SOFTWARE METRICS DEFINITION LANGUAGE

BY

YASSER ELSAYED SHAABAN

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

January 2008

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS
DHAHRAN 31261, SAUDI ARABIA

COLLEGE OF GRADUATE STUDIES

This thesis, written by Yasser Elsayed Shaaban under the direction of his thesis advisor and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

Thesis Committee

[illegible]
Dr. Mohammad Alshayeb (Chairman)

[illegible] Kanaan Faisal
Dr. Jaralla AlGhamdi (Member)

Mahmoud
Dr. Sabri Mahmoud (Member)

Kanaan Faisal
Dr. Kanaan Faisal
Department Chairman

[illegible]
Dr. Salam Zummo
Dean, College of Graduate Studies

10/2/10
Date

عمادة الدراسات العليا
DEANSHIP OF GRADUATE STUDIES

ACKNOWLEDGMENT

Acknowledgement is due to King Fahd University of Petroleum and Minerals for support of this research.

I wish to express my appreciation and thanks to my advisor: Dr. Mohammad Alshayeb for cooperation and support. I also wish to thank members of my thesis committee: Dr. Jaralla AlGhamdi and Dr. Sabri Mahmoud. Special thanks to all members of the Software Engineering Research Group at KFUPM for their constructive feedback.

TABLE OF CONTENS

ACKNOWLEDGMENT....ii

TABLE OF CONTENS....iii

LIST OF TABLES....vii

LIST OF FIGURES....viii

ABSTRACT....ix

ABSTRACT (ARABIC)....x

CHAPTER 1 INTRODUCTION....1

1.1. PROBLEM DEFINITION....1

1.2. RESEARCH GOALS....3

1.3. RELATED WORK....5

1.4. THESIS STRUCTURE....6

CHAPTER 2 SOFTWARE METRICS DEFINITIONS....7

2.1. COMMON SOFTWARE METRICS SUITES....7

2.1.1. Chidamber and Kemerer Metrics Suite....8

2.1.2. Li's Metrics Suite....10

2.1.3. MOOD Metrics Suite....11

2.1.4. Kim's Metrics Suite....12

2.1.5. Other Object Oriented Metrics....13

2.2. CLASSIFICATION OF Software Metrics....14

2.2.1. Classification based on Paradigm *14*

2.2.2. Classification based on Usage *15*

2.3. Issues in Metrics Definitions... 16

2.3.1. Examples of Ambiguity in Metrics Definitions.. *17*

2.3.2. Sources of Metrics Ambiguity... *18*

2.3.3. Addressing Metrics Ambiguity *18*

2.3.4. Metrics Reusability and Extensibility.. *19*

CHAPTER 3 FORMALIZATION OF METRICS DEFINITIONS...........................20

3.1. Abstracting Metrics Definitions USING INTERMEDIATE Meta-models.......21

3.1.1. Meta-models of Metrics Data.. *21*

3.1.2. Meta-models of Software Artifacts... *22*

3.2. Metrics Definitions Formalization.. 27

3.2.1. Early Attempts... *27*

3.2.2. Metrics Meta-models... *28*

3.2.3. Using the Dagstuhl Middle Meta-model *28*

3.2.4. Using XML in Metrics Definitions *30*

3.3. Comparison Between Metrics Definition Approaches 31

CHAPTER 4 FRAMEWORK FOR REPRESENTATION OF SOFTWARE METRICS DEFINITIONS...34

4.1. Metrics MEasurement PROCESS... 34

4.2. Objectives of the Metrics Measurement Framework.. 35

4.3. Elements of the Metrics Measurement Framework.. 36

4.4. the Metrics Measurement Framework 40

CHAPTER 5 THE SOFTWARE METRICS DEFINITION LANGUAGE 41

5.1. SMDL Concepts 42

5.1.1. Meta-Model Base 42

5.1.2. Variables and Queries 43

5.1.3. Intermediate Storage / Meta-Model Database 43

5.1.4. Deep vs. Progressive Evaluation 43

5.1.5. Invalidation Rules 46

5.1.6. Exceptions and Constraints 47

5.2. SMDL Attibutes 48

5.2.1. Metrics Properties 50

5.2.2. Metrics Computation 51

5.3. SMDL Application Progamming Interface (API) 53

5.3.1. API Model Classes 53

5.3.2. API Built-In Queries 53

5.3.3. XMLMath 53

5.3.4. Meta-Model Evaluation and Initialization 55

5.4. SMDL DEFINITION SCHEME 62

CHAPTER 6 APPLYING SMDL 70

6.1. SAMPLE DEFINITIONS IN SDML 70

6.1.1. Depth of Inheritance 70

6.1.2. Weighted Method per Class (WMC) 71

6.1.3. Response for Class (RFC) 71

6.1.4. Number of Children (NOC) 72

6.1.5. Coupling Between Objects (CBO) *73*

6.1.6. Lack of Cohesion in Methods (LCOM) *73*

6.2. Prototype Implementation 74

6.3. Code Parsing 75

6.3.1. SMDL Parser and Editor *75*

6.3.2. SMDL Calculator *78*

CHAPTER 7 CONCLUSION AND FUTURE WORK 81

7.1. CONCLUSION 81

7.2. CONTRIBUTION 81

7.3. FUTURE WORKS 82

APPENDICES 84

APPENDIX A. THE SOFTWARE METRICS DEFINITION LANGUAGE SCHEMA 85

REFERENCES 91

VITA 96

LIST OF TABLES

Table 1 - List of Size Metrics....................13
Table 2 - Classification of Software Metrics by Fenton [1]....................16
Table 3 – Comparison of Modern Approahces to Formalization of Software Metrics......31
Table 4 – Built-in SMDL Queries based in the DMM Model....................55
Table 5: Selected Metrics for Prototype Implementation....................78

LIST OF FIGURES

Figure 1 - Metrics Taxonomy by Sarker 15
Figure 2 - Design Meta-model described by Lanza and Ducasse [20] 23
Figure 3 - The Dagstuhl Middle Meta-model Entities Hierarchy 25
Figure 4 - Relationships of the DMM Meta-model 26
Figure 5 - Formal Definition of the McCabe Metric using Attribute Grammar 27
Figure 6 – Meta-model database example 37
Figure 7 - Metrics Measurement Framework Architecture 40
Figure 8 – Overview of the main SMDL elements 50
Figure 9 - SMDL Role in the Software Metrics Measurement Framework 52
Figure 10 - Portions of the SMDL schema definition (ver. 1.0) 69
Figure 11 - Screenshot of Jaxe Editing the SMDL Schema 76
Figure 12 Screenshots of Creating an SMDL Document 78
Figure 13 Screenshot of SMDL Metrics Calculator 80

ABSTRACT

FULL NAME OF STUDENT	: YASSER ELSAYED SHAABAN
TITLE OF STUDY	: Software Metrics Definition Language
MAJOR FIELD	: Computer Science
DATE OF DEGREE	: January, 2008

Software metrics are becoming more acceptable measures for software quality assessment. However, there is no standard form of representing metrics definitions, which would be useful for metrics exchange and customization. We propose the Software Metrics Definition Language (SMDL), an XML-based description language for defining software metrics in a precise and reusable form. Metrics definitions in SMDL are based on meta-models extracted from either source code or design artifacts, such as the Dagstuhl Middle Meta-model, with support for various abstraction levels. The language also defines several flexible computation mechanisms such as extended OCL queries and predefined graph operations on the meta-model.

ABSTRACT (ARABIC)

خلاصة الرسالة

اسم الطالب الكامل	:	ياسر السيد شعبان
عنوان الدراسة	:	لغة تعريف متريات البرمجيات
التخصص	:	علوم الحاسوب
تاريخ الشهادة	:	صفر، 1431

قياس البرمجيات أصبح أكثر قبولاً لتقييم جودة البرمجيات. بالرغم من ذلك ليس هناك طريقة موحدة لتمثيل تعريفات المتريات، مما يحد من تبادل وتخصيص المتريات .في هذا البحث نقترح لغة لقياس متريات البرمجيات (SMDL)، مبنية على لغة XML ، لتعريف متريات البرمجيات في شكل دقيق وقابل لإعادة الاستخدام . تعريف المتريات بلغة SMDL مبنية على النماذج الفوقية Meta-model مستخرجة من شفرة المصدر أو التصاميم مثل نموذج Dagstuhl الذي يوفر دعم لمختلف مستويات التجريد. اللغة أيضا تعرف العديد من آليات الحساب المرنة مثل استعلامات OCL المطورة وكذلك عمليات الرسوم البيانية المحددة سلفا على النماذج الفوقية.

درجة الماجستير في العلوم

جامعة الملك فهد للبترول والمعادن
الظهران، المملكة العربية السعودية

التاريخ
صفر - 1431

CHAPTER 1

INTRODUCTION

1.1. PROBLEM DEFINITION

Systematic measurement and metrics collection is an essential activity in engineering disciplines. It serves as part of tracking and maintaining the quality attributes of project deliverables and acts as an aid to managerial decisions. The basic principle is that quality improvements need the guidance of quantitative representations of quality attributes taken with proper measurement. The continuous growth of the software industry makes quality assessment of software products a more crucial issue due to the impact of poorly written software. It is therefore clear that systematic and meaningful quality measurement of software artifacts is expected to see wider adoption over the next few years as the industry recognizes its importance in supporting decision making activities throughout the lifecycles of the projects and its roles in improving the development efficiency. Even though software metrics have existed and been in use since the first compiler (e.g. Lines of Code metrics), their industrial adoption has remained limited due to a number of issues. A critical discussion of these factors can be found in Fenton's book [1].

The top challenge among the challenges of software measurement is how to formally represent the definitions of software metrics definitions. While many metrics seem easy to define verbally, they get ambiguous and unclear when it comes down to the actual implementation, particularly across different programming paradigms and environments. In many cases, several methods to compute a given metric exist that produce similar yet different results. A simple example is the source line of code (SLOC) metric. Appearing fairly simple and straightforward, it is easy to see sources of ambiguity. The reference to "lines of code" could be used to refer to the count of: (a) machine instructions (b) complete language statements (c) textual code lines, or (d) specified list of the programming language keywords and expressions. It could even be worse when the metric is used to compare results of similar metrics computed on different platforms and programming languages. For instance, some languages impose certain form of the lines of code (e.g. in Visual Basic) while other languages do not follow the textual line format and impose end-of-line delimiters, as it is the case with C++, Java and C#.

The problem of ambiguity and lack of consistency across the definitions of software metrics is increasingly becoming more relevant due to the fact that modern software products are often developed with multiple languages in heterogeneous environments. A typical modern enterprise project, for example, uses at least three classes of source languages: a server-side language (e.g. JavaServer Pages, JSP), a client side language (e.g. JavaScript and Adobe Action Script) and a presentation language (e.g. HTML/CSS for Web applications and XAML for Windows). Therefore, it is safe to assume that these different components could use different measurement tools to compute and aggregate their metrics. Even with a single tool, aggregating and maintaining the results' consistency among these becomes essential to the measurement process.

The inconsistency in metrics definitions and methods of measurement raises the issue of the lack of extensibility and interoperability across software metrics tools. In fact, it partially contributes to the slow pace of research in this field. For example, when a new metrics is proposed, taking the task of incorporating and integrating the metric into existing metrics tools becomes a major roadblock. The lack of a proper format to present software metric definitions makes it difficult to accommodate new metrics into the exiting tools in a consistent manner that ensures compatibility and accuracy of measurement. With a comprehensive standardized foundation of these definitions, researching and incorporating new metrics to existing tools could become more seamless and accurate across more environments whenever possible.

Our objective of this research is to address the problem of representing software metrics definitions. The main focus is given to software product metrics, which are directly related to source code and design artifacts with special emphasis on the object-oriented paradigm.

1.2. RESEARCH GOALS

The purpose of this research is to introduce a common way of representing software metrics definition elements. In general, these elements fall into one of the following:

- **Metrics Classification Properties:** this includes metric identity, abbreviation, authority information, abstraction level, scope, classification reference, and computation properties.

- **Metric Calculation:** this includes the steps required calculating the metric values for a given element in the code or the design. Values can be either derived from a meta-model that abstracts the product artifacts, variables of other metrics, or even pre-computed / user-given values. Therefore, it can help in metrics extensibility and reusability.

- **Relationships:** a list of the relationships between a metrics and other dependent metrics, e.g. some metrics can be generalizations or specializations of others in terms of computation. This can also be used to relate metrics from similar category or suite.

- **Visualization:** description of how to present metrics output to the user, e.g. whether they are listed in a table per class/ per package or it can be constructed in a matrix that compares classes/packages to each other. Also, the applicable types of charts and their specifications should be included, e.g. bar-chart series, time-series, histogram, etc…

In addition, this new form of representing software metrics has limited usefulness without enabling extensibility and tools support. For example:

- **Metrics Editor**: for creating and customizing metrics in the standard form.
- **Standard API**: for reading, parsing and processing the metric descriptions.

The following summarizes the main goals of this research:

- Provide a standard form for representing software metrics.
- Support both design and code artifacts.
- Support multiple forms of describing metrics computations.
- Enable extensibility and reusability of metrics definitions.
- Provide a standard way to incorporate metrics into a hierarchy / classification system of metrics.

1.3. RELATED WORK

To date, there is no common agreed-upon form of representing software metrics definitions. However, there are some attempts toward formalizing metrics computations and definitions which are discussed in Chapter 3. While some of these approaches address the issues of metrics computation properly, based on abstract meta-models, they do not address the issues of reusability, extensibility and exchangeability of the metrics definitions. Most of the known approaches have several limitations.

1.4. THESIS STRUCTURE

The rest of this thesis is organized into six chapters. In Chapter 2, we review common software metrics and the elements they have in common, providing a possible taxonomy of software metrics. Next, we review several proposed methods for formalizing software metrics definitions and we compare them, in Chapter 3. In Chapter 4, we propose a framework for software metrics definition representation and we discuss the elements that should be considered. In Chapter 5, we introduce the Software Metrics Definition Language (SMDL) based on the proposed framework, highlighting its main components. Chapter 6 provides some examples on applying SMDL including a discussion for a prototype implementation of the language parser and a tool that computed the metrics according to given SMDL definitions. Finally, Chapter 7 reviews the main contributions and the possible future research toward formalization of software metrics definitions.

CHAPTER 2

SOFTWARE METRICS DEFINITIONS

Software metrics use the principles of measurement theory to formulate quantitative measures of software artifacts' that can be used to induce tractable quality measures that gauge the development progress. In this chapter, we survey some of the well-known software metrics suites and present examples of formal metrics definitions and how they are presented. Another goal of this chapter is to select representative metrics that are used as show cases of the proposed metrics definitions language (see CHAPTER 5). We also discuss methods of software metrics classification and present two classification models. Finally, review some of the challenges in standardizing metrics definitions.

2.1. COMMON SOFTWARE METRICS SUITES

Software metrics that are related to a single topic, measure coherent set of attributes, or were introduced by a certain author are often referred to as suites. One of the early suites was proposed by Troy et al. in 1981 an consisted of a set of 24 measures of modularity, size, complexity, cohesion and coupling [2]. Function Points are also popular metrics which focus on the user requirements rather than the software product. However, since the focus here is on object-oriented systems, only relevant suites to object oriented design and implementations are discussed.

2.1.1. Chidamber and Kemerer Metrics Suite

Chidamber and Kemerer introduced their infamous suite of software metrics for object oriented languages in 1991 [3] and was revised later in 1994 [4]. This suite, commonly referred to as C&K, consists of six metrics that measure some internal attributes and used to measure some external quality attributes of object oriented classes.

Although these metrics were intended to be computable from design artifacts only, some however require at least partial access to the implementation source. One particular example is the Lack of Cohesion (LCOM) metric which relies upon the number of times a field is being accessed through calls of the class's methods.

The following summarizes definitions of the C&K suite [3, 4] :

- **Weighted Methods per Class (WMC)**: represents complexity of a class in terms of the methods it encloses. It is computed as the sum of complexities of all member methods of a given class. Method complexities are often assumed to be equal to one.

- **Depth of Inheritance Tree (DIT):** this metric measures the depth of a class in the inheritance hierarchy. For a given class we count the number of ancestors, which should be related to the complexity of the class since a sub-class in a hierarchy inherits complexity of its ancestors.

- **Number of Children (NOC):** this metric measures the number of immediate descendants or subclasses of a given class. A class with high descendants reflects on its complexity as it increases dependencies and the importance of its role in the preprogram hierarchy.

- **Response for a Class (RFC):** represents the size of the response-set of a given class. The response-set includes all methods in the class plus all methods which are invoked by the class's methods.

- **Lack of Cohesion of Method (LCOM):** this is often referred to as LCOM1, since it was the first in a series of other metrics of the lack of cohesion. C&K define LCOM using two sets (P and Q) based on method's access to class fields. P represents pairs of methods which do not access the same fields. Q represents pairs of methods which share at least one field. LCOM1 is given as:

  LCOM1 = Max (P – Q, 0)

- **Coupling between Objects (CBO)**: represents the number of classes to which a class is coupled. A class is coupled to another class if it accesses variables or methods of that other class.

The metrics suite by Chidamber and Kemerer is considered to be one of the major contributions to the object oriented metrics research. It is also one of the most commonly used and validated metrics suites as indicated by numerous citations to the authors' work;

refer for example to [5, 6] and [7]. More details on the origin and objective of this suite can be found in the original paper [4].

2.1.2. Li’s Metrics Suite

It is worthy to mention that W. another metrics suite in [8]that is both complementing and deviating from the original C&K suite, in order to address ambiguity elements they found in the original definition. For example, Li made a distinction between coupling achieved through message passing and coupling through abstract data types, resulting in two new alternative metrics. He also addressed ambiguity issues of the DIT metric that occur in the case of multiple inheritance, in languages that support such feature. Therefore, the following metrics suite has been introduced by Li [8]:

- **Number of Ancestor Classes (NAC):** represents the total number of ancestors of a class, as an alternative to DIT in the case of multiple inheritance.

- **Number of Descendent Classes (NDC):** represents the total number of descendants of an inherited class, as an alternative to NOC.

- **Number of Local Methods (NLM):** represents the total number of accessible methods from outside the class.

- **Class Method Complexity (CMC):** a generalization of WMC which includes all methods included in the class, whether accessible from other classes or not.

- **Coupling through Abstract Data Type (CTA):** represents the number of classes used in a given class in the form of abstract data types (ADT).

- **Coupling through Message Passing (CTM):** represents the number of classes that invoke or access methods of the given class.

