Use Number***
Informal definition	*Number of Classes in the considered Specification where the Attribute is used (excludes the Class where the Attribute is declared).*
Formal definition	***Attribute::AUN(s: Specification): Integer*** *post: result = s.AllClasses()* *-> select(operation_list.operation_body.employs_spec->includes(self))* *-> asSet() -> size()*
Comments	

Operation - Level Functions

Name	***OCV – Operation to Class Visibility***
Informal definition	*Predicate that indicates if a given Class can access the Operation.*
Formal definition	***Operation::OCV(c: Class): Boolean*** *post: result = scope_list->exists(* *(class = c) or* *(scope_type = #PUBLIC) or* *(scope_type = #SPEC) and (class.module.specification=c.module.specification)or* *(scope_type = #MODULE) and (class.module = c.module) or* *(scope_type = #CLASS_TREE) and scoped_class.Descendants()->includes(c) or* *(scope_type = #PROTECTED) and class.Descendants()->includes(c) or* *(scope_type = #DISCRIMINATED) and (scoped_class = c))*
Comments	

Name	***OVN – Operation Visibility Number***
Informal definition	*Number of Classes in the considered Specification where the Operation can be accessed.*
Formal definition	***Operation::OVN(s: Specification): Integer*** *post: result = s.AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *if self.OCV(elem) then* *acc + 1* *else* *acc* *endif)*
Comments	

Name	***OSV – Operation to Specification Visibility***
Informal definition	*Percentage of Classes in the considered Specification where the Operation can be accessed (excludes the Class where the Operation is declared).*
Formal definition	***Operation::OSV(s: Specification): Percentage*** *pre: s.TC() > 1* *post: result = (OVN(s) –1) / (s.TC() -1)*
Comments	

Name	***OUN – Operation Use Number***
Informal definition	*Number of Classes in the considered Specification where the Operation is used.*
Formal definition	***Operation::OUN(s: Specification): Integer*** *post: result = s.AllClasses()* *-> select(operation_list.operation_body.messages_spec.operation* *-> includes(self)) -> asSet() -> size()*
Comments	

Class - Level Predicate Functions

Name	*IsInternal*
Informal definition	*Internal Class predicate – indicates if the Class belongs to the named Specification "s".*
Formal definition	***Class::IsInternal(s: Specification): Boolean*** *post: result = self.module.specification = s*
Comments	

Name	*IsRoot*
Informal definition	*Root Class predicate – indicates that it has no ascendants.*
Formal definition	***Class::IsRoot(): Boolean*** *post: result = Parents()->isEmpty()*
Comments	

Name	***IsLeaf***
Informal definition	*Leaf Class predicate – indicates that it has no descendants.*
Formal definition	***Class::IsLeaf(): Boolean*** *post: result = Children()->isEmpty()*
Comments	

Class - Level Set Functions

Name	***Children***
Informal definition	*Set of directly derived Classes.*
Formal definition	***Class::Children(): Set(Class)*** *post: result = Class.allInstances->select(inherits_from->includes(self))*
Comments	

Name	***Descendants***
Informal definition	*Set of all derived Classes (either directly or indirectly).*

Formal definition	***Class::Descendants(): Set(Class)***
	 post: result = Children()-> iterate(elem: Class;
	 acc: Set(Class)=Children() | acc-> union (elem.Descendants())

Comments	*This Operation is recursive. Notice that even with multiple inheritance the result is a set (no repeated Classes).*

Name	***Parents***
Informal definition	*Set of Classes from which the current Class derives directly.*
Formal definition	***Class::Parents(): Set(Class)*** *post: result = inherits_from*
Comments	

Name	***Ascendants***
Informal definition	*Set of Classes from which the current Class derives directly or indirectly.*

Formal definition	***Class::Ascendants(): Set(Class)***
	 post: result = Parents()-> iterate(elem: Class;
	 acc: Set(Class)=Parents() | acc-> union(elem.Ascendants())

Comments	*This Operation is recursive. Notice that even with common ancestors due to multiple inheritance the result is a set (no repeated Classes).*

Name	***CoupledClasses***
Informal definition	*Set of Classes to which the current Class is coupled (excluding inheritance).*
Formal definition	***Class::CoupledClasses(): Set(Class)*** *post: result = formal_parameters.instanced_as union(* *attribute_list.attribute_type union(* *operation_list.parameter_list.attribute_type union(* *operation_list.return_type union(* *operation_list.operation_body.locals_spec.attribute_type union(* *operation_list.operation_body.employs_spec.attribute_type union(* *operation_list.operation_body.messages_spec.invocation_of.class))))))*
Comments	*This function includes the coupled Classes corresponding to:* - *instantiation of Class parameters* - *Class Attributes* - *parameters of Class Operations* - *return type of Class Operations* - *local Attributes of Class Operations* - *Attributes of other Classes employed by Class Operations* - *recipients of messages sent in the Class Operations implementation body*

Name	***NewOperations***
Informal definition	*Operations defined in the Class that are not overriding inherited ones.*
Formal definition	***Class::NewOperations(): Set(Operation)*** *post: result = DefinedOperations() – InheritedOperations()*
Comments	

Name	***InheritedOperations***
Informal definition	*Number of inherited Operations that are not overridden by locally defined ones.*

Formal definition	***Class::InheritedOperations(): Set(Operation)***
	post: result = Ascendants()-> iterate(elem: Class;
	acc: Set(Operation)=Set{ } | acc->union(elem.operation_list))

Comments	

Name	***OverriddenOperations***
Informal definition	*Number of Operations defined in the Class that override inherited ones = number of inherited Operations that are overridden by locally defined ones.*
Formal definition	***Class::OverriddenOperations(): Set(Operation)*** *post:result = DefinedOperations()->intersection(InheritedOperations())*
Comments	

Name	***DefinedOperations***
Informal definition	*Number of Operations defined in the Class.*
Formal definition	***Class::DefinedOperations(): Set(Operation)*** *post: result = operation_list*
Comments	

Name	***AvailableOperations***
Informal definition	*Number of Operations that may be applied to instances of the Class.*
Formal definition	***Class::AvailableOperations(): Set(Operation)*** *post: result = NewOperations()-> union (InheritedOperations())*
Comments	*The following invariant could be stated in alternative:* *post: result = DefinedOperations()-> union (InheritedOperations())*

Name	***NewAttributes***
Informal definition	*Attributes defined in the Class that are not overriding inherited ones.*
Formal definition	***Class::NewAttributes(): Set(Attribute)*** *post: result = DefinedAttributes() – InheritedAttributes()*
Comments	

Name	***InheritedAttributes***
Informal definition	*Number of inherited Attributes that are not overridden by locally defined ones.*

Formal definition	***Class::InheritedAttributes(): Set(Attribute)***
	post: result = Ascendants()->iterate(elem: Class;
	acc: Set(Attribute)= Set{ } | acc-> union (elem.attribute_list))

Comments	

Name	***OverriddenAttributes***
Informal definition	*Number of Attributes defined in the Class that override inherited ones = number of inherited Attributes that are overridden by locally defined ones.*
Formal definition	***Class::OverriddenAttributes(): Set(Attribute)*** *post: result = DefinedAttributes()->intersection(InheritedAttributes())*
Comments	

Name	***DefinedAttributes***
Informal definition	*Number of Attributes defined in the Class.*
Formal definition	***Class::DefinedAttributes(): Set(Attribute)*** *post: result = attribute_list*
Comments	

Name	***AvailableAttributes***
Informal definition	*Number of Attributes that may be applied to instances of the Class.*
Formal definition	***Class::AvailableAttributes(): Set(Attribute)*** *post: result = NewAttributes()-> union (InheritedAttributes())*
Comments	*The following invariant could be stated in alternative:* *post: result = DefinedAttributes()-> union (InheritedAttributes())*

Class - Level Counting Functions

Name	***CC – Children Count***
Informal definition	*Number of directly derived Classes.*
Formal definition	***Class::CC(): Integer*** *post: result = Children()->size()*
Comments	

Name	***DC – Descendants Count***
Informal definition	*Number of all derived Classes (either directly or indirectly).*
Formal definition	***Class::DC(): Integer*** *post: result = Descendants()->size()*
Comments	

Name	***PC – Parents Count***
Informal definition	*Number of Classes from which the current Class derives directly.*
Formal definition	***Class::PC(): Integer*** *post: result = Parents()->size()*
Comments	

Name	***AC – Ascendants Count***
Informal definition	*Number of Classes from which the current Class derives directly or indirectly.*
Formal definition	***Class::AC(): Integer*** *post: result = Ascendants()->size()*
Comments	

Name	***ON – Operations New***
Informal definition	*Number of Operations defined in the Class that are not overriding inherited ones.*
Formal definition	***Class::ON(): Integer*** *post: result = NewOperations()->size()*
Comments	

Name	***OI – Operations Inherited***
Informal definition	*Number of inherited Operations that are not overridden by locally defined ones.*
Formal definition	***Class::OI(): Integer*** *post: result = InheritedOperations()->size()*
Comments	

Name	***OO – Operations Overridden***
Informal definition	*Number of inherited Operations that are overridden by locally defined ones = Number of Operations defined in the Class that override inherited ones.*
Formal definition	***Class::OO(): Integer*** *post: result = OverriddenOperations()->size()*
Comments	

Name	***OD – Operations Defined***
Informal definition	*Number of Operations defined in the Class.*
Formal definition	***Class::OD(): Integer*** *post: result = DefinedOperations()->size()*
Comments	

Name	***OA – Operations Available***
Informal definition	*Number of Operations that may be applied to instances of the Class.*
Formal definition	***Class::OA(): Integer*** *post: result = AvailableOperations()->size()*
Comments	

Name	***AN – Attributes New***
Informal definition	*Number of Attributes defined in the Class that are not overriding inherited ones.*
Formal definition	***Class::AN(): Integer*** *post: result = NewAttributes()->size()*
Comments	

Name	***AI – Attributes Inherited***
Informal definition	*Number of inherited Attributes that are not overridden by locally defined ones.*
Formal definition	***Class::AI(): Integer*** *post: result = InheritedAttributes()->size()*
Comments	

Name	***AO – Attributes Overridden***
Informal definition	*Number of Attributes defined in the Class that override inherited ones = number of inherited Attributes that are overridden by locally defined ones.*
Formal definition	***Class::AO(): Integer*** *post: result = OverriddenAttributes()->size()*
Comments	

Name	***AD – Attributes Defined***
Informal definition	*Number of Attributes defined in the Class.*
Formal definition	***Class::AD(): Integer*** *post: result = DefinedAttributes()->size()*
Comments	

Name	***AA – Attributes Available***
Informal definition	*Number of Attributes that may be associated to instances of the Class.*
Formal definition	***Class::AA(): Integer*** *post: result = AvailableAttributes()->size()*
Comments	

Specification - Level Set Functions

Name	***AllClasses***
Informal definition	*Set of all Classes belonging to the current Specification.*
Formal definition	***Specification::AllClasses(): Set(Class)*** *post: result= module_list.class_list*
Comments	

Name	***BaseClasses***
Informal definition	*Set of base Classes of Classes from the current Specification that belong to the given "s" Specification.*
Formal definition	***Specification::BaseClasses(s: Specification): Set(Class)*** *post: result= AllClasses().inherits_from->select(IsInternal(s))->asSet()*
Comments	

Name	***SupplierClasses***
Informal definition	*Set of supplier Classes of Classes from the current Specification that belong to the given "s" Specification (excludes inheritance).*
Formal definition	***Specification::SupplierClasses(s: Specification): Set(Class)*** *post: result = AllClasses()->iterate(elem:Class; acc: Set(Class)=Set{ } |* *acc union (elem.CoupledClasses()-> select(IsInternal(s))))*
Comments	

Name	***RelatedClasses***
Informal definition	*Set of Classes from the "s" Specification that are either base or supplier Classes from the ones of the current Specification.*
Formal definition	***Specification::RelatedClasses(s: Specification): Set(Class)*** *post: result = BaseClasses(s) union SupplierClasses(s)*
Comments	

Specification-level counting functions

Name	***TC – Total Classes***
Informal definition	*Total number of Classes in the Specification.*
Formal definition	***Specification::TC(): Integer*** *post: result = AllClasses()->size()*
Comments	*Although, in the general case, the result of navigating two associations is a Bag, here we can guarantee that the result is like a Set since the same Class cannot belong to distinct modules.*

Name	***TON – Total Operations New***
Informal definition	*Total number of new Operations in the Specification.*
Formal definition	***Specification::TON(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.ON())*
Comments	

Name	***TOO – Total Operations Overridden***
Informal definition	*Total number of overridden Operations in the Specification.*
Formal definition	***Specification::TOO(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.OO())*
Comments	

Name	*TOD - Total Operations Defined*
Informal definition	*Total number of defined Operations in the Specification.*
Formal definition	***Specification::TOD(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.OD())*
Comments	

Name	***TOI – Total Operations Inherited***
Informal definition	*Total number of inherited Operations in the Specification.*
Formal definition	***Specification::TOI(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.OI())*
Comments	

Name	***TOA – Total Operations Available***
Informal definition	*Total number of available Operations in the Specification.*
Formal definition	***Specification::TOA(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.OA())*
Comments	

Name	***TAN – Total Attributes New***
Informal definition	*Total number of new Attributes in the Specification.*
Formal definition	***Specification::TAN(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.AN())*
Comments	

Name	***TAO – Total Attributes Overridden***
Informal definition	*Total number of overridden Attributes in the Specification.*
Formal definition	***Specification::TAO(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.AO())*
Comments	

Name	***TAD – Total Attributes Defined***
Informal definition	*Total number of defined Attributes in the Specification.*
Formal definition	***Specification::TAD(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.AD())*
Comments	

Name	***TAI – Total Attributes Inherited***
Informal definition	*Total number of Attributes inherited in the Specification.*
Formal definition	***Specification::TAI(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.AI())*
Comments	

Name	***TAA – Total Attributes Available***
Informal definition	*Total number of available Attributes in the Specification.*
Formal definition	***Specification::TAA(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.AA())*
Comments	

Name	***IL – Inheritance Links***
Informal definition	*Total number of inheritance relations where the derived Classes belongs to the current Specification and the base one belongs to the given “s” Specification.*
Formal definition	***Specification::IL(s:Specification): Integer*** *post: result= AllClasses().Parents()-> select(IsInternal(s))->size()*
Comments	*Notice that IL(s) <= TIL()*

Name	***TIL – Total Inheritance Links***
Informal definition	*Total number of inheritance relations where the derived Classes belongs to the current Specification.*
Formal definition	***Specification::TIL(): Integer*** *post: result = AllClasses()->iterate(elem: Class; acc: Integer = 0 |* *acc + elem.PC())*
Comments	*Alternative post-condition: result = AllClasses().inherits_from->size()*

Name	***CL – Coupling Links***
Informal definition	*Total number of coupling relations where the client Class belongs to the current Specification and the supplier Class belongs to the given "s" Specification (excludes inheritance).*
Formal definition	***Specification::CL(s: Specification): Integer*** *post: result =self.SupplierClasses(s)->size()*
Comments	

Name	***TCL – Total Coupling Links***
Informal definition	*Total number of distinct coupling relations where the client Class belongs to the current Specification (excludes inheritance).*
Formal definition	***Specification::TCL(): Integer*** *post: result = AllClasses()-> iterate(elem: Class; acc: Integer = 0 |* *acc + elem.CoupledClasses()->size())*
Comments	

APPENDIX D – EXAMPLES

This appendix is divided into two sections. Both of them make use of the Royal and Loyal example presented on chapter 3. The first section shows how the script developed by the group QUASAR converts one UML diagram into a textual notation, as explained in section 6.1. The second part shows the results of some functions in FLAME, and of some metrics extracted over the Royal and Loyal example.

D.1 The Converted File

In this section we show how the classes, attributes and operations are mapped to a textual notation, in order to enable OCL constraints to be applied over the model.

Figure D.1 reproduces the Royal and Loyal class diagram presented on chapter 3.