2.1.3. MOOD Metrics Suite

The Metrics for Object Oriented Design (MOOD) suite was introduced by F. B. Abreu in 1995 [9] and it includes six metrics that provide an overview of the design quality of a given object oriented project. Eventually, Abreu attempted to refine and formalize the metrics definitions of his suite using the Object Constraint Language (OCL) which is further discussed in Chapter 3. The metrics suite was later extended with another set of metrics called MOOD2 [10]. The following are the metrics defined in the MOOD metrics suite [9]: (Computation details are omitted here for simplicity, the reader can refer to the source):

- **Method Hiding Factor (MHF) and Attribute Hiding Factor (AHF):** These metrics relate to the quality of encapsulation of a given class. A private method is considered hidden and it can have different degrees (e.g. public / private / protected / package).

- **Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF)**: If a method inherits most of its members (methods and attributes) it is assigned a high MIF or AIF. An independent class has lowest MIF/AIF.

- **Polymorphism Factor (PF):** This metric measures the level of using the override method in inherited classes. It is equal to the ration of overridden methods to the total possible overrides.

- **Coupling Factor (CF):** This metric measures the ratio of already coupled classes to the maximum possible coupling of a given class.

2.1.4. Kim's Metrics Suite

Kim et al. introduced their metrics suite in 1996 [11] which focuses on measuring the complexity of object-oriented programs. This suite partially relies on the C&K metrics and provides a more critical and accurate view of the complexity of software components.

Following is a list of the main metrics defined in the Kim et al. metrics suite with a sample definition:

- **Degree of Reuse (DOR)**, defined as follows:

  DOR(C_i) = sum(k / (t+ tr)), for k=1 to r(C_i), where:

  r(C_i) = reused number of each class C_i in the program (e.g. inherited)

  t = total number of classes in the program

  tr = total sum of all r(Ci)S in the program

- **Degree of Coupling of Inheritance (CBI),** assesses the degree of coupling to which an implementation relies on inherited elements.

- **Degree of Internal Method Complexity (IMC),** based on an effort formula in terms of the number of operators and operands in a given method.
- **Number of classes used in a class (UCL)**, except for its super-classes and subclasses.
- **Number of Send Statements of a Class (MPC).**

2.1.5. <u>Other Object Oriented Metrics</u>

There are several size metrics of object oriented program elements which are considered trivial to compute and therefore do not really form a metrics suite, and hence are not given detailed definitions. Although they might have little value with respect to quality measures, several useful metrics can be derived from them, or in combination with other more sophisticated metrics.

The following table lists some of these size metrics [12].

Table 1 - List of Size Metrics

Metric	**Description**
NCM	number of class methods
NCV	number of class variables
NIM	number of instance methods
NIV	number of instance variables
NMA	number of methods added

NMI	number of methods inherited
NMO	number of methods overridden
NOC	number of children
NOM	number of message sends
NOM	number of local methods
NCM	number of class methods
NCV	number of class variables
NIM	number of instance methods
NPM	number of public methods

2.2. CLASSIFICATION OF SOFTWARE METRICS

Software metrics are often grouped into categories depending on different points of view. Two of the broader goal-oriented categories are: product and process classes of metrics. Product metrics reflect attributes of software product artifacts, whereas process metrics are more concerned with measuring cost and effort as functions of time [1]. The following sections summarize other proposed metrics classification taxonomies.

2.2.1. Classification based on Paradigm

In his survey of software metrics, M. Sarker proposed a taxonomy of metrics based on subject and paradigm. Metrics in this classification are broken down into product and process metrics where the former focuses on the software artifact. Product metrics are further divided based on the programming language paradigm: object-oriented or "traditional." The latter refers to sequential and imperative programs. The following chart illustrates the proposed metrics taxonomy and includes several examples of product metrics [13]:

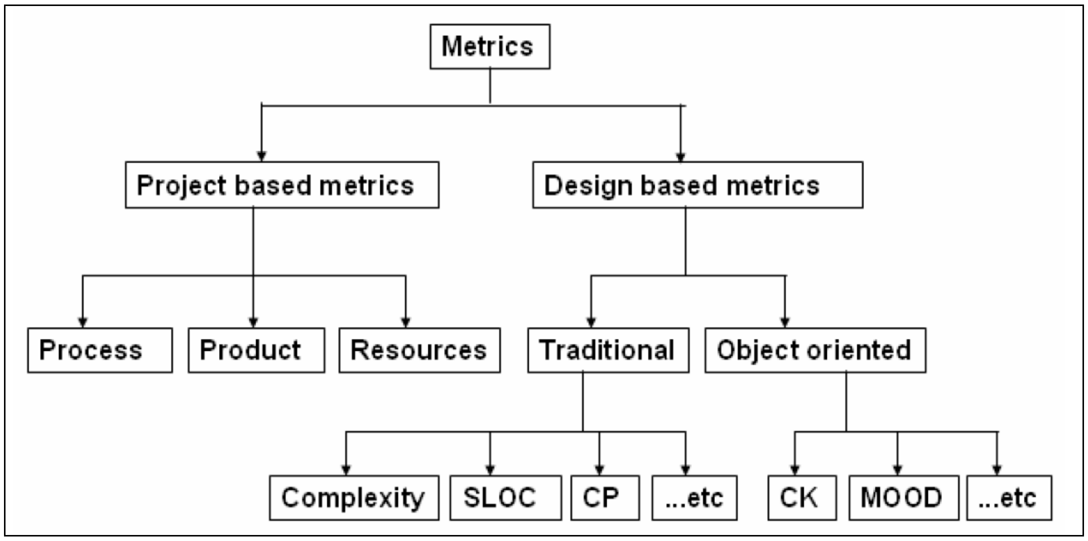
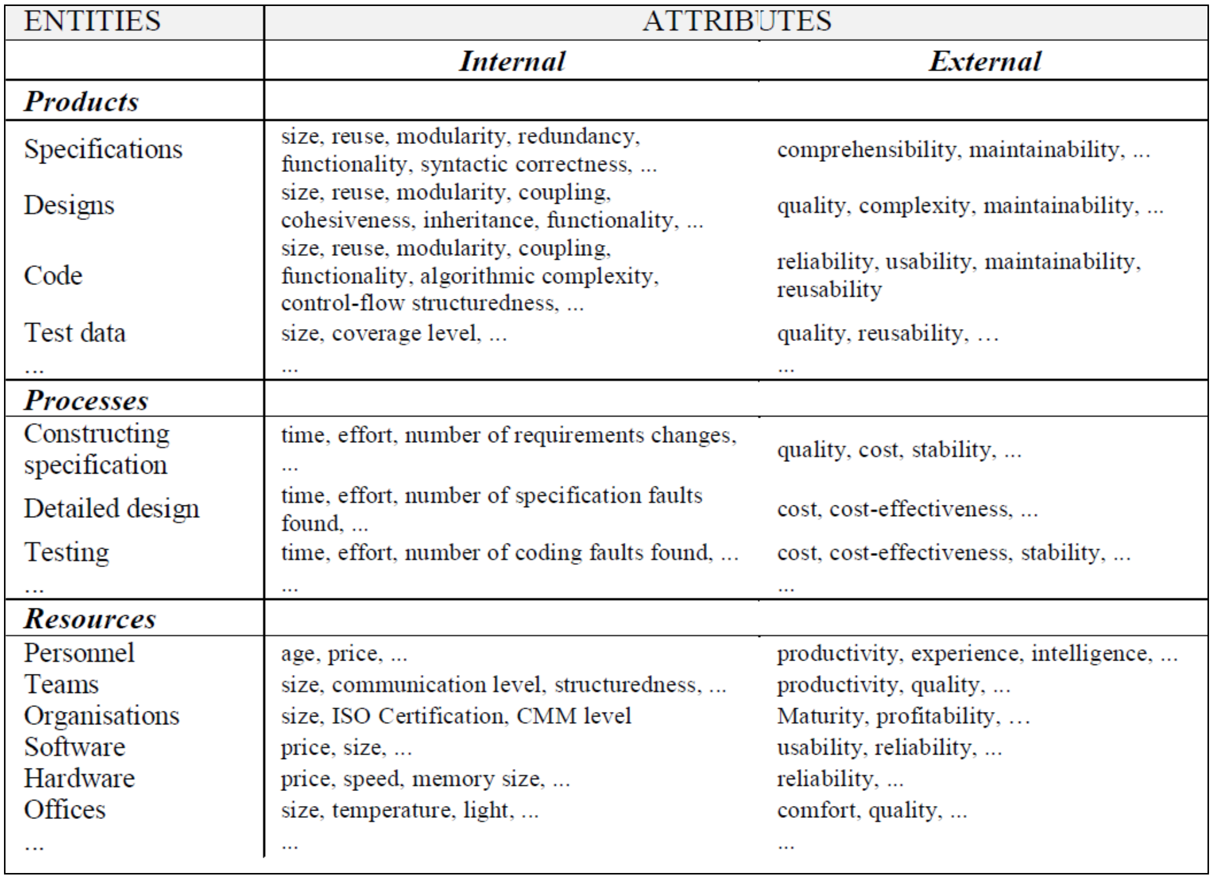


Figure 1 - Metrics Taxonomy by Sarker

2.2.2. Classification based on Usage

Fenton proposed a 2-dimentional classification of software metrics, in his infamous work on the subject, which is based on two different viewpoints: the scope of metrics in the project (e.g. product, process or resources) and the level of visibility they address (which can be internal or external). The following table summarizes the classification scheme along with examples from each category [1]:

Table 2 - Classification of Software Metrics by Fenton [1]



ENTITIES	ATTRIBUTES	
	Internal	***External***
Products		
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, inheritance, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, reusability
Test data	size, coverage level, ...	quality, reusability, ...
...	...	...
Processes		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...	...	...
Resources		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Organisations	size, ISO Certification, CMM level	Maturity, profitability, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...	...	...

2.3. ISSUES IN METRICS DEFINITIONS

Software metrics are defined with the intention of evaluating different software artifacts to produce quality measures and related attributes. In the process of evaluation software quality, metrics are often compared with those of other artifacts similar in class. An important implication here is the assurance of accuracy during the evaluation process which cannot be achieved with ambiguous definitions, especially when evaluating metrics with different tools. Therefore, it is imperative that metrics definition be definitive as much as possible in order to yield consistent results across a heterogeneous set of tools, languages or platforms. Unfortunately, even the simplest metrics easily contain ambiguity at some level, for a variety of reasons, leaving a large room of “flexibility” in the implementation which in turn could result in inconsistent readings by different tools. This is especially a problem since modern software projects tend to use a mixture of different languages in the same product, such as presentation layer languages (JavaScript) and back-end server-side languages (Java or C#). Therefore, the ambiguity of metrics definitions is considered chief among the challenges limiting the wide adoption of metrics research in the software industry. [14] This section provides an overview on the sources of metrics ambiguity, examples of ambiguous definitions, and general approaches to ensure clear metrics definitions.

2.3.1. Examples of Ambiguity in Metrics Definitions

The following are example of ambiguity in some of the C&K metrics:

- Weighted Method per Class (WMC): do we count the inherited method or only newly defined methods? What about method overrides? What is the treatment for overloaded methods? [15]

- Depth of Inheritance (DIT): which route to follow in the case of multiple inheritance? Do we take the longest path or the total number of parents? [8]

- Coupling between Objects (CBO): for method calls to the base class, do we count these into the response set of the base class? The original definition leaves room for interpretation.

2.3.2. <u>Sources of Metrics Ambiguity</u>

The ambiguity in the definition of software metric can be attributed to sources from different levels. First, there is an ambiguity that comes from the definition itself. In such cases the definition does not express how to deal with a general set of different and special cases, leaving a wide room to the interpretation of the tool implementer. This is especially problematic where definitions are intentionally left ambiguous for simplicity. Second, ambiguity caused by a special situation of a specific language when trying to compute the intermediate or meta-model. For example, do we count Enumerations in Java as classes? Enumerations are relatively new to Java (added in 1.5) and the issue was not addressed before. Finally, ambiguity caused by preference in the implementation details. For example, do we count calls to library functions such as *printf()* in the coupling set of a method? The implementer may choose to ignore these calls, for simplicity.

2.3.3. <u>Addressing Metrics Ambiguity</u>

In order to properly eliminate ambiguity in metrics definitions, the sources of ambiguity need to be properly addressed. We can argue that ambiguity at the definition and implementation levels can be eliminated to a long extent by abstracting and binding the \definitions to a well-defined, formally defined meta-model of the measured artifacts. However, there would still be some room left for ambiguity at the layer translating between the programming language and the meta-model, for example. This could still be mitigated with a language-specific translation algorithm, such as the model proposed in[16]. Another approach to eliminating sources of ambiguity can be achieved with a reference-implementation of the metric definition. However, this would be more expensive in terms of effort and still does not resolve language specific features unless the implementation is given in every targeted language.

2.3.4. Metrics Reusability and Extensibility

We can simplify metrics definitions by allowing reuse of existing properly defined metrics which can also help reduce metrics ambiguity. Reusability of metrics is also essential to performance optimizing the implementation as it would allow, for example, a progressive evaluation of the metric values. Additionally, the reusability of metrics definition opens more room for extensibility, thus enabling researchers to derive and examine new metrics more conveniently.

CHAPTER 3

FORMALIZATION OF METRICS DEFINITIONS

To address the common issue of the ambiguity of software metrics definitions, researchers attempted to clarify metrics definitions by putting them in a precise form that can be computed consistently among tools and researchers. This idea of using a standard form to represent metrics definitions can also help to promote reusability of metrics among different tools; making it easier to introduce new metrics and compare different results and variations.

An important element of metrics definitions, which is often overlooked, is enabling reusability of intermediate values or metrics variables. This can be very useful in many aspects such as improving the performance of metrics computation, ease of definition for complicated metrics, and abstracting the definitions of new metrics.

In this chapter, we look at the different proposed approaches to standardize and formalize metrics definitions, and we compare the most recent and mature attempts.

3.1. ABSTRACTING METRICS DEFINITIONS USING INTERMEDIATE META-MODELS

Virtually all modern metrics formalization attempts address the problem of metrics ambiguity by abstracting their definitions to target a higher level of abstraction than the actual source or program. This layer of abstraction acts as the common ground upon which metrics of the same category rely. In this approach, metrics definitions are formulated such that they do not rely on specific language or platform, nor do they become ambiguous by ignoring platform differences. This intermediate layer that contains standard abstractions of language and design artifacts is generally referred to as a meta-model.

Essentially, there are two types of meta-models used in software metrics: models that abstract the artifact to be measures, and models that represent data gathered. The first category is more essential to a metrics definition language since it allows precise formulations of metrics definitions based on an agreed-upon model. The other category captures issues related to metrics data storage, classification and interpretation. It is also important when addressing metrics computation and performance issues. We briefly overview the different approaches related to both categories.

3.1.1. Meta-models of Metrics Data

The first category, meta-models that represent metrics data, is usually represented in the form of a relational database which is considered the most common form of data storage.

Kitchenham et al. In [17], a relational meta-model for representing and storing meta-data needed to compute software metrics and intermediate values of their computation. Another similar relational model can be found in [18] and a discussion on improving the performance of metrics calculations in the relational models is presented in [19]. While these models do not attribute to software metrics definition representation, they constitute a major part of a comprehensive metrics measurement framework, as discussed in Chapter 4.

3.1.2. Meta-models of Software Artifacts

This category includes meta-models used to abstract software artifacts with the purpose of utilizing the intermediate model for formalizing metrics definitions. Lanza and Ducasse introduced a language-independent meta-model for metrics definitions, shown in Figure 2, [20], a language-independent meta-model for metrics definitions was introduced (see Figure 2). This approach limits metric definition to attributes of the meta-model objects and should not be tied to a specific language representation. Another meta-model that captures object-oriented elements, called ODEM, was proposed by Reißing [21], and was later used for metrics definitions by El-Wakil et al. [22]. A third model was proposed by Abreu in [23] which was called the GOODLY. The author aimed to use this model to capture metrics definitions for his metrics suite, MOOD [23].

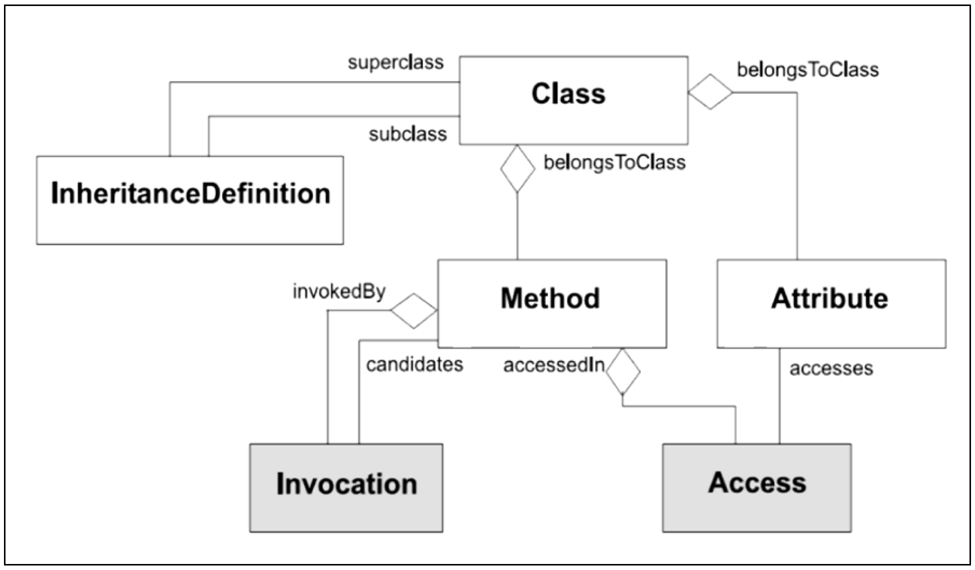


Figure 2 - Design Meta-model described by Lanza and Ducasse [20]

However, a recent study, at time of this research, by McQuillan and Power has found that using these meta-models to be limiting and inadequate to formalizing software metrics definitions. For example, a number of meta-models fail to provide the ability to describe key object-oriented metrics suites such as the C&K suite [24, 25]. The authors also evaluated several specialized, internal meta-models that are used in commercial and open source integrated development environments (IDE’s) of Java. These models are typically used for syntactical and semantical validation against compiler errors, in addition to supporting software refactoring utilities. They include the meta-models used in Eclipse [26] and NetBeans as examples [27]..However, they were found to be limited because they are rather tied to the internal implementation of both IDE’s and can make the task of adapting other programming languages or IDE’s more challenging. Still, the meta-model of Eclipse was later used by McQuillan and Powerthe researchers in [25] as the source of parsing and for computing values of the selected intermediate meta-model in order to streamline the proof-of-concept implementation.