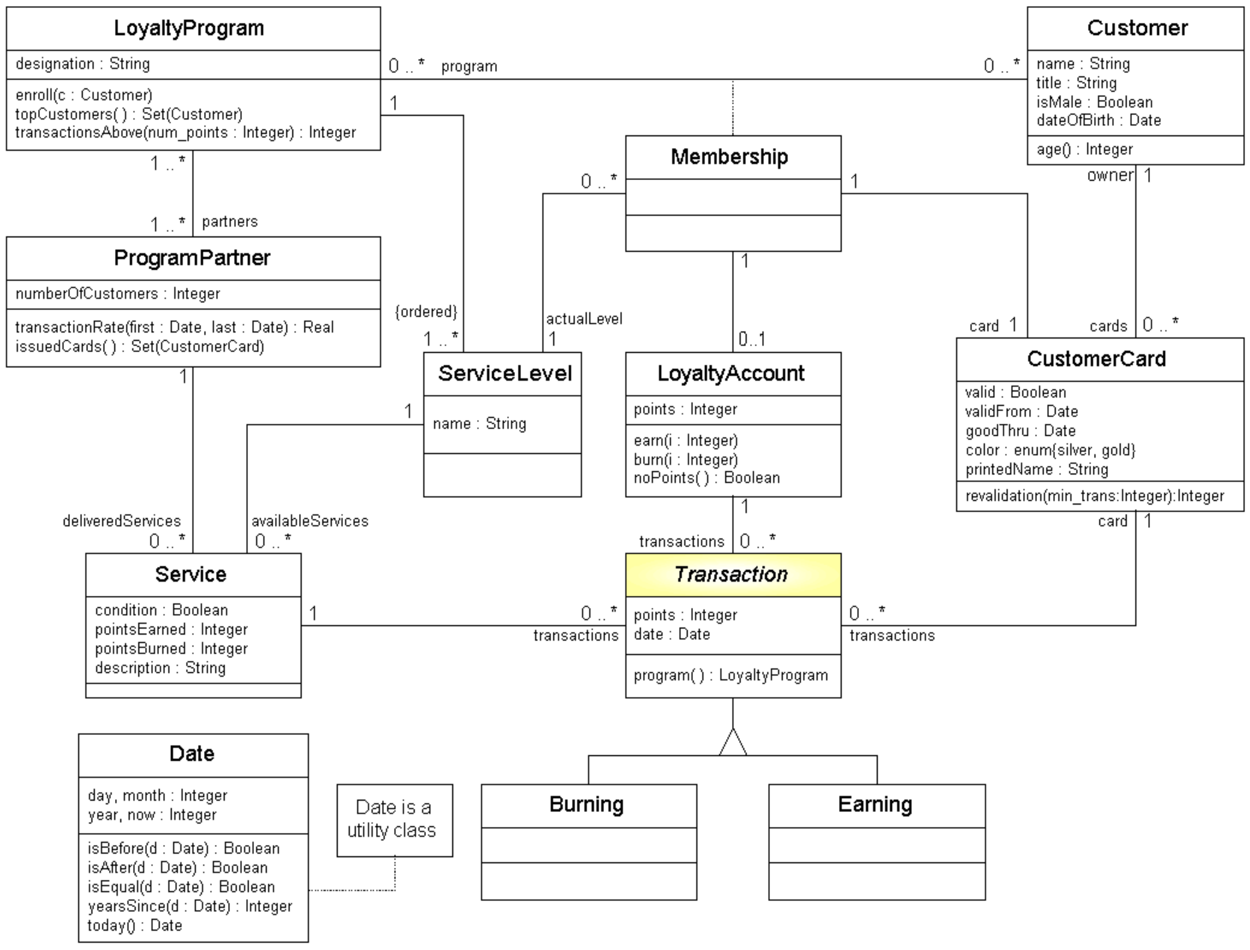

Figure D.1 – *A Reproduction of the Royal and Loyal Class Diagram*

Consider all the classes belong to a Package called RoyalLoyal. We start by creating such Package. Later, consider the creation of class Transaction and its Features. For that we consider all its Attributes, as well as the *program* Operation are private. These elements of the diagram above are mapped as follows.

Package Structure

First, an object of the type Package is created. After, it is named as RoyalLoyal.

```
!create RoyalLoyal_Package: Package
!set RoyalLoyal_Package.name = 'RoyalLoyal'
```

Class Structure

The class structure starts by defining that we are creating an object of the meta-class Class. In the sequence, the properties of Transaction_Class are defined. Finally, Transaction_Class is inserted into the Namespace of the RoyalLoyal Package (which must be already created).

```
!create Transaction_Class: Class

!set Transaction_Class.name ='Transaction'
!set Transaction_Class.isRoot = true
!set Transaction_Class.isLeaf = false
!set Transaction_Class.isAbstract = true

!insert (RoyalLoyal_Package, Transaction_Class) into Namespace_ModelElement
```

Class Attributes

The Attributes follow the same idea. The objects corresponding to the UML meta-model class Attribute are created and their properties are set. Finally the Features (Transaction_points_Attribute and Transaction_date_Attribute) are inserted into the Class (Transaction_Class). Remember all the Features are automatically named with one prefix to indicate the Class from where they belong. This is used to solve *name clashes* [38].

```
!create Transaction_points_Attribute: Attribute

!set Transaction_points_Attribute.name ='points'
!set Transaction_points_Attribute.visibility = #private

!insert (Transaction_Class, Transaction_points_Attribute)
         into Classifier_Feature
!insert (Transaction_points_Attribute, Integer_Class)
         into StructuralFeature_Classifier

!create Transaction_date_Attribute: Attribute
```

[38] See the "*Discussion*" on chapter 5.

```
!set Transaction_date_Attribute.name ='date'
!set Transaction_date_Attribute.visibility = #private

!insert (Transaction_Class, Transaction_date_Attribute)
         into Classifier_Feature
!insert (Transaction_date_Attribute, Date_Class)
         into StructuralFeature_Classifier
```

Class Operation

The Operations are similar to the Attributes and they also have a prefix.

```
!create Transaction_program_Operation: Operation

!set Transaction_program_Operation.name ='program'

!insert (Transaction_Class, Transaction_program_Operation)
         into Classifier_Feature
!insert (Transaction_program_Operation, LoyaltyProgram_Class)
         into BehavioralFeature_Classifier
```

Following these conventions (of creation, setting the properties' values and insertion into the associations), all the objects in the model are created (packages, classes, attributes, operations, associations, association ends, parameters, etc).

D.2 Examples of Quantitative Analysis

The tables below extract the values of both the metrics and functions in FLAME for some specific contexts. For the evaluation, consider all the classes are in the RoyalLoyal Package. All the Attributes and Operations are private.

Context	Acronym	Name	Result[39]
Transaction_points_ Attribute	AUN	Attribute Use Number	`0`
Transaction_Class	FCV	Feature to Classifier Visibility (Transacton_points_Attribute)	`true`
Transaction_Class	FCV	Feature to Classifier Visibility (Customer_name_Attribute)	`false`
Transaction_Class	—	Coupled Classes	`{CustomerCard_Class, LoyaltyAccount_Class,LoyaltyProgram_Class,Service_Class}`
Transaction_Class	—	Feature To Attribute Set (Transaction_Class.allFeatures( ))	`{Transaction_date_Attribute, Transaction_points_ Attribute}`
Transaction_Class	—	Feature To Operation Set	`{Transaction_program_ Operation}`

[39] Applying the Function <<Name>> into <<context>>

		(Transaction_Class.allFeatures( ))	
Transaction_Class	—	New Features	{Transaction_date_Attribute, Transaction_points_ Attribute, Transaction_ program_Operation}
Transaction_Class	—	Defined Features	{Transaction_date_Attribute, Transaction_points_ Attribute, Transaction_ program_Operation}
Transaction_Class	—	Directly Inherited Features	{ }
Transaction_Class	—	All Inherited Features	{ }
Transaction_Class	—	Overridden Features	{ }
Transaction_Class	—	All Features	{Transaction_date_Attribute, Transaction_points_ Attribute, Transaction_ program_Operation}
Transaction_Class	—	New Attributes	{Transaction_date_Attribute, Transaction_points_ Attribute}
Transaction_Class	—	Defined Attributes	{Transaction_date_Attribute, Transaction_points_ Attribute}
Transaction_Class	—	Directly Inherited Attributes	{ }
Transaction_Class	—	All Inherited Attributes	{ }
Transaction_Class	—	Overridden Attributes	{ }
Transaction_Class	—	All Attributes	{Transaction_date_Attribute, Transaction_points_ Attribute}
Transaction_Class	—	New Operations	{Transaction_program_ Operation}
Transaction_Class	—	Defined Operations	{Transaction_program_ Operation}
Transaction_Class	—	Directly Inherited Operations	{ }
Transaction_Class	—	All Inherited Operations	{ }
Transaction_Class	—	Overridden Operations	{ }
Transaction_Class	—	All Operations	{ }
Transaction_Class	—	All Contents	{ }
Transaction_Class	—	Associations	{Transaction_CustomerCard_ Association,Transaction_ LoyaltyAccount_Association, Transaction_Service_ Association}
Transaction_Class	—	All Opposite Association Ends	{Transaction_CustomerCard_ CustomerCard_AssociationEnd, Transaction_LoyaltyAccount_ LoyaltyAccount_Association End,Transaction_Service_ Service_AssociationEnd}
Transaction_Class	—	Opposite Association Ends	{Transaction_CustomerCard_ CustomerCard_AssociationEnd, Transaction_LoyaltyAccount_ LoyaltyAccount_Association End,Transaction_Service_ Service_AssociationEnd}

Transaction_Class	NAN	New Attributes Number	`2`
Transaction_Class	DAN	Defined Attributes Number	`2`
Transaction_Class	IAN	Inherited Attributes Number	`0`
Transaction_Class	OAN	Overridden Attributes Number	`0`
Transaction_Class	AAN	Available Attributes Number	`2`
Transaction_Class	NON	New Operations Number	`1`
Transaction_Class	DON	Defined Operations Number	`1`
Transaction_Class	ION	Inherited Operations Number	`0`
Transaction_Class	OON	Overridden Operations Number	`0`
Transaction_Class	AON	Available Operations Number	`1`
Transaction_Class	PRIAN	Private Attributes Number	`2`
Transaction_Class	PROAN	Protected Attributes Number	`0`
Transaction_Class	PUBAN	Public Attributes Number	`0`
Transaction_Class	PRION	Private Operations Number	`1`
Transaction_Class	PROON	Protected Operations Number	`0`
Transaction_Class	PUBON	Public Operations Number	`0`
Transaction_points_ Attribute	FUN	Feature Use Number	`0`
Transaction_Class	—	Is Root	`true`
Transaction_Class	—	Is Leaf	`false`
Transaction_Class	—	Children	`{Burning_Class, Earning_Class}`
Transaction_Class	—	Descendants	`{Burning_Class, Earning_Class}`
Transaction_Class	—	Parents	`{ }`
Transaction_Class	—	Ascendants	`{ }`
Transaction_Class	CHIN	Children Number	`2`
Transaction_Class	DESN	Descendants Number	`2`
Transaction_Class	PARN	Parents Number	`0`
Transaction_Class	ASCN	Ascendants Number	`0`
Transaction_Class	—	Client	`{ }`
Transaction_Class	—	All Clients	`{ }`
Transaction_Class	—	Contents	`{ }`
Transaction_program _Operation	OUN	Operation Use Number	`0`
RoyalLoyal_Package	—	Is Internal (Transaction_Class)	`true`
RoyalLoyal_Package	—	All Classes	`Burning_Class, CustomerCard_Class, Customer_Class, Earning_Class, LoyaltyAccount_Class,`

			LoyaltyProgram_Class, Membership_Class, ProgramPartner_Class, ServiceLevel_Class, Service_Class, Transaction_Class, Date_Class}
RoyalLoyal_Package	—	Internal Base Classes	{Transaction_Class}
RoyalLoyal_Package	—	Base Classes (RoyalLoyal_Package)	{Transaction_Class}
RoyalLoyal_Package	—	Base Classes in Packages (RoyalLoyal_Package)	{Transaction_Class}
RoyalLoyal_Package	—	Internal Supplier Classes	{CustomerCard_Class, Customer_Class, Date_Class, LoyaltyAccount_Class, LoyaltyProgram_Class, Membership_Class, ProgramPartner_Class, ServiceLevel_Class, Service_Class, Transaction_Class}
RoyalLoyal_Package	—	Supplier Classes (RoyalLoyal_Package)	{CustomerCard_Class, Customer_Class, Date_Class, LoyaltyAccount_Class, LoyaltyProgram_Class, Membership_Class, ProgramPartner_Class, ServiceLevel_Class, Service_Class, Transaction_Class}
RoyalLoyal_Package	—	Supplier Classes in Packages (RoyalLoyal_Package)	{CustomerCard_Class, Customer_Class, Date_Class, LoyaltyAccount_Class, LoyaltyProgram_Class, Membership_Class, ProgramPartner_Class, ServiceLevel_Class, Service_Class, Transaction_Class}
RoyalLoyal_Package	—	Related Classes (RoyalLoyal_Package)	{CustomerCard_Class, Customer_Class, Date_Class, LoyaltyAccount_Class, LoyaltyProgram_Class, Membership_Class, ProgramPartner_Class, ServiceLevel_Class, Service_Class, Transaction_Class}
RoyalLoyal_Package	CN	Classes Number	12
RoyalLoyal_Package	PNAN	Package New Attributes Number	23
RoyalLoyal_Package	PDAN	Package Defined Attributes Number	23
RoyalLoyal_Package	PIAN	Package Inherited Attributes	4

		Number	
RoyalLoyal_Package	POAN	Package Overridden Attributes Number	0
RoyalLoyal_Package	PAAN	Package Available Attributes Number	27
RoyalLoyal_Package	PNON	Package New Operations Number	16
RoyalLoyal_Package	PDON	Package Defined Operations Number	16
RoyalLoyal_Package	PION	Package Inherited Operations Number	2
RoyalLoyal_Package	POON	Package Overridden Operations Number	0
RoyalLoyal_Package	PAON	Package Available Operations Number	18
RoyalLoyal_Package	EILN	External Inheritance Links Number (RoyalLoyal_Package)	2
RoyalLoyal_Package	IILN	Internal Inheritance Links Number	2
RoyalLoyal_Package	PILN	Packages Inheritance Links Number (RoyalLoyal_Package)	4
RoyalLoyal_Package	ECLN	External Coupling Links Number (RoyalLoyal_Package)	10
RoyalLoyal_Package	ICLN	Internal Coupling Links Number	10
RoyalLoyal_Package	PCLN	Packages Coupling Links Number	20
RoyalLoyal_Package	AVN	Attribute Visibility Number (Transaction_points_Attribute)	1
RoyalLoyal_Package	OVN	Operation Visibility Number (Transaction_program_Operation)	1
RoyalLoyal_Package	FVN	Feature Visibility Number (Transaction_points_Attribute)	1
RoyalLoyal_Package	APV	Attribute to Package Visibility (Transaction_points_Attribute)	0
RoyalLoyal_Package	OPV	Operation to Package Visibility (Transaction_program_Operation)	0
RoyalLoyal_Package	FPV	Feature to Package Visibility (Transaction_points_Attribute)	0

Table D.1 – *Some Results for Functions in FLAME, applied to the Royal and Loyal Example*

The result values shown above are very simple and, in this case, can be even calculated by hand, because the Royal and Loyal is a small system. However, for the metrics extraction, even for small systems, the analysis is more complicated. Below we present the metrics results for Royal and Loyal, according to the formalization presented on the fields "*Formal Definition*" of chapter 6.

We have also extracted results from bigger system, with around 60 classes, and validated the formalization with the results stored in data base of the QUASAR group.