The UML meta-model can also be used for metrics definitions, however, it lacks essential relationships with the source code making it difficult to implement some of the fundamental metrics which require code access. [24] However, this does not prevent attempts to extend the UML meta-model to cover some essential code properties. This UML meta-model is currently used by the commercial tool SDMetrics for representing design metrics [28].

Based on the surveyed literature, the most complete and successful meta-model for abstracting language and design appears to be the Dagstuhl Middle Meta-model (DMM). This meta-model was originally developed for facilitating interoperability across reverse-engineering applications. Hence, it was designed to captures most of code and design relationships. The model is elegantly divided into two parts: entities that represent static elements of the program organized in a class hierarchy with relationships among the entities. The relationships are further organized in an inheritance hierarchy that captures the “is-a” inclusion relationship and vice-versa. For example, classes are associated with their methods using the “IsMethodOf” relationship, and both classes and methods entities are of the type “ModelElements”. The “IsMethodOf” relationship is a subset of the “IsPartOf” relationship. Also, when looking up elements with the relationship “Is-Part-Of”, the relationships “IsMethodOf”, “IsFieldOf” and “IsEnumerationLiteralOf” are also included in the search.

The DMM model can be used to capture in details both object-oriented and procedural languages elements and their relationships [29]. The following figures illustrate the essential components of the DMM meta-model entities and relationship.

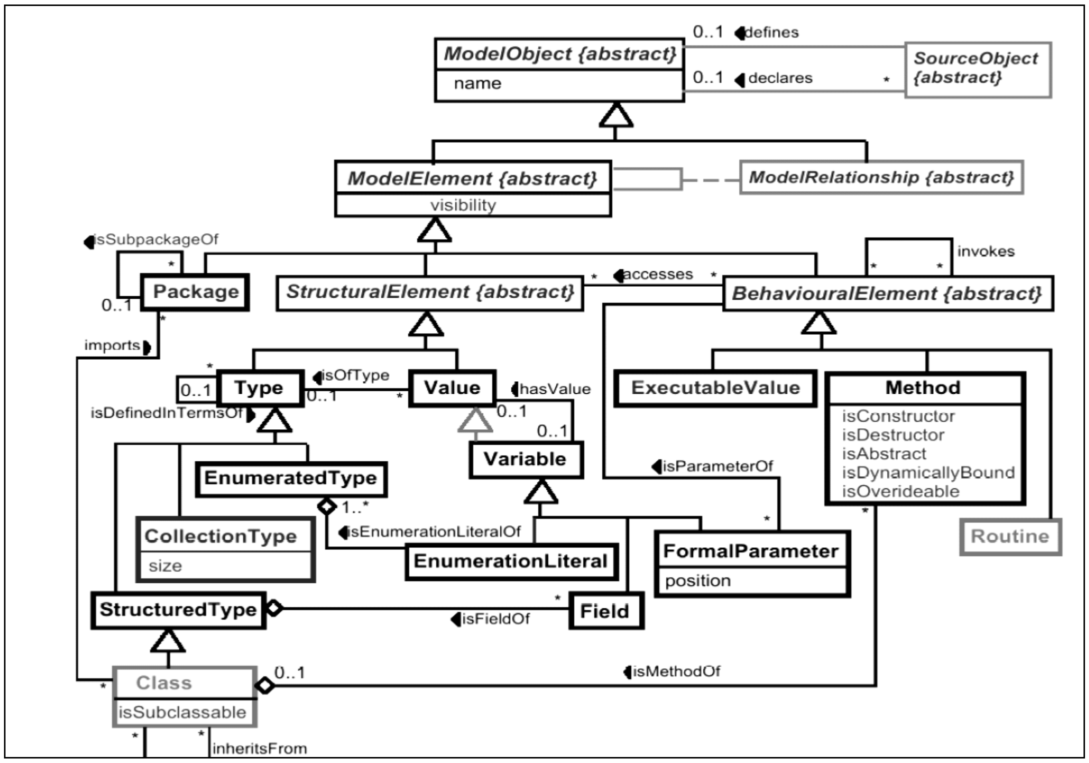


Figure 3 - The Dagstuhl Middle Meta-model Entities Hierarchy

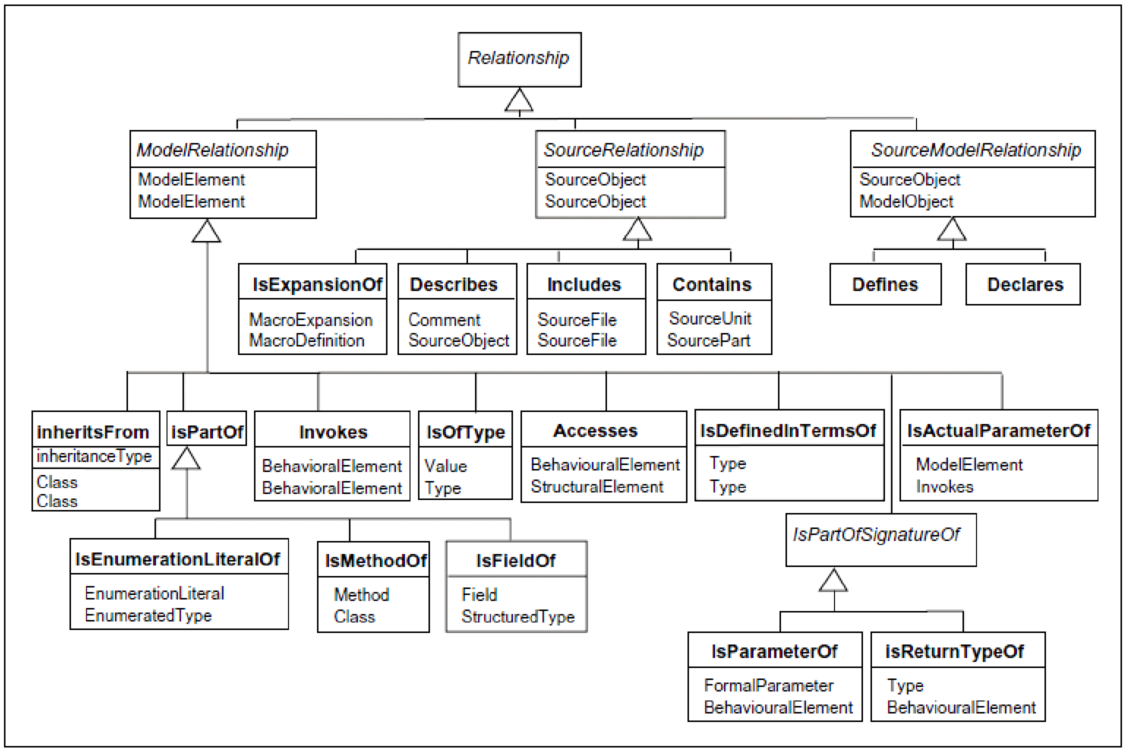


Figure 4 - Relationships of the DMM Meta-model

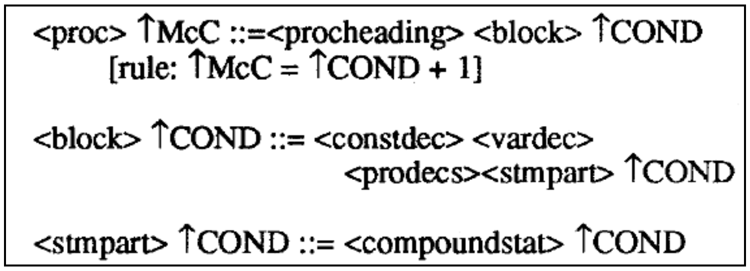
3.2. METRICS DEFINITIONS FORMALIZATION

This section reviews several attempts that have been made toward formalizing software metrics definitions.

3.2.1. Early Attempts

Several attempts to formalize software metrics definitions can be traced to as early as 1991 [30], in which formal definitions for few metrics were suggested such as SIZE OF SOFTWARE. The model was based on software refinement tree model.

Cogan and Hunter proposed introduced an attribute grammar based approach [31]. The main idea was to attach measurement attributes to language definitions in the same way semantic properties of programming languages are defined. This formal approach is language bound, making it very precise, and enables reusability of metrics variable through inheritance of attributes. Figure 5 shows an example of this approach attempting to compute the McCabe's complexity measure.



```
<proc> ↑McC ::=<procheading> <block> ↑COND
      [rule: ↑McC = ↑COND + 1]

<block> ↑COND ::= <constdec> <vardec>
                        <prodecs><stmpart> ↑COND

<stmpart> ↑COND ::= <compoundstat> ↑COND
```

Figure 5 - Formal Definition of the McCabe Metric using Attribute Grammar

3.2.2. <u>Metrics Meta-models</u>

El-Wakil et al. used the ODEM meta-model [21] and XQuery to represent metrics evaluation formulas [22]. Their approach loads the meta-model into and XML DOM tree then processes the tree using XQuery to come up with metrics values. However, their approach was limited to a few metrics because the inherent limitations in the used model, ODEM, did not capture all relationships. In fact it only focuses on design relationships.

Baroni and Abreu [10, 32] suggested an approach for formalizing software metrics definitions based on the Object Constraint Language (OCL), the GOODLY meta-model and UML. GOODLY originally appeared in [23]. As an example, the authors applied the for formalizing CORBA components metrics in [33]. Debnath et al. [34] has done an independent work, yet similar the work of Baroni and Abreu's [32], that uses OCL and UML can be found in [34]. However both attempts suffer from model limitation since their meta-models were only intended for capturing design relationships.

3.2.3. <u>Using the Dagstuhl Middle Meta-model</u>

McQuillan and Power attempted to implement and extend the earlier work of Baroni et al. [32] based on the DMM meta-model. They provided a full definition of C&K metrics suite as an example in [25]. As a prototype, they implemented a tool for executing OCL queries on Java code, that uses the DMM to calculate defined metrics [35]. A discussion of the advantages and limitation of the OCL approach can be found in [36] and [37]. But in general, OCL was meant to describe language constraints and therefore is not efficient in performing simple tasks. Also they did not address reusability of metrics definitions, although their model can be extended to support it.

Lincke and Löwe [38] presented a framework for software metrics definitions based on an abstract meta-model. The model is further expanded and generalized in [16]. Lincke and Löwe idea is to abstract the grammar of different languages (the front end) into a single syntactic and semantic model, hence eliminating the need to refer to the original languages. All grammar and semantics attributes are therefore stored and represented in a common form referred to as the Common Model, which can refer to one of the widely used meta-models such as DMM. This approach is essential to situations where mixed language usage is necessary (e.g. Web applications). This approach addresses the translation between programming language syntax and the common model by defining the grammar needed for the transformation per each language.

The approach is language bound and elements are defined in a grammar language form. To dissolve ambiguity, the author defines complete mapping tied to the language (Java) implementation to the meta-model (DMM). Then it precisely defines the metrics as a BNF grammar with additional special attributes (e.g. for handling loops). While his approach eliminates ambiguity, it is tied to a specific language binding. This method can be helpful for generating metrics parsers, however, it requires specific language binding for each programming languages, and generating a parser. The latter makes it difficult to address reusability and dynamic change of pre-defined metrics, but it would be rather efficient.

3.2.4. Using XML in Metrics Definitions

The Extensible Markup Language (XML) is a widely-used standard language aimed at facilitating data exchange across different systems and platforms [39]. Custom XML rules can be specified with XML Schema documents, another standard format that is also based on XML. Using XML Schema, the rules that XML documents have to follow can be defined. Therefore, XML can be considered a generic standard for language specification. XML power comes from its flexible hierarchal structure and its ability reuse existing elements with virtually unlimited possible sets of associative relationships.

XML usage in software metrics research was proposed in a number of ways. The application of XML to represent metrics data was suggested in [40]. The authors wrote a protocol definition for metrics data exchange that is based on Web Services and XML and called it the Simple Metric Data Exchange Format (SIMDEF). Their goal was to integrate various sources of metrics results into a universal, single repository. The proposal was focused on the exchange protocol and not the representation of metrics definitions. Harrison [41] reported another approach along the same lines.

Margerison researched the use of XML to describe software metrics data in [42]. The author outlined several benefits to using XML that include its flexibility and extensibility. However, there has been no evidence of any progress besides the inception, at the time of this research.

Metrics definitions were also written using XML. In the commercial metrics tool SDMetrics [28] users can apply an XML-based language to define custom design metrics that are based on predetermined relationships of the input design document. This proprietary approach, however, is limited to XMI relationships (based on the UML meta-model) and hence only covers design metrics.

3.3. COMPARISON BETWEEN METRICS DEFINITION APPROACHES

The following table summarizes benefits and shortcomings of the approaches surveyed to formalizing software metrics.

Table 3 – Comparison of Modern Approahces to Formalization of Software Metrics

OCL Queries (Baroni and Abreu 2002) [10, 32]	
Query mechanism: OCL. **Meta-model**: GOODLY.	**Pros**: 1. Application of OCL as a query language. 2. Object-oriented meta-model abstraction. **Cons**: 1. Inadequate meta-model to cover most metrics. 2. Limited to be used with design models. 3. Does not enable evaluating some key metrics.
XQueries on XMI models (El-Wakil et al. 2005) [22]	
Query mechanism: XQuery. **Meta-model**: ODEM.	**Pros**: 1. Use of the XQuery and XML to represent intermediate data. 2. Flexible design meta- model based on ODEM. **Cons**:

	1. Input is limited to design models. 2. Difficult to write queries that manipulate the DOM tree of the meta- model. 3. Anticipated performance overhead due to requirement to create the metrics intermediate data in XML. 4. Does not address re-usability and performance.
OCL with DMM (McQuillan and Power 2006) [24, 25, 35-37]	
Query mechanism: OCL. **Meta-model**: DMM.	**Pros**: 1. Use of the OCL standard language. 2. Use of an open meta-model, DMM. **Cons**: 1. Potential performance issues due to OCL expression evaluation. 2. Does not have address extensibility, reusability of definitions and performance.
DMM based Language Approach (Lincke and Löwe 2006) [16, 38]	
Query mechanism: generates special parsers. **Meta-model**: DMM and UML	**Pros**: 1. Addresses ambiguity between languages and the meta-model. 2. Easily incorporate multiple languages in the same meta-model. **Cons**: 1. More suitable for meta-model description rather than actual metrics definitions. 2. Requires formal derivation of a language-specific parsers to server model translation.

Software Metrics Definition Language (the proposed alternative)	
Query mechanism: Utilizes the concepts of standardized queries and variables. **Meta-model**: multiple meta-model support, such as DMM, OCL and others.	**Pros**: 1. Support a variety of meta-models and different formulation and computation approaches. 2. Addresses problems of performance, ambiguity and reusability. **Cons**: 1. Ambiguity problems of definitions and computations cannot be fully eliminated, e.g. due to vagueness in definitions. 2. XML definitions of metrics could be verbose and harder to read. Human readability could be improved with alternative rendition of the language that uses agile data exchange languages such as JSON (JavaScript Object Notation).

CHAPTER 4

FRAMEWORK FOR REPRESENTATION OF SOFTWARE METRICS DEFINITIONS

This chapter takes a closer look into the process of software metrics measurement to help illustrate the goals, roles and contexts of each component of the proposed measurement solution With this high-level take, we gain better understanding of the problem's requirements and leverage this knowledge to propose a framework for the general case solution of software metrics definitions. We refer to this solution as the Framework for Representation of Software Metrics Definitions.

4.1. METRICS MEASUREMENT PROCESS

The main objective of the measurement process is to come up with quantitative values that represent intrinsic or derived attributes of the measured artifacts. These attributes can then be used to define and assess quality attributes of the artifacts. Repeating this process over the course of project development and to accommodate scalable with variation in platforms and specifications, the measurement process needs to fulfill additional requirements. Examples of these requirements include:

- Provide the ability to formally define metrics computation steps and input requirements.
- Enable flexible metrics definitions, such as defining one metric in terms of other related metrics.
- Process raw input and execute the computation steps.
- Store computed results into a metrics repository and enable exporting the results in the appropriate formats.
- Produce consistent and deterministic results over multiple iterations.
- Satisfy performance constraints and optimize overhead with incremental processing.

4.2. OBJECTIVES OF THE METRICS MEASUREMENT FRAMEWORK

Taking the general requirements of the measurement framework, we can define the following objectives for the Metrics Measurement Framework:

- **Minimizing Computation Ambiguity**: by specifying the computation details based on a common meta-model. This fulfills the requirement of formalizing computation and input requirements.

- **Abstracting Metrics Definitions**: metrics definition should be represented in terms of a meta-model that abstracts design/source code into a general model. This enables producing platform and language agnostic definitions that can be applied to different artifact types. There would still be an ambiguity source during the transformation from the measured artifact to the meta-model which can only be resolved with formal conversion rules.

- **Enabling Metrics Reuse**: by allowing defining metrics in a recursive hierarchy of definitions. This satisfies the flexibility and classification requirements.
- **Extensibility of Metrics Definitions**: additional metrics can be easily added based on built-in meta-model variable and queries or user defined ones.
- **Computation Optimization**: using the concepts of intermediate repository of values and metrics database (e.g. meta-model database) and support for progressive and incremental evaluations.

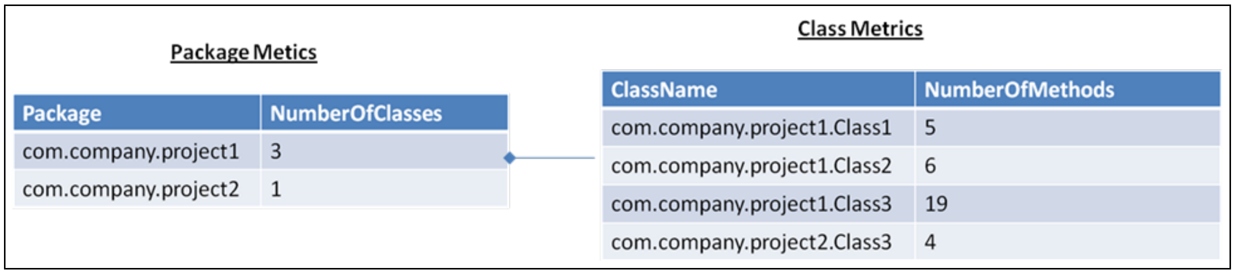
4.3. ELEMENTS OF THE METRICS MEASUREMENT FRAMEWORK

The Metrics Measurement Framework consists of the following components:

- **Parser:** the parser reads the input artifacts and feeds the meta-model database with abstractions sufficient to perform metrics computations. Different types of parsers could be used to accommodate input classes, targeted meta-models, and process inputs at different degrees of scalability. For example, different parsers would be needed for different programming languages and different development architectures. Therefore parsers could be classified based on:

- Input type – e.g. a Java specific parser.
- Processing type – e.g. progressive processing, distributed processing, or all-at-once processing.
- Output type – the parser needs to be designed with a certain meta-model in mind or at least be able to answer specific queries on the input, e.g. calculate the number of classes for the given Java package.