Context	Group	Acronym	Metric Name	Result
RoyalLoyal_Package	MOOD	AIF	Attribute Inheritance Factor	0.l481481
RoyalLoyal_Package	MOOD	OIF	Operations Inheritance Factor	0.1111111
RoyalLoyal_Package	MOOD2	IIF	Internal Inheritance Factor	1
RoyalLoyal_Package	MOOD	AHF	Attribute Hiding Factor	1
RoyalLoyal_Package	MOOD	OHF	Operations Hiding Factor	1
RoyalLoyal_Package	MOOD2	AHEF	Attributes Hiding Effectiveness Factor	0
RoyalLoyal_Package	MOOD2	OHEF	Operations Hiding Effectiveness Factor	0
RoyalLoyal_Package	MOOD	BPF	Behavioral Polymorphism Factor	0
RoyalLoyal_Package	MOOD2	PPF	Parametric Polymorphism Factor	0
RoyalLoyal_Package	MOOD	CCF	Class Coupling Factor	0.75757576
RoyalLoyal_Package	MOOD	ICF	Internal Coupling Factor	1
RoyalLoyal_Package	MOOD2	EIF	External Inheritance Factor (RoyalLoyal_Package)	1
RoyalLoyal_Package	MOOD2	ECF	External Coupling Factor (RoyalLoyal_Package)	1
RoyalLoyal_Package	MOOD2	PRF	Potential Reuse Factor	*
RoyalLoyal_Package	MOOD2	ARF	Actual Reuse Factor	*
RoyalLoyal_Package	MOOD2	PPF	Parametric Polymorphic Factor	0
RoyalLoyal_Package	MOOD2	REF	Reuse Efficiency Factor	*
Transaction_Class	MOOSE	WMC	Weighted Methods per Class	1
Transaction_Class	MOOSE	DIT	Depth of Inheritance Tree	0
Transaction_Class	MOOSE	NOC	Number of Children	2
Transaction_Class	MOOSE	CBO	Coupling Between Objects	5
Transaction_Class	MOOSE	RFC	Response for a Class	1
Transaction_Class	EMOOSE	MPC	Message Pass Coupling	0
Transaction_Class	EMOOSE	DAC	Data Abstraction Coupling	1
Transaction_Class	EMOOSE	NOM	Number of Methods	1
Transaction_Class	EMOOSE	SIZE 2		3
RoyalLoyal_Package	QMOOD	DSC	Design Size in Classes	12
RoyalLoyal_Package	QMOOD	NOH	Number of Hierarchies	2
RoyalLoyal_Package	QMOOD	NIC	Number of Independent Classes	11
RoyalLoyal_Package	QMOOD	NSI	Number of Single Inheritance	2
RoyalLoyal_Package	QMOOD	NMI	Number of Multiple Inheritance	0
RoyalLoyal_Package	QMOOD	NNC	Number of Internal Classes	2
RoyalLoyal_Package	QMOOD	NAC	Number of Abstract Classes	1
RoyalLoyal_Package	QMOOD	NLC	Number of Leaf Classes	11
RoyalLoyal_Package	QMOOD	ADI	Average Depth of Inheritance	2.32579255

RoyalLoyal_Package	QMOOD	AWI	Average Width of Inheritance	0.16666666
RoyalLoyal_Package	QMOOD	ANA	Average Number of Ancestors	0.83333333
Transaction_Class	QMOOD	MFM	Measure of Functional Modularity	0.33333333
Transaction_Class	QMOOD	MFA	Measure of Functional Abstraction	0
Transaction_Class	QMOOD	MAA	Measure of Attribute Abstraction	0
Transaction_Class	QMOOD	MAT	Measure of Abstraction	0
Transaction_Class	QMOOD	MOA	Measure of Aggregation	1
Transaction_Class	QMOOD	MOS	Measure of Association	1
Transaction_Class	QMOOD	MRM	Modeled Relationship Measure	2
Transaction_Class	QMOOD	DAM	Data Access Metric	1
Transaction_Class	QMOOD	OAM	Operation Access Metric	0
Transaction_Class	QMOOD	MAM	Member Access Metric	0
Transaction_Class	QMOOD	DOI	Depth of Inheritance	0
Transaction_Class	QMOOD	NOC	Number of Children	2
Transaction_Class	QMOOD	NOA	Number of Ancestors	0
Transaction_Class	QMOOD	NOM	Number of Methods	1
Transaction_Class	QMOOD	CIS	Class Interface Size	0
Transaction_Class	QMOOD	NOO	Number of Overloaded Operators	0
Transaction_Class	QMOOD	NPT	Number of Unique Parameter Types	0
Transaction_Class	QMOOD	NPM	Number of Parameters per Method	0
Transaction_Class	QMOOD	NOD	Number of Attributes	2
Transaction_Class	QMOOD	NAD	Number of Abstract Data Types	1
Transaction_Class	QMOOD	NPA	Number of Public Attributes	0
Transaction_Class	QMOOD	CSM	Class Size Metric	3
Transaction_Class	QMOOD	CAM	Cohesion Among Methods of Class	*
Transaction_Class	QMOOD	DCC	Direct Class Coupling	1
Transaction_Class	QMOOD	MCC	Maximum Class Coupling	1
Transaction_Class	QMOOD	DAC	Direct Attribute Based Coupling	1
Transaction_Class	QMOOD	DPC	Direct Parameter Based Coupling	0
Transaction_Class	QMOOD	MPC	Maximum Parameter Based Coupling	0
Transaction_Class	QMOOD	CCD	Class Complexity Based on Data	6
Transaction_Class	QMOOD	CCP	Class Complexity Based on Method Parameters	0
Transaction_Class	QMOOD	CCM	Class Complexity based on Members	6

Table D.2 – *Metrics results for the Royal and Loyal Example*

The metrics results marked with an asterisk can not be calculated for this example, due to one of its intermediate result. For example, one possible reason for this is a division by zero.

In the Royal and Loyal example, the metrics related with more than one package are calculated using only the Royal and Loyal package. However, we tested such metrics with systems that are composed of more than one package.

REFERENCES AND BIBLIOGRAPHY

A. Hamie; J. H.; S. K. [1998]. *Modular Semantics for Object-Oriented Models.* Northern Formal Methods Workshop. August, 1998.

Abreu, F. B. [1993]. *Metrics for Object Oriented Software Development.* 3rd International Conference on Software Quality, Lake Tahoe, Nevada, EUA, pages 67-75, October 4th - 6th.

Abreu, F. B. [1995a]. *Design Metrics for Object-Oriented Software Systems.* workshop on Quantitative Methods for Object-Oriented Systems Development (ECOOP'95), Aarhus, Denmark, August 7th - 11th.

Abreu, F. B. [1995b]. *Quantitative Methods for Object-Oriented Systems.* 7th ERCIM Workshop on Object Oriented Databases, Lisbon, Portugal, May.

Abreu, F. B. [1998]. *The MOOD2 Metrics Set (in Portuguese).* R7/98, INESC, Grupo de Engenharia de Software.

Abreu, F. B.; Carapuça, R. [1994]. *Object-Oriented Software Engineering: Measuring and Controlling the Development Process.* 4th International Conference on Software Quality, McLean, Virginia, EUA, 3rd-5th October.

Abreu, F. B.; Cuche, J. S. [1998]. *Collecting and Analyzing the MOOD2 Metrics.* Workshop on Object-Oriented Product Metrics for Software Quality Assessment (ECOOP'98), Brussels, Belgium, pages 258-260, July 21st.

Abreu, F. B.; Ochoa, L. M.; Goulão, M. A. [1997]. *The GOODLY Design Language for MOOD Metrics Collection.* R16/97, INESC, Grupo de Engenharia de Software.

Abreu, F. B.; Ochoa, L. M.; Goulão, M. A. [1999]. *The GOODLY Design Language for MOOD2 Metrics Collection.* Workshop on Quantitative Approaches in Object-Oriented Software Engineering (ECOOP'99), Lisbon, Portugal, June 15th.

Abreu, F. B. [2001]. *Using OCL to Formalize Object Oriented Metrics Definitions.* ES007/2001, INESC, Grupo de Engenharia de Software.

Abreu, F. B.; Melo, W. L. [1996]. *Evaluating the Impact of Object-Oriented Design on Software Quality.* 3rd International Software Metrics Symposium (Metrics'96), Berlin, Germany, March.

Abreu, F. B.; Tribolet, J. S.; Guerreiro, P. D. [2001]. *Object-Oriented Software Engineering: A Quantitative Approach (in Portuguese).* PhD Thesis, Faculty of Sciences and Technology - Universidade Nova de Lisboa, Lisbon, 282 pages, December.

Albrecht, A. J. [1979]. *Measuring Applications Development Productivity.* IBM Applications Development Division Joint SHARE/GUIDE Symposium, Monterey, CA, EUA, pages 83-92.

ANSI/IEEE729 [1990]. *Glossary of Software Engineering Terminology.* American National Standards Institute / Institute of Electrical and Electronics Engineers, New York, EUA.

Araújo, J.; Sawyer, P. [1998]. *Integrating Object-Oriented Analysis and Formal Specification.* Journal of Brazilian Computer Society.