- **Meta-Model Database**: a relational database that stores meta-model representations in a consistent and accessible way. To speed up metrics computations, it could also be used to cache intermediate computations and partial metric results. Taking the number of classes per implementation package and the number of methods per class as an example, this can be represented with the following Entity-Relationship diagram:



Package Metics

Package	NumberOfClasses
com.company.project1	3
com.company.project2	1

Class Metrics

ClassName	NumberOfMethods
com.company.project1.Class1	5
com.company.project1.Class2	6
com.company.project1.Class3	19
com.company.project2.Class3	4

Figure 6 – Meta-model database example

In this example, the table PackageMetrics contains names of parsed packages and the computed number of classes – the information fed from the source parser. The second table contains another set of metrics at the class level. In this case, the number of methods per class is stored. The relational link between the two tables allows for slightly more complex computations that take advantage of this association. For this example, to compute the Number of Methods per Package, a simple relational query could be devised. Depending on the meta-model requirements, this result could be cached for use in more complex calculations, e.g. it could be added to the PackageMetrics table as an extra column.

- **Metrics Definition:** a document in a specific format that contains formal metrics definitions and computation details. In the case of SMDL, which is XML based, this represents definitions of the software metrics based on a certain meta-model and the algorithm needed to perform the computation. An implementation specific design could be made to either centralize or distribute metrics definitions across several documents. For example, SMDL files are designed to be implementation independent and could be used across different tools when the following is supported;
    - The project input type, e.g. the specific programming language
    - The meta-model, e.g. the DMM model or OCL meta-models and their level of coverage.
    - SMDL queries, e.g. support for OCL based queries.

- **Metrics Definition API:** the programming interface to parse and process the definition documents. For SMDL, this is the interface to read metrics definitions, query about the meta-model requirements and access related metrics.
- **Metrics Data Representation:** once metrics values are computed, the system presents the results in a suitable format. The final representation could be tabular or visual, e.g. pie-charts and histograms. A typical capability would be the support of exporting the results in formats that could be used in external analysis and data mining applications, especially when armed with versioning support. For example, when feeding the results to configuration management system, metrics changes could be tracked over the course of a project and help identifying trends and patterns and sources of change could be traced back to their origins.
- **Metrics Tool:** the application that drives the entire measurement process and coordinates operation and access to the system components. Typically this is comprised of the user interface, database access layers, and the application logic associated with programming interfaces of the other components. Examples of user functions are:
    - Load and select metrics definition.
    - Define and parse a source project.
    - Compute metrics for the selected project.
    - Setup of the metrics database.
    - Metrics viewer and export capabilities.

4.4. THE METRICS MEASUREMENT FRAMEWORK

The following diagram sums up the main components of the Metrics Measurement Framework, the relationships, and interfaces between the subcomponents. In the next chapter we introduced the Software Metrics Definition Language which is based on ideas presented in this framework.

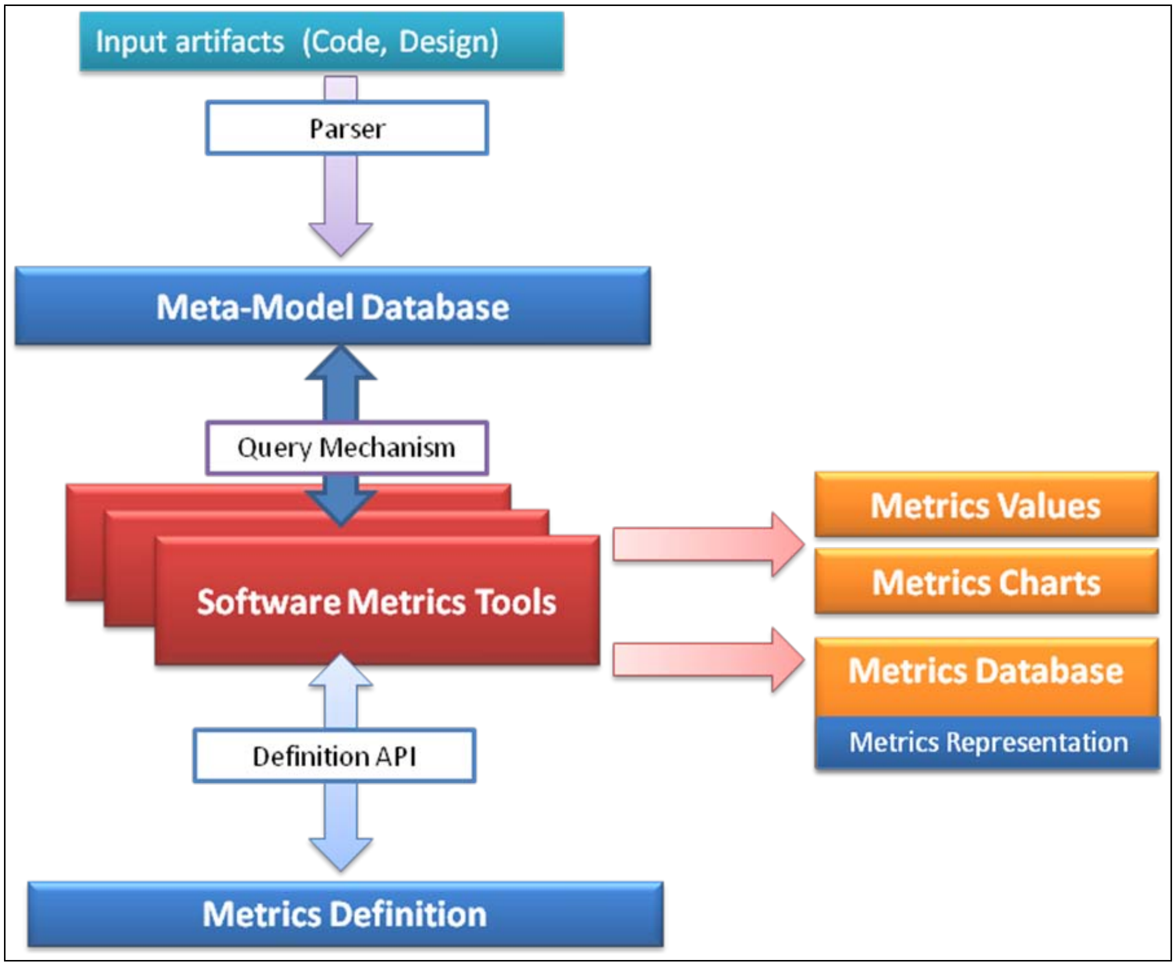


Figure 7 - Metrics Measurement Framework Architecture

CHAPTER 5

THE SOFTWARE METRICS DEFINITION LANGUAGE

In the proposed software metrics measurement framework, models of metric definitions need to fulfill the following requirements: (a) Allow formal expression of metrics computation. (b) Define a meta-model that is derived extracted from source code, design artifacts or pre-computed valued. (c) Enable customization of metrics definitions by either reuse of existing metrics or the intermediate values. (d) Be extensible enough to accommodate alternative meta-models and methods of computation. Optional features include support for visualization expressions and data output representation. The conceptual model of these requirements is illustrated in Figure 8.

In this chapter, we introduce an XML based markup language for representing software metrics definitions designed to meet all requirements of the proposed framework. We will refer to this language as the Software Metrics Definition Language.

The selection of XML as basis for this language is due to its power of expression, flexibility, accessibility and universal support. In particular, hierarchal and relational associations can both be expressed with standard XML notations, which is a key to mimicking complex metrics relationships.

Multiple aspects of metrics data are captured in different sections. The language is divided into four sections that capture different sets of information (details are shown in the schema definition in Figure 2.

5.1. SMDL CONCEPTS

This section describes the main concepts used by the SMDL and how they fulfill their design requirements.

5.1.1. Meta-Model Base

Metric definitions in SMDL are given in terms of expressions that are evaluated based on a pre-defined meta-model which abstracts the artifact to be measured. Examples of meta-models include the Dagstuhl Middle Meta-model and the UML meta-model. Expressions differ in their representation according to the selected meta-model. For example, the UML meta-model based metrics can be expressed in the form of OCL, the standard Object Constraint Language. In SMDL, DMM based definition utilize mathematical expressions and algorithms in the form of MathXML expressions. However, both approaches use the same concepts for the evaluation process: intermediate variables and built-in queries.

5.1.2. Variables and Queries

Variables store values that are potential candidate for use in the evaluation of software metric in SMDL. Variables are defined to be attached or scoped to a given meta-model element, referred to as the scope of the variable. Queries in SMDL represent the approach followed to retrieve data stored in the intermediate mete-model store or to verify the correctness of a given hypotheses. The latter can also be referred to as *Boolean* queries.

5.1.3. Intermediate Storage / Meta-Model Database

Computed variables and results of metric queries are usually stored temporarily in special database of intermediate values. This database aids in proving incremental evaluation of metrics and preventing redundant computations. However such efficiency is not achievable if the model is not aware of *invalidation* rules. The ideas behind intermediate storage, incremental evaluation and invalidation rules are detailed in the following sections.

5.1.4. Deep vs. Progressive Evaluation

There are three basic approaches to compute a particular software metric: complete or deep evaluation, progressive evaluation, and re-computation. Re-computation and invalidation rules are discussed in the next section.

Complete or deep evaluation refers to computing the value of the metrics through complete evaluation of each metric dependent data then applying the metric formula. For example, in order to evaluate the Depth of Inheritance (DIT) for a given class, the parser, in a deep evaluation cycle, needs to look up all parents of the given class return their summation. In this scenario the parser only focuses on the returned the final value of the metric not considering the useful intermediate values that can speed up computing the values for the rest of the classes. Hence, this approach can be very slow as the same procedure would have to be followed for all other classes at hand. However, this approach is evidently useful when dealing with a small subset of a large group of classes.

The second approach, the progressive evaluation, handles the matter incrementally. Given the artifacts under evaluation in arbitrary order, this method would be able to compute the final value of the metrics by passing by the artifacts only once. The computation is organized in a form of pipeline of calculations where each the metric value can be computed only partially. As a side-effect, the computation can result in queuing more artifacts into the pipeline. The process continues until the metric value is fully computed or the queue becomes empty.

The following highlights the algorithm followed in this approach:

```
Var queue = [];
Var metricValues = [];
Var queue = /* queue of artifacts at hand */
Foreach (artifact a in queue)
    If (metricValue[a] is marked “complete”)
            Continue;

    Var partialValue = …
    metricValue[a] += partialValue
    Foreach (metricValue in metricValues)
            If (metricValue is affected by a)
                   Update metricValue;
                   If (metricValue is complete)
                           Mark metricValue “complete”
            If (new artifacts are needed to compute the metric)
```

In our example, we are interested in computing the value of DIT incrementally. A progressive algorithm would pick the given class, look up and queue its direct parent, then continue visiting each node, adding “one” each time another parent is found until the pipeline becomes empty. This approach resembles the procedure followed in the famous Depth First graph traversal algorithm and can be implemented using the Visitor design pattern. A more elaborated version of this algorithm is highlighted in the following example:

```
Var DIT = [];

Foreach (Class c in classesQueue)
    If (DIT[c].status = complete)
            Continue;
    if (c.hasParent)
            /* case 1: parent is complete */
            if (DIT[c.parent] != [] and
                DIT[c.parent].status = complete)
                   DIT[c].val = DIT[c.parent] + 1
                   DIT[c].status = complete
                   Foreach (var value in DIT)
                           If (value.status != complete and
                               Value.parent = c)
                                  Value.val = DIT[c].val + 1

            /* case 2: parent is incomplete */
            Else if (DIT[c.parent] != [])
                   DIT[c].val = DIT[c.parent] + 1
                   Foreach (var value in DIT)
                           If (value.status != complete and
                               Value.parent = c)
                                  Value.val++
            /* case 3: parent is incomplete */
            Else if (c.parent != [])
                   DIT[c] = 1
                   classesQueue.enqueue(c.parent)
```

Progressive evaluation becomes particularly useful for evaluating metrics of a large group of classes. The reason is that each time an artifact is “visited”, the parser can partially compute the value for the current artifact as well as directly related ones. Therefore, for each visit, metrics of several other artifacts get computed at the same time without having to revisit the past artifacts. This results in a computational pipeline which can greatly accelerate the overall evaluation process.

However, like any recursive algorithm, the stopping criteria should be clearly determined in order to avoid infinite or unnecessary calculations. This would be usually determined according to the metric and artifacts under evaluation. For example, the DIT metric can add new classes which are outside the scope of the requested classes, e.g. library classes. The added classes should be excluded from other calculations that do not add up to the value of the metric. That is, while these additional classes are needed for computing values of the rest of the inheritance tree, they should not interfere with the other metrics and should be treated as extra classes.

One drawback to the progressive evaluation is that it requires extra storage for storing intermediate values and the status of the evaluation. The algorithmic complexity is also affected by the “look-ups” needed to ensure that all related metrics are being updated accordingly. On the other hand the pipeline architecture followed is very useful when operating in a parallel or distributed computing environment, with the exception of synchronization overhead. With current rise of multi-core processors and distributed computing, this approach appears more favorable.

5.1.5. Invalidation Rules

Invalidation rules represent actions that need to be undertaken in order to maintain consistency of the computed results. In particular, they determine the values that should be recomputed across the existing set of results in response to a change in a particular element. For example, in the case of DIT, changing the parent class of a certain class would imply invalidation of the values computed for: a) the class itself, and b) all direct and indirect descendants. Metric values, when invalidated, are therefore required to be re-evaluated. While for some cases small changes result in a minor re-calculations, it could result of invalidating the whole set of metrics. Consider for instance the case of class A with multiple children B_1, B_2, …. B_n. Changing the parent of A to C for example would result in invalidation of all computed values of DIT resulting in a negative performance in reaction to a small change. However, such scenarios deemed to have low probability (as they require special organization) in practice and therefore would not overcome the performance gain achieved through progressive computations.

5.1.6. <u>Exceptions and Constraints</u>

Alternative flows of the computation process can be expressed in the form of Exceptions and Constrains. Exceptions refer to special computational cases. For example, the default value for a metric when a certain input is not available. Constraints, on the other hand, refer to pre-conditions that need to be met before in order to evaluate or to continue the evaluation of a given metric. An example of exceptions would be the coupling value of a given class when it references itself. In this case the evaluator should return a value that does not affect the overall result (zero in this case). For certain metrics, the computation shouldn’t proceed before satisfying a given expression, often the pre-condition. For example, the Number of Public Methods, by definition, should skip methods not declared with public visibility and increment the counter otherwise. Expressed in SMDL, this could be written as:

```
…
 <exception condition="c = currentClass" value="0" />

 <condition condition="isPublic = false" action="continue" />
…
```

In this example, an Exception and a Condition are defined. Notice the OCL-like style of expression. This tells the processing engine to evaluate both expressions whenever a new input is encountered. In the case of the Exception, the metric value is assigned a special value of 0 when the expression “c = currentClass” evaluates to True, thus setting the metric value to zero for the class associated with the metric. Notice use of the following attributes:

3. Condition: used to hold the OCL like expression to be evaluated.

4. Value: Exception specific attribute to define the return value.

5. Action: Condition specific attribute that contains the statement to be executed when the condition is met.

5.2. SMDL ATTIBUTES

Attributes in SMDL are used to capture the following aspects of a given metric:

1. Metrics properties: the general attributes of the metric such as the name, the author and version identifier.

2. Visualization attributes: specify how the metrics values can be visualized in the most suitable formats, e.g. size metrics are often used with Treemap representations.

3. Re-use attributes: determine dependencies on the other metrics and their relationship with the computation. For example, metrics that depend on other metrics or can make use of other metrics to speed up the computation can specify their dependencies.

4. Grouping attributes: used to denote the classification groups the metric belongs to. For instance, NPM (Number of Public Methods) can be associated with SIZE and STATIC classes.

5. Conversion rules: results of metrics often need to be compared or aggregated with metrics from different environments, e.g. across multiple programming languages, which potentially follow different computational rules. Conversion rules define transformations necessary to aggregate metrics from incompatible platforms. For example, computing the RFC (Response for Class) metric is slightly different for languages with multiple-inheritance support. Another example the way LOC (Lines of Code) could be computed across languages with different white-space and indentation requirements. The transformation rules could come in handy when applied to multi-language projects, an increasingly common case.

6. Computational attributes: this is the heart of the metric definition which states how the metric should be computed starting from a selected meta-model of the input.

Figure 8 shows the location of these attributes in the SMDL schema.

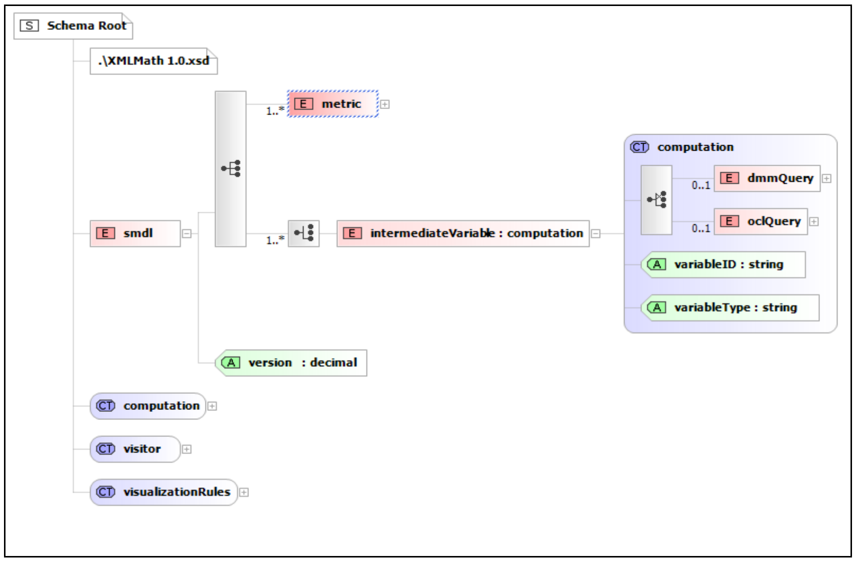


Figure 8 – Overview of the main SMDL elements

Metric Properties and Computation attributes are described in more details in the following segments.

5.2.1. Metrics Properties

This part of the SMDL scheme is designed to capture properties of the software which are not directly related to its evaluation rules yet are essential to applying the metric. Examples include: the metric name, names of the author(s), the metric level, and the evaluation scope (e.g. class, package, method, or application). Extensibility is supported through a customizable key-value-attribute scheme where user-specific and tool-specific attributes can be defined. Metrics are often grouped into smaller collections (according to criteria related to origin and role) that are referred to as *metrics suites* or *classes of metrics*.

5.2.2. <u>Metrics Computation</u>

This is the core of SMDL metrics definitions which expresses the steps necessary to evaluate and arrive at the final values of the metric. Computation expressions make the assumption that a values for the selected meta-model are available and accessible to the computation engine. It defines metrics computations based on queries or variables. It supports both source-level and design level computations using the power of DMM representation.

Variables hold values which can be associated with queries, or they are provided directly (pre-computed). For example, in Quality Assurance (QA) metrics, variables that store the number of defects are of the direct type. Variables can also be grouped and based on other variables in a hierarchy.

Queries are functions associated with the meta-model that return answers needed to arrive at metrics results. The SMDL model supports three different mechanisms to computation specification: OCL queries on the meta-model [36], grammar based [38] and direct invocation of built-in queries. Built in queries are extensions of the DMM model relation that normally return a set of values per query on a given element. Table 1 shows some of the supported queries. An example is using Get_MethodsOf(Class c) to get all methods in a class.

Variables and queries can be associated with conditions and exceptions that express flow of control and special cases. Conditions are expressed in the form of "**A rel B**", where rel is any logical relation. It also supports nesting of conditions. Exceptions define what happens to particular values, .e.g. “if x < 0, x = 0”. This can be useful at different levels for many metrics. Grouping operations such as count, average, sum are also provided.

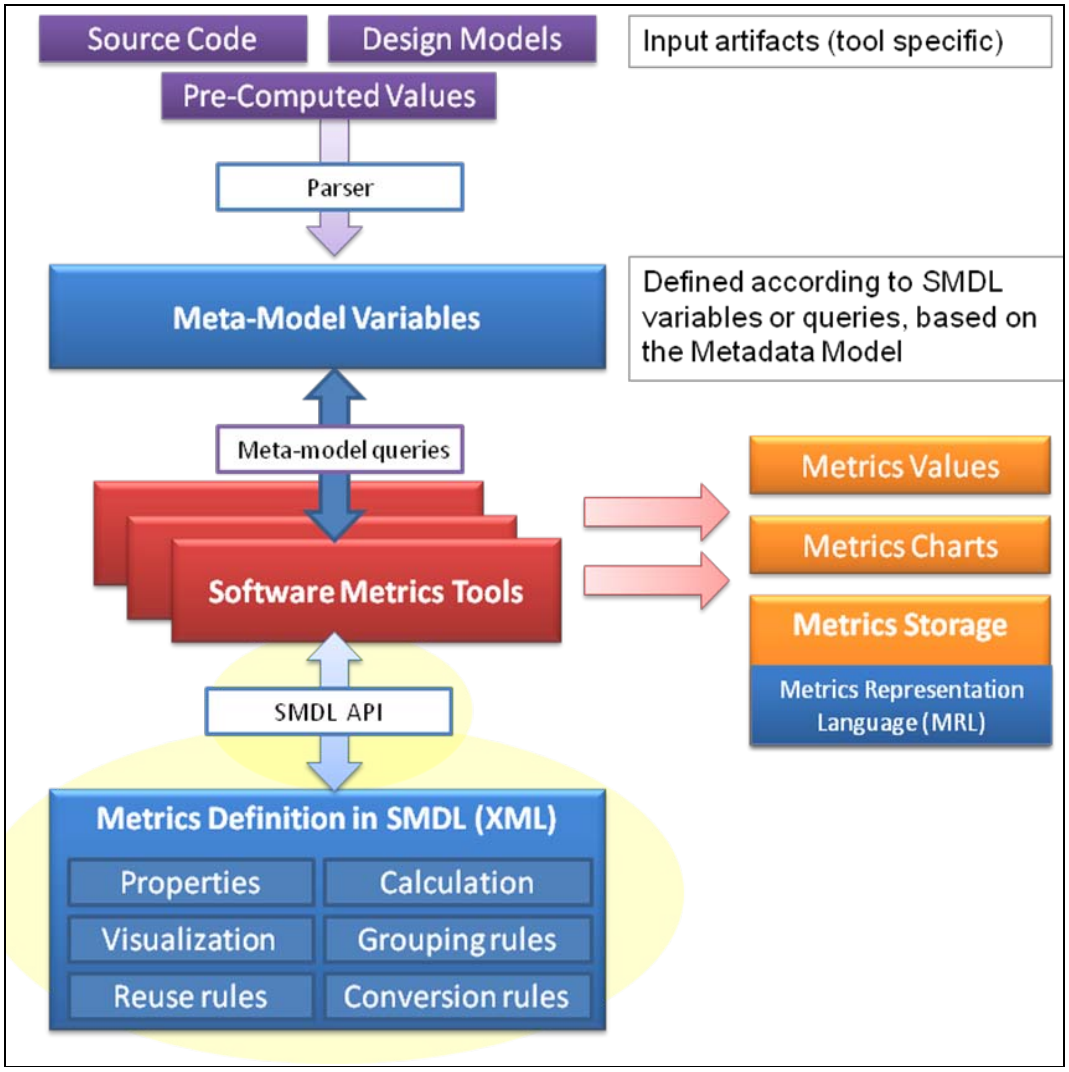


Figure 9 - SMDL Role in the Software Metrics Measurement Framework

5.3. SMDL APPLICATION PROGAMMING INTERFACE (API)

This section describes the required functionality to be implemented in the Application Programming Interface (API) of SMDL.

5.3.1. API Model Classes

The selected meta-model should be mapped to the parser's language and include classes that resemble the hierarchy structure given in the original meta-model in addition to the domain classes of the parser. This hierarchy forms what can be referred to as the meta-model space. For example, when implementing the SMDL API for dealing with the DMM meta-model, the class hierarchy presented in Figure 3 has to be implemented in a manner that preserves relationships and attributes of the model.

5.3.2. API Built-In Queries

In order to evaluate the metrics, the tool needs to be able to perform queries on the meta-model. Table 4 highlights the basic built-in queries in SMDL required for retrieval of the intermediate DMM values.

5.3.3. XMLMath

XML-Math is a flexible XML based language for representing and computing mathematical expressions represented in XML format. It was developed in 2006 by Erik van Zijst and represents a clear way of defining mathematical expressions in XML. The language can represent most of mathematical operations, loops and conditions. The following is an example of an expression written in XMLMath which returns the values from 0 to 9 using the operator *toString*(notice the namespace attribute) [43]:

```
<expression xmlns="http://xmlmath.org/1.0">
  <toString>
    <for iterator="i">
      <start>
        <long value="0"/>
      </start>
      <end>
        <long value="10"/>
      </end>
      <do>
        <linkLong name="i"/>
      </do>
    </for>
  </toString>
</expression>
```

Expressions in SMDL are defined using XMLMath with a slight modification: variable values (if not defined) are assumed to be evaluated from the intermediate meta-model database.

The XMLMath defines the following data types which are also used by SMDL variables:

1. *boolean*
2. *number, which includes: long and double*
3. *string*
4. *list*

The *list* data type is of special importance since it can represent a list of elements in their corresponding data types (e.g. set of integers or strings). In our implementation of SMDL, we introduce another sub-type of the list of elements that defines mathematical sets, referred to as set. This data type differs from an ordinary list in that it does not allow repeated elements.

5.3.4. Meta-Model Evaluation and Initialization

A software metrics tool that follows the framework of SMDL would need to compute or define ways to compute the corresponding meta-model elements. For DMM, the meta-model evaluator comes in the following format, implementing several “built-in” meta-model queries:

```
class DMMMetaModelEvaluater : MetaModelEvaluater
{
    /* retrieve that implement relations of the DMM model */

    StructuralElement[]   Get_Accesses (BehaviouralElement b) { … }
    SourcePart[]  Get_Contains (SourceObject so) { … }
    ModelObject[] Get_Declares (SourceObject so) { … }
    ModelObject[] Get_Defines  (SourceObject so) { … }
    Comment[]     Get_Describes (SourceObject so) { … }
    Value[] Get_HasValue (Variable v) { … }
    Package[]     Get_Imports (Class c) { … }
    SourceFile[]  Get_Includes (SourceFile sf { … })
    Class[] Get_InheritsFrom (Class c) { … }
    BehaviouralElement[]  Get_Invokes (BehaviouralElement be) { … }
    …

}
```

The following table lists all required meta-model queries for the DMM model:

Table 4 – Built-in SMDL Queries based in the DMM Model

Return Type	Query	Desfription
StructuralElement[]	Get_Accesses (BehaviouralElement b)	Returns structural elements that the given behavioral element accesses.
SourcePart[]	Get_Contains (SourceObject so)	Returns SourcePart elements

Return Type	Query	Desfription
		“Contained” in the given SourceObject.
`ModelObject[]`	`Get_Declares (SourceObject so)`	Returns ModelObject’s that are declared in the given SourceObject.
`ModelObject[]`	`Get_Defines  (SourceObject so)`	Returns ModelObject’s that are defined in the given SourceObject.
`Comment[]`	`Get_Describes (SourceObject so)`	Returns comments associated with the given SourceObject.
`Value[]`	`Get_HasValue (Variable v)`	Returns a list of values of the given Variable.
`Package[]`	`Get_Imports (Class c)`	Returns Package’s imported by a given class.
`SourceFile[]`	`Get_Includes (SourceFile sf)`	Returns file’s included by a given source file.
`Class[]`	`Get_InheritsFrom (Class c)`	Returns super-classes of a given class.
`BehaviouralElement[]`	`Get_Invokes (BehaviouralElement be)`	Returns a list of behavioral elements (e.g. methods) invoked by a given element.
`Invokes[]`	`Get_ActualParameterOf (ModelElement me)`	Returns the actual parameters list of a given element.
`Type[]`	`Get_DefinedlnTermsOf (Type t)`	Returns the type used in the definition of a given type, e.g. coupling through ADT.
`EnumeratedType[]`	`Get_EnumerationLiteralOf (EnumeratedLiteral el)`	Returns literals if a given enumeration literals list.
`Field[]`	`Get_FieldsOf (StructuredType st)`	Returns fields of a given structure.

Return Type	Query	Desfription
`Method[]`	`Get_MethodsOf (Class c)`	Returns the list of methods declared within a class.
`Type`	`Get_TypeOf (Value v)`	Returns the Type of a given Value object.
`FormalParameter[]`	`Get_ParameterOf (BehaviouralElement)`	Returns the list of parameters defined in a given BehavioralElement.
`Type`	`Get_ReturnTypeOf (BehavioralElement be)`	Returns the Type of a given BehavioralElement.
`Package[]`	`Get_SubpackagesOf (Package p)`	Returns sub-packages of a given Package.

In order to satisfy the performance goals, through progressive computation, the Visitor design pattern has been used. Metric definitions represented in SMDL correspond to objects that perform the actual evaluation for the metrics. In order to provide a full incremental implementation, the software parser takes each artifact and passes its information to metrics evaluators where they get called every time a software element is ready for evaluation.

The following class, the AbstractVisitor, is a base-class for all the metrics evaluators. It consists of metrics visiting methods that are called when the corresponding program element is parsed. The class also contains helper methods for declaring and updating values in the meta-model intermediate database. Metric evaluators implement the portions necessary to compute values of the metrics following the visitor's pattern event model.

```
class AbstractVisitor
{
    public abstract void visitPackage(string packageName);
    public abstract void visitClass      (string className);
    public abstract void visitMethod (string methodName);
    public abstract void visitField      (string fieldName);
    public abstract void visitValue      (string valueName);
    public abstract void visitVariable   (string variableName);

    public abstract void visitType       (string TypeName);
    public abstract void visitEnumerationType    (string
EnumerationTypeName);
    public abstract void visitStructuredType     (string
StructuredTypeName);
    public abstract void visitFormalParameter    (string
FormalParameterName);
    public abstract void visitRoutine    (string RoutineName);
    public abstract void visitExecutableValue    (string
ExecutableValueName);
    public abstract void visitCollectionType     (string
CollectionTypeName);


    // fullElementName refers the the full qualified name of the object
    // (e.g. package.class.method.variable)
    public void declareVariable(string varType, string varScope, string
fullElementName)
    {
            // register the variable in the temp store and
            // associate it with the given scope
            Store.createVarvarType, varScope, fullElementName);
    }

    public string retrieveVariable(string varScope, string elementName)
    {
            return Store.getVarValue(varScope, fullElementName);
    }


    public string updateVariable(string varScope, string elementName,
string newValue)
    {
            return Store.setVarValue(varScope, elementName, newValue);
    }

    public boolean evaluateCondition(string condition, string operator,
string expectedValue)
    {
            return Store.evaluate(condition, operator, expectedValue);
    }
}
```

All element names represent full qualified names of the static elements. For example the method C in class B of package A should be referred to as "A.B.C".

The parser algorithm therefore is as follows:

```
class Parser
{
    private AbstactVisitor visitor;

    void parse(Class c)   {
            visitor.visitClass(c.name);

            /* visit methods of the given class */
            for (Method m : c.methods)
            {
                  /* visit method parameters */
                  for (FormalParameter p : m.parameters)
                          visitor.visitFormalParameter(p.name);

                  /* visit the actual method */
                  visitor.visitMethod(m.name);

                  /* visit variables used in the method */
                  for (Variable v : m.variables)
                          visitor.visitVariable(v.name);

                  /* visit other classes accesed in this method */
                  for (Type t : m.accesses)
                          visitor.visitType(t.name);
            }

            /* visit fields of the given class */
            for (Field f : c.fields)
            {
                  visitor.visitField(f.name);
                  visitor.visitValue(f.value.name);
            }
            for (Method m : c.methods)
                  visitor.vistiMethod(m.name);
    }
}
```

Therefore, the major role of SMDL under this incremental evaluation is to describe the implementation algorithm of each visiting method in order to come up with the final metric value.

As an example, consider the following expression which can be used for computing the number of public methods in a given class. The metric query can be written in SMDL as (dmmQuery refers to a meta-model query that implements the DMM model):

```
<dmmQuery>
    <description>Compute the number of methods in a given class
    </description>
    <visitor scope="class">
            <variable name="numMethods" type="long" scope="class" />
    </visitor>
    <visitor scope="method">
            <condition expression="isPublic = false" action="continue" />
            <math:expression>
                   <linkLong name="numMethods"/>
                           <add datatype="long">
                                  <long value="1"/>
                           </add>
                   </linkLong>
            </math:expression>
    </visitor>
</dmmQuery>
```

This expression would declare a variable called "methodCount" in the scope of the current class. The declared variable is therefore used in an XMLMath expression to update the value of the variable after each visit.

This SMDL representation is essentially equivalent to the following code (which would be generated during the actual parsing of the metric definition). Notice that the Adapter design pattern is applied here through the VisitorAdapter class in order to avoid implementing all methods of the AbstractVisitor:

```
class ConcreteVisitor : VisitorAdapter
{
    public abstract void visitClass      (string className)
    {
            declareVariable("long", "class", className+".numMethods");
    }
    public abstract void visitMethod (string methodName)
    {
            if (evaluateCondition("isPublic", "equals", "true"))
            {
                  long    temp    =    Long.parse(retrieveVariable("class",
methodName+".numMethods"));
                  updateVariable("class",  methodName+".numMethods",  temp
+ 1);

            }
    }
}
```

Invalidation criteria can also be described in SMDL as in the following example:

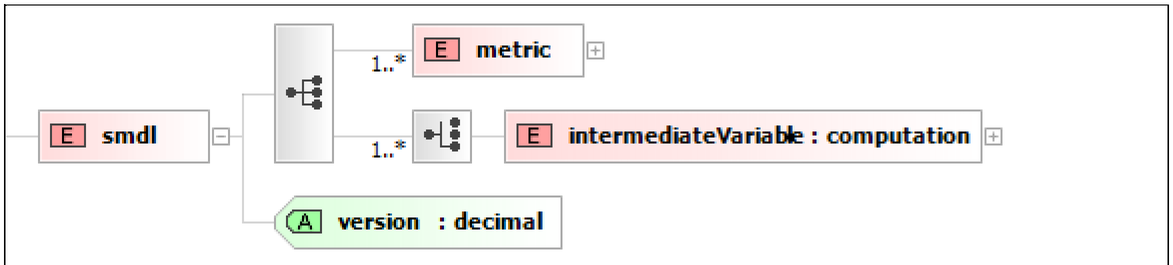
```
<dmmQuery>
    <visitor scope="class">
            <variable name="numMethods" type="long" scope="class" />
    </visitor>
    <visitor scope="method">
            <condition expression="isPublic = false" action="continue" />
            <invalidationCriteria affectedElement="Method"
condition="isPublic = True" scope=”class” />
            <math:expression>
                  <linkLong name="numMethods"/>
                          <add datatype="long">
                                <long value="1"/>
                          </add>
                  </linkLong>
            </math:expression>
    </visitor>
</dmmQuery>
```

The variable "numMethods" is declared under the scope of the current class in the temporary store. When methods of the given class are being evaluated, the current value is retrieved and incremented before writing back to the store. This example invalidates all elements of type “Method” which satisfy the condition “isPublic = true” within the scope of the “class”.

5.4. SMDL DEFINITION SCHEME

The following section describes the contents of the various elements of the SMDL language.