Archer, C.; Stinson, M. [1995]. *Object-Oriented Software Measures.* CMU/SEI-95-TR-002, Carnegie Mellon University, Pittsburgh, PA, EUA, Software Engineering Institute.

Bansiya, J.; Davis, C. [1997a]. *Automated Metrics and Object-Oriented Development.* Dr. Dobbs Journal, pages 42-48.

Bansiya, J.; Davis, C. [1997b]. *An Object-Oriented Design Quality Assessment Model.* University of Alabama, EUA.

Baroni, A. L.; Abreu, F. B. [2002]. *Formalizing Object-Oriented Design Metrics upon the UML Meta-Model.* 16th Brazilian Symposium on Software Engineering, Gramado, Brazil, October 14th-18th.

Baroni, A. L.; Bráz, S.; Abreu, F. B. [2002a]. *Using OCL to Formalize Object-Oriented Design Metrics Definitions.* Workshop on Quantitative Approaches in OO Software Engineering (ECOOP'02), Springer-Verlag, June.

Baroni, A. L.; Goulão, M.; Abreu, F. B. [2002b]. *Avoiding the Ambiguity of Quantitative Data Extraction: An Approach to Improve the Quality of Metrics Results.* Workshop of Work in Progress, 28th Euromicro Conference, Dortmund, Germany, September 4th - 6th.

Basili, V.; Briand, L.; Melo, W. L. [1996]. *A Validation of Object-Oriented Design Metrics as Quality Indicators.* IEEE Transactions on Software Engineering, 22(10), pages 751-760.

Basili, V.; Hutchens, D. [1983]. *An Empirical Study of a Complexity Family.* IEEE Transactions on Software Engineering.

Basili, V.; Zelkowitz, M. V. [1977]. *Designing a Software Measurement Experiment.* Software Life-cycle Management Workshop.

Basili, V. R.; Reiter, R. [1979]. *Evaluating Automatable Measures of Software Development.* Workshop on Quantitative Software Models.

Basili, V. R.; Rombach, H. D. [1988]. *The TAME Project: Towards Improvement-Oriented Software Environments.* IEEE Transactions on Software Engineering, 14(6), pages 758-773.

Basili, V. R.; Turner, A. J. [1975]. *Iterative Enhancement: A Practical Technique for Software Development.* IEEE Transactions on Software Engineering, SE-1(4), pages 390-396.

Belady, L. A. [1979]. *Software Complexity.* Workshop on Quantitative Software Models for Reliability. IEEE TH0067-9.

Bieman, J. M. [1991]. *Deriving Measures of Software Reuse in Object-Oriented Systems.* CS91-112, Colorado State University.

Boehm, B. W. [1981]. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, EUA.

BoldSoft, *ModelRun*, BoldSoft MDE AB, Sweden. http://www.boldsoft.com/products/modelrun/index.html

Booch, G. [1994]. *Object Oriented Analysis and Design with Applications.* The Benjamin Cummings Publishing Company Inc, Redwood City, LA, USA.

Booch, G.; Jacobson, I.; Rumbaugh, J. [1997]. *UML Semantics.* Version 1.0, Rational Software Corporation.

Booch, G.; Rumbaugh, J. [1995]. *Unified Method for Object-Oriented Development - Documentation Set.*

Caldiera, G.; Basili, V. R. [1991]. *Identifying and Qualifying Reusable Software Components.* pages 61-70.

Carnegie Mellon University. *SEI* - Software Engineering Institute. http://www.sei.cmu.edu

Chen, J. Y.; Lu, J. F. [1993]. *A New Metric for Object-Oriented Design.* Information and Software Technology, 35(4), pages 232-240.

Chidamber, S. R.; Darcy, D.; Kemerer, C. F. [1998]. *Managerial Use of Metrics for Object Oriented Software: An Exploratory Analysis.* IEEE Transactions on Software Engineering, 28(8).

Chidamber, S. R.; Kemerer, C. F. [1991]. *Towards a Metrics Suite for Object Oriented Design.* OOPSLA'91, pages 197-211.

Chidamber, S. R.; Kemerer, C. F. [1993a]. *A Metrics Suite for Object Oriented Design.* WP No.249, MIT Sloan School of Management, Cambridge, MA, EUA.

Chidamber, S. R.; Kemerer, C. F. [1993b]. *MOOSE: Metrics for Object Oriented Software Engineering.* Workshop on Processes and Metrics for Object-Oriented Software Development (OOPSLA'93)*,* Washington DC, EUA, September.

Christensen, K.; Fitsos, G. P.; Smith, C. P. [1981]. *A Perspective on Software Science.* IBM Systems Journal, 20(4), pages 372-388.

Clark, T.; Warmer, J. [2001]. *Object Modeling with the OCL: The Rationale behind the Object Constraint Language.* Lecture Notes in Computer Science, Springer-Verlag, Berlin, Germany, October.

Constantine, L. L. [1968]. *Segmentation and Design Strategies for Modular programs.* National Symposium on Modular Programming*,* Cambridge, MA, EUA,

Cook, S.; Daniels, J. [1994]. *Designing Object Systems: Object Oriented Modeling with Syntropy.* Prentice Hall, Hemel Hempstead, United Kingdom.

Curtis, B. [1980]. *Measurement and Experimentation in Software Engineering.* Proceedings of the IEEE, 68(9), pages 1144-1157.

Cybernetic Intelligence GmbH, *OCL Compiler*, version 1.5. http://www.cybernetic.org

DeMarco, T. [1982]. *Controlling Software Projects - Management, Measurement and Estimation.* Prentice Hall, Englewood Cliffs, NJ, EUA.

D'Souza, D. F.; Wills, A. C. [1998]. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison Wesley Longman, Massachusetts.

Duke, D.; King, P.; Rose, G. A.; Smith, G. [1991]. *The Object-Z Specification Language.* 91-1, University of Queensland, Australia, Department of Computing Science.

Elixir Technology, *Elixer Java IDE*, version 2.4. http://www.elixirtech.com/

Elliott, J.J; Fenton, N.E.; Linkman, S. [1998]. *Markham: Structure-Based Software Measurement*. Alvey Project SE/069, Department of Electrical Engineering, South Bank, Polytechnic, 103 Borough Road, London, SE1 OAA, United Kingdom.

Eman, K. E.; Drouin, J. N.; Melo, W. L. [1997]. *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination.* IEEE Computer Society Press.

Emerson, T. J. *A Discriminant Metric for Module Comprehension.* 7th International Conference on Software Engineering.

Emerson, T. J. [1984]. *Program Testing, Path Coverage, and the Cohesion Metric.* IEEE COMPSAC, pages 421-431.

Excel Software, *QuickUML.* http://www.excelsoftware.com/index.html

Fenton, N. [1991]. *Software Metrics: A Rigorous Approach*, Chapman & Hall.

Fenton, N. E.; Pfleeger, S. L. [1997]. *Software Metrics: A Rigorous & Practical Approach.* International Thomson Computer Press, London, United Kingdom.

Firesmith, D. G.; Henderson-Sellers, B. [2000]. *The OPEN Process Framework: an Introduction*: Addison-Wesley Publishing Company.

Fitzsimmons, A.; Love, T. [1978]. *A Review and Evaluation of Software Science.* Computing Surveys, 10(1), pages 2-18.

Fowler, M. [1997]. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley Longman.

Gerhart, S.; Craigen, R.; Ralston, T. [1994]. *Case Study: Paris Metro Signaling System.* IEEE Software, 11(1), pages 21-28, January.

Gilb, T. [1977]. *Software Metrics.* Winthrop Publishers Inc, Cambridge, Massachusetts.

Gilb, T. [1988]. *Principles of Software Engineering Management.* Addison-Wesley.

Grady, R. B.; Caswell, D. L. [1987]. *Software Metrics: Establishing a Company-Wide Program.* Prentice-Hall, Englewood Cliffs, NJ, EUA.

Graham, I. M. [1995]. *A Non-Procedural Process Model for Object-Oriented Software Development*. *Report on Object Oriented Analyses and Design*, vol. 5.

H. Bourdeau; B. C. [1995]. *A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering*, pages 799-821.

Hall, A. [March 1996]. *Using Formal Methods to Develop an ATC Information System.* IEEE Software, 13(2), pages 66-76.

Halstead, M. [1977]. *Elements of Software Science.* Elsevier Computer Science Library / North-Holland, New York, EUA.