The SMDL Root Element



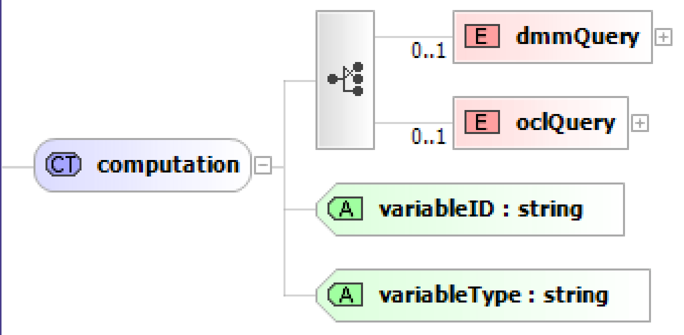
XSD Schema Code

```
<xs:element name="smdl" >
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" name="metric">
        <xs:complexType>
          <xs:all>
            <xs:element name="acronym" type="xs:string"/>
            <xs:element name="title" type="xs:string"/>
            <xs:element minOccurs="0" name="description" type="xs:string"/>
            <xs:element minOccurs="0" maxOccurs="1" name="customProperties">
              <xs:complexType>
                <xs:attribute name="attribute" type="xs:string" use="optional"/>
                <xs:attribute name="value" type="xs:string" use="required"/>
              </xs:complexType>
            </xs:element>
            <xs:element minOccurs="0" name="conversionRules">
              <xs:complexType>
                <xs:attribute name="sourcePlatform" type="xs:string"/>
                <xs:attribute name="targetPlatform" type="xs:string"/>
                <xs:attribute name="forumula"/>
              </xs:complexType>
            </xs:element>
            <xs:element minOccurs="0" name="authority">
              <xs:complexType>
                <xs:sequence>
                  <xs:element maxOccurs="unbounded" name="authors">
                    <xs:complexType>
                      <xs:attribute name="name" type="xs:string"/>
                      <xs:attribute name="date" type="xs:string"/>
                    </xs:complexType>
                  </xs:element>
                  <xs:element name="yearPublished" type="xs:date"/>
                  <xs:element name="sourceName" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element minOccurs="0" name="computation" type="computation"/>
            <xs:element minOccurs="0" maxOccurs="1" name="reusedMetrics">
              <xs:complexType>
                <xs:attribute name="metricName"/>
                <xs:attribute name="variableID"/>
              </xs:complexType>
            </xs:element>
            <xs:element minOccurs="0" name="visualization" type="visualizationRules"/>
          </xs:all>
          <xs:attribute name="metricClass" type="xs:string"/>
          <xs:attribute name="metricSuite" type="xs:string"/>
          <xs:attribute name="scope">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="package"/>
                <xs:enumeration value="class"/>
                <xs:enumeration value="method"/>
                <xs:enumeration value="variable"/>
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="intermediateVariable" type="computation"/>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="version" type="xs:decimal" use="required"/>
  </xs:complexType>
</xs:element>
```

Child Elements

Name	Type	Min Occurs	Max Occurs
metric	metric	(1)	unbounded
intermediateVariable	intermediateVariable	(1)	(1)

Computation DataType



XSD Schema Code

```
  <xs:complexType name="computation" >
```

```
    <xs:all>
      <xs:element minOccurs="0" name="dmmQuery">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="description" type="xs:string"/>
            <xs:element name="unit" type="xs:string"/>
            <xs:element maxOccurs="unbounded" name="visitor" type="visitor"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="oclQuery">
        <xs:complexType>
          <xs:sequence>
            <xs:element minOccurs="0" maxOccurs="unbounded" name="oclQueryVariable">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="query" type="xs:string"/>
                  <xs:element name="variableID" type="xs:string"/>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="mainOclQuery" type="xs:string"/>
            <xs:element name="scope">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="class"/>
                  <xs:enumeration value="package"/>
                  <xs:enumeration value="method"/>
                  <xs:enumeration value="field"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="variableID" type="xs:string"/>
    <xs:attribute name="variableType">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="double"/>
          <xs:enumeration value="long"/>
          <xs:enumeration value="list"/>
          <xs:enumeration value="set"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
```

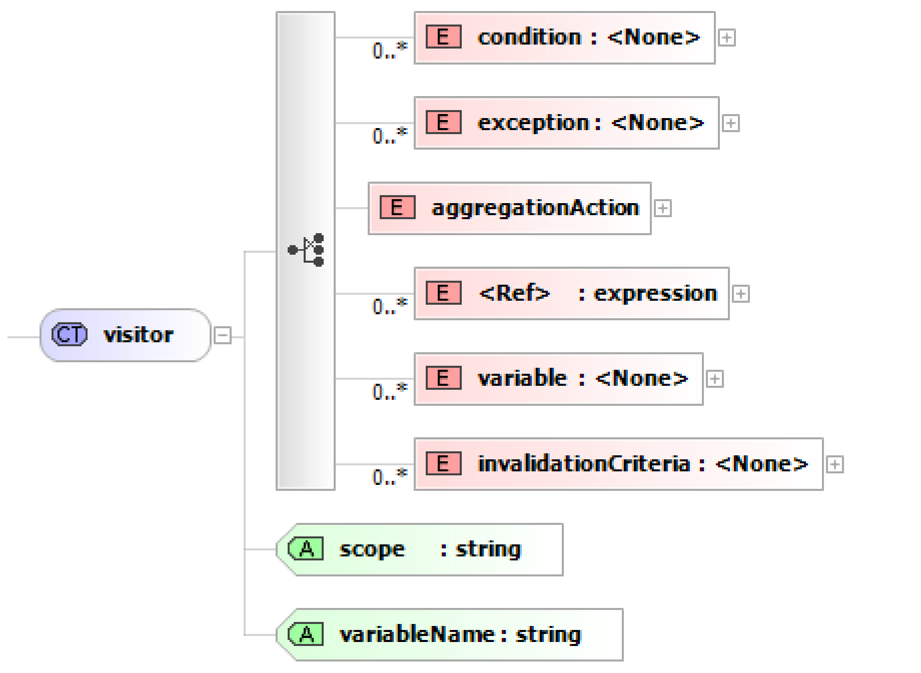
Child Elements

Name	Type	Min Occurs	Max Occurs
dmmQuery	dmmQuery	0	(1)
oclQuery	oclQuery	0	(1)

Child Attributes

Name	Type	Default Value	Use
variableID	variableID		(Optional)
variableType	variableType		(Optional)

Visitor DataType



XSD Schema Code:

```
<xs:complexType name="visitor" >
  <xs:all>
    <xs:element minOccurs="0" maxOccurs="1" name="condition">
      <xs:complexType>
        <xs:attribute name="expression" type="xs:string" use="required"/>
        <xs:attribute name="action" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="skip"/>
              <xs:enumeration value="stop"/>
              <xs:enumeration value="continue"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element minOccurs="0" maxOccurs="1" name="exception">
      <xs:complexType>
        <xs:attribute name="expression" type="xs:string" use="required"/>
        <xs:attribute name="returnValue" type="xs:string" use="required"/>
      </xs:complexType>
    </xs:element>
    <xs:element name="aggregationAction">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="type" type="xs:string"/>
          <xs:element name="operator" type="xs:string"/>
          <xs:sequence/>
        </xs:sequence>
        <xs:attribute name="level">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="package-leve"/>
              <xs:enumeration value="class-level"/>
              <xs:enumeration value="method-level"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="action">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="sum"/>
              <xs:enumeration value="multiply"/>
              <xs:enumeration value="average"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element minOccurs="0" maxOccurs="1" ref="ns0:expression"/>
    <xs:element minOccurs="0" maxOccurs="1" name="variable">
      <xs:complexType>
        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="type">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="long"/>
              <xs:enumeration value="double"/>
              <xs:enumeration value="string"/>
              <xs:enumeration value="list"/>
              <xs:enumeration value="set"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="scope">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="package"/>
              <xs:enumeration value="class"/>
              <xs:enumeration value="method"/>
              <xs:enumeration value="attribute"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
    <xs:element minOccurs="0" maxOccurs="1" name="invalidationCriteria">
      <xs:complexType>
```

```
        <xs:attribute name="affectedElement" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="Method"/>
              <xs:enumeration value="Class"/>
              <xs:enumeration value="Package"/>
              <xs:enumeration value="Field"/>
              <xs:enumeration value="Attribute"/>
              <xs:enumeration value="Parameter"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
        <xs:attribute name="variableName" type="xs:string" use="required"/>
        <xs:attribute name="condition" type="xs:string" use="required"/>
        <xs:attribute name="scope" use="required">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="Method"/>
              <xs:enumeration value="Class"/>
              <xs:enumeration value="Package"/>
              <xs:enumeration value="Field"/>
              <xs:enumeration value="Attribute"/>
              <xs:enumeration value="Parameter"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:complexType>
    </xs:element>
  </xs:all>
  <xs:attribute name="scope">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Package"/>
        <xs:enumeration value="Class"/>
        <xs:enumeration value="Method "/>
        <xs:enumeration value="Field "/>
        <xs:enumeration value="Value "/>
        <xs:enumeration value="Variable"/>
        <xs:enumeration value="Type"/>
        <xs:enumeration value="EnumerationType "/>
        <xs:enumeration value="StructuredType"/>
        <xs:enumeration value="FormalParameter"/>
        <xs:enumeration value="Routine "/>
        <xs:enumeration value="ExecutableValue"/>
        <xs:enumeration value="CollectionType"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="variableName" type="xs:string"/>
</xs:complexType>
```

Child Elements

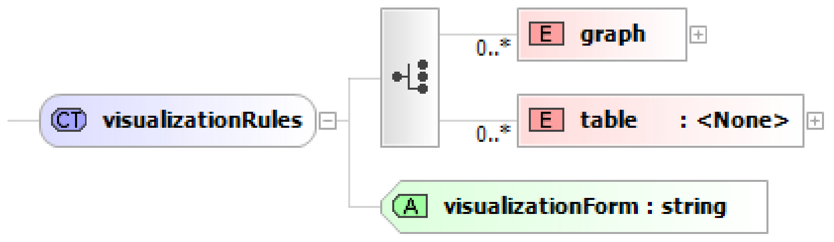
Name	**Type**	**Min Occurs**	**Max Occurs**
condition	condition	0	1
exception	exception	0	1
aggregationAction	aggregationAction	(1)	(1)
expression	tns:expression	0	1

	variable	variable	0	1
	invalidationCriteria	invalidationCriteria	0	1

Child Attributes

	Name	Type	Default Value	Use
	scope	scope		(Optional)
	variableName	variableName		(Optional)

VisualizationRules DataType



XSD Schema Code

```
<xs:complexType name="visualizationRules" >
  <xs:choice>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="graph">
      <xs:complexType>
        <xs:choice>
          <xs:element minOccurs="0" maxOccurs="unbounded" name="dimentionalProperty"
type="xs:string"/>
        </xs:choice>
        <xs:attribute name="graphType" type="xs:string"/>
      </xs:complexType>
    </xs:element>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="table">
      <xs:complexType>
        <xs:attribute name="organizeBy"/>
      </xs:complexType>
    </xs:element>
  </xs:choice>
  <xs:attribute name="visualizationForm">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="Package"/>
        <xs:enumeration value="Class"/>
        <xs:enumeration value="Method "/>
        <xs:enumeration value="Field "/>
        <xs:enumeration value="Value "/>
        <xs:enumeration value="Variable"/>
        <xs:enumeration value="Type"/>
        <xs:enumeration value="EnumerationType "/>
        <xs:enumeration value="StructuredType"/>
        <xs:enumeration value="FormalParameter"/>
        <xs:enumeration value="Routine "/>
        <xs:enumeration value="ExecutableValue"/>
        <xs:enumeration value="CollectionType"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
```

Child Elements

Name	Type	Min Occurs	Max Occurs
graph	graph	0	unbounded
table	table	0	unbounded

Child Attributes

Name	Type	Default Value	Use
visualizationForm	visualizationForm		(Optional)

Figure 10 - Portions of the SMDL schema definition (ver. 1.0)

CHAPTER 6

APPLYING SMDL

This chapter presents sample definitions of software metrics utilizing the SMDL language. It also includes an overview of the prototype implementation.

6.1. SAMPLE DEFINITIONS IN SDML

To show case the application of SMDL, we present a number of metric definitions written in SMDL. Selected metrics include the popular suite of C&K OO design metrics, surveyed in section 2.1.1.

6.1.1. Depth of Inheritance

```
<metric>
    <acronym>DIT</acronym>
    <title>Depth of Inheritance</title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
      <dmmQuery>
        <unit>Class</unit>
            <visitor scope="class" variable=”c”>
                   <variable name="DIT" type="list" scope="class" />
                             <invalidationCriteria affectedElement="Class"
variableName=”parent” condition="parent eq null" scope="Package" />

            <variable name="isVisited" type="long" scope="class" />
            <math:expression>
                   <linkLong name="dit"/>
                           <add datatype="long">
```

```
                                 <long value="1"/>
                          </add>
                  </linkLong>

           </math:expression>
           </visitor>

     </dmmQuery>
   </computation>
 </metric>
```

6.1.2. Weighted Method per Class (WMC)

```
<metric>
    <acronym>WMC</acronym>
    <title> Weighted Method per Class </title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
    <dmmQuery>
            <visitor scope="class">
                  <variable  name="numMethods"  type="long"  scope="class"
/>
            </visitor>
            <visitor scope="method">
                  <math:expression>
                          <linkLong name="numMethods"/>
                                <add datatype="long">
                                        <long value="1"/>
                                </add>
                          </linkLong>
                  </math:expression>
            </visitor>
    </dmmQuery>
    </computation>
  </metric>
```

6.1.3. Response for Class (RFC)

```
<metric>
    <acronym>RFC</acronym>
    <title> Response for Class </title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
    <dmmQuery>
            <visitor scope="class">
                  <variable  name="responseSet"  type="list"  scope="class"
/>
            </visitor>
            <visitor scope="method" variableName="methodSignature">
                  <math:expression>
                          <linkLong name="responseSet"/>
```

```
                                 <add datatype="set">
                                         <lingString                name="”
methodSignature” />
                                 </add>
                          </linkLong>
                  </math:expression>
            </visitor>
    </dmmQuery>
    </computation>
  </metric>
```

6.1.4. Number of Children (NOC)

```
<metric>
    <acronym>NOC</acronym>
    <title> Number of Children </title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
    <dmmQuery>
            <visitor scope="class">
                  <variable  name="responseSet"  type="list"  scope="class"
/>
            </visitor>
            <visitor scope="method" variableName=”m”>
                  <math:expression>
                          <linkList name="responseSet"/>
                                 <add datatype="string">
                                         <lingString name=”m” />
                                 </add>
                          </linkLong>
                  </math:expression>
            </visitor>

            <visitor scope="formalParameter" variableName=”p”>
                  <math:expression>
                          <linkLong name="responseSet"/>
                                 <add datatype="string">
                                         <lingString name=”p” />
                                 </add>
                          </linkLong>
                  </math:expression>
            </visitor>

            <visitor scope="variables" variableName=”v”>
                  <math:expression>
                          <linkLong name="responseSet"/>
                                 <add datatype="string">
                                         <lingString name=”v” />
                                 </add>
                          </linkLong>
                  </math:expression>
            </visitor>
    </dmmQuery>
    </computation>
  </metric>
```

6.1.5. <u>Coupling Between Objects (CBO)</u>

```
<metric>
    <acronym>NOC</acronym>
    <title> Number of Chikdren </title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
    <dmmQuery>
            <visitor scope="class">
                  <variable  name="responseSet"  type="list"  scope="class"
/>
            </visitor>
            <visitor scope="method" variableName=”m”>
                  <math:expression>
                          <linkList name="responseSet"/>
                                <add datatype="string">
                                        <lingString name=”m” />
                                </add>
                          </linkLong>
                  </math:expression>
            </visitor>

            <visitor scope="formalParameter" variableName=”p”>
                  <math:expression>
                          <linkLong name="responseSet"/>
                                <add datatype="string">
                                        <lingString name=”p” />
                                </add>
                          </linkLong>
                  </math:expression>
            </visitor>

            <visitor scope="variables" variableName=”v”>
                  <math:expression>
                          <linkLong name="responseSet"/>
                                <add datatype="string">
                                        <lingString name=”v” />
                                </add>
                          </linkLong>
                  </math:expression>
            </visitor>
    </dmmQuery>
    </computation>
  </metric>
```

6.1.6. <u>Lack of Cohesion in Methods (LCOM)</u>

```
<metric>
    <acronym>LCOM</acronym>
    <title> Lack Of Cohesion Method </title>
    <authority>
      <authors name="C&K" date="1994" />
    </authority>
    <computation>
    <dmmQuery>
            <visitor scope="class">
                  <variable name="lcom" type="long" scope="class" />

                  <variable name="methodInvokes" type="set" scope="class"
/>
                  <math:expression>
                          <intersect>
                          <linkList name=" methodInvokes"/>
                                <add datatype="long">
                                        <linkString name=”m” />
                                </add>
                          </linkList>
                          </intersect>
                  </math:expression>
            </visitor>

            <visitor scope="variables" variableName=”v”>
                  <math:expression>
                          <linkSet name="methodInvokes "/>
                                <add datatype="long">
                                        <linklong>1</linkLong>
                                </add>
                          </linkSet>
                  </math:expression>
            </visitor>
    </dmmQuery>
    </computation>
  </metric>
```

6.2. PROTOTYPE IMPLEMENTATION

As a proof of concept of the proposed in this work, a metrics computation tool called SMDL Metrics Calculator was implemented. The Java-based tool is capable of parsing SMDL files. It uses Java bytecode parsers to read and analyze metrics of java classes based on SMDL definitions. The tool makes use of some of the more advanced SMDL concepts such as progressive and parallel metrics computation.

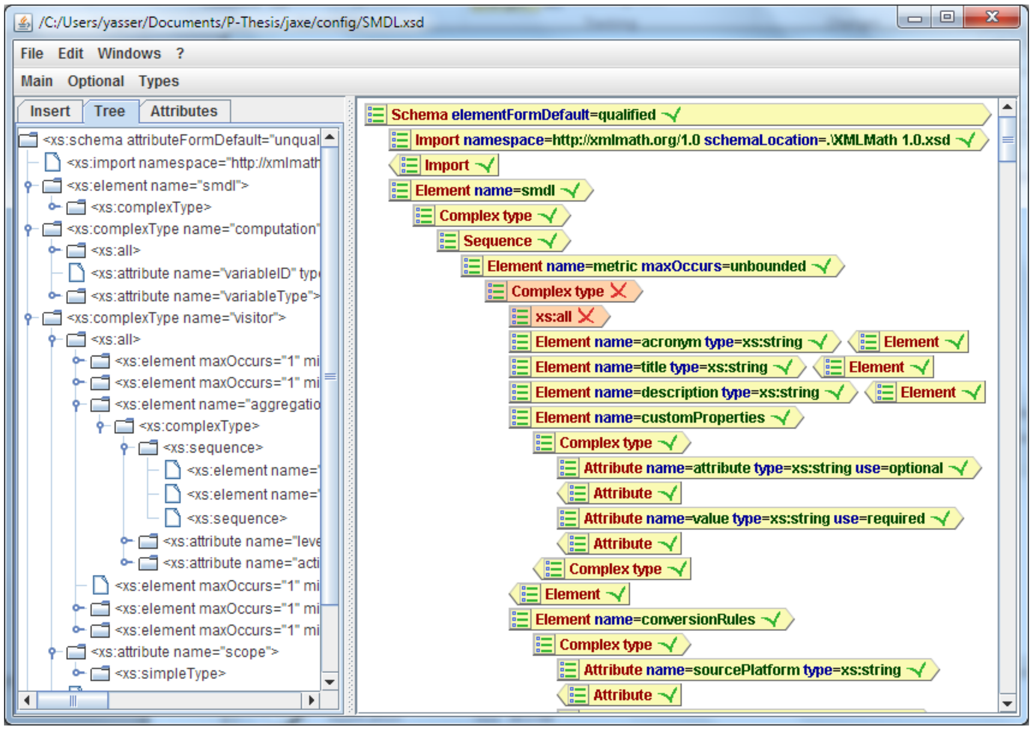
6.3. CODE PARSING

The source code parser used in SMDL Metrics Calculator utilizes BCEL (Byte Code Engineering Library) [44], an open-source Java parser written by Apache group. It was chosen due in part to its powerful capabilities of processing Java bytecode. Additionally, the use of the Visitor design pattern in the framework, for handling progressive parsing, enables a more declarative approach to metric computation definitions that is consistent with the declarative nature of SMDL and satisfies performance requirements of progressive evaluations.