Halstead, M. H.; Gordon, R. D.; and Elshoff, J. L. [1976]. *On Software Physics and GM's PLI Programs.* GM Research Publication GMR-2175, General Motors Research Laboratories, Warren, MI.

Harrison, R.; Counsell, S. J.; Nithi, R. V. [1998]. *An Evaluation of the MOOD Set of Object-Oriented Software Metrics.* IEEE Transactions on Software Engineering, 24(6), pages 491-496.

Harrison, W. [1988]. *Using Software Metrics to Allocate Testing Resources.* Journal of Management Information Systems.

Harrison, W.; Magel, K. [1981]. *A Complexity Measure Based on Nesting Level.* ACM SIGPLAN Notices.

Hecht, M. S. [1977]. *Flow analysis of computer programs.* North-Holland.

Henderson-Sellers, B. [1991]. *A BOOK of Object-Oriented Knowledge*. Sydney, Australia: Prentice Hall PTR.

Henderson-Sellers, B.; Edwards, J.M. [1994]. *BOOK TWO of Object-Oriented Knowledge: the Working Object*. Sydney, Australia: Prentice Hall.

Henderson-Sellers, B. [1996]. *The Mathematical Validity of Software Metrics.* Software Engineering Notes, pages 89-94.

Henderson-Sellers. B; Simons, T.; Younessi, H. [1998]. *The OPEN Toolbox of Techniques*: Addison-Wesley Publishing Company.

Henry, S.; Selig, C. [1990]. *Predicting Source-Code Complexity at the Design Stage.* IEEE Software.

Hoare, C. A. R. [1973]. *Hints on Programming Language Design.* Stanford University Artificial Intelligence memo AIM-224/ STAN-CS-73-403, pages 193-216.

IEEE1061. *Standard for a Software Quality Metrics Methodology.* Institute of Electrical and Electronics Engineers, New York, EUA.

ISO9001. *Quality Systems - Model for Quality Assurance in Design / Development, Production, Installation and Servicing.* Quality Management and Quality Assurance Standards, ISO/IEC.

ISO9126. *Information Technology - Software Product Evaluation - Software Quality Characteristics and Metrics.* International Organization for Standardization, Geneva, Switzerland.

ISO14598. *Software Product Evaluation. Information Technology*, ISO/IEC.

J. Bicarregui; K. L. [1997]. *Towards a Compositional Interpretation of Object Diagrams.* Bird and Meertens, IFIP TC2 Working conference on Algorithmic Languages and Calculi.

Jacobson, I.; Christerson, M.; Johnson, P.; and Övergaard, G. [1992]. *Object-Oriented Software Engineering- A Use Case Driven Approach.* Addison-Wesley / ACM Press, Reading, MA, USA / Wokingham, England.

Jensen, R.; Bartley, J. [1991]. *Parametric Estimation of Programming Effort: An Object-Oriented Model.* Journal of Systems and Software, 15(2), pages 107-114.

Jones, C. [1978]. *Measuring Programming Quality and Productivity.* IBM Systems Journal, 17(1).

Jones, C. B. [1990]. *Systematic Software Development Using VDM.* Prentice-Hall International, Hemel Hempstead, United Kingdom.

Kafura, D.; Henry, S. [1981]. *Software Quality Metrics Based on Interconnectivity.* Journal of Systems and Software, 2(2), pages 121-131.

Karner, G. [1993]. *Metrics for Objectory.* Master Thesis, Linkuping University, Linkuping.

Kelvin, W. T. [1891-1894]. *Popular Lectures and Addresses.*

Kent, A. E. [1999]. *Core Meta-Modeling Semantics of UML: The pUML Approach.* UML'99, October.

Koch, G. [1993]. *Process Assessment: The BOOTSTRAP Approach*. Butterworth-Heinemann Ltd.

Kolence, K. W. *Software Physics.* Datamation.

Konrad, M.; Paulk, M; Graydon, A. [1995]. *An Overview of SPICE's Model for Process Management*. Proceedings of the 5th International Conference on Software Quality, Texas, EUA.

Laemmel, A.; Shooman, M. *Statistical (Natural) Language Theory and Computer Program Complexity.*

Lake, A.; Cook, C. [1992]. *A Software Complexity Metric for C++.* Annual Oregon Workshop on Software Metrics, Silver Falls, Oregon, EUA, March 22nd - 24th.

Laranjeira, L. A. [1990]. *Software Size Estimation of Object-Oriented Systems.* IEEE Transactions on Software Engineering, 16(5), pages 510-522.

Li, W.; Henry, S. [1993]. *Object-Oriented Metrics that Predict Maintainability.* Journal of Systems and Software, 23(2), pages 111-122.

Li, W.; Henry, S.; Kafura, D.; Schulman, R. [1995]. *Measuring Object-Oriented Design.* JOOP (July / August), pages 48-55.

Longworth, H. D.; Ottenstein, L. M.; and Smith, M. R. [1986]. *The Relationship between Program Complexity and Slice Complexity During Debugging Tasks.* IEEE COMPSAC, pages 383-389, October.

Lorenz, M.; Kidd, J. [1994]. *Object-Oriented Software Metrics: A Practical Guide.* Prentice Hall, Englewood Cliffs, NJ, EUA.

Mansfield, M. [1963]. *Introduction to Topology.* Van Nostrand, Princeton, NJ, EUA.

McCabe, T. [1976]. *A Complexity Measure.* IEEE Transactions on Software Engineering, 2(4), pages 308-320.

McClure, C. L. *A Model for Program Complexity Analysis.* 3rd International Conference on Software Engineering.

METKIT [1993]. METKIT - *Metrics Educational Toolkit. Information and Software Technology*. Volume 35, No. 2, February.

Meyer, B. [1995]. *Beyond Design by Contract: Putting More Formality into Object-Oriented Development.* TOOLS EUROPE, Versailles, France.

Meyer, B. [1997]. *Object-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

Mifflin, H. [2000]. *The American Heritage Dictionary of the English Language.* Houghton Mifflin Company.

Mills, H. D. [1988]. *Software Productivity.* John Wiley & Sons.

Moreira, A.; Clark, R. [1996]. *Adding Rigour to Object-Oriented Analysis.* Software Engineering Journal, 11(5), pages 270-280.

Morris, K. L. [1989]. *Metrics for Object-Oriented Software Development Environments.* Master Thesis, Massachusetts Institute of Technology, Cambridge, MA, EUA

Morris, M. F. *Kolence true or false?*

Muller, B.; Gimnich, R. [1997]. *Planning Year 2000 Transformations Using Standard Tools: An Experience Report.* 1st Euromicro Conference on Software Maintenance and Reengineering, Berlin, Germany, pages 94-100, March 17th - 19th.

Myers, G. [1977]. *An Extension to the Cyclomatic Measure of Program Complexity.* ACM SIGPLAN Notices, 12, pages 61-64, October.

Nasa. *Fifteenth Annual Software Engineering Workshop.*

Nasa. *Measures and Metrics for Software Development.*

Odell, J. [1995]. *Meta-Modeling*. OOPSLA'95 Metamodeling Workshop.

OMG. [1997a]. *UML Proposal to the Object Management Group (version 1.1) in response to the OA&D Task Force's RFP-1.* , Object Management Group, Menlo Park, CA, EUA.

OMG; Rational Software Corporation [1997b]. *UML Semantics (version 1.1).* Object Management Group, Menlo Park, CA, EUA.

OMG; Rational Software Corporation [1999]. *Unified Modeling Language Specification (version 1.3).* Object Management Group.

OMG; Rational Software Corporation [2001]. *UML Notation Guide (version 1.4).* Object Management Group, Menlo Park, CA, EUA.

OMG; Rational Software Corporation [1997]. *Object Constraint Language Specification (version 1.1).* Object Management Group, Menlo Park, CA, EUA.

Oracle Technology Network [2002], *Oracle 9i JDeveloper*, version 9i, April. http://otn.oracle.com/products/jdev/content.html

Ott, L. M.; Thuss, J. J. [1991]. *Sliced Based Metrics for Estimation Cohesion.* CS-91-124, Colorado State University, Fort Collins, Colorado, EUA, Computer Science Department.

Oviedo, E. I. [1980]. *Control Flow, Data Flow and Programmers Complexity.* COMPSAC'80, Chicago, IL, EUA, pages 146-152.

Park, R. E. [1992]. *Software Size Measurement: A Framework for Counting Source Statements.* SEI-92-TR-020, Carnegie-Mellon University, Software Engineering Institute.

Parnas, D. L. [1975]. *The Influence of Software Structure on Reliability.* International Conference on Reliable Software, pages 358-362, April 21st - 23rd.

Parnas, D. L., van Schouwen, J., Kwan, P., and Fouger, S. [1987]. *Evaluation of the Shutdown Software for Darlington.* SDS-1.

Paulk, M. C.; Curtis, B.; Chrissis, M. B.; Weber, C. V. [1993]. *Capability Maturity Model (Version 1.1).* IEEE Software, pages 18-27.

Piwowarski, P. *A Nesting Complexity Measure.* SIGPLAN Notices.

Poels, G.; Dedene, G. [1996]. *Formal Software Measurement for Object-Oriented Business Models.* 7th European Software Control and Metrics Conference (ESCOM'96), Wilmslow, UK, pages 115-134, May 15th - 17th.

Poels, G.; Dedene, G. [2001]. *Measuring Event-Based Object-Oriented Conceptual Models. L'Object*.

Porter, A. A.; Selby, R. W. [1990]. *Empirically Guided Software Development using Metric-Based Classification Trees.* IEEE Software, 7(2), pages 46-54.

Pressman, R. S. [2000]. *Software Engineering: A Practitioner's Approach (European Adaptation).* McGraw-Hill Book Company.

R. France; J. B.; M. Larrondo-Petrie; M. Shroff. [1997]. *Exploring the Semantics of UML Type Structures with Z.* International Workshop on Formal Methods for Object-Based Distributed Systems.

RADC. [1984]. *Automated Software Design Metrics.* RADC-TR-S4-27, Griffiss Air Force Base, NY, EUA, Rome Air Development Center, Air Force System Command.

Rains, E. [1991]. *Function Points in an ADA Object-Oriented Design?* OOPS Messenger, 2(4).

Rasmussen, R. W. [2000]. *A Framework for the UML Meta Model.* PhD Thesis, University of Bergen, 112 pages.

Rational Software Corporation [1998], *Rational Objectory Case Tool*, version 4.1. http://www.inf.ufsc.br/poo/ine5383/orydemo/ory.htm

Rational Software Corporation [2001], *Rational Rose.* http://www.rational.com/products/rose/index.jsp

Rocacher, D. [1988]. *Metrics Definitions for Smalltalk.* WP9A, Metric Use in Software Engineering (MUSE), Project ESPRIT no. 1257.

Rombach, D. [1990]. *Design Measurement: Some Lessons Learned.* IEEE Software.

Rubey, R. J.; Hartwick, R. D. [1968]. *Quantitative Measurement of Program Quality.* ACM National Computer Conference, pages 671-677.

Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W. [1991]. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, NJ, EUA.

Ruston, H. [1981]. *Software Modeling Studies: The Polynomial Measure of Complexity.* RADC-TR-81-183, Rome Air Development Center, Air Force Systems Command, Griffis Air Force Base, Rome, NY, July 1981., Griffis Air Force Base, Rome, NY, EUA, Rome Air Development Center, Air Force Systems Command.

Schneidewind, N. F. [1977]. *Modularity Considerations in Real Time Operating Structures.* COMPSAC 77, pages 397-403.

Sears, A. [July 1993]. *Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout.* IEEE Transaction on Software Engineering, 19(7), pages 707-719.

SEI. [1995]. *The Capability Maturity Model: Guidelines for Improving the Software Process.* Addison-Wesley, Reading, MA.

Sharble, R. C.; Cohen, S. S. [1993]. *The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods.* ACM SIGSOFT Software Engineering Notes, 18(2), pages 60-73.

Shepperd, M.; Ince, D. [1993]. *Derivation and Validation of Software Metrics.* Clarendon Press, Oxford, United Kingdom.

Silva, A. M. R.; Videira, C. A. E. [2001]. *UML - Case Tools and Methodologies (in Portuguese).* Centro Atlântico, Portugal.

Sommerville, I. [2000]. *Software Engineering.* Addison-Wesley Longman.

Spivey, J. M. [1992]. *The Z Notation: A Reference Manual.* Prentice Hall, Hemel Hempstead, United Kingdom.

Sybase Inc., *PowerDesigner*, version 9.0. http://www.sybase.com/products/enterprisemodeling/powerdesigner

Tang, M. H.; Kao, M. H.; Chen, M. H. [1999a]. *An Empirical Study of Object-Oriented Metrics.* 6th IEEE International Software Metrics Symposium.

Tang, M. H.; Chen, M. H. [2002]. *Measuring OO Design Metrics From UML.* UML2002*,* Dresden, Germany, October.

Tang, M.H.; Chen, M.H.; Kao, M. [1999b]. *Investigating Test Effectiveness on Object-Oriented Software - A Case Study.* 12th Annual International Software Quality Week.

Tegarden, D. P.; Sheetz, S. D.; Monarchi, D. E. [1992]. *Effectiveness of Traditional Software Metrics for Object-Oriented Systems.* 25th Hawaii International Conference on System Sciences, Maui, HI, EUA, 359-368, January.

Troy, D. A.; Zweben, S. H. [1981]. *Measuring the Quality of Structured Designs.* Journal of Systems and Software, 2(2), pages 113-120.

University of Bremen, *USE - A UML-based Specification Environment*, http://dustbin.informatik.uni-bremen.de/projects/USE/

Waldén, K.; Nerson, J. M. [1995]. *Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems.* Prentice Hall, Hemel Hempstead, United Kingdom.

Warmer, J.; Kleppe, A. [1999]. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley Publishing Company.

Weiser, M. [1982]. *Programmers Use Slices When Debugging.* Communications of the ACM, 25(7), pages 446-452.

Weiser, M. [1984]. *Program Slicing.* IEEE Transactions on Software Engineering, SE-10(4), pages 352-357.

Wolverton, R. W. [1974]. *The Cost of Developing Large-Scale Software.*

Xie, T.; Huang, H.; Chen, X.; Mei, H.; Yang, F. [2000]. *Object Oriented Software Quality Evaluation Technology.* Software Quality Evaluation Group, Department of Computer Science & Technology, Peking University, Tokyo, Japan.

Xie, T.; Yuan, W.; Mei, H.; Yang, F. [1999]. *JBOOMT: Jade Bird Object-Oriented Metrics Tool.* Chinese Journal of Electronics (English Version).

Yourdon, E. [1975]. *Modular Programming.* Techniques of Program and Structure and Design, Prentice Hall, pages 93-136.

Yourdon, E. N. [1989]. *Modern Structured Analysis.* Prentice-Hall / Yourdon Press, Englewood Cliffs, NJ, EUA.

Zuse, H. *History of Software Measures.*

Zuse, H.; Bollmann-Sdorra, P. [1989]. *Using Measurement Theory to Describe the Properties and Scales of Static Software Complexity Metrics.* SIGPLAN Notices, 24(8), pages 22-33.

Zuse, H. [1991]. *Software Complexity: Measures and Methods.* Walter DeGruyter Publisher, Berlim / Nova Iorque.

Zuse, H.; Bollmann-Sdorra, P. [1992]. Workshops in Computing: *Measurement Theory and Software Measures*. Proceedings of the BCS-FACS Workshop on Formal Aspects of Measurement, South Bank University, London, May 5, 1991. Springer Verlag London Ltd, London SW19 7JZ, United Kingdom.

Money don't make your world go round
I'm reaching out to a higher ground
To a warm and peaceful place
I can rest my weary face

Life's answers we try to find
Battling inside our minds
Where do we go from here?
Will all my friends be there?

'Cause we're living, we're living in a crazy maze
And we're fighting, we're fighting to rise above the haze
Light's at the end of the tunnel
The journey may be long
There are many theories
Who's right and who's wrong?

The pressure's on, I have to choose
I have nothing to lose
I close my eyes, I take a chance
Now I dance a different dance

What's the key to a happy life?
A healthy mind and lots of spice
Running barefoot through the trees
That's my idea of free

I pack my bags, I'm on my way
Don't know where I'm gonna stay
I'm on the train bound destiny
I can set my spirit free

Des'ree – Crazy Maze