6.3.1. SMDL Parser and Editor

The SMDL Metrics Calculator contains an SMDL definitions' parser and a visual editor. The parser uses XML parsing libraries written by Altova XMLSpy [45] in Java which enables reading SMDL files and generating the necessary data objects that precisely represent the file contents. Objects are then used for the computation of the software metrics according to the SMDL definitions.

The visual editor for SMDL was built using Jaxe, the Java XML editor. Jaxe provices and user interface for editing XML files using a predefined configuration files. A Jaxe configuration file for SMDL was created. Launching Jaxe with the configuration file, the user is prompted to create and edit SMDL documents. Using this editor, the user can insert, edit and update SMDL elements while maintaining compliance with SMDL specifications. The following figures show screenshots of using the editor.

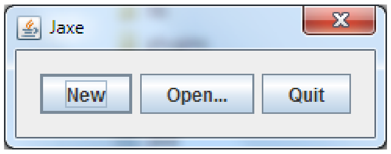


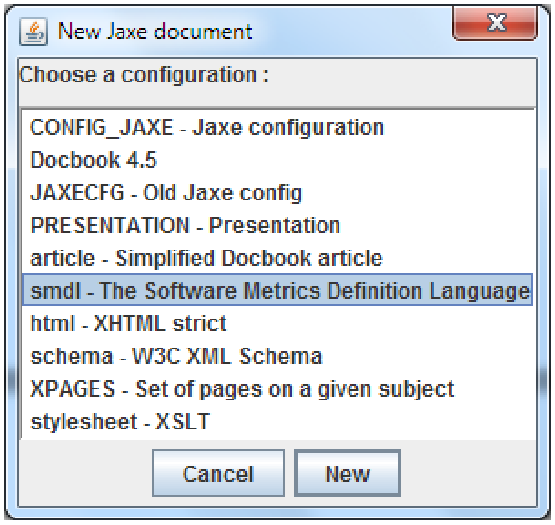
/C:/Users/yasser/Documents/P-Thesis/jaxe/config/SMDL.xsd

```
<xs:schema attributeFormDefault="unqual
  <xs:import namespace="http://xmlmath
  <xs:element name="smdl">
    <xs:complexType>
  <xs:complexType name="computation"
    <xs:all>
      <xs:attribute name="variableID" typ
    <xs:attribute name="variableType">
  <xs:complexType name="visitor">
    <xs:all>
      <xs:element maxOccurs="1" mi
      <xs:element maxOccurs="1" mi
      <xs:element name="aggregatio
        <xs:complexType>
          <xs:sequence>
            <xs:element name="
            <xs:element name="
            <xs:sequence>
          <xs:attribute name="leve
          <xs:attribute name="acti
      <xs:element maxOccurs="1" mi
      <xs:element maxOccurs="1" mi
      <xs:element maxOccurs="1" mi
    <xs:attribute name="scope">
      <xs:simpleType>
```

```
Schema elementFormDefault=qualified
  Import namespace=http://xmlmath.org/1.0 schemaLocation=.\XMLMath 1.0.xsd
  Import
  Element name=smdl
    Complex type
      Sequence
        Element name=metric maxOccurs=unbounded
          Complex type
            xs:all
              Element name=acronym type=xs:string   Element
              Element name=title type=xs:string   Element
              Element name=description type=xs:string   Element
              Element name=customProperties
                Complex type
                  Attribute name=attribute type=xs:string use=optional
                  Attribute
                  Attribute name=value type=xs:string use=required
                  Attribute
                Complex type
              Element
              Element name=conversionRules
                Complex type
                  Attribute name=sourcePlatform type=xs:string
                  Attribute
```

Figure 11 - Screenshot of Jaxe Editing the SMDL Schema

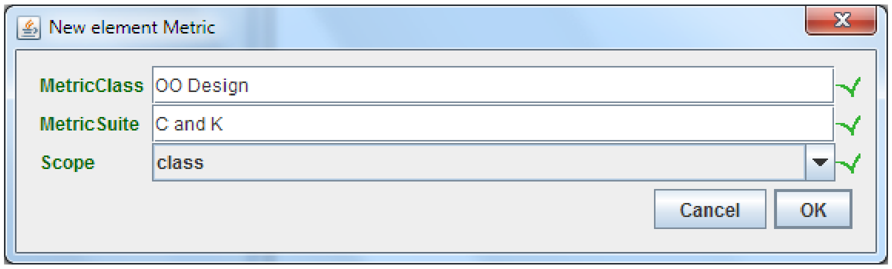




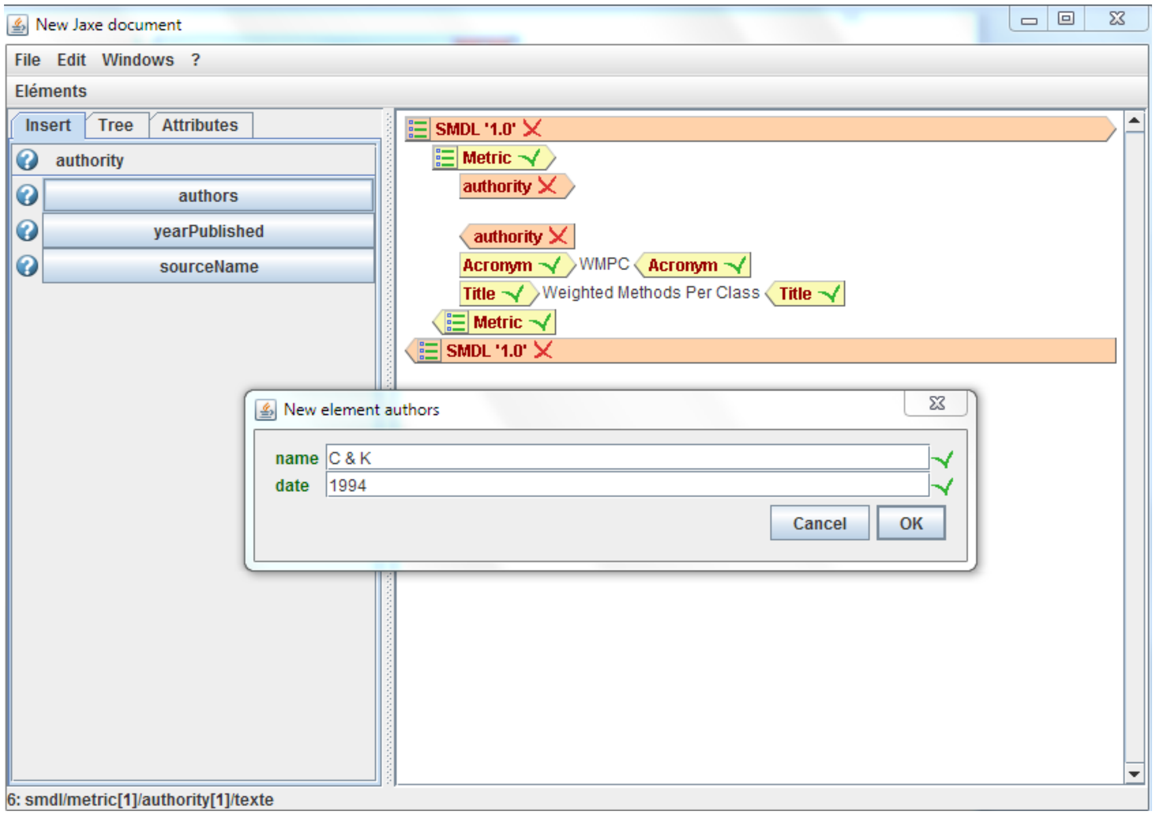
New Jaxe document

Choose a configuration :

CONFIG_JAXE - Jaxe configuration
Docbook 4.5
JAXECFG - Old Jaxe config
PRESENTATION - Presentation
article - Simplified Docbook article
smdl - The Software Metrics Definition Language
html - XHTML strict
schema - W3C XML Schema
XPAGES - Set of pages on a given subject
stylesheet - XSLT



MetricClass OO Design
MetricSuite C and K
Scope class

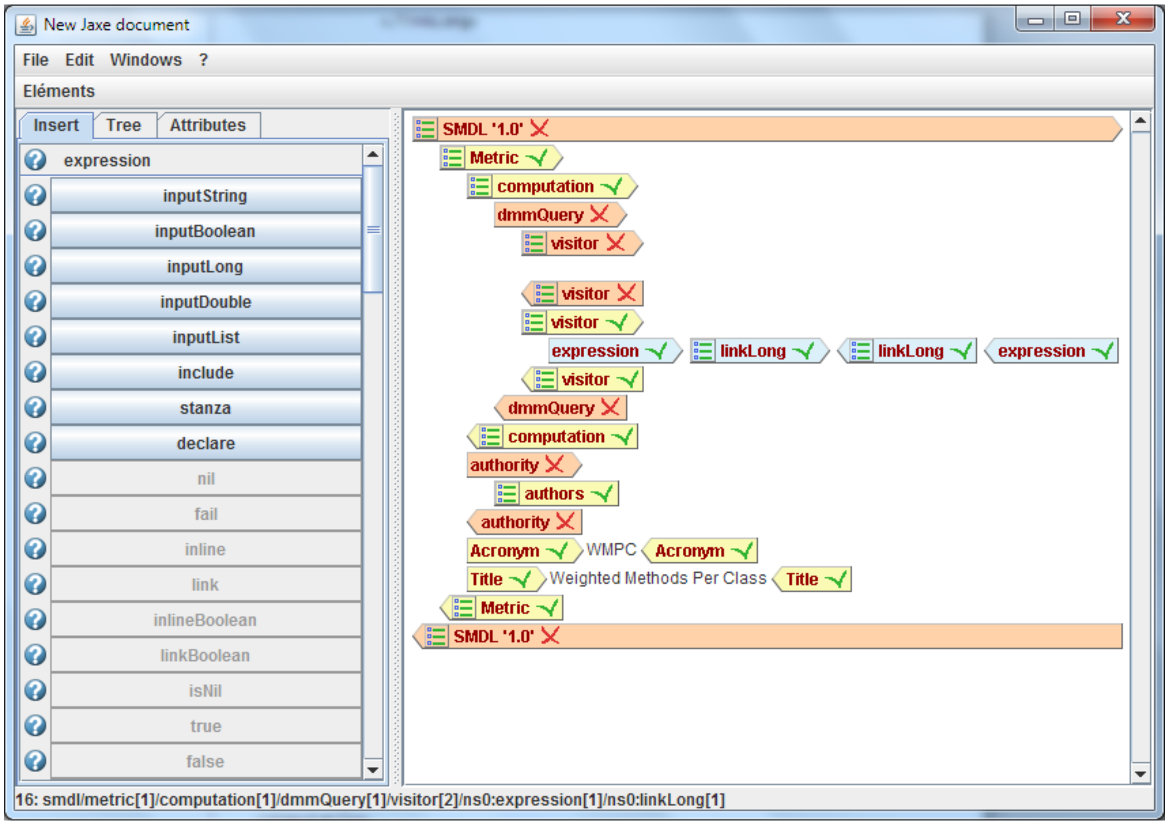


WMPC

Weighted Methods Per Class

name C & K
date 1994

6: smdl/metric[1]/authority[1]/texte



16: smdl/metric[1]/computation[1]/dmmQuery[1]/visitor[2]/ns0:expression[1]/ns0:linkLong[1]

Figure 12 Screenshots of Creating an SMDL Document

6.3.2. <u>SMDL Calculator</u>

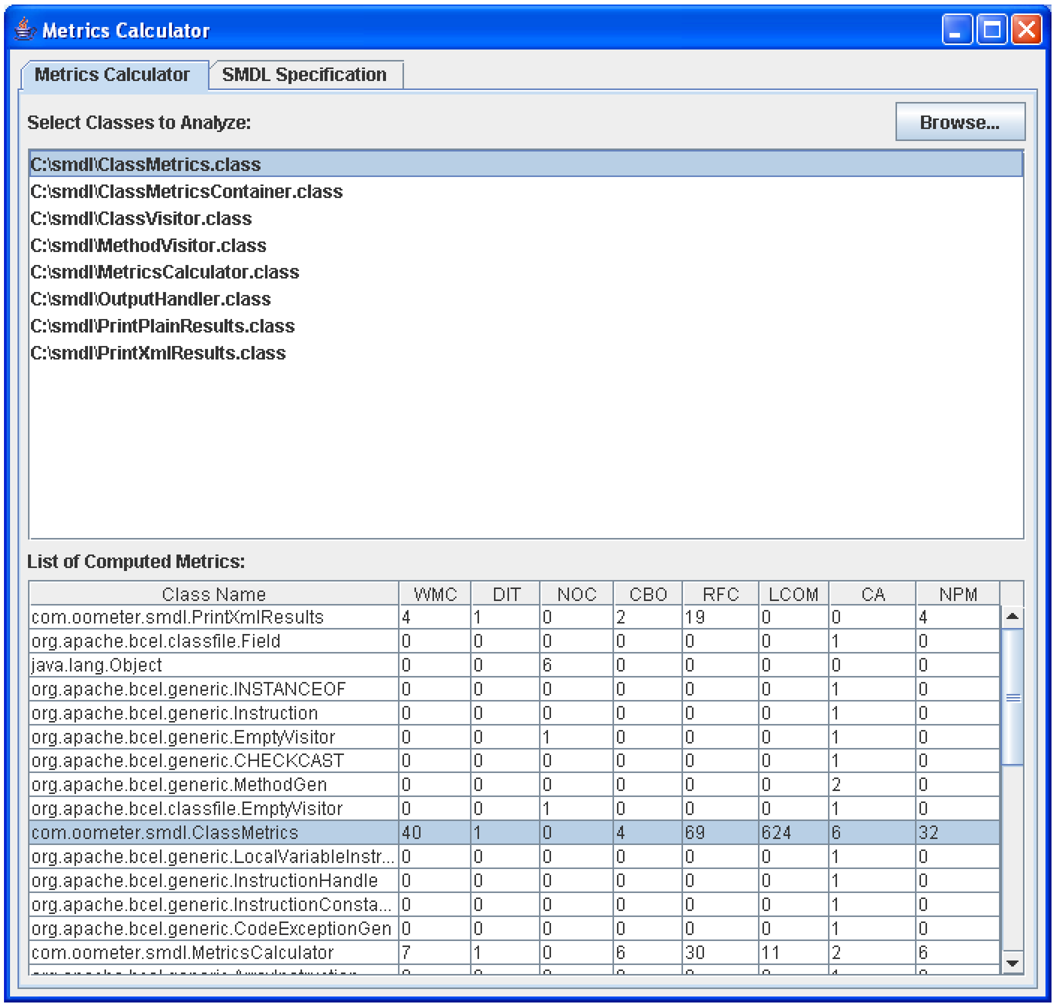
As an implementation example, the C&K metrics suite was selected for the prototype. In particular, parsers and calculators for the following metrics were implemented in our prototype:

Table 5: Selected Metrics for Prototype Implementation

Metric	**Description**
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children

CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
CA	Afferent couplings
NPM	Number of public methods

Figure 10 shows a snapshot of the Metrics Calculator application which displays metrics results after applying the tool on selected Java classes:



Select Classes to Analyze:

```
C:\smdl\ClassMetrics.class
C:\smdl\ClassMetricsContainer.class
C:\smdl\ClassVisitor.class
C:\smdl\MethodVisitor.class
C:\smdl\MetricsCalculator.class
C:\smdl\OutputHandler.class
C:\smdl\PrintPlainResults.class
C:\smdl\PrintXmlResults.class
```

List of Computed Metrics:

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
com.oometer.smdl.PrintXmlResults	4	1	0	2	19	0	0	4
org.apache.bcel.classfile.Field	0	0	0	0	0	0	1	0
java.lang.Object	0	0	6	0	0	0	0	0
org.apache.bcel.generic.INSTANCEOF	0	0	0	0	0	0	1	0
org.apache.bcel.generic.Instruction	0	0	0	0	0	0	1	0
org.apache.bcel.generic.EmptyVisitor	0	0	1	0	0	0	1	0
org.apache.bcel.generic.CHECKCAST	0	0	0	0	0	0	1	0
org.apache.bcel.generic.MethodGen	0	0	0	0	0	0	2	0
org.apache.bcel.classfile.EmptyVisitor	0	0	1	0	0	0	1	0
com.oometer.smdl.ClassMetrics	40	1	0	4	69	624	6	32
org.apache.bcel.generic.LocalVariableInstr...	0	0	0	0	0	0	1	0
org.apache.bcel.generic.InstructionHandle	0	0	0	0	0	0	1	0
org.apache.bcel.generic.InstructionConsta...	0	0	0	0	0	0	1	0
org.apache.bcel.generic.CodeExceptionGen	0	0	0	0	0	0	1	0
com.oometer.smdl.MetricsCalculator	7	1	0	6	30	11	2	6

Figure 13 Screenshot of SMDL Metrics Calculator

The tool is designed to be simple to use. Upon selection of the Java classes of interest, which are *.class* files, the tool computes and presents the defined metrics. Results are presented in a tabular format for each of the selected classes. Partial results are also included for classes associated with the selected classes, such as aggregations and inheritance relationships.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. CONCLUSION

In this research, we presented a novel approach to representing software metrics definitions that enables reusability, extensibility and accuracy of metrics definitions. The proposed Software Metrics Definition Language, SMDL, is an XML based meta-definition language that can be applied to represent metrics definitions. We have shown that using SMDL can simplify metrics definitions by enabling reusability of previous definitions and definition elements. Another advantage to the end users of the language is metrics customization capabilities. We have also implemented a prototype, a proof-of-concept, as a start to motivate adoption of the new approach and demonstrate some of its capabilities.

7.2. CONTRIBUTION

Following is a list of contributions achieved in this research.

- Surveying and comparing the different methods of formalizing software metrics definitions and proposing a more comprehensive, flexible alternative.

- Proposing a framework for software metrics measurement and data collection that abstracts the main component of a complete measurement solution.
- Providing a prototype implementation to demonstrate the proposed approach as a proof of concept.
- Discussion of performance considerations and challenges in metrics evaluation schemes.
- Addressing shortcomings of the alternative approaches to software metrics definitions. The proposed language tackles the important challenges of metrics definitions and can be beneficial to the software engineering research community.

7.3. FUTURE WORKS

Formalizing metrics definitions is only a part of the measurement process. In this work, we focused on the representation of the metrics definitions against a standard meta-model. The bigger picture is more complicated and there is room for improvement in areas such as:

- **Measurement Data**: completing the framework by introducing a language for representing software metrics measurement data. There is already some research in this area which can be integrated into this framework.
- **Metrics Data Analysis**: Software metrics data interpretation and classification mechanisms which can be used for quality measures and indicators.

- **Performance Considerations**: Introducing optimizations of metrics computations though caching or other techniques depending on some heuristics. This can benefit from database query optimization techniques. Results can help build more practical tools that are seamlessly integrated into the development effort.
- **Advanced Computation**: Enabling more complex forms of metrics computations such as comparative metrics (e.g. similarity, stability …).
- **Design Metrics**: Expanding the meta-model to include all design artifacts such as state diagrams, sequence diagrams, and use cases, in order to allow more general forms of design metrics.
- **Concurrency of and Computation Pipeline**: Providing detailed analysis of performance overhead for computing software metrics and discussing potential way of parallelizing the computation process.
- **Visualization**: Enhancement to the visualization description of the metrics definition to support common metrics visualization hierarchies.

APPENDICES

APPENDIX A. THE SOFTWARE METRICS DEFINITION LANGUAGE SCHEMA

The following is source code for the current version of the SMDL Schema.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema                             xmlns:ns0="http://xmlmath.org/1.0"
attributeFormDefault="unqualified"         elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:import              schemaLocation=".\XMLMath              1.0.xsd"
namespace="http://xmlmath.org/1.0" />
  <xs:element name="smdl">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="metric">
          <xs:complexType>
            <xs:all>
              <xs:element name="acronym" type="xs:string" />
              <xs:element name="title" type="xs:string" />
              <xs:element        minOccurs="0"        name="description"
type="xs:string" />
              <xs:element          minOccurs="0"          maxOccurs="1"
name="customProperties">
                <xs:complexType>
                  <xs:attribute     name="attribute"     type="xs:string"
use="optional" />
                  <xs:attribute       name="value"       type="xs:string"
use="required" />
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="0" name="conversionRules">
                <xs:complexType>
                  <xs:attribute name="sourcePlatform" type="xs:string" />
                  <xs:attribute name="targetPlatform" type="xs:string" />
                  <xs:attribute name="forumula" />
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="0" name="authority">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element maxOccurs="unbounded" name="authors">
                      <xs:complexType>
                        <xs:attribute name="name" type="xs:string" />
                        <xs:attribute name="date" type="xs:string" />
                      </xs:complexType>
                    </xs:element>
                    <xs:element name="yearPublished" type="xs:date" />
                    <xs:element name="sourceName" type="xs:string" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="0" name="computation"
type="computation" />
              <xs:element minOccurs="0" maxOccurs="1"
name="reusedMetrics">
                <xs:complexType>
                  <xs:attribute name="metricName" />
                  <xs:attribute name="variableID" />
                </xs:complexType>
              </xs:element>
              <xs:element minOccurs="0" name="visualization"
type="visualizationRules" />
            </xs:all>
            <xs:attribute name="metricClass" type="xs:string" />
            <xs:attribute name="metricSuite" type="xs:string" />
            <xs:attribute name="scope">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="package" />
                  <xs:enumeration value="class" />
                  <xs:enumeration value="method" />
                  <xs:enumeration value="variable" />
                </xs:restriction>
              </xs:simpleType>
            </xs:attribute>
          </xs:complexType>
        </xs:element>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="intermediateVariable" type="computation" />
        </xs:choice>
      </xs:sequence>
      <xs:attribute name="version" type="xs:decimal" use="required" />
    </xs:complexType>
  </xs:element>
  <xs:complexType name="computation">
    <xs:all>
      <xs:element minOccurs="0" name="dmmQuery">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="description" type="xs:string" />
            <xs:element name="unit" type="xs:string" />
            <xs:element maxOccurs="unbounded" name="visitor"
type="visitor" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" name="oclQuery">
        <xs:complexType>
          <xs:sequence>
```

```
            <xs:element        minOccurs="0"        maxOccurs="unbounded"
name="oclQueryVariable">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="query" type="xs:string" />
                  <xs:element name="variableID" type="xs:string" />
                </xs:sequence>
              </xs:complexType>
            </xs:element>
            <xs:element name="mainOclQuery" type="xs:string" />
            <xs:element name="scope">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                  <xs:enumeration value="class" />
                  <xs:enumeration value="package" />
                  <xs:enumeration value="method" />
                  <xs:enumeration value="field" />
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="variableID" type="xs:string" />
    <xs:attribute name="variableType">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="double" />
          <xs:enumeration value="long" />
          <xs:enumeration value="list" />
          <xs:enumeration value="set" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
  <xs:complexType name="visitor">
    <xs:all>
      <xs:element minOccurs="0" maxOccurs="1" name="condition">
        <xs:complexType>
          <xs:attribute name="expression" type="xs:string" use="required"
/>
          <xs:attribute name="action" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="skip" />
                <xs:enumeration value="stop" />
                <xs:enumeration value="continue" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" maxOccurs="1" name="exception">
        <xs:complexType>
```

```
          <xs:attribute name="expression" type="xs:string" use="required"
/>
          <xs:attribute        name="returnValue"        type="xs:string"
use="required" />
        </xs:complexType>
      </xs:element>
      <xs:element name="aggregationAction">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="type" type="xs:string" />
            <xs:element name="operator" type="xs:string" />
            <xs:sequence />
          </xs:sequence>
          <xs:attribute name="level">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="package-leve" />
                <xs:enumeration value="class-level" />
                <xs:enumeration value="method-level" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="action">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="sum" />
                <xs:enumeration value="multiply" />
                <xs:enumeration value="average" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" maxOccurs="1" ref="ns0:expression" />
      <xs:element minOccurs="0" maxOccurs="1" name="variable">
        <xs:complexType>
          <xs:attribute name="name" type="xs:string" />
          <xs:attribute name="type">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="long" />
                <xs:enumeration value="double" />
                <xs:enumeration value="string" />
                <xs:enumeration value="list" />
                <xs:enumeration value="set" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute name="scope">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="package" />
                <xs:enumeration value="class" />
                <xs:enumeration value="method" />
                <xs:enumeration value="attribute" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
      <xs:element               minOccurs="0"              maxOccurs="1"
name="invalidationCriteria">
        <xs:complexType>
          <xs:attribute name="affectedElement" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="Method" />
                <xs:enumeration value="Class" />
                <xs:enumeration value="Package" />
                <xs:enumeration value="Field" />
                <xs:enumeration value="Attribute" />
                <xs:enumeration value="Parameter" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
          <xs:attribute       name="variableName"       type="xs:string"
use="required" />
          <xs:attribute  name="condition"  type="xs:string"  use="required"
/>
          <xs:attribute name="scope" use="required">
            <xs:simpleType>
              <xs:restriction base="xs:string">
                <xs:enumeration value="Method" />
                <xs:enumeration value="Class" />
                <xs:enumeration value="Package" />
                <xs:enumeration value="Field" />
                <xs:enumeration value="Attribute" />
                <xs:enumeration value="Parameter" />
              </xs:restriction>
            </xs:simpleType>
          </xs:attribute>
        </xs:complexType>
      </xs:element>
    </xs:all>
    <xs:attribute name="scope">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Package" />
          <xs:enumeration value="Class" />
          <xs:enumeration value="Method " />
          <xs:enumeration value="Field   " />
          <xs:enumeration value="Value   " />
          <xs:enumeration value="Variable" />
          <xs:enumeration value="Type" />
          <xs:enumeration value="EnumerationType " />
          <xs:enumeration value="StructuredType" />
          <xs:enumeration value="FormalParameter" />
          <xs:enumeration value="Routine " />
          <xs:enumeration value="ExecutableValue" />
          <xs:enumeration value="CollectionType" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="variableName" type="xs:string" />
  </xs:complexType>
  <xs:complexType name="visualizationRules">
    <xs:choice>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="graph">
        <xs:complexType>
          <xs:choice>
            <xs:element        minOccurs="0"        maxOccurs="unbounded"
name="dimentionalProperty" type="xs:string" />
          </xs:choice>
          <xs:attribute name="graphType" type="xs:string" />
        </xs:complexType>
      </xs:element>
      <xs:element minOccurs="0" maxOccurs="unbounded" name="table">
        <xs:complexType>
          <xs:attribute name="organizeBy" />
        </xs:complexType>
      </xs:element>
    </xs:choice>
    <xs:attribute name="visualizationForm">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:enumeration value="Package" />
          <xs:enumeration value="Class" />
          <xs:enumeration value="Method " />
          <xs:enumeration value="Field " />
          <xs:enumeration value="Value " />
          <xs:enumeration value="Variable" />
          <xs:enumeration value="Type" />
          <xs:enumeration value="EnumerationType " />
          <xs:enumeration value="StructuredType" />
          <xs:enumeration value="FormalParameter" />
          <xs:enumeration value="Routine " />
          <xs:enumeration value="ExecutableValue" />
          <xs:enumeration value="CollectionType" />
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:schema>
```

REFERENCES

[1] N. E. Fenton, *Software Metrics: A Rigorous Approach*. London: Chapman & Hall, Ltd., 1991.

[2] D. Troy and S. Zweben, "Measuring the Quality of Structured Design," *The Journal of Systems and Software,* vol. 2, pp. 113-120, 1981.

[3] S. a. C. K. Chidamber, "Towards a Metrics Suite for Object-Oriented Design," in *Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, 1991, pp. 97-211.

[4] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering* vol. 20, pp. 476-493, 1994.

[5] V. Basili, L. Briand, and W. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Transactions on Software Engineering,* vol. 22, 1996.

[6] V. Laing and C. Coleman, "Principal Components of Orthogonal Object-Oriented Metrics," NASA 2001.

[7] M. Alshayeb and W. Li, "An empirical validation of object-oriented metrics in two different iterative software processes," *Transactions on Software Engineering,* vol. 29, 2003.

[8] W. Li, "Another metric suite for object-oriented programming," *The Journal of Systems and Software*, pp. 155-162, 1998.

[9] F. B. e. Abreu, "Design Quality Metrics for Object-Oriented Software Systems," *ERCIM News*, 1995

[10] A. L. Baroni and F. B. Abreu, "An OCL-Based Formalization of the MOOSE Metric Suite," in *Proceedings of the 7th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* Darmstadt, Germany, 2003.

[11] E. M. Kim, S. Kusumoto, T. Kikuno, and O. B. Chang, "Heuristics for Computing Attribute Values of C++ Program Complexity Metrics," in *The 20th Conference on Computer Software and Applications* 1996, p. 104.

[12] "Software Quality Metrics for Object Oriented System Environments," National Aeronautics and Space Administration, June 1995 1995.

[13] M. Sarker, "An overview of Object Oriented Design Metrics," in *Department of Computer Science*. vol. MS Sweden: Umeå University, 2005.

[14] C. Jones, "Strengths and Weaknesses Of Software Metrics," 2006.

[15] N. I. Churcher and M. J. Shepperd, "Comments on 'A metrics suite for object-oriented design'," *IEEE Transactions on Software Engineering*, vol. 21, 1995.

[16] D. Strein, R. Lincke, J. Lundberg, and W. Löwe, "An Extensible Metamodel for Program Analysis," *IEEE Transactions on Software Engineering* vol. 33, 2007.

[17] B. A. Kitchenham, R. T. Hughes, and S. G. Linkman, "Modeling Software Measurement Data," *IEEE Transactions on Software Engineering* vol. 27, pp. 788-804, 2001.

[18] M. Scotto, A. Sillitti, G. Succi, and T. Vernazza, "A Relational Approach to Software Metrics," in *Proceedings of the 2004 ACM symposium on Applied computing* Nicosia, Cyprus, 2004, pp. 1536-1540.

[19] D. Beyer, C. Lewerentz, and A. Noack, "Efficient relational calculation for software analysis," *IEEE Transactions on Software Engineering*, vol. 31, 2005.

[20] M. Lanza and S. Ducasse, "Beyond Language Independent Object-Oriented Metrics: Model Independent," in *Proceedings of the 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002, pp. 77-84.

[21] R. Reißing, "Towards a Model for Object-Oriented Design Measurement " in *5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2001.

[22] M. M. El-Wakil, A. El-Bastawisi, M. B. Riad, and A. A. Fahmy, "A novel approach to formalize and collect Object-Oriented Design-Metrics," in *9th International Conference on Empirical Assessment in Software Engineering (EASE 2005)*, 2005.

[23] F. B. e. Abreu, L. Ochoa, and M. Goulão, "The GOODLY Design Language for MOOD Metrics Collection," INESC 1997.

[24] J. A. McQuillan and J. F. Power, "Some observations on the application of software metrics to UML models," in *Model Size Metrics Workshop of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems* Genoa, Italy, 2006.

[25] J. A. McQuillan and J. F. Power, "Towards re-usable metric definitions at the meta-level," in *PhD Workshop of the 20th European Conference on Object-Oriented Programming* Nantes, France, 2006.

[26] "Eclipse Homepage.," in http://www.eclipse.org/: The Eclipse Foundation, 2007.

[27] "NetBeans Homepage.," in http://www.netbeans.org/: Sun Microsystems, 2007.

[28] SDMetrics, "SDMetrics User Manual," 2006.

[29] T. Lethbridge, S. Tichelaar, and E. Plödereder, "The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering," *Electronic Notes in Theoretical Computer Science*, vol. 94, pp. 7-18, 2004.

[30] L. O. Ejiogu, "TM: a systematic methodology of software metrics," *ACM SIGPLAN Notices*, vol. 26, pp. 124-132, 1991.

[31] B. I. Cogan and R. B. Hunter, "Language-based Approaches to Software Measurement," in *Proceedings of the 3rd International Software Metrics Symposium, 1996.*, Berlin, Germany, 1996.

[32] A. L. Baroni and F. B. e. Abreu, "Formalizing Object-Oriented Design Metrics upon the UML Meta-Model," in *16th Brazilian Symposium on Software Engineering*, Gramado, Brazil, 2002.

[33] M. Goulão and F. B. e. Abreu, "Formal Definition of Metrics Upon the CORBA Component Model," in *Proceedings of the First International Conference on Software Architectures*, Erfurt, Germany, 2005.

[34] N. Debnath, D. Riesco, G. Montejano, R. Uzal, L. Baigorria, A. Dasso, and A. Funes, "A technique based on the OMG metamodel and OCL for the definition of object-oriented metrics applied to UML models," in *The 3rd ACS/IEEE International Conference on Computer Systems and Applications*, 2005.

[35] J. A. McQuillan and J. F. Power, "A definition of the Chidamber and Kemerer metrics suite for the Unified Modeling Language," Department of Computer Science, NUI Maynooth, Co. Kildare, Ireland, Technical Report October 2006 2006.

[36] J. A. McQuillan and J. F. Power, "Experiences of using the Dagstuhl Middle Metamodel for defining software metrics," in *Proceedings of the 4th International*

Conference on Principles and Practices of Programming in Java, Mannheim, Germany, 2006, pp. 194-198.

[37] J. A. McQuillan and J. F. Power, "On the application of software metrics to UML models," in *Satellite Events at the MoDELS 2006 Conference*, 2006.

[38] R. Lincke and W. Löwe, "Foundations for Defining Software Metrics," in *3rd International Workshop on Metamodels, Schemas, Grammars, and Ontologies for Reverse Engineering (ATEM)*, 2006.

[39] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0 (Fourth Edition) - Origin and Goals," World Wide Web Consortium, 2006.

[40] M. Auer, "Measuring the Whole Software Process: A Simple Metric Data Exchange Format and Protocol," in *Proceedings of 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* Mlaga, 2002.

[41] W. Harrison, "A flexible method for maintaining software metrics data: a universal," *The Journal of Systems & Software*, vol. 72, pp. 225-234, 2004.

[42] D. Margerison, "Outline proposal for adopting a generic standard for storing metrics information," 2004.

[43] E. v. Zijst, "XMLMath - XML-Based Mathematical Expression Evaluator.," in *http://www.xmlmath.org/*, 2006.

[44] "BCEL Homepage.," in *http://jakarta.apache.org/bcel/*, 2007.

[45] "Altova XMLSpy Homepage.," in *http://www.altova.com/*, 2007.

VITA

Personal Data:

Name: Yasser Elsayed Mohamed Shaaban

Nationality: Egyptian.

Contact:

Current Address: 1301 1st Ave, Seattle 98101, WA, United States
Phone: (+1) 4252337587

Permanent address: 32 Neaam St, Muharram Bek, Alexandria, Egypt

Email: yasser.shaaban@hotmail.com

Yasser Elsayed Mohamed Shaaban was born in 1983 in Alexandria, Egypt. He received his Bachelor of Science in Software Engineering, with first honors, from King Fahd University of Petroleum and Minerals (KFUPM) in June 2006. He then joined the Information and Computer Science Department at KFUPM as a Research Assistant while pursuing his Master's degree in Computer Science. During the course of his graduate studies, he took advanced computer science and software engineering courses such as Principles of Software Engineering, Advanced Artificial Intelligence, Pattern Recognition, Computer Security, Advanced Computer Algorithms, and Advanced High Performance Computing. During his course of study, he participated in software projects inside and outside the university in various areas from enterprise applications to data mining and parallel computing. His research interests are in software engineering, parallel programming and streaming algorithms.